ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Serializační řešení pro moderní API v jazyce Swift
Student:	Bc. Jakub Vlasák
Vedoucí:	Ing. Dominik Veselý
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Prozkoumejte existující serializační a deserializační techniky pro moderní webové API ve formátu JSON do modelových tříd v jazyce Swift určených k perzistenci. Popište výhody a nevýhody současných řešení. Navrhněte řešení kombinující současné knihovny nebo využijte vámi vytvořené vlastní knihovny, pokud se současná řešení ukážou jako nedostatečná. Případně využijte kombinaci stávajících i vašich knihoven. Po implementaci se zaměřte na výhledy do budoucna a udržitelnost (životnost) výstupů této práce.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 1. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Serializační řešení pro moderní API v jazyce Swift

Bc. Jakub Vlasák

Vedoucí práce: Ing. Dominik Veselý

2. května 2016

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Dominiku Veselému za cenné rady, připomínky a odborné vedení při zpracování této diplomové práce. Dále děkuji Bc. Tomášovi Kohoutovi za rady k realizaci návrhu jednoho z řešení.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 2. května 2016 .....

Odkaz na tuto práci

Vlasák, Jakub. *Serializační řešení pro moderní API v jazyce Swift.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Diplomová práce se zabývá serializačními a deserializačními technikami pro moderní webové API ve formátu JSON do modelových tříd v jazyce Swift určených k perzistenci. Představuje moderní programovací jazyk Swift a JSON formát. Analyzuje zhotovená řešení a popisuje jejich výhody a nevýhody. Následně navrhuje řešení zhotovená na základě předchozí analýzy a představuje možnosti testování daných řešení.

Klíčová slova Analýza, JSON, modelové třídy, serializace, Swift, testování

Abstract

The thesis deals with serialization and deserialization techniques for modern web API in JSON format to model classes in the programming language Swift designated for persistence. It presents the modern programming language Swift and JSON format and analyzes made solutions and describes their advantages and disadvantages. Afterwards proposes solutions made based on previous analyzes and presents options for testing of given solutions.

Keywords Analysis, JSON, model classes, serialization, Swift, testing

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Programovací jazyk Swift je mladým, moderním jazykem a používá se pro vývoj softwaru. Tento software může vyžadovat komunikaci za účelem získání nebo odeslání dat. Při komunikaci se v moderním webovém API pro strukturu dat používá formát JSON. Je potřeba zvážit otázku serializace a deserializace dat v tomto formátu do modelových tříd jazyka Swift určených pro perzistenci. Aktuální řešení tohoto problému v programovacím jazyce Swift je nedostatečné.

Tématem této diplomové práce je vyřešit otázku serializačních a deserializačních technik pro moderní webové API ve formátu JSON do modelových tříd v programovacím jazyce Swift určených k perzistenci. Analyzovat současné přístupy a řešení a zhodnotit jejich výhody a nevýhody. Navrhnout a dostatečně realizovat řešení problému, o kterém pojednává tato diplomová práce.

Kapitola 1

Cíl práce

Následující část se zabývá úvodem do problematiky této diplomové práce. Vysvětluji zde, na co jsem se v rámci diplomové práce zaměřil a co je jejím cílem.

1.1 Uvedení do problematiky

Existuje velké množství různého softwaru. Od komplexních systémů přes jednoduchý software, až po mobilní aplikace. Díky globálnímu rozšíření internetu nejen do podniků a domácností, ale i do mobilních zařízení, se každý může připojit kdykoli a kdekoli. Mobilní telefony, tablety, notebooky a počítače vlastní mnoho lidí a denně je používají ke své práci, koníčkům a zábavě. Právě proto se vyvíjí velké množství softwarů.

Se softwarem jsou spojené informace a data, jenž softwary spravují a zpracovávají. Vzhledem k velkému oběmu dat, vznikla potřeba správy a distribuce. Problém nastává při definici, jak budou data přenášena při komunikaci, jak data uchovávat a zpracovávat, tak aby byla konzistentní a zobrazovala se každému uživateli správně. Dochází tedy k otázce, jak data serializovat při komunikaci a jak je dále zpracovávat a případně ukládat do perzistentních úložišť.

Ke komunikaci se nejčastěji používá formát dat JSON. Tento formát je dostatečně výhodný pro definici struktury dat při komunikaci a je často využíván moderními webovými API.

Před necelýma dvěma lety firma Apple vyvinula nový programovací jazyk Swift [2]. Jeho moderní přístup k programování a pevná základna Apple mu pomohla v rozšíření. Primárně byl Swift určen pro operační systémy firmy Apple. Mezi tyto platformy patří operační systém pro počítače Macintosh OS X a mobilní operační systém iOS. Díky uvedení jazyka jako Open Source se dále rozšiřuje i na další platformy [3].

Při procesu získání dat z JSON a následném ukládání do modelových tříd jazyka Swift, mohou nastat problémy. Swift přináší ideologii bezpečného pro-

gramování, je striktně typový a v určitých případech nedovoluje prázdné hodnoty objektů [4]. To může být u zpracování JSON nevýhodné.

1.2 Stanovení cílů diplomové práce

Cílem diplomové práce je vyřešení problému serializace a deserializace dat z JSON dokumentu do modelových tříd jazyka Swift určených pro perzistenci. Protože JSON má své nedostatky ohledně popisu a definice dat, nastává problém při získávání dat z dokumentu JSON a jejich správné serializaci na cílové modelové třídy. Diplomové práce se zaměřuje na tento problém a analyzuje různé přístupy pro zpracování dat v jazyce Swift.

Následující kapitola Technologie popisuje programovací jazyk Swift a formát JSON. Dále představuje moderní webové API a prozkoumává možnosti řešení perzistentního úložiště.

Protože existují různá řešení, která se daným problémem zabývají, jsou všechny důležité přístupy řádně analyzovány a podrobněji prozkoumány v kapitole Analýza. V úvodu kapitoly Analýza je představeno zpracování JSON dokumentu v jazyce Swift. Následně se analyzují vhodná existující řešení.

Po prozkoumání existujících řešení a možností, které jazyk Swift nabízí, se zaměřím na návrh nejvýhodnějšího možného přístupu a případně realizuji daný návrh v kapitole Návrh a realizace.

Poslední kapitola Testování, se věnuje ověřením kvality řešení. Správného přístupu k testování a vybrání vhodných testů.

Požadovaným výsledek práce je řádná analýza existujících řešení. Následována návrhem a realizací vhodného řešení, které nabídne východisko pro problém serializace JSON dat do modelových tříd moderního jazyka Swift určených pro perzistenci.

Kapitola 2

Technologie

Tato kapitole představuje a popisuje technologie, které souvisí s tématem a se kterými budu dále pracovat. Nejdříve je představen moderní programovací jazyk Swift. Jeho výhody a způsob programování, který zastává. Dále formát dat *Javascript Object Notation* (JSON). Nejenže je zde představen tento formát dat, ale jsou uvedeny i jeho nevýhody a způsob zpracování v jazyce Swift.

Dále v kapitole je představeno *moderní webové API*, tak jak se s ním bude následně pracovat a i důvod proč se používá.

Poslední částí kapitoly jsou *modelové třídy určené k perzistenci*. Zde se zaměřuji na vysvětlení jak data ukládat a jak s nimi pracovat. Rozeberu několik možností s jejich výhodami i nevýhodami. Jelikož je nedílnou součástí této práce řešení perzistentních úložišť, je poslední část kapitoly věnována i problematice uložení dat na disk, souborový systém a pravidla uplatňující se u procesu ukládání dat na operačních systémech OS X a iOS.

2.1 Swift

Swift je moderní programovací jazyk zaměřený na bezpečnost, rychlost a softwarové návrhové vzory, vyvíjen firmou Apple Inc.. Je postaven na tom nejlepším z procedurálního jazyka C a objektového Objective-C. Adoptuje bezpečné programovací vzory a přidává moderní funkce. Zahrnuje nízkoúrovňová primitiva jako jsou typy, řízení toku a operátory. Zároveň obsahuje objektově orientované funkce jako třídy, protokoly a generické typy [4].

Swift byl představen v červnu roku 2014 na Apple konferenci *Apple Worldwide Developers Converence* a verze 1.0 byla vydána v září téhož roku [2]. Jedná se tedy o velmi mladý jazyk. V prosinci 2015 byl zveřejněn zdrojový kód Swiftu a projekt se stal Open Source [5]. Veřejný repozitář se zdrojovými kódy jazyka se nachází na webových stránkách `https://github.com/apple`.

Tento mladý programovací jazyk byl primárně vyvíjen pro platformy firmy Apple mezi něž patří:

iOS operační systém pro mobilní telefony iPhone, zařízení typu tablet iPad a hudební přehrávače iPod

OS X série operačních systémů pro počítače Macintosh

watchOS systém pro náramní hodinky Apple Watch

tvOS systém pro chytré televize AppleTV

Swift není jen čistě programovací jazyk pro tyto platformy, ale může se použít například i v Linuxu. Díky Open Source licenci je možné Swift zkompilovat a spustit na mnoha dalších platformách. [3]

Před vydáním programovacícho jazyka Swift, se programovalo pro OS X a iOS platformy převážně v jazyce Objective-C. Tento jazyk je rozšířením jazyka C o objektově orientované paradigmata se systémem zasílání zpráv inspirovaný jazykem SmallTalk. Jelikož většina projektů vyvíjených před nástupem Swiftu, byla napsána v Objective-C, neopomenula se zpětná kompatibilita Swiftu s tímto jazykem. Tudíž knihovny a frameworky napsané v Objective-C, používané k vývoji aplikací pro platformu OS X a iOS, mohou být jednoduše integrovány a použity ve Swift projektech.

Pro použití knihoven napsaných v Objective-C v kódu jazyka Swift, je potřeba v projektu nadefinovat tzv. *bridging header*, tedy hlavičkový soubor, který zajistí použití Objective-C kódu ve Swiftu. [6]

Swift je navržen tak, aby psaní a udržování korektních programů bylo pro vývojáře co nejjednodušší. K tomuto cíli je zapotřebí psát kód způsobem, který dodržuje určité bezpečnostní zásady, pro které má Swift svou syntaxi.

2.1.1 Bezpečnost

Bezpečné programování je důležitým prvkem na něž se Swift zaměřuje. Je potřeba zabránit nedefinovanému chování a chybám, jenž může způsobit sám vývojář. Swift v tomto případě zastává roli striktního jazyka a dovoluje psát bezpečný kód zásadami jako je nutnost inicializace proměnných před jejich použitím, rozdělením definování proměnných pomocí klíčových slov `var` (proměnné) a `let` (konstanty).

Mezi další funkce bezpečnosti Swiftu patří, že v základu Swiftové objekty nemohou být `nil` (prázdná hodnota). Už při psaní kódu kompilátor nedovolí zkompilovat kód, vyvolá *error* – chybové upozornění, když se používá `nil` objekt. Při nutnosti použití prázdných hodnot programovací jazyk Swift využívá tzv. *optionals*. Optional proměnná může obsahovat `nil`, ale syntaxe Swiftu nutí používat znak `?`, díky kterému programátor kompilátoru oznamuje, že tuto část kódu si ošetří bezpečně sám. [4]

2.1.2 Rychlost

Programovací jazyk Swift byl navržen jako náhrada jazyků z rodiny C (C, C++ a Objective-C) [7]. Tyto jazyky jsou rychlé, ale dnes již nejsou moderní. Swift se snaží být moderním jazykem, jenž představuje nové, ale ověřené přístupy k programování a přitom je stále rychlostně porovnatelný se staršími jazyky. Programátor se už nemusí starat tolik o paměť, protože o to se postará automaticky programovací jazyk. Přesto Swift dokáže být dostatečně rychlý. Nejde ovšem jen o rychlé zpracování problémů, ale i o to, aby výkon byl předvídatelný a konzistentní.

2.1.3 Expresivnost

Čitelnost a pochopení kódu je velmi důležité. Může být jazyk rychlý a silný, ale díky své špatné syntaxi a nedostatečné expresivnosti je pro programátora nečitelný a obtížně se s ním pracuje. Swift se snaží o jednoduchou srozumitelnost a dostatečnou výřečnost. Například pojmenovaní parametrů, přenesené z Objective-C, činí API ve Swiftu lépe čitelnější a pochopitelnější a napomáhá k lepší údržbě kódu. Odvozené typy pomáhají k menší chybovosti, moduly eliminují hlavičky z C a nabízejí jmenné prostory.

Další funkce napomáhající nejen k expresivnosti jazyka jsou [7]:

- Closures sjednoceny s ukazately na funkce
- Tuples (uspořádané n-tice) a možnost návratu více hodnot
- Generika
- Rychlá a stručná iterace, přes rozsah nebo kolekce
- Struktury podporující metody, rozšíření a protokoly
- Funkcionální programovací vzory, jako je *map* a *filter*
- Výkonné ošetřování chyb pomocí `try / catch / throw`

2.2 JavaScript Object Notation

JavaScript Object Notation (zkráceně JSON) je formát dat, který je jednoduchý pro čtení i zápis člověkem a jednoduše strojově generovaný a analyzovatelný. Je založený na podmnožině programovacího jazyka JavaScript a standardu *ECMA-262 3rd Edition*. [8]

Tento formát se v dnešní době hojně používá vedle XML, pro uchovávání dat, přeposílání a serializaci. Je nezávislý na jazyce ve kterém je zpracováván. Jedná se o textový formát, což znamená, že je snadno člověkem čitelný i zapisovatelný. Data formátuje v hierarchické podobě, obsahuje hodnoty v hodnotách, stejně jako XML.

Oproti XML je kratší, rychlejší, srozumitelnější pro čtení a dovoluje uchovávat objekty v polích. XML nadruhou stranu obsahuje tagy, možnost definování hodnoty nebo přidání dodatečných informací.

JSON umí pracovat s objekty, řetězci, čísly v desetinné i celé podobě, boolean hodnotami a prázdnou `null` hodnotou [8]. Je založený na dvou strukturách:

- Kolekce párů jméno a hodnota. V mnoha jazycích označováno jako objekt, slovník, hash tabulka nebo například asociativní pole.
- Seřazený seznam hodnot.

Tyto dvě datové struktury jsou hojně podporovány v podstatě většinou moderních programovacích a skriptovacích jazyků a tak není problém zpracovávat JSON mezi dvěma zcela odlišnými aplikacemi, které mohou být napsané v různých jazycích a založené na různých platformách.

Jak je zmíněno výše, JSON oproti XML neobsahuje tagy, což je nevýhoda v případě, že potřebujeme definovat nebo popsat hodnoty. V XML se díky neomezenému množství tagů, mohou popsat a definovat jednotlivé hodnoty. Dá se specifikovat o jaký typ se jedná, co reprezentuje, jakých hodnot může nabývat a podobně. Ovšem JSON tuto možnost nemá. K hodnotě se přiřadí pouze klíč typu řetězec, což je název hodnoty. Jestli v hodnotě je řetězec, desetinné nebo celé číslo se pozná jedině přečtením dané hodnoty. To v případě komunikace klienta se serverem může vyvolávat časté problémy. Pokud server pošle klientovi data v JSON, musí klient znát strukturu těchto dat a nelze jen odhadovat. To by vedlo k chybám v procesu.

Častým problémem jsou různé typy hodnot. Například server posílá klientovi objekt v JSON, který popisuje knihu. Obsahuje vlastnosti: název, datum vydání, cena a krátký popis.

```
{
  "book": {
    "title": "Example book",
    "dateOfPublication": "12.3.2009",
    "price": 12.3,
    "shortDesc": "..."
  }
}
```

Z příkladu lze vyčíst, že titul, datum vydání, krátký popis jsou řetězce a cena je desetinné číslo. Takto lze definovat v aplikaci modelovou třídu a stačí JSON dokument přečíst a serializovat do objektu. Problém nastane, když cena nebude uvedena. Bez specifikace lze jen odhadovat co může vlastnost obsahovat. I malá změna dokáže způsobit nedefinovaný stav nebo i pád aplikace. Pro vyhnutí se tomuto problému jsou na místě následující možnosti.

Zaprvé pevně specifikovat strukturu a pravidla pro daný projekt. Zhotovit dokumentaci a dodržovat nadefinovanou specifikaci jak v klientsné části, tak i u serverového API. Zde nastává problém pokud dojde k úpravě vlastnosti záznamu, specifikace projektu nebo jiné změně. Jediná i malá změna znamená přepis nejen dokumentace, ale i veškerého závislého softwaru. Pokud klient obdrží z API JSON, jenž nesplňuje požadavky specifikované v dokumentaci, může dojít k nesprávnému zobrazení dat, popřípadě k nefunkčnosti části i celé aplikace. Kontrolovat všechny závislosti a doplnit případné změny je časově náročné. Proto je na místě realizovat dostatečné testy jednotlivých částí a kontrolovat nejen specifikaci ale i webové API.

Druhou možností je dostatečně implementovat proces zpracování JSON formátu. Samozřejmě zde, jakožto u všech uvedených možností, je nutnost specifikovat data a připravit dostatečnou dokumentaci. Dostatečným implementováním zpracování JSON formátu, se myslí dostatečné ošetření všech hodnoty JSON dokumentu. Připravení se na možnost získání jiného typu hodnoty, než bylo očekáváno.

Příkladem může být hodnota ceny z předchozího příkladu objektu knihy. Při syntaktické analýze tohoto objektu je očekávaná hodnota desetinné číslo. V případě chyby při čtení desetinného čísla, je uložena do cílové, datové struktury prázdná hodnota (`nil`). Popřípadě lze vyzkoušet jestli se nejedná o jiný typ a případně provést pokus o přetypování hodnoty. V první možnosti by při stejné situaci došlo k chybě při čtení desetinného čísla a program by spadl nebo by se záznam o knize vůbec nezpracoval.

Tento přístup znamená ošetřování a testování okrajových podmínek. Pro jednotlové vlastnosti zjistit typ hodnoty a provést případný převod na cílový typ, uložení zdrojové a nebo prázdné hodnoty. Podobný kód je špatně čitelný a obtížně udržovatelný.

Problém nastává hlavně při práci s objektem dále v aplikaci v případě, že obsahuje proměnné hodnoty. Je zde potřeba ošetřit každou vlastnost, jestli neobsahuje prázdnou hodnotu. Proto je výhodné definovat vlastnosti jako konstantní (let), zvláště v případě, že danou vlastnost není potřeba měnit. Swift zaručuje, že konstantní vlastnost neobsahuje prázdnou hodnotu. V případě použití proměnných vlastností objektu, je potřeba ošetřit možnost prázdné hodnoty.

