Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Proces vývoje mobilní aplikace pro platformu iOS

Tomáš Hamerník

© 2018 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Tomáš Hamerník

Informatika

Název práce

Proces vývoje mobilní aplikace pro platformu iOS

Název anglicky

The mobile development process for iOS platform

Cíle práce

Cílem práce je popsat ideální postup vývoje mobilní aplikace pro platformu iOS s aplikací těchto postupů při tvorbě reálného projektu. Vytvoří tedy obecný podklad, který řeší problémy od počátečního návrhu aplikace, přez vytvoření základní struktury, volby důležitých knihoven třetích stran, testování až po dokumentaci zdrojového kódu. Tyto poznatky budou aplikovány v praktické části práce, při vývoji mobilní aplikace.

Metodika

Metodika diplomové práce je založena na odborné literatuře a reálných vědomostech v oblasti mobilního vývoje. Na základě nabytých znalostí bude popsán proces mobilního vývoje tak, jak by měl v ideálním případě probíhat a jaké nástroje a technologie k němu použít.

Takto popsaný proces je následně použit při tvorbě demonstrativní mobilní aplikace

Doporučený rozsah práce

50

Klíčová slova

iOS, mobilní vývoj, Swift

Doporučené zdroje informací

KEUR, C.; HILLEGASS, A.; CONWAY, J. iOS Programming: The Big Nerd Ranch Guide. Atlanta: Big Nerd Ranch Guides, 2014. 560s. ISBN 978-0321942050.

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Josef Pavlíček, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 11. 1. 2018

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 11. 1. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 19. 03. 2018

Oficiální dokument * Česká zemědělská univerzita v Praze * Kamýcká 129, 165 00 Praha 6 - Suchdol

Čestné prohlášení

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

V Praze dne 22.3.2018 ______________________________

Poděkování

Rád bych touto cestou poděkoval panu Ing. Josefu Pavlíčkovi, Ph.D. za odbornou pomoc, konzultaci a vedení práce.

Proces vývoje mobilní aplikace pro platformu iOS

Abstrakt

Tato práce pojednává o procesu mobilního vývoje, specificky pro operační systém iOS. Hlavním cílem práce je aplikovat získané znalosti v oblasti mobilního vývoje do realizace vývoje mobilní aplikace v praktické části práce.

Teoretická část práce pojednává o základních krocích v procesu mobilního vývoje. Jedná se například o specifikace zadání a analýzu požadavků, metodiky řízení softwarového projektu a jeho nástroje. Následně je věnována pozornost nástrojům pro návrhy a prezentaci grafického rozhraní. V druhé polovině teoretické práce je věnována pozornost tématům více technickým, jako jsou například architektury mobilních aplikací, základním knihovnám třetích stran nebo testování.

Praktická část poté popisuje tvorbu reálné aplikace na podkladech vědomostí z teoretické části práce a praxe.

Klíčová slova: iOS, Platforma Apple, Swift, mobilní vývoj, programování mobilních zařízení, CocoaPods

The mobile development process for iOS platform

Abstract

This master thesis deals with issue of mobile development process for iOS platform. Main goal is to apply gained knowledges into mobile application development in practical part of this thesis.

Theoretical part contains information about basic topics of mobile development process. As these topics could be considered for example requirement analysis, project management methodology and its tools. In the next step is focus on graphics interface tools. In the second half of the theoretical part is attention more focused on technical topics as mobile application architecture, basic third-party libraries or testing.

Practical part of this thesis contains description of development process on real application based on the gained knowledges from theoretical part and praxis.

Keywords: iOS, Apple platform, Swift, mobile development, mobile devices programming, CocoaPods

Obsah

1 Úvod **12**

2 Cíl práce a metodika **13**

2.1 Cíl práce 13
2.2 Metodika 13

3 Teoretická východiska **14**

3.1 Nápad / Zadání 14
3.1.1 Analýza požadavků 14
3.1.1.1 Zadání 14
3.1.1.2 Technologické požadavky jednotlivých platforem 15
3.1.1.3 Odhady času a ceny projektu 15
3.1.2 Funkční specifikace 16
3.1.3 Souhrn kapitoly 16
3.2 Používané nástroje a metodiky pro řízení IT projektů 16
3.2.1 Metodiky 16
3.2.1.1 Waterfall 17
3.2.1.2 Agile 18
3.2.2 Nástroje 19
3.2.2.1 JIRA 19
3.2.2.2 Asana 20
3.2.2.3 Trello 20
3.2.3 Souhrn kapitoly 20
3.3 UI a nástroje pro jeho zpracování 21
3.3.1 Sketch 21
3.3.1.1 Vlastnosti Sketche 21
3.3.1.2 Ceník 22
3.3.2 Zeplin 23
3.3.2.1 Vlastnosti Zeplinu 23
3.3.2.2 Ceník 24
3.3.3 Invision 24
3.3.4 Souhrn kapitoly 25
3.4 Architektury aplikace a volba správné architektury 25
3.4.1 MVC 25
3.4.2 MVP 27
3.4.3 MVVM 28

3.4.4 Viper 28
3.4.5 Souhrn kapitoly 29
3.5 Git 30
3.6 CocoaPods a knihovny třetích stran 30
3.6.1 Instalace a práce s CocoaPods 30
3.6.2 Základní knihovny třetích stran 31
3.6.2.1 Alamofire 31
3.6.2.2 SnapKit 32
3.6.2.3 SwiftyJSON 33
3.6.3 Souhrn kapitoly 34
3.7 Testování 34
3.7.1 Úvod do problematiky testování mobilních aplikací 34
3.7.2 Unit testy 35
3.7.3 UI testy 35
3.7.4 UX testování 36
3.7.5 Testování komunikace se serverem 36
3.7.6 Testování funkcionalit 37
3.7.7 User acceptance testing (testování uživateli) 38
3.7.8 Souhrn kapitoly 38
3.8 Dokumentace zdrojového kódu 38
3.9 Distribuce aplikace 39
3.9.1 App store distribuce 39
3.9.2 Test Flight 40
3.9.3 Enterprise distribuce 40
3.9.4 Další možnosti distribuce testovacích verzí 40
3.9.5 Souhrn kapitoly 41
4 Vlastní práce 42
4.1 Specifikace požadavků 42
4.2 Analýza požadavků 42
4.2.1 Specifikace základních parametrů 42
4.2.2 Analýza jednotlivých požadavků 43
4.2.3 Časové odhady požadavků 43
4.2.4 Souhrn kapitoly 44
4.3 Návrh UI 44
4.3.1 Tvorba podkladů v nástroji Sketch 44
4.3.2 Zeplin.io 45
4.3.3 Souhrn kapitoly 47
4.4 Git 47
4.5 Volba architektury 49
4.6 Struktura projektu 50

4.7 Realizace požadavků ... 51
4.7.1 Autentifikace ... 51
4.7.2 Uživatelské vakcinace (záznam i čtení) ... 52
4.7.3 Filtrování budoucích a již proběhnutých vakcinací ... 55
4.7.4 Uživatelský profil ... 56
4.7.5 Události v kalendáři ... 57
4.7.6 Tvorba UI ... 57
4.7.7 Souhrn kapitoly ... 58
4.8 Testy ... 59
4.8.1 Unit Testy ... 59
4.8.2 UI testy ... 60
4.8.3 Souhrn kapitoly ... 60
4.9 Dokumentace zdrojového kódu ... 61
4.9.1 Realizace dokumentace zdrojového kódu ... 61
4.9.2 Souhrn kapitoly ... 62
5 Výsledky a diskuse ... 63
6 Závěr ... 65
7 Seznam použitých zdrojů ... 67

Seznam obrázků

Obrázek 1 - Waterfall vs Agile [15] ... 17
Obrázek 2 – JIRA1 [17] ... 20
Obrázek 3 - JIRA2 [17] ... 20
Obrázek 4 – Sketch [20] ... 21
Obrázek 5 - Sketch Artboard [21] ... 22
Obrázek 6 – Zeplin [22] ... 23
Obrázek 7 – Invision [24] ... 24
Obrázek 8 - MVC originální [26] ... 26
Obrázek 9 – Apple MVC [27] ... 27
Obrázek 10 – MVP [26] ... 27
Obrázek 11 – MVVM [26] ... 28
Obrázek 12 – VIPER [26] ... 29
Obrázek 13 - Charles proxy [45] ... 37
Obrázek 14 – Sketch ukázka ... 45
Obrázek 15 - Zeplin dashboard ... 46
Obrázek 16 - Zeplin Get started obrazovka ... 47

Obrázek 17 - Bitbucket 48
Obrázek 18 - Source tree 48
Obrázek 19 - gitignore.io 49
Obrázek 20 - Firebase Authentication 51
Obrázek 21 - JSON struktura vaccines 53
Obrázek 22 - JSON struktura users 54
Obrázek 23 - Profile Common Cell 58
Obrázek 24 - Formátováný komentář 61
Obrázek 25 - Orientace pomocí Mark 62
Obrázek 26 - Část obrazovek aplikace 63

Seznam tabulek

Tabulka 1 - Časové odhady požadavků 44

1 Úvod

V době chytrých telefonů a mobilních zařízení je aktuálním tématem tvorba aplikací pro taková zařízení. Vývoj těchto aplikací je náročný proces, který kombinuje mnoho různých znalostí i oborů dohromady tak, aby vznikl produkt, se kterým bude klient, potažmo uživatelé, spokojeni. Kvalitní tvorba takových aplikací je v jistém smyslu umění, které ne každý ovládá. Umění ne, jen ve smyslu designu a vzhledu, ale i ve stylu kódu, architektury a technologií, které takový produkt dělají jedinečným.

Právě tímto způsobem je třeba k tvorbě aplikací přistupovat, jako k něčemu krásnému, kde na první pohled dokáže každý ocenit funkčnost a design, avšak jen zasvěcený ocení provedení. Oba tyto pohledy jdou ruku v ruce. Výstupem je produkt, který odráží obě strany mince, mobilní aplikace.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je popsat ideální postup vývoje mobilní aplikace pro platformu iOS s aplikací těchto postupů při tvorbě reálného projektu. Vytvoří tedy obecný podklad, který řeší problémy od počátečního návrhu aplikace, přes vytvoření základní struktury, volby důležitých knihoven třetích stran, testování až po dokumentaci zdrojového kódu. Tyto poznatky budou aplikovány v praktické části práce, při vývoji mobilní aplikace.

2.2 Metodika

Metodika diplomové práce je založena na odborné literatuře a reálných vědomostech v oblasti mobilního vývoje. Na základě těchto znalostí bude popsán proces mobilního vývoje tak, jak by měl v ideálním případě probíhat a jaké nástroje a technologie k němu použít. Takto popsaný proces je následně použit při tvorbě demonstrativní mobilní aplikace.

3 Teoretická východiska

3.1 Nápad / Zadání

Základem každé mobilní aplikace bývá vlastní nebo klientské zadaní, které odráží vlastní nápad na základě obchodní či jiné potřeby. Takový nápad nebo zadání by měl mít smysl jak z pohledu obchodního, tak musí být realizovatelné z pohledu technologického. K tomuto účelu se vytváří analýza viz kapitola 3.1.1 Analýza požadavků.

3.1.1 Analýza požadavků

Pod pojmem analýza se pro potřeby této práce míní pojem, který je specifický k projektům softwarového původu, v tomto případě přímo k mobilním aplikacím. Za analýzu se tedy považuje prvotní přetvoření zadání nebo myšlenky klienta do specifického dokumentu. Na téma analýzy a druhů analýzy by se dala napsat samostatná odborná práce, a proto se zde budeme zabývat pouze náhledem na celou problematiku.

3.1.1.1 Zadání

Ačkoliv zadavatel (klient) má svůj nápad nebo požadavek, který chce realizovat, je většinou potřeba dořešit jeho přesnou podobu, využití nebo náročnost jeho provedení. Proto je prvním krokem tyto nejasnosti vyřešit. K tomu je přímá cesta přes osobní schůzky klienta, projektového manažera a zástupců technicky kvalifikovaných pracovníků zodpovědných za realizování požadavků jako takových.
Pro specifikování zadání se dají použít různé metody. Za tyto metody lze považovat například:

- Prototypování s klientem – je možně použít jak fyzický (například papírový prototyp) tak softwarový prototyp.
- Use cases aplikace – Scénáře použití vytvářeného produktu.

Na základě těchto metod je pro řešitele pochopitelnější požadavek klienta a zároveň klient může dojít k závěru, že nějaký z jeho požadavku je irelevantní, velice náročný nebo neproveditelný [13].

3.1.1.2 Technologické požadavky jednotlivých platforem

Pokud je zadání v odpovídajícím stavu tak, že jsou požadavky klienta jasné, je možné řešit technologické provedení na jednotlivých platformách projektu. V oblasti mobilních aplikací se nejčastěji řeší následující platformy:

- Android.
- iOS.
- Serverová část.

Každá platforma má jiné řešení klientova požadavku. Příkladem za všechny může být implementace platební brány. V tomto případě by se v dané části analýzy řešila její implementace pro každou platformu zvlášť, jelikož vybraná platební brána bude mít pro každou platformu odlišnou implementaci a různé problémy s ní spojené. Pro každý majoritní požadavek je tedy vytvořen technologický pohled a struktura řešení, případně řešení problému s tím spojených (například bezpečnost) [13].

Dále je potřeba vyspecifikovat verze operačních systémů a hardware v případě serverové části, na kterých aplikace poběží. Takto zvolené nastavení musí být náležitě zdůvodněno, například:

- Potřebou výkonu.
- Nutností využívat nejnovější technologie systémů.
- Nižší časovou náročností.

3.1.1.3 Odhady času a ceny projektu

Výsledkem analýzy je pro klienta jednak pohled na řešení, ale především časový a tím i cenový odhad. Tento odhad je vytvořen odpovědnými technickými odborníky na základě analýzy požadavků klienta. Pokud klient nesouhlasí s výsledným časovým odhadem, ať už z požadavků na termín nebo cenu, je možné odhady modifikovat například změnou požadavků klienta. Odebere se tak vybraná méně kritická funkcionalita, která bude přidána například v další fázi vývoje projektu nebo se úplně vypustí.

Pokud je vše akceptováno, je možné přejít k dalším krokům specifikování zadání, například k vytvoření funkční specifikace [13].

3.1.2 Funkční specifikace

Funkční specifikace v kontextu této práce je dokument vystavěn jak k business účelům, tak k účelům tvorby samotného produktu. Podle autora Tysona Gilla je funkční specifikace *self-contained master plan of what will be produced* [14], neboli dokument obsahující ultimátní plán toho, co bude vyprodukováno. Podle tohoto tvrzení je tedy tento druh specifikace absolutní odpovědí na všechny otázky, které by mohly vyvstat při vývoji produktu [14].

Tento dokument je vystavěn na základě analýzy požadavků viz kapitola 3.1.1 Analýza požadavků. Obsahuje již přesně vyspecifikované chování jednotlivých obrazovek, prvků a ukazatelů v aplikaci. Jelikož je funkční specifikace absolutní dokument, mohou být jeho součástí také UI (user interface) specifikace a další druhy specifikací, na které je možné se odkazovat. Podle takto vyspecifikovaného zadání se vytváří jednotlivé moduly úkolů a podúkolů, které jsou dále předávány vývojářům. Funkční specifikace je tedy směrodatný dokument, dle kterého se produkt tvoří a na který se klient v případě nedostatků odvolává [14].

3.1.3 Souhrn kapitoly

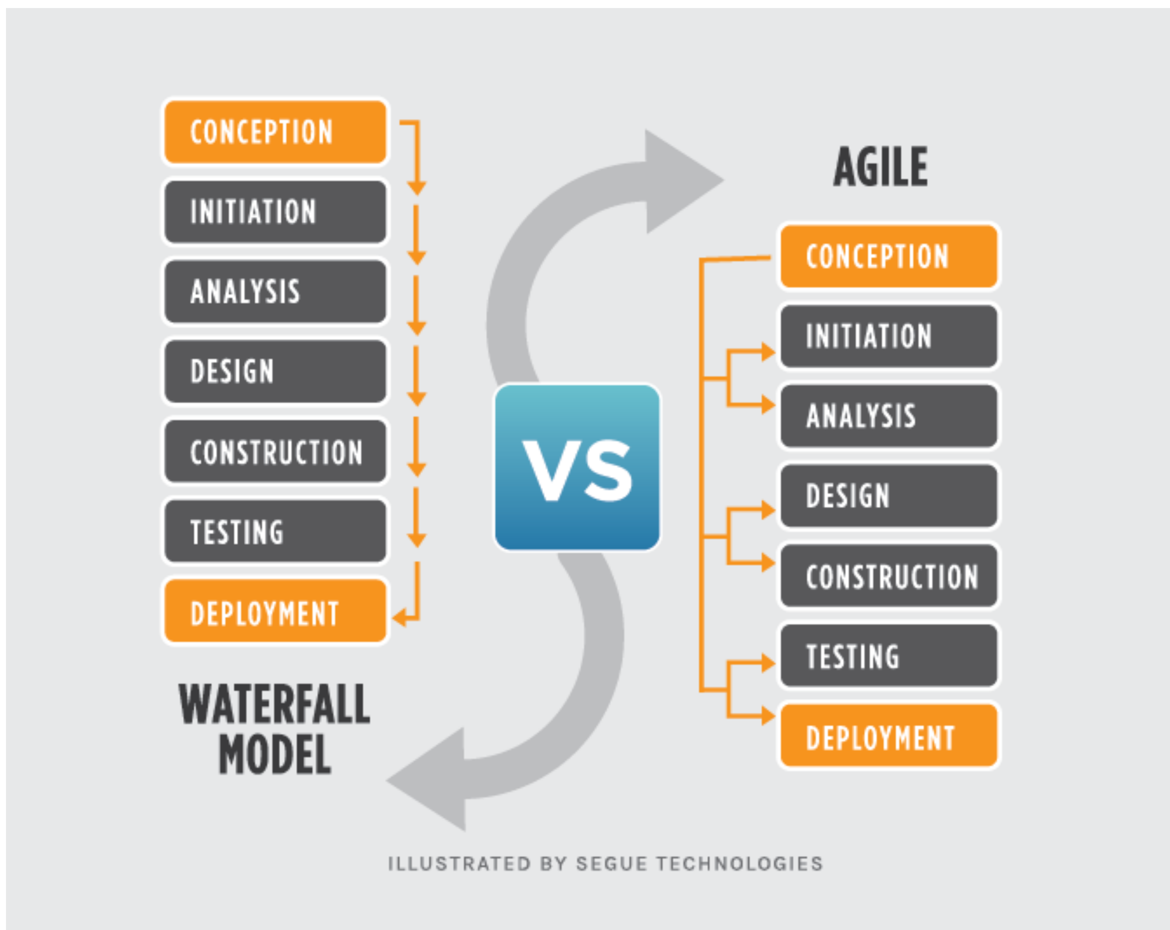
Kapitola popisuje základní problematiku analýzy požadavků, na jejíž základě je následně možné vytvářet funkční specifikaci či další druhy specifikací.

