Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích
Ekonomická fakulta
Katedra aplikované informatiky a matematiky

Bakalářská práce

Vývoj mobilní aplikace pro organizování času pro iOS a její publikování v App Store

Vypracoval: Tom Schuh
Vedoucí práce: Mgr. Radim Remeš

České Budějovice 2019

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH
Ekonomická fakulta
Akademický rok: **2017/2018**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Tom SCHUH**
Osobní číslo:	**E16549**
Studijní program:	**B6209 Systémové inženýrství a informatika**
Studijní obor:	**Ekonomická informatika**
Název tématu:	**Vývoj mobilní aplikace pro organizování času pro iOS a její publikování v App Store**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování:

Cílem práce je vytvořit mobilní aplikaci pro organizaci času a provést její distribuci pomocí elektronického obchodu s aplikacemi. Aplikace bude umožňovat plánovat činnosti, schůzky a další aktivity. Bude schopna ve zvolený čas upozornit na plánovanou aktivitu (notifikace v zařízení, emailem). Aplikace bude schopna propojení s dalším existujícím kalendářem (např. Google nebo Outlook).

Metodický postup:

1. Studium odborné literatury.
2. Návrh, popis vývoje a implementace aplikace.
3. Zhodnocení použitelnosti aplikace pro nasazení v reálném prostředí, vypracování doporučení a závěrů.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **40 - 50 stran**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury:

1. **Borycki, D. (2018).** *Beginning Xamarin Development for the Mac: Create iOS, watchOS, and Apple tvOS apps with Xamarin.iOS and Visual Studio for Mac.* **New York, NY (USA): Apress.**
2. **Michaelis, M. (2018).** *Essential C# 7.0* **(6th ed.). Boston, Massachusetts: Addison-Wesley.**
3. **Nutting, J., Olsson, F., Mark, D. & LaMarche, J. (2014).** *Beginning iOS 7 Development: Exploring the iOS SDK.* **New York, NY (USA): Apress.**
4. **Peppers, J. (2015).** *Xamarin Cross-platform Application Development* **(2nd ed.). Birmingham, UK: Packt Publishing.**
5. **Price, M. J. (2017).** *C# 7.1 and .NET Core 2.0: Modern Cross-Platform Development* **(3rd ed.). Birmingham, UK: Packt Publishing.**
6. **Tavlikos, D. (2014).** *iOS Development with Xamarin Cookbook.* **Birmingham, UK: Packt Publishing.**

Vedoucí bakalářské práce: **Mgr. Radim Remeš**
Katedra aplikované matematiky a informatiky

Datum zadání bakalářské práce: **19. ledna 2018**

Termín odevzdání bakalářské práce: **12. dubna 2019**

doc. Ing. Ladislav Rolínek, Ph.D.
děkan

JIHOČESKÁ UNIVERZITA
V ČESKÝCH BUDĚJOVICÍCH
EKONOMICKÁ FAKULTA
Studentská 13 (26)
370 05 České Budějovice

doc. RNDr. Jana Klicnarová, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 28. března 2018

Prohlášení

Prohlašuji, že svou bakalářskou práci „Vývoj mobilní aplikace pro organizování času pro iOS a její publikování v App Store“ jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury. Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to – v nezkrácené podobě/v úpravě vzniklé vypuštěním vyznačených částí archivovaných Ekonomickou fakultou – elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

……..	……..
Datum	Podpis

Poděkování

Tímto bych rád poděkoval vedoucímu mé bakalářské práce Mgr. Radimu Remešovi, za věcné připomínky, cenné rady při vypracovávání mé bakalářské práce. A dále bych chtěl poděkovat rodině za podporu kterou mi poskytovali během studia

Obsah

1 Úvod a cíl bakalářské práce

Vývoj mobilní aplikace na operační systém iOS byl do nedávné doby možný pouze pomocí jazyka, který byl vyvinut přímo společností, která distribuuje zařízení s tímto systémem. Nyní je ale už možné vyvíjet tyto aplikace pomocí jazyku C#, a to díky vývojové platformě Xamarin. Tato platforma ovšem nepodporuje vývoj pouze pro systém iOS ale i pro ostatní nejpoužívanější operační systémy, jedná se o Android a Windows. To znamená že se napíše univerzální kód a potom se upraví pouze drobnými změnami, aby mohl být použitý na všech zařízeních. Tato cesta vývoje aplikace není ještě úplně běžná ale pokud vývoj platformy Xamarin nebo obdobných vývojových platforem půjde dopředu, ušetřilo by se tím spousty času při programování mobilních aplikací.

Cílem bakalářské práce je vytvořit aplikaci která funguje na principu To-Do listu. Lze přidávat jednotlivé úkoly. Těmto úkolům lze nastavit krátký popisek a kategorie. Dále lze nastavit čas úkolu a upozornění na jednotlivé úkoly v telefonu a pomocí emailu. Aplikace by měla být schopná propojení s kalendářem. Nakonec by aplikace měla být uvedena v oficiálním obchodu AppStore.

V teoretické části bakalářské práce je popsán životní cyklus aplikace a některé mobilní platformy na které lze aplikaci vyvíjet. Větší zaměření je na aplikace pro iOS, architekturu vrstev iOS aplikací a možnosti vývojových platforem. Dále jsou popsány architektonické modely, které se používají pro snadnější vývoj a správu aplikace. Jelikož se tato práce zabývá vývojem na platformě Xamarin, v dalších kapitolách je popsána tato vývojová platforma, způsoby vývoje uživatelského rozhraní na této platformě a popis sestavení aplikace.

Praktická část bakalářské práce obsahuje popis vývoje aplikace pro iOS na vývojové platformě Xamarin, od vytvoření uživatelského rozhraní přes propojení uživatelského rozhraní s logikou aplikace. Dále popis naprogramování samotné logiky, ukládání dat, přidání upozornění a naprogramování propojení s kalendářem. Platforma Xamarin pořád prochází neustálým vývojem, a proto všechny postupy použité v této práci se vztahují k verzím z roku 2019.

2 Návrh a vývoj aplikace

Životní cyklus aplikace lze shrnout do pěti etap. Tyto části jsou konceptualizace, definice, návrh, vývoj a publikování.

- Konceptualizace, v této etapě dochází k vytvoření nápadu pro aplikaci s ohledem na vyřešení potřeb a problému uživatele. Tento nápad by měl vycházet z předběžného průzkumu a zpětné vazby.
- Definice je proces, kdy jsou definováni koncoví uživatelé, zde se také určují základní funkčnost aplikace. Taktéž se určí rozsah aplikace a složitost návrhu.
- Návrh, v této etapě se určí koncept aplikace a vytváří se vizuální návrh aplikace. Také v této etapě dochází k vytváření prvních prototypů pro testování pro koncové uživatele.
- Vývoj je etapa, kdy dochází k vytváření samotné aplikace, po vytvoření první verze zde dochází k opravení funkčních chyb.

  Publikování je poslední etapou. V této fázi je konečně zpřístupněna aplikace konečným uživatelům. Poté jsou u aplikace sledovány statistky a uživatelská zpětná vazba. Data získaná z těchto zdrojů slouží k vylepšování a opravování aplikace pomocí aktualizací a budoucích verzí (Cuello & Vittone, 2013).

2.1 Platformy

Před návrhem aplikace je důležité si promyslet na jakou platformu, popřípadě více platforem, aplikaci vytvořit. V současné době největší počet zařízení používá operační systém Android, na druhém místě je operační systém iOS a další je Windows společně s dalšími ostatními systémy. Ovšem první dva systémy, s převahou převládají na trhu.

Vytváření aplikace pro nejrozšířenější systém Android, znamená potencionálně větší rozsah uživatelů aplikace. S tím že systém Android je nejrozšířenější přichází i problémy, a to mnoho různých rozlišení obrazovky a spoustu verzí tohoto operačního systému. To v podstatě znamená že návrh pro tento systém je dražší. Na druhé straně, vývoj pro systém iOS znamená, zaměřit se na užší exklusivnější trh. Ačkoliv zařízení s iOS mají menší počet uživatelů, výhodou vývoje pro tato zařízení je že mají menší počet rozlišení obrazovky a méně verzí operačního systému. Poslední zařízení s operačním systémem Windows bylo vydáno v roce 2014, proto vytváření aplikace pro tato zařízení již skoro nemá význam (Cuello & Vittone, 2013).

2.1.1 Android

Aplikace pro Android jsou naprogramovány v jazyce Java pomocí vlastního systému knihoven Android. Vývojáři se znalostí standartního jazyka Java, by neměli mít větší problémy s vývojem aplikace pro Android.

Programování aplikací pro tento operační systém lze provést na zařízeních s MacOS, Windows nebo Linux. Existuje několik vývojových prostředí pro tento operační systém a také existuje jedno vývojové prostředí s názvem Android studio, které bylo vydáno přímo od vlastníka tohoto systému. Toto prostředí je poskytováno zadarmo a umožňuje vývojářům používat několik simulátorů, taktéž vývojáři mohou provést testování přímo na připojeném fyzickém zařízení k počítači (Cuello & Vittone, 2013).

2.1.2 iOS

Vývojáři, kteří chtějí začít vyvíjet aplikace pro iOS, by měli mít základní znalosti o objektově orientovaném programování dříve, než přejdou na programovací jazyk používaný pro zařízení s iOS. Tento jazyk se nazývá Switf a jedná se o alternativu k jazyku Objectiv-C.

K vývoji je nutné vlastnit zařízení s nainstalovaným operačním systémem MacOS a spolu s tím je potřeba míti nainstalované vývojové prostředí Xcode. Xcode je vývojové prostředí od společnosti Apple, toto prostředí lze taktéž stáhnout zadarmo. Samotná aplikace pak lze testovat pomocí simulátorů na počítači. Tato možnost testování poslouží při většině času stráveném na vývoji aplikace. Ovšem tato možnost má určitá omezení, například se může zdát, že aplikace v simulátoru je mnohem rychlejší než na samotném zařízení. Proto v ideálním případě by se měli komplexnější testy aplikace provádět na fyzickém zařízení připojeným k počítači. Tyto testy ovšem vyžadují licenci pro vývojáře, která bude později potřeba pro publikování samotné aplikace na Apple Store. Cena této licence se pohybuje okolo 3000 Kč ročně (Cuello & Vittone, 2013).

2.1.3 Cross-platformní vývoj

Stále více společností se zaměřuje na vývoj mobilních aplikací pro různé platformy. Vývoj na více platforem je náročnější jak na náklady, tak i na znalosti vývojářů aplikací, kteří musí mít více znalostí, aby vytvořili aplikace, která funguje na Android, Apple a Windows současně. Právě toto zjednodušuje cross-platformní vývoj. V podstatě se jedná o to, že místo programování ve dvou jazycích se programuje pouze v jednom jazyku a tento jazyk je poté přeložen na nativní jazyk pro určené platformy. Existuje několik možností nástrojů, které jsou používané pro Cross-platformní vývoj. React Native je vytvořen společností Facebook a slouží

k vývoji aplikací pro Android, iOS a Windows. V tomto frameworku se používá jazyk JavaScript a další specifické knihovny pro tento framework. PhoneGap je framework, který vlastní společnost Adobe a je založen na HTML5, CSS3 a JavaScript. Další možností je vývojové prostředí Xamarin o kterém se dočtete níže (Elhady, 2018).

Výhody cross-platformního vývoje:

- Opakovaně použitelný kód – Nástroje pro cross-platformní vývoj umožňují napsat kód v jednom jazyce a poté ho přeložit do několika jiných jazyků pro různé operační systémy a platformy. Poté nemusíte vytvářet aplikaci pro každou platformu.
- Znalosti – Cross-platformní vývoj ušetří spoustu problémů při učení se několika programovacích jazyků pro každou platformu, místo toho nabídne jednu náhradu.
- Udržovatelný kód – Kdykoliv je potřeba upravit nebo aktualizovat aplikaci, stačí pouze jednou upravit kód a poté se změny promítnou do všech aplikací na různých platformách.
- Nákladová efektivita – Cross-platformní vývoj umožňuje ušetřit náklady spojené s tím, že více vývojářských týmů pracuje na několika aplikacích pro různé platformy. Tyto týmy lze nahradit pouze jedním.
- Rozsah – Při publikaci je možno zasáhnout více potencionálních uživatelů, pokud aplikace existuje na několika platformách, než kdyby vývoj probíhal pouze pro jednu z platforem.

  (Elhady, 2018)

Nevýhody cross-platformního vývoje:

- Výkon – Vytvořená aplikace pomocí cross-platformních nástrojů nemusí dosahovat takového výkonu, jako při použití nativních nástrojů pro vývoj aplikace. Proto pokud je prioritou výkon aplikace, měla by se vyvíjet v nativním prostředí.
- Grafika – Cross-platformní nástroje nemusí mít přístup ke všem grafickým knihovnám. Proto zde platí pravidlo jako u výkonu, pokud se aplikace zaměřuje na grafické zpracování, například u mobilních her, je lepší vyvíjet aplikaci v nativním prostředí.
- Specifické funkce – Cross-platformní nástroje umožňují přístup k různým funkcionalitám zařízení, jako například fotoaparát nebo GPS. Pokud ovšem aplikace

potřebuje přístup k specifickým částím hardwaru zařízení, je lepší uchýlit se k vytvoření nativní aplikace.

- Zpožděné aktualizace – Pokud dojde k vydání nové aktualizace pro určitou platformu, která například přidává funkce. Může trvat déle, než se těmto změnám přizpůsobí cross-platformní nástroje.

  (Elhady, 2018)

2.2 Vývoj aplikace pro iOS

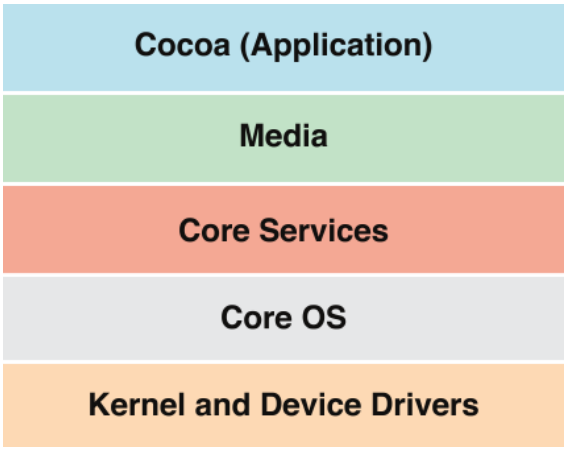
Operační systém iOS je druhým nejrozšířenějším systémem pro mobilní telefony. Tento systém můžeme nalézt pouze na zařízeních od společnosti Apple. Nativní vývojový jazyk pro tento sytém je Swift, vyvinutý společností Apple. Swift je open source programovací jazyk a jedná se o alternativu Objectiv-C. S jazykem Objectiv-C má několik podobných vlastností jako například číselné typy, většinu operátorů atd. Také má několik rozdílných vlastností jako například absenci středníků na koncích příkazů anebo nejsou vyžadovány hlavičkové soubory atd.

Pokud chce vývojář uvést aplikaci na trh musí počítat s přísnějšími podmínkami, než jak je tomu u operačního systému Android. Uživatel vlastnící zařízení se systémem iOS si může aplikaci stáhnout pouze z oficiálního App Storu. V App Storu na rozdíl od GooglePlay, schvaluje aplikaci reálná osoba a tento proces může trvat až několik týdnů (Čapek, 2014).

