Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Návrh a implementace mobilní aplikace pro chytrý vodoměr

Lukáš Šanda

Bakalářská práce
2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2017/2018**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Lukáš Šanda**
Osobní číslo:	**I15351**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Návrh a implementace mobilní aplikace pro chytrý vodoměr**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Cílem práce je návrh a implementaci mobilní aplikace na platformě iOS pro správu a vyhodnocení dat o spotřebě vody chytrého vodoměru.
Teoretická část bakalářské práce bude věnována mobilním platformám a aplikacím se zaměřením na jejich využití v chytrých domácnostech.
V praktické části se student zaměří na vlastní návrh a implementaci mobilní aplikace pro platformu iOS, která bude přes REST API umožnovat základní správu chytrého vodoměru a dále posykovat grafické přehledy o vývoji spotřeby vody.
Výsledná aplikace bude testována v reálném provozu

Rozsah grafických prací:

Rozsah pracovní zprávy: **30**

Forma zpracování bakalářské práce: **tištěná**

Seznam odborné literatury:

Molly Maskrey. Beginning iPhone development with Swift 3: exploring the iOS SDK. New York, NY: Springer Science+Business Media, 2016. ISBN 978-1484222225.
NEUBURG, Matt. IOS 10 programming fundamentals with Swift: Swift, Xcode, and Cocoa Basics. Third edition. Sebastopol, CA: O'Reilly Media, 2016. ISBN 1491970073.

Vedoucí bakalářské práce: **Ing. Jan Fikejz, Ph.D.**
Katedra softwarových technologií

Datum zadání bakalářské práce: **31. října 2017**

Termín odevzdání bakalářské práce: **12. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

Ing. Lukáš Čegan, Ph.D.
pověřený vedením katedry

V Pardubicích dne 20. března 2018

Prohlašuji:
Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 17. 1. 2018

Lukáš Šanda

PODĚKOVÁNÍ

Rád bych poděkoval panu Ing. Janu Fikejzovi, Ph.D. za vedení práce, cenné rady a věcné připomínky k aplikaci z pohledu budoucího uživatele. Děkuji také panu Ing. Petru Koulovi za implementování REST API a konzultace při implementaci komunikace s databází.

ANOTACE

Cílem práce je provést návrh a implementaci mobilní aplikaci, pro operační systém iOS v programovacím jazyce Swift, pro správu a vizualizaci dat o spotřebě vody chytrého vodoměru. Pomocí REST API je aplikaci umožněna základní správa chytrého vodoměru a jsou poskytnuta data pro grafické přehledy o spotřebě vody, která přísluší danému vodoměru přihlášeného uživatele. Přihlášení uživatele do aplikace je též realizováno pomocí REST API. Veškeré požadavky na databázi jsou šifrovány pomocí JSON Web Token pro zachování bezpečnosti. Uživateli je umožněno editovat některé hodnoty vodoměru, jež přísluší danému uživateli. Je též možné, aby uživatel změnil stav vodoměru – uzavřený, otevřený. Aplikace bude testována na reálném zařízení.

KLÍČOVÁ SLOVA

Mobilní aplikace, platforma iOS, jazyk Swift, REST API, chytrý vodoměr.

TITLE

Design and implementation of a mobile application for a smart water meter.

ANNOTATION

The aim is to make a proposal and implementation of a mobile application, for operating system iOS in programming language Swift, for management and data visualization about water consumption of smart water meter. Via REST API, application is capable to manage a smart water meter and data for graphical overviews about water consumption. User authentication is realized via REST API also. All requests on database are encrypted with JSON Web Token for security. The user is allowed to edit some of the water meter parameters. It is also possible for the user to change the water meter status - closed, open. Application is going to be tested on real device.

KEYWORDS

Mobile application, iOS platform, programming language Swift, smart water meter.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM ZKRATEK

API	Application Programming Interface
CRUD	Create Read Update Delete
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
UI	User Interface
URI	Uniform Resource Identifier
XML	Extensible Markup Language

SEZNAM UKÁZEK KÓDU

ÚVOD

V dnešní době není raritou vlastnit chytrý mobilní telefon s osobní asistentkou napojenou na chytrou domácnost. Obecnějším příkladem jsou různé chytré senzory. Pokud je takový senzor spojený s termostatem v bytě, stačí vyslat signál pomocí mobilní zprávy či speciální služby a on spustí ohřev na požadovanou teplotu. U těchto úkonů se nejedná o nijak komplexní procesy, spíše jde o jednoduché akce. Budoucí vize však sahají výrazně dál, a to nejen v domácnostech, ale i v průmyslu. V továrnách jsou již běžně nasazeny zařízení, která provádí automatizované úkony bez nutnosti většího dohledu pověřenou osobou. Stále však převahuje lidský faktor, který je v určitých situacích nepostradatelný. Představy jsou však takové, že budou ve firmě fungovat samostatně, bez nutnosti dohledu, budou samy hlásit vzniklé poruchy a případně i zkontaktují servisní službu. Taková zařízení jsou požadována i pro běžného spotřebitele a jeho domácnost. Jak je již zmíněno výše, můžeme si kombinací mobilního telefonu a chytrého senzoru například regulovat teplotu nebo průtok vody. Velkou výhodou je ono mobilní zařízení. Již existují aplikace, které spolu s regulací umožňují vizualizaci naměřených nebo získaných dat. Tím je spotřebiteli, skrze grafické rozhraní, zlehčeno hospodaření a kontrola nad jeho domácností. Další výhodu, kterou nese mobilní zařízení, je absence přítomnosti z bezprostřední blízkosti domu. Veškeré akce lze provádět kdekoliv. Jedinou nutností je samozřejmě připojení k internetu, neboť zmíněné aplikace pro komunikaci se senzory často používají webové aplikační rozhraní (API), vůči kterému je nejprve uživatel ověřen a poté probíhá požadovaná komunikace. Získaná data ze senzoru jsou uchovávána v databázi. API, na základě požadavku aplikace, tato data požadovaným způsobem vyfiltruje a jsou aplikaci vráceny jako odpověď na požadavek. Většinou jsou navráceny ve formátu JSON či XML. Ať se jedná o správu či kontrolu, lze tak obecně mluvit o monitorování.

Pojmy, které jsou s touto oblastí velmi úzce spojeny, jsou „Internet věcí“ a „Průmysl 4.0“. První část této práce je zaměřena na jejich definování, současný stav, vize a jaký je směr jejich vývoje. Dále je popsán zmíněný chytrý senzor pro vodoměru, který byl využit i v druhé části této práce.

Druhá část je zaměřena na vývoj mobilní aplikace odrážející využití IoT zařízení – chytrého vodoměru. Mobilní aplikace pomocí aplikačního rozhraní umožňuje základní správu chytrého vodoměru a slouží jako uživatelská nadstavba mezi vodoměrem a uživatelem, kterému jsou data z vodoměru vizualizována. Vývoj byl zrealizován v programovacím jazyce Swift a je určena pouze pro zařízení Apple s operačním systémem iOS.

1 ČTVRTÁ PRŮMYSLOVÁ REVOLUCE

1.1 Internet věcí

Ve zkratce řečeno je „Internet věcí“, neboli Internet of Things, vše, co je připojeno k internetu. Pavel Pohanka definuje IoT jako: *„síť propojených objektů (věcí), které jsou jednoznačně adresovatelné s tím, že tato síť je založena na standardizovaných komunikačních protokolech umožňující výměnu a sdílení dat a informací, jejichž analýzou bude možné docílit vyšší přidané hodnoty.“*. Zahrnuje zařízení od obyčejných senzorů až po mobilní telefony nebo chytré náramky. Kombinací těchto zařízení, s automatizovanými systémy, je možné získat důležitá data, analyzovat je a zpětně vyvolat akci na základě těchto dat. Tato akce může být jednoduchou či posloupností několika operací. Může se jednat o zatažení žaluzií či zjištění zásob v lednici a, při nedostatku, jejich dokoupení, díky komunikaci v uzavřené síti mezi IoT zařízeními.