3.2 Používané nástroje a metodiky pro řízení IT projektů

V této kapitole je pozornost zaměřena na metodiky organizování práce v IT projektu a nástroje, které jsou těmto účelům využívány. Výběr metodiky potažmo softwarového nástroje, kterým se vývoj projektu koordinuje, je jeden z prvních kroků ještě předtím, než začne samotný vývoj daného produktu. Na metodiky vývoje by se dala vytvořit celá samostatná práce, proto se zde budeme zabývat pouze nejčastějšími případy daného problému.

3.2.1 Metodiky

Metodikou se rozumí v tomto případě pojem, který popisuje organizaci práce při vývoji softwarového produktu. V této kapitole se zaměříme na dva nejpoužívanější, a to jsou *Waterfall* a *Agile* metodologie. Ilustrace rozdílu mezi Waterfall a Agile metodologií je znázorněn na obrázku 1.



Obrázek 1 - Waterfall vs Agile [15]

3.2.1.1 Waterfall

Jedná se o lineární přístup k vývoji softwaru. Jeho postup může být popsán s drobnými odlišnostmi, ale ve výsledku vždy stejný. Jednotlivé události mohou být například:

- Zkompletovat dokumentaci a požadavky problému.
- Design.
- Tvorba kódu případně unit testů.
- Testování.
- Akceptační testy UAT (User Acceptance Testing).
- Opravy chyb.
- Dodání produktu.

V opravdovém Waterfallu by měla každá činnost začít až když skončí předchozí. V realitě je ještě mezi jednotlivými činnostmi další činnost, jako například akceptace designu ze strany klienta a podobně [15].

Hlavní pozitivem této metodologie je její přímočarost. Všechny specifikace, design a další požadavky jsou upřesněny v raných stadiích tvorby projektu. Dalším pozitivem je dobrá měřitelnost pokroku, jelikož vše je určeno dopředu. Na hraně pozitiva a negativa se nachází neangažovanost klienta v průběhu vývoje. Klient není „otravován“, nicméně nevidí průběžné výsledky, které by mohl komentovat, a tak mohou vznikat problémy na konci projektu.

Oproti tomu negativní stránkou je jistá nepružnost celého procesu. Pokud se něco nevyspecifikuje správně, pak tato metodologie postrádá smysl. Dalším problémem je, že klient uvidí až produkt, který je v podstatě celý hotový a v tu chvíli může zjistit, že vlastně s výsledkem není spokojen. Tato nespokojenost většinou plyne ze špatného zadání [1, 2, 15].

3.2.1.2 Agile

Jedná se o iterativní týmový přístup k vývoji. Pracuje tak, že v časovém období zvané *sprinty,* se doručí balík funkcí nebo oprav naplánovaných na začátku daného sprintu. Každý sprint má tedy definovaný časový rámec, který se pohybuje obvykle v řádu týdnů. Pokud se nedaří splnit předpokládaný plán sprintu, pak přichází na řadu priority jednotlivých úkolů a nestihnuté či nedodělané úkoly se mohou přesunout do sprintu dalšího. Na konci každého sprintu je možné práci konzultovat s klientem a řešit akceptaci výsledků. K tomu je zapotřebí velká angažovanost klienta v průběhu projektu [1, 2, 15].

Výhody agilního přístupu:

- Angažovanost klienta – Vzniká těsná spolupráce s týmem a klientem. Klient si tak i vytváří vztah k produktu jako takovému.
- Rychlé a predikovatelné výsledky – Jelikož vývoj běží ve sprintech, je vždy jasné, co bude dodáno v krátkém časovém horizontu. To vytváří možnost průběžného testování vytvořených funkcionalit a revizí ze strany klienta.
- Predikovatelné náklady – Klient vidí, kolik jednotlivé požadavky stojí a může tak prioritizovat, které funkce jsou pro něj důležité i z hlediska nákladů na jejich tvorbu.
- Prostor pro změny – Klient může být nespokojen s výsledkem, případně ho chtít měnit nebo jinak editovat. V agilní metodice může v rámci další iterace prioritizovat svůj změnový požadavek, a tak ve výhledu několika týdnů dosáhnout změny, která nebyla původně v plánu.
- Kvalitnější produkt – Pokud se práce rozdělí na menší zvládnutelné, testovatelné části, zvedá se tím kvalita produktu jako takového. Je zde možnost rychle reagovat na změny a chyby, které je potřeba opravit [16].

Nevýhody:

- V některých případech může být velké zapojení klienta do projektu nevýhodou například z důvodu jeho časové vytíženosti.
- Metodika funguje jen pokud jsou členové týmu dostatečně odhodlaní, jelikož změnové požadavky a další úpravy mohou být frustrující.
- Při velkých změnách v průběhu projektu nemusí být odhady na celkovou cenu a časovou dotaci projektu přesné [16].

3.2.2 Nástroje

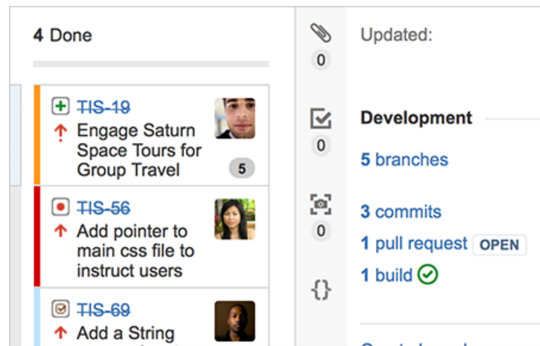
V této kapitole je zmíněno několik známých nástrojů pro management úkolů, používaných při vývoji softwaru.

3.2.2.1 JIRA

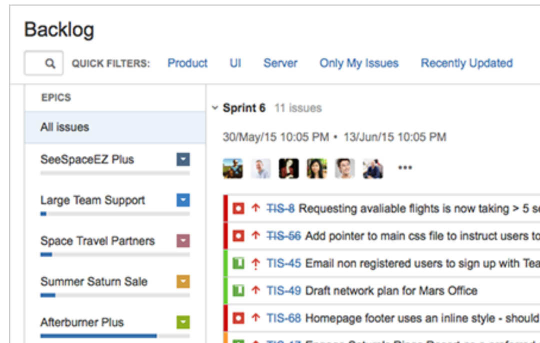
Nástroj JIRA od společnosti Atlassian je jeden z nejrobustnějších agilních nástrojů. Jeho nastavení je editovatelné enormním počtem způsobů, nicméně jeho poslání zůstává stejné. Tím, je sledovat a řídit jednotlivé úkoly při vývoji softwarového produktu.
Základní funkcionality:

- Podpora projektového řízení vzhledem k požadavkům a úkolům.
- Nastavitelné workflow (proces práce s úkoly).
- Sledování časových kapacit pracovníků.
- Sledování stavu úkolů.
- Sledování komunikace v týmu (například komunikace nad problémem jednoho úkolu).
- Reporty a statistiky.
- Plánování sprintů.
- Propojení s vlastní Git službou (verzovací nástroj)
- A mnoho dalších.

Na obrázcích 2 a 3 je ukázka prostředí JIRA. Jak je zde vidět, kromě task managementu (managementu úkolů) je zde funkce vytvářet branche na Gitu (separátní vývojové větve pro daný úkol), sledovat jejich review a další funkce s vývojem spojené. Díky tomu schválené a úkoly spojené do specifické větve mohou samy měnit svůj stav ve workflow a přeřazovat se na adekvátní pracovníky [17].



Obrázek 2 – JIRA1 [17]



Obrázek 3 - JIRA2 [17]

3.2.2.2 Asana

Podobně jako JIRA, je Asana nástroj na správu úkolů v projektu. Tento nástroj je zaměřen více na business a komunikaci, než na potřeby technických pracovníků. Oproti nástroji JIRA není Asana tak komplexní. Záleží na potřebách projektů, ale při větších projektech může být nedostatečná [18].

3.2.2.3 Trello

Opět nástroj pro týmovou komunikaci a správu a stav úkolů jako alternativa pro Asanu. Umožnuje vytvářet úkoly, podúkoly, přidávat komentáře a přílohy, přidělovat úkoly jednotlivým členům týmů a sledovat jejich stav. Trello podobně jako Asana může být v jistých případech neadekvátní náhrada za nástroj JIRA, která plní více funkcí najednou [19].

3.2.3 Souhrn kapitoly

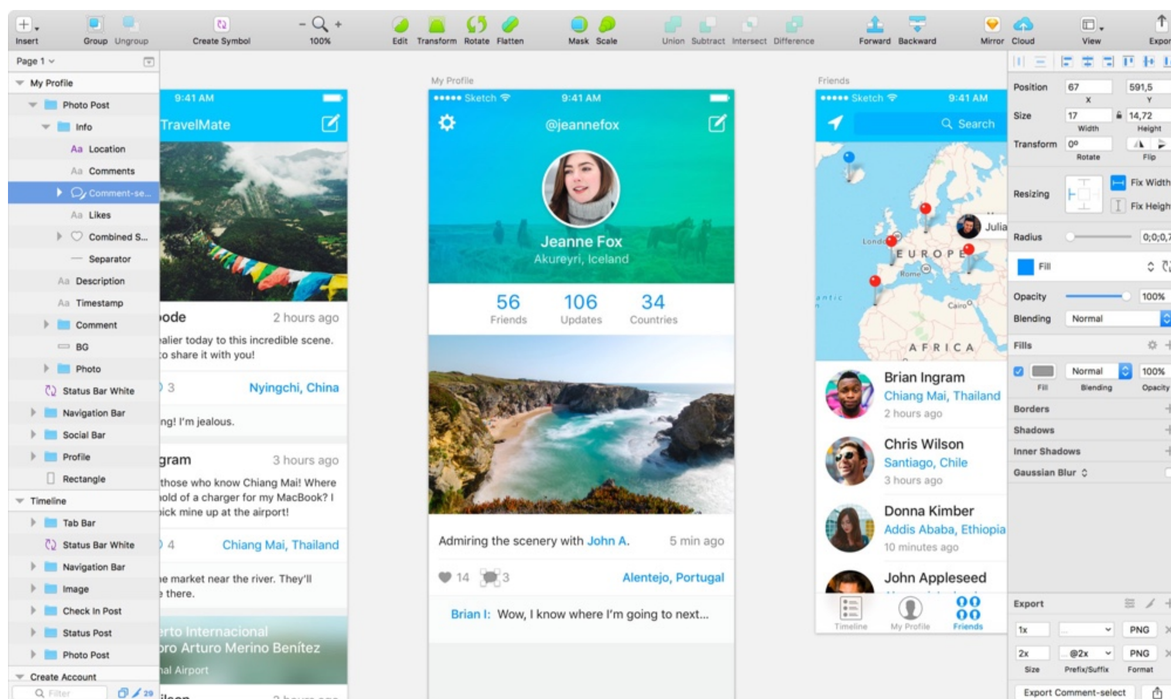
V kapitole byli představeny základní metodiky řízení použité při vývoji mobilních aplikací. Dále byli zmíněny tři nástroje, které se běžně při tomto procesu vývoje využívají.

3.3 UI a nástroje pro jeho zpracování

Co dělá úspěšnou aplikaci úspěšnou, je kromě originálních funkcionalit a nápadu, také její vzhled. O tvorbu adekvátního vzhledu aplikace se stará grafik. V tomto případě, je vhodné využít služby výhradně mobilního grafika, který má přehled v aktuálních trendech a vzhledech aplikací. V následující kapitole se budeme zabývat nejběžnějšími nástroji pro tvorbu UI.

3.3.1 Sketch

Sketch je aplikace pro tvorbu grafických designů aplikací, v tomto případě mobilních aplikací. Jedná se o jeden z nejpoužívanějších nástrojů pro tvorbu potřebných grafických podkladů. Aplikace jako taková je dostupná pouze na OS X El Capitan a novější (20). Výhodou Sketche oproti Photoshopu je jeho jednoduchost a připravenost na tvorbu mobilního UI rozhraní. Další výraznou výhodou je cenová politika tohoto softwaru oproti Photoshopu, která bude zmíněna později v této kapitole. Ukázka softwaru Sketch na obrázku 4.



Obrázek 4 – Sketch [20]

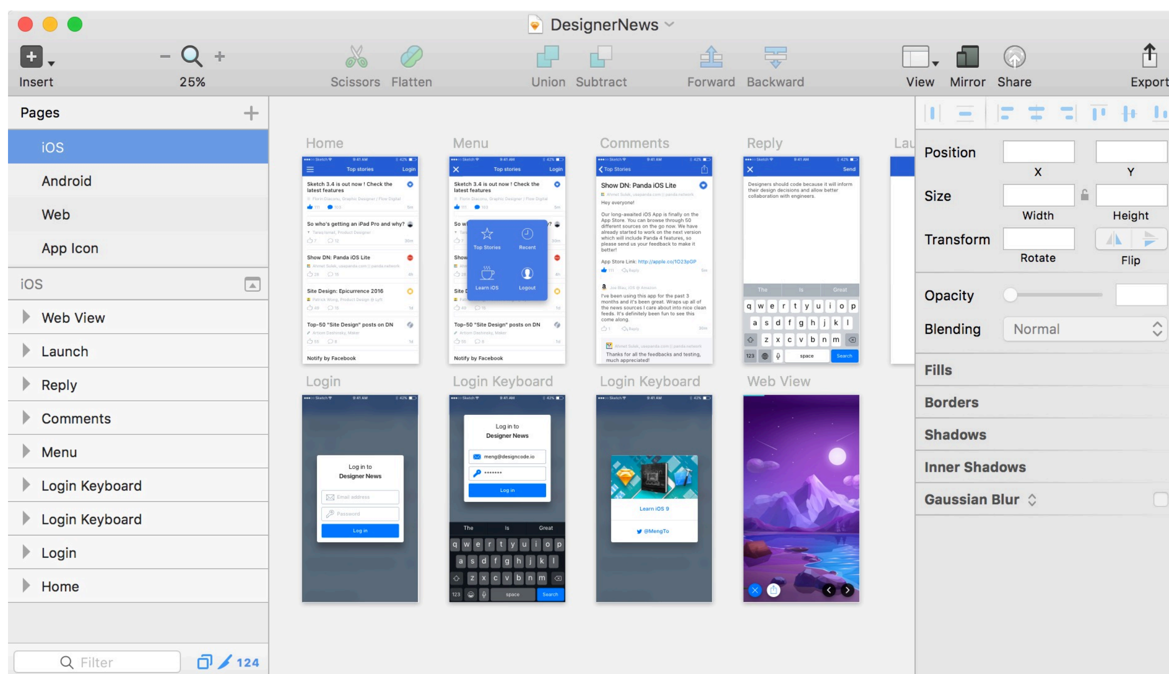
3.3.1.1 Vlastnosti Sketche

Hlavní předností Sketche je celková jednoduchost. I nezkušený uživatel může začít tvořit bez větších problémů. Jednotky, které jsou používány pro rozměry jsou pouze pixely, takže je naprosto jasné, v jakých velikostech se návrh pohybuje.

Sketch má mnoho vhodných funkcí, které jsou k tvorbě mobilního designu vyžadovány. Jedná se například o možnosti:

- Exportovat ikony / obrázky ve velikostech podle platformy,
- vektorové editace,
- tvořit pixel perfect (naprosto přesný) návrh,
- částečný export UI kódu,
- sledování doporučených guide lines (návrhových vzorů) a další.

Hlavním výstupem jsou pak takzvané Artboards. Jedná se vlastně o jednotlivé obrazovky v aplikaci, se kterými uživatel interaguje - viz obrázek 5 [20, 21].



Obrázek 5 - Sketch Artboard [21]

Sketch také obsahuje předlohy pro jednotlivé platformy a jejich verze. Jedná se tedy o základní vzhledy například klávesnice, tlačítek a dalších prvků tak, jak je to přirozené pro dané případy. Tvoří se tak návrh UI v nejaktuálnější podobě.

Práce s tímto nástrojem není předmětem této závěrečné práce, a proto se jím nebudeme hlouběji zabývat, nicméně je jedním z hlavních nástrojů mobilního vývoje v oblasti UI návrhů [3, 4, 20, 21].

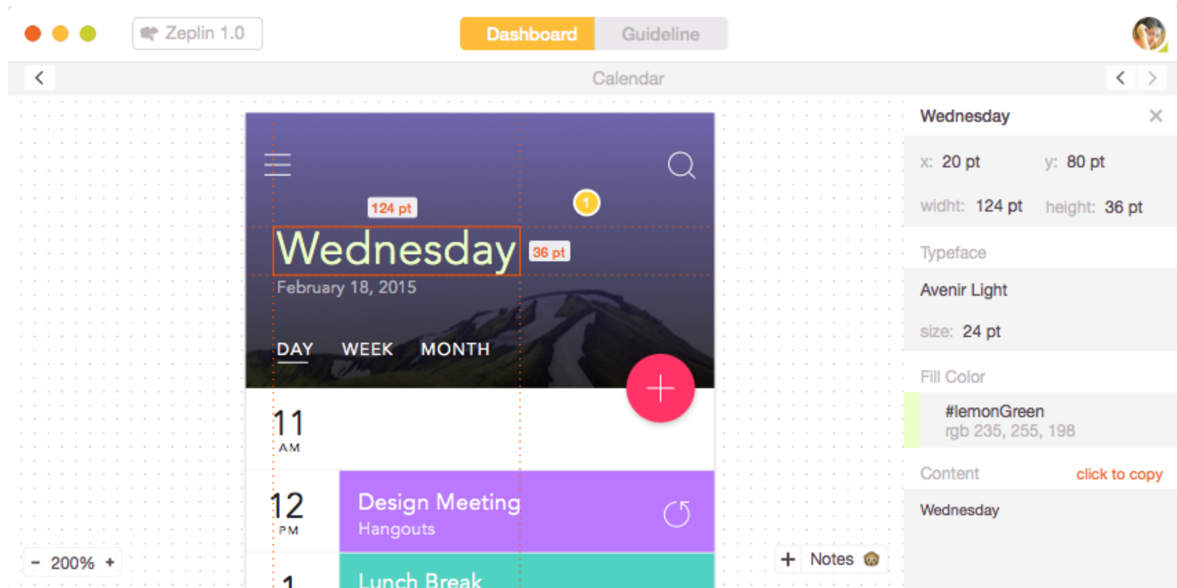
3.3.1.2 Ceník

Aplikace Sketch je placený nástroj s měsíční trial verzí. Pro samostatného uživatele je cena 99$ za rok, ale po skončení licence zůstává uživateli plně funkční aplikace, ovšem bez možnosti získávat další updaty. Obnova licence je pak levnější, a to už za 69$ na další rok.

Pro studenty a učitele je možnost zakoupit licenci o 50% levněji. Pokud by se jednalo o akademický institut je zde licence zdarma [20].

3.3.2 Zeplin

Zeplin je nástroj pro kooperaci mezi designéry a vývojáři. Designér zde zveřejní svoji práci vytvořenou ať už ve Sketchi nebo Photoshopu, a vývojář zde zjistí všechny podstatné informace, aniž by potřeboval mít placený grafický nástroj jako takový a znát jeho ovládání. Zkracuje se tak čas potřebný pro jednání mezi designéry a vývojáři o detailech a zvyšuje to efektivitu a pohodlí vývoje [22]. Ukázka nástroje Zeplin na obrázku 6.



