Mendelova univerzita v Brně
Provozně ekonomická fakulta

Mobilní mapová aplikace pro inventarizaci majetku na platformě iOS

Bakalářská práce

Vedoucí práce:
Ing. David Procházka, Ph.D.

Ivo Pisařovic

Brno 2016

Zde bude vloženo zadání práce

Poděkování

Chtěl bych na tomto místě poděkovat vedoucímu mé práce Ing. Davidu Procházkovi, Ph.D., za cenné rady, připomínky a podporu. Také bych chtěl poděkovat Ing. Janu Kolomazníkovi, Ph.D., a Ing. Jaromíru Landovi, Ph.D., za odborné konzultace. Na závěr velké dík patří celé mé rodině za podporu během celého studia.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Mobilní mapová aplikace pro inventarizaci majetku na platformě iOS**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 3. 5. 2016 ....

Abstract

Pisařovic, I. Mobile mapping application for property inventory on iOS platform. Bachelor thesis. Brno, 2016.

This thesis is focused on design and implementation of a mobile application that simplifies the inventory process. The application allows to create map projects that contains layers composed of geospatial objects including their properties. Important part of the thesis describes the problem of the real-time spatial data synchronisation between different devices.

Keywords

mobile application, mapping, GIS, synchronisation, REST, JSON, iOS

Abstrakt

Pisařovic, I. Mobilní mapová aplikace pro inventarizaci majetku na platformě iOS. Bakalářská práce. Brno, 2016.

Práce se zabývá návrhem a implementací aplikace, která je určena pro zjednodušení inventarizace majetku. Aplikace umožňuje vytvářet mapové projekty, které se skládají z vrstev obsahujících prostorové objekty včetně jejich vlastností. Důležitou součástí práce je popis řešení synchronizace prostorových dat mezi různými zařízeními.

Klíčová slova

mobilní aplikace, mapování, GIS, synchronizace, REST, JSON, iOS

Obsah

1 Úvod **13**
1.1 Cíl práce . . . 13

2 Rešerše **15**
2.1 Existující řešení . . . 15
2.2 Vývojové prostředí . . . 16
2.3 Základní knihovny a API . . . 17
2.3.1 Uložení persistentních dat . . . 17
2.3.2 Uživatelské rozhraní . . . 17
2.3.3 Správa závislostí . . . 17
2.4 Mapovací a polohové služby . . . 17
2.5 Prostorová data . . . 18
2.6 Síťová komunikace, synchronizace . . . 18
2.6.1 Dostupná řešení . . . 18
2.6.2 Současné serverové API . . . 19
2.6.3 Foundation framework . . . 19
2.6.4 RestKit framework . . . 20
2.6.5 Knihovny nižší úrovně . . . 20
2.6.6 Konflikty . . . 20
2.7 Shrnutí . . . 21

3 Metodika **23**
3.1 Základ . . . 23
3.2 Uživatelské rozhraní . . . 23
3.3 Synchronizace . . . 24
3.3.1 Výběr knihoven a frameworků . . . 24
3.3.2 Identifikace objektů . . . 24
3.3.3 Další speciální typy URI . . . 25
3.3.4 Základní stavy synchronizace . . . 25
3.3.5 Přehled standardních operací . . . 26
3.3.6 Přehled odpovědí serveru . . . 26
3.3.7 Diagramy synchronizačních operací . . . 27
3.3.8 Další problémové situace . . . 32
3.4 Ukládání prostorových dat v zařízení . . . 32
3.5 Mapový podklad . . . 33

4 Vlastní práce **35**
4.1 Úvod . . . 35
4.2 Objektové úložiště . . . 35
4.3 Abstrakce úložiště . . . 38
4.4 Mapování dat pro synchronizaci a export . . . 39
4.5 Kompletní synchronizace . . . 42

4.6 Synchronizační fronta 42
4.7 Manuální a periodická rozdílová synchronizace 43
4.8 Příchozí změny 44
4.9 Synchronizace obrázků 45
4.10 Geometrie 46
4.11 Uživatelské rozhraní 46
4.11.1 Uspořádání a návaznost obrazovek 46
4.11.2 Layout 47
4.11.3 Správa mapy 50
4.11.4 Měření dat 51
4.11.5 Validace dat 52
4.11.6 Dočasné atributy 53
4.11.7 Obnovování dat 53
4.12 Konfigurace 54

5 Diskuze **55**
5.1 Příklady využití 55
5.2 Náklady a přínosy 56
5.3 Technické zhodnocení 56
5.4 Další vývoj 57

6 Závěr **59**

7 Reference **61**

Přílohy **65**

A Elektronické přílohy **66**

1 Úvod

S rychlým vývojem mobilních technologií a zejména chytrých telefonů se rozvíjí možnosti využití těchto zařízení pro podnikové účely. Jednou z běžných činností realizovaných celou řadou organizací je inventarizace nemovitého majetku. V případě státní organizace se může jednat o evidování stavu silnic, dopravního značení nebo třeba svodidel. V případě města či vesnice je často nutné kontrolovat stav zeleně, veřejného osvětlení aj. Soukromé organizace potřebují zaznamenávat stav budov, oplocení, pozemků a další infrastruktury. Organizace někdy mívají tyto informace neaktuální, případně zaznamenané pouze písemně, což vede k větší chybovosti a obtížnějšímu použití takových dat. Mobilní aplikace by mohla tento proces inventarizace usnadnit a zpřesnit.

Na trhu existuje celá řada komerčních řešení, která nabízejí služby pro sběr prostorových dat na mobilních zařízeních. Jedná se především o doplňky profesionálních GIS nástrojů. Nicméně tato řešení jsou většinou velmi nákladná a jejich pořízení pro vedlejší podpůrnou činnost by mohlo být neefektivní. Dalším nedostatkem je, že většina mobilních aplikací v této oblasti je zaměřena na prosté měření. Aplikace neumožňují v reálném čase zobrazovat průběh měření na více zařízeních či uživatelsky přívětivě umožnit spolupráci více uživatelů na zaměření dat. Proto jsem se rozhodl této problematice věnovat a zaměřit se především na obousměrnou synchronizaci veškerých dat mezi více zařízeními a serverem. Díky synchronizaci tak aplikace získá přesah využití do širšího spektra odvětví a kromě prosté statické inventarizace ji bude možné použít i jako dynamickou interaktivní aplikaci např. pro sledování pozice služebních vozů, kooperativní odstraňování následků krizových situací nebo třeba interaktivní sdílení strojů na stavbě.

Tato práce volně navazuje na projekt řešený na Provozně ekonomické fakultě Mendelovy univerzity v Brně, který se zabýval vývojem mobilního geografického informačního systému pro platformu *Android*. V rámci tohoto projektu byla implementována mobilní aplikace pro platformu *Android* a také serverový backend umožňující synchronizaci dat. Na vývoji tohoto backendu jsem spolupracoval. Moje bakalářská práce staví na tomto backendu a snaží se vyřešit problematiku měření na platformě *iOS*, která má řadu specifických designových i funkčních omezení.

1.1 Cíl práce

Cílem práce je navrhnout a implementovat mapovou aplikaci pro platformu *iOS*. Aplikace by měla být navržena podle aktuálních designových principů, měla by podporovat vytváření projektů skládajících se z vrstev, vytváření a správu základních geografických objektů (bodu, linie a polygonu), zobrazení těchto objektů na mapovém podkladu, správu metadat a vlastností těchto objektů (barva, symbol). Tyto objekty bude možné pořizovat i automaticky s využitím standardních komponent pro zjišťování polohy mobilních zařízení. Práce bude také obsahovat srovnání růz-

ných přístupů k implementaci vykreslování mapového podkladu, na kterém se budou zobrazovat geografické objekty.

Důležitým dílčím cílem je srovnat možnosti synchronizace naměřených dat mezi mobilním zařízením a serverem a navrhnout řešení základních synchronizačních problémů a řešení komunikace se serverem. Pro serverovou část bude použito existující řešení, na kterém jsem se podílel v rámci předchozího projektu. Synchronizace dat z pohledu mobilního zařízení bude velmi důležitou částí této práce, protože se jedná o funkcionalitu, která není mezi podobnými aplikacemi běžná. V rámci práce navrhnu a vytvořím univerzální soubor tříd, který bude znovupoužitelný v jakékoliv aplikaci, která vyžaduje obousměrnou synchronizaci a nevystačí si se základními možnostmi synchronizace, které nabízí standardní synchronizační API.

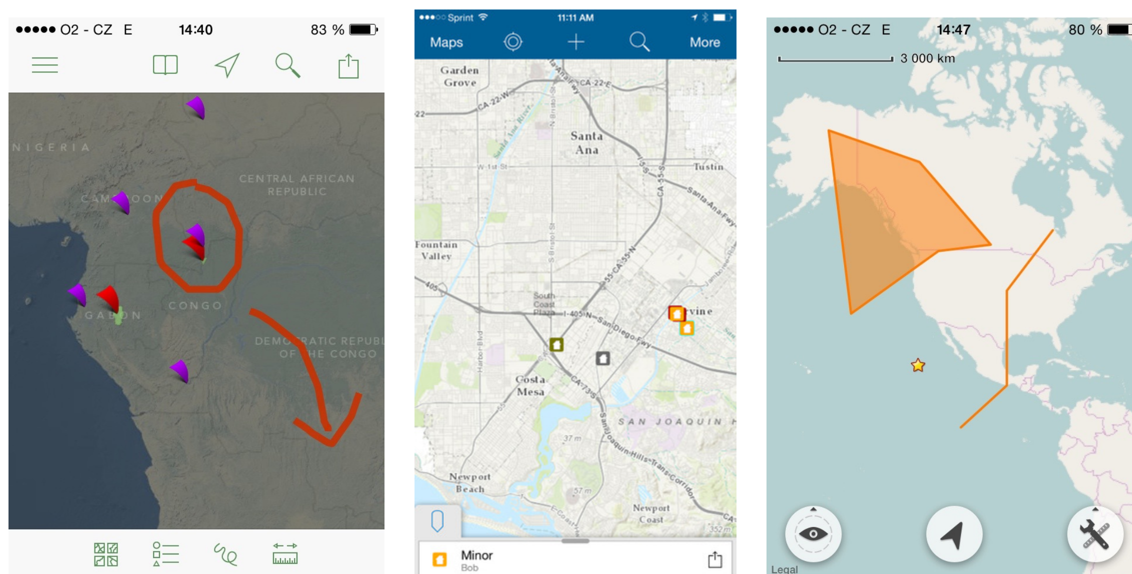
Uživatelské rozhraní i synchronizace budou následně ověřeny implementací experimentální aplikace s výše popsanými vlastnostmi. V práci bude na závěr diskutováno využití aplikaci v rámci podnikové infrastruktury a další rozvoj aplikace.

2 Rešerše

V této části práce nejprve popíši a srovnám existující komplexní řešení a následně se zaměřím na jednotlivé dílčí technologie.

2.1 Existující řešení

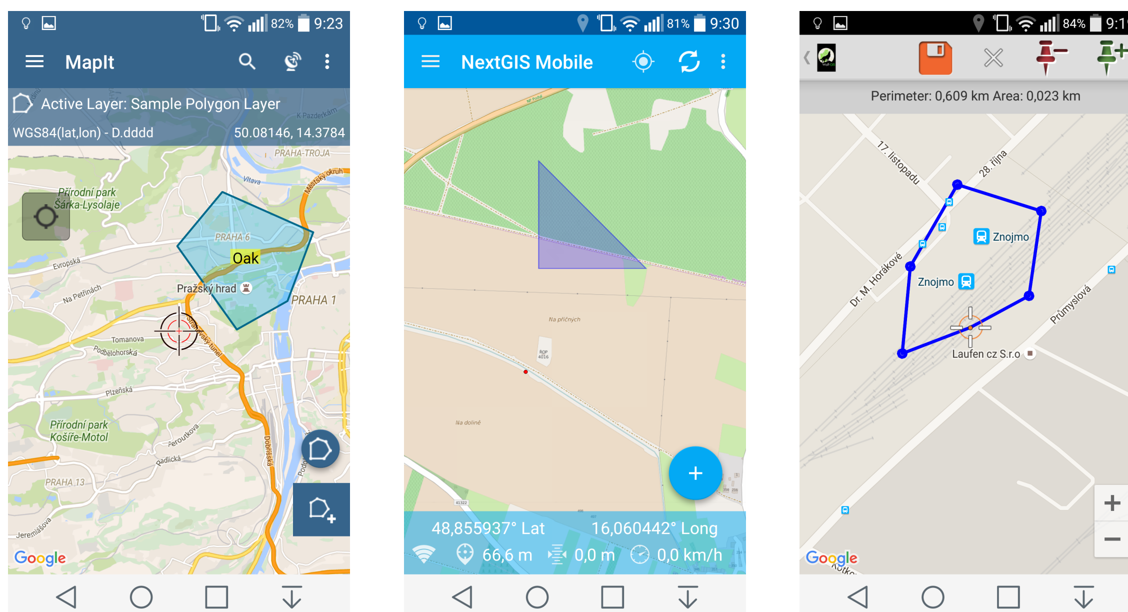
Na trhu mobilních geografických informačních systémů existuje řada produktů. Společnost *Esri* nabízí sérii produktů pro mobilní zařízení – *Collector for ArcGIS*, *Explorer for ArcGIS*, *Survey123 for ArcGIS*, *Snap2Data* aj. Od dalších vývojářů zde budu srovnávat aplikace *MapIt*, *Wolf-GIS*, *NextGIS Mobile*, *Map Plus* nebo *Mappt*. (Google Play, 2015; iTunes, 2015)



Obrázek 1: Ukázky aplikací pro *iOS*. Nalevo aplikace *Explorer for ArcGIS*, která slouží pouze na zobrazování. Uprostřed *Collector for ArcGIS*, která je zaměřená na sběr dat do připravených map. Napravo *Map Plus*, která umožňuje záznam dat, jejich import a export, nikoliv však synchronizaci.

Všechny zmiňované aplikace spolupracují se serverem, jsou však orientovány především na zobrazení již hotových map, případně sběr dat do připravených map a formulářů. Možnosti vytváření a editace samotných map a vrstev jsou omezené. Stejně tak synchronizace naměřených dat mezi více zařízeními není podporována.

Mé navrhované aplikaci se nejvíce podobá aplikace *Map It*. Podporuje správu vrstev a atributů přímo v aplikaci. Podporuje nastavení barvy a popisku prostorových dat v mapě, nikoliv však symbologii bodu. Aplikace má přívětivé uživatelské rozhraní, obsahuje široké možnosti přizpůsobení a podporuje různé mapové podklady (*Google Maps*, *Open Street Map*, *Bing Map*, ...). Podporuje také offline mapy. Obsahuje funkci pro zálohování, export a import. Plnohodnotná synchronizace však není podporována.



Obrázek 2: Ukázky aplikací pro platformu *Android*. Nalevo aplikace *MapIt*, která obsahuje dostatečné množství funkcí na měření a správu vrstev, ale nepodporuje obousměrnou synchronizaci. Uprostřed aplikace *NextGIS Mobile*, která má omezené možnosti přizpůsobení objektů a nepodporuje synchronizaci. Napravo aplikace *Wolf-GIS*, která je zaměřena především na mapování parcel a podporuje pouze polygony.

2.2 Vývojové prostředí

Cílová platforma pro vývoj je *iOS*. Proto se budu orientovat při výběru vývojového prostředí a jazyka pouze na ty, které podporují vývoj pro zařízení s operačním systémem *iOS* od společnosti *Apple*. Jedná se o programovací jazyky *Objective-C* a *Swift*.

Společnost *Apple* oznámila dne 2. 6. 2014 vydání nového programovacího jazyka *Swift* (Zavřel, 2014). *Swift* je moderní a výkonný jazyk kombinující výhody kompilovaných jazyků a dynamiky skriptovacích jazyků. *Swift* je možné v rámci jedné aplikace používat zároveň s jazykem *Objective-C* pomocí tzv. *bridging headers* a díky tomu je možné využívat i starší knihovny třetích stran (Swift and Objective-C in the Same Project, 2015). Pro moji aplikaci zvolím jako hlavní jazyk *Swift*, jehož vývoj se již dá považovat za stabilní a je doporučovaný ze strany *Applu*. Z toho mohu usoudit, že *Apple* bude tento jazyk dále vyvíjet a *Objective-C* bude postupně ustupovat. Musím však počítat s tím, že je to nový jazyk a budou tedy častěji třeba změny v kódu z důvodu vydání nových verzí *Swiftu*.

Vývojových prostředí s podporou *Swiftu* není mnoho. Jmenuji zde oficiální vývojové prostředí *Xcode* od společnosti *Apple* a alternativní prostředí *AppCode* od české společnosti *JetBrains* (JetBrains AppCode: Swift Execution of Your Bright Ideas, 2015). Z těch multiplatformních zde mohu uvést *Qt*, které však pro kompilaci na *iOS* vyžaduje *Xcode* (Qt for iOS, 2015). Obecně všechny prostředí jsou pro operační sys-

tém *OS X* a neexistuje žádné efektivní prostředí pro jiný operační systém.

Pro moje potřeby bude postačující prostředí *Xcode*, které je narozdíl od *AppCode* dostupné zdarma. Prostředí *AppCode* je plně kompatibilní s *Xcode* a v případě potřeby je tedy možné prostředí změnit. Prostředí *Qt* by bylo vhodné, pokud bych tvořil multiplatformní aplikaci, ale to není mým cílem. Mým cílem je vyvinout aplikaci podle designových principů od *Apple* a k tomu bude nejvhodnější oficiální vývojové prostředí *Xcode*.