2.3 Architektura iOS

iOS má vrstvenou architekturu s určitými technologiemi v každé vrstvě. Spodní vrstvy poskytují základní služby, další vrstvy poskytují více a více důmyslnější služby a doplňují nebo vycházejí ze spodní vrstvy. Architektura se dělí do 5 vrstev, které jsou seřazeny v pořadí, jak je uvedeno na obrázku číslo 1 (Apple Inc., 2015).

Obrázek 1: Vrstvy iOS architektury



Zdroj: (Apple Inc., 2015)

2.3.1 Cocoa (Application) Layer

Tato vrstva je hlavně zodpovědná za vzhled aplikace a odezvu na uživatelské akce. Dále definuje několik funkcí pro uživatele, jako jsou např. centrum notifikací, Full-Screen mód, Game Center atd. Také obsahuje různé frameworky pro vývoj aplikací (Apple Inc., 2015):

- **AppKit** – Tento framework implementuje uživatelské rozhraní aplikace včetně oken, kontrolních prvků, různých nabídek, event handling a další.
- **Foundation** – Framwork foundation implementuje například přístup k souborům, celý management dat, notifikace a síťovou komunikaci.

  Core Data – řídí datový model aplikace založený na vzorovém modelu Model-View-Controller (MVC). Tento framework se doporučuje při práci s velkými datovými sadami (Apple Inc., 2015).

2.3.2 Media Layer

Vrstva médií, zahrnuje 2D a 3D grafiku, animace, obrazové efekty, audio a video funkce. Pro detailnější nastudování těchto frameworků doporučuji navštívit dokumentaci pro vývojáře od společnosti Apple. Tato vrstva obsahuje desítky frameworků jako například (Apple Inc., 2015):

- **Grafické technologie** - Core Graphics, Core Animation, SpriteKit, Scene Kit, Metal, OpenGL, GLKit.
- **Textu, typografie a fonty** - Cocoa text systém a Core Text.
- **Obrázky –** Image Capture Core, Core Image, Image Kit, Image I/O.

- **Management barev –** Slouží k rychlé, konzistentní a přesné reprodukci barev, kontrole a kalibraci.
- **Tisk –** Slouží například k podpoře faxu, PDF nebo tiskového zobrazení atd.
- **Audio technologie –** AV Foundation, OpenAL, Core Audio.

Video technologie – AVKit, AV Foundation, Core Media, Core Video (Apple Inc., 2015)

2.3.3 Core Services Layer

Tato vrstva poskytuje základní služby aplikacím, ale nemá přímý vliv na uživatelské rozhraní. Také tato vrstva vychází z technologií Core Layer a Kernel and Device Drivers Layer. V této vrstvě můžeme nalézt (Apple Inc., 2015):

- **Social Media Integration –** Frameworky, které sdílejí obsah různými sociálními službami. Jednou z nich je Accounts, který poskytuje přístup k podpoře různých typů účtů a ukládá je v databázi účtů. Druhým je Social, tento framework poskytuje API pro posílání požadavků na podporované služby sociální médií.
- **iCloud Storage –** Tyto frameworky zajišťují ukládání dat na iCloud. Document storage je viditelný pro uživatele, slouží například k ukládání dokumentů, prezentací apod. Key-value storage poskytuje ukládání malého množství dat, například u her při ukládání scóre. Posledním frameworkem je Core Data storage, tento framework je určený pro ukládání strukturovaného obsahu na databázové bázi.
- **CloudKit –** Poskytuje aplikaci kontrolu nad tím, kdy a jak byla data uložena na iCloud.
- **File Coordination –** Eliminuje nesrovnalosti systému souborů v důsledku překrývajících se operací, čtení a zápisu z konkurenčních procesů.
- **Bundles and Packages** – Spravuje mechanizmus svazků. Svazky zapouzdřují související zdroje v hierarchické struktuře souborů.
- **Internationalization and Localization –** Používá se například k úpravě textu podle toho, v jaké zemi se uživatel nachází.

  Serurity Services – Implementuje vrstvu služeb na vysoké úrovni pro zjednodušení bezpečnostních řešení. Poskytuje následující funkce: Ověření uživatele, certifikáty, klíč, autorizační služby, služby Keychain a další (Apple Inc., 2015).

2.3.4 Core OS Layer

Tato vrstva poskytuje služby v oblasti hardwaru a sítí nízké úrovně. Tyto služby jsou založeny na zařízeních ve vrstvě Kernel a Device Drivers. Tato vrstva také implementuje funkce týkající se zabezpečení aplikace (Apple Inc., 2015):

- **Gatekeeper** – umožňuje zablokovat instalaci aplikace, která nepochází z Mac App Store a identifikuje vývojáře. Pokud aplikace nemá podpis s certifikátem ID vývojáře vydaným od společnosti Apple, nespustí se v systému, pokud nemá systém povolený spouštět tyto aplikace.
- **App Sandbox** – poskytuje obranu proti kradení, poškození nebo smazaní uživatelský dat, jestliže škodlivý kód zneužívá aplikaci. AppSandbox umožňuje popsat spolupráci, aplikace se systéme. Systém pak uděluje aplikaci pouze přístup, který potřebuje k jejímu fungování.

  Code Signing – Operační systém OS X používá bezpečnostní technologie, které vám umožní ověřit, zda byla vaše aplikace opravdu vytvořena vámi. Systém je schopen zjistit jakoukoliv změnu aplikace, ať už byla provedena náhodně či škodlivým kódem (Apple Inc., 2015).

Tato vrstva také obsahuje následující technologie a frameworky:

- **Accelerate** – Tento framework pomáhá zrychlovat složité operace a zlepšit výkon pomocí vektorové jednotky. Vektorové jednotky zvyšují výkon aplikací, které využívají paralelní data, jako například 3D grafické zobrazování, zpracování obrazu a videa, nebo komprese zvuku atd.
- **Disk Arbitration** – Disk Arbitation oznamuje vaši aplikaci, kdy jsou připojeny a odpojeny místní a vzdálené svazky.
- **OpenCL** – Umožňuje vysoce výkonný paralelní proces GPU, který je k dispozici pro univerzální výpočty. Pomocí OpenCL vytvoříte jádra pro výpočty, které jsou poté dostupny na grafickou kartu nebo CPU pro zpracování.
- **Open Directory** – je architektura adresářových služeb, které poskytují centralizaci cest pro načítání informací uložených v místních nebo síťových databázích. Tyto služby zpravidla poskytují přístup ke shromážděným informacím o uživatelích, skupinách, počítačích, tiskárnách a ostatních informacích, které jsou v síťovém prostředí.

- **Systém Configuration** – tento rámec pomáhá aplikaci s nastavením sítě a zjištěním, zda je možné se k nim připojit. Poskytuje přístup k informacím o konfiguraci sítě. Umožňuje aplikacím zjištění dosažitelnosti vzdáleného přístupu. Upozorňuje aplikaci, když dojde ke změnám v síťovém stavu a síťovém nastavení. Poskytuje flexibilní schéma pro definování a přístup k uloženým preferencím a stávajícím síťovím nastavením (Apple Inc., 2015).

2.3.5 Kernel and Device Drivers Layer

Tato vrstva je nejnižší z vrstev OS X. Obsahuje kernel, ovladače a BSD části systému. Hlavně je založena na open source technologií. OS X rozšiřuje tuto nejnižší vrstvu prostředí s několika klíčovými technologiemi, které umožňují jednoduší vývoj softwaru. Tyto následující funkce se nacházejí ve vrstvě Kernel and Device Drivers Layer (Apple Inc., 2015):

- XPC – je komunikační technologie OS X, která doplňuje App Sandbox.
- Caching API – je rozhraní pro ukládání do mezi paměti s nízkou úrovní. Tato technika ukládání do mezi paměti je důležitá pro maximalizaci výkonu aplikací. Pokud je ale překročena dostupná paměť, může docházet v celém systému k degeneraci výkonu.
- In-Kernel Video Capture – poskytuje programovací rozhraní C++ pro vytváření ovladačů pro zachycení videa. Tato funkce nahrazuje rozhraní API QuickTime, grabber API jako prostředek pro získání videa do OS X.
- Kernel – základ pro rozhraní operačního systému OS X je UNIX. Prostředí Kernel je postaveno na Mach 3.0 a poskytuje vysoce výkonné síťové zařízení a podporu pro více integrovaných souborových systémů.
    - Mach – poskytuje některé z nejdůležitějších funkcí operačního systému. Hodně z funkcí, které Mach poskytuje je transparentní pro aplikace. Řídí procesorové zdroje, jako je využití procesoru, paměti a zpracovává plánování atd.
- 64-Bitový Kernel – poskytuje několik výhod:
    - Efektivněji podporuje velké konfigurace paměti
    - Maximální velikost vyrovnávací mezi paměti je zvýšena.
    - Je vylepšen výkon při práci se specializovaným síťovým hardwarem.
- Device-Driver Support – nabízí objektivně orientované frameworky pro vývoj ovladačů na zařízení, nazývající se I/O Kit framework. Tyto frameworky usnadňují vytváření

ovladačů pro systém OS X a poskytují velkou část infrastruktury, kterou potřebují. I/O kit je modulární a rozšiřitelný.

- BSD - je vlastní verze operačního systému Berkeley Software Distribution (BSD). BSD slouží jako základ pro souborové systémy a síťová zařízení OS X. navíc poskytuje několik programovacích rozhraní a služeb.

  A další jako například Network Support, IPC and Notification mechanisms, Threading Support (Apple Inc., 2015).

2.4 Vývoj na vývojové platformě Xamarin

Nyní si popíšeme, jak vlastně platforma Xamarin funguje. Zdrojový kód napsaný v C# je kompilován pomocí Xamarin.iOS. Xamarin.iOS používá speciální podskupinu Mono frameworku. Mono je open-source projekt pro vytvoření softwarového frameworku se standardem ECMA kompatibilního s rozhraním .NET Framework, včetně kompilátoru C#. Samotný rámec umožňuje přístup k určitým funkcím iOS platformy. Xamarin.iOS zkompiluje kód do přechodného jazyku ECMA CIL (common intermediate language). Po této kompilaci do jazyka CIL je nutné ho znovu zkompilovat do nativního strojového kódu, který může být spuštěn na iOS zařízení. Proces převodu kódu z jazyka CIL do nativního kódu probíhá pomocí SDK nástroje „Mtouch“. Tento nástroj vrátí balíček aplikací, který může být nasazen do iOS simulátoru nebo do zařízení s operačním systémem iOS (Giri, 2016).

2.4.1 Vývoj uživatelského rozhraní

Uživatelské rozhraní lze vyvíjet na platformě Xamarin dvěma způsoby. Jedním z těchto způsobů jsou nativní API s názvem Xamarin Nativ (vývojáři uvádějí spíše Xamarin.iOS nebo Xamarin.Android), který je integrován do Visual Studia. Jedná se o knihovny C#, které umožňují vývojářům přístup k iOS SDK a Android SDK, tyto zabalené API, nativní pro mobilní platformy dovolují přístup k požadovaným funkcím skrze standartní sadu nástrojů C#. Pro iOS funguje tato API na principu návrháře pro formát storyboardu iOS a zachovává plnou kompatibilitu s formátem storyboardu, takže lze editovat v Xcode nebo Visual Studiu (Applikey team, 2018).

Zde si uvedeme výhody využití nativních API pro vývoj uživatelského rozhraní ve vývojové platformě Xamarin. Pokud potřebujete menší velikost aplikace, doporučuje se sáhnout po Xamarin Nativ, protože použití Xamarin.Forms vede většinou k větší velikosti aplikace. S větší velikostí aplikace přichází delší načítání aplikace, a to zejména v systému Andorid. To se ale již zlepšilo od roku 2016 kdy Xamarin a Mircosoft spojily síly a udělali

veliké pokroky ve výkonnosti. Další výhoda se také týká výkonosti aplikace, pokud je potřeba použít mnoho animací nebo komplikované uživatelské rozhraní. Xamarin.Forms, jak už název naznačuje, byl navržen, aby nabídl rámec zaměřený na jasné toky informací, založeních na formách nebo rozhodnutích aplikace. Jestliže je tedy potřeba vysoce dynamického obsahu, může být vhodnější uchýlit se k navržení uživatelského prostředí aplikace v nativním prostředí. Další výhoda se týká toho, jestliže chcete využít některé z funkcí, které nejsou dispozici na obou platformách. Pokud jedna z platforem přijde s novým zařízením na trh, přidává nové funkce i ovládací prvky, které jsou pro určitou dobu na této platformě exklusivní. Vždy je nutno aby, jste tyto ovládací prvky implementovat sami ve vaší aplikaci pomocí Xamarin.Forms. Tím se ale zvýší strávený čas nad vývojem aplikace. Pokud je tedy potřeba implementovat ovládací prvky které nejsou cross-platformní, je nejspíše lepší využít nativního vývojového prostředí pro tuto platformu (Nathan, 2017).

Druhý způsob vývoje uživatelského rozhraní je Xamarin.Forms. Xamarin.Forms je kompletní sada nástrojů pro vytváření cross-platformní aplikace. Nástroje pracují pomocí knihoven Xamarin.iOS a Xamarin.Android. A dovolují psát front-end v C# a XAML. Dokonce dovoluje sdílet tento kód mezi implementacemi. Aplikace vytvořené pomocí Xamarin.Forms jsou vybavené všemi vlastnostmi pro co nejlepší cross-platformní aplikace. To umožňuje vývojářům ušetřit čas i peníze, protože přibližně 80-95 % kódu je cross-platformní a pouze 20-5 % kódu obsahuje příkazy pro vytvoření spojení mezi nativními API (Applikey team, 2018).

Nyní si uvedeme některé výhody použití Xamarin.Forms. Jak již bylo uvedeno, u tohoto způsobu vývoje uživatelského rozhraní je velikou výhodou možnost sdílení kódu mezi platformami. Tato vlastnost Xamarin.Forms má veliký dopad na celkové náklady a na čas strávený při vývoji. Jelikož není potřeba vyvíjet uživatelské prostředí na každou platformu zvlášť, jako je tomu u Xamarin Nativ. Tato vlastnost také pomáhá, pokud je nutné dostat cross-platformní aplikaci rychle na trh. Jednou z největších výhod Xamarin.Forms při používání pro cross-platformní aplikace, je jednodušší aktualizace a údržba samotné aplikace. Jelikož je většina uživatelského rozhraní sdílená, znamená to, že aktualizace funkcí lze mnohem snadněji rozšířit na Android i iOS. Za poslední výhodu bych označil to, že pokud má vývojář zkušenosti s technologií Microsoft a C #, pak je může v Xamarin.Forms maximálně využít. To znamená, že můžete trávit daleko méně času učením se vytváření uživatelského rozhraní v nativním prostředí (Nathan, 2017).

2.5 Potřebné nástroje pro vývojovou platformu Xamarin

Pro vývoj na platformě Xamarin potřebujete buďto nejnovější verzi Visual Studio Enterprise, Visual Studio Community nebo Visual Studio Professional. A do jedné z těchto verzí Visual Studio si poté už stačí stáhnout balíček od Xamarin. Další možností, jak využít tuto vývojovou platformu, je si stáhnout Xamarin studio. Xamarin studio bylo poprvé představeno v roce 2013 jako samostatné vývojové prostředí na Windows a MacOS, pro vývoj mobilních aplikací. Tyto dvě možnosti jsou stejné jak pro MacOS i pro Windows. Dále se nástroje liší podle používaného operačního systému.