Obrázek 6 – Zeplin [22]

3.3.2.1 Vlastnosti Zeplinu

Zeplin obsahuje všechny informace co potřebuje vývojář k tvorbě UI. Jedná se například o možnosti [22]:

- Vidět přesně rozměry jednotlivých prvků.
- Přesně změřit vzdálenosti prvků od sebe.
- Získat informace o fontech, barvách, odsazení textu a další.
- Možnost exportovat si nahrané ikony / obrázky jedním klikem bez nutnosti komunikovat s designérem ve správných velikostech.
- Možnost jednoduše vkládat komentáře do návrhů, pokud je něco nejasné nebo daný návrh postrádá požadovaný asset (obrázek).

3.3.2.2 Ceník

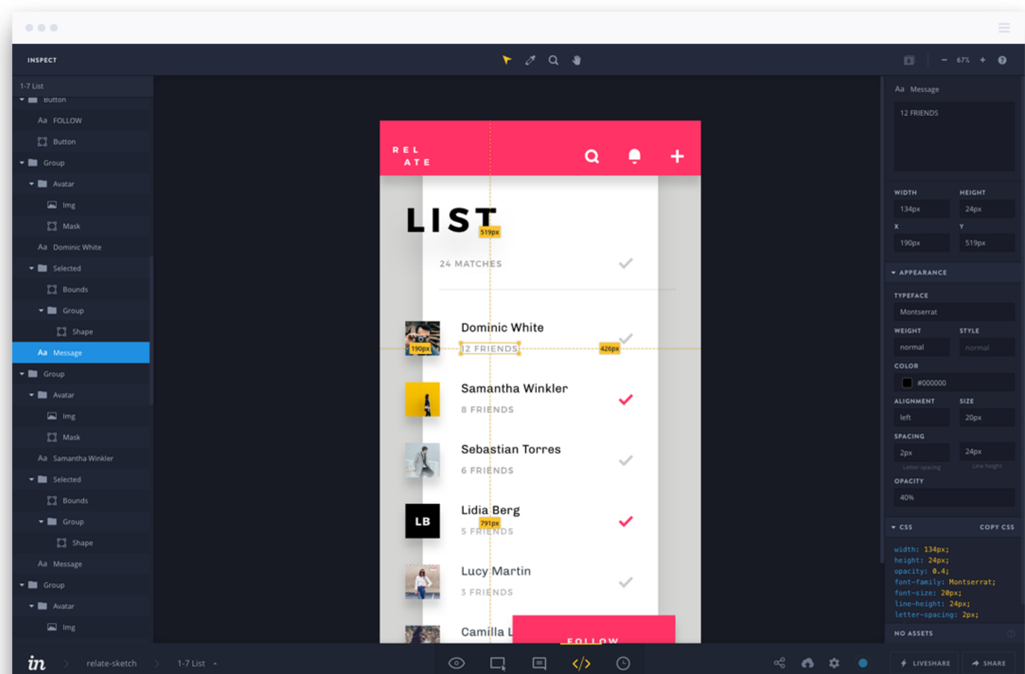
V případě Zeplinu je jeden aktivní projekt zdarma. U více projektů jde zde poplatek [23]:

- Do 3 projektů 17$ mesíčně
- Do 12 projektů 26$ měsíčně
- 6.75$ za uživatele měsíčně a neomezeně mnoho projektů

Poplatek lze částečně obcházet zaváděním schémat v jednom projektu, která oddělují projekty v „projektu“.

3.3.3 Invision

Nástroj Invision je podobně jako Zeplin vytvořen pro komunikaci mezi designéry a vývojáři. Designer zde nahraje své návrhy a může přidat gesta, animace a přechody mezi obrazovkami během několika minut. Vývojář tak vidí zamýšlenou posloupnost chodu aplikace. Ukázka nástroje Invision na obrázku 7.



Obrázek 7 – Invision [24]

Kromě vývojářů mohou takový nástroj využít i pro prezentaci dema aplikace klientovi, aniž by byla napsána jediná řádka kódu. Klient má tak možnost vkládat své myšlenky a připomínky, které napomáhají agilnímu vývoji viz kapitola 3.2.1.2.

Cenová politika je podobně jako u Zeplinu založena na měsíčních poplatcích s možností trial verze zdarma pro jednu ukázku [24].

3.3.4 Souhrn kapitoly

V kapitole byl popsán grafický nástroj Sketch, který je v běžné praxi hojně využíván k tvorbě grafických podkladů mobilních aplikací. Dále byla pozornost věnována nástrojům pro distribuci grafických podkladů vývojářům, potažmo klientovi.

3.4 Architektury aplikace a volba správné architektury

Základním kamenem při vývoji mobilních aplikací je jejich softwarová aplikační architektura. V tomto případě budou zmíněny pouze nejčastější architektury aplikovatelné na platformu iOS.

Mobilní softwarová aplikační architektura je množina technik a principů pro vytvoření plně funkční mobilní aplikace v závislosti na odvětví a požadavcích klienta pro bezdrátová mobilní zařízení jako jsou chytré telefony a tablety.

Volba architektury je důležitá fáze, jelikož při volbě špatné architektury se může rozsáhlý projekt stát nedebugovatelným (neopravitelným), nepřehledným, netestovatelným a celkově zmateným [25].

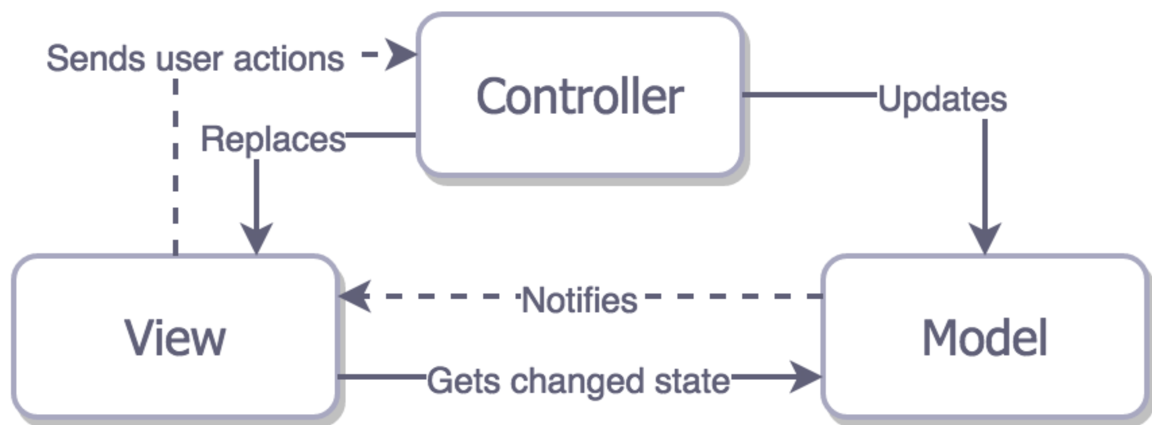
3.4.1 MVC

Model View Controller je základní návrhový vzor architektur pro iOS aplikace. Objekty v aplikaci mají jedno ze tří rolí a těmi jsou:

- Model - Jedná se o objekty, které uchovávají a zpracovávají data. Například se může jednat o strukturu osoby v adresáři a podobně. Modelové objekty pak mohou mít vazby na další modelové objekty vazbami 1-1 a 1-N. Takové modelové objekty mohou být pak přepoužívány napříč celou aplikací.
- View – Je objekt, který uživatel v aplikaci jako takové může vidět. View objekty se mohou samy vykreslit a reagovat na uživatelovy akce. Hlavním účelem View objektů je prezentovat data a umožnit je editovat. View objekty bývají většinou přepoužitelné (například UI pro buňky v tabulce).
- Controller – Tento objekt v podání MVC společnosti Apple jedná jako prostředník mezi View a Modelem. Implementuje logiku, předává změny do View objektu a zpět do modelu - viz. Obrázek 9.

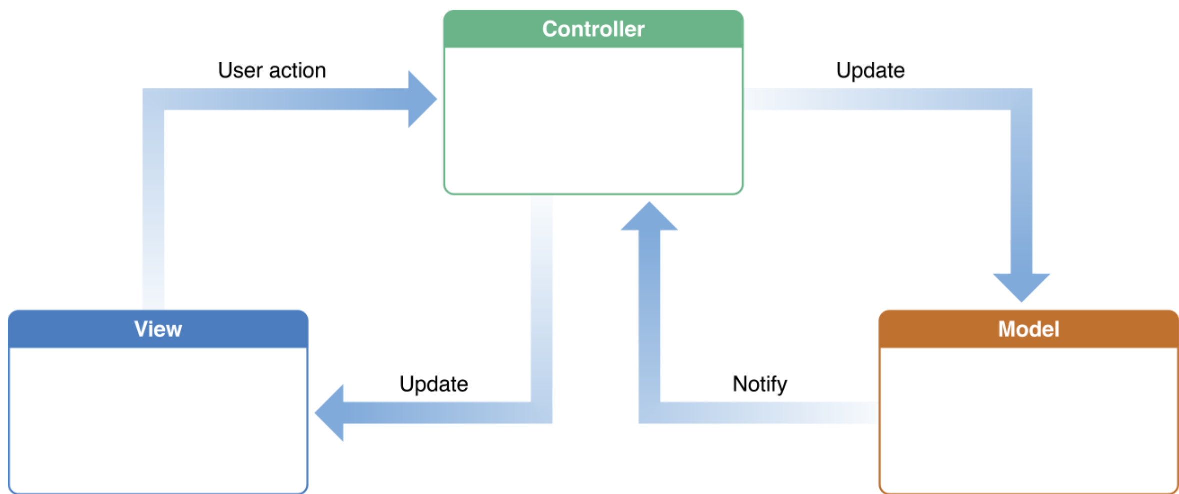
Vzor definuje kromě rolí objektů také způsob, kterým mezi sebou komunikují. Každá z rolí objektů je od další abstraktně odstíněna hranicí, přes kterou komunikuje s dalšími typy objektů.

V klasickém MVC návrhu (viz obrázek 8) je *View* objekt bezstavový a je renderován *Controllerem* jakmile se změní *Model* objekt. To ovšem není žádoucí, jelikož každý druh objektu ví o dalších dvou. Tato vlastnost zamezuje znovupoužitelnosti. Proto společnost Apple Inc. upravila návrhový vzor do podoby více vyhovující moderním standardům aplikačního iOS vývoje [5, 6, 26].



Obrázek 8 - MVC originální [26]

MVC v podání společnosti Apple (viz obrázek 9) je Controller zprostředkovatelem komunikace mezi View objektem a Model objektem tak, aby o sobě navzájem nevěděly. To dělá pouze controller nepřepoužitelným a to už je přijatelné. Problém, který tím vzniká, je udržitelnost velikosti Controller objektu, který může dosáhnout enormních rozměrů při implementaci logiky. Dalším problémem je netestovatelnost View objektů, které jsou těsně provázány s Controllery. Nedá se testovat logika, aniž by se zavolala nějaká z UIView metod, jako je *viewDidLoad* a další. To způsobí načítání všech View objektů, a to není dobré pro unit testování [26, 27].

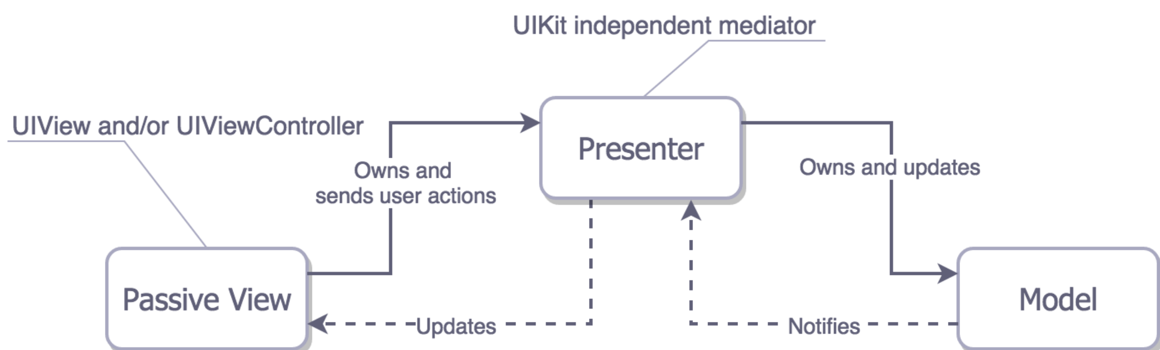


Obrázek 9 – Apple MVC [27]

Pro MVC v podobě, jakou prezentuje společnost Apple je tedy možné se rozhodnout, pokud v daném projektu nevadí již zmíněné nedostatky. Hlavní výhodou této architektury je její jednoduchost na použití a její obecná známost. Kód tak může udržovat i méně zkušený vývojář, který se v projektu rychle zorientuje. Obecně je tento vzor použitelný pro menší projekty, na které by jiná architektura byla zbytečně komplikovaná [7, 26].

3.4.2 MVP

Model View Presenter (viz obrázek 10) je architektura na první pohled podobná Apple verzi MVC architektury. Je zde ovšem markantní rozdíl a to, že Presenter nemá nic společného s životním cyklem ViewControlleru a View může být mockovatelné (uměle vytvořené). V případě MVP jsou UIViewController podtřídy také View, ale ne Presenters. Toto řešení poskytuje testovatelnost, která na druhou stranu odebírá rychlost vývoje, jelikož je nutné manuálně vytvořit vazby mezi daty a akcemi.



Obrázek 10 – MVP [26]

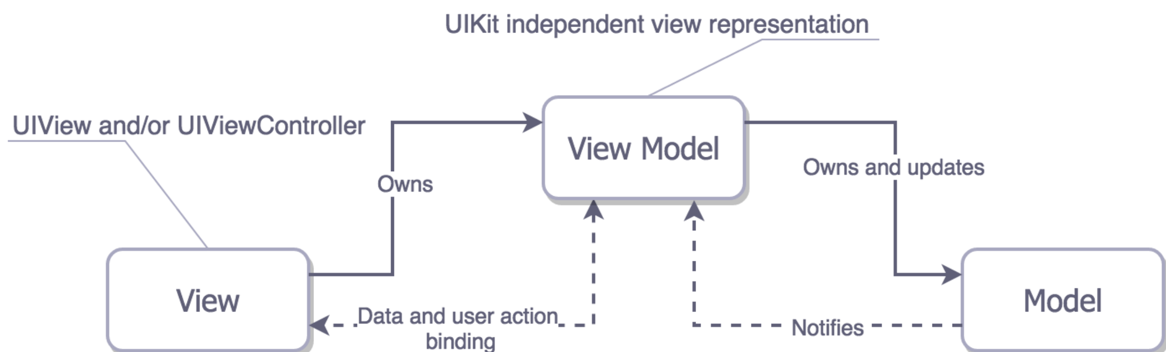
V této architektuře je tedy rozdělení zodpovědností jednotlivých rolí rozděleno především mezi Presenter a Model s tím, že View je „hloupý" objekt (jen zobrazuje předaná

data). Tím je zajištěna dobrá testovatelnost logiky. Nevýhodou je složitější tvorba a množství kódu, které je třeba na vytváření patřičných protokolů [8 ,9, 26].

3.4.3 MVVM

Poslední z řady těchto druhů architektur je *Model View ViewModel* architektura (viz obrázek 11). Jedná se podobnou myšlenku jako je tomu u MVP. MVVM zachází s Controllerem jako s View objektem a není zde žádná těsná vazba mezi View a Model objektem.

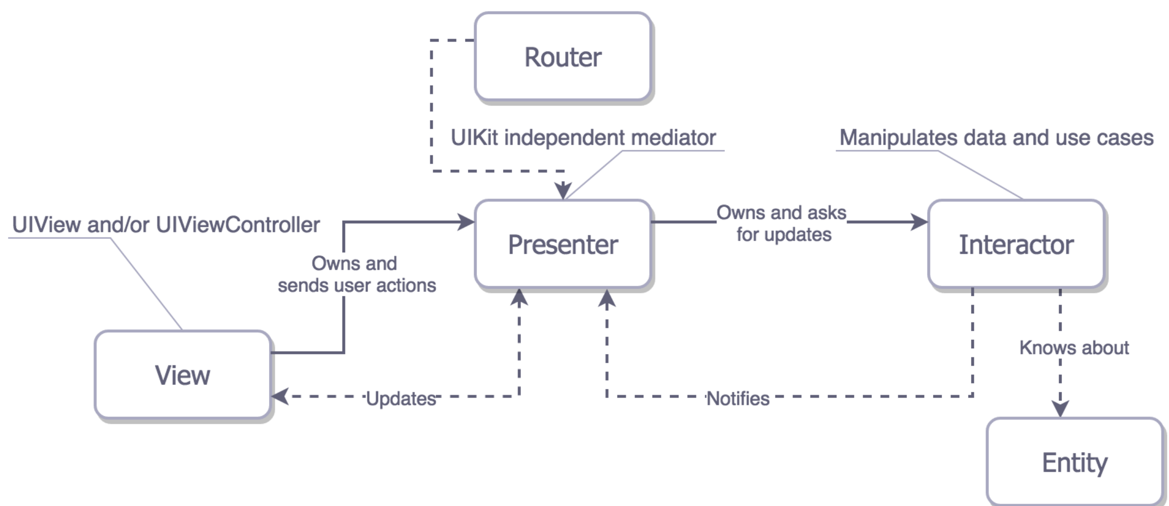
Novinkou v tomto návrhu jsou bindings (vazby). Tyto vazby nejsou mezi View a Model objektem, ale mezi View a ViewModel objektem. ViewModel je nezávislá reprezentace View a jeho stavu. ViewModel vyvolává změny v Modelu a mění se podle změn Modelu. K realizaci vazeb je nejideálnější používat knihovnu ReactiveCocoa respektive ReactiveSwift. Jednodušeji se dají vazby vytvářet pomocí Callbacků a didSet property. MVVM kombinuje vlastnosti různých přístupů, je dobře testovatelné a vyžaduje méně kódu díky vazbám než například MVP [26, 28].



Obrázek 11 – MVVM [26]

3.4.4 Viper

Tento návrh zvedá dělení zodpovědností na další úroveň. *VIPER* (View Interactor Presenter Entity Router) dělí odpovědnosti na pět vrstev (ilustrace na obrázku 12).



Obrázek 12 – VIPER [26]

- View – Reprezentuje View objekt jako v předchozích architekturách.
- Interactor – Obsahuje logiku spojenou s daty nebo networkingem (komunikaci přes internet), jako je vytváření instancí nebo zpracování dat ze serveru.
- Presenter – Obsahuje logiku spojenou s UI a provolává metody na Interactor.
- Entities – Čisté datové objekty.
- Router – Odpovědný za přechody mezi jednotlivými moduly.

Za modul v této architektuře se dá považovat prakticky cokoliv, od jedné obrazovky po celý use case (funkce) dané aplikace.

VIPER má jednoznačně nejlepší rozdělení odpovědností a tím i dobrou testovatelnost. Na druhou stranu je náročný na vývoj i množství kódu, jelikož i pro třídy s malou odpovědností se musí vytvořit daný interface (protocol). V případě této architektury je zapotřebí si velice dobře rozmyslet, zda je to pro danou aplikaci nutné a nebude to jen zpomalovat vývoj [26].

3.4.5 Souhrn kapitoly

Na závěr je potřeba zmínit, že striktní dodržování jedné nebo druhé architektury není ku prospěchu projektu. V praxi je běžné kombinovat různé architektury dohromady v určitých případech, a tím tak docílit ideálního rozložení odpovědností a velikosti kódu. V kapitole byly popsány základní druhy architektur běžně využívaných při tvorbě mobilních aplikací.