Třetí možností je definovat čtení tak, že pokud se bude lišit některá z hodnot, nepřečte se celý záznam. Proces zpracování JSON objektu tedy probíhá tak, že se postupně čtou jednotlivé hodnoty a pokud hodnota je povinná a ve zdrojovém dokumentu neexistuje, případně nesplňuje požadavky, je proces zastaven a daný objekt se nezpracuje.

Tímto přístupem lze zajistit korektnost celého objektu. Nedojde tedy k případu, že objekt obsahuje jen část vlastností, který následně nelze řádně zobrazit uživateli nebo zpracovat. Ovšem výhodou je, že objekt není ztracen jako v případě třetí zde uvedené možnoti.

Ideální variantou se jeví zkombinovat předchozí dvě možnosti. Tedy vytvo-

řit objekt a přidávat postupně hodnoty vlastností tak, jak jsou čtena z JSON dokumentu. Je potřeba určit typy vlastností a jestli jsou jednotlivé vlastnosti objektu stěžejní nebo ne. Pokud ano, definovat je jako konstanty a pokud ne, definovat je jako proměnné. Případně definovat u konstatních hodnot jejich základní hodnotu v případě, že nelze přečíst hodnota vlastnosti ze zdrojových dat. K chybě při čtení dojde pouze v případě, že povinná vlastnost nemá definovanou základní hodnotu a nelze převést z JSON dokumentu.

Jaký přístup zvolit záleží na návrhu projektu a požadavcích na výsledný produkt. Ovšem nejlépe vychází pravděpodobně poslední možnost. Kombinace prázdných hodnot a vyvolání chyby při čtení objektu, na základě stanovených priorit.

Praktický příklad se striktním zpracováním dat z JSON dokumentu je představen v části 3.1.

2.2.1 JSON Schema

Protože bylo potřeba rozhodnout jak budou data strukturovaná v XML dokumentu, bylo specifikováno *XML Schema Definition* (XSD) [9]. Účelem XSD je definovat jaké elementy a atributy se mohou objevit v dokumentu, výchozí a pevné hodnoty atributů a záznamů, atd. Podobnou cestou se vytvořila definice *JSON Schema*. Popisuje existující datové formáty, vytváří čitelnou dokumentaci jak pro člověka, tak pro stroj. Nabízí kompletní strukturální validaci užitečnou pro automatizované testy nebo klientsky zadaná data. [10]

Je to jedna z dalších variant jak řešit popis dat v JSON. Výhodou je, že je pevně specifikovaný formát, definují se možné hodnoty a objekty, které se v daném JSON dokumentu nacházejí. Nevýhodou může být v některých případech složitost. Někdy existuje jen krátký a jednoduchý JSON dokument a vytvářet pro tento případ schéma je zbytečné.

Tento přístup není často využíván. Jak bude vidět později v kapitole Analýza, většina řešení jsou postavena na JSON formátu bez použití JSON Schema. Je to pravděpodobně z toho důvodu, že některá již hotová API JSON Schema nepodporují a nebo není tento přístup dostatečně známý. Jedná se o teoretický model a v praxi skoro nepoužívaný.

2.2.2 Serializace a deserializace JSON

Proces serializace a deserializace JSON ve Swiftu se v mnoha případech provádí pomocí knihoven třetích stran. Apple vývojářům nabízí *Foundation framework*, jenž tento i mnoho dalších problémů řeší.

Foundation framework definuje základní vrstvu Objective-C tříd. Obsahuje užitečné primitivní objektové třídy a několik paradigmat, které definují funkcionalitu v Objective-C neobsaženou [11]. Základní nadtřídou všech Foundation tříd, ze které ostatní třídy dědí a vycházejí, je NSObject. Mezi často používané třídy patří například NSString, NSNumber, NSDate a další.

Většina tříd zastupující primitivní typy jsou neměnitelné. To znamená, že jakmile se inicializuje objekt s určitou hodnotou, tato hodnota zůstane objektu a nelze měnit. Říká se jim taktéž statické objekty. Zatímco proměnlivé, neboli dynamické objekty, často označované jako mutable, mohou za dobu své existence mít různou hodnotu. Příkladem může být zmíněná třída na práci s řetězci NSString, která obsahuje metody pro porovnávání, hledání či úpravu řetězců, ale hodnota v objektu nelze měnit. NSMutableString je podtřída třídy NSString a dovoluje měnit hodnotu řetězce dané instance. To může být k užitku, pokud je hodnota potřeba měnit v průběhu existence instance. Například v případě, kdy je nutné změnit část řetězce, nějaký podřetězec. Zatímco u NSString je potřeba vytvořit novou instanci a jí následně přiřadit upravenou hodnotou z předchozí instance.

Foundation framework upravený pro Swift je dostupný ve veřejných repozitářích firmy Apple. Tento projekt obsahuje implementaci Foundation API pro platformy, kde není dostupné Objective-C. Výhodou použití této verze Foundation frameworku je, že na platformách nepodporující Objective-C stačí jen jazyk Swift. Na OS X, iOS a dalších Apple platformách by se ale měl používat Foundation, jenž je obsažen v operačním systému, což je verze s Objective-C a lze použít v projektu napsaném ve Swiftu.

Mezi třídy, které jsou zajímavé pro serializaci a deserializaci JSON patří:

NSData objektově orientované úložiště pro pole bytů

NSJSONSerialization serializace JSON do Foundation objektů a zpět

NSError objekt zapouzdřující informace o chybovém stavu

NSNull reprezentace prázdné hodnoty

Třída NSData se typicky používá jako úložiště dat a je užitečná pro serializaci dat mezi aplikacemi. Datové objekty jsou uloženy v NSData instanci a mohou být kopírovány nebo přeposílány mezi aplikacemi. Proto je tato třída hojně využívána při serializaci JSON objektů.

NSJSONSerialization je třída, jenž zastupuje serializaci a deserializaci JSON formátu do objektů z Foundation frameworku.

Serializace Foundation objektů do JSON pomocí NSJSONSerialization třídy je možná za následujících podmínek [12]:

- objekty nejvyšší úrovně jsou NSArray nebo NSDictionary
- všechny objekty jsou instance tříd NSString, NSNumber, NSArray, NSDictionary nebo NSNull
- klíče slovníků jsou NSString
- čísla nejsou NaN nebo nekonečno

Tyto a další podmínky lze ověřit pomocí metody `isValidJSONObject` nebo použitím metody pro konverzi do JSON. Pokud data nebude možné převést, funkce vrátí `nil` a NSError objekt popisující chybu.

V jazyce Swift se místo NSArray, NSDictionary nebo NSString používá Array, Dictionary a String.

Funkce pro serializaci dat do JSON:

```
class func JSONObjectWithData(_ data: NSData,
    options opt: NSJSONReadingOptions) throws -> AnyObject
```

Funkce pro deserializace dat z JSON:

```
class func dataWithJSONObject(_ obj: AnyObject,
    options opt: NSJSONWritingOptions) throws -> NSData
```

Jak je vidět z definice funkcí objekt v němž jsou serializované objekty je NSData. Funkce provádí serializaci NSData do formátu JSON a naopak. Pomocí parametru `NSJSONReadingOptions` resp. `NSJSONWritingOptions` je specifikováno nastavení pro čtení resp. zápis. Bližší informace o nastavení funkce lze získat z dokumentace [12].

Při práci s JSON ve Swiftu se pracuje se slovníkovou formou struktury dat, jenž obsahuje data ve formátu klíč-hodnota. Z webového API se data získají v NSData a prvním krokem je deserializace dat z tohoto objektu do slovníku pomocí třídu NSJSONSerialization.

2.3 Moderní webové API

Pod pojmem API, neboli *Application Programming Interface* se rozumí rozhraní umožňující interakci například s aplikací, hardwarem, serverem nebo databází, použitím množiny funkcí. Cílem API je definování funkcí, které jsou nezávislé na konkrétní implementaci. Dovoluje programátorovi psát software, jenž odstíní přímou práci s cílenou platformou. API lze použít na mnoha místech, ovšem z důvodu zaměření práce na moderní webové API se zde zabývám pouze API webovým.

Webové API se používá pro výměnu informací mezi serverem a klientem. Do skupiny moderních webových API se dá zařadit *REST* nebo starší *SOAP*. REST je vedoucí programovací model pro webové API [13]. SOAP, neboli *Simple Object Access Protocol*, je starší než REST. Slouží pro posílání zpráv v XML se standardním formátem pro popis webových služeb *Web Service Description Language* (WSDL) [14]. V dnešní době se používá ve specifických případech a častěji se vývojáři přiklání k REST řešení.

Protože zaměření této diplomové práce je na serializační řešení pro moderní webové API v jazyce Swift, nebudu zde popisovat dopodrobna všechna webová API. Zaměřím se na to nejpoužívanější a tomuto tématu nejvíce vhodné REST API.

2.3.1 REST

Zakladatelem Representational State Transfer (REST) je Roy Fielding, který navrhl a popsal REST ve své disertační práci *Architectural Styles and the Design of Network-based Software Architectures* v roce 2000. Webovou službu REST dnes používá mnoho moderních Web 2.0 poskytovatelů služeb jako je Google, Yahoo nebo Facebook. [15]

REST používá zdroje a definuje k nim přístup. Získat informace z toho zdroje lze použitím HTTP metody GET. Další manipulace se zdrojem jako je modifikace objektů lze provést pomocí POST, PUT a DELETE metod. Zdrojem může být cokoli: reálný objekt nebo abstraktní věc jako je adresa, jméno apod. Zdroj v REST API koresponduje k jedné nebo více entitám v data modelu. Má jednoznačný identifikátor URI a reprezentaci. Ta může být v XML, JSON, v textové podobě aj.

Správný návrh REST aplikace se označuje jako RESTful. Základní principy REST aplikací definují následující: Existuje klient a server, kteří spolu komunikují. REST aplikace je bezstavová, používá cache, je navržena jako „vrstevný systém“ a má jednotné rozhraní [13].

Jelikož náplní této práce není návrh či implementace REST API, plně postačí jen toto představení REST jako služby.

2.3.2 Proč používat webové API v aplikacích

Swift se v nynější době převážně používá pro vývoj mobilních aplikací s operačním systémem iOS a tyto mobilní aplikace často komunikují s centrálním serverem k získání nebo sdílení informací. Může se jednat například o aplikaci určenou pro posílání a přijímání zpráv mezi uživateli, získání informací od uživatele. Aplikace může být zaměřena jako informativní. Tedy získává obsah ze serveru. Mezi podobné aplikace můžeme zařadit, čtečku mediálního obsahu, elektronické noviny, prohlížeč fotek a obrázků a mnoho dalších. Mobilních aplikací využívající server pro zpracování nebo uchovávání informací je nepřeberné množství a stále se vyvíjejí nové.

Není potřeba se zaměřit pouze na mobilní platformu iOS. Aplikace pro počítače se systémem OS X mohou pracovat na podobném ne-li stejném principu a jsou taktéž vyvíjeny pomocí programovacího jazyka Swift. Zároveň je možno vyvíjet aplikaci ve Swiftu i pro televizi s operačním systémem AppleTV nebo chytré hodinky AppleWatch. Ve všech zmíněných případech je potřeba komunikace mezi určitým centrálním prvkem, jako je server a klientem. Proto je webové API v mnoha aplikacích potřeba.

Využití serveru pro zpracování, nebo uchování informací má výhody nejen pro centralizované úložiště dat, ale i co se týče výpočetního výkonu. Mobilní aplikace jsou určeny především pro zobrazení informací a zpracování vstupu od uživatele. Informace samotné jsou zpracovávány serverem. Proč je toto výhodné se dá jednoduše popsat na příkladu.

Příkladem je mobilní aplikace, nabízející náhled do databáze o velkém množství záznamů. Tyto záznamy obsahují mnoho textových informací, ale i mediální obsah jako jsou obrázky. Při vyhledávání v záznamech je potřeba zadat vstupní parametry pro provedení výpočtu, jenž určí parametry pro vyhledání potřebných záznamů. V případě, že aplikace má tyto informace uchované v lokálním úložišti, tedy v telefonu, není potřeba přístupu k internetu. Ovšem data mohou být v řádech sta megabyte. Nejedná se o výhodné řešení, protože databáze zabírá příliš místa v mobilním zařízení. Zobrazení všech záznamů najednou je nemyslitelné, protože samotné načtení dat a zobrazení trvá příliš dlouho. Zároveň pro uživatele nemá žádný smysl zobrazovat všechna data najednou pokud, se jedná o sta či tisíce dat. Co je pro uživatele důležité, že v tomto množství může nalézt požadovanou informaci, popřípadě může záznamy filtrovat.

U mobilních zařízení je důležité nezabírat mnoho výpočetního výkonu a místa na disku. Nejenže výpočet může zabírat hodně času a můžeme tak omezit ostatní procesy, ale vytěžováním procesoru se znatelně rychleji vybíjí baterie zařízení. Provádění výpočetně složitých úkonů, jako například vyhledávání ve velkém množství dat, nebo provádění složitých výpočtů, je nevýhodné.

Pokud se při řešení použije server s centralizovanou databází a aplikace (klient) komunikuje s tímto serverem pomocí webového API, je postaráno o výkon a místo. Aplikace sice musí mít přístup k internetu, aby mohla komunikovat se serverem, ovšem samotná komunikace se může omezit na aktuální potřebu. Neboli uživatel v určitém momentě potřebuje vyhledat záznamy, stačí jen odeslat požadavek serveru a stáhnout potřebné informace. Server může odpovědi cashovat nebo použít další funkce k urychlení odpovědi klientovi. To je ovšem nad rámec tohoto textu.

Je obvyklé že uživatel s mobilním zařízením má přístup k internetu. Ať se jedná o lokální připojení přes Wi-Fi nebo mobilní připojení LTE, 3G a další. Mobilní internet je dostatečně rychlý na běžnou komunikaci se serverem.

Výsledkem řešení použití serveru tedy je, že aplikace nezabírá příliš místa v zařízení a nezabírá příliš výpočetního výkonu. Zobrazuje vždy aktuální data pro jakéhokoli uživatele a nemusí se starat o záznamy samotné. V případě nutnosti připojení k internetu se mohou některé výsledky (poslední, nejčastější apod.) hledání ukládat, tedy cashovat. Nejenom informace ale i zpracování informací je jednotné, díky použití centralizované správy dat.

V mnoha případech mobilní, popřípadně i desktopová aplikace není jediná a používá se i aplikace webová. I z toho důvodu je na místě použít server, aby veškeré informace byly konzistentní a dostupné na všech platformách.

Výše jsou na příkladu uvedeny důvody, proč je výhodné použít server při tvorbě mobilní aplikace. K nim lze přidat i další důvody proč se používá centralizované úložiště a výpočetní prostředky. Záznamy v databázi často nejsou neměnné. Je potřeba je aktualizovat či jinak měnit a často je i požadavek, že každý uživatel má přístup ke stejným datům jako ostatní. To lze řešit i synchronizací databáze, ale uživatele je tímto přístupem omezen, protože

aktualizace celé databáze trvá mnohem déle, než načtení určitých záznamů. Proto je výhodné použití serveru a moderního webového API.

2.3.3 Typy dat

Informace jenž jsou sdíleny mezi serverem a aplikací, ať už na mobilních nebo jiných zařízeních, mohou být jen jednoduché řetězce, nebo serializované mediální objekty jako jsou obrázky, zvukové stopy apod. Může se ovšem jednat i o složitější struktury dat, představující objekty s určitými vlastnostmi a i kolekce objektů. Je proto důležité přemýšlet nad tím, jak tyto informace uchovávat při přenosu tak, aby druhá strana mohla jednoduše informace zpracovat a použít.

Nejpoužívanější formáty dat pro komunikaci, jsou JSON a XML. Při vývoji pro Apple platformy se nejčastěji pracuje s formátem JSON, kterému je věnovaná část 2.2.

Předmětem informací pro komunikaci mezi serverem a klientskou aplikací je často obsah. Tedy informace, jenž aplikace nabízí. Zde lze zařadit: galerii obrázků, články, různé texty, zvukové záznamy, videa atd. Aplikace zaměřené na komunikaci mohou sdílet informace jako jsou zprávy mezi uživateli, status uživatele, obrázky nebo další údaje. Dále se může jednat i o uživatelské nastavení aplikace, jenž se zálohuje na server, nebo aktuální informace od uživatele a mnoho dalších.

Často se používá i registrace uživatelů a jejich přihlášení do aplikace. To i autentifikace může být také zpracováno na serveru.

2.4 Modelové třídy určené k perzistenci

Modelové třídy se užívají k definici objektů používaných napříč aplikací. Definují vlastnosti i strukturu dat. Modelové třídy určené k perzistenci zajišťují uchování dat i v případě uzavření aplikace. Data se opět načtou po spuštění a lze k nim přistupovat, mazat je a nebo upravovat.

Nejintuitivnějším řešením pro persistentní uchování dat je ukládat data do souboru zápisem na disk. Tedy použít souborový systém vybrané platformy. Další možností je databáze *SQLite* a nebo použití *Property list*. Případně využít frameworky, které se touto problematikou zabývají.

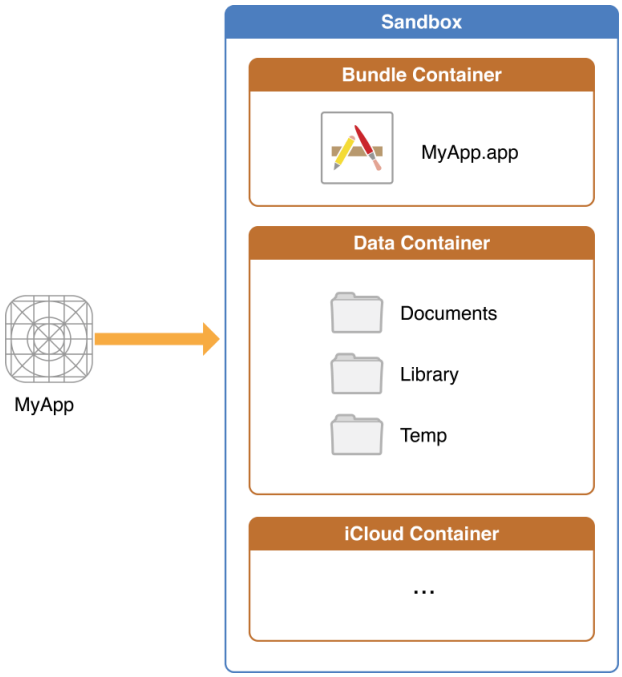
V části 2.2.2 byla uveden Foundation framework. Tento framework, mimo jiné, nabízí i třídu *NSUserDefaults* zajišťující perzistentní úložiště v rámci aplikace. Pokud je ovšem požadavek spravovat větší množství dat, lze použít framework určený ke správě modelové vrstvy v aplikaci – *Core Data*. Z vybraných možností se jedná o nejpropracovanější řešení. Níže jsou podrobněji popsány jednotlivé varianty a jejich vhodné použití.