Poprvé byl pojem „Internet věcí“ použit v prezentaci, kterou publikoval Kevin Ashton, v roce 1999. Ten v ní uvádí, že „lepší vnímání světa“ bychom docílili propojením senzorů a sdílením dat mezi určitými systémy. Jedním z nejdůležitějších milníků pro IoT byly roky 2008 a 2009. V této době, dle odhadu společnosti Cisco, překročil počet zařízení připojených k internetu, počet světové populace. Na základě této skutečnosti je k zmíněným datům datován vznik IoT.

V roce 2014 článek „Harvard Business Review“, který publikoval Michale Porter a James Heppelmann, popisuje IoT zařízení jako inteligentní, do internetu připojené produkty, které splňují tři základní podmínky:

1. Fyzické komponenty zahrnující mechanické a elektrické části,
2. „Inteligentní“ komponenty obsahují senzory, mikroprocesory, datové uložiště, software, operační systém a rozšířené uživatelské rozhraní,
3. Komponenty konektivity zahrnují porty, anténu a protokoly, které umožňují bezdrátové či kabelové propojení s produktem.

To, co dělá zařízení „inteligentním“, jsou senzory a mikroprocesory, které umožňují pokročilé chování. Například bezpečnostní kamery u domů, které reagují na pohyb nebo náramky, které upozorní osobu trpící úplavicí cukrovou na nízkou hladinu cukru v krvi. Na níže uvedeném obrázku, Obrázek 1 – Vizualizace IoT, je vyobrazena ilustrace různých zařízení, které jsou označována jako zařízení IoT. Od mobilních telefonů, přes herní konzole, až po

domácnosti. Všechna tato zařízení spojuje jedno – na základě interakce od spotřebitele jsou shromažďována data a jejich analýzou je docíleno k zefektivnění služeb.

Zařízení jsou k internetu připojeny z různých důvodů: zjišťování polohy v reálném čase, pro navigační systém v automobilu, kontaktování bezpečnostní agentury v případě nezákonného vniknutí na soukromí pozemek nebo zmíněný náramek, který ukládá data o zdravotním stavu, která mohou být později uživatelem zobrazena a případně vizualizována či v případě nutnosti předána lékaři.

Obrázek 1 – Vizualizace IoT, zdroj: [9]

Tato kapitola čerpala ze zdrojů: [8] [9] [10] [13]

1.1.1 Sdílení dat IoT zařízení

Dnešní doba zažívá spoustu událostí spojených s únikem či krádeží informací, uživatelé internetu jsou neustále pod rizikem útoku hackera nebo jsou před tímto činem varováni ze všech stran. Při každé příležitosti, kdy webová stránka, aplikace nebo služba žádá o přístup k informacím, uživatel bez rozmyšlení tento požadavek zamítá. Pro zařízení IoT je sdílení dat klíčové takřka nutné. Umožňují vládě, veřejným i soukromým organizacím, na základě sesbíraných informacích, znovu přemýšlet a případně předělat business plány.

Nasbírané Informace jsou klíčová pro specifické důvody, které mohou být užitečné například pro dopad na ekonomiku. Jedna ze studií odhaduje, že 35 % amerických výrobců využívá data z chytrých snímačů již v rámci plánování produkce. *„Internet věcí nabízí příležitost být více efektivní, šetří čas, peníze a emise v procesech.“*, říká Evans Gorski o

Internetu věcí. Americká firma, Concrete Sensors, vyvinula zařízení, které při vložení do betonu ukládá informace o stavu betonu – jeho síle, teplotě či relativní vlhkosti.

Tato kapitola čerpala ze zdrojů: [9] [10] [13] [27]

1.1.2 Internet věcí v současnosti

V dnešní době jsou IoT zařízení stále na svém počátku. V současné době IoT zařízení v chytré domácnosti dokáže:

- Spustit alarm v případě požáru,
- aktivovat úniková světelná značení,
- zkontaktovat požární službu,
- spustit požární systém.

Vize do budoucnosti jsou takové, že chytré domácnosti by umožňovaly:

- Detekovat požár, spustit alarm či dokonce předejít celému incidentu. Například problém se zásuvkami, odložená žehlička či nahromaděné toxické výpary v uzavřeném prostoru,
- vypnout elektrické, větrací a plynové systémy, které by jinak mohly pomáhat v rozšíření ohně,
- přenášet schéma stavby na heads-up displej v helmě požárníka, který obsahuje informace o místech se silným ohniskem, okolních podmínkách a strukturálním poškození či rozmístění osob v perimetru,
- zasílat informace o zdravotním stavu obětí
- přesměrovat civilní dopravu mimo místo vzniku nehody, pro zajištění dostupnosti záchranným službám, a volnou cestu do nejbližší nemocnice.

Výčet těchto možností je pouze ilustrativní a demonstruje, jakým směrem by se mohl svět IoT zařízení ubírat. Tato zařízení umožní kvalitnější a pohotovější rozhodování, která jsou založena na skutečných datech, nikoli na intuici.

Tato kapitola čerpala ze zdrojů: [8] [10] [13]

1.1.3 Důvod existence IoT

„Cílem IoT je propojení zařízení, systémů a služeb za účelem poskytnutí více dat, která mohou být převedena na informace a informace na znalosti, které lze následně aplikovat“, uvádí Pavel

Pohanka ve svém článku [8] o IoT. Taková zařízení pak mohou, na základě získaných znalostí, provádět adekvátní rozhodnutí.

Jedná se o jednoduchou rovnici – čím schopněji budou zařízení moci poskytovat data o okolním světě, tím bude více dat k analyzování a jejich využití. Lze tak mluvit o určité evoluci internetu a odbourání potřeb náročně manipulovat s určitým zařízením. Takto bude vše dostupné odkudkoliv a přístupné i okolním zařízení v uzavřené síti.

Zdroj: [8][10][13]

1.1.4 Zařízení a aplikace

Pro běžného spotřebitele lze zmínit například chytré spotřebiče, televize, reproduktory nebo hračky. Co se týče veřejných nebo neveřejných podniků, můžeme mluvit o chytrých senzorech, komerčním bezpečnostním systému nebo o technologiích, spojených s „inteligentním městem“ – monitorování dopravy či povětrnostních podmínek. Zařízení, která lze používat kdekoliv, mohou být například klimatizace, termostaty, osvětlení nebo bezpečnostní systém.

Zdroj: [11]

1.1.5 Trendy IoT zařízení a očekávaný růst

Přední světová výzkumná a poradenská společnost Gartner odhaduje, že celkový počet všech IoT zařízení v roce 2017 přesáhl počet 8.4 miliard dolarů, což je o 31 % více, než v roce 2016. Společnost Intel velmi hrubě předpokládá růst z 2 miliard zařízení, z roku 2006, na 200 miliard pro rok 2020. Tento počet by představoval 26 IoT zařízení pro každého člověka na Zemi. Analytická společnost IHS Markit byla v tomto odhadu trochu konzervativnější a uvedla, že počet připojených zařízení bude v roce 2025 až 75,4 miliard a 125 miliard v roce 2030.

Zdroj: [11][12][13]

1.2 Globální síť SIGFOX

Jedná se o světově vedoucího poskytovatele IoT služeb. Jejich hlavní předností je předělání konceptu propojení IoT. Razantně snižuje cenu a spotřebu energie nutnou pro bezpečné spojení senzorů ke cloudu. Momentální životnost zařízení činí 5 až 15 let na baterii. Později však nebude třeba baterie dobíjet, neboť budou sami energii generovat. Poskytované služby jsou kompatibilní s Bluetooth, GPS 2G/3G/4G a Wi-Fi. Další výhodou je pořizovací cena modemů i komunikace. Ta se pohybuje v řádech desítek korun. Díky překonávání cenových i

technických bariér, které do jisté míry rozvoj IoT brzdily, funguje tato síť již ve 45 zemích světa.

Typickým příkladem využití sítě SIGFOX v Evropě jsou aplikace pro odečet vody, elektřiny, plynu, parkovací senzory, Průmysl 4.0 nebo zabezpečení zařízení.