2.5.1 Vývojová platforma Xamarin.iOS – MacOS

Vývoj pomocí platformy Xamarin na MacOS je snazší. Jediné, co je potřeba na zařízení s MacOS, je nainstalovat Xcode. S tímto IDE se již stáhnout potřebné balíčky a simulátor.

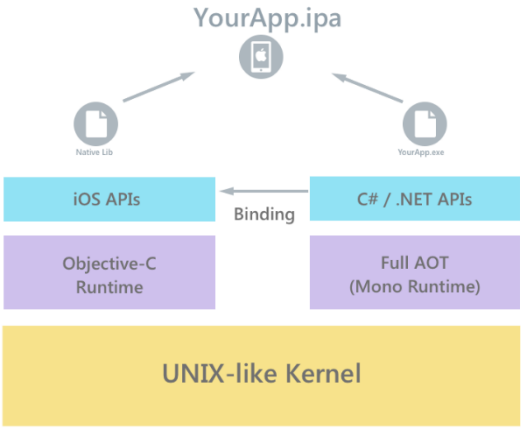
2.5.2 Vývojová platforma Xamarin.iOS – Windows

U operačního systému Windows je to všechno o něco komplikovanější, protože je potřeba nativního vývojového prostředí. To lze vyřešit dvěma způsoby. Jedním z nich je pomocí virtuálního počítače. V mém případě se jedná o VMWare 12 Player na který je potřeba nainstalovat jednu z nejnovějších verzí MacOS. Poté se nástroje shodují i druhým způsobem, a to je pomocí vzdáleného přístupu k fyzickému zařízení s operačním systémem MacOS. Další nástroj, který je potřeba je simulátor pro zařízení s operačním systémem iOS. Pro další postup je potřeba na počítač s MacOS nainstalovat Xcode a Visual studio pro MacOS. Posledním krokem je spárovat zařízení Windows se zařízením MacOS. U všech těchto nástrojů je potřeba dát si pozor, aby byly dostupné v co nejvyšších verzí, a to i Mono framework, který se nainstaluje spolu s Visual Studiem. Jak je patrné, vývoj na zařízení s operačním systémem iOS je na systému Windows náročnější, a to nejen při přípravě vývojového prostředí, ale také kvůli hardwarové náročnosti.

2.6 Architektura aplikace

Xamarin.iOS aplikace se spouští pomocí Mono frameworku a k tomu se používá Full Ahead of Time (AOT) kompilace a C# kód do strojního jazyka ARM (určitý druh architektury procesorů). Toto běží vedle Runtime kompilace Objectiv-C. Pomocí těchto kompilací se sestaví samotná aplikace. Oba tyto procesy probíhají na jádře UNIX-like kernel, přesněji na XNU. XNU je operační systém kernel, který byl vyvinut společností Apple roku 1996 pro použití v zařízeních s MacOS operačním systémem. Tyto kompilace jsou popsány na obrázku číslo 2 (Microsoft a. s., 2017).

Obrázek 2: Diagram sestavení aplikace

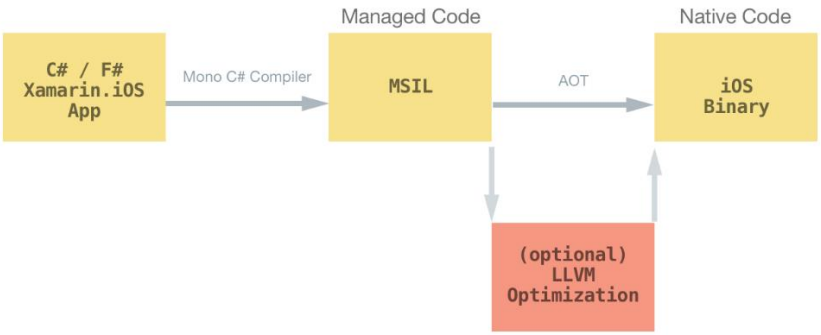


Zdroj: (Microsoft a. s., 2017)

Při kompilaci kódu na platformě Xamarin, je spuštěn kompilátor Mono C# jak již bylo uvedeno víše. Tento kompilátor kompiluje C# do jazyka Common Intermediate Language (CIL). Pokud spustíme aplikaci Xamarin.iOS na simulátoru, kompiluje modul .NET Common Language Runtime jazyk CIL pomocí kompilátoru typu Just In Time do nativního kódu, který lze poté spustit na správné architektuře aplikace.

Existuje zde však omezení od společnosti Apple, které nepovoluje provádění dynamického generování kódu v zařízení. Pro dodržování tohoto nařízení, platforma Xamarin používá kompilátor AOT ke kompilování CIL. Tento proces vytváří nativní kód, který může být optimalizován pomocí překladače LLVM a poté použit pro procesory ARM. Tento proces kompilace si můžete prohlédnout na obrázku číslo 3. Tento diagram není úplně aktuální, jelikož Microsoft Intermediate Language (MSIL) byl přejmenován na Common Intermediate Language (CIL) (Microsoft a. s., 2017).

Obrázek 3: Diagram kompilátoru



Zdroj: (Microsoft a. s., 2017)

2.7 Architektonické modely

Architektonický model přináší obraz systému, není to architektura jako taková. Architektonický model je koncept, který řeší a vymezuje některé nezbytné soudržné prvky softwarové architektury. Mnoho architektonických modelů může implementovat stejný model a tím sdílet související vlastnosti. Proč používat architektonické modely (AAHN INFOTECH, 2012)?

- Rozložení komplexnosti – Pokud je aplikace příliš komplexní a složitá, tak architektonický model pomůže zjednodušit její vývoj a správu, tím že aplikaci rozloží zodpovědnost mezi několik entit na základně jednotné odpovědnosti
- Testovatelnost – Zjednoduší testování aplikace za běhu, pokud je potřeba přidat nové funkce nebo předělat jednu z hlavních tříd.
- Méně kódu – Čím méně kódu je, tím méně chyb v něm existuje.

Existuje několik typů architektonických modelů:

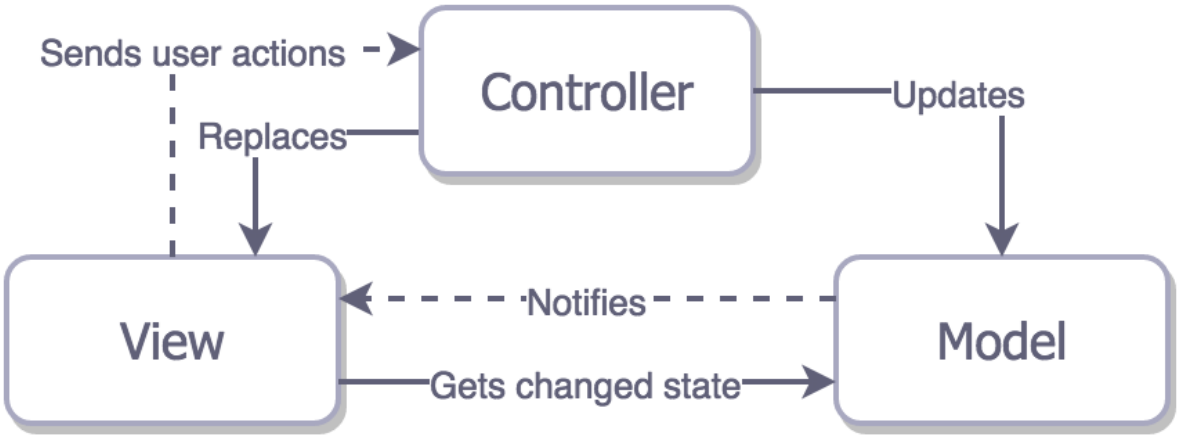
1. Broken Pattern – Tento model se používá pro distribuování softwarových systémů s oddělenými komponenty, které reagují na vzdálené vyvolání služby. Je zodpovědný za řízení komunikace, například jako předávání žádostí nebo přenos výsledků.
2. MV(X) – Tento model rozděluje aplikaci do tří částí: Model, Views a Controller. V případě různých variant ještě na Presenter a ViewModel. Část Model je zodpovědná za data nebo vrstvu pro přístup k datům, která manipuluje s daty. Views zobrazuje informace uživateli a je zodpovědná za prezentační vrstvu (GUI). Controller je prostředník mezi vrstvami Model a Views, v podstatě je zodpovědná za změny v části Model a tím že reaguje na akce uživatele v části Views a poté aktualizuje zobrazení v části Views a upravuje část Model.
3. Presentation-Abstraction-Control (PAC) – určuje strukturu interaktivních softwarových systémů ve formě spolupracujících částí. Každá z těchto částí je zodpovědná za určité funkce aplikace. Dělí se na tři části, které mají za účel oddělení interakce člověka od funkčního jádra a komunikaci s jinými částmi.
4. Mircrokernel pattern – se používá u systémů, které se musí umět přizpůsobit systémovým požadavkům. Odděluje funkční jádro od rozšířené funkčnosti a dále od částí specifického pro každého uživatele zvlášť.

5. Reflection pattern – poskytuje mechanismus pro dynamické změny v systému. Podporuje změny základních částí systému, jako je například volání funkcí nebo typové struktury. Aplikace, která používá tento model je rozdělena na dvě části. Meta úroveň poskytuje informace o určitých vlastnostech systému a Base úroveň obsahuje logiku aplikace.
6. Pipes and Filters – tento model poskytuje strukturu pro systémy, které zpracovávají proud dat. Každý krok zpracování je zde zapouzdřen v komponentě s názvem filtr. Data poté procházejí mezi těmito filtry (AAHN INFOTECH, 2012).

2.7.1 Model View Controller

V tomto případě klasického MVC modelu, část View nemá žádný stav. Prostě je znovu vykreslený komponentou Controller potom co se změní Model. Pokud se na to podíváme z pohledu webové stránky, tak je pokaždé celá vykreslena, vždy když kliknete na odkaz aby, jste se mohli dostat na jinou stránku. Tento model v původním stavu není příliš vhodný pro iOS aplikaci, protože všechny jeho tři části jsou úzce spojeny a každá z těchto částí ví o ostatních dvou, a to snižuje opakovatelnost u každé z nich. Funkčnost původního modelu MVC je popsána v obrázku číslo 4 (Orlov, 2015).

Obrázek 4: architektura MVC



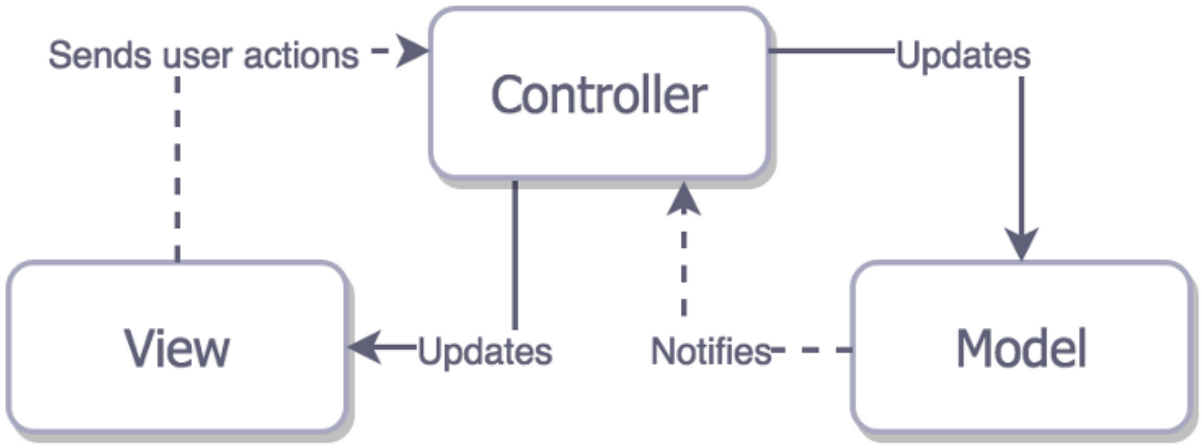
Zdroj: (Orlov, 2015)

2.7.2 Apple Model View Controller

Pro vývoj na iOS aplikacích se předpokládá použití upraveného MVC modelu. Controller je zde prostředníkem mezi ostatními částmi. To znamená že model definuje nejen role částí, ale i způsob vzájemné komunikace mezi částmi. Každá z těchto tří částí je oddělena

abstraktními hranicemi a komunikuje s ostatními částmi přes tyto abstraktní hranice. Výběr určitých objektů jedné části se někdy označuje jako vrstva. Mnoho těchto objektů bývá v aplikacích opakovaně použito a jejich rozhraní bývá dobře definované. Jednotlivé rozvržení částí toho upraveného modelu si můžete prohlédnout na následujícím obrázku číslo 5 (Orlov, 2015).

Obrázek 5: Architektura Apple MVC



Zdroj: (Orlov, 2015)

2.7.3 Model View Presenter

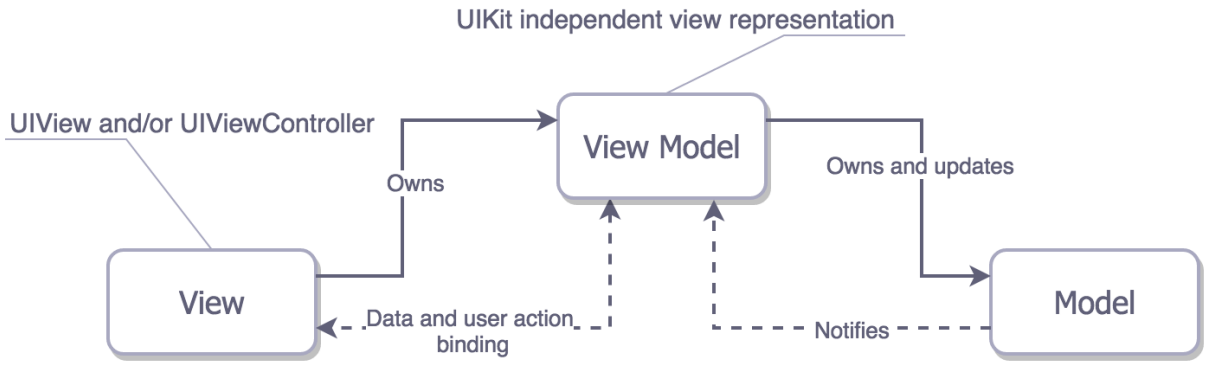
MVP je zkratka pro Model-View-Presenter a má původ v modelu MVC. Podobně jako model MVC odděluje tři části aplikace do specializovaných komponent. Tento model se nejvíce používá pro vytváření uživatelských rozhraní. Zde přebírá funkci prostředníka Presenter. Model je rozhraní, co má za úkol určovat data, která se zobrazí v uživatelském rozhraní. View je zde pasivní rozhraní které zobrazuje data z části Model a posílá uživatelské příkazy do Presenteru. Presenter poté podle příkazů uživatele načte data z části Modelu a zformátuje je pro zobrazení ve View. Jednou z běžných vlastností MVP je, že je zde hodně dvou směrného odesílání. Například Presenter dostane příkaz od View, že bylo stisknuto „uložit“, následně zavolá metodu pro ukládání. Jakmile je ukládání hotovo, Presenter zavolá zpátky View skrze rozhraní a View může zobrazit, že ukládání bylo hotovo (Orlov, 2015).