3.5 Git

Pod pojmem Git se nachází verzovací kontrolní nástroj, který je primárně určen pro softwarový vývoj. Účelem je kromě samotné zálohy souborů, umožňovat práci více lidem na stejném projektu. Projekt je možné členit na jednotlivé vývojové větve (podle funkcionality, daného člověka nebo jiného kritéria) a tyto větve následně spojovat do jednotného celku. Dále je možné sledovat změny provedené v jednotlivých commitech (příspěvcích) do daného projektu nebo se vracet na určitý příspěvek. Git lze v dnešní době považovat za standartní verzovací nástroj pro softwarový vývoj. Téma Gitu je velice obsáhlé, a proto tato kapitola popisuje jen zjednodušený princip a jeho roli v uplatnění při mobilním vývoji [10, 29].

3.6 CocoaPods a knihovny třetích stran

Jedná se o dependency injection manager (manažer, který řeší správu knihoven třetích stran pomocí vkládání závislostí) pro Swift a Objective-C projekty. Obsahuje přes 40 tisíc knihoven třetích stran, které jsou využívány ve více než 2.9 miliónech aplikacích [30].

3.6.1 Instalace a práce s CocoaPods

CocoaPods je vytvořeno v *Ruby*. To v současné době není problém, jelikož veškeré aktuální operační systémy OS X potažmo MacOS Ruby obsahují od verze 10.7. Instalace se provede jednoduše příkazem v terminálu:

```
sudo gem install cocoapods
```

V tomto příkazu tkví celý problém instalace CocoaPods. Nyní je potřeba je implementovat do požadovaného projektu. Nejdříve je nutné se nasměrovat do složky daného projektu v terminálu:

```
cd ~/Cesta/k/projektu
```

Poté je možné provézt inicializaci souboru *Podfile*. Tento soubor bude obsahovat konfigurace pro knihovny třetích stran, které mají být do projektu importovány (ukázka Podfile bude demonstrována později). Pro vytvoření Podfilu je používán příkaz:

```
Pod init
```

Tento soubor je pak možné otevřít prakticky v jakémkoliv textovém editoru a upravovat jeho konfiguraci [30, 31]. Podfile bude vypadat přibližně takto:

```
platform :ios, '9.0'
target 'projekt' do
  use_frameworks!
  pod 'Alamofire',  :git => 'https://github.com/Alamofire/Alamofire.git', :branch =>
'swift4'
  pod 'Crashlytics'
  pod 'Fabric'
  pod 'Swinject'
end
```

Pokud je projekt tvořen v jazyce Swift je nutné explicitně říci, že je potřeba používat frameworky, jinak by při instalaci podů vznikla chyba. Za klíčovým slovem *pod* je vždy název dané knihovny případně i cesta ke specifické větvi na Gitu, nebo verzi knihovny. Konfigurace podfilu je psaná v jazyce Ruby. Pokud je vše připraveno tak jak má, stačí soubor uložit a v terminálu zpustit příkaz [30, 31]:

```
pod install
```

Tímto příkazem se stáhnou všechny dependency (závislé soubory), které dané knihovny obsahují a vytvoří se *project.xcworkspace* soubor. Od této chvíle je potřeba projekt spouštět pouze přes tento soubor, který obsahuje jak spojení projektu tak daných knihoven. Pro použití se knihovna jednoduše importuje do souboru, ve kterém je třeba ji použít například [31]:

```
import Alamofire
```

3.6.2 Základní knihovny třetích stran

V této části budou popsány nejdůležitější a nejpoužívanější knihovny třetích stran *Alamofire, SnapKit a SwiftyJSON.*

3.6.2.1 Alamofire

Alamofire je Swift knihovna pro http komunikaci používaná v aplikacích pro systémy iOS a MacOS. Je postavená na základě Apple rozhraní, nicméně řeší mnoho běžných problémů. Alamofire poskytuje request a response metody, možnosti vkládání JSON (JavaScript Object Notation) / querry parametrů, validaci kódů odpovědí a mnoho dalších vlastností. Alamofire je vystavěn nad nativní URLSession, nicméně tvorbu networking vrstvy (vrstva aplikace pro internetovou komunikaci) značně zjednodušuje [32, 33]. Základní metody Alamofire jsou:

- Upload – pro nahrání dat a souborů
- Download – Pro stahování souborů nebo pokračování již aktivního stahování
- Request – Pro ostatní http požadavky

Ukázka requestu pomocí knihovny Alamofire [33]:

```
  Alamofire.request(
    URL(string: "http://pozadovanaURL")!,
    method: .get,
    parameters: ["klic": "hodnota"])
  .validate()
  .responseJSON { (response) -> Void in
}
```

Zde je vidět, jak jednoduše je vytvořen http GET požadavek na danou URL s parametry a na to se v bloku vrátí odpověď, se kterou je možné dále pracovat. Aktuální verze knihovny požaduje [32, 33]:

- iOS 8.0+
- Xcode 8.3+
- Swift 3.1+

3.6.2.2 SnapKit

SnapKit je knihovna, která umožňuje jednoduše a přehledně vytvářet Auto Layout / GUI (grafické uživatelské prostředí) pomocí kódu místo *Interface Builderu*.

Hlavní výhody jsou:

- Rychlost editování.
- Možnost tvořit komplikované grafické komponenty jednoduše a jasně.
- Dobře čitelný change log (záznam o změnách v kódu).

Nevýhody:

- Pro začínající vývojáře hůře čitelné (návody na tvorbu GUI jsou většinou pouze pomocí interface builderu)
- Potřebná představivost (nutné pro představení si pozic prvků, jejich velikostí a další)
- Není možnost náhledu (pro kontrolu je nutné spustit aplikaci)

Ukázka GUI kódu psaná pomocí knihovny SnapKit [34]:

```
var box = UIView()
self.view.addSubview(box)
box.snp.makeConstraints { (make) -> Void in
        make.width.height.equalTo(50)
        make.center.equalTo(self.view)
    }
```

Je vytvořen UIView box. Toto view je přidáno jako subView (pod view) do view celého objektu, kterého se daný kód týká. Následně je pomocí SnapKitu použit blok pro tvorbu

constraints (vazeb). View box bude mít tedy nastavenou šířku a výšku na 50 bodů a bude centrováno na centrální pozici view dané třídy.

Takto se dají UI objekty vkládat do sebe, určovat pozice vůči svým nadřazeným objektům, sobě navzájem, display a další. Extrémní využití má SnapKit při tvorbě komplexních grafických rozhraní, které by se daly v Interface Builderu vytvořit pouze částečně a poté by se musely i tak upravovat pomocí kódu. Dále je velice výhodné použití při tvorbě animací a to tak, že v animačním bloku se za pomocí SnapKitu bude měnit některá nebo více constraint (vazeb). Objekty pak mohou různě poskakovat, měnit velikosti a další.

SnapKit odstraňuje problémy Interface Builder souborů, jako je zastaralý *.xib* nebo *.storyboard*, který se z nepochopitelných důvodů edituje už jen při otevření. Code review je tak mnohem přehlednější, jelikož vývojář kontrolující dané změny nemusí číst jistou verzi XML, ve které jsou zobrazeny změny storyboardu [34, 35, 36].

Aktuální verze SnapKit knihovny požaduje:

- iOS 8.0+
- XCode 9.0+
- Swift 3.0+

3.6.2.3 SwiftyJSON

Podobně jako Alamofire tato knihovna zjednodušuje práci vývojáře oproti standardním postupům. SwiftyJSON je knihovna pro zpracování JSON dat jednodušeji a přehledněji.

JSON (JavaScript Object Notation) je datová struktura běžně využívaná pro přenos dat. JSON struktura může být složena z čísla, textového řetězce, pravdivostní hodnoty, objetu, pole nebo prázdné hodnoty. Ke každé hodnotě je vždy daný originální klíč, podle něhož se pak data zpracovávají.

Příklad JSON struktury:

```
{
  "id": 4,
  "email": "user@email.cz",
  "firstName": "Jan",
  "lastName": "Novak",
  "profilePhoto": "http://google.com",
  "phone": "420777666555",
  "languages": [
    1,
    2
  ]
}
```

Takový JSON, tedy zařízení dostane z odpovědi v komunikaci, kterou řeší Alamofire a tato data, která byla v odpovědi získána, jsou zpracována pomocí SwiftyJSON a rozdělena do property (konstant / proměnných) a objektů tak, jak vývojář potřebuje.

Příklad zpracování JSON odpovědi pomocí SwiftyJSON:

```
let json = JSON(data: response)
let id = json["id"].intValue
let email = json["email"].stringValue
let firstName = json["firstName"].stringValue
let lastName = json["lastName "].stringValue
let profilePhoto = json["profilePhoto "].stringValue
let phone = json["phone "].stringValue
let languages = json["languages "].arrayValue
```

Do každé konstanty bude přiřazena hodnota, která je v JSON struktuře pod daným klíčem. Stejně by se k hodnotám přistupovalo, i kdyby se jednalo o objekt. V takovém případě by se vnořilo do objektu přes jeho specifický klíč a následně k jeho hodnotám přes klíče těchto hodnot.

Zpracovaná data pomocí SwiftyJSON se obvykle provádí v Model objektu nebo přímo na vrstvě síťové komunikace pomocí *parser* metody (metoda, která se stará právě o toto zpracování dat), která často vrací objekt dat, se kterým je možné dále pracovat (zobrazovat jeho hodnoty uživateli, přepočítávat hodnoty nebo je dále předávat) [37, 38].

3.6.3 Souhrn kapitoly

V první části kapitoly byla pozornost věnována nástroji CocoaPods, který se využívá pro vkládání knihoven třetích stran do iOS projektu. Dále byly popsány tři vybrané základní knihovny, které je příhodné při vývoji využívat.

3.7 Testování

Testování je nedílnou součástí procesu tvorby mobilních aplikací. Testování má mnoho úrovní, které mohou být považovány za separátní problematiky. V této kapitole je popsáno několik principů testování a jejich využití.

3.7.1 Úvod do problematiky testování mobilních aplikací

Výsledky a testované problémy se mění aplikaci od aplikace. V obecné rovině je ale nutné specifikovat předpokládané výsledky testované problematiky tak, aby naměřené zjištění bylo porovnatelné. Tyto předpoklady se týkají jak testů při vývoji jako takovém (Unit testing - viz. následující kapitola), tak i funkčního testování specifického scénáře použití nebo komunikace se serverem.

S ohledem na téma práce je nutné podotknout specifika testování mobilních zařízení. Tyto specifika je třeba brát v potaz:

- Testovat na zařízeních (různých zařízeních s různými rozměry a výkonem)
- Testovat na rozdílných operačních systémech (pokud aplikace podporuje více verzí než pouze nejnovější)

Kromě nejčastějších druhů testů, o kterých tato kapitola dále pojednává, je třeba testovat například změnu konektivity k datové síti, přerušení chodu aplikace při specifických procesech (načítaní obsahu, spuštěné kameře a dalších) [11, 39, 40].

3.7.2 Unit testy

Jak je z názvu patrné, Unit testy testují jednotlivé funkce, které něco produkují nebo modifikují. Jedná se o automatizované testy, které tvoří vývojář. Tyto testy by měly být:

- rychlé na provedení,
- nezávislé jeden na druhém – výsledek jednoho testu by neměl ovlivňovat další,
- opakovatelné – výsledek by měl být pokaždé stejný. To znamená používat statická data, která se v průběhu času nemění,
- vyhodnocující samy sebe – test by měl říct, jestli prošel nebo neprošel.

Implementováním těchto testů se dá předejít chybám při refaktoringu (přepisu) složitějších funkcí nebo ověření výsledků bez reálných dat, pokud například nejsou k dispozici.

Vývojové studio Xcode obsahuje modul XCTest, který umožňuje Unit testování. Vytvoří se tak soubory, které budou obsahovat třídy XCTestCase a testovaný bude vybraný target (cíl), kterým je ve většině případů právě tvořený projekt. Třída XCTestCase obsahuje několik metod, které je možno přepsat a upravit tak jejich chování. Za nejběžnější je možné považovat například metodu *setUp()*, které se zavolá před začátkem každého dílčího testu. Obdobně je tomu například u metody *tearDown()*, která je volána po konci každého testu [11, 41, 42].

3.7.3 UI testy

Další z možných automatizovaných testů je test uživatelského rozhraní (UI test). Tyto testy je opět možné vytvořit přímo v prostředí vývojového studia Xcode za pomocí modulu XCTest a jeho třídy XCTestCase. Využití těchto testů je omezené zejména z důvodu proměnlivého obsahu aplikací. Nejběžněji se tyto testy využívají pro testování statických

funkcionalit, jako je například přihlašování, registrace, změna uživatelského profilu a podobně. Pokud by se testovala proměnná data, jako je obsah tabulek a podobně, mohla by nastat možnost, že zde nebudou žádná nebo požadovaná data a z tohoto důvodu ani nemusí existovat daný UI prvek, se kterým by měl příslušný UI test interagovat. Tvorba testu je triviální nahrání daného průchodu aplikací, vyplněním potřebných inputů a ukončení tohoto nahrávání. Studio při tomto procesu generuje potřebný kód a zaznamenává inputy do příslušné testovací funkce. Test je zdárně dokončen, pokud dojde na konec vytvořeného scénáře průchodu aplikací [12, 41, 42].

3.7.4 UX testování

Testování user experience (uživatelské zkušenosti) je velice obsáhlé téma, zaměřující se na sledování použitelnosti funkcionalit, rozmístění UI prvků, přístupnosti aplikace. Jedná se tedy o umění vytvořit aplikaci, jejíž použití je příjemný zážitek.

V praxi se při návrhu aplikace o tyto problémy stará UX designer. V pozdější fázi vývoje, kdy je již aplikace v produkci je mnoho nástrojů, jak uživatelskou zkušenost s aplikací sledovat a testovat. Běžně se pro sledování chování uživatelů v aplikaci používá například Google Analytics (později Firebase), Appboy, Fabric a další nástroje. Tyto prostředky jsou schopné analytikům potažmo UX designerům říci, co uživatelé nevyužívají tak, jak je předpokládáno, zlomové body, kde uživatelé opouští od jednotlivých scénářů použití a podobně. Takové informace je následně možné zapracovat a upravit chování, UI nebo cokoliv, co způsobuje dle těchto nástrojů problém. Takto zapracované změny je možné aplikovat přímo na všechny uživatele, nebo je testovat například pomocí A/B testování [39].

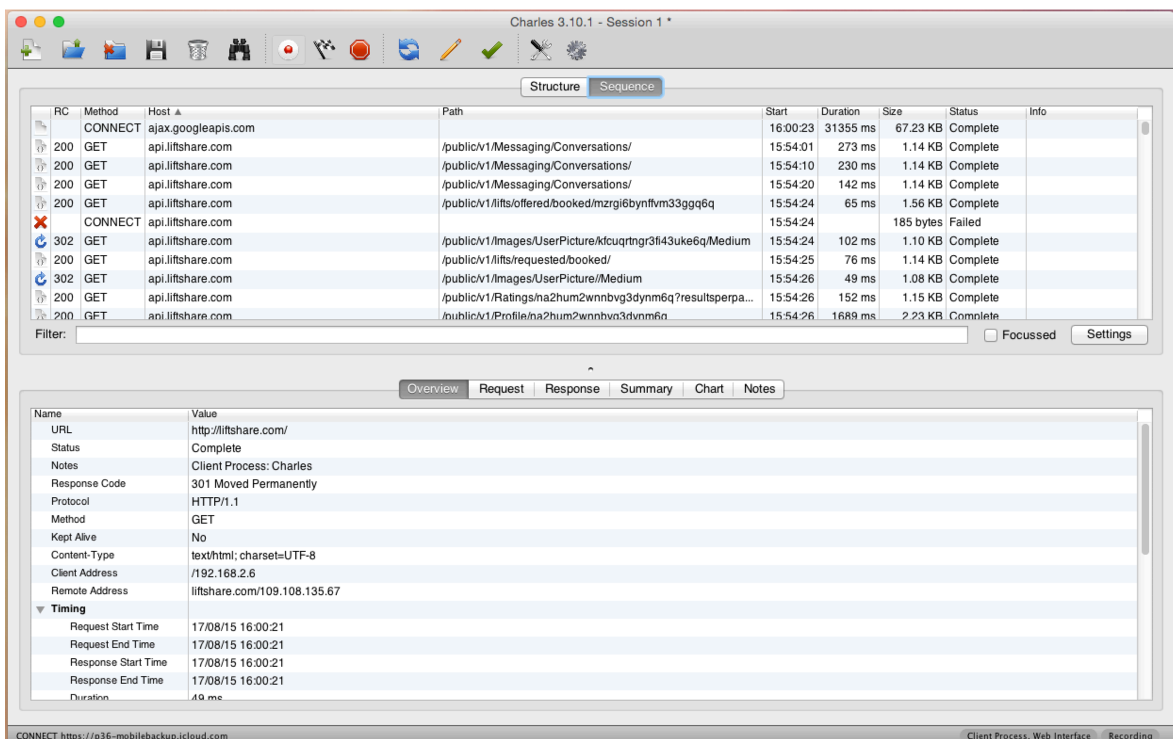
A/B testování je kontrolovaný experiment, kdy se vytvoří více variant, která jsou vydány koncovým uživatelům. Na těchto verzích se následně zaznamenávají informace o změně chování uživatelů, která je zapříčiněna danou změnou nebo reakce na novou funkčnost aplikace. Tímto způsobem se dá testovat cokoliv - od změny pozice tlačítka pro přidání nových funkcionalit aplikace [43].

3.7.5 Testování komunikace se serverem

Tento druh testování je důležitý jak v podání vývojáře tak v případě testera. Základním předpokladem je API specifikace (například na službách Apiary, Swagger). Oproti této specifikaci vývojář vytváří aplikaci a ověřuje, zda přijatá data jsou validní s daným předpisem. Pro ověřování validity dat by měl využívat tento druh testování také tester. Pokud

zjistí, že data, která aplikace přijímá nejsou / jsou v pořádku, může rozlišovat problémy, za které může koncová aplikace a za co může serverová část aplikace. To samé platí pro odesílání dat z koncové aplikace. Pokud se odesílají správná data ve správném formátu a na tento požadavek server odpovídá nepříznivě, je možné nahlásit příslušnou chybu na patřičnou osobu kompetentní k jejímu řešení.

Pro tyto účely je v běžné praxi používán například software Charles (ukázka na obrázku 13). Jedná se o software, který umožňuje sledovat všechnu HTTP / SSL / HTTPS komunikaci mezi zařízením a serverem. Je zde možné přesně vidět požadavky, odpovědi a hlavičky jednotlivých požadavků. Také čas pro jejich volání, simulovat rychlost připojení, opakované vysílání požadavků v krátkých časových intervalech a mnoho dalších informací [44].



Obrázek 13 - Charles proxy [45]

3.7.6 Testování funkcionalit

V tomto případě se tester zaměřuje výhradně na testování funkční stránky jednotlivých případů použití. Odpovídá na otázky, zda tato funkcionalita opravdu funguje, nebo jestli uživatel může dokončit požadovaný úkon. Za běžný defekt je považováno například nemožnost projít dále z důvodu neexistence vyžadovaného UI prvku (například tlačítko), nebo chybě při komunikaci se serverem [39].