2.4.1 Ukládání na disk

Možnost ukládat data manuálně na disk je výhodné ve chvíli, kdy se řeší specifický problém, při němž je potřeba mít data a jejich správu plně pod kontrolou. Nebo dodatečná funkčnost, jenž žádné již existující řešení nenabízí. Tento přístup nabízí možnost specifikovat samotný proces uložení dat a jejich strukturu. Ovšem tento přístup přináší i zodpovědnost za uložená data a nutnost přesně specifikovat přístupy, strukturu a manipulaci s daty. Také je nutné řádně otestovat navržený systém, protože jediná chyba může způsobit ztrátu důležitých informací. To vše si je potřeba zajistit a to bere čas i prostředky.

Souborový systém na platformách OS X a iOS je založen na Unixovém souborovém systému. Jedná se o stejný souborový systém pro obě platformy a adresářový systém je pro tyto platformy podobný. Liší se ve způsobu, jak daná platforma organizuje uživatelská data a data aplikací. [1]

V případě mobilního systému iOS nemá uživatel přímý přístup k souborovému systému jako celku. Limitovaný přístup mají i aplikace. Je jim přidělen převážně přístup do složek v rámci jejich tzv. *sandboxu*. Instalátor v průběhu instalace aplikace vytvoří několik kontejnerů (viz obr. 2.1). Každý z nich má specifickou roli. Z bezpečnostních důvodů nemá aplikace přístup mimo své kontejnery.



Obrázek 2.1: Sandbox iOS aplikace [1]

Pro ukládání dat jsou určeny adresáře v datovém kontejneru. Adresáře

rozlišují o jaká data se jedná a tím i kde jsou uložena. Doporučení od Apple je ukládat data do adresářů takto [1]:

- `Documents/` – slouží pro data vytvořená uživatelem. Uživatel může k těmto datům přistupovat.
- `Library/Application support/` – pro podpůrná data aplikace. Skrytá pro uživatele.
- `tmp/` – dočasné soubory.
- `Library/Caches/` – data pro účely vyrovnávací paměti, která je potřeba zanechat déle než dočasná, ale ne déle než podpůrná data. Často se jedná o data potřebná pro zvýšení výkonu aplikace.

Souborový systém OS X je navržen pro počítače Macintosh a jak uživatel, tak software má přístup do celého souborového systému. Nejsou zde omezení jako v systému iOS. Adresářová struktura je rodělena na doménové části. Jednotlivé domény mají své určení a jako vývojáři i uživatelé, by jej měli dodržovat pro správný chod systému. [1]

Uživatelská doména soubory patřící přihlášenému uživateli. Jsou zde soubory všech uživatelů.

Lokální doména zahrnuje aplikace a soubory aplikací lokálně uloženy v systému a sdíleny mezi všemi uživateli v rámci systému.

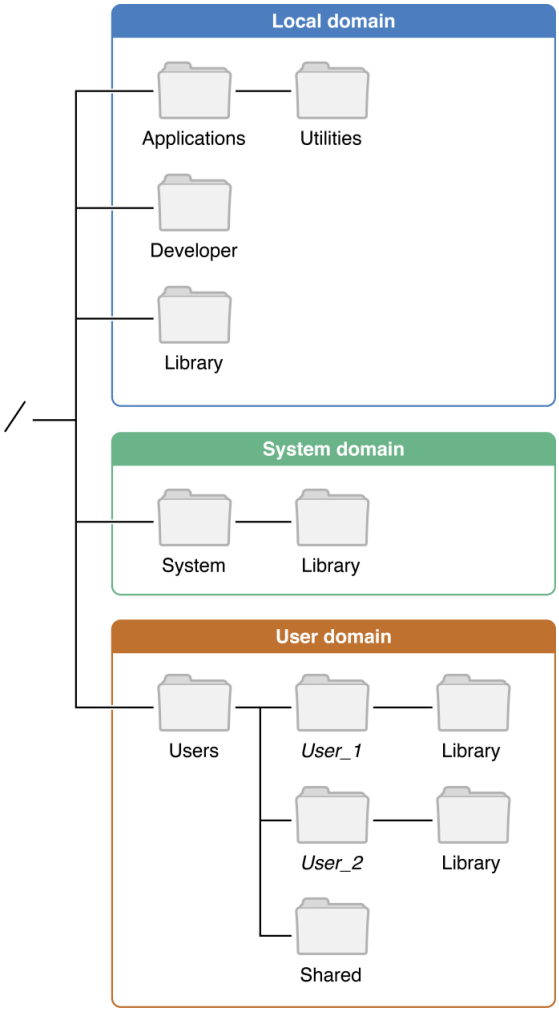
Síťová doména sem patří soubory a aplikace, jenž jsou sdíleny na lokální síti mezi všemi uživateli. Tyto soubory jsou často umístěny na serveru a jsou spravovány administrátory sítě.

Systémová doména obsahuje systémový software a soubory potřebné pro běh systému.

Obrázek 2.2 zobrazuje souborový systém a rozdělení adresářů do jednotlivých domén. Není zde síťová doména. Ta se často podobá lokální.

Aplikace jsou nainstalovány do lokální domény ve složce `/Applications`. Ve stejné doméně je doporučeno ukládat podpůrná data aplikace do adresáře `Library`. Tento adresář se nachází na třech různých místech a každý z nich je určen pro jiné účely:

- `Library` – lokalizován v uživatelské doméně, v adresáři uživatele. Slouží pro ukládání dat specifických pro uživatele.
- `/Library` – v lokální doméně. Určen pro ukládání podpůrných dat aplikace a sdílí tyto data mezi uživateli v rámci systému.
- `/System/Library` – adresář je vyhrazen pro systémové použití.



Obrázek 2.2: Souborový systém OS X [1]

Pro práci se soubory nabízí Foundation framework třídu *NSFileManager*. Tato třída je určena pro manipulaci se soubory a jejich obsahem. Používá se pro vyhledání, vytváření, kopírování a přesouvání souborů a složek. Také se dá použít k získání informací o souboru a nebo ke změně atributů. [16]

2.4.2 NSUserDefaults

Foundation framework mimo jiné nabízí i třídu *NSUserDefaults* poskytující programové rozhraní pro interakci se základním nastavením systému. Základní nastavení systému umožňuje aplikaci přizpůsobit chování tak, aby vyhovovalo potřebám uživatele [17]. Zahrnuje i informace o nastavení aplikace zadané uživatelem. Může sem patřit například barevný vzhled aplikace, vybrané jed-

notky pro zobrazení vzdálenosti, další měrné jednotky, jak často se má ukládat dokument editovaný v aplikaci, apod.

Z NSUserDefaults se načítají zpravidla informace při spuštění aplikace. Ukládají se zde převážně uživatelské volby a nastavení. Data jsou uložena v databázi a třída tyto informace uchovává v mezipaměti, aby se nemuselo při každém přístupu s databází pracovat. Třída používá pro perzistenci dat *Property list* jako databázi. Jedná se o soubor, jenž reprezentuje strukturovaná data. Více o Property list v sekci 2.4.3.

Pomocí metody `synchronize` se synchronizují data mezi mezipamětí a databází. Tato metoda se automaticky spouští v pravidelných intervalech. Je nutné zavolat pouze v případě, kdy je potřeba synchronizovat data v daném momentě. Například při uzavření aplikace.

Data která lze ukládat do NSUserDefaults databáze se shodují s JSON daty, jenž získáme z NSJSONSerialization. Veškerá data jsou neměnná (imutable), i když je uložíme jako dynamická (mutable). Pokud je potřeba data editovat je nutné je v rámci NSUserDefaults přepsat.

Databáze je vytvořena automaticky, při prvním spuštění aplikace. Je ihned k dispozici a není nutné se starat o její vytvoření a inicializaci. To přináší i omezení, které se vztahuje k velikosti ukládaných dat. Data se načítají vždy po startu aplikace a to i když nejsou potřeba. Tím nastává i omezení na správu dat z NSUserDefaults na hlavním vlákně a nelze spravovat data z jiného vlákna. Je sice ošetřena bezpečnost vláken, ale odpadá možnost načítání nebo ukládání dat na pozadí.

Přístup k datům se provádí pomocí slovníku ve formátu klíč-hodnota. Veškeré hodnoty jsou uloženy pod klíčem, jenž je reprezentován jako řetězec. Pod tímto klíčem lze data uložit i číst. Klíč nemá žádná omezení ani konvence pro pojmenování. Jedná se o podobný popis dat jako je formát JSON. Pod klíčem je hodnota a tato hodnota je vždy *AnyObject*, tedy není znám její typ. Jednotlivé hodnoty lze získávat i nastavovat pomocí metod specifických pro daný typ. Například metoda `floatForKey` pro získání desetinného čísla. Při čtení hodnoty je nutné specifikovat její typ. V případě specifikace špatného typu, se vrátí prázdná hodnota.

NSUserDefaults třída je určena pro malé množství dat. Sice nejsou nikde specifikované limity pro velikost ukládaných dat, ale obecně se zde ukládají informace maximální velikosti v řádek jednotek megabyte, spíše kilobyte. Také díky nemožnosti vyhledávání a ukládání složitějších struktur, či multimédií, se nedostává takových možností jako u databází.

Používání třídy NSUserDefaults pro ukládání dat v aplikaci je dobrá volba, pokud je nutné ukládat data spojená s nastavením aplikace a nebo nastavení a hodnoty specifikované uživatelem. Přitom se neočekává, že data budou často editovaná. Nevýhodou tohoto řešení totiž je, že je potřeba znát hodnoty, které se pod daným klíčem ukládají. Zároveň je nutné znát klíč, pod kterým byla hodnota uchována. Pro ukládání modelů a složitějších záznamů se tento způsob nehodí.

2.4.3 Property list

Property list, známý také pod názvem *plist*, je jedna z dalších možností jak ukládat data podobně jako při použití NSUserDefautls. Výhodou tohoto řešení je, že lze specifikovat vyběr kdy data načíst a i na jakém vlákně se proces spustí. Odpadá limitace předchozího řešení, ale přichází i zodpovědnost za správné ošetření práce s vlákny. NSUserDefaults, jak bylo již zmíněno, používá Property list jako svoji databázi. Používá se i na dalších místech pro ukládání metadat, atributů, stastavení a dalších informací používaných aplikacemi napříč systémem [18].

V podstatě se jedná o XML soubor, který specifikuje strukturu klíč-hodnota. Má omezení v podobě možných hodnot pro zápis. Patří sem primitiva jako řetězce nebo čísla a kolekce slovník a pole. Tímto je stanoveno omezení na složitost zapisovaných dat. Pro uchování složitých modelů a nebo i jednoduchých s vazbami, není toto řešení vhodnou volbou.

Na druhou stranu pokud je potřebné zapisovat jednoduchá data, která lze s lehkostí serializovat do primitiv a kolekcí, je toto řešení přijatelné. Omezení jsou jinak podobná NSUserDefaults, až na možnost výběru kdy data načítat a na jakém vlákně.

2.4.4 SQLite

Další možností je použít relační databázový systém SQLite. Jedná se o systém publikovaný pod volnou licencí *public domain*, která dovoluje volnou dostupnost a použití zdarma k jakýmkoli účelům. SQLite je malá knihovna, která pracuje na bezserverovém principu, tedy nepotřebuje separátní serverový proces pro běh. Pracuje jako SQL databázový stroj, čte a zapisuje data přímo do diskových souborů. Kompletní SQL databáze s tabulkami, triggery a pohledy je uložena v jediném souboru. Databázový formát je multiplatformní a dá se tedy kopírovat mezi 32-bit a 64-bit systémy nebo mezi big-endian a little-endian architekturamy. Jedná se o kompaktní databázi. [19]

Nejedná se o klasickou klient-server databázi, tak jak je často známo, ale naopak je databáze uložena v souboru a předpokládá se, že k datům přistupuje software, který běží na stejném stroji. Výhodou je, že SQLite není potřeba konfigurovat. Jediné co je potřeba specifikovat, soubor se kterým se má pracovat. Transakce jsou plně ACID kompatibilní a podporují bezpečný přístup pro více vláken.

SQLite se často používá nejen na platformách iOS a OS X, ale i na mobilních systémech Android.

Výhodou použití databáze SQLite je jeho malá velikost, jednoduchost a to, že databáze je uložena v souboru vedle aplikace. Není potřeba konfigurovat databázi a ani server pro běh databáze. Díky tomu, že je uložena v jediném souboru je její zálohování a případně i přesun velice jednoduchý.

Ačkoli SQLite podporuje nejpoužívanější dotazovací jazyk definovaný ve standardu *SQL92*, nepodporuje ho kompletně. Některé funkce nejsou implementované. Není například implementován RIGHT OUTER JOIN a FULL OUTHER JOIN, ale pouze LEFT OUTER JOIN. Také chybí funkce ALTER TABLE jako je DROP COLUMN, ALTER COLUMN a ADD CONSTRAINT. Pohledy VIEW jsou v SQLite pouze ke čtení. Nelze je použít pro mazání, vkládání ani aktualizování hodnot. [20]

Nevýhodou je, že při zápisu většího množství dat má databáze nízký výkon. Vkládání dat je v SQLite jako vlastní transakce a tu dokončí až, když jsou všechny data uložena bezpečně na disku. To se dá vyřešit rozdělením procesu ukládání velkého množství dat na více transakcí.

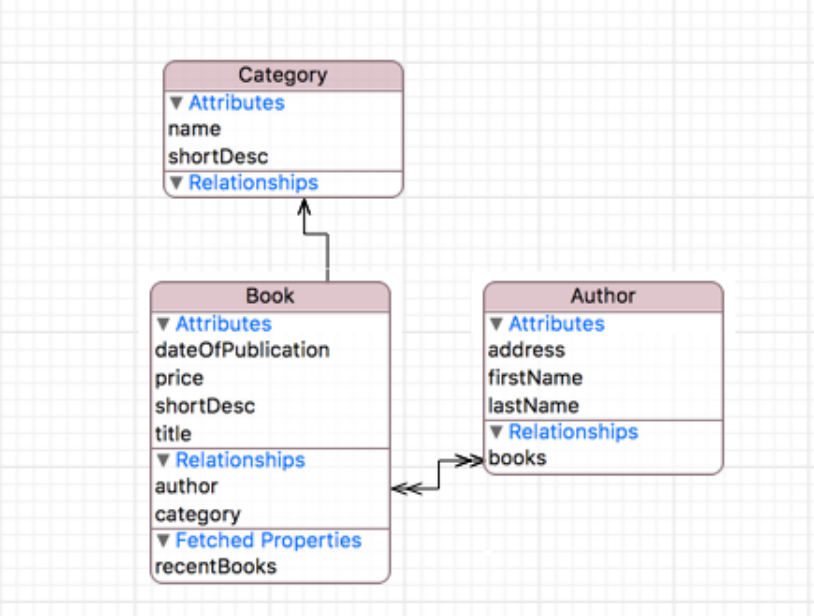
SQLite se hodí pro malé nasazení a to převážně u mobilních aplikací, nebo malých a středních webů. Díky jedinému souboru na disku a možnosti práce s databází bez nutnosti použití serveru. Někdy lze najít SQLite jako cache dat nad některým větším databázovým systémem, kvůli její rychlosti čtení a jednoduchosti. Rozloží se tak i zátěž vlastního databázového serveru.

2.4.5 Core Data

Framework Core Data se používá ke správě modelové vrstvy aplikace. Jedná se o nadstavbu nad databází *SQLite* a zastupuje funkci *objektově relačního mapování* (ORM). Nabízí obecné a automatizované řešení pro časté úkoly spojené s životním cyklem objektů, jejich správy a perzistence. Typicky se sníží o 50 až 70 procent množství napsaného kódu pro podporu modelové vrstvy [21]. Je to způsobeno tím, že vývojáři nemusí implementovat, testovat nebo optimalizovat mnoho funkcí, jenž jsou často potřeba k práci s daty.

Mezi množstvím funkcí, které tento framework nabízí, patří i automatická validace vlastností hodnot, relace mezi objekty, nástroj na migrační schéma, seskupování, filtrování a organizace dat v paměti a uživatelském rozhraní a další. Nabízí také sofistikované řešení dotazů nad databází. Místo psaní SQL dotazů se používají instance třídy *NSPredicate*, kde se pomocí řetězce specifikují logické podmínky pro získání nebo filtraci dat.

Při vytváření databáze s Core Data, stačí do projektu importovat Core Data framework a nadefinovat schéma a databázi. Je potřeba popsat jednotlivé entity, jejich vlastnosti a vztahy mezi nimi. Pro schéma se používá tzv. model správy objektů, instance třídy *NSManagedObjectModel*. Díky této třídě může Core Data mapovat záznamy v persistentním úložišti k objektům, jenž jsou použity v aplikaci. Díky nástroji pro vývoj aplikací v jazyce Swift, vývojové prostředí *Xcode*, je navržení a vytvoření takového schématu velice jednoduché. Tento nástroj nám nabízí přehledné, grafické prostředí pro tvorbu entit, vlastností i vztahů. Zobrazuje informace v tabulce i v grafickém modelu. Vytvoření schématu je velice podobné návrhu relační databáze. Není potřeba se ovšem starat o primární a cizí klíče, triggery a podobně. Zdrojový soubor pro Core Data model má koncovku `.xcdatamodeld`.



Obrázek 2.3: Příklad diagramu entit v Xcode Data Model editoru

Core Data zařizuje veškerou interakci s úložišti externích dat. Díky tomu se aplikace může soustředit na byznys logiku. Skládá se ze tří primárních objektů: kontext správy objektů (*NSNSManagedObjectContext*), koordinátor perzistentního úložiště (*NSPersistentStoreCoordinator*) a model správy objektů (*NSManagedObjectModel*).[21]

Model správy objektů se stará o popis dat, jenž budou zpřístupněny Core Data. Koordinátor perzistentního úložiště je zodpovědný za realizaci instancí entit, které jsou definované v modelu. Je to prostředník mezi kontextem a modelem. Vytváří nové instance entit v modelu a získává existující instance z perzistentního úložiště. Perzistentní úložiště může být na disku a nebo v paměti. Konečně kontext správy objektů je v aplikaci nejpoužívanější. Při čtení data z perzistentního úložiště, je jejich kopie uložena do kontextu a dále se s ní pracuje v daném kontextu. Data se pak musí synchronizovat s persistentním úložištěm, aby se dali znovu použít při dalším spuštění aplikace. Při ukládání se kontext ujistí, zda jsou data ve validní formě. Tudíž když jsou data v kontextu, nejsou ještě uložena v persistentním úložišti a data v kontextu a v perzistentním úložišti se mohou lišit. Protože jsou data z kontextu načtená v paměti a jsou připravena na použití, je k nim přístup a práce s nimi rychlejší.