2.7.4 Model View ViewModel

MVVM je zkratka pro Model-View-ViewModel a jedná se o poslední z řady modelů MV(X). Tento model a jeho prvky jsou zobrazeny na obrázku číslo 6. Komponenty View a Model jsou nám již známi z modelu MVC. Jako prostředník zde funguje ViewModel a není zde žádný úzký vztah mezi Model a View. V praxi to vypadá tak že ViewModel vyvolá změnu v Model a sám se aktualizuje s aktualizovaným Model. Díky vazbě mezi View a ViewModel,

je View aktualizován odpovídajícím způsobem. ViewModel je tedy zodpovědný za zobrazování metod, příkazů a dalších funkcí, které pomáhají při udržování stavu View. A také manipulovat s částí Model, jako s výsledkem akcí ve View a spouštět v části View samotné události (Orlov, 2015).

Obrázek 6: Architektura MVVM



Zdroj: (Orlov, 2015)

2.7.5 VIPER

Viper nepochází z kategorie MV(X), a proto je jeho rozložení je zajímavé. Tento model rozděluje aplikaci do pěti částí:

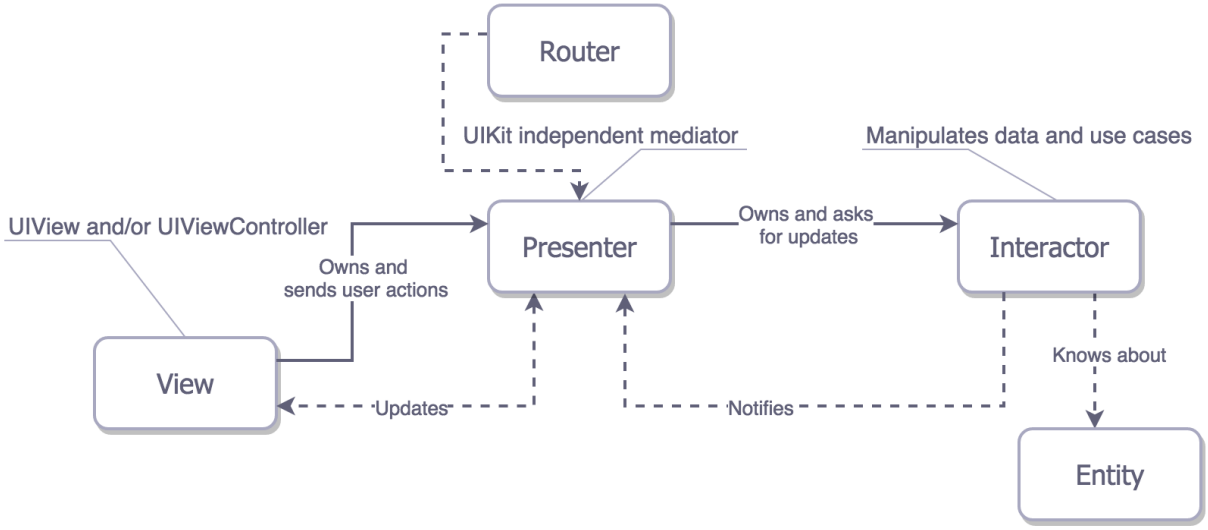
- Interactor – Obsahuje logiku týkající se dat nebo sítě, jako je vytváření nových instancí entit nebo jejich načítání ze serveru.
- Presenter – Obsahuje uživatelské rozhraní související s logikou aplikace a vyvolává metody v Interactor.
- Entities – Toto jsou pouze datové objekty, nejedená se o datovou vrstvu, protože za přístup k datům má zodpovědnost Inceractor.
- Router – Má zodpovědnost za přecházení mezi moduly VIPERu.
- View – Je třída, která má celý kód pro zobrazení uživatelského rozhraní a získání jeho odpovědi.

Viper je architektura řízená delegací, takže většina komunikace mezi vrstvy probíhá prostřednictvím delegace. Jedna vrstva volá jinou pomocí protokolu. Z protokolu volá funkci a volaná vrstva odpovídá tomuto protokolu. Na obrázku číslo 7 můžete vidět jednotlivé rozvržení částí a cesty komunikace mezi nimi.

Výhody VIPERu se z části podobají MV(X) modelům, a to tím, že jednotlivé části jsou zde od sebe dobře izolovány, a proto je jednodušší opravovat chyby. Stačí pouze aktualizovat

určitý modul. Také tento model vytváří velmi dobré prostředí pro jednotkové testování. Jelikož je každý modul nezávislý na ostatních, tak je zde velmi jednoduché dělit práce mezi jednotlivé vývojáře. VIPER by se měl použít pouze pokud jsou požadavky na aplikaci velmi dobře stanoveny. Změny v požadavcích mohou způsobit zmatek a chybový kód. Proto se u malých aplikací vyplatí využít modelů MV(X) a VIPER použít pouze pokud celý vývojářský tým pochopí celou složitost aplikace (Alam, 2017).

Obrázek 7: Architektura Viper



Zdroj: (Orlov, 2015)

3 Vývoj mobilní aplikace

Vytvořená aplikace je ve formě jednoduchého plánovače pro aktivity. Jednotlivé aktivity nebo úkoly jsou dále popisovány jako Tasky, jelikož je tak pojmenována třída, která je pro ně datovým typem. Task má jednotlivé parametry, kterými jsou název, krátký popisek, zapnutí lokální notifikace, datum začátku Tasku, přidání do Apple kalendáře, doba trvání Tasku a kategorie. Popřípadě si uživatel může vytvořit i vlastní kategorie. Pro práci s daty je zde vytvořená třída TaskManager a o ukládání a načítání dat ze souboru se stará třída DataStorage.

3.1 Návrh architektonického modelu

Na obrázku číslo 8 vidíte návrh architektury aplikace. Při tomto návrhu bylo dbáno na to, aby byly dodrženy základní zásady architektury MVVM. To znamená, že zde neprobíhá žádná těsná komunikace mezi částmi View a Model. Jednotlivé části architektury si probereme v následujících kapitolách.

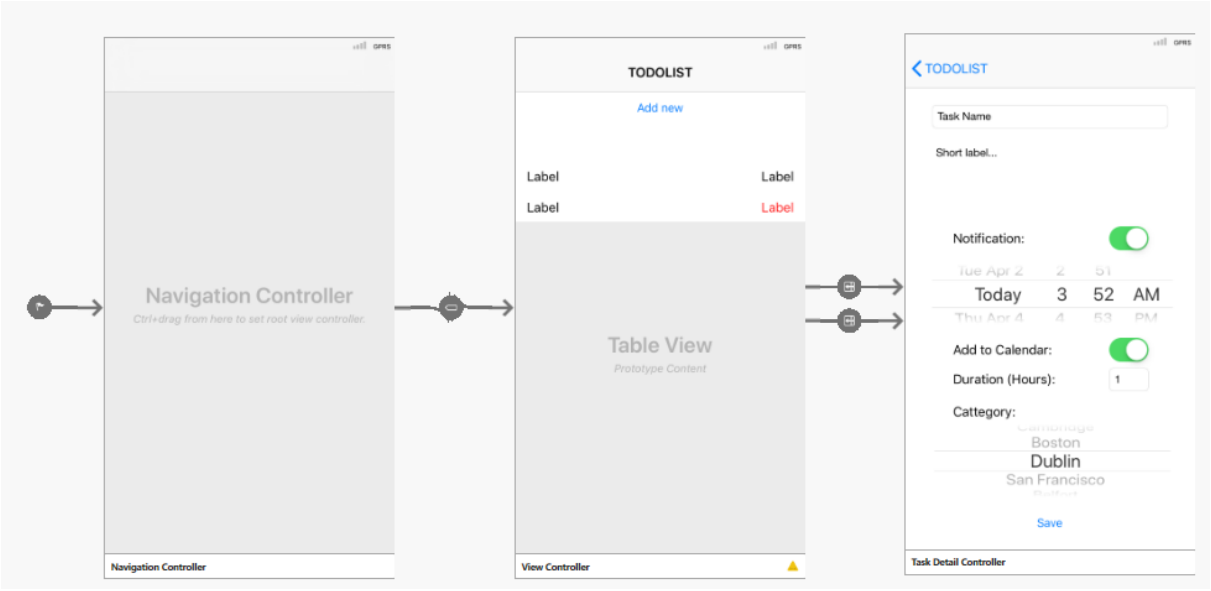
Obrázek 8: Návrh modelu MVVM

Zdroj: (Autor)

3.2 Část View

Grafické rozhraní aplikace je vytvořené pomocí nástroje Xamarin.iOS. Tento nástroj funguje na principu drag and drop a jeho používání funguje velmi intuitivně. Přidání jednotlivých elementů se provádí jejich přetažením na pracovní plochu. Poté se jim přidají nebo nastaví určité vlastnosti, které jsou následně využity v kódu. Aplikace byla navržena do dvou obrazovek, jak je uvedeno na obrázku číslo 9. Levá obrazovka s názvem ViewController, zobrazuje tabulku Tasků a tlačítko přidání. Pravá obrazovka obsahuje detail samotného Tasku se všemi parametry, které se u něj dají nastavit. Tato obrazovka se nazývá TaskDetailController. Dále je na obrázku číslo 9 vidět prvek NavigationController, umístěný v levé části. Tento prvek má za důsledek částečně automatické zpracování přechodu mezi obrazovkami. Vazba mezi prvky NavigationController a ViewController určuje, že ViewController je takzvaný „root“. To znamená, že se zobrazí jako první po zapnutí aplikace.

Obrázek 9: Návrh uživatelského rozhraní



Zdroj: (Autor)

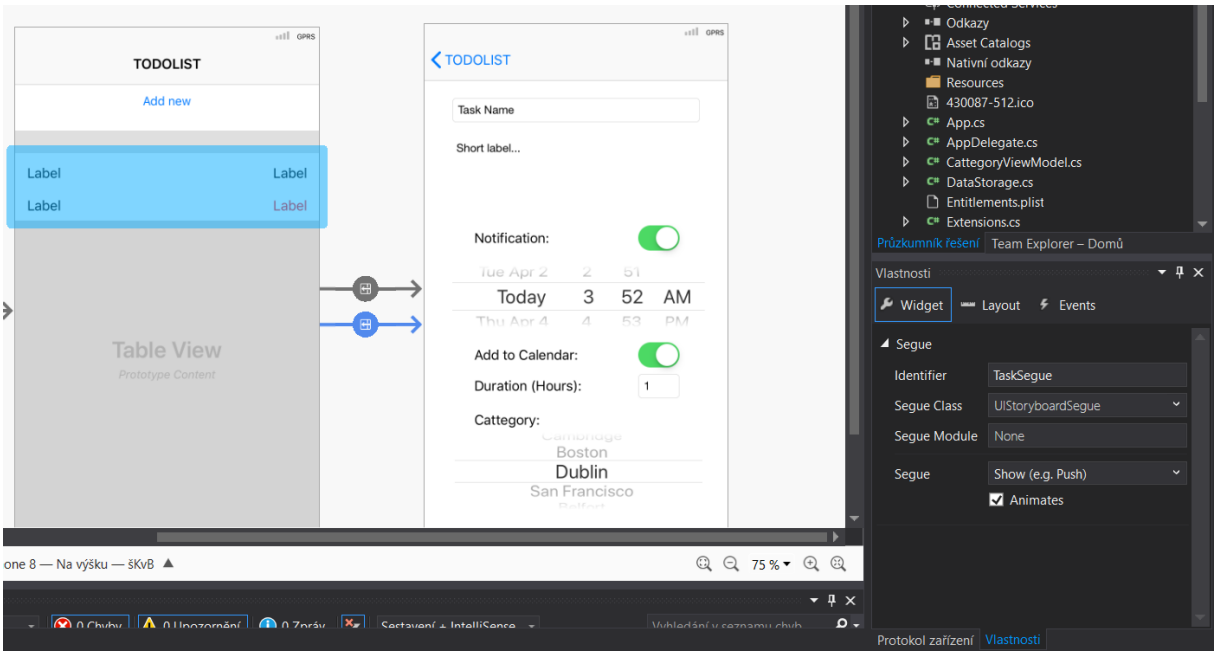
3.2.1 Zobrazení textu

Za zobrazování textu v této aplikaci je zodpovědný prvek Label, který je zde použit několikrát. Využité vlastnosti Labelu v této aplikaci, jsou pouze identifikátor a grafické úpravy textu, jako je například zarovnání a barva textu. Zobrazení textu, které není třeba dynamicky měnit, se může nastavit přímo v Xamarin.iOS. Pokud je ale nutné text změnit v průběhu užívání aplikace, je ho možné upravit v kódu pomocí identifikátoru a vlastností Text.

3.2.2 Přechody mezi obrazovkami

Jak již bylo uvedeno, po přidání prvku NavigationController se částečně automaticky hlídá přechod mezi obrazovkami. Tyto přechody jsou řešeny pomocí zásobníku. Pokud se otevře nová obrazovka, tak je přidána do zásobníku a následně se automaticky zobrazí tlačítko v horní části obrazovky, které odkazuje na předcházející obrazovku. Toto tlačítko se zobrazuje tak dlouho, dokud se uživatel neocitne zpátky na obrazovce „root“. Přechod na novou obrazovku již závisí na návrhu uživatelského rozhraní aplikace. Různé možnosti přechodu na novou obrazovku jsou znázorněny modrými šipkami, jak je možné vidět na obrázku číslo 10.

Obrázek 10: Nastavení přechodu mezi obrazovkami



Zdroj: (Autor)

V této aplikaci je přechod na obrazovku TaskDetailController možný dvěma způsoby. První ze způsobu je pomocí tlačítka Add New. Po stisknutí tohoto tlačítka se uživateli zobrazí TaskDetailController s předdefinovaným hodnotami. Uživatel poté může vytvořit nový Task a uložit ho pomocí tlačítka Save.

Druhá možnost je přes již vytvořený Task. Tato možnost je komplikovanější, jelikož předává index Tasku v tabulce společně se samotným Taskem, který byl stisknut na obrazovce ViewController. Tato akce je nutná proto, aby se mohli hodnoty již vytvořeného Tasku nastavit jako výchozí. Proto musí být také nastaven identifikátor, aby se na něj mohlo později odkazovat v kódu. Nastavení tohoto identifikátoru můžete vidět na obrázku číslo 10. Podobně jako na obrázku číslo 10, se nastavují veškeré vlastnosti všech ostatních prvků uživatelského rozhraní. V této akci se také nastaví vlastnost Segue, která určuje animaci při otevírání obrazovky TaskDetailController. V kódu se poté volá metoda PrepareForSegue (Ukázka kódu 1). V této metodě se podle identifikátoru zjistí, jaký segue byl metodě předán (řádek 32). Poté se určí, kam se data budou předávat (řádek 34) a následně dojde k získání informace o tom, která buňka byla stisknuta. Tyto data se poté předají metodě SetTask. Metoda SetTask je definována ve třídě TaskDetailController a dále je popsána v podkapitole TaskDetailController.