3.7.7 User acceptance testing (testování uživateli)

Tento druh testů je jedním z posledních, který zastřešuje proces testování a tvorby aplikace obecně. Testery by měli být koncoví uživatelé (v mnoha případech se jedná ale o další testery ze strany klienta, pro kterého je aplikace vytvářena). V těchto testech se vyhodnocuje naprosto všechno dohromady, a to tedy použitelnost, funkcionalita, vzhled, reálné podmínky prostředí (například rychlost sítě) až po celkový dojem. Všechny poznatky, problémy a připomínky by měly být zaznamenány co nejpřesněji. Tedy přesný postup, jak chybu replikovat, podmínky, ve kterých byla chyba nalezena, druh zařízení a operačního systému na kterém se daný defekt projevil a další [46].

Pokud to podmínky dovolují, pak by uživatel na konci testování měl vyplnit dotazník, který by přinesl odpovědi na otázky [46]:

- Co se uživateli na aplikaci nejvíce líbilo.
- Co pro uživatele bylo nejvíce frustrující.
- Jak pro uživatele bylo jednoduché aplikaci používat.
- Zda by ji uživatel doporučil přátelům.
- Jak by aplikaci ohodnotil.

3.7.8 Souhrn kapitoly

V kapitole bylo pojednáváno o významnosti testování a druzích testů jako takových. Popsány byly vybrané druhy testů jako jsou například Unit, UI nebo UAT (user acceptance testing) testy.

3.8 Dokumentace zdrojového kódu

Dokumentace zdrojového kódu může být podceňovanou a ne častou činností z mnoha důvodů (například časový nátlak). Přesto je velice podstatná pro přehlednost a orientaci potenciálně nových členů týmu, kteří se mohou na aplikaci podílet. Ve vývojovém studiu Xcode existuje syntaxe, která přenese popisy funkce, parametrů, properties, protokolů a všeho dalšího do jejich globálního popisu v Quick Help inspektoru. Popisy v inspektoru se mohou lišit podle formátu komentáře.

Obecný příklad:

```
/// Popis funkce, vlastnosti
///
/// - Parameters:
///   - parametr 1: Popis parametru 1
///   - parameter x: Popis parametru x
///  - Returns: Návratová hodnota pokud nějaká je
```

Xcode je schopný generovat šablonu struktury komentáře pro označený kus kódu přes záložku Editor → Structure → Add documentation, nebo pomocí klávesové zkratky Options + CMD + /.

Jako další běžné značky jsou používány například // TODO: nebo // FIXME: které vyvolávají varování. Značka // MARK: se používá pro separování logických částí kódu v jednom souboru a je zřetelná v navigátoru zdrojového kódu. Vývojář tak jasně vidí, kde jsou v souboru například metody pro řešení nastavení UI a kde už jsou metody pro akce, které mají jednotlivé prvky přiřazené [47].

3.9 Distribuce aplikace

Tato kapitola se zabývá možnostmi distribuce vytvořené aplikace jak pro případ produkce, tak pro případy testování.

3.9.1 App store distribuce

Nejdůležitějším druhem distribuce je publikování na App store společnosti Apple, kde bude aplikace přístupná pro veřejnost. Tento proces má mnoho kroků a také řadu omezení. Z omezení potažmo předpokladů je příhodné zmínit například:

- Aplikace nesmí padat.
- Aplikace musí používat App Purchase (nástroj pro platby v aplikacích) pro finanční transakce uvnitř aplikace.
- Aplikace nesmí používat mikrofon, kameru a další služby bez povolení uživatele.

Všechny předpoklady a omezení jsou ověřovány ve schvalovacím procesu, ve kterém je aplikace reálně spuštěna a testována společností Apple.

Aby bylo možné aplikaci do schvalovacího procesu odeslat, je nutné provézt řadu předchozích kroků.

- Nutné vytvořit App ID, který jedinečně identifikuje aplikaci.
- Dále je třeba vytvořit distribuční certifikát.
- Dalším krokem je Provisioning Profile, který potřebuje ke svému vytvoření právě certifikát z předchozího kroku.

- Nutné je také registrovat zařízení, pro která je možné podepsat aplikaci daným certifikátem. Vývojář registruje své zařízení jednoduše ve studiu Xcode ve chvíli, kdy se snaží vytvořit aplikační archiv. Pro ostatní je nutné zařízení registrovat na stránce Apple Developer pomocí unikátního identifikátoru zařízení.

Pokud jsou všechny kroky splněny, je možné nahrát aplikaci do iTunesConnect služby, ve které je možné spravovat aplikace připravené k distribuci, nebo již v distribuci. Aby bylo možné provézt samotnou publikaci je nutné vyplnit další patřičné formuláře na stránce, přidat fotky aplikace, klíčová slova a další parametry.

Pro možnost publikovat aplikace do obchodu a podepisovat je, je nutné mít zaplacen vývojářský profil. Tento profil stojí 99 amerických dolarů ročně [48, 49].

3.9.2 Test Flight

Jak z názvu vyplývá, jedná se o distribuci aplikace pro testování. Taková distribuce se provádí opět přes službu iTunesConnect. Z tohoto plynou stejné předpoklady, jako je tomu u distribuce produkční verze, s tím rozdílem že formality jako jsou popisy, fotky aplikace a další není nutné vyplňovat. Je tedy třeba vytvořit odpovídající App ID, certifikát, provisioning profil a mít registrovaná zařízení právě pro testery [48, 49].

3.9.3 Enterprise distribuce

Tento druh distribuce je určen primárně pro vnitrofiremní účely potažmo testování. Jedná se o účet, kde hlavní výhodou je možnost distribuce jakýchkoliv verzí, bez nutnosti definovat zařízení, pro které je daný certifikát validní. Takto podepsaná aplikace pak může být rozšířena neomezenému počtu lidí, kteří ovšem nejsou považováni za veřejnost. Pokud by takto podepsaná aplikace byla dostupná široké veřejnosti, například pomocí odkazu na přístupných webových stránkách, je možné, že společnost Apple tento účet zablokuje. Dalším specifikem je, že přes tento účet není možné distribuovat aplikace do App Storu široké veřejnosti. Na to je nutné mít klasický vývojářský účet. Členství v Enterprise programu stoji 299 amerických dolarů ročně [48, 49].

3.9.4 Další možnosti distribuce testovacích verzí

Kromě oficiálních cest k distribuci aplikací, je značné množství nástrojů třetích stran, které umožňují distribuce testovacích aplikací. Opět zde ale platí předpoklady App ID, certifikát, provisioning profil a registrovaná zařízení. Jedním z nich je například služba

Fabric.io, kde podobně jako v Test Flightu, je možné vydávat testovací verze pro testery, kteří mají k projektu přístup. Obdobným nástrojem je také HockeyApp, která plní stejný účel. Obě tyto služby obsahují také analytické nástroje [50, 51].

3.9.5 Souhrn kapitoly

Tato kapitola obsahuje pohled na možnosti distribuce vytvořené aplikace, jak pro účely produkční, tak pro účely testování. Jsou zde zmíněny možnosti oficiální distribuce pomocí služeb společnosti Apple, tak i možnosti testovacích distribucí pomocí nástrojů třetích stran.

4 Vlastní práce

Praktická část práce se zabývá implementací poznatků z teoretické části práce a praxe při tvorbě reálné aplikace. Tato aplikace bude splňovat funkci očkovacího průkazu.

4.1 Specifikace požadavků

Pro řešení problematiky je třeba specifikovat zadání požadavků, které by aplikace měla splňovat. Jelikož se jedná o očkovací průkaz, který by mohl nahradit papírový záznam, hlavní výhodou takového elektronického záznamu musí být nutně možnost ukládání dat mimo lokální zařízení a automatické vytváření událostí v kalendáři. Z toho plyne několik následujících požadavků, které by měla aplikace obsahovat.

- Možnost registrace / přihlášení uživatele emailem a heslem (ověřované proti reálné autentifikaci).
- Seznam následujících a již proběhlých očkování (načtených z databáze podle uživatele).
- Uživatelský profil s dostupnými osobními informacemi.
- Možnost přidat nový očkovací záznam.
- Možnost vytvořit událost do kalendáře.
- Jednoduchý design a ovládání.

4.2 Analýza požadavků

V analýze požadavků budou rozebrány předpoklady pro tvorbu dané aplikace, jednotlivých požadavků a následně jejích časový odhad pracnosti.

4.2.1 Specifikace základních parametrů

Jelikož se tato práce zabývá tématem mobilního vývoje pro platformu iOS, bude tedy požadavkem vytvořit aplikaci pouze pro tento operační systém. Dále je stanoveno, že podpora aplikace bude pouze pro nejnovější operační systémy iOS 11.x a to jen na mobilních zařízeních iPhone. Tímto předpokladem odpadá nutnost úprav aplikace pro tablety a malá zařízení iPhone 4S a starší. Dále se tímto rozhodnutím teoreticky zjednodušuje tvorba z pohledu přístupu k nejnovějším API, které daný operační systém nabízí. Nevýhodou takového základního předpokladu může být nižší uživatelská základna v případě, že by se

jednalo o produkt určený k produkčnímu nasazení do App Store. Jazykem pro vývoj je stanoven nejnovější Swift 4.

4.2.2 Analýza jednotlivých požadavků

Vytvoření uživatele a autentifikace bude provedena pomocí *Firebase Authentication* od společnosti Google. Jedná se o nástroj, který vytvoří uživatele podle žádané metody (v tomto případě pomocí emailu a hesla), vytvoří o uživateli základní záznam a vygeneruje UID (uživatelský identifikátor). Uživatel přihlašující se do aplikace bude pokaždé identifikován proti této službě.

Data o uživateli budou následně uložena pomocí služby *Firebase Database*. V této databázi budou uloženi jak uživatelé, tak jejich očkovací záznamy a možnosti očkování (vakcinace) jako takové. Detailněji bude tato služba popsána v pozdějších kapitolách této práce. Uživatelský profil bude vytvořen na základě zadaných informací.

Předposledním z požadavků je vytvoření události do kalendáře. V tomto případě k tomuto účelu bude využita nativní knihovna, která vytvoří celodenní záznam do uživatelova kalendáře v zařízení.

Tvorba UI bude realizována pomocí knihovny *SnapKit*, která umožňuje tvorbu UI prvků pomocí kódu. Knihovna bude využita z důvodu přehlednosti a jasnosti chování jednotlivých prvků a možnosti tvorby komplexnějšího grafického rozhraní v budoucnu. Další možností je použít takzvané StoryBoards (posloupnost obrazovek vytvářených v grafickém editoru). Tato možnost je obecně nevhodná z důvodů nepřehlednosti, generování zbytečných změn ve struktuře upravovaného souboru a tak následných konfliktů při práci více vývojářů na jednom projektu. Poslední možností je využití Xib souborů, které reprezentují podobně jako u StoryBoardu grafickou podobu jednotlivých obrazovek potažmo UI prvků. Tato metoda je o něco vhodnější než volba StoryBoardů, jelikož negeneruje zbytečný kód a přechody mezi jednotlivými obrazovkami řeší vývojář ručně pomocí patřičných příkazů v kódu. Při tvorbě komplexnějších prvků ale opět nastává nepřehlednost a zmatek v jednotlivých souborech, který je nutné řešit.

4.2.3 Časové odhady požadavků

V následující tabulce jsou uvedeny odhady na jednotlivé požadavky, které je nutné vypracovat pro vytvoření funkční aplikace daného typu.

Možnost registrace / přihlášení uživatele emailem a heslem (ověřované proti reálné autentifikaci)	8 hodin
Seznam následujících a již proběhlých očkování (načtených z databáze podle uživatele)	12 hodin
Uživatelský profil s dostupnými osobními informacemi	4 hodiny
Možnost přidat nový očkovací záznam	6 hodin
Vytvořit událost do kalendáře	3 hodiny
Design	10 hodin

Tabulka 1 - Časové odhady požadavků

4.2.4 Souhrn kapitoly

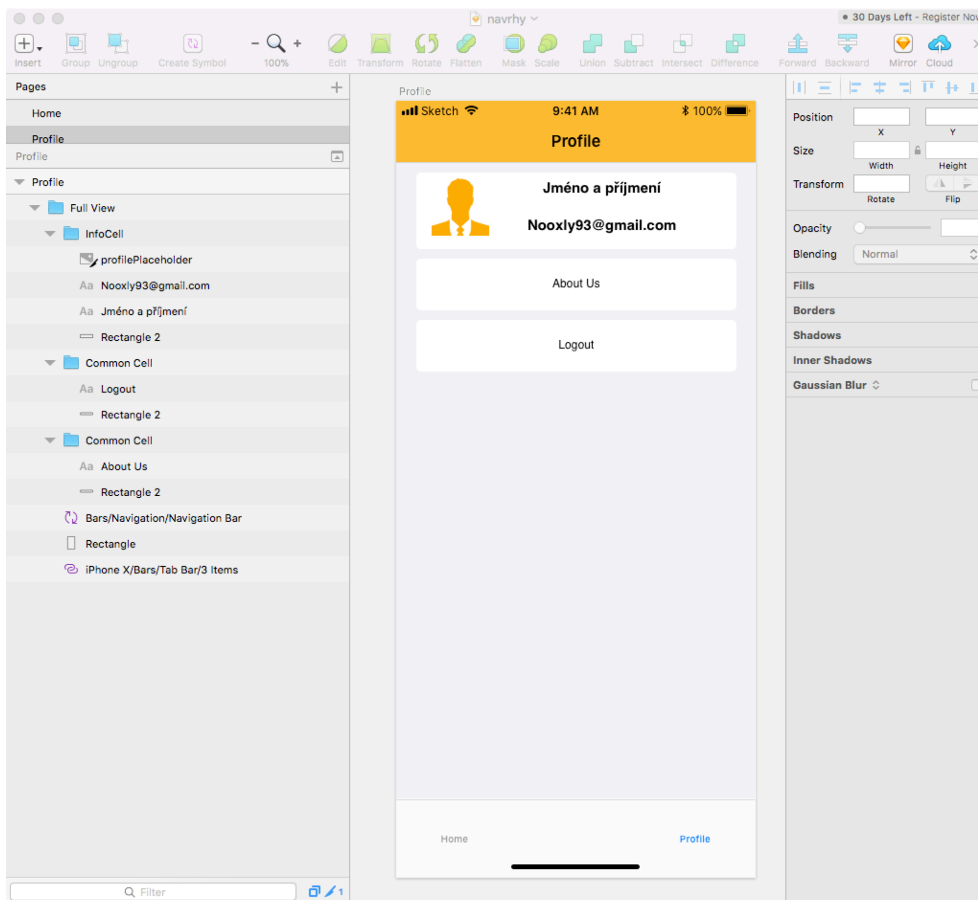
Kapitola popisuje specifikaci základních parametrů projektu. Dále jsou v kapitole rozebrány postupy řešení jednotlivých požadavků. Kapitola je zakončena časovým odhadem pracnosti realizace jednotlivých požadavků.

4.3 Návrh UI

Tvorbu grafických podkladů by měl mít v praxi na starost grafik, nejlépe se specializací na mobilní zařízení. Tato kapitola obsahuje ukázku tvorby UI v softwarovém nástroji *Sketch* a export do služby *Zeplin.io*.

4.3.1 Tvorba podkladů v nástroji Sketch

Jak již bylo zmíněno v teoretické části, kromě Photoshopu se na tvorbu grafických podkladů aplikací hojně využívá nástroj Sketch. V tomto případě je zde modelovaná každá obrazovka. Výhoda použití daného softwaru je v již částečně existujících grafických objektech, které jsou identické s nativními objekty operačního systému iOS.

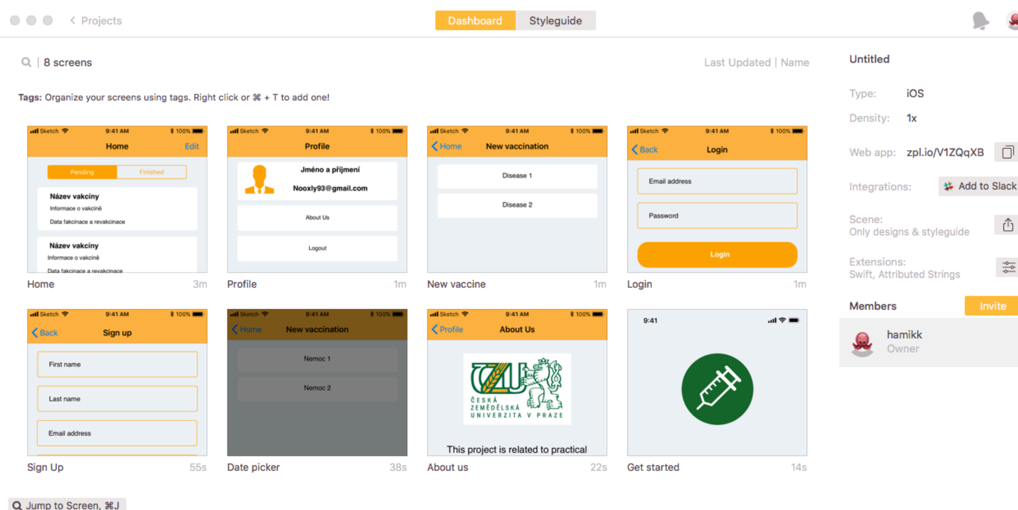


Obrázek 14 – Sketch ukázka

Z obrázku 14 je vidět, jak je obrazovka složena. Celá obrazovka je jeden Artboard, který obsahuje složku prvků tvořící obrazovku Profile. Když je prvek vymodelovaný, je možné ho jednoduše duplikovat jako je tomu například u prvku *Common Cell*, který je přepoužitý stejně, jako je tomu v reálné aplikaci. Obrázek, který je v horní buňce InfoCell je označen jako *exportable*, což znamená že bude moci být exportovaný v různých pro iOS přívětivých velikostech, například v nástroji Zeplin.io viz. následující podkapitola. Obdobně jsou vytvořeny i ostatní obrazovky aplikace.