Zdroj: [14]

1.2.1 SimpleCell Networks a.s.

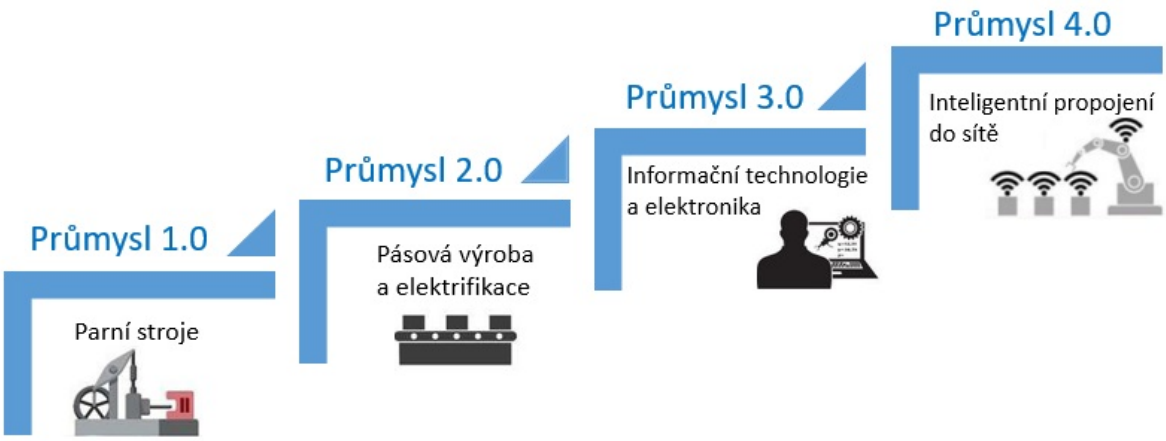
Jedná se o první veřejný český mobilní operátor sítě IoT služeb, který je určen pro IoT zařízení a je založen na technologii SIGFOX. V tuto chvíli je pokryto 95 % populace České Republiky a 92 % jejího území. Již přes 80 firem v České republice vyvíjí zařízení s integrovanou technologií SIGFOX. V druhém kvartálu roku 2018 je v plánu pokrýt 98 % populace a 96 % území České Republiky.

Zdroj: [14][15]

1.3 Průmysl 4.0

Tento pojem reprezentuje 4. průmyslovou revoluci ve výrobě a průmyslu. Jedná se o aktuální trend automatizace průmyslových technologií. Poprvé bylo označení „Průmysl 4.0“ ve městě Hannover v roce 2011 jako název projektu – Industrie 4.0. Zahrnuje kyberfyzikální systémy, Big Data, cloudové služby, roboty a technologie IoT, k realizaci inteligentních, průmyslových a výrobních cílů. Všechny tyto pojmy utvářejí celek a dávají továrně přístavek „inteligentní“ továrna. Toto zařízení, také nazývané „továrna budoucnosti“, je klíčový pojem pro čtvrtou průmyslovou revoluci.

Tento výraz má své kořeny v Německu. Projekt, se zmíněným výrazem „Industrie 4.0“ (odtud Průmysl 4.0), jehož cílem byla digitalizace výroby v organizaci Hannover Messe v roce 2011. Obrázek 2 – Čtvrtá průmyslová revoluce vizualizuje pokrok průmyslu vázaný ke konkrétnímu období průmyslové revoluce.



Obrázek 2 – Čtvrtá průmyslová revoluce, zdroj: [16]

Čtvrtá revoluce je charakterizována jako prolnutí informačních technologií s produkčními procesy způsobem, který je charakteristický pro autonomní stroje. Typická továrna, jež prošla inovací čtvrté revoluce, disponuje silnou robotizací, samostatností strojů, umělou inteligencí a využitím internetu věcí.

Tato kapitola čerpala ze zdrojů: [16][17][18]

1.3.1 Dopad na fungování firem

Z pohledu fungování bude revoluce znamenat ještě komplexnější propojení všech procesů, které jsou spojeny s děním v továrně/firmě od vývoje, výrobu produktu, jeho distribuci až po jeho případný servis. Pro tyto úkony budou mnohem více využívány autonomní roboti, které budou řídit celkový proces samostatně. Tím však dojde k odbourání nutnosti lidského elementu, neboť roboti zastanou práci, kterou dosud vykonávat pouze člověk.

Systémy, které se budou starat o sklad materiálu, budou řídit samostatně situace, kdy bude potřeba doplnit chybějící materiál. V takovém případně odešlou objednávku na základě sesbíraných dat o využívání daného materiálu a případně může objednat určité zboží do zásoby, z důvodu nadměrného používání. Výrobní linky budou modulární a snadno modifikovatelné, takže nebude problém vyrábět laciněji malé série. Podle odhadů bude dopad na efektivitu, produktivitu a časovou úsporu až 30 %.

Údržba robotů či strojů bude na pokročilé úrovni. Porouchaná zařízení budou sama informovat o poruše či o nutnosti servisního zásahu. Nebude se však jednat o signálu vzniklé poruchy, ale o konkrétní závadě.

Zdroj: [16][18]

1.3.2 Dopad na společnost

Negativní dopad bude mít Průmysl 4.0 na trh práce. Na jedné straně jsou autonomní a samostatné zařízení, které ušetří monotónní, fyzicky náročnou a případně i nebezpečnou práci. Na odvrácené straně je obrovský pokles poptávky po dělnících a výrobních profesích. Důsledkem bude přesun výrobních továren, založených na levné lidské práci, z rozvojových zemí zpátky do průmyslově vyspělých zemí. Velkou výhodou bude však velký zájem o osoby znalé tyto technologie.

Zdroj: [16][18]

1.4 Chytré měřiče

Jde o zařízení, které má zaměnit tradiční vodní, elektrický či plynový měřič. Takové zařízení se přimontuje namísto klasického měřiče a spotřebiteli je poté poskytnuto rozhraní, kde si může snadno data o spotřebě zobrazit. Takové rozhraní může být zařízením v domácnosti či aplikací v mobilním telefonu. Zde je poté vizuálně zobrazena spotřeba a její cena pro aktuální čas. Senzory jsou doposud většinou na těžko docílitelných místech či zabezpečeny před vnějšími živly. Díky chytrému měřiči a jeho rozhraní, lze snadno docílit ke snížení spotřebě či její správě – nastavení denního/měsíčního limitu či jeho uzavření v případě vysokého úbytku.

Zdroj: [19]

1.4.1 Výhody

Běžné současné domácnosti musí obcházet odborník, který sepisuje informace z klasického měřiče a domácnost se o své spotřebě, co se peněz týče, může pouze domnívat. S chytrým měřičem má onu spotřebu nejen v m^3, ale i její finanční přepočet. Z grafického zobrazení, vztaženého k určitému času, lze vypozorovat, kde by domácnost mohla ušetřit. Lze takto snížit emise, ušetřit peníze a získat lepší dohled nad domácností.

Chytrý měřič, jak je již z předchozích kapitol a jasné, patří do zařízení IoT. Zařízení mezi sebou komunikují a sdílejí mezi sebou podstatná data. Pokud bude spotřebitel disponovat celou řadou IoT zařízení, bude mu velmi usnadněn přístup k důležitým datům, které by běžný člověk jen těžko získával.

1.5 Monitorování energií

Na základě průzkumů bylo zjištěno, že náklady na energie jsou druhou největší položkou firemního hospodaření. Pro nákupní centra tato položka představuje 25 až 40 % výdajů za provoz budovy. První místo u firemního hospodaření tvoří mzdy, které lze těžko snížit, stejně jako náklady na materiál, neboť s tím přichází určité snížení kvality. Energie tento problém nesdílí – snížení energií naopak umožňuje lepší finanční hospodaření.

Zdroj: [5]

1.5.1 Monitorování vody

Voda se s klimatickými změnami a růstem populace stává více a více vzácným zdrojem. Její zachování má vysokou prioritu, a obecně problematika vodního hospodářství, po celém světě. Jedním z problémů je osamostatňování obyvatel do vlastních obydlí. To směřuje k menší efektivnosti spotřeby vody než u sdílených ubytovacích prostor.