2.3 Základní knihovny a API

2.3.1 Uložení persistentních dat

V rámci aplikace bude potřeba ukládat uživatelská data a nastavení. K tomu využiji standardní knihovny a nástroje. Pro ukládání větších objemů dat využiji framework *CoreData*, pro uživatelská nastavení třídu *NSUserDefaults*. Pro citlivá data, jako je uživatelské jméno a heslo, je možné využít *Keychain API* (Nahavandipoor, 2014).

2.3.2 Uživatelské rozhraní

Při navrhování uživatelského rozhraní se budu držet směrnic vydaných společností *Apple* (iOS Human Interface Guidelines, 2015). Budu při tom využívat framework *Cocoa Touch* (Neuburg, 2015) a obecné vzory pro navrhování uživatelských rozhraní (Tidwell, 2011).

2.3.3 Správa závislostí

V rámci mé aplikace budu potřebovat knihovny třetích stran, a proto bude vhodné použít nějaký nástroj na správu závislostí.

V první řadě zde uvedu *CocoaPods* (CocoaPods, 2015), nástroj na správu závislostí s centrálním repozitářem s více jak 100 tis. balíčky. Jako protiklad můžu uvést nástroj *Carthage* (Carthage, 2015), který je na rozdíl od *CocoaPods* decentralizovaný a přenechává vývojáři více volnosti, ale také více práce.

2.4 Mapovací a polohové služby

Pro implementaci mapových služeb existuje řada frameworků. V první řadě zde uvedu základní *Map Kit framework* (Location and Maps Programming Guide, 2015). V operačním systému *iOS* do verze 5.1 využíval tento framework *Mapy Google*, novější verze využívají *Apple mapy*.

Mapy Google se nyní implementují do *iOS* pomocí samostatného SDK (Google Maps SDK for iOS, 2015).

OpenStreetMap nabízí řadu frameworků (Frameworks – OpenStreetMap Wiki, 2015). Pro využití mapových podkladů z *OpenStreetMap* mohu také použít *Map Kit framework*, ve kterém lze změnit pozadí mapových dlaždic (OSM in MapKit, 2015).

Další alternativou může být renderování mapy z vektorových dat (Introducing Mapbox GL, 2015). Tento způsob nabízí vysoké možnosti přizpůsobení a ušetření datového přenosu. Velkou výhodou *Mapboxu* je možnost upravit si styl mapy způsobem podobným stylování webových stránek pomocí kaskádových stylů (CSS).

Výše zmíněné frameworky a knihovny obsahují zabudované funkce pro lokaci zařízení. Pokud by tato funkcionalita pro lokaci nebyla dostatečná, mohu využít vysoce konfigurovatelný framework od společnosti Apple (Core Location Framework Reference, 2016).

2.5 Prostorová data

V rámci aplikace bude třeba přenášet data mezi zařízením a serverem a dále tato data ukládat v lokální databázi zařízení.

K dispozici existuje řada formátů a jazyků pro zápis strukturovaných geografických dat. Zmíním zde oblíbený formát *JSON* a jeho geografickou verzi *GeoJSON* (The GeoJSON Format Specification, 2015). Odlišný způsob představují jazyky *KML* (Keyhole Markup Language, 2015) a *GML* (Geography Markup Language, 2016) založené na metajazyku *XML*. Výhoda formátu *JSON* spočívá v jeho menší datové náročnosti a lepší čitelnosti. Jazyky založené na *XML* mají oproti formátu *JSON* výhodu ve větší striktnosti dokumentu a podporují složitější dotazy nad daty (např. hledání dat podle cesty). Vzhledem k tomu, že má aplikace bude provádět složitější operace přímo v databázi a zvolený formát bude sloužit především pro přenos dat mezi zařízením a serverem, bude zde vhodnější použít *JSON*.

Prostorová data bude třeba ukládat do databáze v zařízení. Bude se jednat o základní geometrické objekty jako body, linie a polygony. Linie a polygony se popisují více body – uzly. První možností je ukládat souřadnice uzlů každého takového útvaru běžným způsobem jako řádky v tabulce spojené `1:n` vazbou k příslušnému objektu (linii nebo polygonu). Výhodou tohoto přístupu je více možností operací nad daty, nevýhoda je složitější implementace kvůli složitějšímu schématu databáze. Druhá varianta ukládání spočívá ve vytvoření řetězce ve formátu *GeoJSON*. Například polygon pak bude v lokální databázi uložen jako řetězec obsahující *JSON* pole bodů. Každý prvek pole bude obsahovat souřadnice určitého bodu – uzlu polygonu.

```
"geometry": {
    "type": "LineString",
    "coordinates": [
        [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 0.0]
    ]
}
```

Protože se jedná o obyčejný řetězec a zároveň ve formátu *GeoJSON*, mohu ho využít přímo pro export dat do souboru nebo odeslání dat na server při synchronizaci bez nutnosti konverze.

2.6 Síťová komunikace, synchronizace

2.6.1 Dostupná řešení

Pro komunikaci mezi zařízením a databází na serveru existuje řada technologií a typů rozhraní. Jako příklad zde uvedu *REST* rozhraní (Fielding, 2000), *SOAP* protokol (Quaine, 2007), standard *CORBA* (Common Object Request Broker Architecture, 2016), framework *WCF* (Windows Communication Foundation, 2016) apod. Každá technologie využívá jiný formát přenášených dat a jiný způsob reprezentace operací. Zatímco *REST* systém pracuje s formátem *JSON*, *SOAP* využívá jazyk *XML*.

Komplexnější komunikační službu poskytuje *Google* s jeho knihovnou pro cloudové posílání zpráv (Google Cloud Messaging, 2015). Knihovna umožňuje zasílat krátké zprávy mezi zařízením a serverem. Zprávy jsou ve formátu *JSON*. Tato knihovna by byla v rámci mé aplikace zajímavá především z pohledu *downstream* zpráv, tedy rozhraní pro posílání zpráv ze serveru do zařízení. Toho bych využil pro posílání aktualizací do zařízení. Nevýhoda této knihovny je závislost na centrálním *GCM Connection Serveru* a také omezená velikost zpráv (4 kB). Vlastní prostorová data by tedy bylo potřeba posílat jinou cestou.

CloudKit je API od společnosti *Apple* umožňující sdílet data mezi aplikacemi na zařízeních *iOS*, desktopovými systémy *OS X* a webovými aplikacemi (iCloud for Developers, 2016). *CloudKit* podporuje také tzv. *push notifikace*. Nevýhodou je možnost využití pouze na systémech od společnosti *Apple*.

Další alternativní službou je *FireBase* (Firebase - Build Extraordinary Apps, 2016), která umožňuje ukládat data v serverové *NoSQL* databázi ve formátu *JSON*. *FireBase* je stejně jako *Google Cloud Messaging* a *Cloud Kit* zdarma, avšak pouze pro limitovaný počet uživatelů a omezené velikosti prostoru pro ukládání dat.

Při výběru technologie jsem však omezen současnou implementací serveru, a proto se budu držet technologií podporovaných serverem, jak jsou popsány v následující sekci.

2.6.2 Současné serverové API

Pro synchronizaci využiji existující serverové řešení, které obsahuje API rozhraní postavené na *Representational State Transfer (REST)* architektuře. Server umožňuje komunikaci prostřednictvím formátu *JavaScript Object Notation (JSON)*, částečně s využitím formátu *GeoJSON* – formátu pro ukládání geografických dat vycházejícím z formátu *JSON*. Kompletní přehled podporovaných požadavků a možných odpovědí je součástí elektronických příloh této práce. Základním požadavkem při výběru nástrojů pro síťovou komunikaci tedy bude podpora komunikace ve formátu *JSON*. Tento formát je dobře čitelný a přitom úsporný oproti formátům založeným na *SGML*. Úspora přenosu dat je pro mobilní zařízení velmi důležitým faktorem. (Úvod do JSON, 2015; JSON: The Fat-Free Alternative to XML, 2015)

Současné serverové řešení je již využíváno aplikací pro platformu *Android*. Na to je třeba dbát při případných úpravách serverového API kvůli vzájemné kompatibilitě různých platforem.

Následuje přehled možných řešení pro implementaci síťové komunikace. Všechny dále zmiňované technologie jsou kompatibilní s rozhraním serveru.

2.6.3 Foundation framework

Foundation framework obsahuje řadu tříd pro síťovou komunikaci. Jak uvádí Nahavandipoor (2014), třída *NSURLSession* poskytuje obalení požadavků na server do formy úloh, které mají rozsáhlé možnosti konfigurace. Ve frameworku je také k dispozici starší alternativa – třída *NSURLConnection*. Obě třídy jsou abstraktní strukturou zastřešující primitivní třídy jako *NSURLRequest*, *NSURLResponse* apod. (Thompson, 2015)

2.6.4 RestKit framework

Pro síťovou komunikaci existuje řada nezávislých frameworků a knihoven. Jmenuji zde především komplexní a vyzrálý *RestKit* (RestKit/RestKit · GitHub, 2015) se širokou základnou přispěvatelů. Výhodou frameworku *RestKit* je především možnost jednoduchého mapování dat z lokální databáze do těla požadavku odesílaného na server a naopak mapování dat z odpovědi serveru do lokální databáze. *RestKit* také podporuje mapování relací a automatické generování URI podle obsahu, tzv. *routing*.

2.6.5 Knihovny nižší úrovně

Použití komplexních frameworků v aplikaci by zjednodušilo vývoj a kód, avšak pro moji navrhovanou aplikaci by konfigurační možnosti frameworku nemusely být dostatečné. Proto ještě níže uvádím jednodušší frameworky a knihovny, které zajišťují jednotlivé potřebné funkce odděleně a bude tak snazší je spojit do celku, který by se choval podle našich požadavků.

Mezi jednodušší a méně komplexní knihovny mohu zařadit *SwiftHTTP* (2015), který obaluje *NSURLSession* s cílem zjednodušit programové struktury pro vytvoření požadavku na server.

SwiftyJSON (2015) je knihovna umožnující zpracovat řetězec obsahující *JSON* a následně pracovat s datovou strukturou pomocí tzv. *subscriptů*, jak to známe z dynamicky typovaných jazyků jako je *Python* nebo *PHP*. Knihovna také umí převzít data ve formě slovníku *NSDictionary* nebo pole *NSArray* a vygenerovat z nich řetězec ve formátu *JSON*. Práce s daty je díky této knihovně pohodlnější než s použitím základních objektů *NSArray* a *NSDictionary* díky jednoduššímu přetypování a ošetření nedefinovaných hodnot.

Pro plynulost uživatelského rozhraní bude třeba provádět synchronizaci dat se serverem asynchronně. K tomu využiji nízkoúrovňové API nazvané *Grand Central*

Dispatch, které umožňuje provádět multitasking pomocí fronty úloh, tzv. *dispatch queue* (Nahavandipoor, 2014).

Synchronizaci nových změn ze serveru do mobilního zařízení bude potřeba provádět periodicky. K tomu využiji třídu *NSTimer* (Neuburg, 2015), která ve zvoleném intervalu vysílá signály (tzv. *fires*) do námi zvolené instance objektu.

2.6.6 Konflikty

V zařízení bude třeba zajistit aktuálnost a konzistenci všech naměřených dat a v souvislosti s tím řešit konfliktní situace, které mohou nastat při synchronizaci.

Jednotlivé konfliktní situace a možnosti jejich řešení popisuje Liang a Hu (2014). Zaměřují se především na synchronizaci souborů, nicméně principy jsou částečně stejné i při synchronizaci prostorových dat.

Jiný zdroj (Pascuala a Xhafa, 2011) popisuje synchronizační algoritmus z pohledu zařízení. Doporučuje ukládat identifikátor poslední úspěšné synchronizované položky v zařízení místo v tabulce na serveru. Položky v zařízení označuje příznakem, pokud ještě nebyly synchronizovány. Autoři nejprve rozčlení prvky (resp. změněné položky) do množin podle jejich umístění (zařízení, server) a podle toho, zda jsou kandidáti na konflikt. Následně pomocí množinových operací stanoví výsledné skupiny prvků. Pro každou z těchto výsledných skupin je znám způsob, jak pro prvky dané skupiny vyřešit konfliktní situaci a dokončit synchronizaci.

McCormack (2015) doporučuje budovat systém, který bude v každém stavu vždy validní a konzistentní. Čtení a zápis by měly být atomické operace a řešení konfliktu pak spočívá v uživatelském výběru, kterou verzi chce uživatel zachovat. Jinými slovy nedojde v případě konfliktu k přepsání dat, ale vytvoří se nová položka se stejným názvem se změněnými daty. Pokud bych nechtěl nechat uživatele rozhodovat o správné verzi (což někdy může být i nemožné – uživatel nedokáže někdy rozpoznat správnou verzi) a nechat rozhodnutí na serveru nebo aplikaci, musím si stanovit deterministický systém pro určení prorit. Deterministický proto, aby v každém případě se provedlo rozhodnutí podle stejných pravidel a všechna zařízení se rozhodla stejně. V nejjednodušší variantě pak půjde o systém, kdy první (příp. poslední) změna má nejvyšší prioritu a přepíše tak ostatní změny z jiných zařízení.

2.7 Shrnutí

Pro moji aplikaci zvolím vývojové prostředí *Xcode* jako jediné nativní prostředí pro vývoj aplikací na platformu *iOS*, programovací jazyk *Swift* jako moderní jazyk a objektové úložiště *CoreData*. Pro zobrazování mapových podkladů zvolím některou z uvedených mapových služeb. Za účelem jejich srovnání a výběru budou v metodice sestavena kritéria, která jednotlivé služby ohodnotí. Pro ukládání prostorových dat připadají v úvahu dvě možnosti, v rámci metodiky proběhne jejich srovnání na základě poznatků získaných z experimentální aplikace. Knihovny a technologie, které použiji pro síťovou komunikaci a synchronizaci, jsou z části dány technologie-

mi, které podporuje server. V rámci metodiky vytvořím prototyp aplikace využívající framework *RestKit* a otestuji, zda bude vhodný pro použití v mé aplikaci. Pokud ne, použil bych knihovny nižší úrovně. V poslední části rešerše byly popsány základní synchronizační problémy a konflikty, v rámci metodiky budou dále rozvinuty včetně uvedení kompletního přehledu všech dostupných operací a akcí.

3 Metodika

3.1 Základ

Pro vývoj aplikace použiji vývojové prostředí *Xcode* v aktuální stabilní verzi 7.2 a poslední verzi programovacího jazyka *Swift* (2.1). V první fázi si připravím základní strukturu aplikace. Podle designových principů *Apple* a standardních *Apple* rozhraní vytvořím uvítací obrazovku, obrazovku pro první přihlášení, nastavení a výpis projektů. Logo aplikace upravím do požadované velikosti podle specifikace (iOS Human Interface Guidelines, 2015). V této základní aplikaci budu experimentovat s různými knihovnami a přístupy za účelem jejich srovnání a otestování.

3.2 Uživatelské rozhraní

Při návrhu designu budu dbát principů *Apple Human Interface Guidelines*, které kladou důraz na tři faktory:

- Podřízenost – Aplikace musí uživateli pomáhat splnit úkol, ne ho měnit a dělat věci proti jeho vůli.
- Srozumitelnost – Zahrnuje čitelný text, pochopitelné ikony a správné vizuální vyvážení grafických prvků.
- Hloubka – *Apple* doporučuje tvořit realistické vrstvení designu a přirozené animace na podporu přehlednosti.

Apple nedoporučuje používat barvené přechody a stíny, klade důraz na plochý papírový design, prosté barvy, využívání negativního místa, průhlednosti a velkých kvalitních obrázků. Pro tlačítka a jiné aktivní prvky je nutné ponechat dostatek volného prostoru v okolí, aby nedocházelo k nechtěným překlepům na dotykovém displeji. Pro méně časté akce se v poslední době často používají textová tlačítka namísto obrázkových.

Animace, přechody a modální dialogová okna by měla podporovat uživatelovu orientaci a hiearchii zobrazených informací. To bude v mé aplikaci podpořeno jednoduchými a intuitivními gesty. Použiji např. swipe gesto pro zobrazení akcí ve výpisech. Gesta budou také použita pro pohyb na mapě a práci s mapovými objekty.

Aplikace bude určena nejen pro mobilní zařízení, ale také pro tablety. Proto u často používaných obrazovek připravím i verzi pro tablety, která bude lépe využívat velký displej. Vytvoření responzivního layoutu bude dosažano kombinací tříd *UISplitViewController*, kontejnerů (speciální *UIView* umožňující vložit jiný *UIViewController*) a technologie *Size Classes*. *UISplitViewController* umožňuje zobrazení postranního panelu, avšak má příliš striktně definované chování. Chová se totiž vždy podle vzoru *master – detail*, přičemž z *master* kontroleru se načítá *detail* kontroler a postranní panel je vždy nalevo. Toto chování někdy nemusí být přípustné. Na-

příklad u mapy není žádoucí při změně aktivní vrstvy načítat celou mapu znovu. Proto bude nutné vytvořit vlastní obdobu *UISplitViewControlleru* prostřednictvím kontejnerů a *Size Classes*.

3.3 Synchronizace

3.3.1 Výběr knihoven a frameworků

Pro lepší poznání synchronizačních problémů jsem vytvořil prototypy pomocí různých knihoven. Pro stahování a aktualizaci knihoven jsem zvolil nástroj *CocoaPods*, který podporuje všechny knihovny, které budu v aplikaci pořebovat. V první fázi jsem otestoval framework *RestKit*. Ten se však ukázal jako nevhodný. Obsahuje sice pohodlné mapování *JSON* dat přímo do úložiště *Core Data*, avšak pro většinu operací potřebných v mé aplikaci je nevhodný, protože požadavky provádí ihned. Nelze požadavek odložit na později do fronty. Mobilní zařízení může být nějakou dobu bez připojení k internetu a je třeba uchovávat všechny požadavky persistentně, dokud nebudou vyřízeny. To může nastat až následující den při novém spuštění aplikace. Framework *RestKit* by tedy zde neměl žádný přínos.