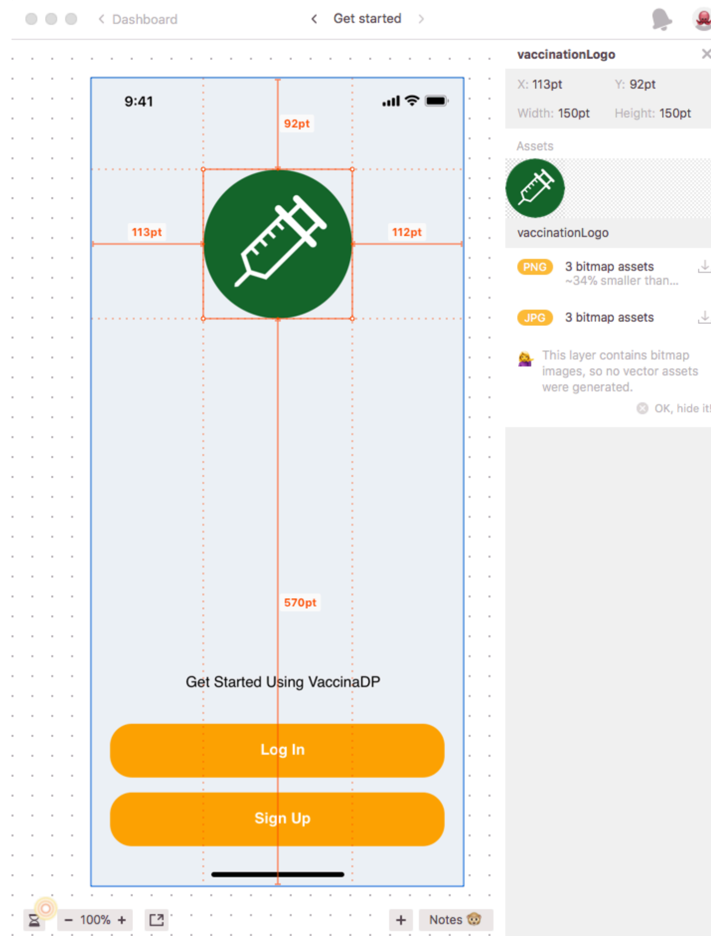
4.3.2 Zeplin.io

Pro ideální předání grafických podkladů v mobilním vývoji příslušným vývojářům je nástroj Zeplin. Uživatelé tohoto nástroje nepotřebují mít znalosti Photoshopu nebo Sketche. Vše je lehce dostupné a viditelné. Na obrázku 15 je přehled všech obrazovek v daném projektu. Kromě toho také seznam členů projektu, kteří mohou obsah vidět, komentovat nebo editovat.



Obrázek 15 - Zeplin dashboard

Na jednotlivých obrazovkách jsou pak k nalezení rozměry jednotlivých prvků, vzdálenosti prvku od prvků sousedních, nebo hranic obrazovky. Pro textový obsah je zřetelný font a velikosti. Dále přesné konfigurace barev. Pokud jsou na obrazovce assety (obrázky), je možné je exportovat jedním kliknutím ve formátu a velikosti, který pokud byl správně vytvořen v softwaru Sketch, stačí pouze vložit do projektu ve vývojovém prostředí Xcode. Ukázka první obrazovky aplikace na obrázku 16.



Obrázek 16 - Zeplin Get started obrazovka

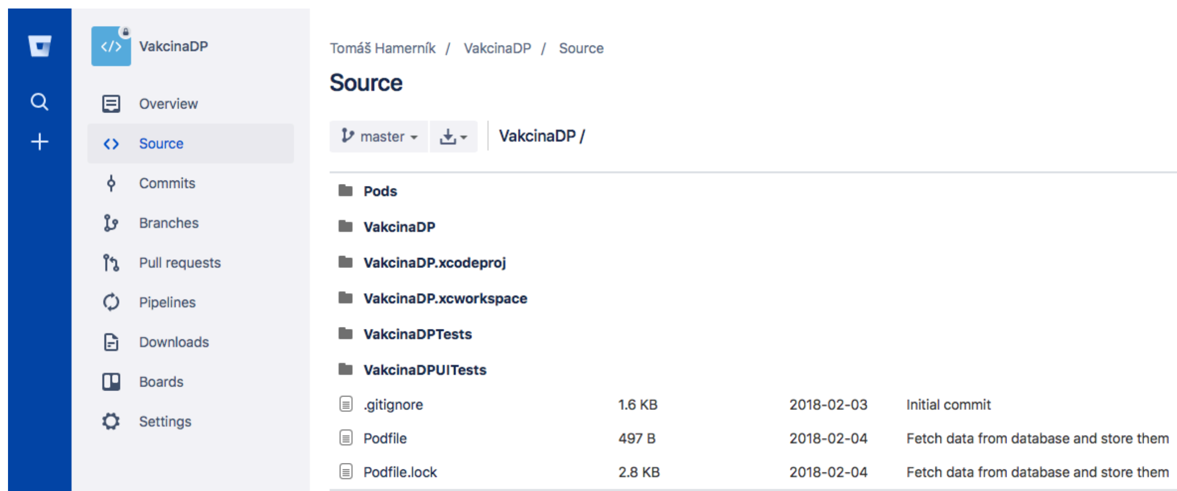
4.3.3 Souhrn kapitoly

Kapitola popisuje tvorbu a ukázku grafických podkladů pomocí softwarového nástroje Sketch a následnou distribuci těchto podkladů pomocí nástroje Zeplin.io.

4.4 Git

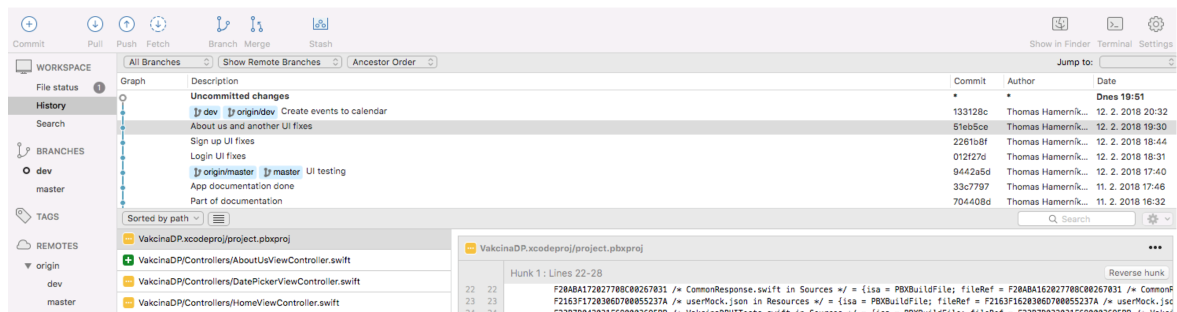
Pro verzování a vývoj projektu byl použit nástroj bitbucket.org (viz obrázek 17) od společnosti Atlassian. Jedná se o gitový nástroj pro verzování a správu projektů s pokročilými

možnostmi propojení do dalších systémů společnosti Atlassian jako je například JIRA.



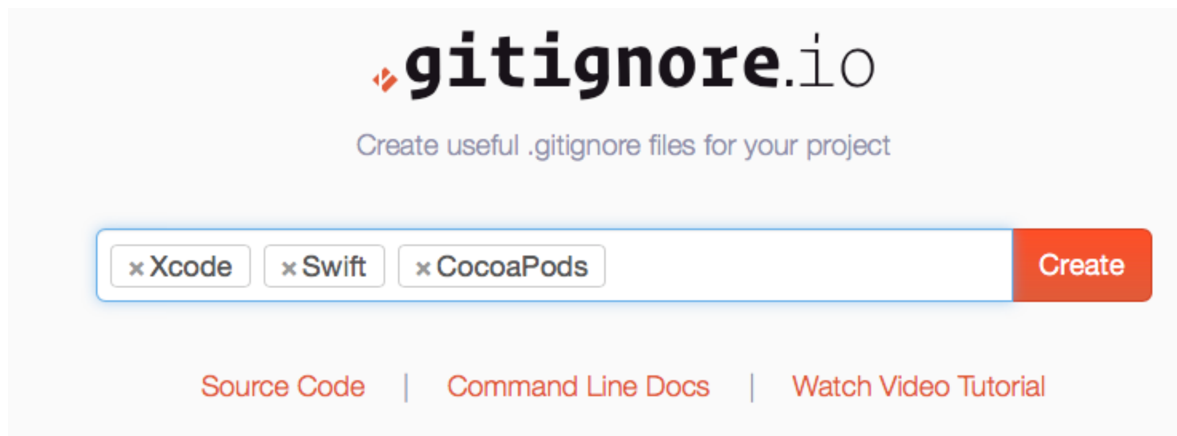
Obrázek 17 - Bitbucket

K ovládání byl použit klient *SourceTree* (viz obrázek 18), který supluje nutnost znát příkazy pro využívání Gitu, které by jinak bylo nutné znát a využívat v příkazové řádce (terminálu). Dále poskytuje přehledné rozhraní pro inspekci jednotlivých commitů, větví, konfliktů a mnoho dalších funkcí.



Obrázek 18 - Source tree

Pro to, aby se do gitového repozitáře nevpisovaly zbytečné soubory, které generuje jak studio Xcode tak i Swift nebo nástroj Cocoapods, je příhodné používat soubor *.gitignore*, který obsahuje nastavení, která definují, co bude a co nebude do repozitáře posíláno. Pro jednoduchou tvorbu takového nastavení je použita stránka gitignore.io (obrázek 19), která podle požadovaných parametrů vše připraví.



Obrázek 19 - gitignore.io

4.5 Volba architektury

Volba architektury obecně záleží na využití předpokládaných technologií a rozsahu potažmo složitosti aplikace. V nejjednodušších aplikacích se dá aplikovat MVC, které ale i v těchto případech může působit chaoticky a nepřehledně. Architektura VIPER je na druhou stranu překomplikovaná pro dané zadání aplikace, a proto také není nutné využívat takto komplexní řešení. Ideálním možným je tedy architektura založená na MVVM a rozšířená o další servisní vrstvy. Tím se docílí separace pravomocí v rámci jednotlivých tříd, modulů aplikace. Odpadne tak problém nepřiměřeně velkých a ultimátně odpovědných controllerů (tříd, které ovládají jednotlivé obrazovky), které zodpovídají za vše, co se děje.

Realizace je vytvořena pomocí protokolu, který definuje množinu vlastností a metod. Tento protokol je splňován ve třídě viewModelu, kde se nachází jeho implementace spolu s případnými dalšími úpravami dat. Tento protokol je předán controlleru pomocí dependency injection (vkládání závislostí) při inicializaci daného controlleru. Ten tedy pouze přistupuje k metodám a vlastnostem daného viewModel protokolu a reaguje na jejich výsledek, a to většinou změnou uživatelského rozhraní nebo přechodem na jinou obrazovku. Nestane se tak, že by se veškeré zásadní operace odehrávaly přímo v controlleru jako takovém.

Příklad:

- VaccineModeling protokol:

```
protocol VaccineModeling {

    func writeVaccineToDatabase(vaccines: [VaccineRecord]?, completion: @escaping (_
response: CommonResponse) -> Void )
}
```

- VaccineModel třída implementující protokol:

```
class VaccineModel: VaccineModeling {
   func writeVaccineToDatabase(vaccines: [VaccineRecord]?, completion: @escaping
(CommonResponse) -> Void) {
        guard let user = StorageHelper.user else {
            completion(.failure)
            return
        }
       databaseService.writeUserToDatabase(user: User(id: user.id, firstName:
user.firstName, lastName: user.lastName, email: user.email, vaccineArray: vaccines))
        completion(.success)
    }
}
```

- NewVaccineViewController který má pomocí dependency injection vložený protokol modelu:

```
class NewVaccineViewController: BaseViewController {
   private let viewModel: VaccineModeling
init(viewModel: VaccineModeling) {
        self.viewModel = viewModel
        super.init()
    }
}
```

4.6 Struktura projektu

Projekt je separován do několika logických skupin, které vycházejí z principu architektury a separování odpovědností jednotlivých tříd (struktur) a souborů. Struktura projektu vypadá přibližně následovně:

- ViewControllery - jako ovladače jednotlivých obrazovek.
- Modely - což jsou struktury reprezentující data jako taková. V tomto případě se jedná o struktury *User, Vaccine a VaccineRecord*. Tyto struktury obsahují vlastní inicializátory z JSON objektů, které v tomto případě fungují jako parsery JSON objektu a přiřadí jeho hodnoty do patřičných vlastností struktur. Pokud je to třeba, struktury obsahují také funkci, která naopak z hodnot parametrů vytvoří Dictionary (objekt, kde je ke každému klíči přiřazena hodnota) tak, aby tato struktura mohla být odeslána a zapsána do databáze jako JSON.

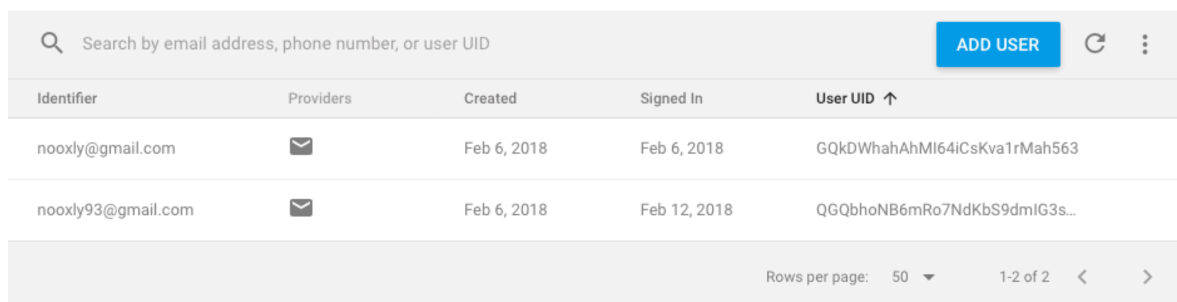
- View Modely – odstiňuje odpovědnosti od View Controllerů. Pokud je potřeba provést operace jako například stáhnout data, pak controller zavolá patřičnou funkci ve View Modulu, která provolá metodu patřičné služby, zpracuje data a předá je controlleru, který na ně reaguje.
- Services (služby) – Jedná se o třídy (protokoly), které fungují podobně jako View Modely s tím rozdílem, že řeší pouze jednu specifickou činnost. V tomto projektu je například *AuthService*, která řeší pouze přihlášení, odhlášení a registraci uživatele. Dále je zde k nalezení například *DatabaseFetchService*, která se stará o přístup k databázi, případně o zápisy do databáze. Tyto služby jsou pomocí dependency injection vloženy jako závislost při vytváření View Modelů.
- Views – jedná se o ručně specifikované objekty (například tlačítka, která zobrazují načítání), nebo buňky tabulek a další.

4.7 Realizace požadavků

Tato kapitola se věnuje jednotlivým požadavkům a jak jsou tyto požadavky realizovány s praktickými ukázkami.

4.7.1 Autentifikace

Jelikož aplikace nemá vytvořenou vlastní serverovou část a v tomto případě aplikace nevyžaduje složité operace, bylo příhodné zvolit službu, která by požadovanou funkci umožňovala. Pro tyto účely byla zvolena služba FirebaseAuth. Jedná se o nástroj od společnosti Google, který umožňuje vytvářet uživatele a ověřovat přístup vůči již vytvořeným uživatelům. Každý uživatel má pak jedinečný identifikátor, který je možné využívat k dalším funkcím, jako například přiřazování dat jednotlivým uživatelům. Na obrázku 20 je příklad vytvořených uživatelů ve Firebase konzoli.



Obrázek 20 - Firebase Authentication

Pro tyto účely byla vytvořená samostatná služba AuthService, která se stará o přístup k danému modulu služby Firebase. Tyto záznamy byly vytvořeny pomocí metod, které jsou přístupné v daném Firebase modulu - viz následující ukázka:

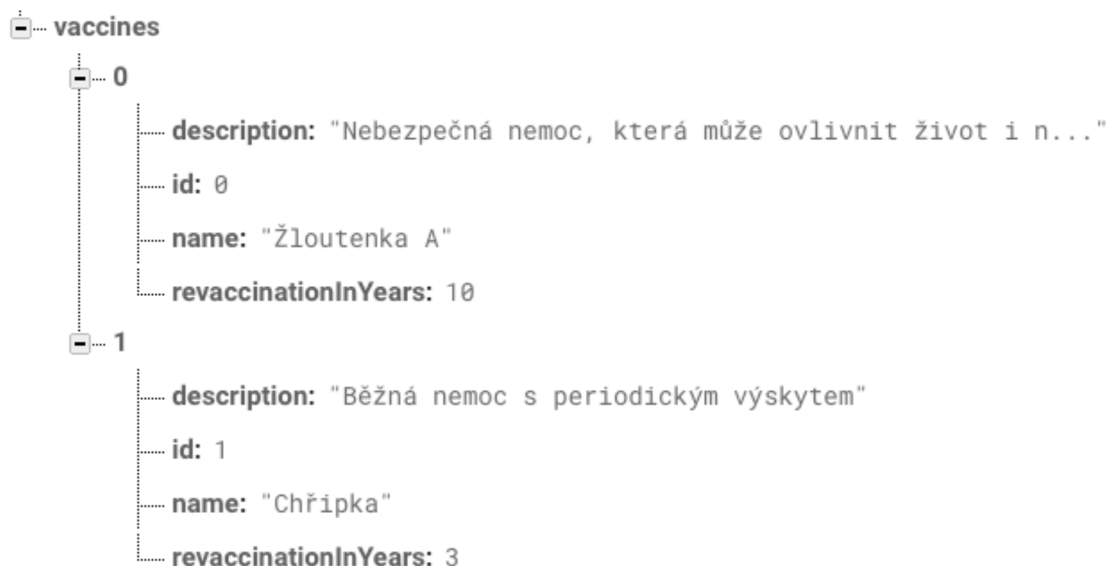
```
func signUp(with email: String, password: String, firstName: String, lastName: String,
completion: @escaping (_ response: CommonResponse) -> Void) {

        Auth.auth().createUser(withEmail: email, password: password) { [weak self] (user,
error) in
            if error == nil {
                if let id = user?.uid {
                    let userToWrite = User(id: id, firstName: firstName, lastName:
lastName, email: email, vaccineArray: [] )
                    userToWrite.store()
                    self?.databaseService.writeUserToDatabase(user: userToWrite)
                    completion(.success)
                } else {
                    completion(.failure)
                }
            } else {
                completion(.failure)
            }

        }
    }
```

Jedná se tedy o metodu, která registruje uživatele s emailem, heslem a jménem. Uživatel se registruje do služby pomocí `Auth.auth().createUser` metody. Tato metoda obsahuje blok (closure), ve kterém je vytvořený uživatel dostupný, případně chybu (error), pokud by se registrace nevydařila. Takto vytvořený uživatel je následně zapsán do databáze přes službu (service), která řeší tento problém (viz následující podkapitola). Data o uživateli se ještě uloží šifrovaně do klíčenky zařízení pomocí knihovny *KeychainSwift*, která pouze zjednodušuje nativní rozhraní pro její ovládání. Pokud vše proběhne v pořádku, pak se do návratového bloku propíše hodnota „úspěch" (.success). Pokud ne, pak nastává chyba (.failure). Na tyto výsledné hodnoty reaguje controller. Pokud je registrace úspěšná, může pokračovat v nastaveném chování a vpustit uživatele do aplikace jako přihlášeného. Pokud nastane chyba, pak controller oznámí chybu.