Všechny spravované objekty musí být registrované kontextem správy objektů. Kontext si uchovává záznamy o změnách dat. Díky tomu, že kontext nepřepisuje perzistentní data, ale sleduje změny, které byly provedeny, je možnost využít i funkce k vrácení provedených změn.

Bohužel framework Core Data je zastaralý a byl prve navržen pro jazyk

Objective-C. Obsahuje nedostatky při použití s jazykem Swift. Pro někoho kdo s Core Data nikdy nepracoval, se toto řešení jeví jako složité a špatně jej chápe. Než se programátor seznámí se všemi funkcemi a praktikami, stráví mnoho času v dokumentaci. Tento framework je komplexní a nabízí plno funkcí, ale naučit se ho může trvat. Správné nastavení a použití je mnohdy složitě proveditelné. Na druhou stranu, když už frameworku programátor rozumí, dokážeme s ním pracovat rychle a efektivně. Nabízí příjemnější řešení než je práce s čistým SQLite.

Výhodou tohoto přístupu je to, že odstraňuje starost o správu objektů jako takovou. Stačí nadefinovat byznys logiku a jednotlivé entity, jejich vlastnosti a vztahy. Díky kontextu správy objektů lze měnit data, aniž by se ovlivnila data uložená na disku a až po kontrole je teprve uložit do persistentního úložiště. Framework nabízí zastínění nízkoúrovňové logiky a odstraňuje nustnost se starat o implementaci archivace dat a způsob jejich uložení. Většina věcí se provádí automaticky primárně přes použitý kontext.

2.4.6 Realm

Je nutné neopomenout zmínit i projekt *Realm*. Tento projekt míří na post náhrady SQLite a Core Data. Jedná se o multiplatformní databázi podporující iOS, OS X, (jazyky Objective-C a Swift) a Android (jazyk Java). Dovoluje sdílení Realm souborů mezi platformamy, stejné vysoce-úrovňové modely pro Javu, Swift a Objective-C. Dovoluje zapisovat podobnou byznysovou logiku na všechny tyto platformy. [22]

Chlubí se svojí jednoduchostí v použití a rychlostí. Nejedná se o nadstavbu nad SQLite. Místo toho používá vlastní perzistentní úložiště. Rychlostí díky návrhu, jenž nepodporuje kopírování. Tvrdí, že je jejich řešení rychlejší, než nadstavba a často i než samotné SQLite [23]. Podporují pokročilé funkce jako šifrování, grafové dotazy a snadné migrace.

Snaží se udržet API co nejjednodušší a drží se pouze nejpoužívanějších tříd a jedné pro migraci, které bohatě stačí pro správu databáze. Také díky zrušení nadstavby ORM, se odstranily i problémy s rychlostí a správou. Data jsou přímo vystavena jako objekty a dotazovaná kódem.

Projekt je veden jako veřejný s podporou veřejnosti. Stejnojmenná společnost, která tento projekt vede, dodává i jednoduché doplňky pro použití tohoto řešení. Z mého pohledu se jedná o velice slibný projekt a kvůli stáří Core Data a jejich nedostatkům se může jednat o příjemnou náhradu.

2.4.7 Shrnutí

Pro správu jednoduchých dat, u nichž není složitá struktura a je potřeba ukládat malé množství informací se nejlépe hodí Property list řešení. S ním spojené NSUserDefaults se hodí k ukládání uživatelem definovanými informacemi, nastavení aplikace a případně malá data, jenž jsou potřeba pro běh aplikace.

Pokud je požadavkem řešit velké množství dat, nebo složitější modely se vztahy, je výhodné zvolit databázi. SQLite je běžné relační řešení, které je ve vývoji aplikací takřka standardem. Je nejrozšířenější a otestované. Pro odstranění problému s řešením nízkoúrovňové práce s databází, Apple nabízí framework Core Data, jenž je ORM nad SQLite a poskytuje funkce pro jednodušší práci s daty.

Stále ovšem existuje možnost zapisovat data do souborového systému. Tento krok je vhodné volit, pokud jsou vyžadovány specifické požadavky. Zpracování tohoto řešení přináší více práce a vyžaduje více času než řešení předchozí.

Projekt Realm, který nabízí řešení databáze pro mobilní platformy, se snaží vyplnit nedostatky SQLite a Core Data. Jedná se o zajímavý projekt a dá se předpokládat jeho značný potencionál na správu dat v programovacím jazyce Swift.

Kapitola 3

Analýza

Náplní kapitoly je analyzovat knihovny řešící stejné téma a nebo mající podobný cíl, jako tato diplomová práce. První část kapitoly uvádí do problému syntaktické analýzy JSON dokumentu ve Swiftu za použití standardního frameworku Foundation. Další části jsou zaměřeny na knihovny, řešící pouze syntaktickou analýzu JSON. Následně knihovny, které se zabývají serializací JSON objektů do modelových tříd ve Swiftu, knihovny řešící i perzistentní úložiště. Poslední část kapitoly představuje ostatní knihovny, které jsou nápomocny při řešení problému této diplomové práce, ale nemusí přímo řešit daný problém.

Některé knihovny jsou jednoduché a jiné komplexní. U některých lze nalézt dostatečnou dokumentaci a u jiných autoři odkazují na testy, pro prozkoumání funkčnosti. Pro tuto analýzu jsou vybrány nejrozšířenější knihovny, které jsou reprezentativní v rámci dané problematiky. U jednotlivých knihoven došlo k prozkoumání dokumentace a následné vyzkoušení funkčnosti a možností na praktickém příkladu.

3.1 Současný stav

Následuje představení příkladu[1], jak se pracuje s JSON dokumentem ve Swiftu, bez použití knihoven třetích stran. V příkladu je použit pouze framework Foundation, který byl zmíněn výše. Jak a odkud získat data není předmětem tohoto příkladu, co je v tomto případě důležité, je jaká data jsou k dispozici. Data lze získat z webového API a nebo jsou přečtena ze souboru. V tomto případě jsou data následující:

```
{ "people": [
  {
    "person": {
      "name": "Jane",
```

[1] Příklad je na přiloženém CD v souboru s názvem `json_parser_example.playground`

```
      "age": "22"
    }
  } , {
    "person": {
      "name": "Alex",
      "age": "34"
    }
  }]
}
```

Stručně řečeno, jedná se o objekt `people`, který obsahuje pole objektů osob, jejichž vlastnosti jsou jméno a věk.

Pro tyto účely jsou data načtena v objektu `jsonData`, jenž je instancí třídy NSData. JSON dokument deserializuji pomocí třídy NSJSONSerialization takto:

```
var json : Dictionary<String, AnyObject>!
do {
  json = try NSJSONSerialization.
  JSONObjectWithData(jsonData,
                     options: NSJSONReadingOptions())
                     as? Dictionary<String, AnyObject>
} catch {
  print(error)
}
```

Deserializovaná data jsou uložena v proměnné `json` a nachází se ve slovníku (Dictionary). Teď už jen stačí získat jednotlivé hodnoty co jsou objektem zájmu. Následuje příklad ve kterém jsou přečteny všechny osoby z objektu `people` a odeslány do výstupu. Kód byl doplněn o výpis chyb. V praxi by bylo na místě chyby vypisovat do chybového výstupu, popřípadě je ještě dále zpracovat. Pro tuto ukázku ovšem plně postačí tento kód:

```
guard let people = json["people"]
  as? Array<[String:AnyObject]> else {
  print("Error reading json.")
  return
}
for item in people {
  guard let person = item["person"] as? [String: AnyObject]
    else {
      print("Error when reading person.")
      continue
  }

  guard let name = person["name"] as? String,
```

```
    let age = person["age"] as? Int
    else {
      print("Error when reading person's properties.")
      continue
  }

  print("\(name)'s age is \(age)")
}
```

Bylo použito jsem klíčové slovo `guard`, jenž zajistí, že pokud není splněna podmínka, nepokračuje se dále. Jedná se o podobný princip `if` podmínky. Jak je vidět z kódu, Swift díky své striktní typovosti a optionals, znatelně prodlužuje délku kódu. Předtím než jakoukoli hodnotu lze použít, je potřeba ověřit její typ. Kód je sice bezpečnější a předejde se tak chybám, ale snižuje se čitelnost a prodlužuje se délka napsaného kódu.

Kód byl spuštěn a výsledkem byl následující výpis:

```
Jane's age is 22
Alex's age is 34
```

Deserializace z JSON do proměnných proběhla úspěšně a nedošlo k chybě. Co se ovšem stane v případě, když se změní věk z `"age": 22` na `"age": "22"`. Tedy věk bude řetězec a ne celé číslo. Po změně a opětovném spuštění, dochází k chybě u čtení vlastností objektu `person` a vypíše se chybová hláška. Pro vyhnutí se chybě je nutné přidat další podmínku, která zajišťuje, že věk může být řetězec nebo celé číslo a případně dodá funkci pro konverzi.

Z ukázky je patrné, že kód je poměrně dlouhý pro jednoduchou deserializaci krátkého JSON dokumentu. Přidáním podmínek, které řeší různé varianty typů hodnot, se jen prodlouží. Každou další hodnotou přichází další podmínka a kód je čím dál tím méně čitelný. Kdyby nebylo použito klíčové slovo `guard`, kód by byl o něco méně čitelný, protože `if` podmínka by kód jen prodloužila.

Nevýhodou tohoto přístupu je délka kódu a s ním i nižší čitelnost. Lze vzít v potaz, že předchozí příklad obsahuje krátký a jednoduchý JSON dokument. V praxi se pracuje s mnohem rozsáhlejšími daty. Pokud se vyvíjí například mobilní aplikace, je časté, že se vývojáři nechtějí zdržovat ošetřováním JSON dat a vypisováním složité dokumentace, když projekt má trvání pár měsíců. Podrobné návrhy a postupy se uplatňují spíše u velkých projektů a systémových aplikací, než u projektů menších. Vývojáři proto spíše vyhledávají rychlé a efektivní řešení, v podobě knihoven třetích stran zajišťující práci s JSON dokumenty.

3.2 SwiftyJSON

Knihovna *SwiftyJSON* od autora Ruoyu Fu [24], se zabývá syntaktickou analýzou JSON dokumentu a nabízí jednoduché řešení pro práci s JSON ve Swiftu.

Odstraňuje nutnost ošetřování chyb při čtení hodnot. Není potřeba složitě ošetřovat cestu ke zvolené hodnotě. Díky jednoduchému řetězení parametrů se lze dostat k požadované vlastnosti a hodnotu získat zavoláním vlastnosti typu, jenž chceme přečíst (`string`, `double` atd.).

Získání jména první osoby z příkladu 3.1 lze realizovat následovně:

```
let name = json["people"][0]["person"]["name"].string
```

Pokud hodnota neexistuje, cesta není správná a nebo se jedná o špatný typ, výsledkem je prázdná hodnota. V opačném případě je výsledkem požadovaná hodnota. Zápis je elegantní, jednoduchý a čitelný. Jestli je potřeba vypsat chybu v případě získání prázdné hodnoty, učiní se tak zavoláním `.error`.

Nabízí i další možnosti jak specifikovat cestu, jako například pomocí pole. Knihovna umožňuje číst různě volitelné a nevolitelné vlastnosti objektů. Volitelné čte tak, že vrací hodnotu a nebo `nil`. Při použití vlastnosti s návěštím `Value` (např.: `json["name"].stringValue`) lze dostat hodnotu a pokud není specifikovaná získáme hodnotu výchozí. Ta záleží na typu, pro řetězec to bude prázdný řetězec a pro číslo to bude nula. Taktéž lze tímto způsobem zjistit zda hodnota existuje.

SwiftyJSON je knihovna zaměřená na serializaci a deserializaci JSON objektů. Má jednoduchou syntaxi a kód je dobře čitelný. Výhodou je možnost specifikovat volitelné a nevolitelné vlastnosti objektů. Nenabízí ale možnost mapování JSON do modelových tříd, práci s Core Data nebo NSUserDefaults je nutné zajistit bez pomoci této knihovny.

3.3 Swift-json

Další knihovnou, která řeší stejný problém je *Swift-json* od Dan Kogai [25]. Tato knihovna obsahuje podobnou funkcionalitu jako SwiftyJSON. Nabízí dotazovací vlastnosti, které je možno využít k zjištění jestli hodnota je určitý typ nebo jestli existuje. Oproti SwiftyJSON se dá dotázat o jaký typ se jedná pomocí vlastnosti `type`. To může být velice užitečné při určování typu hodnot, když není jistota co se získá, ale i tak si s hodnotou přejeme pracovat. Dokonce se lze dotázat pomocí `isLeaf`, jestli je hodnota konečná, tedy listem JSON stromu. Co ovšem tato knihovna neřeší jsou základní hodnoty, tedy pokud je hodnota prázdná, vždy je výsledkem prázdná hodnota a ne základní jako je například prázdný řetězec. Když hodnota je prázdná je výstupem NSNull.

3.4 Gloss

Gloss knihovna nabízí nejen syntaktickou analýzu JSON, ale i mapování JSON do objektů a zpět. Jak je uvedeno v dokumentaci, jedná se o další JSON parsovací knihovnu řešící stejný problém jako ostatní, ale se zaměřením na specifické problémy. Mezi tyto problémy patří [26]:

- Neměnitelné modely – Model by neměl vyžadovat, aby všechny vlastnosti byly dynamické.

- Serializace – Knihovny řešící parsování JSON jsou užitečnější, když nabízejí i transformaci z JSON, ne jen do JSON.

- Snadné vytváření a přizpůsobitelnost modelu – Modely by měly být kompaktní a deserializace by měla povolovat volitelné transformace hodnot.

- Srozumitelnost – Knihovna by měla být srozumitelná i na úrovni zdrojového kódu.

Z předchozího popisu je patrné, že je knihovna navržena, podle častých problémů s JSON ve Swiftu. Neměnitelné modely je podobným principem, který byl představen v části 2.2. Přístupu kdy se používají konstanty a ne dynamické proměnné.

Z příkladů v dokumentaci lze zjistit, že model je struktura definující vlastnosti objektu. Tato struktura musí přijímat protokol *Decodable*. Tento protokol povoluje dekódovat objekt z JSON. Pro enkódování do JSON se použije protokol *Encodable*. Následně stačí jen implementovat inicializační metodu `init?(json:)`. V této metodě se specifikuje, jak se namapují hodnoty z JSON, jenž je v parametru metody, do vlastností objektu. Mapování se řeší elegantně pomocí binárních operátorů `<~~` a `~~>`. Pokud model obsahuje nevolitelné vlastnosti, je nustnost jejich mapování ošetřit pomocí podmínek. Jestliže existuje případ, kdy povinná hodnota je prázdná, lze pomocí podmínek definovat, jestli se objekt vytvoří a nebo se vlastnosti přiřadí základní hodnota.

Tento přístup je elegantní a ušetří hodně kódu. Mapování je řešeno přímo v modelu, což je výhodné v případě změny vlastností nebo modelu.

Dále je uvedena jednoduchá ukázka inicializace modelu `person` z příkladu na začátku této kapitoly. Pro ukázku jsou definovány vlastnosti jméno jako nevolitená a věk je volitelná (otazník za názvem proměnné).

```
struct Person: Decodable {
  let name: String
  let age: Int?

  init?(json: JSON) {
    guard let name: String = "name" <~~ json
      else { return nil }

    self.name = name
    self.age = "age" <~~ json
  }
}
```

Gloss také dovoluje pracovat s *vnořenými objekty*. V JSON se může jednat například o objekt adresa, která má vlastnosti město, ulice a číslo popisné. Každá osoba má adresu a aby bylo poznat, že daná adresa patří k osobě je tento objekt vnořen jako její vlastnost. Knihovna Gloss řeší tento problém jednoduchým způsobem. Objekt adresy opět obsahuje protokol Decodable a v objektu osoby je jako vlastnost. Mapování je stejné jako u jednoduchých objektů. Podobně fungují NSURL adresy nebo enumerační hodnoty.

Pro deserializaci objektu se přiřadí objektu protokol `Glossy` a implementuje se funkce `toJSON()`, která volá `jsonify(_:)`. Funkce vypadá následovně:

```
func toJSON() -> JSON? {
  return jsonify([
    "name" ~~> self.name,
    "age" ~~> self.age
    ])
}
```

Opět jednoduchý a elegantní kód. Další zajímavou funkcionalitou je možnost definovat vlastní funkce. To přidává možnost implementovat řešení specifických problémů. Učiní se jednoduše, pomocí klíčového slova `extension`, jenž nabízí Swift pro implementaci dodatečných funkcí třídě, struktuře, enumerátoru nebo protokolu.

Bylo otestováno mapování a když byl správný JSON dokument, tak vše proběhlo v pořádku. Když ale hodnota neexistovala knihovna toto nezaznamená a nevrací hodnotu. Když se jedná o povinnou vlastnost modelu, objekt se nevytvoří. Pokud je vlastnost volitelná, objekt se vytvoří, ale vlastnost bude prázdná hodnota (`nil`). Bohužel se tedy knihovna nestará o chyby a neinformuje jestli hodnota v JSON existuje nebo má jiný typ. To musí programátor implementovat sám.

Tato knihovna je zajímavá a nabízí elegantní a čitelné řešení mapování objektů do modelových tříd. Oproti knihovnám zabývajících se pouze syntaktickou analýzou nenabízí podrobnější specifikaci cesty k hodnotám, zjištění typu hodnot a podobně. Není určena pro detailní analýzu JSON dokumentu. Zaměřuje se na jednoduché mapování do modelových tříd. Často je vyžadována pouze tato funkčnost a může být dostačující. Ovšem v případě, že je nutné podrobněji pracovat s JSON dokumentem je tato knihovna nedostačující.

Další nevýhodou knihovny je, že nenabízí možnost jak zjistit příčinu chyby. Pokud hodnota existuje přiřadí ji, ale pokud dojde k chybě je nutné ošetřit tuto chybu ručně. Ošetřovat chyby v modelu asi není nejlepším řešením, ale v tomto případě se tomu nevyhneme.