Za několik posledních desetiletí proběhlo několik studií zabývající se spotřebou vody v domácnosti, kde byla subjektem testování kuchyň. Byly analyzovány zvyky spojené s mytím nádobí ve čtyřech evropských zemích. Data byla naměřena pomocí chytrého senzoru v 81 domácnostech, a navíc byl zaznamenán video záznam pro spojení činností ke spotřebě vody. Díky tomu bylo umožněno rozlišit, zda byla spotřebována voda při mytí, vaření či jako pitný zdroj. Toto vedlo k sestavení matematického modelu spotřeby vody. Pozdější stejná studie, pro 558 domácností, zjistila vyšší spotřebu vody v kuchyni u mužů než u žen. Rovněž bylo prokázáno, že je mytí nádobí v užitkové nebo odpadní vodě odvozeno z kulturních a geografických podmínek nebo národních zvyků.

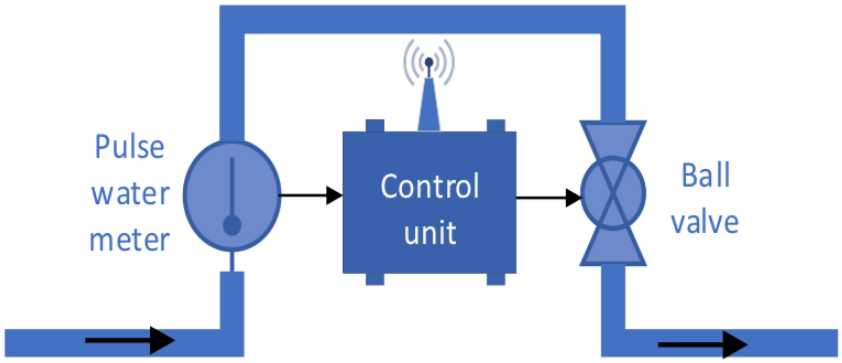
Zatímco vizualizace dat, získaných ze senzoru, je určitě krok správným směrem pro přehlednost nad domácností, snížení spotřeby vody současnou generací vodních měřičů může být až 1 %. Tato hodnota může být zavádějící, protože senzory doposud monitorují pouze určité aspekty používání vody v domácnosti.

Tato kapitola čerpá ze zdrojů: [2][3][4]

1.6 Chytrý vodoměr

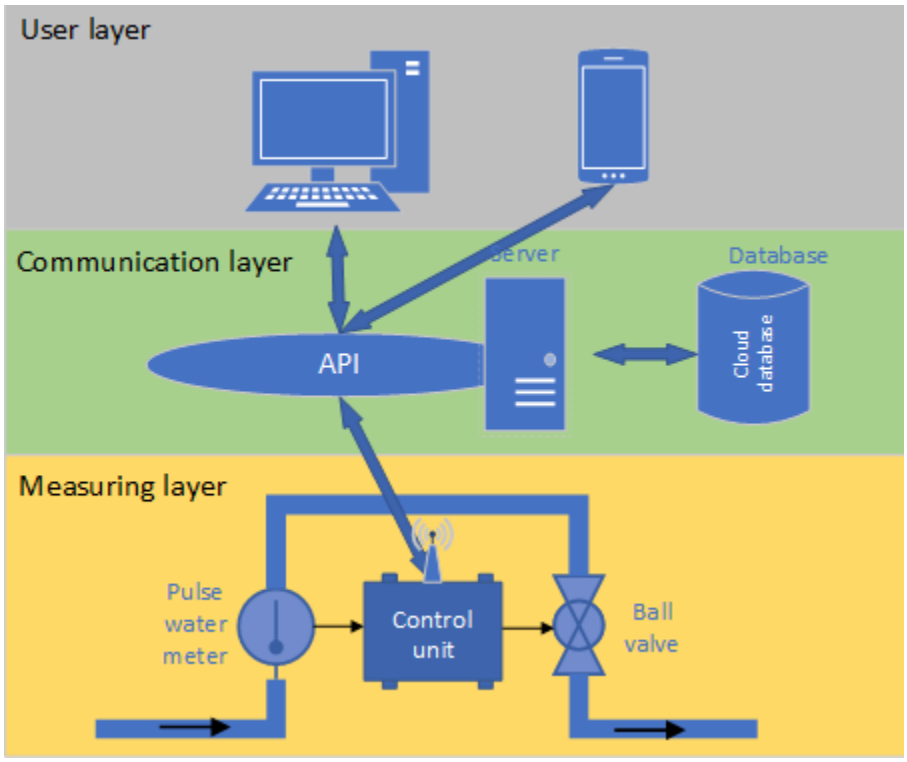
Na níže uvedené obrázku – základní koncept chytrého vodoměru, který je, včetně originálního znění popisu jeho dílčích prvků, uveden v práci pana Ing. Jana Fikejze, PhD.[20], jsou uvedeny čtyři základy části:

a) Pulzní vodoměr – poskytuje informace o naměřené spotřebě vody. K tomuto účelu lze využít jakéhokoliv pulzního vodoměru,
b) Centrální kontrolní jednotka – jednočipový minipočítač, *Raspberry Pi Zero*, který je rozšířený o jednotku obsahující standartní operační systém reálného času založený na operačním systému Linux *Raspbian*. V této jednotce je implementovaný software, který zajišťuje:
   i. Příjem informací o průtoku z vodoměru,
   ii. Komunikaci s aplikačním rozhraním pomocí bezdrátové sítě,
   iii. Pravidelné přepočty vodního průtoku, na základě zavedených pravidel v aplikační logice,
   iv. Kontrolu dvoucestného kulového ventilu
c) Záložní baterie – jedná se o záložní napájecí zdroj, pokud dojde k výpadku proudu. Pro zajištění větší stability, při poklesu hlavního napájecího zdroje, byl záložní zdroj doplněn o vhodný kondenzátor,
d) Dvoucestný kulový ventil – spravuje průtok vody na pomocí centrální kontrolní jednotky, která na základě uživatelského požadavku nebo v důsledku předdefinované události změní jeho stav.



Obrázek 3 – základní koncept chytrého vodoměru, zdroj: [20]

Systém, umožňující řešení chytrého vodoměru, lze rozdělit do tří vrstev – uživatelská vrstva, komunikační vrstva a vrstva správy vodoměru. Toto rozdělení, včetně ilustrace komunikace mezi těmito vrstvami, je uvedeno na níže uveden obrázku – Koncept sytému chytrého vodoměru.



Obrázek 4 – Koncept sytému chytrého vodoměru, zdroj: [20]

Pro umožnění manipulace s daty (vizualizace či jejich správa), bylo nutné využít programové aplikační rozhraní (API). Toto rozhraní je podrobněji rozvedeno v následující kapitole 1.7.

Zdroj: [20]

1.7 Application Programming Interface

Application Programming Interface neboli API, je rozhraní, které obsahuje sbírku tříd, funkcí či procedur, které může programátor využívat při vývoji. Může být využito například pro komunikaci serveru s aplikací či webovou stránkou nebo pro zajištění komunikace dvou a více aplikací. Existuje několik druhů API. Pro vývoj aplikace eVodoměr bude využita architektura REST, která je orientována datově. Její alternativy, například XMP-RPC nebo SOAP, jsou orientovány procedurálně.

Zdroj: [22]

1.7.1 Web API

Jedná se o aplikační programovací rozhraní pro webový server nebo webový prohlížeč. Většinou je limitován na webové aplikace typu **Client-side**, doplněn o webové frameworky, které neobsahují detailní serverové implementace.