Ukázka kódu 1: Metoda PrepareForSegue

```
30      public override void PrepareForSegue(UIStoryboardSegue segue, NSObject sender)
31      {
32          if (segue.Identifier == "TaskSegue")
33          {
34              var navctrl = segue.DestinationViewController as TaskDetailController;
35              if (navctrl != null)
36              {
37                  var source = TaskTableView.Source as TaskTVS;
38                  var rowPath = TaskTableView.IndexPathForSelectedRow;
39                  var item = source.GetItem(rowPath.Row);
40                  navctrl.SetTask(this, item, (int) rowPath.Row);
41              }
42          }
43      }
```

Zdroj: (Autor)

(Ukázka kódu 2) Tlačítko „Save“, které je definováno v TaskDetailControlleru, má nastavený pouze identifikátor. Při stisku tohoto tlačítka dochází ke spuštění metody Save_TouchUpInside a na jejím konci dojde k přechodu na předchozí obrazovku (řádek 177). Tato metoda vytvoří třídu TaskManager a následně se pomocí ní volá metoda AddTask nebo EditTask. Tato třída je popsána i s uvedenými metodami v části věnující se třídě TaskManager. Pokud je TaskDetailController otevřen pomocí stisku na již vytvořený Task, tak se nastaví proměnná bool EditingEnable na hodnotu True. Tato proměnná určuje, zda se bude editovat stávající Task nebo se bude vytvářet zcela nový (řádek 163). Jak již bylo zmiňováno, předává se index Tasku, který byl stisknutý. Poté podle toho metoda Save_TouchUpInside zachází se vstupy a volá správnou metodu třídy TaskManager. Pokud se jedná o editaci, musí se navíc metodě EditTask předat index původního Tasku. Dále se ve funkci Save_TouchUpInside zpracovávají veškeré vstupy a kontroluje se, zda jsou ve správném formátu. Tato kontrola se hlavně týká vstupu, který určuje délku jednotlivých Tasků, jelikož se proměnná získaná

z tohoto vstupu dále používá při propojení s Apple kalendářem. Proto se u této proměnné ověřuje, zda se jedná o číslo, zda není menší nebo rovno nule a jestli není větší než 24.

Ukázka kódu 2: Metoda Save_TouchUpInside

```
163         if (!EditingEnable)
164         {
165             DateTime inputDate = DateTime.SpecifyKind(Extensions.NSDateToDateTime(DatePicker.Date), DateTimeKind.Utc).ToLocalTime();
166             taskManager.AddTask(new Task(TaskName.Text, TaskShortLable.Text, Cattegory, DatePicker.Date, NotificationSwitch.On,
167                 CalendarSwitch.On, durationNumber));
168         }
169         else
170         {
171             DateTime inputDate = DateTime.SpecifyKind(Extensions.NSDateToDateTime(DatePicker.Date), DateTimeKind.Utc).ToLocalTime();
172             taskManager.EditTask(new Task(TaskName.Text, TaskShortLable.Text, Cattegory, DatePicker.Date, NotificationSwitch.On,
173                 CalendarSwitch.On, durationNumber, currentTask.CalendarEventID), IndexOfEditTask);
174             EditingEnable = false;
175         }
176
177         this.NavigationController.PopViewController(true);
```

Zdroj: (Autor)

3.2.3 Vstup pro datum a času Tasku

Prvek DatePicker slouží jako vstup času a datumu pro konkrétní Task. DatePicker má opět nastavený identifikátor jako každý vstupní prvek. Dále je nastavena vlastnost Minimum Date a Mode. Mode určuje, zda je DatePicker vstupem pouze pro čas, datum nebo obojí. V kódu se poté při získáni času určí lokalizace časového pásma a typ UTC. Z tohoto vstupu je získána proměnná typu NSDate. Tento typ je používán společností Apple v nativním programovacím jazyku Swift. S tímto typem proměnné se v C# špatně pracuje, kvůli absenci určitých metod. Například metoda, která převádí čas do řetězce v určitém formátu. Proto se při práci s tímto typem proměnné využívá pomocná statická třída Extensions, která převádí typ NSDate na typ DateTime (Ukázka kódu 3). Tato třída převádí časové datové typy NSDate a DateTime pomocí dvou metod.

Ukázka kódu 3: Třída Extensions

```
11      public static class Extensions
12      {
13          public static NSDate DateTimeToNSDate(this DateTime date)
14          {
15              if (date.Kind == DateTimeKind.Unspecified)
16                  date = DateTime.SpecifyKind(date, DateTimeKind.Local);
17              return (NSDate)date;
18          }
19
20          public static DateTime NSDateToDateTime(this NSDate date)
21          {
22              return ((DateTime)date).ToLocalTime();
23          }
24      }
```

Zdroj: (Autor)

3.2.4 Výběr kategorie

V nástroji Xamarin.iOS je nastavený pouze identifikátor prvku PickerView. PickerView slouží jako vstup pro kategorii Tasku. Abychom ho mohli v této aplikaci použít, byla vytvořena třída CattegoryViewModel, která dědí ze třídy UIPickerView a upravuje přepisovatelné metody jejího rodiče. Třída je navrhnuta tak, aby se v konstruktoru předávala generická kolekce cattegoryList typu řetězec, ve které jsou načteny samotné kategorie. V třídě jsou přepsány následující metody. Metoda vracející počet položek v PickerView, což je to samé, co počet položek v cattegoryList. Dále metoda GetComponentCount určující, kolik má PickerView komponentů a metoda GetTitle, která vrací název položky z cattegoryList. Poslední funkcí, která je v této třídě upravena je metoda Selected. Když uživatel změní vybranou položku, tak se pomocí EventHandleru získá index této položky a metoda Selected nastaví vybranou položku do vlastnosti SelectedCattegory. Vlastnost SelectedCattegory slouží k získání vybrané položky při ukládání vstupů.

(Ukázka kódu 4) V obrazovce TaskDetailController se pomocí identifikátoru přiřadí upravená třída k elementu PickerView, který byl vytvořenen v Xamarin.iOS (řádek 60). Taktéž se v obrazovce TaskDetailController vytváří kolekce kategorií. Při uživatelské změně se pomocí EventHandleru zavolá funkce Selected (řádek 62). Tato vybraná kategorie se poté přiřadí do vlastnosti Cattegory třídy TaskDetailController. Při vybrání položky „Other“ se vytvoří AlertView, který slouží k vytvoření vlastní kategorie pro daný úkol (řádek 68). Této třídě UIAlertView se nastaví požadované vlastnosti, jako je například název, popisek a styl. Poté se samotný AlertView zobrazí pomocí metody Show (řádek 80).

Při editaci Tasku se musí nastavit položka, která je předávaná z editovaného Tasku. To se provádí pomocí získaného indexu položky, který je nastaven v editovaném Tasku. Pomocí tohoto indexu a metody Select se nastaví položka, která má být vybrána.

Ukázka kódu 4: Použití třídy CattegoryViewModel

```
58          var cattegoryViewModel = new CattegoryViewModel(CattegoryList);
59
60          CattegoryPicker.Model = cattegoryViewModel;
61
62          cattegoryViewModel.CattegoryChange += (sender, e) =>
63          {
64              Cattegory = cattegoryViewModel.SelectedCattegory;
65              Console.WriteLine(Cattegory);
66              if (Cattegory == "Other")
67              {
68                  var AlertInput = new UIAlertView();
69                  AlertInput.Title = "Own cattegory";
70                  AlertInput.AddButton("OK");
71                  AlertInput.Message = "Please enter your cattegory";
72                  AlertInput.AlertViewStyle = UIAlertViewStyle.PlainTextInput;
73                  AlertInput.Clicked += (object s, UIButtonEventArgs ev) =>
74                  {
75                      if (ev.ButtonIndex == 0)
76                      {
77                          Cattegory = AlertInput.GetTextField(0).Text;
78                      }
79                  };
80                  AlertInput.Show();
81              }
82          };
83
```

Zdroj: (Autor)

3.2.5 Vstup pro text

Prvek TextField slouží jako vstup pro text nebo číslo. V nástroji Xamarin.iOS je pro něj nastaven identifikátor a formát textu. Tento vstup je v aplikaci použit pro získání názvu Tasku a délky Tasku. Výstup z tohoto prvku je datový typ řetězec, proto při získávání doby trvání Tasku musí docházet k ověření tohoto řetězce, zda se jedná o číslo nebo text. Toto je ověřené pomocí metody TryParse (ukázka kódu 5). Pokud výstup není možné převést na integer pomocí této metody, tak se uživateli zobrazí AlertView, který ho informuje o tom, že vstup zadal ve špatném formátu. Toto všechno se děje při volání metody Save_TouchUpInside ve třídě

TaskDetailController. Proto je uveden na posledním řádku těla podmínky příkaz return (řádek 140), který zastaví průběh ukládání při zadání chybných dat.

Ukázka kódu 5: Kontrola vstupu duration

```
133             bool durationNumberBool = Int32.TryParse(Duration.Text, out durationNumber);
134             if (!durationNumberBool)
135             {
136                 var okCancelAlertController = UIAlertController.Create("Duration error", "Value of duration is not number",
137                     UIAlertControllerStyle.Alert);
138                 okCancelAlertController.AddAction(UIAlertAction.Create("OK", UIAlertActionStyle.Default, null));
139                 PresentViewController(okCancelAlertController, true, null);
140                 return;
141             }
```

Zdroj: (Autor)

3.2.6 Vstup pro krátký popisek

Jako vstup pro krátký popisek je použit prvek TextView. Tento prvek implicitně slouží k zobrazení delšího textu, ale je to jediný způsob, jak získat delší text. Proto je použit k získání krátkého popisku Tasku. Aby uživatel intuitivně poznal že se jedná o jeden ze vstupů, je mu v kódu přidán okraj, který je běžný pro vstup TextField (Ukázka kódu 6).

Ukázka kódu 6: Nastavení vzhledu TextView

```
48                 TaskShortLable.Layer.BorderColor = UIColor.LightGray.CGColor;
49                 TaskShortLable.Layer.CornerRadius = 5.0f;
50                 TaskShortLable.Layer.BorderWidth = 0.25f;
```

Zdroj: (Autor)

3.2.7 Zapnutí a vypnutí notifikace/přidání do kalenáře

Switch slouží jako vstup pro datový typ bool. Tento vstup je použit u nastavení notifikace a přidání úkolu do kalendáře. U tohoto prvku je použita akorát vlastnost identifikátoru.

3.2.8 Zobrazení tabulky Tasků

Pro zobrazení tabulky Tasků byl použit prvek TableView, kterému byly nastaveny vlastnosti velikost a identifikátor tabulky. Dále bylo nastaveno rozložení prvků v buňce tabulky. Jedná se například o vzdálenosti prvků od okraje nebo od ostatních prvků. Veškeré informace zobrazené v buňce tabulky jsou zobrazeny pomocí prvků Label. Vlastnost identifikátor musí být nastavena u samotné buňky i u každého prvku Label v buňce.

Aby se v prvcích buňky zobrazovali hodnoty Tasků, je nutné vytvořit třídu TaskCell, která dědí ze třídy UITableViewCell. Třída TaskCell obsahuje metodu UpdateCell (Ukázka kódu 7). Tato metoda je volána pokaždé, když je zobrazena tabulka a nastavuje prvkům Label hodnoty samotného Tasku (řádek 15-29). Tento Task je předán jako parametr metody UpdateCell. V buňce se pak zobrazují parametry: název, čas ve zkrácené formě a kategorie Tasku. Také se v této metodě porovnává čas nastavený pro začátek Tasku s aktuálním časem. Pokud začátek Tasku proběhl, zobrazí se uživateli několik vykřičníků, které indikují že Task již probíhá nebo měl proběhnout (řádek 25).

Ukázka kódu 7: Metoda UpdateCell

```
13      internal void UpdateCell(Task task)
14      {
15          NameLabel.Text = task.Name;
16
17          CattegoryLabel.Text = task.Cattegory;
18
19          DatetimeLabel.Text = String.Format("{0} - {1} Hours", NSDateFormatter.ToLocalizedString(task.DateAndTime, NSDateFormatterStyle.Medium,
20              NSDateFormatterStyle.Short), task.Duration);
21
22
23          int reminder = DateTime.Compare(DateTime.Now, Extensions.NSDateToDateTime(task.DateAndTime));
24
25          if (0 <= reminder)
26          {
27              ReminderLabel.Text = "!!!";
28          }
29          else
30          {
31              ReminderLabel.Text = "";
32          }
33
34      }
```

Zdroj: (Autor)

Samotná tabulka zobrazuje data pomocí přiřazeného zdroje. Pro tento zdroj je vytvořena třída TaskTVS, která dědí ze třídy UITableViewSource. Třída TaskTVS dostává v konstruktoru kolekci samotných Tasků. Ve třídě jsou upraveny přepisovatelné metody rodičovské třídy. Jmenovitě metoda GetCell, která vrací buňku, do samotné tabulky, pomocí metody UpdateCell z třídy TaskCell. Dále metoda RowInSection, která vrací počet řádků v sekci. A metoda RowSelected, která vrací počet vybraných řádků.

Ukázka kódu 8: Metoda EditActionsRow

```
50      public override UITableViewRowAction[] EditActionsForRow(UITableView tableView, NSIndexPath indexPath)
51      {
52          var action = UITableViewRowAction.Create(UITableViewRowActionStyle.Default, "Done", (arg1, arg2) =>
53          {
54              var cell = tableView.CellAt(arg2);
55              TaskManager taskManager = new TaskManager();
56              taskManager.DeleteTaskOnIndex(indexPath.Row);
57              tasks.RemoveAt(indexPath.Row);
58              tableView.DeleteRows(new NSIndexPath[] { indexPath }, UITableViewRowAnimation.Fade);
59
60              tableView.ReloadData();
61          });
62          return new UITableViewRowAction[] { action };
63      }
```

Zdroj: (Autor)

Dále je ve třídě TaskTVS vytvořená akce přetažení neboli „Swipe“ (Ukázka kódu 8). Tato akce slouží k dokončení Tasku nebo jeho odstranění z kolekce. Akce Swipe je provedena

pomocí metody EditActionForRow (řádek 50). V této metodě se vytváří akce, která má popisek „Done“. Pomocí indexu, kde se akce odehrála je zjištěno, jaký Task má být odstraněn. Task je odstraněn z kolekce pomocí třídy TaskManager a metody DeleteTaskOnIndex. Následně je buňka odstraněna ze samotné tabulky (řádek 58) a nakonec se zavolá metoda ReloadData, která data znovu načte.

3.2.9 ViewController

UIViewController je název prvku pro samotnou obrazovku. Typů těchto obrazovek je několik, jako například UIView, který pouze zobrazuje informace a nemůže zpracovávat žádná data nebo UITableViewController, který je specializovaný pro práci s tabulkou.

Aby se mohli zpracovávat uživatelské akce, je nutné nastavení vlastnosti identifikátoru a třídy samotné obrazovky. Pro každou z těchto obrazovek je následně automaticky vygenerována třída v kódu, kde se již upravuje chování obrazovky. U obrazovky ViewController je vytvořená třída implicitně při vytváření projektu, a to třída ViewController. V této třídě je definována metoda ViewWillAppear, která oznamuje třídě ViewControlleru, že obrazovka této třídy bude zobrazena (Ukázka kódu 9). V metodě ViewWillAppear je vytvořena instance třídy TaskManager, která poté předává vlastnost TaskList. Tato vlastnost, která vrací kolekci všech Tasků, je poté předána do konstruktoru třídy TaskTVS. Tato třída je poté přiřazena prvku UITableView, který byl vytvořený v Xamarin.iOS. Nejdříve je ale prvku UITableView a třídě TaskManager přiřazena hodnota null, a to kvůli aktualizaci a znovu načtení kolekce Tasků do tabulky.