3.5 Argo

Na řadě je knihovna od společnosti *Thoughbot, Inc.* s názvem *Argo*. Prezentuje se jako knihovna, která dovoluje získat modely z JSON nebo podobných struktur v podobě, která je typově bezpečná, konzistentní a jednoduše rozšiřitelná [27]. S použitím Argo není potřeba se starat o validaci kódu k ujištění se, že příchozí data jsou správného typu nebo nejsou prázdná. Dokonce hlásí explicitní chyby stavů v případě, že nenajde očekávané hodnoty.

Základní koncept Arga je, že v zájmu zachování typové bezpečnosti, by se měl model úspěšně dekódovat jen pokud všechny parametry jsou řádně splněny. Tudíž Argo zastává přístup, který ve výsledku nevrací objekt, pokud některé podmínky nebyly splněny. Pokud jsou ale vlastnosti objektů definovány jako volitelné, objekt ve výsledku i tak bude, protože nedojde k chybě.

Argo pracuje na podobném principu jako Gloss. Tedy objekt je definován strukturou. Má své vlastnosti a může, ale nemusí, obsahovat konstruktor. Aby se nemuselo pracovat s modelem přímo, lze použít klíčové slovo `extension` a rozšířit tento model o statickou funkci pro dekódování, kde se nadefinuje mapování z JSON do vlastností modelu. Kód rozšíření může vypadat takto:

```
extension Person: Decodable {
  static func decode(json: JSON) -> Decoded<Person> {
    return curry(Person.init)
      <^> json <| "name"
      <*> json <|? "age"
  }
}
```

A k dekódování vybrané osoby se provede následovně:

```
let person : Person? = decode(json)
```

Knihovna Argo používá vzory z funkcionálního programování jako jsou `map` (`<^>`) a `apply` (`<*>`). Tyto vzory napomáhají v lepším zpracování JSON, jakožto slabě typovanému formátu ve Swiftu, jenž je striktně typovaný.

Opět, jako v případě Gloss, pro označení modelu jako dekódovatelného je potřeba implementovat protokol `Decodable`. Mapování objektu je pomocí jednoduché syntaxe a snadno čitelné. Dále využívá operátor `<|`, pro mapování hodnot do vlastností modelu. `json <| "name"` analogie k `json["name"]`. Při použití operátoru `<|?` se mapuje volitelná hodnota. Pokud není nalezena hodnota nastaví se daná vlastnost jako prázdná (nil).

V případě nalezení hodnoty se operátor pokusí přetypovat hodnotu do očekávaného typu. Když nastane chyba, funkce vrátí chybový stav `.TypeMismatch( expected: String, actual: String)` a v případě, že hodnotu nenalezne vrací chybový stav `.MissingKey(name: String)`. To zajišťuje informovanost o procesu syntaktické analýzy a lze tak snadno zjistit kde nastala chyba.

Vnitřně knihovna používá funkctionální programování a genetiku pro řešení volitelných (optionals) hodnot a neznámých typů.

Knihovna nenabízí mapování modelu do JSON formátu. Zaměřuje se pouze na převod z JSON do modelových tříd Swiftu. Je sice pravda, že ve většině případů stačí získávání dat a v případě, že je potřeba některá data převést do JSON, tak stačí jednoduše namapovat vlastnosti modelu do slovníku. Ovšem při práci se složitějšími modely se tato funkcionalita hodí. Tento opačný přístup zajišťuje knihovna jménem *Ogra*, které je věnovaná část níže.

Oproti Gloss knihovně je Argo propracovanější a nabízí jednoduchou a elegantní syntaxi pro převod do modelových tříd. Velikou výhodou jsou chybové stavy, které informují programátora o chybách a tak je jednodušší je identifikovat a případně opravit. Implementace statické funkce pro model, částečně odděluje logiku od modelu a nezatěžuje model implementací konstruktoru. Bohužel chybí převod z modelu do JSON formátu, což Gloss zajišťuje.

Tato knihovna se jeví jako nejlepší, co se týče získání dat z JSON do modelových tříd Swift. Nabízí rozumné řešení volitelných i povinných hodnot a chybové stavy jsou výhodou. Mám s ní i praktické zkušenosti, kdy jsem ji použil na některých projektech. Vyhovovala mi jednoduchost a elegance přístupu. Také jsem oce`nil` chybový výstup. Nevýhodou je že chybí jakákoli možnost práce s perzistentním úložištěm, na to se knihovna nezaměřuje.

3.6 Ogra

Knihovna *Ogra* od autora Craig Edwards, je opakem Argo knihovny. Stará se o převod modelových objektů do JSON reprezentace. Používá datové struktury poskytované Argo knihovnou. [28]

Syntaxe a způsob provedení je podobná tomu co používá Argo. Implementace protokolu `Encodable`, zajišťuje mapování konkrétního objektu do JSON formátu.

Jedná se o výborný doplněk k Argo knihovně, pokud je potřeba převádět objekty do JSON formátu. Jak bylo napsáno výše u popisu Arga, mapování z modelové třídy do JSON formátu není tak častá záležitost a mnohdy se dá provést velice jednoduše i bez použití knihovny. Ovšem v případě, že jsou složitější modely a je požadavek je serializovat do JSON, tak tato knihovna ulehčuje práci.

3.7 Decodable

Decodable je knihovna od Johannese Lunda. Autor se inspiroval Argo knihovnou a tak jsou tyto knihovny velice podobné, ale tato nepoužívá funkcionální operátory [29]. Dekódování tedy vypadá následovně:

```
extension Person: Decodable {
  static func decode(json: AnyObject) throws -> Person {
    return try Person(
                name: json => "name",
```

```
                age: json => "age"
            )
    }
}
```

Protokol `Decodable`, statická funkce to vše je velice podobné. Zde se vypisuje které hodnoty se mapují na které hodnoty objektu. Není to jako Argo knihovny, kde se procházejí vlastnosti postupně a přiřazují se tak jak jsou v pořadí nadefinovány. Obsahuje 3 struktury pro definování chyby: `TypeMismachError`, `MissingKeyError` a `RawRepresentableInitializationError`. Jaké chyby popisují, lze snadno odvodit z názvu.

Ve výsledku je Decodable podobné Argo přístupu a zde si myslím, že jde hlavně o vkus programátora co si vybere. Obě tyto knihovny nabízejí stejnou funkcionalitu, jen v různém provedení.

3.8 Groot

Předchozí knihovny neřešily importování dat do databáze. Knihovna *Groot* od Guillermo Gonzalez se touto problematikou zabývá. Nabízí jednoduchou serializaci Core Data objektových grafů do a nebo z JSON formátu.

Jedná se o řešení převážně napsáné v programovacím jazyce Objective-C a využívající anotace v Core Data modelu pro [30]:

- mapování atributů a vztahů do JSON
- transformaci hodnot pomocí pojmenovaných `NSValueTransformer` objektů
- prezentaci objektového grafu
- podpora pro objektovou dědičnost

Mapování hodnot provádí pomocí anotací. Entity, atributy a vztahy mají ve správě objektových modelů přiřazené *user info*. User info je slovník hodnot, ve kterém se definují metadata ve dvojici klíč-hodnota. Jednotlivé hodnoty anotací specifikují určité funkčnosti. Například klíč `JSONKeyPath` specifikuje mapování vlastnosti modelu na klíče JSON objektu. `JSONTransformerName` definuje název transformace, která bude použita pro změnu typu hodnoty. Dále jsou zde anotace entity, které definují jaké vlastnosti jsou unikátní nebo název entity při použití dědičnosti.

Primární a cizí klíče se v ORM neřeší, jediné co je, jsou univerzální identifikátor pro objekty. Tento identifikátor si spravuje Core Data a nelze měnit. Nelze ho tudíž použít pro zápis identifikátoru z dat v JSON, které se získali z webového API. Groot nabízí ukládat identifikátor tak, že se nadefinuje atribut v modelu zastupující tento identifikátor a název hodnoty, na kterou se má identifikátor mapovat. Jediné co je případně nutné vyřešit je jeho unikátnost.

Groot poskytuje nejen možnost deserializace JSON objektů do Core Data, ale i serializaci Core Data objektů do JSON formátu, jednoduše pomocí funkce `JSONDictionaryFromObject()`. Tato funkce vrací JSON objekt ve slovníkovém formátu.

Problém nastává pokud se model liší od JSON. Pokud při čtení JSON dokumentu knihovna nenalezne požadovaný klíč vloží do vlastnosti objektu prázdnou hodnotu i když je vlastnost definována jako povinná. Proces ani nevrací chybu. V případě že klíč nalezne ale hodnota má jiný typ než se očekává i v případě transformace, dojde k chybovému stavu.

Dojde-li při operaci získání objektu z JSON k chybě, výsledkem je objekt NSError, který obsahuje informace o chybě. Lze tak vypsat obsah chyby a analyzovat příčinu selhání.

Knihovna Groot nenabízí žádný příkladový projekt. Vyzkoušení knihovny vyžaduje vytvoření projektu, základního modelu a připojení API popřípadě načtení JSON dokumentu ze souboru. Konkrétně u této knihovny dochází k chybě, která není v dokumentaci řešena a díky absenci příkladového projektu se obtížně zjišťuje správné použití knihovny. Dokumentace je spíše strohá a tak lze jen odhadovat co vše knihovna dokáže.

Jedná se o zajímavý přístup, ale obsahuje nedostatky, které mohou omezit použití této knihovny. Původně byla tato knihovna cílena na jazyk Objective-C, následně po příchodu Swiftu, byla přidána podpora pro Swift. Výhodou je, že se nemusí psát kód pro mapování do modelových tříd a zpět. Vše se provádí pomocí anotací, a tak není potřeba implementovat mapování přímo v kódu. Zároveň se jedná o nevýhodu, protože když je potřeba cokoli přepsat, změnit nebo přidat, je nutné implementovat novou funkcionalitu mimo anotace v kódu. Logika k modelu, je na více místech, což je nevýhodou při udržování kódu.

3.9 Sync

Konkurentem knihovny Groot je *Sync* od společnosti Hyper. Jedná se o moderní knihovnu, která je mladší než Groot a je implementovaná v programovacím jazyce Swift. Využívá paradigmatu „konvence nad konfigurací“ pro usnadnění vývoje. Automaticky mapuje hodnoty klíčů v tzv. CamelCase nebo snake_case formátu do Core Data. Pro správu operací používá vlákna na pozadí a zajišťuje jejich bezpečné ošetření. Nabízí porovnávání změněných, vložených a nebo smazaných objektů. Mapuje vztahy objektů automaticky a obsahuje tzv. chytrý proces aktualizace, který aktualizuje lokální data pouze v případě, že se liší od zdrojových z JSON dokumentu.[31]

Synchronizaci lze provést pomocí třídy *Sync*. Nabízí i příjemnou možnost specifikovat podmínku, která data se mají synchronizovat. Například ta, jenž jsou den stará a podobně.

Využívá funkce knihovny *DATAStack*, která odstraňuje množství napsaného kódu pro Core Data a nahrazuje ji instancí třídy DATAStack. Tuto instanci následně Sync používá pro synchronizaci. Dále se specifikuje podmínka synchronizace a název entity, která se má synchronizovat.

Knihovna Sync mapuje vlastnosti modelu tak, že porovnává klíče hodnot JSON s názvy atributů. Rozeznává CamelCase i snake_case formáty klíčů. Například pokud je klíč hodnoty ve formátu CamelCase `createdAt` nebo ve formátu snake_case `created_at`, namapuje se na vlastnost modelu `createdAt`, bez nutnosti bližší specifikace.

Core Data řeší unikátnost objektů pomocí identifikátoru objectID, knihovna Sync se zaměřila na unikátnost pomocí vzdáleného klíče z API (často hodnota id). Sync tedy vyžaduje, aby id hodnota existovala u každého modelu, protože jinak neví jak objekty rozlišit. V základním nastavení používá název atributu `remoteID` a na tuto vlastnost mapuje hodnotu s klíčem `id` z JSON. Ovšem lze jako identifikátor nastavit jinou vlastnost, pomocí anotace `hyper.isPrimaryKey` u vybraného atributu. Princip je tedy podobný jako u knihovny Groot.

Pokud se JSON hodnoty a model liší, dají se upravit jednotlivé atributy opět pomocí anotací, podobně jako při použití knihovny Groot. Mapování vztahů je provedeno podobně u obou knihoven.

Při nutnosti mapování slovníků nebo polí je potřeba definovat vlastnost v modelu jako `Binary Data`.

Lze provést výpis stavu mapování pomocí notifikací, použitím třídy *NSNotificationCenter*. Díky notifikacím se odposlouchávají změny v Core Data. Změny obsahují informace o tom co bylo vloženo, změněno nebo smazáno. Pro zjištění chyby při procesu se používá proměnná error, kde je opět podrobný výpis chyby stejně jako u knihovny Groot.

Knihovna Sync je propracovaná a nabízí mnoho funkcí, které ulehčují práci se synchronizací modelů s API. Nabízí elegantní řešení pro případ, že model lokální a vzdálený jsou stejné a nebo hodně podobné. V případě, že se ovšem modely liší, neobsahuje moc možností, jak synchronizaci přizpůsobit. Pokud se neshodují typy hodnot a nebo jsou hodnoty umístěny jinde, než jak jsou v lokálním modelu, data se nenamapují a v lokálním úložišti budou chybět.

Po podrobné analýze jsem zjistil následující. Při stejném pojmenování hodnot a dodržení struktury modelu, je vše v pořádku. Synchronizace probíhá bez problémů a řešení je jednoduché. Knihovna Sync zvládne i volitelné atributy. Knihovna automaticky transformuje různé typy hodnot. Celé číslo do řetězce, ale i z řetězce do celého, případně desetiného čísla. Samozřejmě s čím si neporadí je, že když je očekávaná hodnota celé číslo a zdrojová hodnota je řetězec s různými znaky. Dojde k chybě a synchronizace se zastaví.

Problém nastal při využití možnosti výpisu změn pomocí notifikací. Když dojde k chybě mapování a využíváme výpis změn pomocí notifikací, nejdříve se vypíše chyba a proces se zastaví, ale z notifikací lze vyčíst, že objekty byly uloženy. Tudíž nejdříve dojde k tomu, že se informuje o uložení objektů, ale

následně dojde k chybě a žádné objekty nebyly uloženy. Informace z notifikací neodpovídají realitě v případě chyby.

Také jsem nenašel žádnou možnost přizpůsobení, podobně jako nabízí Groot. Tedy implementování specifického problému mapování.

Knihovna má perfektně vyřešené mapování modelu z API a synchronizace funguje velice dobře. Výhodou je přetypovávání a řešení prázdných hodnot. Nevýhodou je informovanost při chybách a ukončení celé synchronizace, když dojde k chybě i v případě, že se jedná o jediný objekt v kolekci objektů. Dále neřeší pokud jsou modely rozdílné a nenabízí možnost přizpůsobení.

3.10 Ostatní knihovny

Následující část kapitoly obsahuje knihovny, které neřeší problematiku JSON serializace do modelových tříd Swift, ale zaměřují se na jiné problémy. Každopádně jsou užitečné a v případě této diplomové práce je lze použít, protože se jedná o témata hodně blízká.

3.10.1 Alamofire

Alamofire je knihovna zabývající se HTTP komunikací ve Swiftu. Nabízí funkce jako je kódování URL, JSON a plist parametrů, nahrávání dat, validace HTTP odpovědí, autentifikace s NSURLCredential a mnoho dalších. [32]

Jedná se o nejpoužívanější knihovnu, ačkoli jich existuje i více. Některé byly navrženy pro programovací jazyk Objective-C a následně se přidala funkcionalita pro Swift. Tato ovšem vznikla spolu s programovacím jazykem Swift a nabízí jednoduché řešení pro HTTP komunikaci. Společnost Alamofire Software Foundation, jenž stojí za tímto projektem, začínala už s knihovnou *AFNetworking*. Zabývá se HTTP komunikací stejně jako Alamofire, ale pro Objective-C. Když vyšel Swift 1.0, společnost vyvinula Alamofire s předchozími zkušenostmi z AFNetworking a tak vznikla jednoduchá a silná knihovna, kterou nyní používá velké množství aplikací.

Knihovna sice pracuje s JSON, ale nezabývá se propracovanější syntaktickou analýzou a nebo zpracováním do modelových tříd jazyka Swift. JSON řeší jen jakožto parametry HTTP komunikace. Používá NSJSONSerialization pro vytvoření reprezentace parametrů objektu, který nastavuje do těla požadavku.

Často se používá právě pro komunikaci s webovým API a získáním, případně odesláním, dat v JSON formátu. Požadavek ze serveru je často prvním bodem při získání JSON dat.

3.10.2 MagicalRecord

Core Data framework pro podporu perzistentního úložiště a dotazování je poměrně starý a má své nedostatky. Správa dat za použití Core Data API se může jevit jako neohrabaná a těžkopádná. Proto společnost Magical Panda

Software realizovala knihovnu *MagicalRecord*. Jedná se o jednoduchou a populární knihovnu pro ulehčení práce s Core Data. Byla inspirována ActiveRecord načítáním z Ruby on Rails. Cílem této knihovny je [33]:

- zjednodušit kód napsaný pro Core Data
- podpora pro jednoduché, čisté a jednořádkové načítání dat
- podpora modifikací NSFetchRequest, pokud je optimalizace požadavku vyžadují

Tyto cíle splňuje pomocí metod, jenž zabalují běžné nastavení Core Data, dotazů a správy. Jedná se o knihovnu, jenž zjednodušuje a zpřehledňuje práci s Core Data.

Částečně řeší i převod JSON objektů do Core Data objektů. Pro importování dat do databáze lze použít metoda, jenž přijímá JSON slovník a pokusí se mapovat jednotlivé hodnoty ze slovníku do modelu. Očekává však, že klíče a hodnoty souhlasí s cílovým modelem. Pokud tomu tak není, dá se ručně nadefinovat některé vlastnosti v metodách `willImport` a `didImport`.

3.10.3 CoreValue

Další knihovnou pro lepší práci s Core Data je *CoreValue* od Benedikta Terhechte. Jedná se o jednoduchou nadstavbu nad Core Data, která přidává funkcionalitu výhodnou pro programovací jazyk Swift. Core Data je framework původně vytvořen pro programovací jazyk Objective-C. Swift představil univerzální typy hodnot, které jsou rychlé, bezpečné, prosazují neměnnost, ale při použití Core Data frameworku se tehto výhod nevyužije. CoreValue by měla tento nedostatek vyřešit. Její funkcí je zapouzdření hodnot do Core Data objektů a rozbalení Core Data objektů do proměnných. Také obsahuje jednoduchou abstrakci pro dotazy, aktualizaci, ukládání a mazání. [34]

Pracuje se strukturami a dobře řeší `let` a `var` vlastnosti. Autor se inspiroval Argo knihovnou pro serializaci a deserializaci JSON dat. Používá stejného principu curry implementace. V podstatě se s touto knihovnou pracuje podobně jako s Argo a to tak, že se definuje model pomocí struktury definující `fromObject` metodu. Zde se specifikuje mapování jednotlivých vlastností. Pro opačný postup, tedy ze struktury do Core Data, se použije protokol `BoxingStruct`.