Zdroj: [21]

1.7.2 Representational State Transfer – REST API

Jedná se o architekturu rozhraní, které je navržené pro distribuované prostředí, které bylo představeno, v roce 2000, v disertační práci „Architectural Styles and the Design of Network-based Software Architectures“ od Roye Fieldinga, který je spoluautorem protokolu HTTP. Ve zkratce jde o službu, díky které je umožněno pomocí HTTP požadavků provádět čtyři základní operace CRUD – vytvářet, číst, aktualizovat či úplně mazat data z databáze. Tyto operace jsou vykonávány na základě čtyř základních metod přístupu:

- GET – metoda zajišťující přístup ke zdrojovým informacím,
- POST – metoda pro vytvoření dat,
- PUT – metoda pro změnu dat. Tato operace je velmi podobná operaci pro vytvoření, avšak voláme konkrétní objekt v databáze, skrze jeho URI, který chceme změnit,
- DELETE – metoda pro smazání zdrojových informací.

Zdroj: [1]

1.8 Mobilní aplikace eVodoměr

Tato kapitola obsahuje shrnutí o aplikaci, která byla cílem praktické části této práce. Detailní popis je uveden v kapitole Návrh a Implementace.

1.8.1 Zadání

Primárním cílem práce je provést návrh a implementaci mobilní aplikaci, pro operační systém iOS v programovacím jazyce Swift, pro správu a vizualizaci dat o spotřebě vody chytrého vodoměru. Pomocí REST API je aplikaci umožněna základní správa chytrého vodoměru a jsou poskytnuta data pro grafické přehledy o spotřebě vody, která přísluší danému vodoměru přihlášeného uživatele. Přihlášení uživatele do aplikace je též realizováno pomocí REST API. Veškeré požadavky na databázi jsou šifrovány pomocí JSON Web Token pro zachování

bezpečnosti. Uživateli je umožněno editovat některé hodnoty vodoměru, jež přísluší danému uživateli. Je též možné, aby uživatel změnil stav vodoměru – uzavřený, otevřený. Aplikace bude testována na reálném zařízení.

1.8.2 Základní popis aplikace

Při prvním spuštění, které následuje po instalaci, aplikace zobrazí tzn. **Wizard**. Ten pro aplikace představuje tutoriál nebo přehled o tom, co aplikace nabízí či co umí. V případě aplikace eVodoměr, *Wizard* obsahuje přivítání uživatele a informace o tom, co aplikace uživateli umožňuje – vizualizovat si svá data prostřednictvím grafů a daný senzor konfigurovat. Po dokončení prohlídky se již *Wizard* nikdy nezobrazí, pokud nedojde k odinstalování aplikace a opětovné instalaci.

V dalším kroku, pro další možnou interakci, je nutné se do aplikace přihlásit. Aplikace je doplňkem webové stránky evodomer.cz, není tedy možné se z aplikace zaregistrovat. Pokud uživatel nemá doposud účet vytvoří, musí přejít na zmíněnou webovou stránku a tam registraci provést. Mezitím, co se uživatel přihlásil a je přesměrováván na seznam všech senzorů, je do lokální databáze uložena role daného uživatele. Tato role je dekódována z JSON Web Tokenu, který byl uživateli, na základě přihlášení, zaslán serverem. Ten je poté využíván v každém požadavku na serveru, pro autentizaci uživatele.

Po přesměrování je vyslán na server požadavek o data všech senzorů, které přísluší přihlášenému uživateli. Dokud není tabulka naplněna daty nebo nedojde k vypršení životnosti požadavku, je zobrazen indikátor probíhající aktivity. V případě vypršení požadavku nebo výskytu určité chyby, aplikace zobrazí upozornění s příslušným textem, reagující na vyskytlou situaci. Po naplnění tabulky jsou uživateli zobrazeny informace o stavu senzoru (aktivní/neaktivní), stavu uzávěru (otevřený/uzavřený) nebo o celkové spotřebě vodoměru. Z tohoto okna může uživatel přejít dvěma způsoby – odhlásit se pomocí tlačítka, které je součástí navigační lišty nebo přejít na detail senzoru, kliknutím na položku v tabulce. Při odhlášení dojde k přesměrování uživatele na přihlašovací okno.

V případě výběru položky z tabulky, dojde k přesměrování uživatele na detail senzoru. Aplikace mezitím odešle požadavek, pro získání doplňujících informací o senzoru. Ve výchozím stavu detailu, je uživateli zobrazena spotřeba k aktuálnímu dni a za aktuální měsíc, prostřednictvím koláčových grafů, finanční přepočet aktuálních spotřeb a, ve spodní části okna, sloupcový graf, s hodinovou spotřebou v rámci aktuálního dne. K tomuto grafu se váže kontrolní panel, kde si uživatel může změnit zkoumané období (den/měsíc/rok) a dané obchodí

procházet. Tuto akci lze provádět i pomocí přímého výběru sloupce v grafu. To znamená, že pokud uživatel vybere rok, jako zkoumané období, a poté klikne na sloupec, příslušící určitému měsíci, graf na to zareaguje, aplikace změní období v kontrolním panelu na „měsíc“ a graf se překreslí pro hodnoty k vybranému měsíci. Při výběru daného období či data termínu, dojde k vyslání požadavku na server o požadovaná data. Po získání dat dojde k překreslení sloupcového grafu. Tato akce je však dovolena pouze pro období rok a měsíc. Koláčové grafy navíc obsahují propočet aktuální spotřeby ku limitu, který je na vodoměru nastaven. Tento přepočet je vyobrazen jak vizuálně, jako část grafu, tak pomocí procent uprostřed každého grafu. Při překročení tohoto limitu dojde k automatickému uzavření ventilu.

Průchod aplikace je napojen na navigační třídu, která zajišťuje přechody mezi okny. Díky němu je v levé části navigační lišty tlačítko pro přechod zpět na předchozí okno – kolekci senzorů. V detailu senzoru je namísto tlačítka pro odhlášení, tlačítko pro zobrazení menu. Uživatel se, prostřednictvím menu, může například odhlásit či provést konfiguraci senzoru. Menu konkrétně nabízí možnosti pro:

- Kolekce senzorů
- Editaci senzoru
- Stav uzávěru
- Konfigurace
- Odhlášení

Toto není však standartní zobrazení. V tento moment je uplatněna role, která přísluší uživateli a byla uložena při jeho přihlášení. Ta může nabývat dvou stavů – *uživatel* a *administrátor*. Výše zmíněný výčet položek je příslušný pro roli **administrátor**. V opačném případě není zobrazena položka *Konfigurace*.

Kolekce senzorů je další možnost, jak se dostat zpět na výpis všech senzorů. Obdobou je již zmíněné levé tlačítko v navigační liště. Při přechodu dojde opět k požadavku na server, kdyby došlo v průběhu k aktualizaci dat buď ze strany serveru nebo ze strany uživatele.

V **editačním oknu** je uživateli umožněno nastavovat atributy vodoměru, jako je název, pro přehlednost v tabulce senzorů, Cena za m³ či denní a měsíční limit. Pokud si uživatel přeje hodnoty změnit, upraví je buď skrze textová pole nebo pomocí komponenty *UIStepper*, kde pomocí inkrementačního a dekrementačního tlačítka lze hodnotu měnit. Po potvrzení všech změn je nutné data uložit pomocí tlačítka, které se nachází ve spodní části okna. Před odesláním

jsou hodnoty validovány, aby nebyl poslán požadavek s nekonzistentními datovými typy. Poté je vyslán požadavek na server pro úpravu hodnot vodoměru.