4.7.2 Uživatelské vakcinace (záznam i čtení)

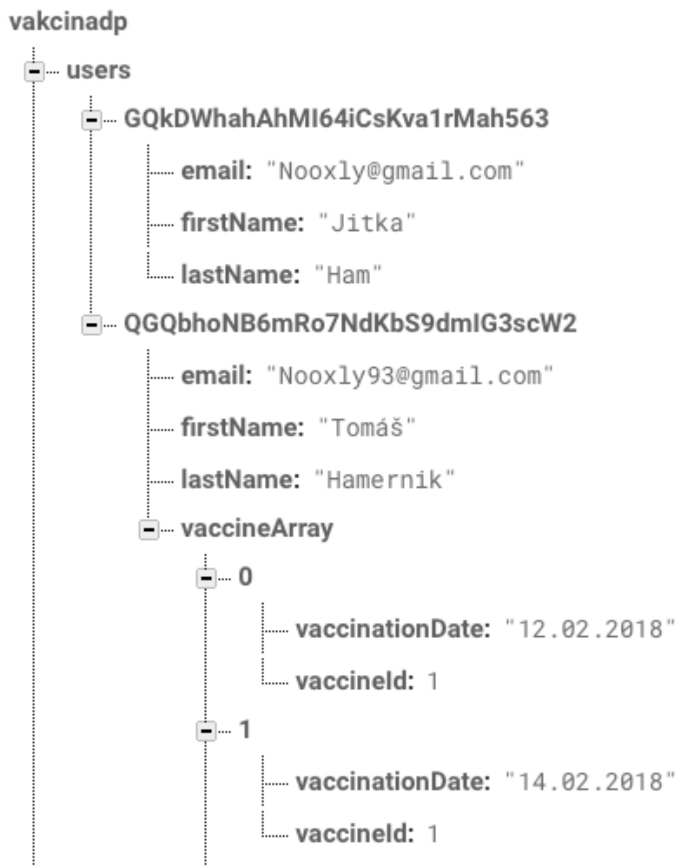
Jako v případě autentifikace uživatele i zde je použita služba Firebase, tentokrát ovšem FirebaseDatabase. Jedná se o realtime databázi (změny v ní jsou propisovány v reálném čase). Databáze je realizována pomocí jednoho JSON souboru. Pokud by struktura dat byla složitější a obsahovala vícero vrstev, pak operace v ní budou značně pomalé. Nicméně pro účely této aplikace je výkonnost této služby naprosto dostačující. Na obrázku 21 je vidět JSON struktura vakcín, které jsou pro uživatele v aplikaci dostupné. V tuto chvíli se dají další vakcíny do

databáze přidat pouze přímo. V reálném projektu by takovou funkcionalitu řešil admin v patřičném rozhraní.



Obrázek 21 - JSON struktura vaccines

Na obrázku 22 je zobrazena JSON struktura objektu uživatelů. Zde je tolik podobjektů, kolik je v aplikaci registrovaných uživatelů. Ty jsou identifikování podle UID, které jím byli přiřazeni při registraci službou FirebaseAuth. Jsou zde patrné informace, které byly do databáze zapsány spolu s registrací uživatele jako je email a jméno. Uživatelé mohou mít také sadu objektů, které nesou informace o jejich vakcinacích. Aby nebylo nutné přiřazovat celé objekty vakcín, nesou pouze informaci o ID vakcíny a datu vakcinace, které si uživatel nastavil. Tato zmenšená struktura je v projektu aplikace reprezentována modelem VaccineRecord.



Obrázek 22 - JSON struktura users

Pro zápis a načtení těchto hodnot z databáze se opět používají přístupové metody FirebaseDatabase. Tyto metody jsou ovládány ve službě (service) DatabaseFetchService. V následujícím příkladu, je vidět zápis do databáze:

```
func writeUserToDatabase(user: User) {
       let userParams = user.toParameters()
      databaseReference.child("users").child(user.id).setValue(userParams)
   }
```

Databáze odkazuje na uzel „users" a jeho poduzel, který reprezentuje uživatele podle UID viz obrázek 22. Zde je pak nastavena hodnota v podobě Dictionary (klíč: hodnota). Vložení nových hodnot pod uzel s daným UID přepíše všechny stávající hodnoty, proto pokaždé, když je třeba aktualizovat záznam o uživateli, je třeba znát jeho předchozí podobu.

Příklad: Přidávám-li uživateli vakcinaci, pak je nutné mít všechna jeho osobní data i předchozí vakcinace. K těmto datům je přidána vakcinace nová a celý tento objekt je danou metodou aktualizován (přepsán) v databázi.

V případě načítání dat z databáze je logika podobná. Pomocí reference na databázi se nastaví její pozorování pro změny. Ve chvíli, kdy se tento pozorovatel nastaví, stáhne hodnoty a zůstane připojen. Pokud by se nějaká z hodnot databáze změnila, pak stáhne hodnoty znovu. Příklad stažení dat uživatele:

```
  func getUserDataFromDatabase(for userId: String, completion: @escaping (_ response: User)
-> Void) {
        databaseReference.observe(.value) { (snapshot) in
            let jsonObj = JSON(snapshot.value as Any)
            let userJson = JSON(jsonObj["users"][userId])
            let user = User(with: userJson, id: userId)
            completion(user)
        }
    }
```

V tomto příkladu je patrné, že data (snapshot) jsou konvertovány do JSON objektu. U tohoto objektu je extrahován pouze objekt „user“, který je pod uzlem daného UID. Pomocí toho JSON objektu je následně vytvořena (inicializována) struktura User následně:

```
 init(with json: JSON, id: String) {
        self.id = id
        self.firstName = json["firstName"].stringValue
        self.lastName = json["lastName"].stringValue
        self.email = json["email"].stringValue
        let jsonArray = json["vaccineArray"].arrayValue
        var vaccineArray = [VaccineRecord]()
        for vaccineRecord in jsonArray {
            let id = vaccineRecord["vaccineId"].intValue
            let vaccinationDate = DateHelper.dateFromAPI(string:
vaccineRecord["vaccinationDate"].stringValue)
            let record = VaccineRecord(vaccineId: id, vaccinationDate: vaccinationDate)
            vaccineArray.append(record)
        }
        self.vaccineArray = vaccineArray
    }
```

Jednotlivým vlastnostem struktury User se přiřadí zpracované hodnoty JSON objektu dle patřičných klíčů. Jelikož záznamy o vakcinacích jsou pole, je nutné tímto polem projít, extrahovat hodnoty a inicializovat objekt VaccineRecord s patřičnými hodnotami, které jsou následně vloženy do pole a toto pole struktura User očekává.

4.7.3 Filtrování budoucích a již proběhnutých vakcinací

Každý uživatel ve svojí datové struktuře nese informaci o vakcinacích v podobě pole VaccineRecord. Tento objekt obsahuje ID dané vakcíny a datum vakcinace. K filtrování jsou tedy zapotřebí informace z tohoto objektu a pole dostupných vakcín.

```
func vaccineSort(with user: User, vaccines: [Vaccine]) -> (finished: [Vaccine], pending:
[Vaccine]) {
        var finished = [Vaccine]()
        var pending = [Vaccine]()

        guard let vaccineArray = user.vaccineArray else { return (finished,pending) }

        let finishedRecords = vaccineArray.filter({ (record) -> Bool in
            guard let date = record.vaccinationDate else { return false }
            return date < Date()
        })

        let pendingRecords = vaccineArray.filter({ (record) -> Bool in
            guard let date = record.vaccinationDate else { return false }
            return date >= Date()
        })

        for record in finishedRecords {
            for var vaccine in vaccines {
                if vaccine.id == record.vaccineId {
                    vaccine.vaccinationDate = record.vaccinationDate
                    finished.append(vaccine)
                }
            }
        }

        for record in pendingRecords {
            for var vaccine in vaccines {
                if vaccine.id == record.vaccineId {
                    vaccine.vaccinationDate = record.vaccinationDate
                    pending.append(vaccine)
                }
            }
        }


        return (finished, pending)
    }
```

O tento filtr se stará příslušná funkce obsažená ve VaccIceModel souboru. Vstupem této funkce je uživatel, který má potřebnou informaci o svých vakcinacích a pole všech vakcín. Záznamy, které jsou již proběhlé a které teprve nastanou, se pomocí funkce filtr a porovnání data vakcinace s datem dnešním roztřídí do příslušných polí. Dalším krokem je už pouze najít celé objekty vakcín, které přísluší danému ID v roztříděných polích záznamů. Návratovou hodnotou funkce je *tuple* (Swift struktura), složený z těchto dvou polí.

4.7.4 Uživatelský profil

Uživatelský profil je jednoduchou reprezentací informací, které uživatel poskytl při registraci do aplikace a jsou uložené v databázi. Kromě těchto informací, obsahuje profil také obrazovku About Us a Logout z aplikace. Uživatelské informace (jméno, UID, email) se ukládají v klíčence zařízení šifrovaně pomocí knihovny KeychainSwift, která je nadstavbou nad nativním ovládáním klíčenky.

4.7.5 Události v kalendáři

Vytvoření události do kalendáře se aplikace pokusí v momentě vytvoření záznamu o nové vakcinaci. Pro přístup do kalendáře je nutné požádat uživatele o oprávnění k této službě přistupovat. Pokud by uživatel přístup nepovolil, pak se události nemohou vytvářet, dokud své rozhodnutí implicitně nezměnil v nastavení telefonu, potažmo této aplikace.

```
private func addEventToCalendar(title: String, description: String?, startDate: Date,
endDate: Date, completion: ((_ success: Bool, _ error: NSError?) -> Void)? = nil) {
        let eventStore = EKEventStore()

        eventStore.requestAccess(to: .event, completion: { (granted, error) in
            if (granted) && (error == nil) {
                let event = EKEvent(eventStore: eventStore)
                event.title = title
                event.isAllDay = true
                event.startDate = startDate
                event.endDate = endDate
                event.notes = description
                event.calendar = eventStore.defaultCalendarForNewEvents
                do {
                    try eventStore.save(event, span: .thisEvent)
                } catch let e as NSError {
                    completion?(false, e)
                    return
                }
                completion?(true, nil)
            } else {
                completion?(false, error as NSError?)
            }
        })
    }
```

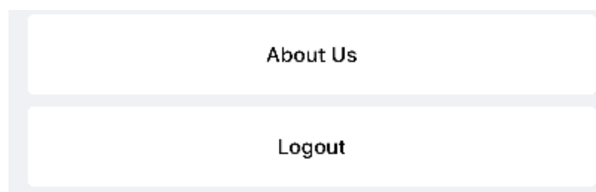
V implementaci je vidět pokus o získání přístupu a následné chování, pokud uživatel přístup udělí, nebo již udělil. Vytvoří se tedy událost s daným datem, nadpisem a popiskem a takto vytvořená událost se uloží. Uživatel pak v daný den vidí v kalendáři celodenní událost, která mu připomíná danou vakcinaci.

4.7.6 Tvorba UI

Pro tvorbu UI byla využita knihovna SnapKit. Díky této knihovně lze jednoduše vytvářet UI aplikace pomocí kódu, bez nutnosti používat Storyboard nebo .xib souboru. Výhody tohoto přístupu již byly zmíněny v teoretické části práce.

```
let contentView = UIView()
      contentView.backgroundColor = .white
      contentView.layer.cornerRadius = 4
      self.addSubview(contentView)
      contentView.snp.makeConstraints { (make) in
          make.leading.equalTo(16)
          make.trailing.equalTo(-16)
          make.top.equalTo(4)
          make.bottom.equalTo(-4)
          make.height.equalTo(54)
      }

      let label = UILabel()
      label.font = UIFont.systemFont(ofSize: 14, weight: UIFont.Weight.medium)
      label.textAlignment = .center
      label.textColor = .black
      label.numberOfLines = 0
      contentView.addSubview(label)
      label.snp.makeConstraints { (make) in
          make.edges.equalTo(0)
      }
```

V této ukázce je tvořena buňka tabulky, která je použita v profilu uživatele. Jelikož základní contentView (view pro obsah buňky) je roztaženo přes celou šířku tabulky a výšku buňky jako takové, je zde vytvořeno contentView nové, které je vloženo jako subView (view vložené v jiném view). To kromě nastavení vzhledu má také právě díky knihovně SnapKit nastavené vazby ke svému superView. Zde je řečeno, že nově vytvořené contentView má odsazení 16 bodů zleva i zprava a 4 body od spodní části a horní části svého superView. Také má nastavenou konstantní výšku. Obdobně je tomu u labelu (popisku), který je vložen už do takto zmenšeného view. Celým tímto procesem je dosažen čistější a přirozenější vzhled daného prvku viz obrázek 23.



Obrázek 23 - Profile Common Cell

4.7.7 Souhrn kapitoly

Tato obsáhlá kapitola popisuje realizaci jednotlivých požadavků na funkce aplikace. Jsou zde vysvětlené technologie použité pro vytváření uživatelů, jejich ukládání a čtení dat z databáze služby Firebase od společnosti Google. Dále je popsán princip filtrování záznamů, obsah uživatelského profilu a vytváření záznamů do kalendáře. V poslední řadě kapitola obsahuje také ukázku UI kódu pomocí knihovny SnapKit, která je použita napříč celou aplikací.

4.8 Testy

V problematice testování je v praktické části věnována pozornost Unit a UI testům, které jsou implementovány v aplikaci.

4.8.1 Unit Testy

Princip Unit testů byl již popsán v teoretické části práce. Aplikace obsahuje 3 Unit testy. První z nich testuje zpracování JSON dat ze statického JSON souboru. Druhá převod dat do parametrů typu [String: Any]. Třetí testuje filtrování vakcín na statických datech tak, aby bylo možné říci, zda filtrovací funkce fungují.

Zpracování ze statického JSON souboru:

```
guard let url = Bundle.main.url(forResource: "userMock", withExtension: "json") else {
            XCTFail("Missing file: userMock.json")
            return
        }

        let data = try Data(contentsOf: url)
        let json = JSON(data)

        self.userUnderTest = User(with: json, id: "someUserId")

        XCTAssertEqual(self.userUnderTest.email, "Nooxly93@gmail.com")
        XCTAssertEqual(self.userUnderTest.vaccineArray?.last?.vaccineId, 0)
```

Nejprve se nalezne cesta k danému souboru. Z tohoto souboru se poté vytvoří Data, která jsou vzápětí převedena do JSON objektu pomocí knihovny SwiftyJSON. Takovýto JSON objekt lze použít při inicializace struktury User, která již řeší zpracování dodaného JSON objektu. Výsledek testu je úspěšný, pokud jsou zpracované hodnoty stejné, jako očekávané výsledky.

Test pro převod do parametrů:

```
   let parametersDict = self.userUnderTest.toParameters()
        XCTAssertEqual(parametersDict["firstName"] as! String, "Tomáš")
```

Jedná se jednoduše o pokus vytvořit [String: Any] parametry z objektu uživatele vytvořeného podle statických dat.

Test filtrování:

```
   let databaseService = DatabaseFetchService()
        let vaccine1 = Vaccine(id: 0, name: "Chřipka", description: "lorem ipsum",
revaccinationInYears: 10, vaccinationDate: nil)
        let vaccine2 = Vaccine(id: 1, name: "Žloutenka", description: "lorem ipsum",
revaccinationInYears: 3, vaccinationDate: nil)
        let vaccines = [vaccine1, vaccine2]

        let viewModel = VaccineModel(databaseService: databaseService)
        let testTuple = viewModel.vaccineSort(with: self.userUnderTest, vaccines: vaccines)
          XCTAssertEqual(testTuple.finished.count, 2)
        XCTAssertEqual(testTuple.pending.count, 2)
```

V tomto případě je nutné inicializovat daný view model, který funkci obsahuje. Je také zapotřebí uměle vytvořit pole vakcín tak, aby se nemusela při testu stahovat. Následně je možné provolat funkci, kde jejím vstupem je struktura User vytvořena ze statických dat a umělé vytvořené pole vakcín. Test je úspěšný, pokud rozdělení počtu prvků v polích odpovídá předpokládané hodnotě.

4.8.2 UI testy

UI testy jsou ideálním nástrojem pro testování neměnných průchodů aplikací. V tomto případě je vytvořen UI test pro přihlášení uživatele, vytvoření vakcinačního záznamu a odhlášení z aplikace. Test je úspěšný, pokud dojde konce.

```
app.buttons["Log In"].tap()

let emailTextField = app.textFields["email"]
emailTextField.tap()
emailTextField.typeText("Nooxly93")
emailTextField.typeText("@")
emailTextField.typeText("gmail.com")

let passwordTextField = app.textFields["Password"]
passwordTextField.tap()
passwordTextField.typeText("password")
app.buttons["Login"].tap()
app.navigationBars["Home"].buttons["Add"].tap()


let tableQuery = app.tables
tableQuery.staticTexts["Žloutenka A“].tap()
app.datePickers.pickerWheels[“Today“].swipeUp()
app.buttons["Confirm"].tap()
app.tabBars.buttons["Profile"].tap()
tablesQuery.staticTexts[“Logout“].tap()
```

V ukázce je vidět, jak test zmáčkne tlačítko pro přihlášení. Následně vyplní přihlašovací údaje a přihlásí se. Jako přihlášený se dostane do nabídky vytváření vakcín, kde vybere datum a potvrdí výběr. Tím se vytvoří vakcinační záznam. V posledním kroku přejde test do záložky profilu a odhlásí uživatele.

4.8.3 Souhrn kapitoly

V kapitole je popsána implementace Unit a UI testů použitých v aplikaci. Unit testy jsou zaměřeny na zpracování JSON datové struktury a převod dat do podoby [String: Any]. Dále je popsán Unit test ověřující filtrování vakcinačních záznamů. V případě UI testu je vysvětlen postup ukázky, která simuluje průchod aplikací.

4.9 Dokumentace zdrojového kódu

V této kapitole je popsána realizace dokumentace zdrojového kódu vytvořené aplikace.

4.9.1 Realizace dokumentace zdrojového kódu

Pro dokumentaci zdrojového kódu byl použit standartní formát popsaný v teoretické části práce. Nejobsáhlejší dokumentaci mají metody protokolů view modelů nebo service (služeb). Princip je možné vidět na následující ukázce z protokolu AuthServicing, který je implementován ve třídě AuthService:

```
  /// Create new user for given input and pass data to databaseService for making a record.
    ///
    /// - Parameters:
    ///   - email: Users email
    ///   - password: Users password
    ///   - firstName: Users first name
    ///   - lastName: Users last name
    ///   - completion: Common completion block for Success or Failure
    /// - Returns: Void
    func signUp(with email: String, password: String, firstName: String, lastName: String, 
completion: @escaping (_ response: CommonResponse) -> Void)
```

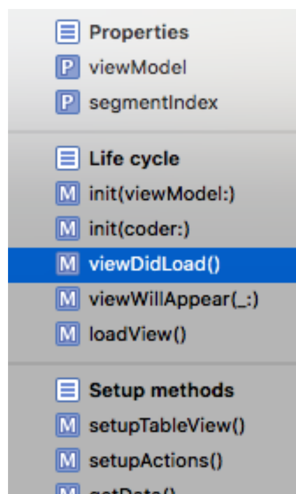
V prvním řádku bez klíčových slov se vytváří obecný popis (description). Tato metoda tedy vytváří uživatele na službě Firebase a informace o uživateli předává další service (službě), která je uloží do Firebase databáze. Parametry, které vstupují do této funkce jsou email, password, firstName, lastName a completion blok. Tento blok informuje o tom, zda byly operace v této funkci úspěšné či nikoliv. Návratová hodnota jako taková není žádná a proto Void. Takto naformátovaný komentář je v menu nápovědy pro danou metodu vyobrazen tak, jako je tomu na obrázku 24.