Ukázka kódu 9: Metoda ViewWillAppear

```
16      public override void ViewWillAppear(bool animated)
17      {
18          base.ViewWillAppear(animated);
19
20          TaskTableView.Source = null;
21          TaskManager taskManager = null;
22
23          taskManager = new TaskManager();
24          TaskTableView.Source = new TaskTVS(new ObservableCollection<Task>(taskManager.TaskList));
25      }
```

Zdroj: (Autor)

3.2.10 TaskDetailController

Tato obrazovka, jako již předešlá, má nastavený identifikátor a třídu. Proto aby se mohla vytvořit třída v kódu, která bude zpracovávat akce uživatele. Její rodič je tedy také třída UIViewController. Třída TaskDetailController obsahuje několik metod a vlastností. Do vlastnosti Cattegory se ukládá vybraná kategorie Tasku. Tato vlastnost je vytvořená proto aby se hodnota mohla předat z jedné metody do druhé. Vlastnost currentTask napomáhá při editaci

jednotlivých Tasků. V této vlastnosti je Task, který má být editován a jeho hodnoty jsou nastaveny jako výchozí. Taktéž k editaci Tasků napomáhá vlastnost IndexOfEditTask. Vlastnost EditingEnable rozhoduje o tom, zda se jedná o editaci stávajícího nebo vytváření nového Tasku.

První metodou je ViewWillAppear. Tato metoda je volána před zobrazením obrazovky (Ukázka kódu 10). To znamená, že se v této metodě nastavují prvky uživatelského rozhraní, které jsou zobrazeny na obrazovce TaskDetailController. V této metodě se nastaví časová zóna prvku UIDatePicker (řádek 45 a 46). Dále se nastaví UITextView společně s prvkem UIPickerView, který byl již rozebrán výše. Následně se podle vlastnosti EditingEnable rozhodne, zda se jedná o vytváření nového či editování stávajícího Tasku (řádek 82). Pokud se jedná o editaci, následuje nastavení všech vlastností editovaného Tasku na výchozí, pro uživatelské rozhraní. Také je zde podmínka, která se stará o nastavení kategorie při editaci Tasku. Tato podmínka rozhoduje, jestli se jedná o uživatelem vytvořenou kategorii (řádek 89).

Ukázka kódu 10: Metoda ViewWillApear - TaskDetailController

```
41      public override void ViewWillAppear(bool animated)
42      {
43          base.ViewWillAppear(animated);
44
45          DatePicker.Locale = NSLocale.CurrentLocale;
46          DatePicker.TimeZone = NSTimeZone.LocalTimeZone;
47
48          TaskShortLable.Layer.BorderColor = UIColor.LightGray.CGColor;
49          TaskShortLable.Layer.CornerRadius = 5.0f;
50          TaskShortLable.Layer.BorderWidth = 0.25f;
51
52
53          var CattegoryList = new List<String>...;
57          var cattegoryViewModel = new CattegoryViewModel(CattegoryList);
58          CattegoryPicker.Model = cattegoryViewModel;
59
60          cattegoryViewModel.CattegoryChange += (sender, e) =>...
81
82          if (EditingEnable)
83          {
84              TaskName.Text = currentTask.Name;
85              TaskShortLable.Text = currentTask.ShortLabel;
86              NotificationSwitch.On = currentTask.Notification;
87              CalendarSwitch.On = currentTask.CalenderEvent;
88              Duration.Text = currentTask.Duration.ToString();
89              if (CattegoryList.Contains(currentTask.Cattegory))
90              {
91                  CattegoryPicker.Select(CattegoryList.IndexOf(currentTask.Cattegory), 0, true);
92              }
93              else
94              {
95                  CattegoryPicker.Select(CattegoryList.Count - 1, 0, true);
96                  Cattegory = currentTask.Cattegory;
97              }
98
99              DatePicker.Date = currentTask.DateAndTime;
```

Zdroj: (Autor)

Další metoda ve třídě TaskDetailController je volána při stisknutí již vytvořeného Tasku v tabulce obrazovky ViewController (Ukázka kódu 11). Tato metoda dostává parametr, odkud se data předávají, Task, který se bude editovat a index Tasku v tabulce. Tato metoda nastaví veškeré vlastnosti, které jsou potřeba při editaci Tasku.

Ukázka kódu 11: Metoda SetTask

```
103     public void SetTask(ViewController d, Task task, int index)
104     {
105         Delegate = d;
106         currentTask = task;
107         EditingEnable = true;
108         IndexOfEditTask = index;
109     }
```

Zdroj: (Autor)

Další metoda Save_TouchUpInside se volá při stisknutí tlačítka Save. Tato metoda slouží k ukládání Tasku, která již byla rozebrána v textu výše, v podkapitole Přechody mezi obrazovkami.

3.3 Část ViewModel

Část ViewModel obsahuje šest tříd. Třídu Task, která je datovým typem jednotlivých úkolů, třídu AppDelegate, která zpracovává speciální stavy aplikace a třídu App, která pomáhá při propojení s Apple kalendářem. Dále třídu UNCD, která určuje, co se bude dít s notifikacemi při určitém stavu, třídu Extension, která byla již popsána výše, v podkapitole Vstup pro datum a čas, a třídu TaskManager, která zpracovává data a komunikuje s vrstvou Model. Části vrstvy ViewModel jsou popsány níže v podkapitolách

3.3.1 Třída Task

(Ukázka kódu 12) Třída Task je vytvořený datový typ pro samotné aktivity nebo úkoly a obsahuje jeho veškeré parametry. Počínaje hořejškem se jedná o název, krátký popisek, kategorii, čas a datum začátku, nastavení notifikace, nastavení propojení s kalendářem, trvání aktivity a ID. Pod toto ID se ukládá event v Apple kalendáři. Konstruktor má jedno rozšíření, a to právě ID pro event, které nemusí být nastaveno, pokud Task nebude přidán do Apple

Ukázka kódu 12: Třída Task

```
11  public class Task
12  {
13      public String Name { get; set; }
14      public String ShortLabel { get; set; }
15      public String Cattegory { get; set; }
16      public NSDate DateAndTime { get; set; }
17      public bool Notification { get; set; }
18      public bool CalenderEvent { get; set; }
19      public int Duration { get; set; }
20      public string CalendarEventID { get; set; }
21
22      public Task(string name, string shortlabel, string cattegory, NSDate dateandtime, bool notification, bool calendarEvent, int duration)...
33
34      public Task(string name, string shortlabel, string cattegory, NSDate dateandtime, bool notification, bool calendarEvent, int duration, string calendarEventID)...
45  }
```

Zdroj: (Autor)

kalendáře.

3.3.2 Třída TaskManager

Třída TaskManager pracuje se samotnými Tasky. Stará se o přidávání, mazání a editaci Tasků. Taktéž spravuje notifikace a propojení s Apple kalendářem. Tyto obě části si probereme zvláště v podkapitolách. Třída TaskManager obsahuje generickou kolekci Tasků s názvem tasks a instanci třídy DataStorage. Dále obsahuje vlastnosti: počet tasků s názvem Count, indexer přistupující do kolekce tasks a kolekci Tasků s názvem TaskList. V konstruktoru této třídy se vytváří instance třídy DataStorage a volá se metoda GetTasks třídy DataStorage, která vrací kolekci Tasků ze souboru. Tato kolekce se přiřadí kolekci tasks (Ukázka kódu 13).

Ukázka kódu 13: Konstruktor TaskManager

```
18      public TaskManager()
19      {
20          dataStorage = new DataStorage();
21          tasks = dataStorage.GetTasks();
22      }
```

Zdroj: (Autor)

Metoda AddTask (Ukázka kódu 14) přidává Task do kolekce a kontroluje nastavení Tasku. Pokud uživatel nastavil, že chce přidat Task do Apple kalendáře, je zavolána metoda CreateCalendarEvent. Tato metoda vrací ID Eventu v kalendáři, který se poté přiřadí do vlastnosti CalendarEventID samotného Tasku (řádek 43). Následně se přidá Task do kolekce, a poté se kolekce seřadí od nejstaršího po nejnovější pomocí třídy Extensions, podle datumu a času. To proto aby se Tasky zobrazovali v tabulce postupně. Poté se zkontroluje, zda chce uživatel přidat notifikaci Tasku. Pokud ano, volá se metoda AddNewNotification (řádek 51). Nakonec se volá metoda SaveTasks, třídy DataStorage, která ukládá kolekci.

Ukázka kódu 14: Metoda AddTask

```
39      public void AddTask(Task task)
40      {
41          if (task.CalenderEvent)
42          {
43              task.CalendarEventID = CreateCalendarEvent(task);
44          }
45
46          tasks.Add(task);
47          tasks.Sort((x, y) => DateTime.Compare(Extensions.NSDateToDateTime(x.DateAndTime), Extensions.NSDateToDateTime(y.DateAndTime)));
48
49          if (task.Notification)
50          {
51              AddNewNotification(task);
52          }
53
54          dataStorage.SaveTasks(tasks);
55      }
```

Zdroj: (Autor)

Metoda EditTask (Ukázka kódu 15) edituje Task. Tato metoda je volána ze třídy TaskDetailController. Parametry metody jsou Task po editaci a index původního Tasku v kolekci. Po zavolání metody se zkontroluje nastavení přidání do kalendáře (řádek 59). Pokud se tato hodnota změnila, pomocí podmínky se zjistí, jestli chce uživatel přidat Task do kalendáře. Pokud ano, zavolá se metoda CreateCalendarEvent. Pokud ne volá se metoda RemoveCalendarEvent a následně se odstraní ID eventu v Tasku. Tato podmínka funguje obdobně pro nastavení notifikace. Následně se zajistí, zda byl Task nějak změněný. Pokud ano, volá se funkce UpdateCalenderEvent a UpdateNotification. Poté se přiřadí již editovaný Task pomocí indexu na původní pozici v kolekci. Na konec se kolekce tasks seřadí a pomocí třídy DataStrorage uloží.

Ukázka kódu 15: Ověření změny nastavení přidání do kalendáře

```
59          if (tasks[index].CalenderEvent != task.CalenderEvent)
60          {
61              if (task.CalenderEvent)
62              {
63                  task.CalendarEventID = CreateCalendarEvent(task);
64              }
65              else
66              {
67                  RemoveCalendarEvent(task);
68                  task.CalendarEventID = "";
69              }
70          }
```

Zdroj: (Autor)

Metoda DeleteTaskOnIndex (Ukázka kódu 16) maže Task. Metoda má jako parametr index, na kterém má Task v kolekci smazat. Než tak ale metoda provede, musí smazat notifikaci a také event z Apple kalendáře. Nejdříve se ale zkontroluje, zda má Task notifikaci nebo jestli je uložen v kalendáři, aby nedocházelo k výjimce při mazání neexistující notifikace nebo Tasku v kalendáři.

Ukázka kódu 16: Metoda DeleteTaskOnIndex

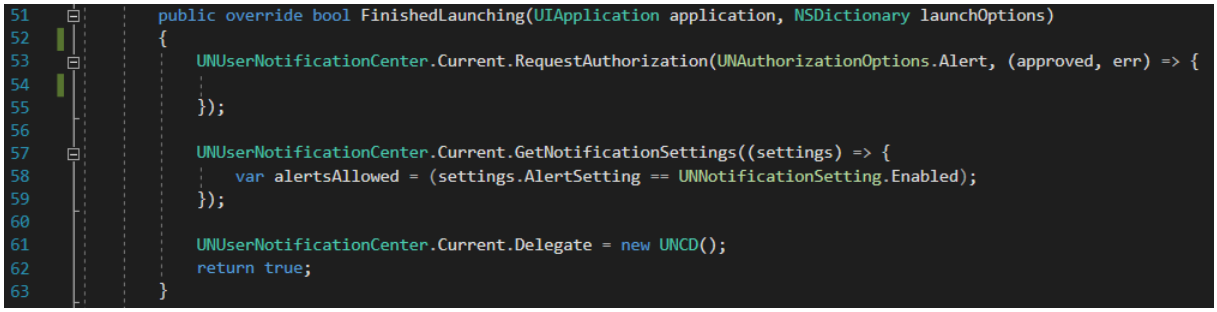
```
105         public void DeleteTaskOnIndex(int index)
106         {
107             if (tasks[index].Notification)
108             {
109                 RemoveNotification(tasks[index]);
110             }
111             if (tasks[index].CalenderEvent)
112             {
113                 RemoveCalendarEvent(tasks[index]);
114             }
115             tasks.RemoveAt(index);
116             dataStorage.SaveTasks(tasks);
117         }
```

Zdroj: (Autor)

3.3.3 Notifikace

Notifikace v aplikaci spravuje třída TaskManager. Pro správu notifikace jsou vytvořeny tři metody a jedna třída. Před nastavením notifikace se musí aplikace uživatele dotázat, zda může aplikace nastavovat lokální notifikace (Ukázka kódu 17). To se děje ve třídě AppDelegate. V této třídě se přepíše metoda FinishedLaunching, která je volaná po spuštění aplikace. V této metodě se aplikace dotáže na povolení o lokální notifikace (řádek 53). Metoda na řádku 57 kontroluje, zda se nezměnilo nastavení povolení notifikace.

Ukázka kódu 17: Metoda FinishedLaunching

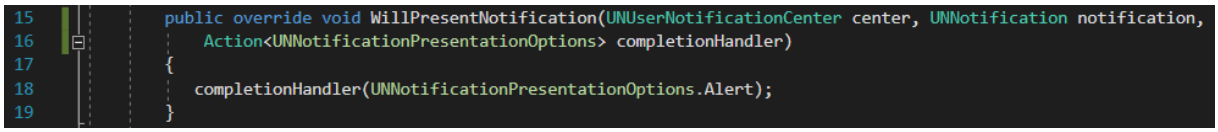


```
public override bool FinishedLaunching(UIApplication application, NSDictionary launchOptions)
{
    UNUserNotificationCenter.Current.RequestAuthorization(UNAuthorizationOptions.Alert, (approved, err) => {

    });

    UNUserNotificationCenter.Current.GetNotificationSettings((settings) => {
        var alertsAllowed = (settings.AlertSetting == UNNotificationSetting.Enabled);
    });

    UNUserNotificationCenter.Current.Delegate = new UNCD();
    return true;
}
```

Zdroj: (Autor)

Třída UNCD popisuje, co se má stát s notifikací, pokud je aplikace v popředí (Ukázka kódu 18). To znamená, že je aplikace spuštěná a zobrazená na displeji. Tato třída dědí z rodiče UNUserNotificationCenterDelegate. V té je přepsána metoda WillPresentNotification a říká, že se notifikace má zobrazit stejně jako kdyby aplikace nebyla v popředí.

Ukázka kódu 18: Metoda WillPresentNotification



```
public override void WillPresentNotification(UNUserNotificationCenter center, UNNotification notification,
    Action<UNNotificationPresentationOptions> completionHandler)
{
    completionHandler(UNNotificationPresentationOptions.Alert);
}
```

Zdroj: (Autor)

Metoda AddNewNotification (Ukázka kódu 19) přidává notifikaci. Tato metoda dostane v parametru Task, pro který má být notifikace přidána. Vytvoří se nová instance třídy UNMutableNotificationContent. Tato třída obsahuje upravitelné vlastnosti pro obsah notifikace, které jsou postupně nastaveny. Jako Title je nastaveno název Tasku a jeho kategorie. Poté je v Body nastaven krátký popisek Tasku. Následně je nastaven Badge na 0. Badge je červené kolečko, které ukazuje počet notifikací u ikony aplikace. Tato aplikace tuto funkci nepoužívá, proto je nastaven na 0. Následně se musí vytvořit datový typ času NSDateComponents. Tento čas nelze převést pomocí předdefinované metody, proto je předváděn postupně po jednotlivých časových údajích. Poté je nastaven trigger (řádek 150), který určuje způsob odpočtu notifikace. UNCalendarNotificationTrigger je způsob, který nastaví notifikaci na určitý čas a datum. Poté je vytvořeno ID notifikace. ID je vytvořeno z názvu a času notifikace. Poté se všechny tyto nastavené hodnoty nastaví, jako parametry metody FromIndetifier třídy UNNotificationRequest. Nakonec se tato třída přidá pomocí metody AddNotificationRequest do notifikačního centra zařízení.