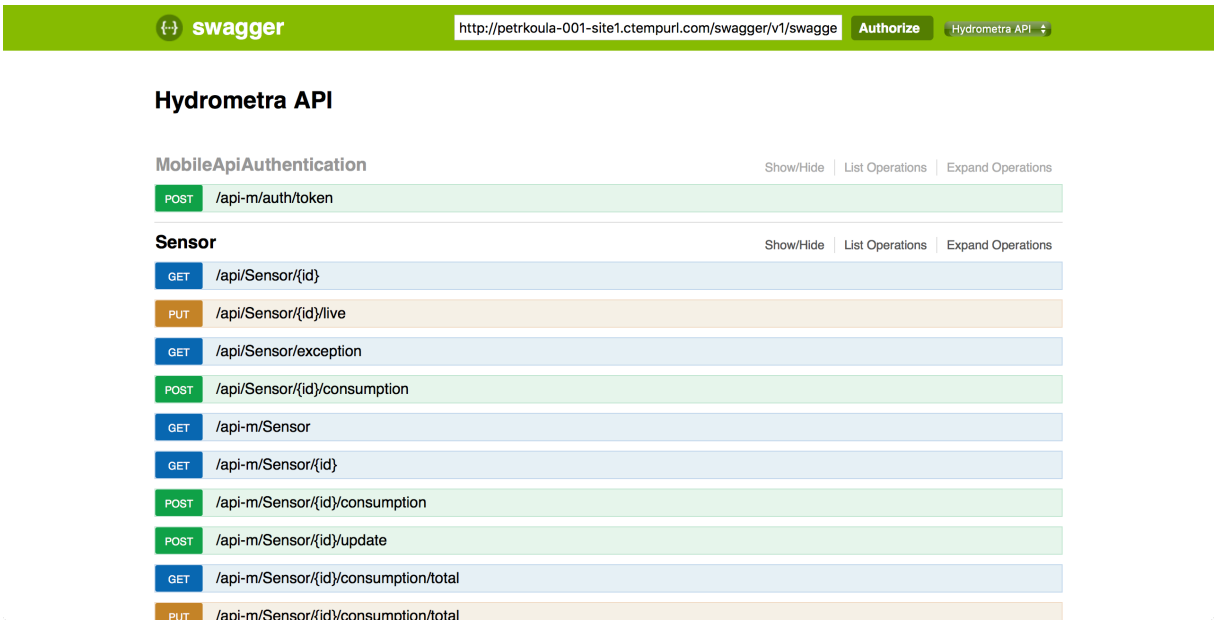
Stav uzávěru je okno, kde je uživateli zobrazen stav ventilu s možností jeho změny. Pomocí barevné indikace tlačítka, a textu v okně, je prezentován jeho stav. Toto tlačítko slouží pro manuální změnu stavu senzoru. Při kliknutí, je na server odeslán požadavek o otevření nebo uzavření ventilu. Aplikace poté se zpožděním vysílá požadavky, dokud nedojde ke změně stavu ventilu. Poté jsou přepsány včetně časového razítka, kdy k akci došlo.

Konfigurační okno je k dispozici pouze uživateli v roli administrátora. Jedná se okno, kde je umožněno měnit hodnoty spojené s počtem impulzů daného vodoměru či změnit hodnota celkové spotřeby. Proces změny hodnot je stejný, jako v **editačním okně**.

Aplikace disponuje pouze online verzí. To znamená, že v případě, že uživatel nemá k dispozici přístup k internetu, není možné si informace o senzoru zobrazit. Pokud dojde k odpojení sítě v průběhu požívání, je uživateli o této situaci zobrazeno upozornění a není možné aplikaci plnohodnotně využívat.

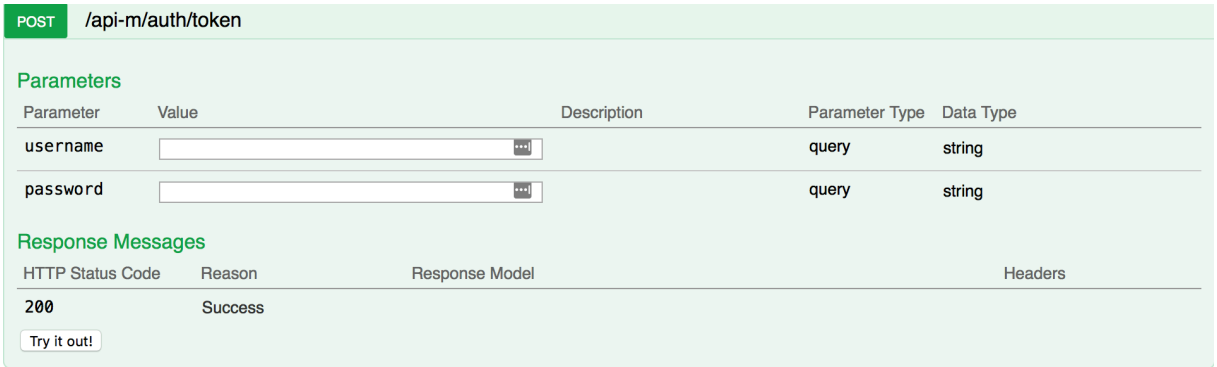
1.8.3 Komunikace s databází

Pomocí REST API, které bylo popsáno v kapitole 1.7.2, je aplikaci umožněna komunikace s databází. Funkci, které byli navrženy Ing. Petrem Koulou, jsou rozděleny do sekcí dle jejich zařazení. Toto API je též využíváno webovou stránkou http://evodomer.cz, která je obdobou mobilní aplikace. Z důvodu těchto dvou odlišných přístupů je API rozděleno na webové operace a mobilní operace. Toto rozlišení je realizováno pomocí klíčových slov **/api/** a **/api-m/**. Nutnost implementovat oba přístupny je především proto, že ověření uživatele je v mobilní aplikaci pomocí JSON Web Tokenu, kdežto pro webovou aplikaci pomocí *Cookies*.



Obrázek 5 – Webová stránka s operacemi API, zdroj: vlastní

Každá operace je definována metodou zpracování a její URI. U všech požadavků, kromě přihlášení uživatele, je nutné doplnit hlavičku, která obsahuje autentizaci uživatele. Autentizace uživatele je zprostředkována pomocí JWT tokenu, který byl získán při přihlášení. Požadavek může též obsahovat parametry, které filtrují požadovaná data či, jak je vidět níže, můžou parametry obsahovat hodnoty pro ověření uživatele. Celý proces přihlášení je popsán v kapitole 3.1.1.



Obrázek 6 – Operace REST API, zdroj: vlastní

Zdroj: [23]

VÝVOJ MOBILNÍ APLIKACE

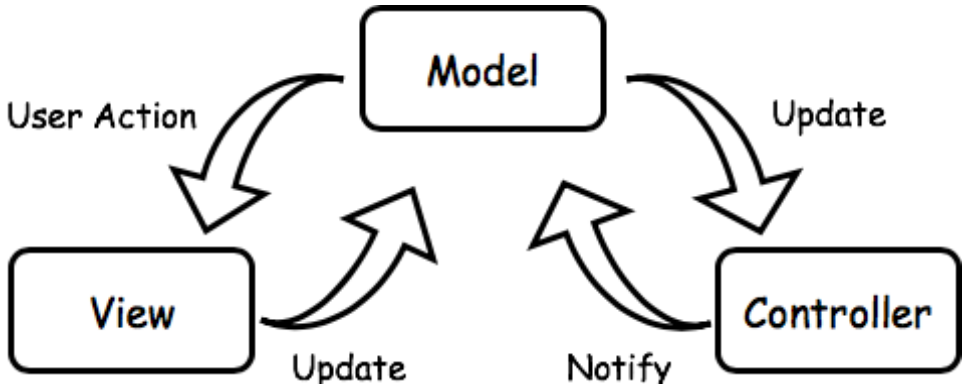
2 POUŽITÉ TECHNOLOGIE

Jak bylo již popsáno výše, aplikace bude psána v nativním jazyce Swift verze 4.0. Tento jazyk je jednou ze dvou nativních variant, kterou lze pro vývoj zvolit. Druhou, starší variantou, je programovací jazyk Objective-C. Pro vývoj bylo zvoleno vývojové prostředí Xcode, které je k dispozici v oficiálním obchodě AppStore v aktuální verzi 9.3. Vývoj v tomto prostředí je podmíněn vlastnictvím zařízení s operačním systémem macOS. Existují varianty, jak vyvíjet aplikace pro operační systémy od společnosti Apple, ale Xcode je nativním prostředím. V tomto prostředí lze vytvářet aplikace pro všechna dobře známá zařízení a jejich operační systémy– Apple Watch (watchOS), iPhone a iPad (iOS), Mac a MacBook (macOS) a Apple TV (tvOS). V tomto prostředí je možné vyvíjet projekt, jejímž výsledkem bude aplikace pro kombinaci výše zmíněných platforem.