Práce s touto knihovnou je jednoduchá a nabízí užitečnou funkčnost. Nejenže ulehčuje práci s Core Data, ale i plně podporuje programovací jazyk Swift. Nabízí i metody pro získání objektů z databáze pomocí dotazů.

3.11 Shrnutí

V této kapitole jsem analyzoval syntaktickou analýzu JSON v programovacím jazyce Swift bez použití knihovny třetí strany. Následně byly prozkoumány jednotlivé knihovny řešící problematiku serializace JSON objektů do modelových tříd Swift.

Nejprve jsem zhodnotil knihovny SwiftyJSON a Swift-json, které se zaměřují pouze na syntaktickou analýzu JSON dokumentu. Poté jsem analyzoval knihovny Gloss, Argo, Ogra a Decodable, které se zabývají mapováním JSON do modelových tříd a zpět. Zde bych vyhodnotil jako nejvýhodnější řešení Argo knihovnu a v případě, že je nutné mapovat z modelových tříd do JSON formátu, tak zvolit knihovnou Ogra.

Následovaly knihovny Groot a Sync, jenž se oproti ostatním zaměřují na Core Data a nabízí mnohem komplexnější řešení. Groot řeší mapování modelů pomocí anotací, což přináší jednoduchou implementaci. Jedná se zároveň o nevýhodu, pokud je potřeba upravit mapování modelů. Sync se zde jeví jako propracovanější řešení díky automatickým transformacím a jednoduchému použití, ale postrádá možnost větší přizpůsobitelnosti.

V sekci 3.10 jsem představil knihovny, jenž nepřímo řeší problém této diplomové práce. Zde jsem uvedl Alamofire jakožto řešení pro moderní webové API. Pro jednodušší práci a doplnění nedostatků Core Data, jsem uvedl MagicalRecord a CoreValue. MagicalRecord knihovna zjednodušuje práci s Core Data a CoreValue se zaměřuje na práci s programovacím jazykem Swift.

Všechny zde analyzované knihovny mají alespoň krátkou dokumentaci, kde popisují své řešení a funkce. Některé mají dokumentaci propracovanou, některé jen stroze popisují svoji funkčnost. Často se stávalo, že ačkoli dokumentace se zdála vyčerpávající, po vyzkoušení a otestování jsem zjistil, že není zdokumentováno všechno. Některé funkce chybí a jiné nejsou popsané. Přišel jsem na funkčnost, kterou dokumentace ani nepopisuje, jako například automatická transformace v knihovně Sync.

Některé knihovny dodávají příklady použití v dokumentaci. Jiné odkazují na unit testy, které by měly informovat o funkčnosti a funkcích knihovny. Často jsem ale z testů dostatečně nevyčetl co vše knihovna nabízí, protože testy nebyly dostatečně informativní.

Ideálním případem je, když knihovna obsahuje příkladový projekt. Může se jednat jen o jednoduchý projekt reprezentující základní funkčnost knihovny, kde si lze vyzkoušet jak se knihovna chová za určitých podmínek. Pokud ovšem takovýto projekt knihovna nenabízí, je nutné vytvořit projekt, importovat knihovnu a správně použít podle dokumentace.

Problém, který u většiny knihoven shledávám je, že neobsahují dostatečnou dokumentaci a popis jak se používají a co všechno nabízejí. Prezentování funkčnosti knihoven na základě testů je podle mého názoru nedostatečné. Doplnit příkladový projekt, který obsahuje základní funkčnost a možnost úprav podle uvážení, je nejlepší řešení.

Knihovny lze rozdělit do tří skupin. V první skupině jsou ty, které řeší syntaktickou analýzu (SwiftyJSON a Swift-json). Druhá skupina zahrnuje knihovny řešící serializaci nebo deserializaci JSON objektů do modelových tříd jazyka Swift (Gloss, Argo, Ogra a Decodable). Třetí skupina obsahuje knihovny řešící serializaci a deserializaci JSON objektů do Core Data objektů (Groot a Sync).

Kapitola 4

Návrh a Realizace

Analýza provedená v předchozí kapitole, představila knihovny řešící problém serializace a deserializace JSON objektů v programovacím jazyce Swift. Ukázala, že některé problémy jsou dostatečně vyřešené, ale jiné případy obsahují neúplná nebo nedostatečná řešení.

Protože jsem v kapitole Technologie v sekci 2.4 představil nejeden z přístupů modelových tříd v jazyce Swift, rozdělil jsem tuto kapitolu na více částí. Rozdělení jsem provedl podle toho, na jaký problém se zaměřuje a s jakými daty se pracuje.

Pokud se pracuje s jednoduchými strukturami dat a ukládá se jen malé množství, zvolí se pravděpodobně Property list a nebo NSUserDefaults. Sekce 4.1 se zabývá tímto problémem a jaké řešení použít.

Jestliže se pracuje s větším množství dat nebo složitějšími modely, s největší pravděpodobností se použije databáze. V tomto případě je zapotřebí zvolit více komplexní přístup než v případě prvním. Návrh řešení a případnou realizaci návrhu proberu v sekci 4.2.

Posledním případem je zápis do souboru. Zde v sekci 4.3 navrhnu obecné řešení, jaký postup lze zvolit a co by mohlo v dané situaci pomoci. Tato poslední sekce bude nejobecnější, protože lze jen obtížně odhadovat, jaká bude struktura dat, jestli půjde o mediální obsah a jaké budou vztahy mezi daty. Jedná se o specifický přístup, proto se zde zaměřím na obecný návrh a doporučení.

4.1 Práce s hodnotami

Prvním návrhem, kterým se chci zabývat, je návrh řešení při použití malého množství dat s jednoduchou strukturou, popřípadě žádnou. Tento případ je poměrně častý při vývoji mobilních aplikací a aplikací s malým množstvím informací. Data pro zpracování mohou být jen informativní nebo mohou sloužit ke specifikaci nastavení provedená uživatelem. Také se často jedná o informace pro přihlášení uživatele, jeho uživatelské jméno, email a další identifi-

kační údaje. K těmto informacím není potřeba složité databáze, struktury ani procesy. Stačí jednoduché úložiště dat. Zde je výhodné použít Property list nebo NSUserDefaults.

Jaké z těchto dvou řešení zvolíme, záleží na požadavcích. Výhody a nevýhody jsem popsal v části 2.4.2 a 2.4.3. Jen připomenu případy použití. Pokud je potřeba ukládat nastavení a informace, které jsou nunté při startu aplikace, je vhodné použít NSUserDefaults. Jestli je požadavkem kontrola správy přístupu k datům, je výhodné zvolít Property list. Práce s oběma úložišti je v podobná. Častým řešením je použití varianty s třídou NSUserDefaults, protože je vše nastaveno a připraveno k použití při startu aplikace.

Jestliže se pracuje s webovým API, které se používá pro získání a odesílání JSON dat, což je častým případem, doporučuji zvolit knihovnu pro komunikaci s API. V kapitole Analýza jsem představil knihovnu Alamofire a tu doporučuji, protože s ní mám pozitivní praktické zkušenosti a k tomuto účelu plně vyhovuje. Každopádně analýza knihoven a řešení pro komunikaci s webovým API je nad rámec tohoto textu. V tomto případě slouží jen pro účely získání a odeslání JSON dokumentu.

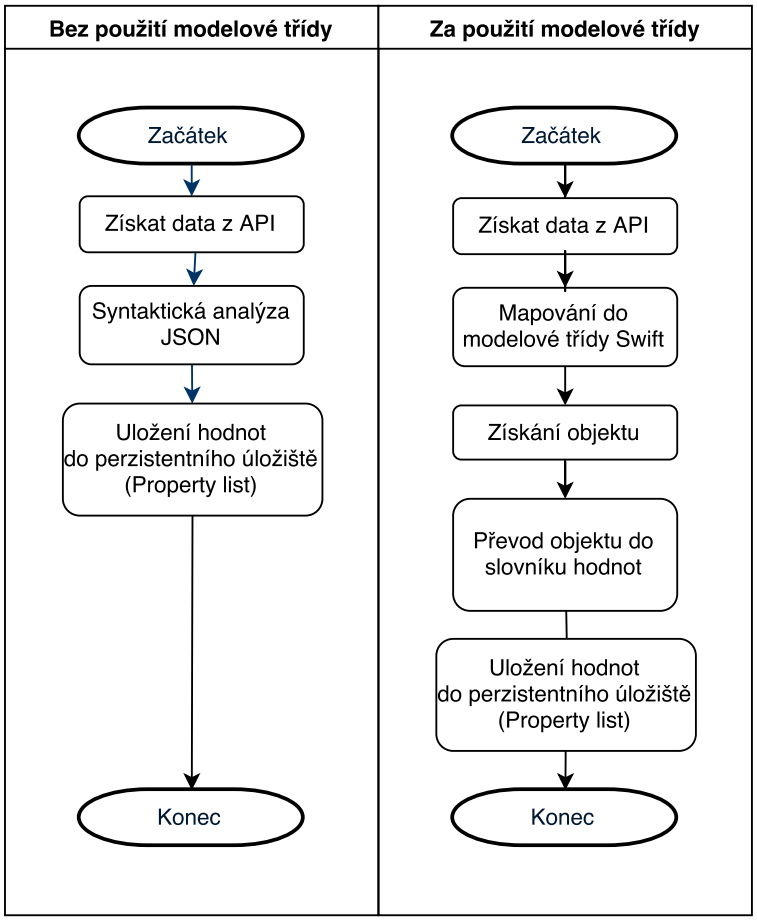
Je tedy vyřešena otázka získání dat a jejich uložení, zbývá navrhnout převod dat z JSON dokumentu do úložiště a zpět. Není potřeba převod do složitých struktur a ani vazby mezi nimi. Úložiště zde zastupuje databáze, která je reprezentovaná strukturou klíč-hodnota, tedy stejnou strukturou jako je JSON. Teoreticky by stačilo přečíst data z JSON a uložit do této databáze. Tímto způsobem se ale ztratí přehled o uložených datech a nelze určit, co bylo a jak to bylo uloženo.

Zde bych, po analýze kterou jsem provedl, zvolil knihovnu řešící syntaktickou analýzu JSON. Není potřeba převodů do struktur a nebo modelových tříd jazyka Swift. Často stačí jen práce s jednotlivými hodnotami. Proto by tato knihovna měla být dostačující.

V případě, že je potřeba zpracovat data, jenž mají určitou strukturu, lze zvolit přístup, kdy se serializují data do modelové třídy Swift a odtud se následně uloží do perzistentního úložiště.

Jedná se o složitější postup, než v prvním navrhovaném případě jednoduchého ukládání hodnot z JSON do databáze. V prvním případě se pouze přečtou data, zkontrolují, případně se změní klíč nebo cesta k hodnotě a uloží do perzistentního úložiště. Struktura dat může být stejná, jako v JSON a nebo lze zvolit strukturu vlastní. To záleží na konkrétním způsobu struktury dat. V tomto přístupu není potřeba deserializovat struktury.

Ve druhém navrhovaném případě, serializace do modelové třídy, je potřeba implementovat převod do modelové třídy a získat objekt se kterým lze dále pracovat. Tento objekt ovšem nelze uložit do slovníkového perzistentního úložiště, protože do struktury klíč-hodnota lze ukládat pouze primitivní typy, pole a slovníky. Je potřeba objekt serializovat do slovníku hodnot. Provedou se dvě mapování, jedno do struktury a druhé do perzistentního úložiště.



Obrázek 4.1: Diagram aktivit uložení dat z API do perzistentního úložiště Property list

Co se týče serializace lokálních dat do formátu JSON, je odpověď triviální. Stačí vzít data a uložit je do slovníku pod klíče tak, jak vyžaduje webové API. O odeslání JSON dokumentu se postará knihovna zajišťující komunikaci s API.

Přístup s modelovou třídou je v tomto případě zbytečně složitý. Výhodou tohoto řešení je přehlednost u složitějších struktur a nebo více dat. Proto se přikláním k prvnímu navrhovanému přístupu, tedy použít jednoduchou knihovnu pro syntaktickou analýzu. K realizaci stačí vybrat jednu z analyzovaných knihoven, která tento problém řeší.

4.1.1 Výběr knihovny

Protože se jedná nejedná o složitý problém je návrh jednoduchý. Co je potřeba v takovémto případě řešit je správná volba knihoven. Není ani potřeba navrhovat vlastní řešení, protože knihovny, jenž jsem analyzoval, jsou dostatečné.

Z analýzy vyplývá, že je na výběr ze dvou, které jsou pro tento případ dostatečné: SwiftyJSON a Swift-json. Obě nabízejí požadovanou funkčnost. Pro většinu případů syntaktické analýzy JSON plně dostačuje knihovna SwiftyJSON.

V případě nutnosti rozšířené analýzy JSON dokumentu, je vhodnější knihovna Swift-json. Nabízí možnosti dotazování o jaký typ se jedná, jestli je hodnota listem stromu a podobně. Tuto knihovnu lze zvolil v případě, že je potřebné analyzovat složitý JSON dokument a vybrat z něj jen některé hodnoty.

Při zvolení přístupu používající modelové třídy, je nutné zvolit knihovnu, která zajišťuje mapování hodnot do objektu. Z analýzy vyplívá, že nejvhodnějším řešením je knihovna Argo. Tato knihovna je jednoduchá a nabízí dostatečné řešení pro daný problém. Zajistí import JSON objektu do modelové třídy Swiftu. Následně je potřeba implementovat metodu, pro serializaci vlastností objektu do slovníku a uložení do příslušné databáze.

4.2 Práce s modely

Zajímavým případem je správa neprimitivních modelů a většího množství dat. Zde už nestačí jednoduchá práce s hodnotami. Je potřeba získávat modely, které mohou obsahovat i vztahy mezi sebou. Mají složitější struktury, než jen pár primitivních hodnot. V tomto případě je potřeba zvážit komplexní přístup a navrhnout vhodné řešení.

Co se týče ukládání dat, je nutné zvolit databázi, která nabízí modely se vztahy a práci s nimi. Nejčastěji používanou databází v programovacím jazyce Swift je SQLite a s ním framework Core Data[2]. Core Data framework nabízí dostatečné řešení pro specifikování databáze, entit, atributů, vztahů a práci s nimi.

Zvolení řešení pro získání JSON z webového API opět ponechávám na vývojáři konkrétního projektu, stejně jako v předchozím návrhu. Jen doporučím knihovnu Alamofire, z důvodu její dostatečné funkčnosti.

Následuje návrh řešení serializace a deserializace modelů. Tento případ je zajímavý a nabízí se zde více přístupů. Možná řešení, podle analýzy a uvážení, jsou tři, ale každé má své výhody i nevýhody a proto jsem se rozhodl zde tyto návrhy představit a zhodnotit. Všechny tři využívají jednu z výše analyzovaných knihoven. Protože ani jeden z nich plně nevyhovovuje, rozhodl jsem se navrhnout vlastní přístup, který je představen jako poslední.

4.2.1 Tři možná řešení

Prvním řešením je použití knihovny, která se nepřímo zabývá převodem JSON do modelu Core Data. Jedná se o knihovnu MagicalRecord. Sice se tato

[2] Ačkoli v kapitole Technologie byl představen i projekt Realm, nebudu se jím zde zabývat, protože se nejedná o příliš rozšířenou databázi. Častým případem je použití frameworku Core Data.

knihovna primárně zaměřuje na ulehčení a zjednodušení práce s Core Data, ale obsahuje i funkci pro importování JSON, v podobě slovníku klíč-hodnota, do modelu. Pokud se klíče hodnot shodují s modelem a hodnoty jsou správně definovány, tak knihovna dokáže importovat data z JSON do databáze. Nabízí i možnost přizpůsobení jednotlivých hodnot, definováním dodatečných informací v metodě `willImport` před samotným importem.

Druhé řešení zahrnuje knihovnu Groot. Provádí serializaci a deserializaci Core Data modelu do a z JSON dokumentu. Díky anotacím lze definovat, které klíče se mapují na jaké vlastnosti objektu a nabízí i možnost přizpůsobení procesu mapování jednotlivých hodnot v kódu. Jak jsem uvedl dříve, je tato knihovna starší a to, že se definice pro jediný model nachází na více místech najednou je nevýhodné.

Třetím řešením je použít knihovnu Sync. Tato knihovna řeší synchronizaci webového API s lokální databází. Je to dobře propracované řešení, ale je striktní a nedovoluje pokročilé přizpůsobení. Lze definovat jen jaký klíč se použije při mapování na danou vlastnost. Při použití tohoto řešení, je potřeba mít stejnou strukturu modelů jako je struktura dat v JSON.

Všechny tři případy nabízejí vhodné řešení. V případech, kdy se synchronizované modely shodují, je nejvýhodnější použít knihovnu Sync. Za pomocí pár řádku kódu, lze synchronizovat i složité modely. V případě že se modely liší, je potřeba zvolit řešení MagicalRecord a nebo Groot. Hlavním úkolem MagicalRecord, ale není práce s JSON a proto neřeší kontrolu JSON formátu. Pro dostatečné ošetření JSON dokumentu je nedostatečné.

Groot nabízí možnost řešení problému s jednotlivými hodnotami pokud se neshodují s lokálním modelem. Lze definovat názvy klíčů zdrojových hodnot, které se mají mapovat na danou vlastnost objektu a nebo transformace hodnot v případě rozdílných typů. V případě knihovny MagicalRecord lze implementovat dodatečné transformace nebo úpravy před importem a knihovna Sync nenabízí možnosti úpravy, kromě definování klíče zdrojové hodnotě v JSON u každé vlastnosti objektu.

Nastává problém pokud je potřeba upravit proces serializace a nebo definovat cestu k hodnotě. Příkladem může být objekt obsahující vnořený objekt, který nemá smysl ukládat jako samostaný objekt. Může se jednat o objekt města, který obsahuje vlastnosti název, země a lokace. Lokace je vnořený objekt, který obsahuje souřadnice zeměpisné šířky a délky (viz níže).

```
{
  "name": "Prague",
  "country": "Czech Republic",
  "location": {
    "latitude": 50.07554,
    "longitude": 14.43780
  }
}
```

V praxi jsem se s tímto případem setkal poměrně často. Lokace nemá vlastní identifikátor, tudíž je obtížné synchronizovat ji jako samostatný objekt. Navíc to z pohledu databázového návrhu nedává smysl. Co je v lokální databázi potřeba, je jeden objekt se čtyřmi vlastnostmi: název, země, zeměpisná šířka a délka.