Obrázek 24 - Formátováný komentář

Obdobně jsou popsány další funkce a vlastnosti napříč celou aplikací. Součástí dokumentace zdrojového kódu je také členění souborů do logických částí například pomocí syntaxe // MARK: jako je tomu v následující ukázce.

```
/// Controller handlig main home view
class HomeViewController: BaseViewController {

    // MARK: - UI Objects
    weak var segmentControl: UISegmentedControl!
    weak var tableView: UITableView!

    // MARK: - Properties
    private var viewModel: VaccineModeling

    /// Computed property. Everytime this value is changed the tableView reload its content
    private var segmentIndex = 0 {
        didSet {
            self.tableView.reloadData()
        }
    }

    // MARK: - Life cycle
    init(viewModel: VaccineModeling) {
        self.viewModel = viewModel
        super.init()
    }
```

Zde je kromě obecných popisů vidět právě členění, kde jsou k nalezení UI objekty, kde vlastnosti třídy nebo začátek metod definující životní cyklus controlleru. Pomocí takto vyznačených celků se dá lépe orientovat viz obrázek 25.



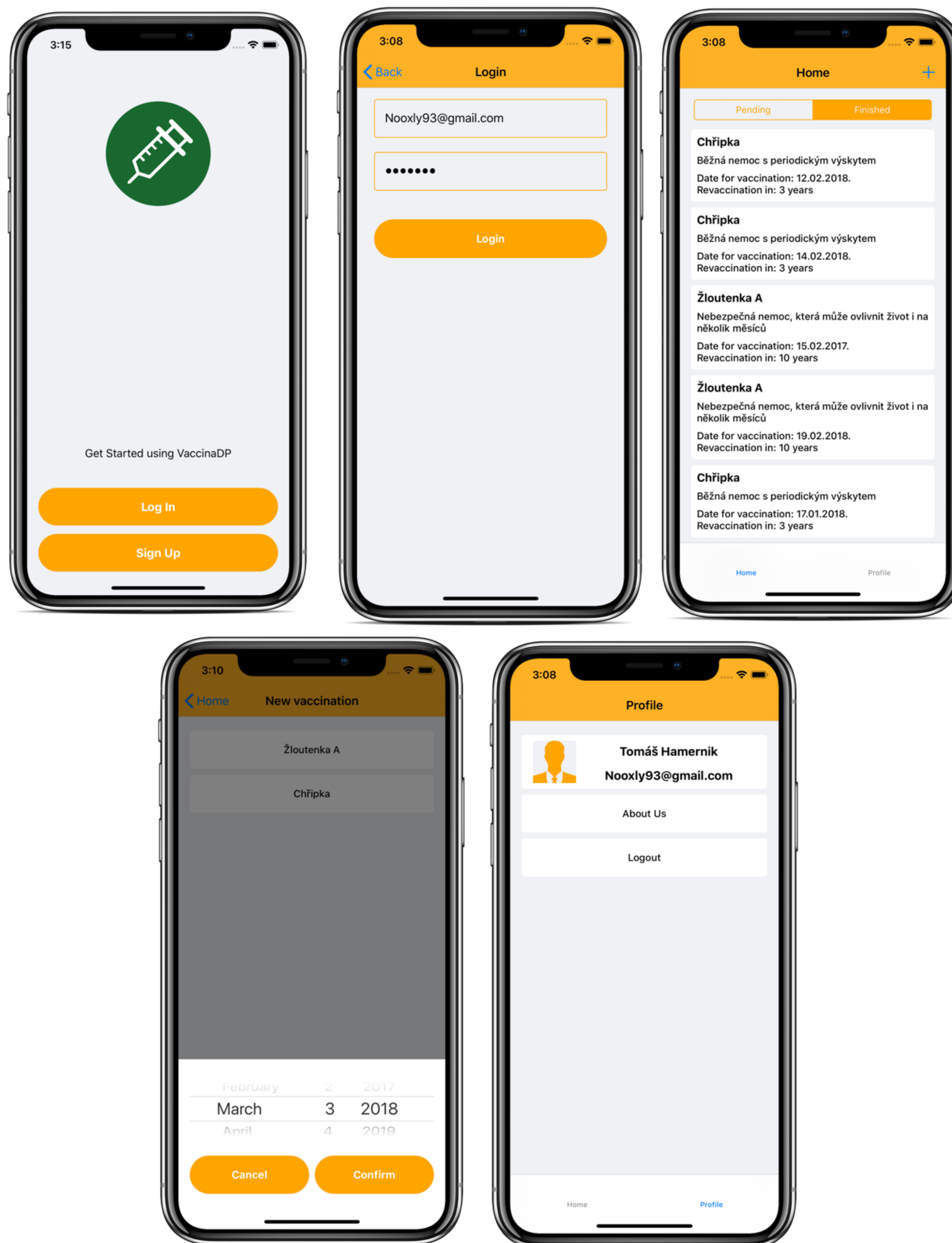
Obrázek 25 - Orientace pomocí Mark

4.9.2 Souhrn kapitoly

V kapitole byl popsán princip implementace formátovaných komentářů pro dokumentaci zdrojového kódu. Dále také jejich podoba v nápovědě a využití při orientaci v souborech projektu studia Xcode.

5 Výsledky a diskuse

Práce obsahuje teoretické podklady pro tvorbu praktické části, která se zabývá implementací postupů při tvorbě mobilní aplikace. Právě v praktické části bylo dosaženo výstupu procesu tvorby mobilní aplikace formou funkčního softwaru. Ukázka aplikace je k nalezení na obrázku 26. Zdrojový kód je přiložen jako elektronická příloha práce.



Obrázek 26 - Část obrazovek aplikace

Tvorba aplikace tedy prošla procesem od návrhu, specifikace požadavků, analýzy, návrhem grafického rozhraní aplikace až po realizaci. Pokud by se mělo jednat o produkční

aplikaci uplatnitelnou pro širokou veřejnost, bylo by nutné vytvořit vlastní serverovou část aplikace. Dále by bylo vhodné aplikaci rozšířit například o možnosti záznamů svého dítěte či dalších členů rodiny, detailní informace o vakcínách spolu s jejími účinky a další. Takto aplikovaný postup vývoje je využitelný v běžné praxi pro tvorbu mobilních aplikací na platformu iOS.

6 Závěr

Diplomová práce se zabývá procesem mobilního vývoje pro platformu iOS. Takto popsaný proces bylo následně možné aplikovat v praktické části práce, kde byla vytvořena funkční mobilní aplikace na základě získaných znalostí.

První polovina teoretické části práce se zabývá problematikou analýzy zadání a metodologií řízení softwarových projektů a jejich nástrojů. Jsou zde vysvětleny pojmy Agile a Waterfall development spolu s patřičnými výhodami a nevýhodami. Dále byla pozornost věnována nástrojům, které jsou pro řízení softwarových projektů ideální. Nejvýraznějším z nich je nástroj JIRA, který je velice funkčně obsáhlý a velmi rozšířený. V dalším kroku bylo přistoupeno k nástrojům pro tvorbu grafického rozhraní aplikací a jejich distribuci vývojářům. V dnešní době je jedním z nejběžnějších softwarových nástrojů pro tvorbu grafického rozhraní aplikací nástroj Sketch, kterému byla věnována celá kapitola. Hotové výstupy ze Sketche je třeba distribuovat programátorům. K tomu je možno využít několika nástrojů, přičemž v této práci jsou zmíněny dva, a to Zeplin a Invision. Druhá polovina teoretické části se věnuje tématům více technickým. V první řadě se jedná o architektury aplikací. Zde je věnována pozornost čtyřem nejznámějším druhům architektur, jejich výhodám a nevýhodám. V další kapitole byl také zmíněn verzovací nástroj, v tomto případě Git. Běžnou součástí mobilního vývoje jsou také aplikace třetích stran. Zde byla pozornost zaměřena na jeden specifický nástroj, kterým je CocoaPods. Tento nástroj vkládá knihovny třetích stran jako závislosti do projektu. Z knihoven třetích stran jsou vybrány tři známé knihovny, které jsou hojně využívány v praxi. Jedná se o Alamofire, SnapKit a SwiftyJSON. K těmto knihovnám jsou přiloženy i praktické ukázky jejich použití. Dalším obsáhlým tématem je testování. Zde jsou popsány základní druhy testování, využívané v běžné praxi. Větší pozornost je věnována Unit a UI testům, které je možné implementovat při vývoji v praktické části práce. Předposlední kapitola pojednává o dokumentaci zdrojového kódu. V poslední kapitole jsou rozebrány základy distribuce vytvořené aplikace.

Praktická část implementuje poznatky z teoretické části práce a praxe tak, aby alespoň částečně simulovala proces mobilního vývoje pro operační systém iOS. V první části je tak vytvořena specifikace požadavků, které by měla aplikace splňovat. Tyto požadavky následně projdou analýzou, kde je vytvořen teoretický popis řešení daného požadavku. Po připraveném řešení bylo třeba vytvořit grafické podklady, které tvoří základ pro vzhled aplikace. K tvorbě podkladů byl využit softwarový nástroj Sketch. Výstupy byly následně exportovány do nástroje Zeplin, který byl taktéž zmíněný v teoretické části práce. S takto vytvořenými

podklady bylo možné přejít k tvorbě projektu samotného, což zahrnuje verzování pomocí nástroje Git (v tomto případě specificky nástroj Bitbucket). Následovala volba architektury aplikace. Tou byla zvolena upravená verze MVVM, která odstiňuje logiku z view controllerů. V architektuře se pak nacházejí další vrstvy služeb, které se starají o specifické úkoly jako je například autentifikace uživatele, nebo komunikace s databází. Po volbě architektury bylo nutné projekt strukturovat do logických částí jako jsou například modely, služby, view controllery a další. Následně bylo možné přistoupit k tvorbě aplikace jako takové a realizaci jednotlivých požadavků. Tato realizace je detailně popsaná vždy pro každý požadavek. Jedná se například o realizaci databáze pomocí služby Firebase Database, nebo tvorbu grafického rozhraní pomocí knihovny SnapKit. Současně s vytvářením aplikace bylo vytvořeno několik Unit testů, které jsou zde detailně popsány. Dále byl vytvořen také UI test, který simuluje průchod aplikací od přihlášení, přes přidání záznamu až po odhlášení uživatele. Poslední kapitola se věnuje dokumentaci zdrojového kódu.

7 Seznam použitých zdrojů

Knižní zdroje:

1. SIMS, Ch. JOHNSON, H. L. *Scrum: a Breathtakingly Brief and Agile Introduction*. Dymaxicon, 2012. 54 s. ISBN 978-1937965044.
2. SHORE, J. WARDEN, S. *The art of agile development*. Sebastopol, CA: O'Reilly Media, c2008. 440 s. ISBN 978-0596527679.
3. SCHWARZ, D. *Jump Start Sketch: Master the Tool Made for UI Designers*. SitePoint, 2016. 150 s. ISBN 978-0994346964.
4. MORSON, S. *Designing for iOS with Sketch*. Berkeley, CA: Apress, 2015. 178 s. ISBN 978-1484214596.
5. KEUR, C. HILLEGASS, A. CONWAY, J. iOS *Programming: The Big Nerd Ranch Guide*. Atlanta: Big Nerd Ranch Guides, 2014. 560 s. ISBN 978-0321942050.
6. Gauchat, J. D. *iOS Apps for Masterminds 3rd Edition.* Mink Books, 2017. 828 s. ISBN 1979692181.
7. NEUBURG, M. *Programming iOS 11. Eighth edition*. Sebastopol, CA: O´Reilly, 2018. 1172 s. ISBN 978-1491999226.
8. INVERSEN, J. EIERMAN, M. *Mobile App Development for iOS and Android, Edition 2.0*. Prospect Press, 2017. 352 s. ISBN 978-1-943153-27-5.
9. GASTON, C. H. *Swift 3 Object-Oriented Programming - Second Edition*. Packt Publishing - ebooks Account, 2017. 370 s. ISBN 978-1787120396.
10. LOELIGER, J. MCCULLOUGH, M. *Version control with Git. Second edition*. Beijing: O'Reilly, 2012. 256 s. ISBN 978-1449316389.
11. KNOTT, D. *Hands-on mobile app testing: a guide for mobile testers and anyone involved in the mobile app business*. New York: Addison-Wesley, 2015. 256 s. ISBN 978-0134191713.
12. PENN, J. *Test iOS apps with UI Automation: bug hunting made easy*. Dallas, Texus: Pragmatic Bookshelf, 2013. 200 s. ISBN 978-1937785529.

Elektornické zdroje:

13. SELECTBS. *What is a Software Development Process?.* [online]. [cit. 2017-09-05]. Dostupné z: http://www.selectbs.com/analysis-and-design/what-is-a-software-development-process.

14. TYSON , G. *Project Planning for Successful Software Development*. [online]. [cit. 2017-09-05]. Dostupné z: http://www.informit.com/articles/article.aspx?p=26397&seqNum=4.
15. LOTZ, M. *Waterfall vs. Agile: Which is the Right Development Methodology for Your Project?*. [online]. [cit. 2017-09-07]. Dostupné z: https://www.seguetech.com/waterfall-vs-agile-methodology/.
16. SEGUETECH. *8 Benefits of Agile Software Development*. [online]. [cit. 2017-09-07]. Dostupné z: https://www.seguetech.com/8-benefits-of-agile-software-development/.
17. MYJIRA. *Jira Software*. [online]. [cit. 2017-09-10]. Dostupné z: http://www.myjira.cz/produkty/jira-software.html.
18. ASANA. *About Asana*. [online]. [cit. 2017-09-10]. Dostupné z: https://www.asana.com.
19. TRELLO. *Trello je levný, fleixibilní a vizuální způsob, jak organizovat cokoliv s kýmkoliv*. [online]. [cit. 2017-09-11]. Dostupné z: https://trello.com.
20. SKETCHAPP. *The digital design toolkit*. [online]. [cit. 2017-09-11]. Dostupné z: https://www.sketchapp.com.
21. DESIGNCODE. *Learn Sketch 3*. [online]. [cit. 2017-09-17]. Dostupné z: http://v1.designcode.io/sketch.
22. ZEPLIN. *Build pixel perfect apps in peace*. [online]. [cit. 2017-09-17]. Dostupné z: https://zeplin.io.
23. ZEPLIN. *Simple plan, for everyone*. [online]. [cit. 2017-09-17]. Dostupné z: https://zeplin.io/pricing.
24. INVISIONAPP. *Design better. Faster. Together*. [online]. [cit. 2017-09-17]. Dostupné z: https://www.invisionapp.com.
25. DAR, H. *Architecture of mobile software applications*. [online]. [cit. 2017-10-02]. Dostupné z: https://www.slideshare.net/hassandar18/architecture-of-mobile-software-applications.
26. ORLOV, B. *iOS Architecture Patte*rns. [online]. 2015. [cit. 2017-10-06]. Dostupné z: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52.
27. APPLE INC. *Model-View-Controller*. [online]. [cit. 2017-10-06]. Dostupné z: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

28. ZABLOCKI, K. *Good iOS Application Architecture: MVVM vs. MVC vs. VIPER*. [online]. [cit. 2017-10-06]. Dostupné z: https://academy.realm.io/posts/krzysztof-zablocki-mDevCamp-ios-architecture-mvvm-mvc-viper/.
29. ATLASSIAN. *What is Git*. [online]. [cit. 2017-10-15]. Dostupné z: https://www.atlassian.com/git/tutorials/what-is-git.
30. COCOAPODS. *What is Cocoapods*. [online]. [cit. 2017-10-15]. Dostupné z: https://cocoapods.org.
31. GREENE, J. *CocoaPods Tutorial for Swift: Getting Started*. [online]. 2017. [cit. 2017-11-09]. Dostupné z: https://www.raywenderlich.com/156971/cocoapods-tutorial-swift-getting-started.
32. GITHUB. *Alamofire.* [online]. [cit. 2017-11-09]. Dostupné z. https://github.com/Alamofire/Alamofire.
33. DOUGLAS, A. *Alamofire Tutorial: Getting Started*. [online]. 2017. [cit. 2017-11-13]. Dostupné z: https://www.raywenderlich.com/147086/alamofire-tutorial-getting-started-2.
34. GITHUB. *SnapKit an auto layout DSL for iOS & OSX*. [online]. [cit. 2017-11-13]. Dostupné z: https://github.com/SnapKit/SnapKit.
35. MIRASING, A. *How to write auto layout constraints with SnapKit in iOS*. [online]. 2016. [cit. 2017-11-17]. Dostupné z: https://medium.com/cocoaacademymag/how-to-write-auto-layout-constraints-with-snapkit-in-ios-c5f95c7c695d.
36. SNAPKIT. *SnapKit an auto layout DSL for iOS & OSX*. [online]. [cit. 2017-11-17]. Dostupné z: http://snapkit.io.
37. GITHUB. *The better way to deal with JSON data in Swift*. [online]. [cit. 2017-12-02]. Dostupné z: https://github.com/SwiftyJSON/SwiftyJSON.
38. HUDSON, P. *How to parse JSON using SwiftyJSON*. [online]. [cit. 2017-12-02]. Dostupné z: https://www.hackingwithswift.com/example-code/libraries/how-to-parse-json-using-swiftyjson.
39. ARSENE, C. *17 Strategies for end to end mobile testing on both iOS and Android*. [online]. 2016. [cit. 2017-12-10]. Dostupné z: https://ymedialabs.com/17-strategies-for-end-to-end-mobile-testing-on-both-ios-and-android/.
40. GURU99. *Getting started with iOS testing*. [online]. [cit. 2017-12-10]. Dostupné z: https://www.guru99.com/getting-started-with-ios-testing.html.

41. TAM, A. *iOS Unit Testing and UI Testing Tutorial*. [online]. 2017. [cit. 2017-12-12]. Dostupné z: https://www.raywenderlich.com/150073/ios-unit-testing-and-ui-testing-tutorial.
42. ZABLOCKI, K. *Testing iOS Apps*. [online]. 2017. [cit. 2017-12-12]. Dostupné z: http://merowing.info/2017/01/testing-ios-apps/.
43. OPTIMIZELY. *iOS A/B Testing*. [online]. [cit. 2017-12-14]. Dostupné z: https://www.optimizely.com/optimization-glossary/ios-ab-testing/.
44. CHARLESPROXY. *Web debugging proxy application*. [online]. [cit. 2017-12-15]. Dostupné z: https://www.charlesproxy.com.
45. QAMAZING. *Charles*. [online]. [cit. 2017-12-15]. Dostupné z: https://qamazing.files.wordpress.com/2015/08/charles1.png.
46. SELIM, T. *Six Tips for User Acceptance Testing*. [onlilne]. 2017. [cit. 2017-12-16]. Dostupné z: https://developer.amazon.com/blogs/appstore/post/1a250865-9fe5-479a-99b2-d9f335ceb690/six-tips-for-user-acceptance-testing.
47. THOMPSON, M. COOK, N. *Swift Documentation*. [online]. 2015. [cit. 2017-12-18]. Dostupné z: http://nshipster.com/swift-documentation/.
48. APPLE. *Choosing a Membership*. [online] [cit. 2017-12-19]. Dostupné z: https://developer.apple.com/support/compare-memberships/.
49. AGRAWAL, V. *How to Submit ani OS App to the App Store*. [online]. 2017 [cit. 2017-12-19]. Dostupné z: https://code.tutsplus.com/tutorials/how-to-submit-an-ios-app-to-the-app-store--mobile-16812.
50. HOCKEYAPP. *Distribution.* [online]. [cit. 2017-12-19]. Dostupné z: https://hockeyapp.net/features/distribution/#s
51. FABRIC. *Build. Understand. Grow*. [online]. [cit. 2017-12-19]. Dostupné z: https://get.fabric.io/#