Toto prostředí umožňuje vše, co je nutné pro vývoj – implementace kódu, navržení designu či testování. Další výhodou je velká škála simulovaných zařízení. Xcode nabízí při vývoji všechny velikosti zařízení, které jsou na trhu – od chytrých hodinek Apple Watch až po iPad Pro. Tato skutečnost je velkou výhodou pro vývojáře, neboť umožňuje vyzkoušení funkčnost vyvíjené aplikace ve všech nabízených velikostech. Výchozími fyzickými zařízeními, na kterých byla aplikace testována, byly mobilní telefony značky Apple, modely iPhone 5s a iPhone 7.

2.1 Architektura

Základní architekturou, která je využita pro vývoj aplikace v prostředí Xcode s pomocí jazyka Swift, je architektura MVC *Model-View-Controller*. Ta rozděluje aplikaci do pomyslných tří částí, aby se například oddělila implementace grafických prvků od logiky aplikace. *Model* představuje data aplikace, se kterými aplikace pracuje. *View* je to, co je uživateli vizuálně prezentováno. Jedná se tedy o grafické rozhraní. Úkolem *Controlleru* je tyto dva předchozí prvky spojit a utvořit tak celek.

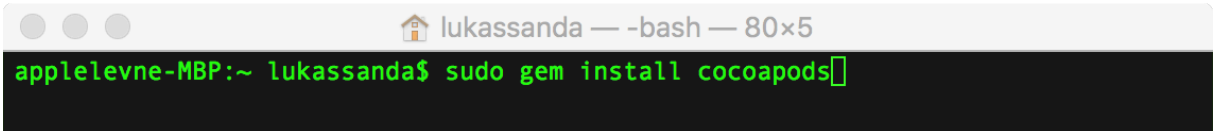


Obrázek 7 – Architektura MVC, zdroj: vlastní

Zdroj: [24]

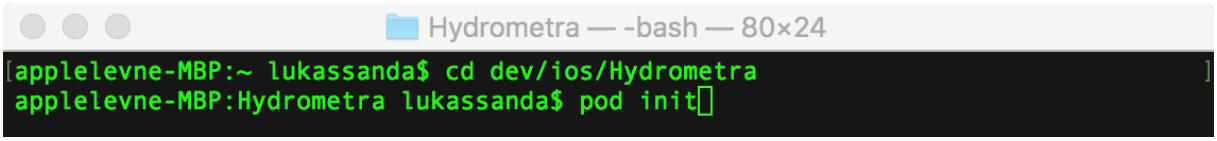
2.2 Manažer závislostí – CocoaPods

Velkou výhodou jazyka Swift je kompatibilita s frameworkem Cocoa Touch. Ten umožňuje snadnou implementaci projektu třetí strany, pomocí manažeru závislostí CocoaPods. To znamená, že si lze do aplikace zaimplementovat jiný projekt jako doplňující knihovnu. Taková knihovna je v projektu prezentována jako **Pod**. Tyto knihovny jsou k dispozici skrze platformu GitHub, kde jsou vyvěšeny pro veřejnost pod licencí MIT. Tento manažer je spouštěn pomocí programovacího jazyka Ruby, který je dostupný od operačního systému MacOS 10 nebo jej lze snadno doinstalovat. Pro nainstalování CocoaPods, pro Xcode verze 8 a 9, stačí do terminálu zadat.



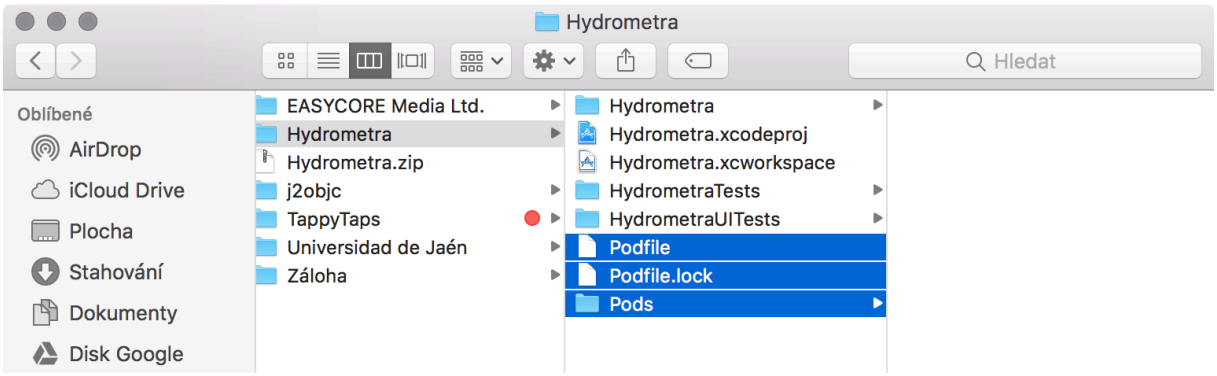
Obrázek 8 – Instalace CocoaPods, zdroj: vlastní

Aby mohl uživatel využívat tento manažer, musí být pro požadovaný projekt inicializován. Je nutné přejit do adresáře projektu a zadat příkaz *pod init*. Toho je docíleno pomocí následujících dvou příkazů.



Obrázek 9 – Inicializace CocoaPods v projektu, zdroj: vlastní

Projekt následně obsahuje soubory, které jsou vázány na námi implementovaný manažer. Každý vygenerovaný soubor má svou vlastní úlohu.



Obrázek 10 – Obsah adresáře projektu, zdroj: vlastní

Vygenerovaný adresář *Pods* obsahuje všechny námi implementované projekty jako doplňující knihovny. V *Podfile* lze definovat výchozí platformu a její verzi, pro kterou bude projekt definován. V neposlední řadě obsahuje námi požadované knihovny a verzi, kterou si přejeme zahrnout do projektu. Verze je nepovinným údajem a pokud není uvedena, je manažerem stažena nejnovější stabilní verze. Nakonec je možné projektu definovat knihovny, které budou použity pouze pro testování. Soubor *Podfile.lock* zajišťuje generování aktuálně nainstalované

verze **Podu** v projektu. V níže uvedeném případě zajistí, že *SnapKit* bude vygenerován ve verzi 4.0.0, která je v projektu naistalována a ne jiná.

Podfile

```
# Uncomment the next line to define a global platform for your project
 platform :ios, '11.0'

target 'Hydrometra' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for Hydrometra
  pod 'Charts'
  pod 'Alamofire'
  pod 'SideMenu'
  pod 'SnapKit', '~> 4.0.0'
  pod 'JWTDecode'


  target 'HydrometraTests' do
    inherit! :search_paths
    # Pods for testing
  end

  target 'HydrometraUITests' do
    inherit! :search_paths
    # Pods for testing
  end

end
```

Obrázek 11 – obsah souboru Podfile, zdroj: vlastní

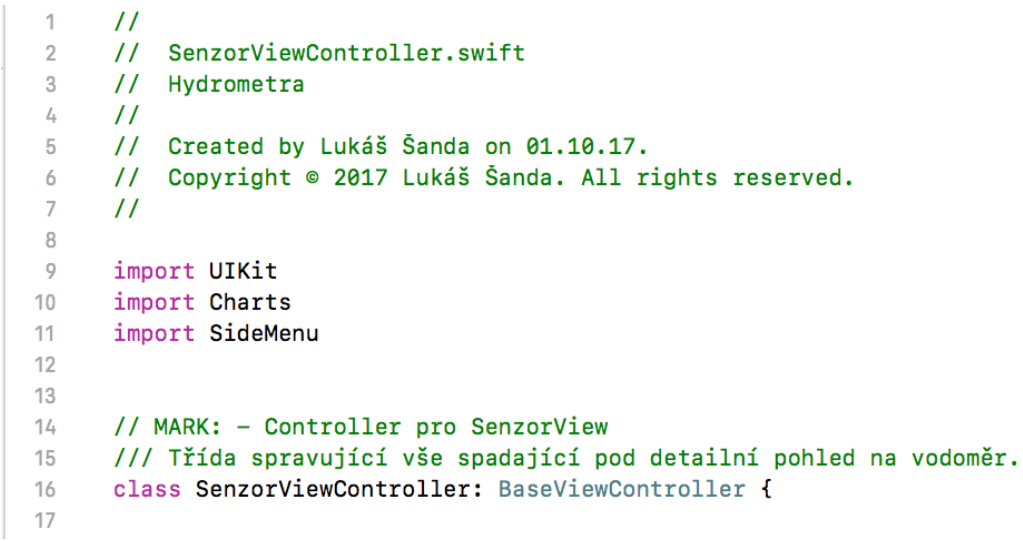
Pokud však takto nadefinujeme verzi doplňujícího projektu, nedojde při vyvolání hromadné aktualizace, pomocí níže uvedeného příkazu, k aktualizování na novější verzi. Avšak v průběhu aktualizaci nás manažer na tuto skutečnost upozorní.