Ukázka kódu 17: Metoda AddNewNotification

```
135     private void AddNewNotification(Task task)
136     {
137         var content = new UNMutableNotificationContent();
138         content.Title = task.Name + " - " + task.Cattegory;
139         content.Body = task.ShortLabel;
140         content.Badge = 0;
141
142         NSDateComponents dateNotifiComp = new NSDateComponents();
143         DateTime dateNotifi = Extensions.NSDateToDateTime(task.DateAndTime);
144         dateNotifiComp.Minute = dateNotifi.Minute;
145         dateNotifiComp.Hour = dateNotifi.Hour;
146         dateNotifiComp.Day = dateNotifi.Day;
147         dateNotifiComp.Month = dateNotifi.Month;
148         dateNotifiComp.Year = dateNotifi.Year;
149
150         var trigger = UNCalendarNotificationTrigger.CreateTrigger(dateNotifiComp, false);
151         dateNotifi = dateNotifi.AddSeconds(-dateNotifi.Second);
152         var requestID = task.Name + dateNotifi.ToString();
153         var request = UNNotificationRequest.FromIdentifier(requestID, content, trigger);
154         Console.WriteLine(requestID);
155
156         UNUserNotificationCenter.Current.AddNotificationRequest(request, (err) => {
157             if (err != null)
158             {
159                 System.Console.WriteLine(err);
160             }
161         });
162     }
```

Zdroj: (Autor)

Metoda RemoveNotification (Ukázka kódu 20) odstraňuje notifikaci. Metoda dostane jako parametr Task, u které ho má být odstraněna notifikace. Nejdříve se vytvoří ID notifikace, které se skládá z názvu, času a datumu. Dále se pomocí třídy UNUserNotificationCenter notifikace odstraní. Nejdříve se odstraní notifikace čekající na spuštění pomocí metody RemovePendingNotificationRequests (řádek 174) a dále se odstraní notifikace, již doručené uživateli pomocí metody RemoveDeliveredNotifications (řádek 175).

Ukázka kódu 18: Metoda RemoveNotification

```
171     private void RemoveNotification(Task task)
172     {
173         var requestID = new string[] { task.Name + Extensions.NSDateToDateTime(task.DateAndTime).ToString() };
174         UNUserNotificationCenter.Current.RemovePendingNotificationRequests(requestID);
175         UNUserNotificationCenter.Current.RemoveDeliveredNotifications(requestID);
176     }
```

Zdroj: (Autor)

Metoda UpdateNotification aktualizuje notifikaci, pokud dojde ke změně Tasku. Tato metoda dostává dva parametry, původní Task a již editovaný Task. V této metodě se postupně volají dvě předešlé metody. Nejdříve se volá metoda RemoveNotification s parametrem původního Tasku a poté se volá metoda AddNewNotification s parametrem již editovaného Tasku.

3.3.4 Propojení s Apple kalendářem

Pro veškerou práci s eventy je potřeba přidat knihovnu EventKit. Pro propojení aplikace s Apple kalendářem se používá několik metod a jedna třída. Pro práci s eventy v kalendáři je používána třída EventStore. Tato třída umožňuje přístup ke kalendáři. V této aplikaci je vytvořena třída App (Ukázka kódu 21), která vytváří jednu instanci třídy EventStore a staticky ji zpřístupňuje. Třída EventStore funguje jako databázový engine, a proto by měla být dlouhodobá. Třída App poskytuje EventStore jako vlastnost. Pro vytvoření třídy App byl použit doporučený vzor z dokumentace Xamarinu.

Ukázka kódu 21: Třída App

```
11      public class App
12      {
13          public static App CurrentApp...
17          private static App currentApp;
18          public EKEventStore EventStore...
22          protected EKEventStore eventStore;
23
24          static App()
25          {
26              currentApp = new App();
27          }
28
29          protected App()
30          {
31              eventStore = new EKEventStore();
32          }
33
34      }
```

Zdroj: (Autor)

Před přidáním Tasku do Apple kalendáře, se musí aplikace dotázat, zda může upravovat data Apple kalendáře. Pro toto dotázání musí být v aplikaci uvedeny popisy užití (Ukázka kódu 22). Tyto popisy užití jsou nastaveny v souboru Info.plist, který musí být editován z textového editoru. V tomto souboru jsou přidány následující řádky.

Ukázka kódu 22: Úpráva dokumentu Info.plist

```
42      <key>CFBundleName</key>
43      <string>Calendars</string>
44      <key>NSCalendarsUsageDescription</key>
45      <string>Calendars App needs Calendar Access</string>
46      <key>NSRemindersUsageDescription</key>
47      <string>Calendars App needs Reminder Access</string>
48      <key>NSLocationAlwaysUsageDescription</key>
49      <string>Calendars App needs Location Access</string>
50      <key>NSLocationWhenInUseUsageDescription</key>
51      <string>Calendars App needs location access</string>
52      <key>XSAppIconAssets</key>
53      <string>Assets.xcassets/AppIcon.appiconset</string>
```

Zdroj: (Autor)

Samotné dotazování probíhá při ukládání Tasku, v metodě Save_TouchUpInside (Ukázka kódu 23). Dotázání uživatele se spustí, pokud chce uživatel přidat Task do kalendáře. Dotázání proběhne pomocí třídy App a její vlastnosti, která poskytuje instanci třídy EventStore. V třídě EventStore dochází k volání metody RequestAccess. Pokud uživatel odmítne přístup k datům kalendáře, zobrazí se mu AlertView informující ho o tom, že byl přístup zamítnut.

Ukázka kódu 23: Dotázání uživatele na přístup ke kalendáři

```
115     if (CalendarSwitch.On)
116     {
117         App.CurrentApp.EventStore.RequestAccess(EKEntityType.Event,
118         (bool granted, NSError e) =>
119         {
120             if (granted)
121             {
122 
123             }
124             else
125             {
126                 var okCancelAlertController = UIAlertController.Create("Access Denied", "User Denied Access", UIAlertControllerStyle.Alert);
127                 okCancelAlertController.AddAction(UIAlertAction.Create("OK", UIAlertActionStyle.Default, null));
128                 PresentViewController(okCancelAlertController, true, null);
129             }
130         });
131     }
```

Zdroj: (Autor)

Metoda CreateCalendarEvent (Ukázka kódu 24) vytváří záznam Tasku v kalendáři. Záznamy v kalendáři se nazývají eventy. Nejdříve se vytvoří třída EKEvent, která je přidružená k instanci EventStoru ze třídy App (řádek 188). Dále se přiřadí čas a datum začátku. Poté se pomocí třídy Extension přidá doba trvání Tasku k času a datumu začátku, čímž vznikne datum a čas konce eventu. Nakonec se přiřadí název společně s krátkým popiskem a proběhne uložení eventu. Metoda CreateCalendarEvent nakonec vrátí vygenerované ID eventu pro jeho další správu.

Ukázka kódu 24: Metoda CreateCalendarEvent

```
186     private string CreateCalendarEvent(Task task)
187     {
188         EKEvent ekEvent = EKEvent.FromStore(App.CurrentApp.EventStore);
189         ekEvent.StartDate = task.DateAndTime;
190         ekEvent.EndDate = Extensions.DateTimeToNSDate(Extensions.NSDateToDateTime(task.DateAndTime).AddHours(task.Duration));
191         ekEvent.Title = task.Name;
192         ekEvent.Notes = task.ShortLabel;
193         ekEvent.Calendar = App.CurrentApp.EventStore.DefaultCalendarForNewEvents;
194 
195         App.CurrentApp.EventStore.SaveEvent(ekEvent, EKSpan.ThisEvent, true, out NSError error);
196         if (error != null)
197         {
198             Console.WriteLine(error);
199             return null;
200         }
201         else
202         {
203             Console.WriteLine("event was added into calendar");
204             return ekEvent.EventIdentifier;
205         }
206     }
```

Zdroj: (Autor)

Metoda RemoveCalendarEvent (Ukázka kódu 25) odstraňuje Task z kalendáře. Tato metoda dostává jako parametr Task, který má být odstraněn z kalendáře. Nejdříve se vytvoří třída EKEvent pomocí třídy EventStore a metody EventFromIdentifier (řádek 179). Tato metoda má parametr ID eventu, ke kterému je potřeba přistupovat. V případě aplikace je tento identifikátor uložen jako vlastnost Tasku, s názvem CalendaerEventID. Dále se zavolá metoda RemoveEvent, kde je jako parametr předán aplikací vytvořený event pomocí identifikátoru. Tato metoda odstraní event z kalendáře.

Ukázka kódu 19: Metoda RemoveCalendarEvent

```
177     private void RemoveCalendarEvent(Task task)
178     {
179         EKEvent eKEvent = App.CurrentApp.EventStore.EventFromIdentifier(task.CalendarEventID);
180         App.CurrentApp.EventStore.RemoveEvent(eKEvent, EKSpan.ThisEvent, true, out NSError err);
181         if (err != null)
182         {
183             Console.WriteLine("error on removing event from calendar");
184         }
185     }
```

Zdroj: (Autor)

Metoda UpdateCalendarEvent aktualizuje event po změně Tasku. Tato metoda funguje obdobně jako metoda UpdateNotification s tím rozdílem, že obsahuje pouze jediný parametr Task a vrací nově vytvořený identifikátor.

3.4 Část Model

Část Model pracuje pouze s jednou třídou. Tato třída je nazvána DataStorage, která ukládá nebo načítá data. S touto třídou komunikuje pouze jediná třída z vrstvy ViewModel, a to třída TaskManager. Třída TaskManager vytváří instanci třídy DataStorage a využívá její dvě metody. Jelikož data, která je potřeba uložit, nemají mezi sebou žádné složité vztahy, tak ukládá aplikace data do souboru. Pro práci se soubory je přidána knihovna Systém.IO. Pro ukládání dat do Apple zařízení existuje speciální složka. Tato složka se nazývá MyDocuments. V této složce má každá aplikace svoji podsložku. Složka MyDocuments je dostupná pomocí iTunes, proto je možné přesunout data z jednoho zařízení do druhého. Přístup do této složky se vytvoří v konstruktoru třídy DataStorage a následně se vytvoří i název souboru pro ukládání dat (Ukázka kódu 26).

Ukázka kódu 20: Konstruktor třídy DataStorage

```
17          public DataStorage()
18          {
19              dataFilePath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
20              filename = Path.Combine(dataFilePath, "data.csv");
21          }
```

Zdroj: (Autor)

Metoda SaveStorage (Ukázka kódu 27) ukládá data do souboru. Jako parametr dostává generickou kolekci Tasků, které má uložit. Ukládání probíhá pomocí StreamWriteru, který si sám hlídá, jestli soubor existuje. Pokud soubor neexistuje, sám ho vytvoří. Při novém zapisování StreamWriter vymaže celý obsah souboru a zapíše jej znovu. Následně je jenom určen formát řetězce, ve kterém se ukládají data. Formát řetězce odděluje každou vlastnost Tasku tildou.

Ukázka kódu 21: Metoda SaveTasks

```
public void SaveTasks(List<Task> tasks)
{
    string shotlabel;
    using (var sw = new StreamWriter(filename))
    {

        for (int i = 0; i < tasks.Count; i++)
        {
            shotlabel = tasks[i].ShortLabel.Replace("\n", "÷");
            shotlabel = shotlabel.Replace("\r", "÷");
            shotlabel = shotlabel.Replace("\t", "÷");
            Console.WriteLine(shotlabel);
            sw.WriteLine(string.Format("{0}~{1}~{2}~{3}~{4}~{5}~{6}~{7}", tasks[i].Name, shotlabel, tasks[i].Cattegory,
                Extensions.NSDateToDateTime(tasks[i].DateAndTime).ToString("MM/dd/yyyy HH:mm"), tasks[i].Notification.ToString(),
                tasks[i].CalenderEvent.ToString(), tasks[i].Duration, tasks[i].CalendarEventID));
        }
    }
}
```

Zdroj: (Autor)

Metoda GetTasks (Ukázka kódu 28) načítá data ze souboru. Tato metoda vrací generickou kolekci Tasků. Aby nedocházelo k chybám před čtením ze souboru, ověří se, zda existuje daný soubor. Pokud ne, vrátí se prázdná kolekce. Pokud soubor existuje, tak se pomocí StreamReaderu čte řádek po řádku. Každý řádek se separuje pomocí metody Split a následně se z tohoto separovaného řádku vytváří instance třídy Task, která se přidá do kolekce.

Ukázka kódu 22: Metoda GetTasks

```
public List<Task> GetTasks()
{
    List<Task> output = new List<Task>();
    string line = "";
    //File.Delete(filename);
    if (File.Exists(filename))
    {
        using (StreamReader sr = new StreamReader(filename))
        {
            while ((line = sr.ReadLine()) != null)
            {
                string[] separateLine = line.Split(new char[] {'~'});
                output.Add(new Task(separateLine[0], separateLine[1].Replace("÷", "\n"), separateLine[2],
                    Extensions.DateTimeToNSDate(DateTime.ParseExact((separateLine[3]), "MM/dd/yyyy HH:mm", null)),
                    bool.Parse(separateLine[4]), bool.Parse(separateLine[5]), int.Parse(separateLine[6]), separateLine[7]));
                Console.WriteLine("Loading from file" + separateLine[7]);
            }
        }
    }
    return output;
}
```

Zdroj: (Autor)

4 Závěr

V teoretické práci je popsán vývoj a návrh aplikace. Jsou zde uvedeny i jiné platformy mimo iOS, pro které je možno vyvíjet. Hlavně je však v této části popsán vývoj aplikace pro iOS zařízení. Dále jsou popsány vrstvy architektury iOS aplikací, jelikož jejich znalost napomáhá při vývoji aplikace. Okrajově jsem zmínil pojem cross-platformní vývoj. Popsal jsem vývojovou platformu Xamarin a možnosti vývoje uživatelského rozhraní pomocí této platformy, od této volby se odvíjí samotný vývoj aplikace. Na závěr je v teoretické části popsán architektonický model MV(X) a jeho různé varianty. A také je zde popsán jeden z nejnovějších architektonických modelů VIPER.

Výstupem praktické části je aplikace, která má plnit funkci jednoduchého TO-DO listu. V aplikaci je možno vytvořit úkol, kterému lze nastavit různé parametry. Tyto parametry jsou název, krátký popisek, přidání upozornění, nastavení datumu a času, délka úkolu, nebo nastavení přidání do Apple kalendáře, kde je možné vidět od kdy do kdy úkol probíhá. Dále si uživatel může nastavit kategorii, popřípadě si může vytvořit svoji vlastní. Bohužel nebyly v praktické části splněny všechny cíle. Notifikace pomocí emailu nefunguje a aplikace není publikována v AppStoru. Toto bylo zapříčiněno mým špatným časovým rozložením vývoje aplikace a nedostatečnou informovaností o vývojářské licenci, která je nutná pro publikování aplikace a testování na fyzickém zařízení.

5 Summary and keywords

The aim of this bachelor work is to create a sheduler application for IOS mobile phones, using the Xamarin development platform and publishing the aplication in the App Store. Users can plan activities, meetings, and other tasks, for all of this task he can set a notification, a category or write a short description of the activity.

The theoretical part of the bachelor thesis presents the development platform Xamarin, the tools used for the aplication development and the description of a creation an IOS application by using the Xamarin development platform in Windows operating systems.