Další prototyp jsem vytvořil s použitím knihoven *SwiftHTTP*, *AFJSONRequestOperation* a *SwiftyJSON*. Použití těchto jednodušších jednoúčelových knihoven mi umožní vytvořit vlastní řešení odpovídající charakteru navrhované aplikace. Toto vlastní řešení bude mít podobné rozhraní jako framework *RestKit*, ale navíc bude zajišťovat i persistentní frontu požadavků a jiné speciální operace s daty (jako např. odesílání obrázku na server).

3.3.2 Identifikace objektů

Objekty musí být v rámci synchronizace jedinečně identifikovány. Tato identifikace musí být jednotná v rámci celého systému, tedy na centrálním serveru i na všech připojených zařízeních. Identifikátor objektu se proto vytváří na serveru, nikoliv v zařízení. Při vytvoření objektu v zařízení se nejprve vytvoří pouze dočasný identifikátor pro potřeby zařízení a až v okamžiku úspěšné synchronizace obdrží zařízení pernamentní a jednotný identifikátor. Zařízení bude tento identifikátor používat pro všechny další synchronizační operace s daným objektem.

Některé objekty obsahují další podřazené objekty. Pro příklad zde uvedu vrstvu, která obsahuje body jako podřazené objekty. Samotný identifikátor bodu je unikátní pouze v rámci jedné vrstvy a proto v absolutní cestě daného bodu je zahrnut i identifikátor vrstvy. Z toho vyplývá, že bude třeba synchronizovat objekty v určitém pořadí. Nejprve je potřeba úspěšně synchronizovat nadřazené položky, obdržet jejich identifikátor a až poté lze synchronizovat podpoložky.

Cesty k objektům jsou tvořeny jako URI adresy. Níže uvádím obecné schéma, pomocí kterého lze identifikovat objekty až do třetí úrovně:

```
/typ[/schéma[/id][/typ[/schéma][/id][/typ[/id]]]]
```

Hranaté závorky označují volitelnost. Schéma je tvořeno uživatelským jménem a je používáno pro potřeby serveru. Jedinou výjimku tvoří sdílené vrstvy (označené atributem `public = true`), které mají speciální schéma `shared`. U objektu třetí úrovně se již schéma nepoužívá, protože se vždy jedná o objekty ve stejném schématu jako je nadřazená položka druhé úrovně.

Příklad URI ukazující na projekt s identifikátorem `m123` ve schématu `clienttestuser`.

```
/project/clienttestuser/m123
```

Příklad URI na vytvoření nové hodnoty atributu pro mapový objekt s identifikátorem `123456` ve sdílené vrstvě s identifikátorem `l123456`. Vrstva je ve schématu `shared`.

```
/layer/shared/l123456/feature/123456/attribute
```

Příklad URI pro objekt typu *LayerInProject* – objekt spojující projekt s vrstvou. Projekt je ve schématu `clienttestuser`, ale vrstva je ve sdíleném schématu `shared`.

```
/project/clienttestuser/m123/layer/shared/l123456
```

3.3.3 Další speciální typy URI

Adresa pro stažení všech dat pro přihlášeného uživatele a vygenerování nového ID zařízení:

```
/sync/begin
```

URI využívaná pro periodické stahování změn ze serveru v rámci rozdílové synchronizace:

```
/sync/event
```

Pro získání kontaktních informací o uživateli slouží URI:

```
/user/uzivatelske_jmeno
```

3.3.4 Základní stavy synchronizace

V rámci synchronizace mohou nastat tři situace, po kterých musí následovat synchronizace, aby byly data konzistentní:

1. Objekt v zařízení je aktualizován a je tak novější než stejný objekt na serveru a v jiných zařízeních.
2. Objekt na serveru je změněn jiným zařízením a je tím pádem novější než objekt v zařízení.

3. Objekt je změněn ve více zařízeních současně (nebo krátce po sobě) ještě před tím, než se první změna synchronizuje do zařízení, které provedlo druhou změnu. Na serveru je nastavena politika, která v případě konfliktu upřednostní první požadavek o změnu.

3.3.5 Přehled standardních operací

V rámci *REST* rozhraní existují čtyři typy operací s objekty:

- Stažení (`GET`) – Zařízení požádá server o zaslání dat určitého objektu nebo více objektů. Tato operace bude mít dvojí využití. Při periodické synchronizaci bude zařízení žádat prostřednictvím této operace o zaslání změn ze serveru. Druhé, méně časté využití je stažení dat na požádání uživatele, např. pro obnovení výpisu map nebo stažení obrázku.
- Vytvoření (`POST`) – Zařízení zašle na server nový objekt. Server objekt vytvoří v databázi a zašle zpět unikátní identifikátor nově vytvořeného objektu pro další komunikaci. Server také zašle *hash* jako časové razítko změny. Ten je použit při dalších požadavcích pro detekování konfliktů.
- Úprava (`PUT`) – Zařízení zašle na server změněný objekt. Server změny uloží. Který objekt má server změnit, je určeno identifikátorem, popř. URI adresou složenou z více identifikátorů (např. vrstva přidaná do mapy je identifikována URI adresou tvořenou identifikátorem mapy a identifikátorem vrstvy). Po úspěšném uložení pošle server do zařízení nové časové razítko (*hash*).
- Smazání (`DELETE`) – Zařízení požádá server o smazání objektu.

3.3.6 Přehled odpovědí serveru

Následuje přehled kódu, kterými server odpovídá na požadavky ze zařízení.

- Typické odpovědi při úspěšném vyřízení požadavku:
  - `200 OK` – pro `GET` požadavek při nalezení požadovaného objektu na serveru,
  - `201 Created` – po úspěšném vytvoření objektu,
  - `202 Accepted` – po úspěšném uložení změn objektu,
  - `204 No content` – po úspěšném smazání objektu.
- Konfliktní odpovědi:
  - `404 Not Found` – Položka již byla na serveru smazána. Je potřeba smazat lokální objekt, pokud existuje.
  - `409 Conflict` – Položka na serveru byla změněna ještě dalším zařízením. Je potřeba aktualizovat zařízení podle první přijaté změny. Konflikty na serveru

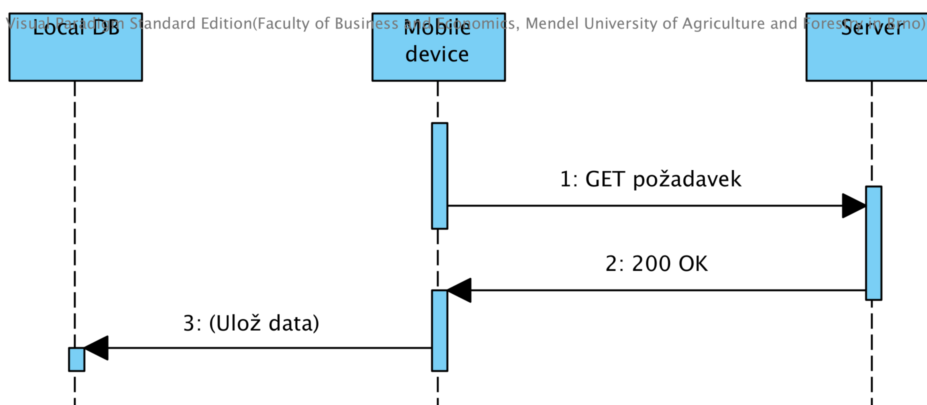
jsou detekovány podle časového razítka, které se jako *hash* posílá se všemi objekty.

  - `460 Device ID Lost` – Platnost ID zařízení na serveru vypršela je potřeba získat nové device ID odesláním požadavku na stažení všech dat a následným sloučením stažených dat a dat v databázi zařízení.

- Chybové odpovědi:
  - `500` a jiné kódy neuvedené výše – Nastala chyba synchronizace způsobená buď chybou v API na serveru, nebo špatným formátem či obsahem požadavku ze zařízení. Tuto situaci je třeba vhodně ošetřit a pokusit se o její nápravu, příp. informování uživatele.

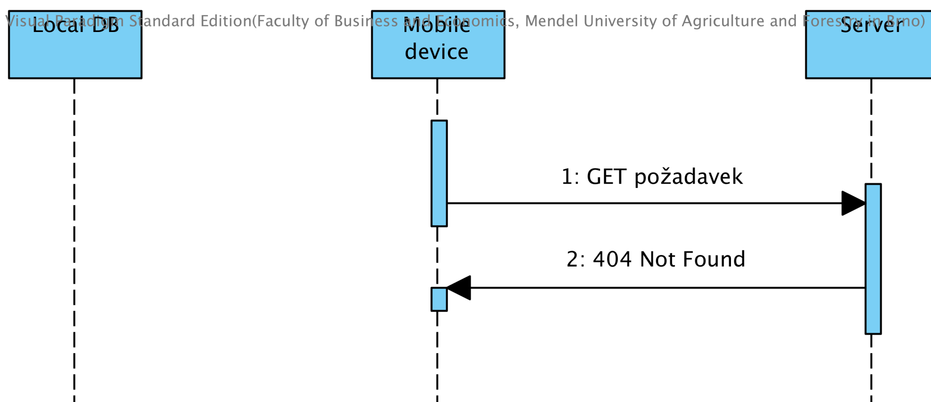
3.3.7 Diagramy synchronizačních operací

Následuje série diagramů zobrazující všechny kombinace operací a možných odpovědí serveru s návazností na akci v lokální databázi. První dva diagramy znázorňují operaci `GET` (obr. 3 a 4). Následují operace `POST` (obr. 5 a 6), `PUT` (obr. 7, 8 a 9), `DELETE` (obr. 10 a 11) a na závěr obecný diagram chybové odpovědi (obr. 12).

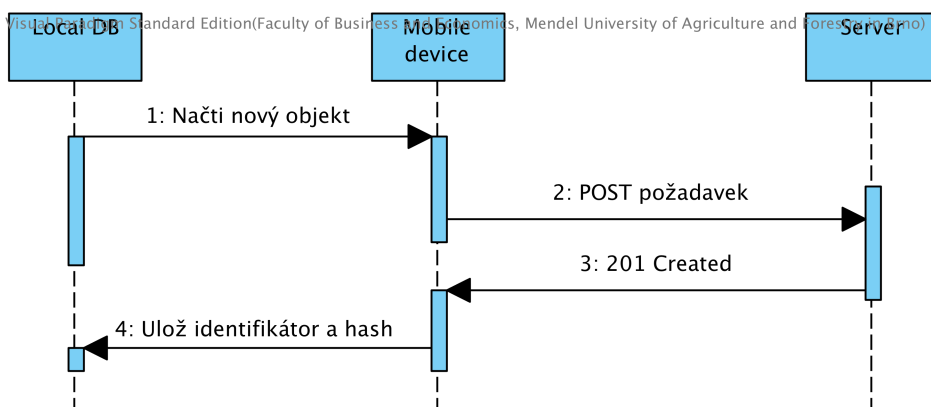


Visual Paradigm Standard Edition(Faculty of Business and Economics, Mendel University of Agriculture and Forestry in Brno)

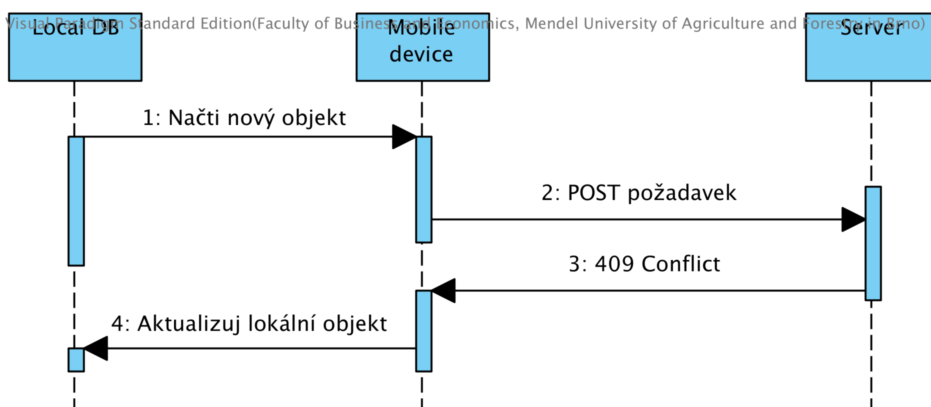
Obrázek 3: Diagram zobrazuje požadavek na stažení obsahu ze serveru a uložení tohoto obsahu do databáze v zařízení.



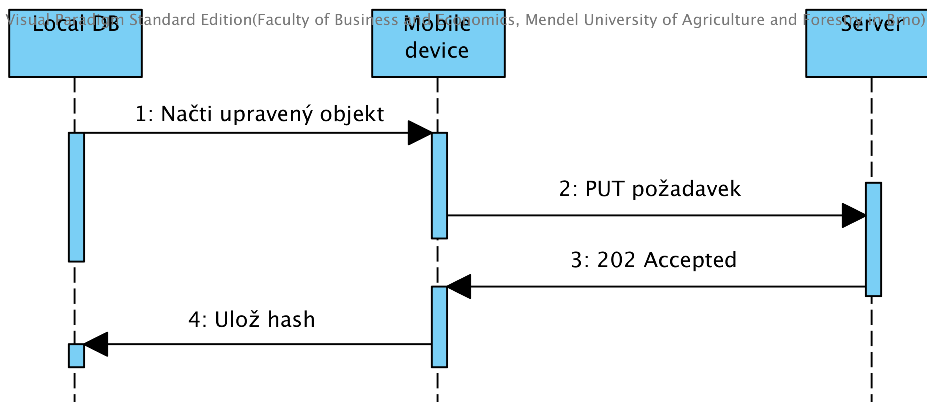
Obrázek 4: Diagram zobrazuje požadavek na stažení obsahu ze serveru. Server požadovaný obsah nenašel a vrací `404 Not Found`.



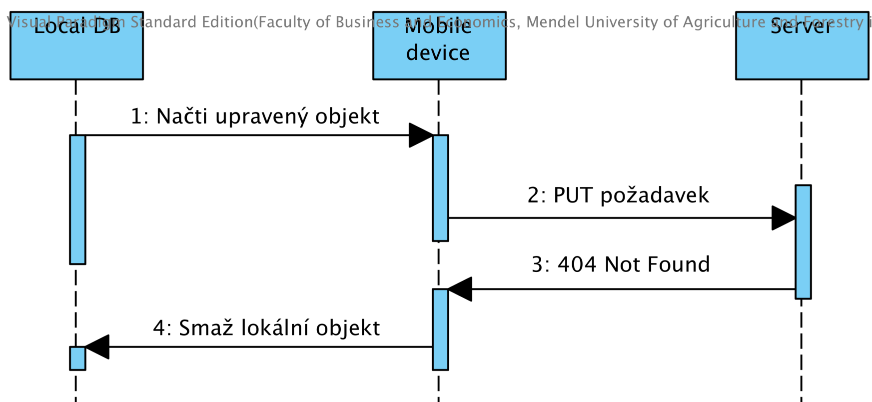
Obrázek 5: Diagram zobrazuje požadavek na vytvoření nového objektu. Server vytvoří objekt a vrátí `201 Created`.



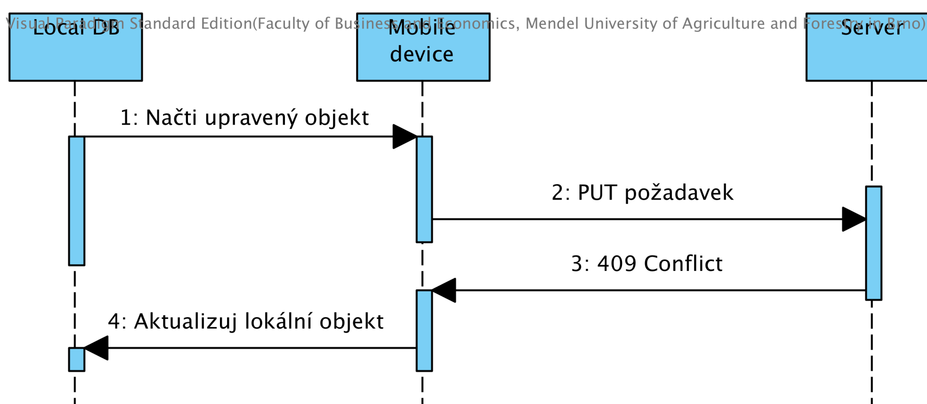
Obrázek 6: Diagram zobrazuje požadavek na vytvoření nového objektu. Na serveru již existuje podobný objekt a je porušena unikátnost. Server vrací `409 Conflict` společně s daty již existujícího objektu. Zařízení aktualizuje svůj nově vytvářený objekt daty ze serveru.