Knihovny Sync a Groot si s tímto neporadí. MagicalRecord si s tímto poradí v případě, že se tato funkcionalita implementujeme před samotným importem. Implementace tohoto případu musí obsahovat i kontrolu zda klíč pro lokaci existuje a obsahuje slovník hodnot zeměpisné šířky a délky. Dále jestli se jedná o požadovaný typ hodnot, apod.

Jak bylo představeno na příkladu, existuje problém i se zdánlivě jednoduchou strukturou. Vyhodnotil jsem tyto tři řešení jako nedostatečné. Dostatečné řešení jsou knihovny Groot a Sync v případě, že se model z webového API a klientský model shodují. Proto jsem se rozhodl realizovat řešení, které vyhovuje těmto podmínkám a nabízí dostatečnou možnost přizpůsobitelnosti.

4.2.2 Návrh a realizace řešení CoreMapper

Při návrhu jsem se zaměřil na vyřešení přizpůsobitelnosti procesu pro mapování JSON do modelových tříd Swift a uložení do prezistentního úložiště. Vybral jsem knihovny, které jsou vhodné pro tuto realizaci:

Argo pro syntaktickou analýzu JSON a převod JSON dokumentu do modelových tříd programovacího jazyka Swift.

CoreValue pro převod z modelových tříd programovacího jazyka Swift do perzistentního úložiště Core Data.

MagicalRecord pro jednoduší práci s Core Data.

Argo a CoreValue knihovny nabízí stejný přístup a práce s nimi je velice podobná. Je potřeba definovat objekt a implementovat protokoly. Mapování se definuje stejnou syntaxí. MagicalRecord zde používám pouze pro zjednodušení práce s Core Data a významnou roli v řešení nemá.

Toto řešení je postavené na tom, že se definují entity v databázi, tedy objekty v Core Data. Jejich vztahy a vlastnosti. Následně se definují struktury zastupující jednotlivé entity. Tyto třídy obsahují identifikátor, který je také definovaný v databázi jako jeden atribut. Identifikátor může být řetězec a nebo celé číslo.

Ve struktuře se specifikuje název entity, kterou struktura zastupuje a název identifikátoru. Dále je potřeba definovat vlastnosti struktury stejně jako v modelu. Lze připsat i další funkcionalitu například v podobě metod. Tato reprezentace obejktu se používá v rámci aplikace.

Následně je potřeba implementovat metodu `decode` z Argo protokolu `Decodable` a `fromObject` metodu z `CoreMapperStruct`. Toto řešení spočívá v propojení

dvou knihoven a jejich správné projení obstarává struktura `CoreMapperStruct`. Tuto strukturu jsem nadefinoval tak, aby dokázala zpracovat uložení i více objektů do databáze pomocí CoreValue.

Pro uložení jednoho objektu lze zavolat metoda `save` daného objektu. Pro uložení více objektů lze použít metoda `saveInBatch`, kterou má každý objekt zděděný z CoreMapperStruct. Tato metoda postupně prochází všechny objekty, které má vložit do databáze a zjistí, jestli objekt již existuje a jen ho aktualizuje a nebo vytvoří nový. To vše na základě identifikátoru. Identifikátor je tedy v tomto řešení stejně důležitý jako u knihoven Groot a Sync.

Protokol `Decodable` lze rozšířit o funkce, které pomáhají při dekódování hodnot do vlastností modelu. Takto jsem implementoval metodu pro převod řetězce do celého čísla a nebo získání NSDate z desetinného čísla. Podobnou funkcionalitu lze jednoduše doplnit dle libosti.

Při popisu předchozích tří návrhů jsem představil na příkladu problém s vnořeným objektem. Tento problém Argo knihovna řeší velice elegantně a to pomocí jediného řádku kódu. Získání zeměpisné šířky a délky pomocí Argo knihovny vypadá následovně:

```
<*> json <| ["location", "latitude"]
<*> json <| ["location", "longitude"]
```

Toto řešení otevírá možnosti přizpůsobení serializace JSON dokumentu do lokálního modelu a uložení do perzistentního úložiště. Stačí definovat databázi a struktury zastupující jednotlivé entity v databázi. Tyto struktury musí obsahovat název zastupující entitu, název atributu identifikátoru a definování mapování hodnot. Argo i CoreValue tímto přístupem nabízejí možnost přizpůsobitelnosti při definici procesu převodu modelu.

Realizoval jsem tento návrh a zhotovil jsem jednoduchý projekt, který toto řešení demonstruje na příkladu. Jedná se o jednoduchou aplikaci, která zobrazuje zprávy posílané mezi uživateli. Příklad obsahuje databázi s dvěma tabulkami reprezentující zprávy a uživatele. Následně jsou zde dvě struktury zastupující entity v databázi `MessageEntity` a `UserEntity`. Tyto dvě entity jsou v zájemném vztahu N:1. Aplikace po spuštění načte data z databáze a zobrazí zprávy a email uživatele ke kterému zpráva patří.

Jako příklad jsem přiložil soubor, který obsahuje JSON dokument reprezentující možná data. Je zde tlačítko pro aktualizaci, které načte data ze souboru a uloží do databáze za pomocí tohoto řešení. Projekt je na přiloženém CD, pojmenován jako CoreMapper. Zde lze řešení vyzkoušet.

Proces zpracování JSON do modelové třídy a následné uložení do databáze lze provést jednoduše. Následuje příklad z projektu CoreMapper, kde se zpracovávají data zpráv, obsahující i jednotlivé uživatele, které jsou načítány z JSON souboru a uloženy do databáze. Příklad obsahuje i odchycení výjimek a výpis případných chyb:

```
do {
```

```
    let decodedMessages:[MessageEntity] =
      try decodeWithRootKey("messages", object: json)

    try MessageEntity.saveInBatch(decodedMessages,
                                  context:context)
} catch let e as NSError {
    print(e)
} catch let e as MapperError {
    print(e)
}
```

Definoval jsem i jednoduché funkce pro provedení dekódování JSON objektu do modelové třídy jako jsou `decodeWithRootKey` a `decode`. Tyto metody dekódují jeden objekt a nebo pole objektů.

Řešení obsahuje i výpisy chybových hlášek a jsou rozděleny na chyby při syntaktické analýze JSON a při importování do databáze. Tudíž jsou rozlišené chybové hlášky a lze jednoduše najít příčinu chyby. Chyby po vypsání obsahují dostatečné informace o tom, co je špatně.

Toto řešení je vhodné při nutnosti přizpůsobení procesu serializace a deserializace JSON dokumentu. Definováním struktury, která zastupuje entitu v databázi. Nejenže ji lze obohatit o další funkčnost, ale je pod kontrolou proces serializace a deserializace.

4.2.3 Výkon

Po realizaci řešení, jsem se zaměřil na výkon. Změřil jsem výpočetní čas zrealizovaného řešení CoreMapper a porovnal s knihovnami Argo a Groot. Měřil jsem dobu trvání zpracování a importování JSON dat do databáze. Všechna měření byla realizována na stejných datech a za stejných podmínek. Nejdříve jsem změřil dobu trvání procesu pro objekty bez jakékoli vazby (viz tabulka 4.2 a graf 4.3) a následně jsem změřil i pro objekt s jednoduchou vazbou (viz tabulka 4.1 a graf 4.2). Data bez vazby obsahují objekty popisující uživatele a data s vazbou obsahují objekty popisující zprávu s vazbou na uživatele. Data z prvního a druhého měření na sobě nejsou závislá. [3]

Objekty mají jednoduchou strukturu, ale vyžadují potřebnou transformaci identifikátoru z řetězce do celého čísla a hodnoty datumu z desetiného čísla do NSDate.

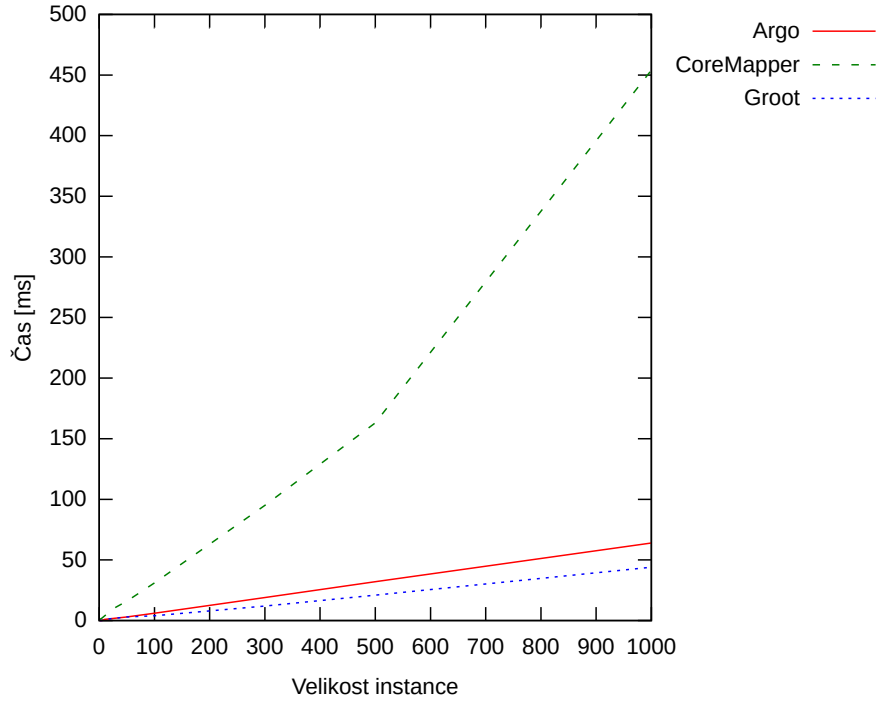
Pro porovnání jsem změřil na stejných datech samotnou knihovnu Argo, která sice neimportuje data do databáze, ale zpracovává JSON do objektu v paměti. Knihovna Argo je v tomto měření za účelem porovnání části CoreMapper, protože CoreMapper Argo knihovnu využívá. Groot je zde pro porovnání s CoreMapper jakožto podobné řešení.

[3] Testy pro měření výkonu jsou dostupné na přiloženém CD.

Měření bylo provedeno opakovaně a z výsledků byla vybrána průměrná hodnota. Měřil se výpočetní čas v milisekundách na počítači Macbook Pro s procesorem 2,2 GHz Intel Core i7.

Tabulka 4.1: Výsledky měření výpočetního času knihoven – data bez vazby

Velikost instance	Argo [ms]	CoreMapper [ms]	Groot [ms]
1	0	1	0
5	0	2	0
10	1	4	1
30	2	11	2
50	3	16	3
100	6	31	4
300	19	95	12
500	32	163	21
1000	64	454	44

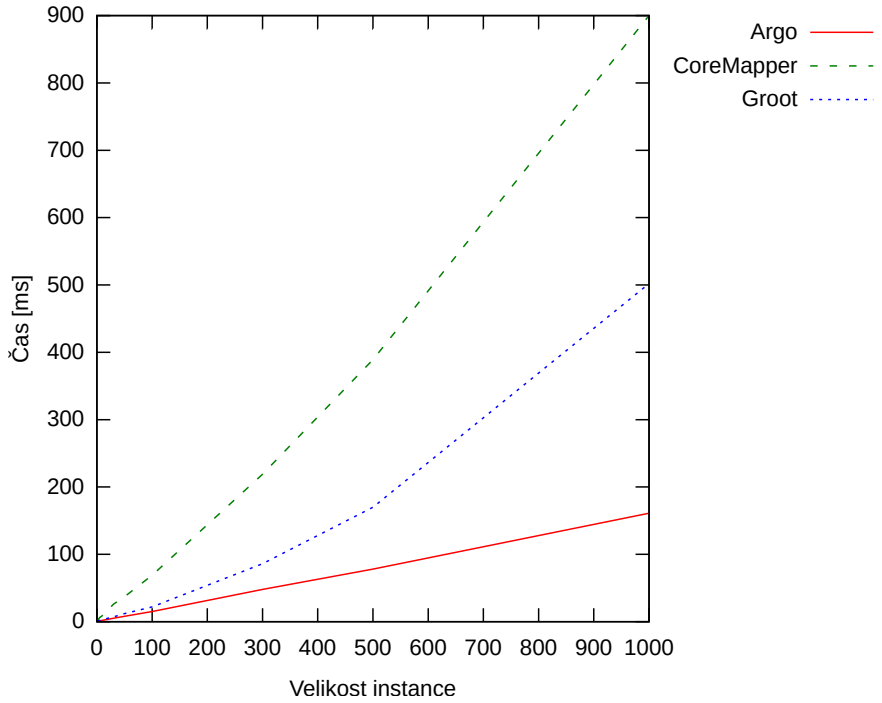


Obrázek 4.2: Graf výpočetního času knihoven – data bez vazby

Z výsledků je patrné, že CoreMapper je pomalejší, než knihovna Groot. Jedná se o očekávaný jev, protože CoreMapper se skládá ze dvou knihoven a prochází tedy dvěma fázemi. Závisí na rychlosti knihovny Argo a CoreValue. Z grafu lze snadno vypozorovat, že Argo knihovna nemá takový vliv na výkon,

Tabulka 4.2: Výsledky měření výpočetního času knihoven – data s vazbou

Velikost instance	Argo [ms]	CoreMapper [ms]	Groot [ms]
1	0	2	1
5	1	6	2
10	2	9	3
30	5	26	7
50	8	37	12
100	15	69	22
300	48	219	86
500	78	398	170
1000	161	900	502



Obrázek 4.3: Graf výpočetního času knihoven – data s vazbou

ale že velký podíl na pomalém chodu CoreMapper má knihovna CoreValue. Mělo by se tedy zaměřit na optimalizaci knihovny CoreValue, zvláště u dat bez vazby.

Z měření lze vypozorovat i to, že Argo je lehce pomalejší nežli Groot v případě dat bez vazby. Důvod je pravděpodobně ten, že Groot neřeší podrobnou příčinu chyby tak jako Argo.

Groot při nenalezení klíče nemapuje vlastnosti objektu i když je vlastnost

povinná. Argo i CoreValue tento problém řeší a vypíší chybu. CoreValue také zabaluje objekty programovacího jazyka Swift do Core Data objektů, zatímco Groot pouze přiřazuje hodnoty. To je příčinou proč je CoreMapper znatelně pomalejší, než Groot řešení.

Z měření a návrhu tedy vyplívá, že CoreMapper je pomalejší nežli Groot knihovna, ovšem nabízí jednodušší a srozumitelnější řešení s možností plné přizpůsobitelnosti. Také obsahuje lepší řešení pro modelové třídy Swiftu a ošetření povinných i volitelných hodnot.

4.3 Práce se soubory

Práce se soubory vyžaduje specifické řešení a obtížně se navrhuje přístup, který použít při neznalosti konkrétního problému. Proto jsem se zde rozhodl navrhnout možné řešení, které lze zvolit při ukládání dat do souboru.

Perzistence dat zajištěná pomocí ukládání dat do souboru na disk, je popsána v kapitole Technologie v části 2.4.1. Jsou zde popsány možnosti ukládání na disk, kam přesně by se data měla ukládat a k čemu jsou určeny jednotlivé adresáře.

K samotnému návrhu přístupu v případě zápisu do souboru. Jak jsem již napsal, záleží na konkrétních datech, přístupu a implementaci modelových tříd a tak se zde zaměřím na popis případů, které mohou nastat.

Pokud se pracuje s malými daty a nebo s jednotlivými hodnotami je na místě použít stejný postup jako je popsán v části 4.1. Sice se nemusí použít Property list, ale data lze jednoduše serializuje ze slovníku do souboru podle vlastního uvážení.

V případě použití modelů, je nejvýhodnější přístup zvolit knihovnu, která toto řeší. Na základě analýzy je nejlepším řešením knihovna Argo a případně knihovna Orga pro serializaci do JSON. Obě tyto knihovny mají stejnou syntaxi a použití. Argo nabízí řešení, které je dostatečné pro mapování jednotlivých hodnot do modelových tříd a z modelových tříd se může implementovat funkcionalita, která serializuje objekt do souboru. Opět díky možnosti jednoduché přizpůsobitelnosti se dá model upravit tak, aby vyhovoval návrhu lokálních struktur.

4.4 Shrnutí

V této kapitole jsem se zaměřil na návrh a realizaci pro řešení problému serializace a deserializace JSON dokumentu. Kapitola je rozdělena na tři části, podle struktury dat a přístupu který se použije.

V první části jsem sa zaměřil na jednotlivé hodnoty. Zhodnotil jsem dva různé přístupy a došel jsem k závěru, že nejvýhodnější je použít knihovnu pro syntaktickou analýzu SwiftyJSON. V případě, že je požadavkem použít

modelové třídy, zvolil jsem knihovnu Argo, která splňuje dostatečně řešení tohoto problému.

V další části jsem se již zabýval modely a databází. Tato část je nejzajímavější, protože obsahuje komplexní řešení. Využívá databázi a modelové třídy programovacího jazyka Swift. Představil jsem tři možné přístupy za použití knihoven. Jelikož jsem shledal všechny tři přístupy nedostatečné, navrhl jsem a realizoval řešení, které jsem nazval CoreMapper. V tomto řešení jsem využil výhody Argo a CoreValue knihoven a implementoval potřebné funkce pro jejich propojení. Výsledkem je řešení, které dovoluje libovolnou přizpůsobitelnost a přitom je jednoduché k použití. Následně jsem

Poslední část se zabývá možným řešením při zápisu dat do souboru. Zamyslel jsem se nad obecným řešením, které by mělo vyhovovat ve většině případů. Tímto řešením je využití knihovny Argo a modelových tříd jazyka Swift a implementace serializace dat do souboru v každé modelové třídě.

Kapitola 5

Testování

Testování by mělo být nedílnou součástí každého projektu. Pomůže nalézt chyby a ověřit, že daná aplikace nebo knihovna je v pořádku a splňuje požadavky. Díky testování se zkoumá kvalitu softwaru a jeho funkcionalita.

Ověřování kvality softwaru se provádí různými způsoby. Testy se rozdělují do kategorií podle toho co se testuje a jak. Každý software se testuje jinak. U aplikací se testuje nejen funkčnost, ale i uživatelské rozhraní. Oproti tomu u knihoven a modulů, uživatelské rozhraní testovat nemá smyl. Proto je důležité správně zvolit případy a testy podle projektu.