The practical part describes the development of a mobile applications from the creation of the user interface, the programming of the application logic to a connection with an Apple calendar.

Keywords: Xamarin development platform, IOS, App Store, scheduler, mobile phone

6 Seznam použité literatury

AAHN INFOTECH. (2012, June 23). Software Architecture - Definition. Retrieved from https://web.archive.org/web/20120623081009/aahninfotech.com/arct_pattern.html#

Alam, S. M. (2017, March 19). VIPER Design Pattern in Swift for iOS Application Development. Retrieved from https://medium.com/@smalam119/viper-design-pattern-for-ios-application-development-7a9703902af6

Apple Inc. (2015, September 16). About Developing for Mac. Retrieved from https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html#//apple_ref/doc/uid/TP40001067-CH204-TPXREF101

Apple Inc. (2015, September 16). Cocoa Application Layer. Retrieved from https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html#//apple_ref/doc/uid/TP40001067-CH274-SW1

Apple Inc. (2015, September 16). Core OS Layer. Retrieved from https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40001067-CH9-SW1

Apple Inc. (2015, September 16). Core Services Layer. Retrieved from https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40001067-CH270-BCICAIFJ

Apple Inc. (2015, September 16). Kernel and Device Drivers Layer. Retrieved from https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemTechnology/SystemTechnology.html#//apple_ref/doc/uid/TP40001067-CH207-BCICAIFJ

Apple Inc. (2015, September 16). Media Layer. Retrieved from https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40001067-CH273-SW1

Applikey Team. (2018, June 29). Xamarin Forms vs Xamarin Native: What Fits You Best? Retrieved from https://applikeysolutions.com/blog/xamarin-forms-vs-xamarin-native-what-fits-you-best

Cuello, J., & Vittone, J. (2013). *Designing Mobile Apps*. José Vittone.

ElHady, H. (2018, October 1). Your Guide to Cross-Platform Mobile App Development Tools. Retrieved from https://instabug.com/blog/cross-platform-development/

Giri, P. (2016, June 9). Understanding the Xamarin Framework. Retrieved from https://www.3pillarglobal.com/insights/understanding-xamarin-framework

Microsoft a. s. (2017, March 21). Architektura aplikací iOS - Xamarin. Retrieved from https://docs.microsoft.com/cs-cz/xamarin/ios/internals/architecture

Orlov, B. (2015, November 28). iOS Architecture Patterns. Retrieved from https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52

Williams, N. (2017, September 28). Xamarin.Forms or Xamarin Native: How to Choose. Retrieved from https://arctouch.com/blog/xamarin-forms-xamarin-native/

Čapek, P. (2014). *Moderní metody tvorby nativních multiplatformních mobilních aplikací* (Doctoral dissertation, Univerzita Tomáše Bati ve Zlíně, Zlín, Česká republika). Retrieved from https://digilib.k.utb.cz/bitstream/handle/10563/30263/%C4%8D%C3%A1pek_2014_dp.pdf?sequence=1&isAllowed=y

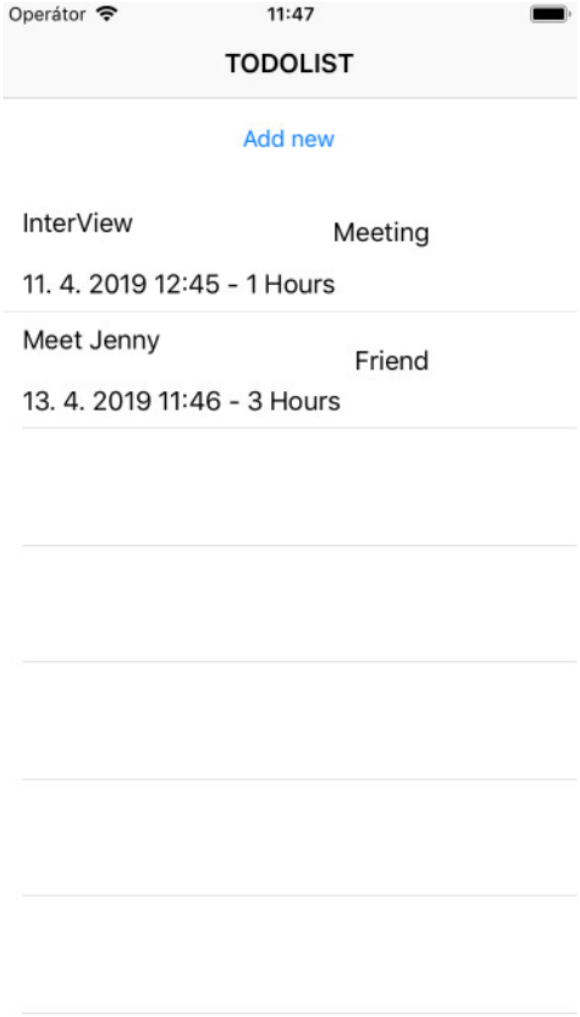
7 Seznam obrázků

8 Seznam ukázek kódu

9 Seznam příloh

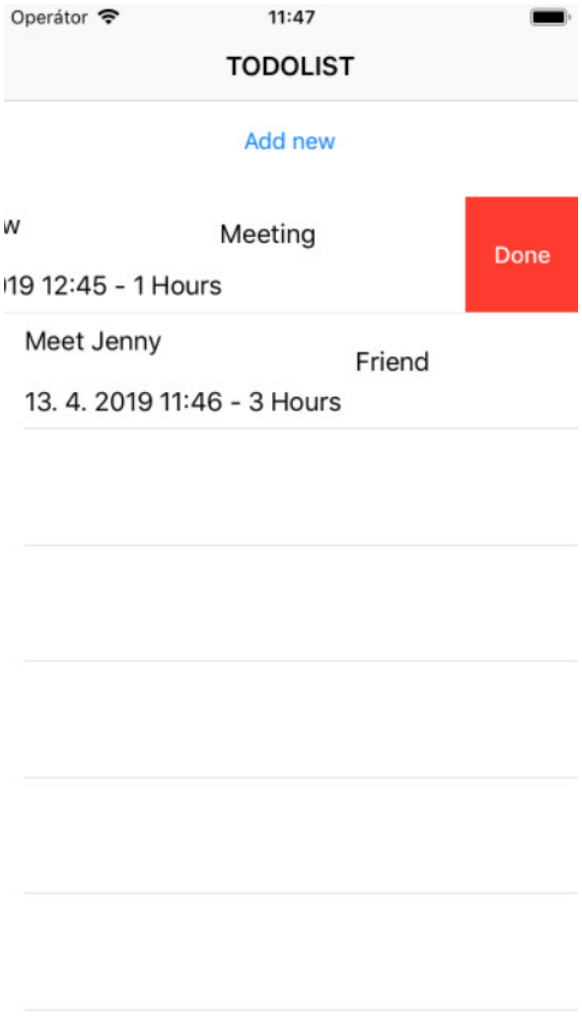
10 Přílohy

Příloha 1: Seznam úkolů



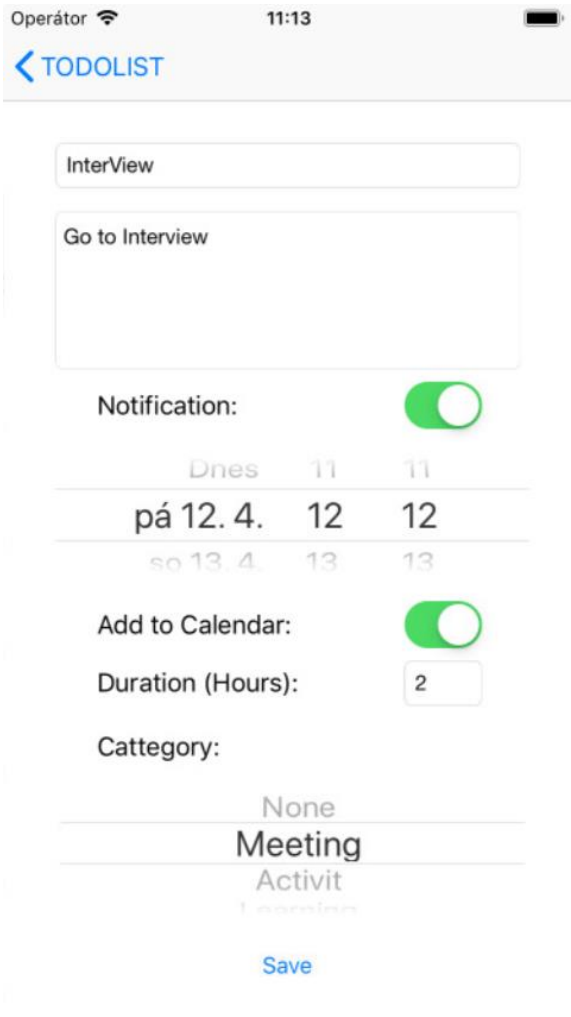
Zdroj: (Autor)

Příloha 2: Označení úkolu za hotový



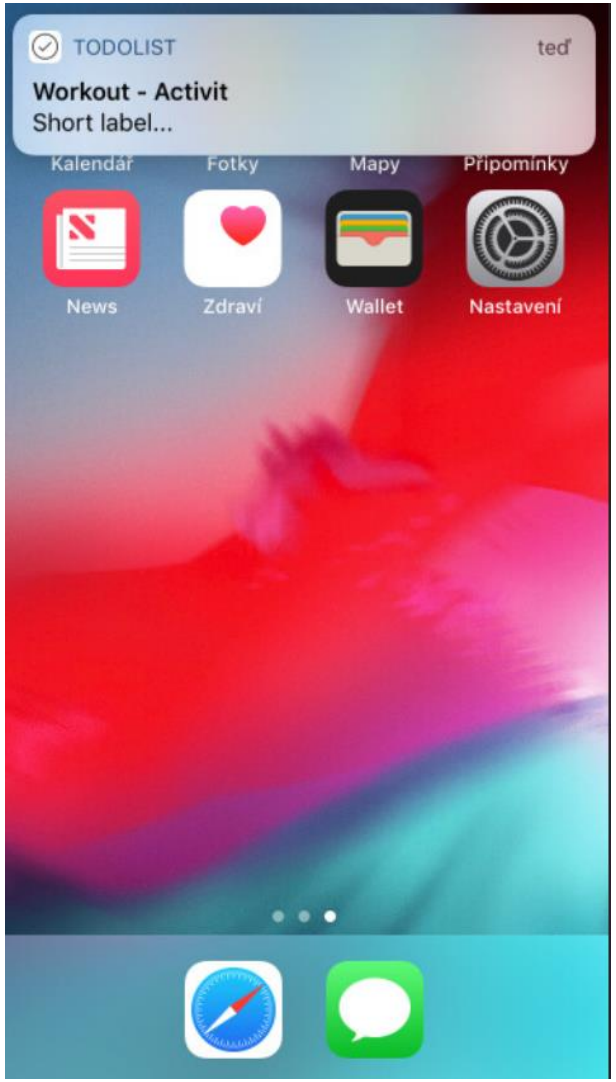
Zdroj: (Autor)

Příloha 3: Detail úkolů (editace/vytvoření)



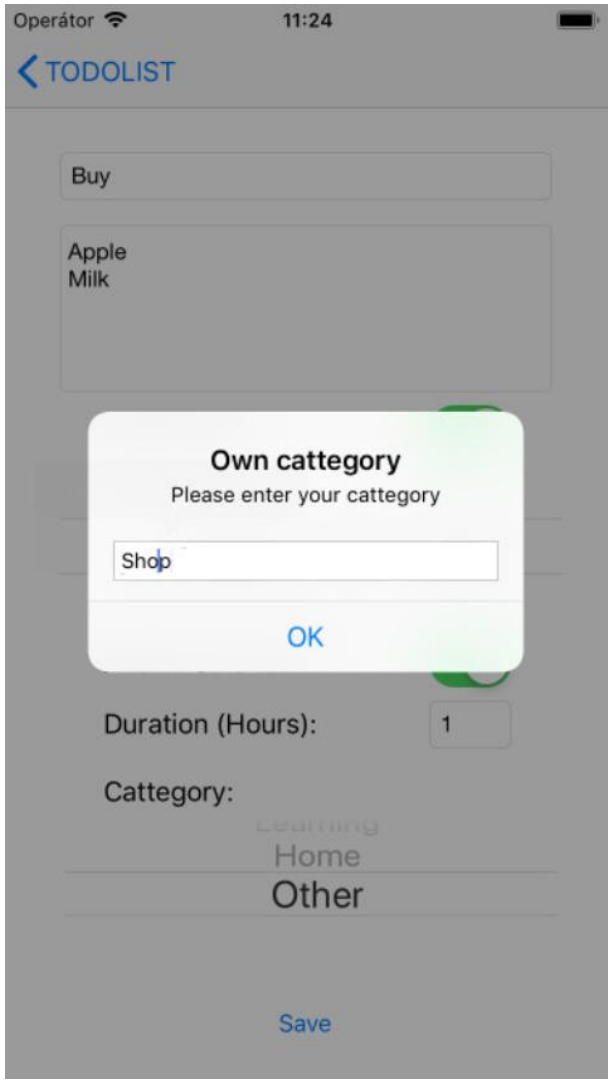
Zdroj: (Autor)

Příloha 4: Zobrazení notifikace



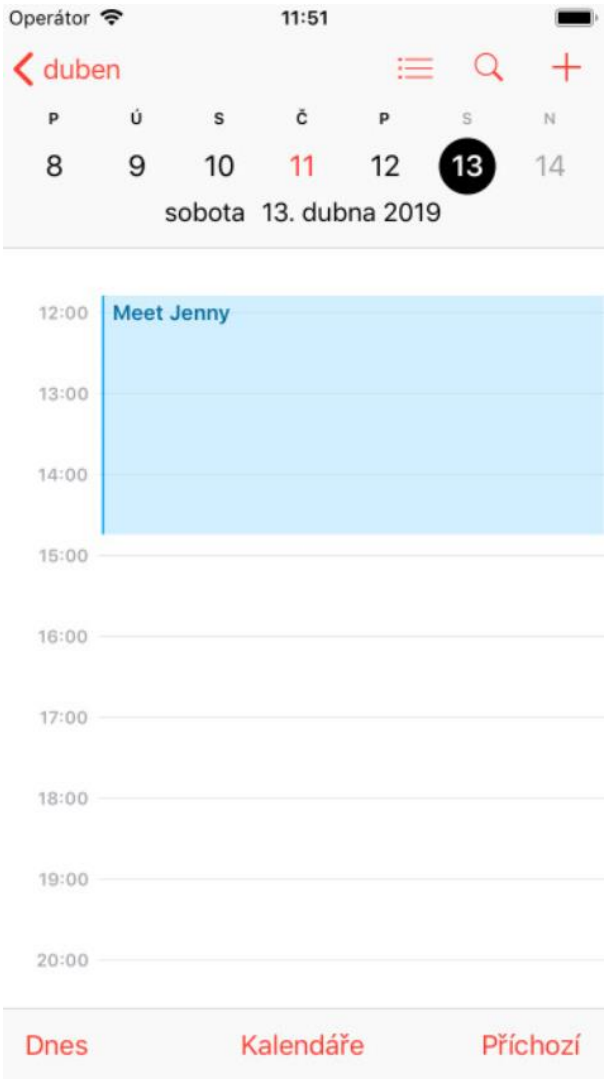
Zdroj: (Autor)

Příloha 5: Přidání vlastní kategorie



Zdroj: (Autor)

Příloha 6: Ukázka propojení s kalendářem



Zdroj: (Autor)