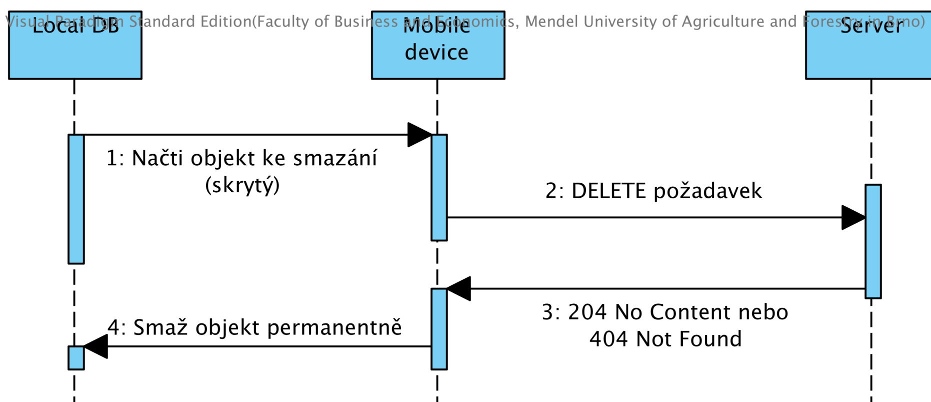
Obrázek 7: Diagram zobrazuje požadavek na změnu objektu. Na serveru se uloží změny a server vrátí odpověď `202 Accepted`



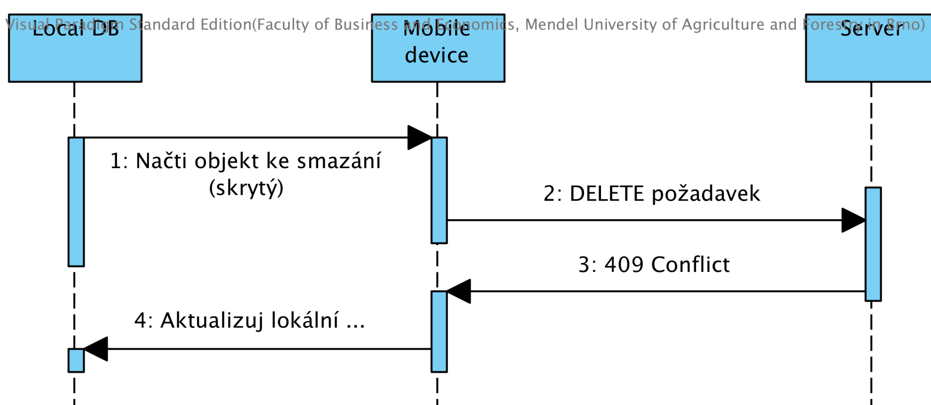
Obrázek 8: Diagram zobrazuje požadavek na změnu objektu. Na serveru byl již objekt smazán a server vrací `404 Not Found`. Zařízení smaže lokální objekt.



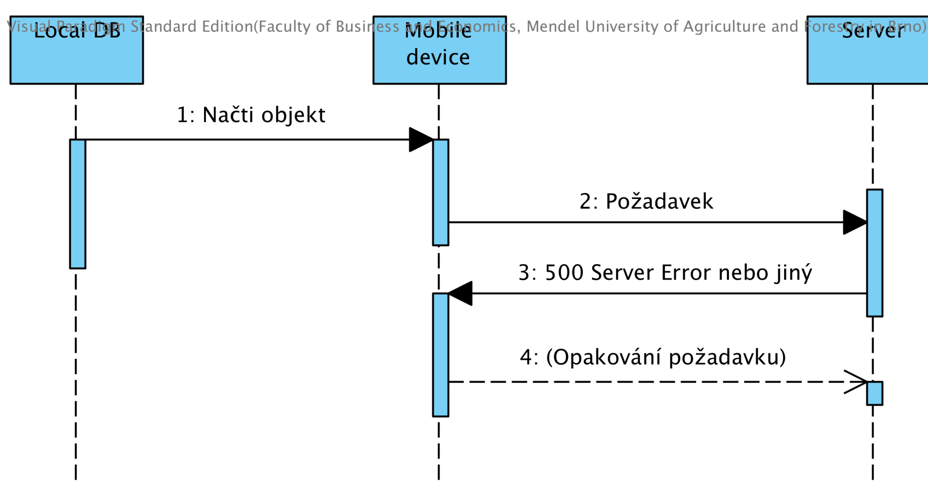
Obrázek 9: Diagram zobrazuje požadavek na změnu objektu. Na serveru byl již objekt změněn jiným zařízením a server vrací `409 Conflict`. Zařízení aktualizuje lokální objekt novými daty z odpovědi serveru.



Obrázek 10: Diagram zobrazuje požadavek na smazání objektu. Pokud uživatel smaže v zařízení objekt, nejprve dojde pouze k jeho skrytí. (U některých typů objektů je nutné šířit příznak skrytí i na podpoložky.) Server následně vrací `204 No Content` v případě úspěšného smazání. V případě, že požadovaný obsah nenašel, vrací `404 Not Found`. V obou případech dojde po přijetí odpovědi k trvalému smazání objektu v zařízení.



Obrázek 11: Diagram zobrazuje požadavek na smazání objektu. Objekt v zařízení není ihned smazán, ale pouze skryt. Pokud server vrátí konfliktní odpověď `409 Conflict`, zařízení zruší skrytí objektu a aktualizuje jeho parametry podle parametrů přijatých ze serveru v rámci odpovědi.



Obrázek 12: Pokud při některé z výše uvedených operací vrátí server chybu `500 Server Error`, respektive jinou než některou z očekávaných odpovědí, zařízení pozastaví synchronizaci, informuje uživatele a pokusí se o opětovnou synchronizaci později.

3.3.8 Další problémové situace

Na prototypu byly otestovány konfliktní situace. Bylo zjištěno, že bude potřeba kromě standardních synchronizačních problémů řešit i následující situace:

- Je potřeba ošetřit situaci, kdy dojde vlivem automatické synchronizace ke změně položky, kterou uživatel právě edituje. Např. vrstvu, které uživatel mění název, mezitím někdo jiný smaže. U důležitých položek (např. vrstev) budu uživatele informovat při pokusu o uložení, že objekt již byl smazán. U méně důležitých položek (jako jsou atributy vrstvy) nebudu zobrazovat informaci uživateli, uložení položky ignoruji. Je však potřeba také ošetřit, aby se položka již nesynchronizovala, protože na serveru již neexistuje a došlo by k přerušení synchronizace.
- Při prvním přihlášení je potřeba stáhnout najednou všechna data ze serveru. To samé se ale musí stát také při změně přihlašovacích údajů, při vážné chybě synchronizace nebo po vypršení platnosti identifikátoru zařízení na serveru, který posílá zpět stav `460 Device ID Lost`. Při tomto stažení není dobré vše v zařízení nejprve smazat a poté stahovat, protože uživatel by mohl přijít o ještě nesynchronizovaná data. Vhodné řešení je použít sjednocení nových dat s aktuální databází a poté vytvořit nový pokus o odeslání dosud nesynchronizovaných dat.
- Periodická synchronizace bude asynchronní a je tedy potřeba při přijetí změny ze serveru nějakým způsobem informovat všechny související kontrolery, aby aktualizovaly změněné položky.
- Synchronizační požadavky je nutné vyřizovat v určitém pořadí. Nelze synchronizovat podpoložku, pokud ještě nebyla úspěšně synchronizována nadřazená položka.
- Je potřeba ošetřit, aby nedošlo k vyřizování fronty vícekrát paralelně. Fronta musí tedy být singleton a musí uchovávat svůj aktuální stav (připravena, probíhá synchronizace, chyba atd.).

3.4 Ukládání prostorových dat v zařízení

V úvahu přichází dvě varianty. První možnost je ukládat prostorová data do samostatné entity. Tato varianta je však náročná na implementaci a mohlo by dojít k neúměrnému nárůstu množství položek v související entitě se souřadnicemi a ke zpomalení načítání objektů. Druhá varianta je ukládat prostorová data přímo k objektu jako řetězec nebo jako pole uložené ve formě binárních dat nebo datový typ *transformable*. Otestoval jsem variantu s binárními daty i s typem transformable. Obě varianty však zabírají příliš mnoho paměti. Uložení dat ve formátu *GeoJSON* jako řetězec zabírá několiknásobně méně prostoru než v případě pole a lze jej jednoduše přímo využít při synchronizaci bez nutnosti výpočetně náročné konverze. Konverzi bude akorát potřeba provádět při zobrazování a editaci geometrie na mapě.

Tam však nikdy nebude potřeba načítat tolik objektů naráz jako při synchronizaci a proto se nejedná o výkonnostní problém.

Pro ukládání prostorových dat tedy bude použit obyčejný řetězec, který by měl usnadnit implementaci a měl by být i výkonnostně a paměťově nejúspornější mezi srovnávanými přístupy.

3.5 Mapový podklad

Srovnání mapových služeb je uvedeno v tab. 1. Úroveň splnění jednotlivých kritérií je ohodnocena stupnicí: 1 (nesplněno), 2 (splněno částečně), 3 (splněno). Při stanovování kritérií jsem vycházel ze základních požadavků na aplikaci a ohledu na budoucí vývoj.

Tabulka 1: Srovnání mapových služeb

Kritérium	Map Kit	Google Maps SDK	Open-Street-Map	Mapbox GL
Souřadnicový systém WGS84	3	3	3	3
Podpora bodů, linií a polygonů vč. obarvení a vlastního symbolu	3	3	3	3
Podpora intuitivních gest pro práci na mapě	3	3	2	2
Dokumentace a uživatelská aktivita	3	3	2	1
Stabilita vývoje	2	3	3	2
Kvalita a typy mapových podkladů	2	3	2	2
Celkem:	16	18	15	13

V rámci srovnání největšího počtu bodů dosáhla služba *Google Maps SDK*, kterou použiji jako výchozí mapovou službu pro moji aplikaci.

4 Vlastní práce

4.1 Úvod

V rámci této části práce popíši implementaci zásadních komponent aplikace. Výsledkem bude aplikace umožňující správu všech základních objektů (projekty, vrstvy, atributy atd.), zobrazení mapy včetně mapových objektů, jejich přidávání a editaci. Aplikace bude schopna všechny tyto objekty synchronizovat směrem na server i stahovat změny ze serveru v rámci rozdílové i kompletní synchronizace včetně řešení konfliktních a chybových situací.

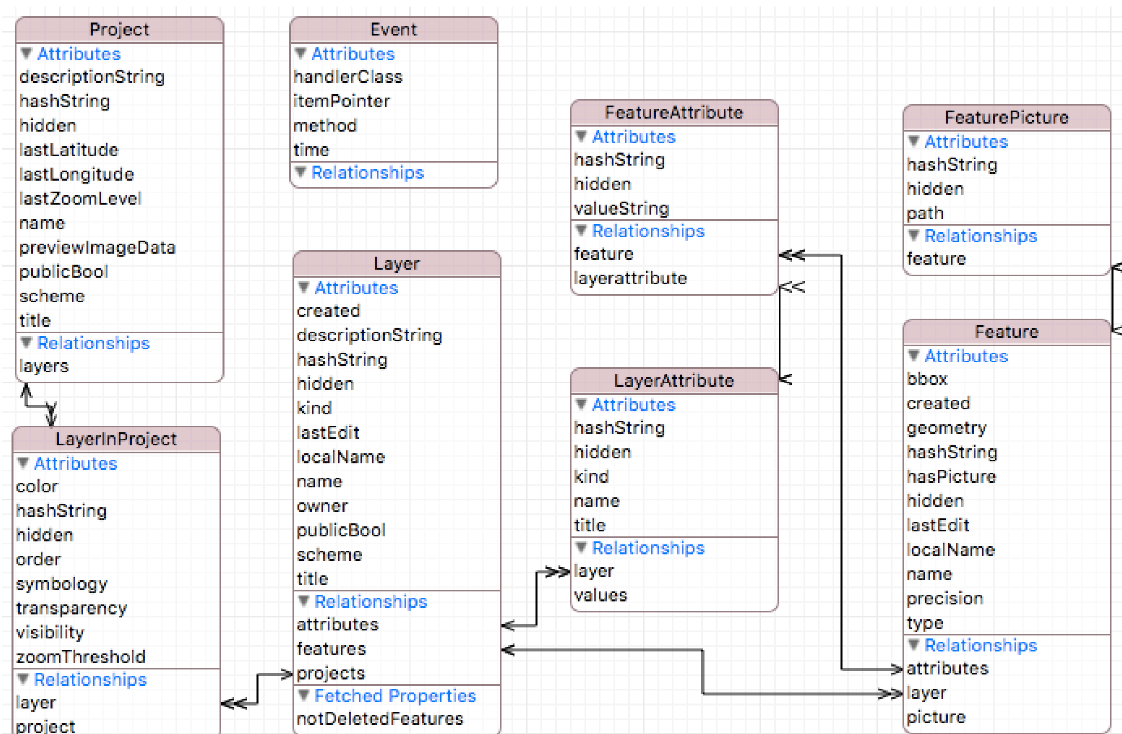
V první části této kapitoly bude popsán systém centrálního úložiště, který zajišťuje jednotný způsob práce s daty a jejich integritu mezi lokální databází a daty na serveru. Další část se zabývá mapovánín neboli konverzí dat mezi lokální databází a formátem pro komunikaci se serverem. Na to navazuje popis tříd obsluhujících jednotlivé typy synchronizace (kompletní, rozdílová, ...) a také vysvětlení implementace speciálního typu synchronizovaných objektů – obrázků. Následující sekce obsahuje stručný popis třídy *Geometry* zajišťující jednotný přístup ke geometrii v rámci uložených prostorových dat. Následuje několik podsekcí popisujících hlavní vlastnosti uživatelského rozhraní (layout, mapa, měření, validace, obnovování výpisů, ...). Na závěr je krátká sekce věnující se konfiguračnímu souboru, ve kterém lze nastavit základní parametry celé aplikace.

4.2 Objektové úložiště

Pro ukládání dat byl použit *Core Data* framework. Na následujícím entitně-relačním diagramu (obr. 13) jsou všechny databázové objekty a jejich vazby. Úložiště obsahuje tyto objekty:

- *Project* – Projekt je někdy označován také jako mapa. Obsahuje název, popis, poslední pozici, úroveň přiblížení kamery apod. Pomocí objektu *LayerInProject* jsou k němu připnuty vrstvy.
- *Layer* – Objekt pro ukládání informací o vrstvách. Vrstva může obsahovat atributy (*LayerAttribute*) a mapové objekty (*Feature*).
- *LayerAttribute* – Atribut vrstvy je popsán názvem a typem. Typ atributu může být číslo nebo řetězec.
- *LayerInProject* – Objekt definující spojení mezi projektem a vrstvou. Obsahuje také uživatelskou definici vzhledu vrstvy – symboliku, barvu a průhlednost objektů. Dále obsahuje pořadí vrstvy v rámci všech vrstev připnutých ke stejnému projektu.
- *Feature* – Mapový objekt typu bod, linie nebo polygon, obsahuje geometrii a může obsahovat hodnoty atributů (*FeatureAttribute*) a obrázek (*FeaturePicture*).

- *FeatureAttribute* – Hodnota atributu. Obsahuje odkaz na definici atributu vrstvy (*LayerAttribute*).
- *FeaturePicture* – Obrázek mapového objektu. Každý mapový objekt může mít maximáně jeden obrázek. Obrázky jsou v zařízení ukládány do souborového systému a v databázi je uložena pouze cesta a *hash*.
- *Event* – Objekt reprezentující synchronizační požadavek ve frontě. Obsahuje odkaz na synchronizovaný objekt a název třídy obsluhující daný typ objektu.



Obrázek 13: Diagram objektů v úložišti

Jednotlivé objekty jsou implementovány rozšířením třídy *NSManagedObject*. To umožňuje snadnější přístup k atributům a také přidávání vlastních generovaných atributů a funkcí. Následující ukázka kódu reprezentuje objekt *FeaturePicture*. Kromě standardních atributů obsahuje tento objekt i odkaz na související objekty (odkaz na mapový objekt *Feature*), dále jsou zde implementovány funkce pro práci s obrázky. Lze tak načítat a ukládat obrázek mapového objektu přímo přes atribut `image` bez nutnosti znalosti vnitřní implementace a způsobu ukládání obrázků v souborovém systému.

```
@objc(FeaturePicture)
class FeaturePicture: Model, ModelProtocol {
    @NSManaged var hashString: String
    @NSManaged var hidden: Bool
```

```
    @NSManaged var path: String

    @NSManaged var feature: Feature?

    /// Dynamic property for loading and saving the feature picture in
        the file system instead of saving it to the database.
    var image: UIImage?{
        get {
            if(feature == nil){
                return nil
            }else{
                let fileManager = FileManager()
                return fileManager.imageForPath(path)
            }
        }
        set (image) {
            if(feature != nil){
                // check the target dir and create it if not exists
                let fileManager = FileManager()
                fileManager.makeDirectory(imageDir)

                if(image != nil){
                    // save the image
                    let compression = CGFloat(config.featurePictureSize
                        .jpegCompression)
                    fileManager.saveImageAsJPEG(image!, path: imagePath
                        , compression: compression)

                    // save the file path to db
                    self.path = imagePath

                }else{
                    // delete the image file if this property is being
                        set to nil
                    if(path != ""){
                        fileManager.delete(path)
                        self.path = ""
                    }
                }
            }
        }
    }
    ...
}
```

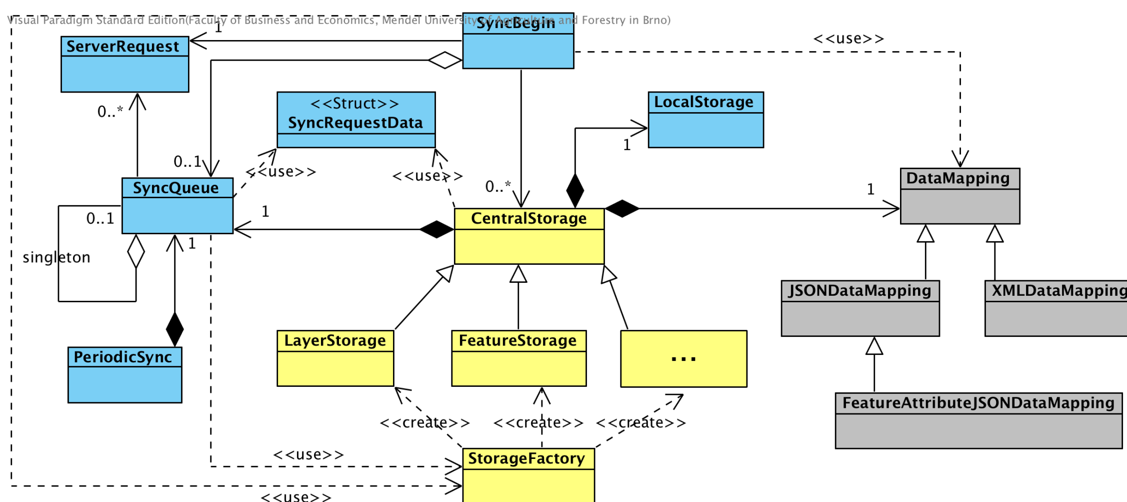
Pro ukládání ostatních dat, které nejsou přímým předmětem synchronizace a které nemají množinovou povahu, byla použita třída *NSUserDefaults*. Je zde uložen např. e-mail uživatele, synchronizační interval nebo dočasné ID zařízení.