Tato kapitola se zaměřuje na testování knihoven a řešení, které jsem v této diplomové práci uvedl. Zabývá se otázkou jak by se správně mělo zajistit ověření funkčnosti a jaké testy zvolit. Nezapomenout na okrajové podmínky, ale testovat i obecnou funkcionalitu a splnění požadavků. Také uvádí jak otestovat projekt, ve kterém se používá jedno z řešení pro serializaci a deserializaci JSON.

Kapitolu jsem rozdělil na tři části. V první části pojednávám o metodě *FURPS* a jak ji využít pro testování knihoven. Druhou část věnuji unit testování. Jak se testují jednotlivé funkce a na co by se nemělo zapomenout. V poslední části se zaměřuji na systémové testování. Tedy testování systému jako celku.

5.1 Požadavky na testování

Pro ověření kvality řešení doporučuji se držet metody FURPS. Následováním této metody pro ověření kvality softwaru, by měla být výsledná knihovna dostatečně otestovaná a zvýší se tím i její kvalita. Rozhodl jsem se každou oblast této metody popsat z hlediska testování knihovny pro serializaci a deserializaci JSON. Následuje výčet oblastí metody FURPS a popis na co je potřeba se v rámci každé oblasti zaměřit.

Funkčnost (Functionality) Ověření jestli se knihovna správně chová a spl-

ňuje požadovanou funkčnost. Zpracovat unit testy pro ověření funkčnosti jednotlivých funkcí a ověřit funkčnost celku. Ověřit zda se správně namapoval objekt, jestli knihovna správně převede identifikátor, atd.

Použitelnost (Usability) Určuje zda je knihovna snadno použitelná a přívětivá. Je potřeba ověřit jestli se vývojář při používání knihovny neztratí a pochopí jak se používá. Je nutné správně volit názvy metod a zdokumentovat knihovnu. Zhotovit návod pro instalaci, první použití a podobně. Knihovna by měla být průhledná. Tedy je potřeba zajistit, že vývojář, který knihovnu používá, ví co se děje a všemu rozumí.

Spolehlivost (Reliability) Říká že když dojde k chybě, tak knihovna dokáže chybu ošetřit a vrátit chybový stav. Nespadne při přetížení. Musí se ověřit zda se knihovna nebude chovat jinak při stejných datech, vždy vypíše správný důvod chyby, apod.

Výkon (Performance) Zkontrolovat zda výpočet netrvá příliš dlouho a nezatěžuje zbytečně procesor. Pokud je struktura JSON dokumentu složitá, ověřit jak dlouho výpočet potrvá.

Podporovatelnost (Supportability) Knihovna dokáže pracovat s knihovnami pro práci s databází, zvládá práci s vlákny, atd. Nabízí podporu standardních knihoven, které se používají při vývoji v programovacím jazyce Swift. Funguje správně na všech potřebných platformách a podobně.

Pokud se testuje podle výše zmíněných oblastí a splní se požadavky, měla by kvalita knihovny být dostatečná.

5.2 Unit testování

Pro testování funkčnosti jednotlivých částí se používají unit testy. Díky těmto testům lze ověřit funkčnost jednotlivých částí a přístupů.

V jazyce Swift se nejčastěji pro unit testování používá *XCTest*. Jedná se o framework, který nabízí API pro testování jednotkových funkcí, ale nabízí i například měření času. Lze použít i knihovnu třetí strany *Quick*. Autor knihovny Quick se inspiroval knihovnou *RSpec*, používanou v projektech napsaných v jazyce Ruby. Využívá tedy stejného zápisu a struktury testů. Quick přichází spolu s knihovnou *Nimble*, která řeší kontrolu shody mezi očekávaným výsledek a výsledkem skutečným.

Při analýze knihoven jsem zjistil, že všechny knihovny používají pro testování XCTest. Některé knihovny ovšem testování vynechaly úplně. Ze zde analyzovaných se jedná o Swift-json.

Pokud se rozhodne napsat unit testy, je potřeba nejdříve provést analýzu a sestavit seznam případů užití. Pro každý případ užití následně zhotovit testy.

V rámci návhu CoreMapper jsem implementoval unit testy pro ověření, jestli funguje vše správně. Protože jsou použity zde knihovny, které jsou už otestovány, není potřeba testy provádět znovu. Napsal jsem tedy jen testy, které jsou důležité pro ověření, že řešení funguje správně. Mezi tyto testy patří ověření, zda vstupní data v JSON formátu, se skutečně uloží do databáze. Testy jsem zhotovil za použití knihoven Quick a Nimble.

Pro testování podobného případu je potřebné, jako první krok, získat JSON dokument. Je na výběr ze dvou možností. První možností je uložit data pro testování do souboru a před testováním soubor nahrát. Záleží na každém testu, ale často je potřeba takovýchto souborů více. Může se jednat i o případ, kdy pro každý test je nutný jeden soubor. Další možností je vytvořit například třídu, která obsahuje data ve slovníku tak, jak bychom je získali z API. Data jsou uložena přímo v kódu. Ty lze následně použít u jednotlivých testů. Může se jednat o proměnné, nebo funkce, které proměnné vygenerují. V projektu CoreMapper jsem použil oba tyto přístupy. Data pro ukázku jsou uložena v JSON souboru a pro testování se používají data uložena v proměnných.

Unit testy by u podobných řešení měly ověřovat, zda se data opravdu nacházejí v modelové třídě. Jestli při mapování špatného typu vrátí proces chybu. Pokud se mapuje volitenlá vlastnost objektu a na vstupu je prázdná hodnota, je nutné ověřit, že je ve výsledku ve vlastnosti objektu uložena prázdná hodnota. Otestovat obecně, jestli se správně mapují hodnoty do vlastností modelu. Ověřit, zda se data správně uloží do perzistentního úložiště a podobně.

Pokud je projekt ve kterém se používá některé řešení pro serializaci a deserializaci JSON, není nutné psát testy pro dané řešení, pokud toto řešení již bylo otestováno. Na co se ovšem nesmí zapomenou je ověření projektového nastavení a případné úpravy provedené v projektu.

5.3 Systémové testování

Systémové testy ověřují software jako celek. Lze zde zařadit více druhů testů a přístupů pro ověření zda software splňuje požadavky. Systémové testování se může automatizovat a nebo provádět manuálně.

Do této části jsem zařadil testy pro ověření splnění požadavků. Zodpovídající otázku zda opravdu řešení nabízí danou funkcionalitu a podobně. Zda při integraci knihovny do projektu vše funguje a lze použít v běžných projektech. Otestovat zda se knihovna chová správně při použití jiných knihoven, které se často používají. Jedná se tedy hlavně o otázky použitelnosti, spolehlivosti, výkonu a podporovatelnosti.

V případě knihovny se jedná o testy, které lze obtížně automatizovat. Proto se musí sestavit plán a opět případy užití, podle kterých se knihovna otestuje. Tyto testy by se měli provádět pravidelně při každé aktualizaci a nebo minimálně při každé nové verzi. Integrační testy, pro otestování zda knihovna funguje s použitím dalších knihoven a podobně, se provádí složitě. Proto není

na místě testovat každou malou změnu, nejlépe však při každé nové verzi knihovny.

Pro ověření výkonu lze využít nástroje vývojového prostředí Xcode – *Instruments*. Tento nástroj nabízí funkce pro sledování zatížení procesoru, práce s databází nebo soubory a využití paměti zařízení. Druhou možností je použít XCTest framework, který nabízí možnost testování výkonu. Tuto možnost jsem ostatně použil v případě měření výkonu CoreMapper, Argo a Groot knihovny v kapitole Návrh a Realizace (viz 4.2.3).

Projekty používající dané řešení musí opět otestovat zda použitá knihovna funguje správně v daném případě. Také zda splňuje požadavky a skutečně je výhodná pro daný projekt.

Při analýze knihoven jsem testoval jednotlivé funkce. Knihovny jsem testoval na příkladových aplikacích a zkoušel jsem hlavně jak se vypořádají s danými problémy. Vše jsem prováděl manuálně. Při návrhu a implementaci jsem opět zkoušel jednotlivé případy a testoval funkčnosti. Vlastní řešení jsem doplnil o unit testy, které se provádějí automaticky. Ostatní testy jednoduše zautomatizovat nelze. Proto kromě unit testů je vhodné řešení manuálně ověřit zda knihovna vyhovuje, ať už pomocí nástroje Instruments a nebo na projektech, pro tento účel speciálně vytvořených.

Závěr

Cílem diplomové práce bylo analyzovat existující řešení serializačních a deserializačních technik pro moderní webové API ve formátu JSON do modelových tříd v jazyce Swift určených k prezistenci. Po provedení analýzy navrhnout a realizovat řešení, které dostatečně řeší výše uvedený cíl diplomové práce.

Cíl diplomové práce byl splněn v plném rozsahu. Provedl jsem úspěšnou analýzu a navrhl a realizoval řešení pro různé přístupy.

Nejprve jsem představil a popsal programovací jazyk Swift a formát JSON. Představil jsem i moderní webové API REST a možnosti řešení perzistentního úložiště v jazyce Swift. Díky perzistentnímu úložišti jsem došel k závěru, že je potřeba rozdělit problém na tři části. První částí je ukládání dat do NSUserDefaults, popřípadě Property list. Druhou částí je databáze SQLite a framework Core Data a třetí část je práce se soubory.

Analyzoval jsem nejdůležitější existující řešení. Zhodnotil jsem je, vyjmenoval jejich výhody a nevýhody a rozdělil do tří kategorií podle toho, na co se zaměřují a v rámci těchto kategorií porovnal.

Následně jsem v další kapitole vyhodnotil, že přístup k návrhu řešení je nejlepší rozdělit na tři různé přístupy, podle situace. První z nich se zabývá malými daty, kdy pracujeme pouze s hodnotami. Zde jsem vyhodnotil, že existující řešení jsou dostatečná a doporučil jsem, jak za daných situací postupovat. Druhý přístup se zabývá modely, kdy máme struktury dat a ukládáme je do databáze. Zde jsem vyhodnotil existující řešení a shledal je v určitých případech jako nedostatečná. Proto jsem navrhl řešení vlastní, za použití dvou existujících knihoven. Toto řešení jsem implementoval a porovnal s ostatními knihovnami a zhodnotil jsem výkon. V poslední části jsem doporučil jaké východisko zvolit při práci se soubory.

V poslední kapitole jsem se zaměřil na testování. Doporučil jsem přístupy pro návrh a správné provedení testů knihoven a řešení pro serializaci JSON v programovacím jazyce Swift.

Podařilo se mi zjistit, že existují přístupy, které problém řeší částečně a nebo nedostatečně. Některé knihovny se zaměřují pouze na malou část a jiná

řeší komplexní přístup. Shledal jsem dostatečně vyhovující knihovnu Argo, která se zabývá importováním JSON dat do modelových tříd jazyka Swift. Tuto knihovnu jsem ostatně i použil ve své vlastní implementaci.

Firma Apple se o jazyk Swift stará a nyní pracuje na jeho novější verzi 3.0. Předpokládám, že přijde s řešením problému této diplomové práce. Můj názor je, že nejdůležitějším krokem v řešení tohoto problému je aktualizovat zastaralý ORM framework Core Data pro moderní jazyk Swift a představit řešení pro serializaci a deserializaci dat v JSON formátu. Bohužel k tomu zatím nedošlo a proto je potřeba k tomuto problému přistupovat tak, jak jsem představil v této práci.

Největší problém shledávám v nedostatečné informovanosti vývojářů. Je mnoho řešení, ale některá obsahují nedostatky a jiné řeší jen část problému.

Závěrem dodávám, že realizované řešení je funkční a v aktuální formě se dá aplikovat na reálný projekt. Rád bych se v budoucnosti zaměřil na vytvoření návodu pro použití, kde seznámím uživatele, jak knihovnu použít a jak s ní pracovat. Také se chci zaměřit na optimalizaci výkonu knihovny CoreValue, kterou používám v řešení CoreMapper.

Literatura

[1] Apple Inc.: File System Basics [online]. 2015, [cit. 2016-03-13]. Dostupné z: https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html

[2] Apple Inc.: Swift Has Reached 1.0 [online]. 2014, [cit. 2016-03-10]. Dostupné z: https://developer.apple.com/swift/blog/?id=14

[3] Apple Inc.: Swift - Platform Support [online]. 2016, [cit. 2016-03-11]. Dostupné z: https://swift.org/about/#platform-support

[4] Apple Inc.: Swift - Features [online]. 2016, [cit. 2016-03-11]. Dostupné z: https://swift.org/about/#features

[5] Apple Inc.: Swift is Open Source [online]. 2015, [cit. 2016-03-10]. Dostupné z: https://developer.apple.com/swift/blog/?id=34

[6] Apple Inc.: Swift and Objective-C in the Same Project [online]. 2016, [cit. 2016-03-23]. Dostupné z: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html

[7] Apple Inc.: Swift - Modern [online]. 2016, [cit. 2016-03-11]. Dostupné z: https://developer.apple.com/swift/

[8] 1&1 Internet, Inc.: Introducing JSON [online]. 20-?, [cit. 2016-03-11]. Dostupné z: http://json.org

[9] Refsnes Data: XML Schema Tutorial - What is an XML Schema? [online]. 1999, [cit. 2016-03-23]. Dostupné z: http://www.w3schools.com/xml/schema_intro.asp

[10] Zyp, K.: The home of JSON Schema [online]. 2015, [cit. 2016-03-23]. Dostupné z: http://json-schema.org

[11] Apple Inc.: The Foundation Framework [online]. 2013, [cit. 2016-03-06]. Dostupné z: https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/ObjC_classic/index.html

[12] Apple Inc.: NSJSONSerialization Class Reference [online]. 2014, [cit. 2016-03-06]. Dostupné z: https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSJSONSerialization_Class/index.html

[13] doc. Ing. Tomáš Vitvar, P.: Lecture 8: SOAP and REST - REST [online]. 2015, [cit. 2016-03-08]. Dostupné z: http://humla.vitvar.com/slides/mdw/lecture8.html#!/18/v1

[14] Kosek, J.: Inteligentní podpora navigace na WWW s využitím XML - Využití webových služeb a protokolu SOAP při komunikaci [online]. 2012, [cit. 2016-03-23]. Dostupné z: http://www.kosek.cz/diplomka/html/websluzby.html

[15] Alex Rodriguez, IBM: RESTful Web services: The basics [online]. 2015, [cit. 2016-03-08]. Dostupné z: https://www.ibm.com/developerworks/webservices/library/ws-restful/

[16] Apple Inc.: NSFileManager Class Reference [online]. 2015, [cit. 2016-03-13]. Dostupné z: https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSFileManager_Class/index.html

[17] Apple Inc.: NSUserDefaults Class Reference [online]. 2014, [cit. 2016-03-13]. Dostupné z: https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSUserDefaults_Class/index.html

[18] Apple Inc.: About Property Lists [online]. 2010, [cit. 2016-04-08]. Dostupné z: https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html

[19] Hipp, R.: About SQLite [online]. [cit. 2016-03-18]. Dostupné z: https://www.sqlite.org/about.html

[20] Tutorialspoint: SQLite Limitations [online]. 2016, [cit. 2016-04-10]. Dostupné z: http://www.tutorialspoint.com/sqlite/sqlite_overview.htm

[21] Apple Inc.: What Is Core Data? [online]. 2015, [cit. 2016-03-18]. Dostupné z: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreData/index.html

[22] Realm: Realm is a replacement for SQLite & Core Data [online]. 2016, [cit. 2016-04-10]. Dostupné z: https://realm.io

[23] Realm: Realm - Fast [online]. 2016, [cit. 2016-04-10]. Dostupné z: https://realm.io/news/introducing-realm/#fast

[24] Fu, R.: SwiftyJSON [online]. 2016, [cit. 2016-03-29]. Dostupné z: https://github.com/SwiftyJSON/SwiftyJSON

[25] Kogai, D.: Swift-json [online]. 2016, [cit. 2016-03-30]. Dostupné z: https://github.com/dankogai/swift-json

[26] Kellaway, H.: Gloss - Why Gloss? [online]. [cit. 2016-03-30]. Dostupné z: http://harlankellaway.com/Gloss/

[27] Thoughbot, Inc.: Argo [online]. 2016, [cit. 2016-03-31]. Dostupné z: https://github.com/thoughtbot/Argo

[28] Edwards, C.: Ogra [online]. 2015, [cit. 2016-03-31]. Dostupné z: https://github.com/edwardaux/Ogra

[29] Lund, J.: Decodable [online]. 2016, [cit. 2016-04-04]. Dostupné z: https://github.com/Anviking/Decodable

[30] Gonzalez, G.: Groot [online]. 2015, [cit. 2016-03-31]. Dostupné z: https://github.com/gonzalezreal/Groot

[31] Hyper: Sync [online]. 2016, [cit. 2016-04-06]. Dostupné z: https://github.com/hyperoslo/Sync

[32] Alamofire Software Foundation: Alamofire [online]. 2016, [cit. 2016-04-06]. Dostupné z: https://github.com/Alamofire/Alamofire

[33] Magical Panda Software: MagicalRecord [online]. 2016, [cit. 2016-04-06]. Dostupné z: https://github.com/magicalpanda/MagicalRecord

[34] Terhechte, B.: CoreValue [online]. 2016, [cit. 2016-04-11]. Dostupné z: https://github.com/terhechte/CoreValue

Příloha **A**

Seznam použitých zkratek

ACID Atomic, Consistent, Isolated, Durable

API Application programming interface

HTTP Hypertext transfer protocol

JSON Javascript object nation

LTE Long Term Evolution

ORM Objektové relační mapování

REST Representational state transfer

SQL Structured Query Language

XML Extensible markup language

Příloha B

Instalační příručka

B.1 CoreMapper

Zdrojový kód implementace jednoho z řešení je na přiloženém CD. Ke zkompilování zdrojového kódu je potřeba mít nainstalované vývojové prostředí Xcode. Projekt otevřete pomocí souboru `CoreMapper.xcworkspace`.

Příloha C

Obsah přiloženého CD

Zdrojové kódy implementace jsou součástí projektu vývojového prostředí Xcode. Tento projekt obsahuje unit testy i testy výkonu.

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace CoreMapper s testy
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `json_parser_example.playground` ..... příklad zpracování JSON dat v programovacím jazyce Swift
- `text` ..... text práce
  - `DP_Vlasak_Jakub_2016.pdf` ..... text práce ve formátu PDF