4.3 Abstrakce úložiště

Běžně používaný kód pro práci s úložištěm *Core Data*, vypisování a aktualizaci dat v kontrolerech, jako *UICollectionViewController* a *UITableViewController*, je příliš dlouhý a v mé aplikaci by bylo nutné používat stejný kód na více místech. Proto jsem pro zjednodušení běžných operací a výpisů vytvořil skupinu tříd, které abstrahují všechny běžně používané operace pro načítání i ukládání dat (obr. 14).

Základem je třída *CentralStorage*, od které dědí moduly obsluhující jednotlivé objekty (např. třída *LayerStorage* zajišťující operace s objektem *Layer*). Třída *CentralStorage* obsahuje funkce pro načítání objektů z úložiště (vč. seřazení a vynechání objektů určených ke smazání), filtrování objektů podle identifikátoru, názvu apod. Dále obsahuje funkce pro vytváření, úpravu a mazání objektů. Následující kód obsahuje ukázku, jak jednoduše lze pak v kontroleru načíst z databáze vrstvu, upravit její název a uložit ji. Při zavolání funkce `update()` je vrstva uložena a také je změna vrstvy zapsána do synchronizační fronty.

```
let layer = storage.getByTitle("Název vrstvy")
layer.title = "Nový název vrstvy"
storage.update(layer)
```



Obrázek 14: Zjednodušený diagram tříd centrálního úložiště

Třída *CentralStorage* spolupracuje s třídou *LocalStorage*. Zatímco *CentralStorage* zajišťuje změny objektů v lokálním úložišti i jejich synchronizaci, třída *LocalStorage* se stará pouze o změny v lokálním úložišti. V některých speciálních případech nechceme položku synchronizovat, ale pouze uložit v lokálním úložišti. Může se jednat např. o obrázek mapového objektu. Při přijetí změny obrázku ze serveru musíme nastavit atribut `hasPicture` souvisejícího mapového objektu, ale tuto změnu již nemusíme posílat na server. Na serveru se obdobná úprava provede automaticky. V takovém případě stačí použít následující příkaz, který změnu uloží pouze v lokálním úložišti.

```
storage.local.update(layer)
```

Pro načítání objektů do kontrolerů jsem vytvořil třídy *DatabaseTableViewController* a *DatabaseCollectionViewController*. Obsahují funkce pro načítání dat z centrálního úložiště do tabulky nebo dlaždicového výpisu, jako např. funkci `tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)` a další funkce z delegátů *UITableViewDataSource* a *UICollectionViewDataSource*. Dědí od tříd *BaseTableViewController* a *BaseCollectionViewController* a využívají třídu *BaseControllerActions*. Tyto nadřazené třídy obsahují kromě funkcí pro načítání obsahu z databáze do kontrolerů také další obecné funkce používané ve všech kontrolerech. Jako příklad zde uvedu funkci pro kontrolu, zda došlo ke změně uživatelského jména (je tedy potřeba provést úplnou synchronizaci), nebo funkci pro zobrazení uvítací přihlašovací obrazovky při prvním spuštění aplikace.

4.4 Mapování dat pro synchronizaci a export

Každý objekt má jiné atributy a jinou strukturu synchronizovaných dat. Pro snadnou definici a tzv. namapování atributů z dat ve formátu *JSON* do lokální databáze a opačným směrem jsem vytvořil ve třídě *CentralStorage* funkce `getMapping`, `getOutputMapping`, `getRelationsURLMapping`, `getCreateURL`, `getItemURL`, `getItemByURLParts` aj. Tyto funkce jsou následně přetíženy v potomcích třídy *CentralStorage* a je tak možné definovat zvlášť pro každý objekt způsob konverze dat. Následuje ukázka třídy *FeatureStorage* obsahující přetížení funkce `getMapping`. Funkce `getOutputMapping` není přetížena, protože vstupní i výstupní data jsou shodná a není třeba implementovat totožné vstupní i výstupní mapování. Dále je přetížena funkce `getRelationsURLMapping`, která určuje mapování relací objektu na další objekty podle částí adresy URI. V následující ukázce se použije třetí část URI adresy pro spojení objektu *Feature* s vrstvou. Třetí část URI adresy pro objekty typu *Feature* je identifikátor vrstvy, taková URI může vypadat např. takto `/layer/username/l123456/feature/12`.

```
class FeatureStorage: CentralStorage{
...

override func getMapping() -> NSDictionary? {
    return [
        "hash": "hashString",
        "id": "name", // interpret id as name for unity with other
            entities.
        "properties": [
            "has_picture": "hasPicture",
            "created": "created",
            "last_edit": "lastEdit",
            "precision": "precision"
        ],
        "geometry": [
```

```
            "type": "type",
            "coordinates": "geometry"
        ],
        "bbox": "bbox"
    ]
}

override func getRelationsURLMapping() -> NSDictionary? {
    return [
        3: "Layer"
    ]
}
```

Přetížení funkce `getItemByURLParts` umožní získat odkaz na objekt *Feature* na základě URI. Použije se třetí a pátá část URI pro jednoznačnou identifikaci objektu *Feature*. Třetí část obsahuje identifikátor vrstvy a pátá část relativní identifikátor mapováho objektu *Feature* (např. `/layer/username/l123456/feature/12`). Toto se používá pro nalezení objektu při příchozí změně ze serveru.

Naopak funkce `getCreateURL` a `getItemURL` slouží pro vytvoření URI objektu. Tyto funkce se využívají při odesílání změn na server.

```
override func getCreateURL(item: NSManagedObject) -> String {
    let feature = item as! Feature
    let layer = feature.layer
    return "/layer/"+layer.scheme+"/"+layer.name+"/feature"
}

override func getItemURL(item: NSManagedObject) -> String {
    let object = item as! Feature
    return "/layer/"+object.layer.scheme+"/"+object.layer.name+"/
       feature/"+object.name
}

override func getItemByURLParts(urlParts: [String]) -> NSManagedObject?
    {
    if((urlParts.count)==6){
        return getByLayerAndFeature(urlParts[3], featureName: urlParts
           [5])
    }
    return nil
}
```

Třída *FeatureStorage* obsahuje dále přetížení funkcí `newEntity` pro nastavení výchozích hodnot a přetížení funkce `getDefaultSort` pro určení výchozího řazení objektů typu *Feature* ve výpisech.

```
override func newEntity() -> NSManagedObject {
    let item = super.newEntity() as! Feature
    item.created = NSDate()
    item.lastEdit = NSDate()
    return item
```

```
}

override func getDefaultSort() -> [NSSortDescriptor] {
    return [NSSortDescriptor(key: "name", ascending: true)]
}

...
}
```

Třída *FeatureStorage* obsahuje také funkce pro načítání objektů způsobem specifickým pro mapové objekty, např. funkce `getAllForLayer` slouží pro načtení všech objektů pro danou vrstvu (kromě těch určených ke smazání při synchronizaci), funkce `getByLayerAndFeature` získá objekt *Feature* podle vrstvy a identifikátoru *Feature*.

```
func getAllForLayer(layer: Layer)->NSFetchedResultsController{
    let predicate = NSPredicate(format: "("+getDefaultPredicate()+")
        AND (layer = %@)", layer)
    return local.getAll(entityName, predicate: predicate, sort:
        getDefaultSort(), delegate: delegate)
}

func getByLayerAndFeature(layerName: String, featureName: String)->
   Feature?{
    let predicate = NSPredicate(format: "(layer.name = %@) AND (name=%@
        )", layerName, featureName)
    let items = local.getAll(entityName, predicate: predicate, sort:
        getDefaultSort(), delegate: delegate)
    if(items.fetchedObjects?.count > 0){
        return items.fetchedObjects?.first as! Feature?
    }
    return nil
}
```

Pro samotnou konverzi dat, přetypování, uložení a načítání jsem vytvořil třídu *DataMapping* a její potomky *JSONDataMapping* a *KMLDataMapping*. Třída obsahuje funkce, které zpracují definici mapování z potomka třídy *CentralStorage*, zpracují přijatá data, resp. načtená data, zajistí jejich správné přetypování a konverzi. Datový typ je určen podle datového typu definovaného pro danou položku v *Core Data*. Prázdné hodnoty (nilové) jsou při konverzi vynechány. Díky tomu je možné použít stejnou definici mapování pro `POST` i další typy požadavků. V případě `POST` požadavku nejsou v okamžiku vytvoření u většiny objektů známy všechny atributy a prázdné atributy nechceme na server posílat. Jedná se například o identifikátor objektu nebo *hash* hodnotu. Tyto hodnoty jsou přijaty až ze serveru po vyřízení synchronizačního požadavku.

Překrytím třídy *DataMapping* lze zajistit výstup do různých formátů. V mé aplikaci je používán formát *JSON* pro synchronizaci se serverem (třída *JSONDataMapping*). V budoucnu je plánováno přidat další formáty pro export dat do souboru, jako např. jazyk *KML*.

4.5 Kompletní synchronizace

Při prvním přihlášení uživatele je třeba provést kompletní synchronizaci. Ta se provádí speciálním požadavkem na server s URI adresou `/sync/begin`. Server zašle odpověď obsahující veškerá data přístupná přihlášenému uživateli, nové ID zařízení a uživatelské kontaktní informace. Po dobu provádění kompletní synchronizace musí být pozastavená rozdílová periodická synchronizace (bude popsána dále).

Pokud jsou data úspěšně přijata, smažou se všechny objekty v úložišti a postupně se vytvoří nové objekty podle přijatých dat. Je zde opět využito mapování pomocí třídy *DataMapping*.

Kompletní synchronizaci je třeba provést i v případě vypršení ID zařízení na serveru. Zařízení se o tom dozví na základě chybové odpovědi `460 Device ID Lost`, kterou obdrží při jakémkoliv požadavku na server. V tomto případě však nedojde v zařízení ke smazání všech dat, ale o sloučení přijatých dat s aktuálními objekty v úložišti. Toto řešení je kvůli zachování dat, které uživatel naměřil před získáním nového ID zařízení při měření v offline režimu.

Ve všech ostatních případech se provádí kvůli úspoře datových přenosů a kvůli rychlosti zpracování pouze rozdílová synchronizace. Rozdílová synchronizace je prováděna prostřednictvím synchronizační fronty, jak je popsáno v následující sekci.

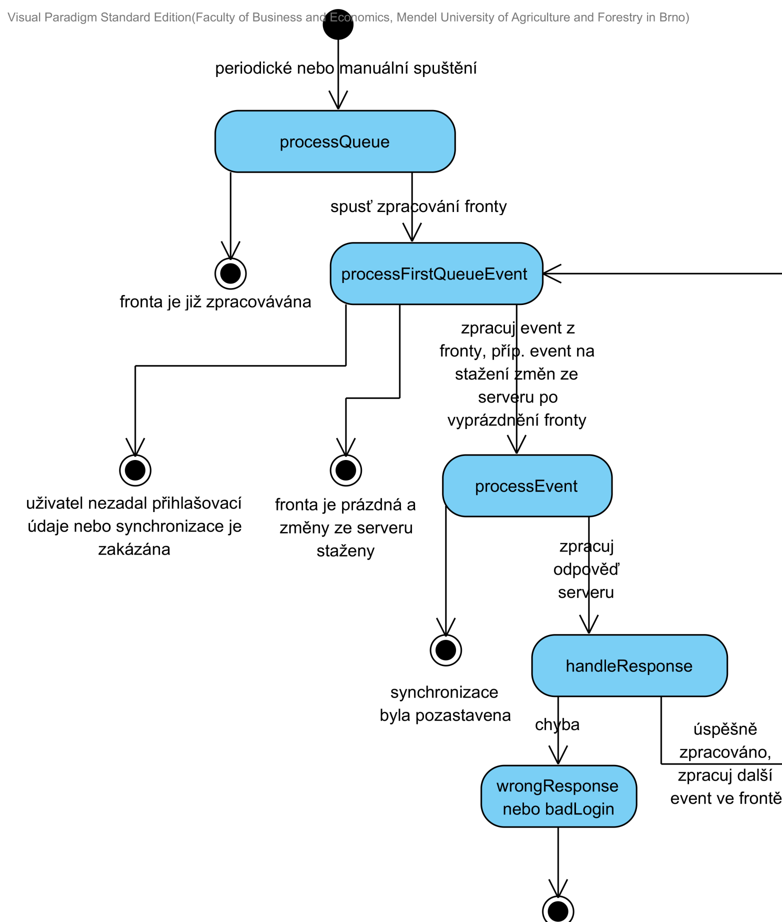
4.6 Synchronizační fronta

Synchronizační fronta zajišťuje uspořádání a zpracování série požadavků na server. Jedná se o nové změny ze zařízení nebo požadavek na stažení změn ze serveru. Synchronizační fronta tedy implementuje rozdílovou synchronizaci.

Jak již bylo uvedeno, požadavky je třeba vyřizovat v určitém pořadí sériově. Fronta požadavků je implementována ve třídě *SyncQueue*. Tato třída obsahuje funkce pro vytvoření nové události (databázový objekt *Event*), funkce pro spouštění a zastavování vyřizování požadavků ve frontě, funkce pro prvotní zpracování úspěšných i neúspěšných odpovědí serveru.

Při zpracování požadavku na server se nejprve vytvoří instance úložiště podle typu synchronizačního požadavku (potomek *CentralStorage*). Tato instance je následně požádána o vygenerování dat a sestavení URI na základě obdrženého objektu a typu požadavku (*create*, *put*, *delete*). Tyto vygenerované údaje jsou odeslány na server. Samotný HTTP požadavek na server je zajištěn třídou *ServerRequest*. Odesílaná data i odpověď jsou pro snazší předávání mezi objekty obaleny do struktury *SyncRequestData*.

Po obdržení odpovědi se data pošlou opět do instance příslušného úložiště, kde je odpověď zpracována vč. vyřešení konfliktních situací. Toto je implementováno převážně v samotné třídě *CentralStorage*. V potomcích může být toto zpracování přetíženo pro jiný způsob zpracování dat než mapováním z formátu *JSON*. Je to využito v případě synchronizace obrázků, které nejsou přenášeny ve formátu *JSON*, ale jako binární data.



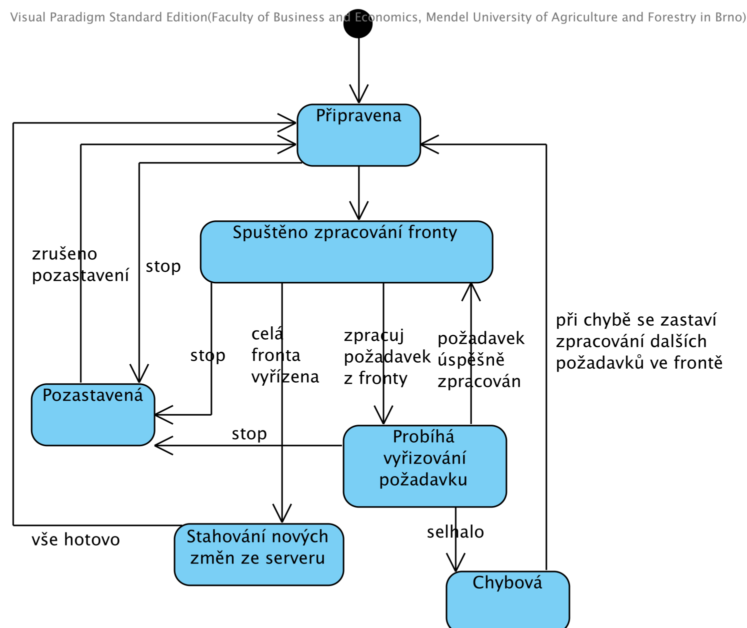
Obrázek 15: Diagram znázorňující zpracování fronty.

Celý proces zpracování fronty je znázorněn v diagramu (obr. 15). Na dalším diagramu jsou stavy synchronizační fronty (obr. 16).

4.7 Manuální a periodická rozdílová synchronizace

Pokud uživatel provede nějakou akci, po které by měla bezprostředně následovat synchronizace, zavolá se funkce pro vyřízení synchronizační fronty. Fronta se však začne vyřizovat pouze, pokud již není aktuálně vyřizována. Tím se předejde nechtěnému nahodilému paralelnímu zpracování fronty. Synchronizační fronta je implementována podle návrhového vzoru Singleton.

Kromě manuální synchronizace po akci uživatele se provádí ještě periodická synchronizace (třída *PeriodicSync*). V intervalu, který uživatel může měnit v nastavení aplikace, se volá funkce ve třídě *SyncQueue*, která spustí vyřizování fronty, pokud již neprobíhá.



Obrázek 16: Diagram zobrazující stavy, ve kterých se může synchronizační fronta nacházet.

4.8 Příchozí změny

V rámci periodické synchronizace se vytváří i speciální typ požadavku pro získání nových změn ze serveru. Tento požadavek se vytváří přímo v rámci třídy *SyncQueue* po vyřízení všech předchozích požadavků, přičemž není ukládán do fronty v úložišti, aby nedošlo k množení těchto požadavků. Pro obsluhu přijatých změn ze serveru slouží třída *IncomingEventResolver* jako potomek *CentralStorage*. Tato třída projde všechny přijaté změny a pro každou přijatou změnu identifikuje typ objektu. Na základě typu objektu vytvoří instanci modulu obsluhující daný objekt. V této instanci je následně zavolána funkce vyřizující příchozí změny ze serveru. V rámci této funkce (součástí *CentralStorage*) se rozlišuje typ změny (*create*, *update* nebo *delete*) a je využito mapování dat pro uložení do lokálního úložiště. Tuto funkci je možno přetížit v potomcích a zajistit tak speciální vyřízení příchozí změny. Toho je využito např. ve třídě *FeaturePictureStorage* pro smazání staženého obrázku při změně obrázku na serveru.

```
override func handleIncomingPutEvent(event: JSON) {
    super.handleIncomingPutEvent(event)
    // delete local image after change on server
    let item = getItemByURL(event["uri"].stringValue) as?
        FeaturePicture
    item?.image = nil
}
```

4.9 Synchronizace obrázků

Obrázky mapových objektů tvoří speciální případ úložiště. Každý mapový objekt může mít maximálně jeden obrázek. Samotný soubor není stahován ze serveru automaticky v rámci periodické synchronizace, ale až na požádání uživatele (po kliknutí na tlačítko *Download* v kontroleru zobrazujícím detail mapového objektu – *FeatureDetailController*).

V rámci periodické synchronizace je stahován pouze *hash*, na základě kterého zařízení pozná, že se obrázek na serveru změnil. Po změně obrázku na serveru zařízení smaže stažený obrázek, pokud existuje, a zobrazí tlačítko pro stažení nového obrázku.

Jestli daný mapový objekt má obrázek, zařízení rozpozná podle jeho atributu `hasPicture`. Tento atribut však není synchronizován při změně obrázku, pouze při změně nadřazeného mapového objektu *Feature*. Pokud tedy do zařízení přijde ze serveru změna na obrázek, zařízení musí náležitě upravit i atribut `hasPicture` v souvisejícím objektu *Feature*. Aby tato změna byla přirozená, využil jsem k tomu přetížení funkcí v objektech *Feature* a *FeaturePicture* následovně.

V první ukázce se jedná o část třídy *FeaturePicture*. Pokud uživatel smaže obrázek, dojde ke skrytí objektu (atribut `hidden`). Souvisejícímu objektu *Feature* se náležitě aktualizuje atribut `hasPicture`.

```
/**
 Updates the related feature's hasPicture property if FeaturePicture is
      going to be deleted.

 - parameter key: Changed property key.
 */
override func didChangeValueForKey(key: String) {
    if(key == "hidden"){
        feature?.hasPicture = !hidden
    }
    super.didChangeValueForKey(key)
}
```

Druhá ukázka je ze třídy *Feature*. Pokud dojde ke smazání objektu permanentně (např. přijetím změny ze serveru), změní se tím atribut `picture`. Pokud je jeho hodnota rovna `nil`, i atribut `hasPicture` bude `nil`.

```
/**
 Update the hasPicture property according to existance of a related
     FeaturePicture object.

 - parameter key: Changed property key.
 */
override func didChangeValueForKey(key: String) {
    if(key == "picture"){
        hasPicture = picture != nil
    }
    super.didChangeValueForKey(key)
```

```
}
```

Databázový engine *SQLite*, na kterém je postaven framework *Core Data*, není příliš vhodný na ukládání větších objemů dat. Proto ukládám obrázky do souborového systému. V databázi se uchovává pouze cesta k souboru a *hash* hodnota. Pro skrytí implementace ukládání a načítání jsem toto umístil přímo do třídy reprezentující objekt *FeaturePicture* (ukázka viz sekce 4.2 Objektové úložiště).

Obrázky jsou v souborovém systému tříděny do složek podle vrstev. Je to kvůli tomu, aby v případě smazání vrstvy bylo možné jednoduše smazat všechny související obrázky. Mazání obrázkových souborů se děje automaticky při smazání objektu v databázi. Je k tomu využito přetížení funkce `didSave` ve třídách *Layer* a *FeaturePicture*.

4.10 Geometrie

Geometrie je v databázi ukládána jako řetězec ve formátu *JSON*. Pro načítání objektů do mapy, vypočítávání *bounding boxu*, automatické uzavírání polygonu, postupné přidávání a odebírání bodů z linií a polygonů je zapotřebí různým způsobem k datům reprezentujícím geometrii přistupovat a různě je načítat. Například na jednom místě je potřeba geometrie jako pole, jinde jako cesta *GMSPath*.

Další komplikací je specifikace formátu *GeoJSON* pro různé typy mapových objektů. Zatímco bod obsahuje pouze pole s dvěma souřadnicemi, polygon obsahuje tříúrovňové pole definující vnější obvod polygonu a otvory uvnitř polygonu. Pro sjednocení přístupu k různým typům mapovým objektů a jejich editaci jsem vytvořil třídu *Geometry*. Z této třídy uvedu zde jako příklad funkci, která umožňuje vrátit geometrii objektu jako jednorozměrné pole bez ohledu na typ objektu. To je užitečné při dalším zpracování geometrie bez nutnosti řešit typ mapového objektu.

4.11 Uživatelské rozhraní

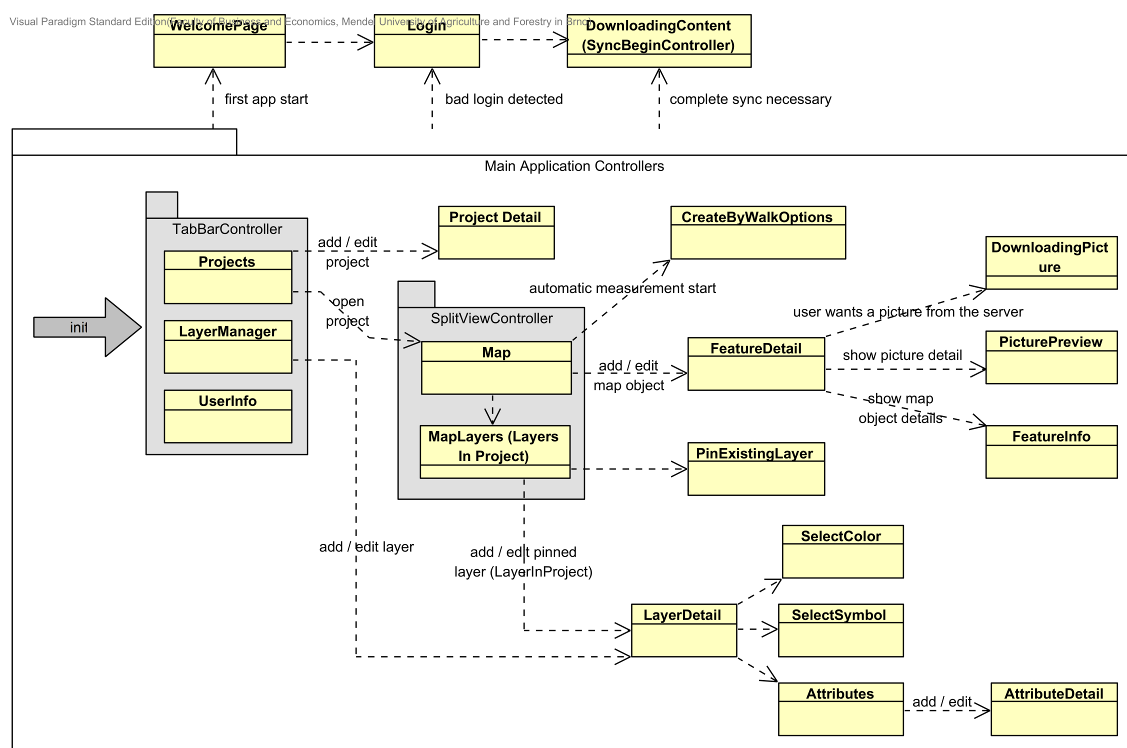
4.11.1 Uspořádání a návaznost obrazovek

Na obr. 17 jsou znázorněny všechny podstatné kontrolery aplikace a vazby mezi nimi. Pro snadnou čitelnost není v obrázku obsah jednotlivých kontrolerů.

V diagramu jsou kontrolery rozdělené na hlavní kontrolery a pomocné kontrolery. První kontroler, který se zobrazí přihlášenému uživateli, je *UITabBarController* s třemi kartami – projekty, správcem vrstev a informacemi o uživateli. Na tyto kontrolery navazují obrazovky pro editaci projektů, vrstev atd. Kontrolery jako *ProjectDetailController* nebo *LayerDetailController* slouží zároveň pro přidání nové položky i její editaci. Kontroler *LayerDeatilController* slouží zároveň pro editaci vrstev (*Layer*) i připnutých mapových vrstev (*LayerInProject*). V rámci tohoto kontroleru se programově skrývá sekce s nastavením vzhledu (tato sekce je zobrazená pouze pro editaci vrstvy připnuté do projektu).

Kontroler *PinExistingLayer* obsahuje výpis vrstev pro připinování do projektu. Tento kontroler je potomkem kontroleru pro správu vrstev (*LayersController*). Má pouze odlišné chování, výpis je stejný. Naproti tomu *LayersInProjectController* nemá nic společného s *LayersController*, protože vzhled i chování je značně odlišné a dědičnost by zde neměla žádný význam.

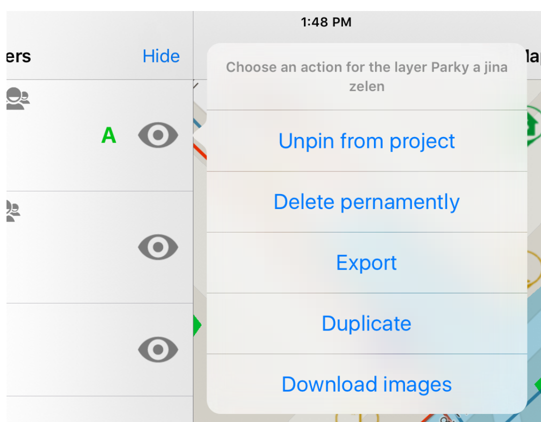
Pomocné kontrolery slouží k zobrazení úvodních informací při prvním spuštění aplikace (*WelcomePageController*), pro zobrazení formuláře pro zadání uživatelských údajů (*LoginController*) a pro zobrazení průběhu kompletní synchronizace (*SyncBeginController*). Tyto kontrolery jsou iniciovány v případě potřeby v některém z hlavních kontrolerů. Např. pokud jsou kdekoliv v aplikaci detekovány špatné přihlašovací údaje, zobrazí se *LoginController* apod.



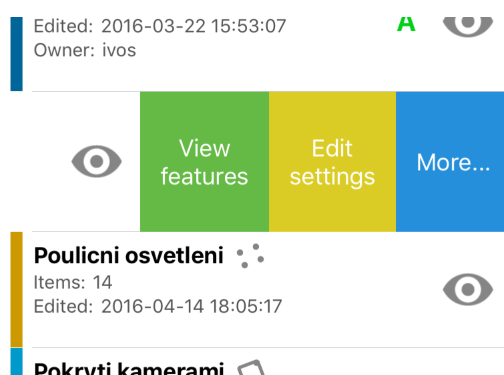
Obrázek 17: Zjednodušený diagram přechodů mezi jednotlivými obrazovkami aplikace

4.11.2 Layout

Vzhledem k vysoké komplexnosti aplikace – velkému množství možností, akcí a zobrazovaných informací – bylo nutné zjednodušit obsah jednotlivých kontrolerů a rozčlenit informace do více obrazovek. Méně důležité nabídky a informace jsou skryty a zobrazují se až samostatně po rozkliknutí nebo otevření detailu položky. V tabulkových a dlaždicových výpisech jsou využívány dialogová okna ve stylu *AlertSheet* pro zobrazení více akcí (obr. 18). Pro úsporu místa jsou v tabulkových výpisech



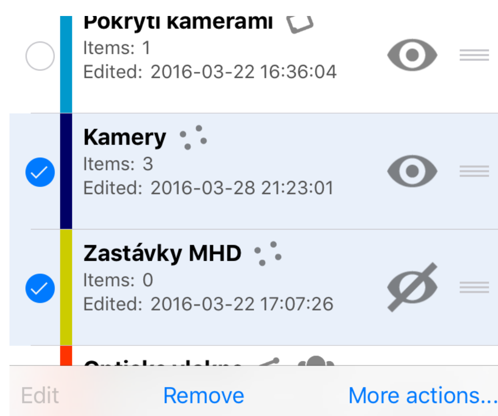
Obrázek 18: Dialogová nabídka s dalšími akcemi pro mapovou vrstvu.



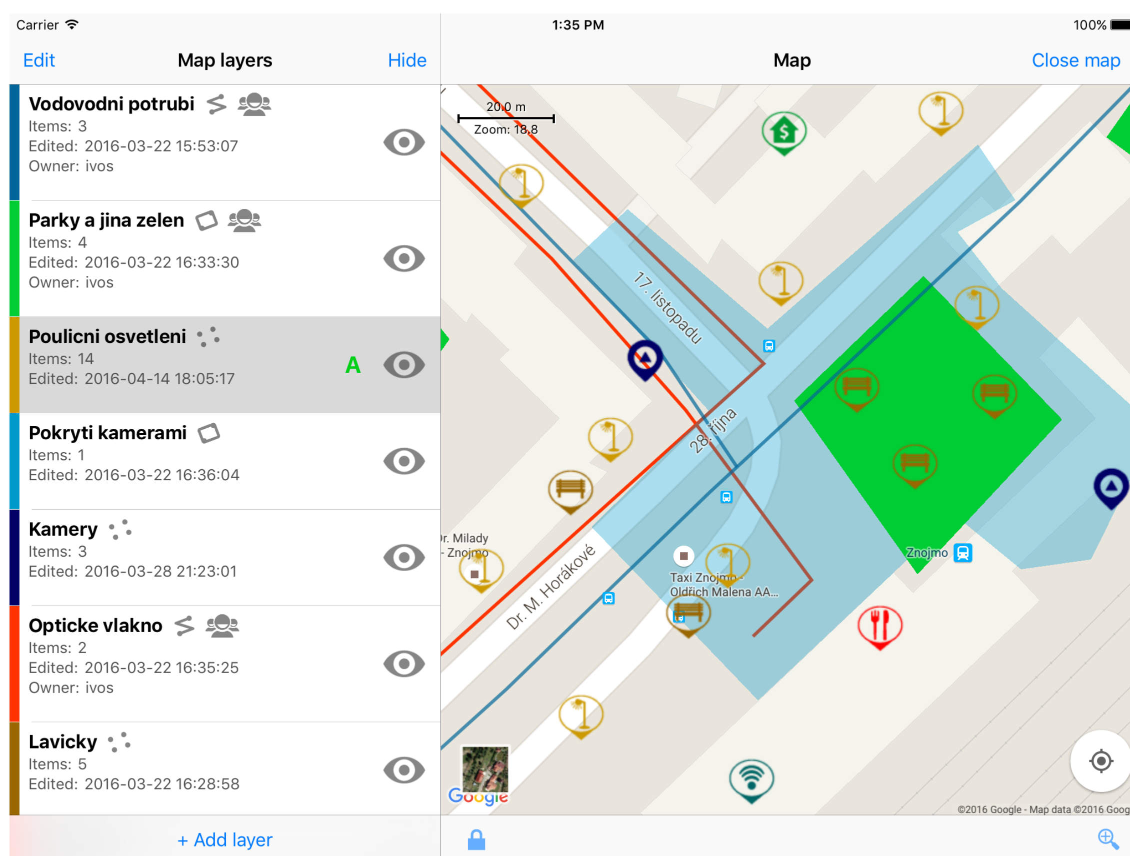
Obrázek 19: Akce mapové vrstvy zobrazené po přetažení položky v tabulkovém výpisu doleva.

využívána gesta. Po přetažení položky doleva se na pravé straně objeví akce pro danou položku (obr. 19). Hromadné akce se aktivují pomocí tlačítka *Edit* umístěného vlevo nahoře. Následně uživatel může označovat položky. Na spodu se po označení položek vysune lišta s akcemi pro zvolené položky (obr. 20). V detailu mapového objektu se akce obrázku zobrazují také v dialogovém oknu po kliknutí na samotný obrázek.

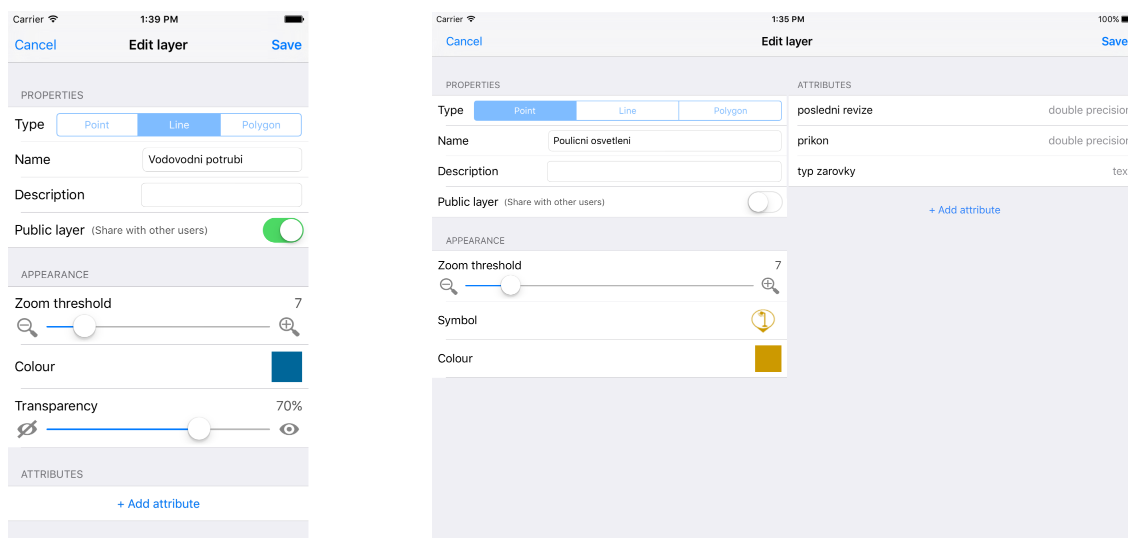
Pro lepší využití velkého displeje na tabletech má aplikace na některých obrazovkách rozdílný layout pro menší a větší displeje. Na obr. 21 je hlavní zobrazení s mapou a připnutými vrstvami v panelu nalevo, který lze skrýt. Na menších displejích se kontroler s mapou a kontroler s vrstvami zobrazují jednotlivě přes celý displej zařízení. Toho bylo docíleno s využitím objektu *Container* a programovými úpravami layoutu - změnami tzv. *constraints*. Jiný případ adaptivního layoutu je při editaci vrstvy. Na velkém displeji jsou dva stejně široké sloupce, vlevo vlastnosti vrstvy, vpravo atributy. Na menším displeji se atributy zobrazí pod vlastnostmi vrstvy (obr. 22).



Obrázek 20: Hromadné akce pro označené řádky v tabulkovém výpisu.



Obrázek 21: Zobrazení mapy s postranním panelem na tabletu. Postranní panel s připnutými vrstvami lze skrýt. Ukázka zobrazuje fiktivní inventarizaci města. Na mapě lze vidět bodové objekty jako pouliční osvětlení, lavičky, kamery (tmavě modré) apod. Modré a červené linie zobrazují inženýrské sítě. Zelený polygon představuje park a světle modré polygony znázorňují oblasti pokryté kamerovým systémem. Každý objekt v mapě může mít atributy, např. datum instalace daného předmětu, datum příští revize, elektrotechnické parametry (např. příkon lampy pouličního osvětlení, typ použité žárovky) nebo kontaktní údaje na správce.



Obrázek 22: Editace mapové vrstvy na *iPhonu* (vlevo) a dvousloupcové zobrazení na *iPadu* (vpravo). Na menších displejích jsou atributy umístěny pod vlastnostmi vrstvy, na větších displejích jsou vlastnosti v levém sloupci a atributy vrstvy v pravém. Editace existujících atributů probíhá stejně jako v tabulkových výpisech, tedy pomocí gesta přetažení řádku na stranu.

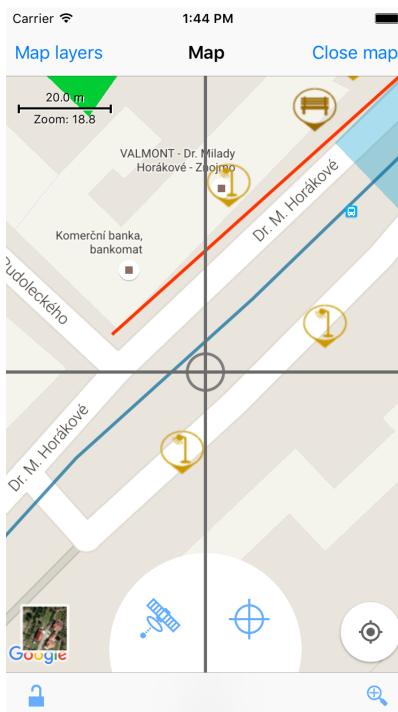
4.11.3 Správa mapy

Pro abstrakci funkcí pro práci s mapou jsem vytvořil třídu *MapManager*. Třída obaluje rozhraní *Google Maps SDK*. Díky tomuto zapouzdření by případná budoucí změna mapové služby znamenala pouze změny v této třídě a nenarušila by ostatní funkčnost aplikace. Třída *MapManager* kromě správy mapových objektů, jejich stylizaci a operací týkajících se aktuálního výřezu mapy obsahuje také funkce pro přidávání dočasných objektů. Tyto dočasné objekty jsou využívány při přidávání nových dosud neuložených objektů do mapy. Třída *MapManger* obsahuje funkce pro přidání editačních symbolů a markerů. V souvislosti s dočasnými objekty a editačními symboly nastala komplikace při obnovování mapových objektů po příchozí změně ze serveru a současnému přidávání/editování objektu. Bylo nutné ošetřit, aby se po obnovení mapových objektů obnovily i dočasné objekty a editační symboly.

Další funkcí třídy *MapManager* je filtrování zobrazovaných objektů podle limitu přiblížení vrstvy. Ten určuje, od jaké úrovně zoomu jsou objekty v dané vrstvě viditelné. Pro snazší orientaci uživatele byla také implementována funkce `zoomToLayer`, která změní výřez mapy a úroveň přiblížení tak, aby byly viditelné všechny objekty aktuálně aktivní vrstvy. Pokud je tato úroveň přiblížení nižší než minimální limit přiblížení pro zobrazení objektů a uživatel by neviděl žádný objekt, použije se místo toho pozice prvního objektu v aktivní vrstvě s přiblížením odpovídajícím limitu přiblížení vrstvy. Při vypočítávání se využívá *bounding box* linií a polygonů.

4.11.4 Měření dat

Přidávání mapových objektů probíhá vždy v rámci jedné aktivní vrstvy. Aktivní vrstva se zvolí kliknutím na položku na obrazovce s připnutými vrstvami, tzv. *Map layers*. Pro zamezení nechtěné editace je mapa ve výchozím stavu zamknuta. Po kliknutí na ikonu zámku dojde k jejímu odemčení. Při změně aktivní vrstvy se vrstva automaticky zamkne.

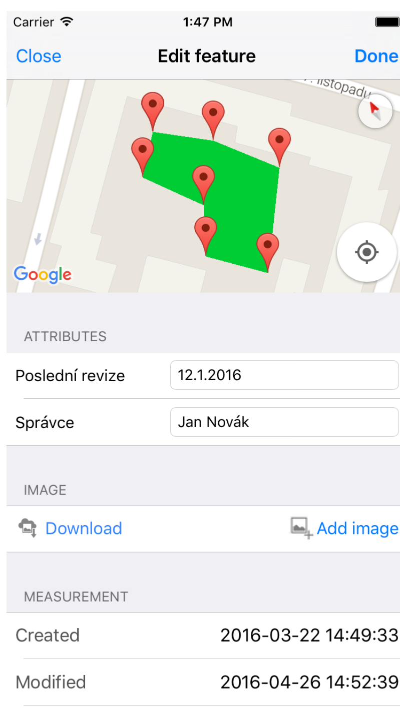


Obrázek 23: Ovládací prvky pro přidání bodu.

Na obr. 23 jsou zobrazeny ovládací prvky pro přidání bodu do mapy. Bod lze přidat podle aktuální polohy zařízení nebo manuálně na střed aktuálního pohledu mapy. Po přidání se bod zobrazí v mapě a je označen žlutým kolečkem. Následně se zobrazí kontroler pro editaci mapového objektu, na větším displeji v pravém postranním panelu s mapou nalevo, na menším displeji přes celou obrazovku s mapou nahoře (obr. 24).

Linie a polygony jsou přidávány podobným způsobem. Pro znázornění editovaného objektu se používají markery v rohových bodech mapového objektu. U linií a polygonů lze kromě manuálního měření zahájit i automatické periodické měření, kdy dochází k přidávání nových bodů do geometrie objektu podle lokace uživatele v nastaveném intervalu. Při ukládání se kontroluje minimální počet bodů v geometrii (linie musí mít minimálně dva, polygon tři).

Editace mapových objektů je realizována klepnutím na daný objekt. Právě editovaný bod je označen žlutým kolečkem, editovaná linie a polygon má rohové body označeny červenými markery.



Obrázek 24: Editace mapového objektu na iPhonu.

4.11.5 Validace dat

V této sekci uvedu různé příklady validace v mé aplikaci.

Pro validaci dat zadaných uživatelem jsem využil knihovnu *SwiftValidator*. V rámci této knihovny jsem našel chybu ve funkci pro kontrolu číselných hodnot. Chybu jsem opravil, můj *pull request* byl schválen a *commit* úspěšně spojen s hlavním repozitářem.

Po zadání neplatné hodnoty se uživateli zobrazí červený rámeček kolem textového pole. V mé aplikaci využívám především kontrolu prázdnosti a kontrolu, zda je zadáno validní číslo. Dále se kontroluje vlastník vrstvy u sdílených vrstev. Není možné editovat a mazat vrstvu, u které se neshoduje vlastník se současným uživatelem. Přidávat mapové objekty i editovat již existující však u takové vrstvy lze.

U některých objektů je třeba kontrolovat unikátnost názvu (např. u vrstev). Tato kontrola probíhá pouze v zařízení, nikoliv na serveru. Může tedy dojít k existenci více objektů se stejným názvem v případě vytvoření takových objektů na více zařízení naráz před vzájemnou synchronizací. Běžně by však k této situaci docházet nemělo a stačí proto kontrola unikátnosti v zařízení. Ta se provádí těsně před uložením v případě detekování změny názvu. Kontrola se provádí dotazem do centrálního úložiště. K tomu slouží funkce `getByTitle(title: String)` ve třídě *CentralStorage*.

Pro snadné přidání nových vrstev je implementován *placeholder* pro název vrstvy. Není tedy třeba název vrstvy zadávat, název se předvyplní automaticky. *Placeholder* je ve formátu `LayerXX`, kde `XX` je první neobsazené číslo hleda-

né vzestupně od čísla rovnajícího se počtu vrstev v aktuálním projektu (funkce `getUniqueTitle()->String` ve třídě *LayerStorage*).

Před uložením každého objektu je třeba zkontrolovat, zda již nebyl editovaný objekt smazán vlivem příchozích změn ze serveru. Pokud ano, zobrazí se uživateli dialog s informací a uložení se zruší.

V případě hromadných akcí je potřeba uložit odkazy na označené objekty do pomocného pole a až poté zobrazit uživateli dialog s výběrem akce nebo potvrzením smazání. Pokud by totiž v mezičase, kdy uživatel rozmýšlí nad volbou v dialogu, přišly změny ze serveru, mohlo by dojít ke změně položek ve výpisu a uživatel by pak provedl akci pro jiné objekty než původně zamýšlel.

4.11.6 Dočasné atributy

Při editaci vrstvy lze zároveň editovat i její atributy. Aby uživatel měl možnost vrátit změny atributů, nejsou změny atributů ihned ukládány do úložiště a synchronizovány, ale jsou měněny pouze jejich kopie. O správu dočasných atributů se stará třída *TempLayerAttributes*. Po uložení vrstvy tato třída porovná dočasné atributy s těmi uloženými (pokud existují) a provede příslušné akce v centrálním úložišti (vytvoření, úpravu nebo smazání). Pokud je smazán atribut vlivem synchronizace a zároveň stejný atribut editován uživatelem, je tento atribut při ukládání přeskočen.

Podobným způsobem jsou zpracovávány hodnoty atributů v editaci mapového objektu.

4.11.7 Obnovování dat

Pro obnovování dat ve výpisech lze využít delegát *NSFetchedResultsControllerDelegate*, který informuje o všech změnách objektů, které byly před tím načteny pomocí příslušné instance *NSFetchedResultsController*. Lze tak efektivně aktualizovat jen nezbytné řádky v *UITableView* vč. animace. Nicméně delegát informuje pouze o změnách, které jsou přímo v entitě, na kterou byl dotaz do lokálního úložiště vytvořen. Neinformuje o změnách souvisejících objektů, které jsou s hlavními objekty v relaci. To je potřeba v situacích, kdy se ve výpisu objevují i data ze souvisejícího objektu, např. výpis s připnutými vrstvami zobrazuje informace o objektech *Layer*, *LayerInProject* a také počet mapových objektů (*Feature*). Proto ještě bylo třeba zavést druhý způsob obnovování dat, a to kompletní obnovení výpisu. Tento způsob se používá např. po příchozí změně ze serveru. Pro tento účel jsem vytvořil správce notifikací, který využívá třídu *NSNotification*. Jednotlivé instance *UIViewController* se zaregistrují u správce notifikací a sdělí mu, u kterých objektů si přejí hlídat změny. Při příchozích změnách ze serveru se u každé změny uloží do pole typ změněného objektu. Po zpracování všech změn ze serveru se projde pole změněných typů objektů a kontrolery, které si alespoň jeden z těchto typů registrovaly, jsou notifikovány. Po této notifikaci kontroler smaže cache ve své instanci potomka *CentralStorage* a obnoví svůj výpis tak, že znovu načte všechna data a obnoví všechny aktuálně viditelné řádky.

V kontroleru, který obsluhuje výpis, je nutno ošetřit, aby nedošlo zároveň k obnovování výpisu pomocí delegátu a zároveň pomocí kompletního obnovení. V tom případě by nedošlo ke korektnímu načtení dat. Také je třeba pozastavit kompletní obnovování, pokud uživatel je v editačním módu tabulkového nebo dlaždicového výpisu. Při obnovení by totiž došlo ke zrušení uživatelského výběru položek k editaci, příp. by mohlo dojít i k tomu, že uživatel označí některé položky, poté se výpis obnoví a výběr se posune na jiné položky, které uživatel nemá zájem editovat. Obnovení se proto provede až uživatel dokončí editaci.

4.12 Konfigurace

V aplikaci jsem vytvořil konfigurační soubor `config.swift` obsahující struktury definující výchozí hodnoty uživatelského nastavení (jako je URL serveru nebo interval periodické synchronizace), dále jsou zde uloženy konstanty (např. API klíč k *Mapám Google*, formát obrázků odesílaných na serveru, ...).

Tento konfigurační soubor obsahuje také seznam dostupných symbolů a barev, který uživatel může použít pro nastavení vzhledu objektů vrstvy. Umístění v konfiguračním souboru umožní jednoduchou změnu těchto symbolů pro přizpůsobení aplikace konkrétnímu účelu.

5 Diskuze

5.1 Příklady využití

V první části bych zde chtěl uvést příklady využití mé aplikace v praxi. Základním způsobem využití aplikace je inventarizace jakéhokoliv majetku pro podniky všech velikostí, státní správu, neziskové organizace i jednotlivce. Inventarizace může probíhat staticky (offline), tzn. že proběhne měření v terénu, popř. manuální zanesení budov, pozemků, toků a dalších objektů do mapy, poté jsou naměřená data exportovány do formátu jako je *KML* nebo *JSON* a zpracovány v nějakém desktopovém geografickém informačním systému.

Mnohem zajímavější je však využití aplikace včetně synchronizace. Díky tomu je možné přidat do mapování více interaktivity, usnadnit a urychlit komunikaci mezi pracovníky a umožnit jejich spolupráci v rámci jednoho mapového projektu. Může se jednat např. o energetickou společnost, která pomocí aplikace eviduje veškerý majetek (kancelářské budovy a trafostanice jako body, rozvody elektrického proudu jako linie, oblasti pokryté vlastními zdroji jako polygony apod.). Díky synchronizaci je však možné velmi usnadnit nejen evidenci majetku, ale i jeho údržbu a servis. Pracovník zjistí při revizi závadu na sloupu elektrického vedení a tuto závadu ihned zaznamená přes mobilní aplikaci do mapy. Změna se ihned synchronizuje na server a operátor v kanceláři ihned vidí místo a popis závady. Na místo proto vyšle techniky. Technici si mohou ve svých zařízeních zobrazit hlášení závady a jednoduše tak být navigování přímo na místo. Jakmile technici závadu na místě odstraní, zaznamenají změnu stavu sloupu elektrického vedení do mobilní aplikace. Operátorovi v centrále se zobrazí upozornění, že je závada již odstraněna a může provést např. zátěžový test sítě.

Podobný případ využití může být na obecních úřadech nebo jiných státních orgánech pro evidenci majetku, evidenci a aktuální stav pronajímaných bytů nebo seřizování hlasitosti místního rozhlasu. Jeden pracovník bude v terénu zaznamenávat do mapy slyšitelnost rozhlasu v různých místech a druhý pracovník v centrále ihned seřizovat hlasitost. Městská policie může do mapy zanášet nálezy poškozeného majetku podobně jako v případě energetické společnosti.

Podnikatel živící se zemědělstvím vlastní několik polí v různých oblastech a část lesa. Aplikaci využije pro zaznamenávání údajů o sklizni, množství použitého osiva a hnojiv, výnosů apod. Pokud následně data exportuje a zpracuje ve vhodné desktopové aplikaci, může získat cenná historická data, která se dají využít pro vyhodnocení prosperity podnikání, výnosnosti jednotlivých oblastí a pro další efektivní rozhodování a řízení firmy.

Zvláštním případem využití je firma provozující osobní dopravu ve větším městě (taxi služba). Taková firma nepotřebuje evidovat žádný majetek, nicméně může aplikaci využít pro sledování aktuálních pozic vozů. Operátor může sledovat aktuální volné vozy, do mapy vkládat pozice čekajících zákazníků a navigovat nejbližší vozy přímo na místo.

5.2 Náklady a přínosy

Jedním z možných způsobů využití je proces inventarizace, který byl popsán v předchozích částech. Je obtížné vyčíslit konkrétní finanční přínos, protože závisí na typu organizace a množství spravovaného majetku, příp. jiných objektů k evidenci. S využitím mobilní aplikace se celý proces inventarizace zjednodušuje a zvyšuje se pravděpodobnost, že spravovaná data budou více odpovídat realitě. To sníží počet chybných manažerských rozhodnutí, případně i zrychlí tato rozhodnutí díky rychlejšímu transferu dat z terénu k vedení a díky rychlejším možnostem interpretace dat. Data jsou s využitím dalších technologií snadno zálohovatelná a archivovatelná a poskytují tak nesčetné možnosti dalších analýz, které jsou důležité pro strategické plánování.

Velmi důležitá jsou takto nasbíraná data v případě krizových situací. Uvedu zde jako příklad orkán, který způsobí četné škody. Pracovníci záchranných složek musí vyrazit do ulic, zaměřit škody a zhodnotit další možná nebezpečí. V takové situaci i jakýchkoliv podobných (např. při záplavách) je čas esenciální proměnnou. Možnost rychle a přesně nasbírat a sdílet data mezi všemi pracovníky může velmi urychlit proces stabilizace krizové situace a včas lokalizovat, které oblasti a objekty potřebují pomoc nejdříve.

Pro přehled přínosů mé aplikace zde budu uvažovat firmu, která aktuálně provádí mapování v terénu v papírové podobě ručním zapisováním do mapy. Po zavedení mé aplikace by firma ušetřila díky nižší spotřebě papíru, méně častému tisku, redukci nákladů spojených s interpretací dat zapsaných v terénu na papír apod. Výhodou zavedení aplikace by byla také lepší přesnost mapování, menší chybovost, snadnější oprava chybných dat a kooperace více pracovníků v rámci jednoho projektu. Přidaná hodnota pro firmu by byla snadná možnost dodatečné analýzy dat z mobilního zařízení v některém desktopovém geografickém informačním systému a např. predikce budoucích událostí s využitím systémů umělé inteligence.

Bezprostřední jednorázové náklady tvoří výdaje na hardware, tedy zejména nákup mobilních zařízení (telefonů, tabletů a outdoorových pouzder), případně také nákup počítačů pro zpracování dat. Další náklady vzniknou při zaškolování pracovníků. V případě, že podnik bude chtít využívat synchronizaci, bude potřebovat kvalitní připojení k internetu (náklady na paušál mobilnímu operátorovi) a licence na využívání serveru, která se odvíjí od předpokládané zátěže. Některé firmy však již mají potřebná zařízení zakoupená dříve pro jiné účely a počáteční náklady se tím značně sníží.

5.3 Technické zhodnocení

Některé funkce se v mobilních zařízeních dělají lépe, některé hůře. V rámci aplikace jde pohodlně a srozumitelným způsobem pracovat s projekty, vrstvami, atributami, exportovat data apod. Jsou to standardní výpisy a operace, jak je uživatelé znají z mnoha jiných aplikací.

Z uživatelského hlediska je o něco komplikovanější práce s mapovými objekty. Manuální přidávání objektů na malém displeji telefonu a nepřesném dotykovém ovládání vyžaduje odlišný způsob práce s mapovými objekty, než jaký známe z desktopových aplikací. Při automatickém měření prostorových dat pomocí senzorů v zařízení je nutno počítat s určitou nepřesností a prodlevou v budovách, zastavěných oblastech, hustém lese a obecně všude, kde je horší příjem mobilního signálu a signálu z družic GPS.

Dalším nebezpečím je spolupráce více uživatelů na stejném projektu. Při neopatrném zacházení může jeden pracovník smazat jinému pracně nasbíraná data, jejichž měření trvalo celý den. Je potřeba, aby se kooperující pracovníci dohodli na pravidlech úpravy objektů a jasně si vymezili, kdo bude kterou oblast editovat, aby nedocházelo k častým změnám stejných položek u více uživatelů současně.

Z výše uvedených důvodů je potřeba určité zaškolení pracovníků, aby zvládli s aplikací pracovat a bylo to pro ně stejně pohodlné jako zakreslování na papír. Měli by alespoň obecně chápat princip synchronizace, měření dat pomocí GPS apod.

Synchronizace lépe funguje, pokud mají všechny zařízení kvalitní a stabilní připojení k internetu. Pokud vypadne připojení k internetu, data se samozřejmě nesmažou a jsou synchronizovány v okamžiku obnovení spojení. To však může být problém z důvodu možného výskytu velkého množství konfliktů. Pokud bude spolupracovat více pracovníků a bude jim střídavě selhávat připojení k internetu, mohou být zmatení z prodlev a zaznamenat jeden objekt vícekrát a způsobit synchronizační konflikty a nejednotné měření.

5.4 Další vývoj

V budoucnu bude vývoj aplikace pokračovat směrem k rozšiřování možností práce s daty v mobilním zařízením tak, aby se aplikace více přibližovala plnohodnotným desktopovým geografickým informačním systémům, ale zároveň aby byla zachována přehlednost a snadné ovládání na malých dotykových obrazovkách.

Nyní zde uvedu demonstrativní výčet funkcí, které budou pravděpodobně v blízké budoucnosti implementovány: vyhledávání míst v mapě podle názvů (tzv. *geocoding*), filtrování vrstev ve výpisu, hledání mapových objektů podle textu a číselných hodnot obsaženého v jejich atributech, možnost přidat atribut k více vrstvám současně (nový správce atributů podobný nynějšímu správci vrstev), vícenásobný výběr a hromadná editace mapových objektů přímo v mapě, export prostorových dat do dalších formátů apod.

Značné rozšíření možností využití aplikace v praxi by přinesla funkce pro získávání dat z dalších senzorů a komponent (např. měření hluku pomocí mikrofonu pro budování senzorických sítí) nebo funkce pro přepínání mezi podlažími pro inventarizaci i uvnitř budov.

Dále uvedu několik vylepšení týkajících se synchronizace. Selektivní přeskakování položek v synchronizační frontě by sloužilo k urychlení synchronizace a možnosti přeskočit synchronizační položky v případě chyby. Synchronizační fronta by rozu-

měla hierarchii položek a mohla by rozhodovat, které lze přeskočit a které na sebe navazují a nelze je přeskočit. Díky tomu by bylo možné paralelně zpracovávat více položek naráz.

Aktuálně se seznam již synchronizovaných položek ukládá na serveru. To může být problém v případě výpadku spojení při přenášení dat ze serveru do zařízení. Data nemusí dojít kompletní, ale přitom na serveru se již zapsala jako odeslaná a do zařízení tedy již nikdy nedojdou. Bylo by tedy vhodné zrušit tento způsob evidence synchronizovaných položek na serveru a zavést uchovávání identifikátoru poslední synchronizační události přímo v zařízení. Na serveru by pak byl pouze seznam událostí s identifikátory.

Z důvodu vypršení identifikátoru zařízení na serveru je jednou za čas potřeba provést kompletní synchronizaci. To může zahrnovat velké datové přenosy a časově náročné zpracování přijatých dat v zařízení. Proto by bylo efektivnější zavést novou formu rozdílové synchronizace, při které by došlo k porovnání dat v zařízení s daty v databázi na serveru a poslání pouze rozdílných dat. To by mohlo být realizováno formou hierarchického srovnání stromové struktury obsahující hodnoty *hash* (identifikátory poslední změny) jednotlivých objektů ze zařízení s aktuálními hodnoty *hash* z databáze serveru.

U důležitějších objektů jako jsou projekty nebo vrstvy by bylo vhodné v případě konfliktu zobrazovat uživateli dialog s výběrem akce pro vyřešení konfliktu. Aktuálně je implementováno pravidlo první změny a toto pravidlo může způsobit nežádoucí přepsání dat a zmatení uživatele.

Při dalším vývoji tedy bude cílem vylepšit synchronizaci tak, aby byla více uživatelsky přívětivá, lépe srozumitelná i pro nezaškoleného uživatele.

6 Závěr

Jak již bylo zmíněno v úvodu, tato práce staví na výsledcích projektu řešeného na Provozně ekonomické fakultě, v rámci kterého se mj. implementovala obdobná aplikace pro platformu *Android* a serverový backend, který jsem pomáhal vyvíjet. V rámci bakalářské práce jsem zcela samostatně navrhl a implementoval aplikaci pro platformu *iOS* a udělal potřebné úpravy na serveru.

Dílčím cílem této práce bylo navrhnout uživatelské rozhraní aplikace, které by umožňovalo správu projektů, vrstev a dalších objektů. Tento dílčí cíl jsem naplnil navržením a implementací designu jednotlivých obrazovek a vytvořením vazeb mezi nimi. Design minimalizuje množství zobrazených informací pro malé displeje a zároveň obsahuje flexibilní rozvržení pro využití prostoru na větších zařízeních. Uživatelské rozhraní obsahuje dialogy pro výběr akcí, panely (které lze skrýt), využívá efektivně gesta pro manipulaci s položkami. Design jednotlivých obrazovek a přechody mezi nimi byly vytvořeny v tzv. *storyboardu* v prostředí *Xcode*. Změny uživatelského rozhraní za běhu aplikace byly implementovány v jednotlivých kontrolerech s využitím tříd z frameworku *Core Data*.

Součástí tvorby rozhraní byl i výběr mapové služby zobrazující mapový podklad a zobrazování různých typů geografických objektů. Pro mapový podklad byl zvolen *Google Maps SDK*. Pro manipulaci s mapovými objekty byla vytvořena třída *MapManager*. Aplikace umožňuje přidávat a zobrazovat všechny požadované typy objektů (body, linie, polygony) včetně jejich stylizace.

Velmi důležitým dílčím cílem práce bylo umožnit obousměrnou synchronizaci naměřených dat i vytvořených objektů se serverem. Za tímto účelem jsem vytvořil přehled všech možných synchronizačních operací a konfliktních situací včetně způsobu jejich vyřešení. V rámci implementace jsem vytvořil soustavu tříd nazvanou centrální úložiště zajišťující snadné a jednotné načítání, ukládání a synchronizaci dat. Tyto třídy spolupracují s úložištěm *Core Data*, obsahují funkce pro vytváření požadavků na server a jejich zpracování, konverzi dat mezi úložištěm *Core Data* a formátem *JSON*, načítání a obnovu dat v kontrolerech apod.

Všechny dílčí cíle byly splněny, aplikace byla dokončena a obsahuje veškerou požadovanou funkcionalitu. Proběhlo otestování a aplikace včetně synchronizace funguje bez znatelných problémů. Aplikace je tak komplexním funkčním nástrojem pro práci s prostorovými daty a zajisté najde uplatnění v různorodých podnikových i jiných oblastech a bude užitečným pomocníkem při každodenní práci.

7 Reference

BUTLER, H., DALY, M., DOYLE, A., GILLIES, S., SCHAUB, T., SCHMIDT, CH. *The GeoJSON Format Specification* [online]. [cit. 2015-11-16]. Dostupné z: http://geojson.org/geojson-spec.htmlexamples.

Carthage [online]. [cit. 2015-11-27]. Dostupné z: https://github.com/Carthage/Carthage.

CocoaPods [online]. [cit. 2015-11-27]. Dostupné z: https://cocoapods.org/.

Common Object Request Broker Architecture [online]. [cit. 2016-03-01]. Dostupné z: https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture.

Core Location Framework Reference Apple inc. [online]. [cit. 2016-02-26]. Dostupné z: https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CoreLocation_Framework.

daltoniam/SwiftHTTP · GitHub [online]. [cit. 2015-11-16]. Dostupné z: https://github.com/daltoniam/SwiftHTTP.

FIELDING, R. T. *Representational State Transfer (REST)*, Fielding Dissertation: CHAPTER 5 [online]. [cit. 2016-03-21]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Firebase - Build Extraordinary Apps [online]. [cit. 2016-03-22]. Dostupné z: https://www.firebase.com/.

Frameworks – OpenStreetMap Wiki [online]. [cit. 2015-11-16]. Dostupné z: http://wiki.openstreetmap.org/wiki/Frameworks.

Geography Markup Language [online]. [cit. 2016-02-20]. Dostupné z: http://www.opengeospatial.org/standards/gml.

Google Cloud Messaging [online]. [cit. 2015-11-29]. Dostupné z: https://developers.google.com/cloud-messaging/gcm.

Google Maps SDK for iOS | Google Developers [online]. [cit. 2015-11-16]. Dostupné z: https://developers.google.com/maps/documentation/ios-sdk/.

Google Play Google Inc. [online]. [cit. 2015-11-28]. Dostupné z: https://play.google.com/store.

HALPERIN, E. *iCloud for Developers* [online]. [cit. 2016-03-21]. Dostupné z: https://developer.apple.com/icloud/. Je to API rozhraní ke cloudovému úložišti iCloud na serverech společnosti Apple.

HALPERIN, E. *Introducing Mapbox GL* [online]. [cit. 2015-11-16]. Dostupné z: https://www.mapbox.com/blog/mapbox-gl/.

iOS Human Interface Guidelines [online]. [cit. 2015-11-27]. Dostupné z: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/.

iTunes Apple Inc. [online]. [cit. 2015-12-04]. Dostupné z: https://itunes.apple.com.

JetBrains AppCode: Swift Execution of Your Bright Ideas [online]. [cit. 2015-11-16]. Dostupné z: https://www.jetbrains.com/objc/?fromMenu.

JSON: The Fat-Free Alternative to XML [online]. [cit. 2015-11-16]. Dostupné z: http://www.json.org/xml.html.

Keyhole Markup Language (KML) [online]. [cit. 2016-02-20]. http://www.opengeospatial.org/standards/kml/.

Liang, Chao; Hu, Luokai *SyncCS: A cloud storage based file synchronization approach*, Journal of Software, 9(7), 1679-1686. Retrieved from http://search.proquest.com/docview/1545506746?accountid=28016.

Location and Maps Programming Guide. Displaying Maps [online]. [cit. 2015-11-16]. Dostupné z: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/MapKit/MapKit.html.

McCormack, D. *Data Synchronization* [online]. [cit. 2015-11-29]. Dostupné z: https://www.objc.io/issues/10-syncing-data/data-synchronization/.

Nahavandipoor, V. *iOS 8 Swift Programming Cookbook*, Sebastopol: O'Reilly Media, 2014. 978-1-4919-0867-9.

Neuburg, M. *iOS 8 Programming Fundamentals with Swift*, Sebastopol: O'Reilly Media, 2015. 978-1-491-90890-7.

OSM in MapKit [online]. [cit. 2015-11-16]. Dostupné z: http://wiki.openstreetmap.org/wiki/OSM_in_MapKit.

Pascuala, V., Xhafa, F. *Evaluation of contact synchronization algorithms for the Android platform*, Mathematical and Computer Modelling, Volume 57, Issues 11–12, June 2013, Pages 2895-2903, ISSN 0895-7177, http://dx.doi.org/10.1016/j.mcm.2011.12.039.

Qt for iOS - Building from Source [online]. [cit. 2015-11-27]. Dostupné z: http://doc.qt.io/qt-5/building-from-source-ios.html.

Quaine, N. *SOAP Basics 1 : What is SOAP?* [online]. [cit. 2016-03-21]. Dostupné z: http://www.soapuser.com/basics1.html.

RestKit/RestKit · GitHub [online]. [cit. 2015-11-16]. Dostupné z: https://github.com/RestKit/RestKit.

Swift and Objective-C in the Same Project. iOS Developer Library. [online]. [cit. 2015-11-27]. Dostupné z: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html.

SwiftyJSON/SwiftyJSON · GitHub [online]. [cit. 2015-11-16]. Dostupné z: https://github.com/SwiftyJSON/SwiftyJSON.

Thompson, M. *From NSURLConnection to NSURLSession - objc.io* [online]. [cit. 2015-11-16]. Dostupné z: https://www.objc.io/issues/5-ios7/from-nsurlconnection-to-nsurlsession/.

Tidwell, J. *Designing Interfaces, Second Edition*, Sebastopol: O'Reilly Media, 2011. 978-1-449-37970-4.

Úvod do JSON [online]. [cit. 2015-11-16]. Dostupné z: http://www.json.org/json-cz.html.

Windows Communication Foundation [online]. [cit. 2016-03-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx.

Zavřel, R. *Apple představil programovací jazyk Swift a 4000 nových API, které umožní neuvěřitelné věci, podívejte se jaké!* [online]. [cit. 2015-11-27]. Dostupné z: http://www.letemsvetemapplem.eu/2014/06/02/apple-predstavil-programovaci-jazyk-swift-4000-novych-api-ktere-umozni-neuveritelne-veci-podivejte-se-jake/.

Přílohy

A Elektronické přílohy

V elektronické podobě jsou k této práci přiloženy následující soubory:

- zdrojové soubory aplikace pro vývojové prostředí *Xcode* (nutná instalace závislostí pomocí *CocoaPods*),
- HTML stránka obsahující tabulku s přehledem REST API serveru – seznam všech dostupných operací a testovací data (tabulka je z interních zdrojů).