Hydrometra — -bash — 74×5

```
applelevne-MBP:Hydrometra lukassanda$ pod update
```

Obrázek 12 – příkaz pro aktualizaci nainstalovaných knihoven, zdroj: vlastní

Pro použitý doplňující knihovny je nutné přidat následující řádek, pro její importování, do souboru, ve kterém bude využita. Dle určité programátorské konvence se takový příkaz vkládá na první řádky, mezi komentář obsahující *Copyright* a začátek implementace.



```
1   //
2   //  SenzorViewController.swift
3   //  Hydrometra
4   //
5   //  Created by Lukáš Šanda on 01.10.17.
6   //  Copyright © 2017 Lukáš Šanda. All rights reserved.
7   //
8
9   import UIKit
10  import Charts
11  import SideMenu
12
13
14  // MARK: - Controller pro SenzorView
15  /// Třída spravující vše spadající pod detailní pohled na vodoměr.
16  class SenzorViewController: BaseViewController {
17
```

Obrázek 13 – importování projektu z CocoaPods, zdroj: vlastní

Zdroj: [25]

3 NÁVRH A IMPLEMENTACE

V této kapitole budou detailně popsány stěžejní funkce mobilní aplikace. Podkapitoly jsou rozděleny podle pohledu v aplikaci. Popsané funkce jsou zpracovávány dvojím způsobem – synchronně či asynchronně. Asynchronní jsou většinou ty, které jsou použity pro komunikaci se serverem. Tato komunikace je vykonávána pomocí pomocné knihovny **Alamofire**, ve verzi 4.7.1.

Volitelným parametrem požadavků této knihovny jsou parametry, které umožňují zasílat data na server, hlavička, která definuje doplňující informace požadavku (typ akceptovatelných dat, formát akceptovatelných dat aj.) a kódování. Kódování je dvojího základního typu – **URLEncoding** a **JSONEncoding**. V případě specifických požadavků si lze nadefinovat vlastní. **URLEncoding** se využívá, pokud je požadováno zašifrování parametrů v URL adrese. Naopak **JSONEncoding** se využívá v případě, pokud chceme parametry zašifrovat jako objekt a zaslat je v těle HTTP požadavku.

Veškerá komunikace je zabezpečena pomocí standartu **JWT** – JSON Web Token. Jedná se o způsob zabezpečení komunikace s pomocí JSON objektu. Informace zaslané touto technologií jsou digitálně podepsané, proto jde o důvěryhodná data. Jsou podepsány pomocí veřejného/soukromého klíče, který je generován pomocí **RSA**, nebo pomocí autentizačního kódu, generovaného **HMAC** algoritmem – Keyed-hash Message Authentication Code.

Zdroj: [7]

3.1 Přihlašovací okno

Toto okno je první, co uživatel uvidí, při běžném spuštění aplikace. V rámci tohoto okna se lze do aplikace pouze přihlásit, není tedy možné zaregistrovat nového uživatele. Z toho důvodu obsahuje pohled pouze dvě textová pole, pro uživatelské jméno a heslo, a tlačítko pro potvrzení.

3.1.1 Přihlášení uživatele

Proto, aby se mohl uživatel přihlásit, musí vyplnit své uživatelské jméno a heslo. Poté je pomocí statické třídy *AuthenticationAPI* zprostředkováno kontaktování serveru a ověření, pomocí REST API, zda je uživatel uveden v databázi uživatelů.

```
AuthenticationAPI.shared.login(username: username, password: password)
```

Ukázka kódu 1 – funkce pro přihlášení uživatele

Tato třída má definovaného předka *AlamofireManager*. Tento předek definuje atributy, které jsou sdíleny obdobnými statickými třídami, jako je zmíněná *AuthenticationAPI*. Definuje *baseURL* – URL adresa odkazující na adresu serveru, instanci knihovny **Alamofire** a výchozí hodnoty pro hlavičku požadavku. Výše zmíněná funkce zajišťuje kontaktování serveru s požadovanými hodnotami.

```
func login(
    username: String,
    password: String,
    completionHandler: @escaping (_ responseObject: String?, _ error: Error?) -> ())
```

Ukázka kódu 2 – definice funkce pro přihlášení

Funkce je doplněna o URI daného požadavku; parametry (uživatelské jméno a heslo) a o specifickou definici hlavičky.

```
let url = "\(baseURL)/api-m/auth/token"
let parameters: Parameters = [
    "username" : username,
    "password" : password
]
headers = ["Accept": "application/json"]
```

Ukázka kódu 3 – doplňující parametry požadavku HTTP

V dalším kroku je, pomocí instance knihovny, zprostředkován požadavek, který obsahuje výše zmíněné parametry. Navíc je zde uvedeno kódování a metoda požadavku – **POST**. Výstupem této funkce je výstupní parametr.

```
@escaping (_ responseObject: String?, _ error: Error?) -> ()
```

Ukázka kódu 4 –výstupní parametr funkce pro přihlášení

Ten zajišťuje, že tato asynchronní funkce, po jejím vykonání vrátí požadovaný objekt, v tomto případě typu *String*.

```
Manager.reqest(url, method: .post, parameters: parameters, encoding: URLEncoding.default, headers: headers).responseJSON {
(response) in
    switch response.result {
    case .success:
        if response.response?.statusCode == 200, let json = response.result.value as? NSDictionary {
            completionHandler(json["token"] as? String, nil)
        } else {
            completionHandler(nil, RuntimeError("Chyba při dynamickém přetypování."))
        }
    case .failure(let error):
        completionHandler(nil, error)
    }
}
```

Ukázka kódu 5 – implementace požadavku HTTP na REST API pro přihlášení uživatele

V případě úspěchu funkce vrátí zmíněný token, který se uloží do lokální databáze telefonu a je poté využíván v dalších požadavcích na server. Aplikace je postavena dvoustavově – přihlášená osoba je v roli uživatele nebo v roli administrátora. Tato role je reprezentována skrze hodnotu, která je zakódována v serverem zaslaném tokenu.

Pokud nastal problém na straně serveru, funkce vrátí chybu, která se vyskytla. Tento výsledek může být způsobem skrze dva scénáře– problém s připojením k síti či nekorektně zadané uživatelské údaje. Nejprve se zkontroluje, zda je k dispozici připojení k síti. To je zajištěno pomocí pomocné knihovny **Reachability**, kterou vytvořil Ashley Mills. V případě obou stavů dojde k zobrazení systémového upozornění se specifickým textem, který je vázán k dané situaci.

```
if error == nil {
    // Uložení tokenu do lokální paměti
    if let token = token {
        UserDefaults.standard.userLoginToken = token
    }
    // Dekódování role uživatele
    JWTManager.shared.decode()

    // Přechod na kolekci vodoměrů
    let storyboard: UIStoryboard = .collection
    let collectionView = storyboard.instantiateViewController(withIdentifier: "collectionNavigationController")
     self.show(collectionView, sender: nil)
} else {
    //Pokud není k dispozici internetové připojení či byly zadány špatně přihlašovací údaje
    if ReachabilityManager.shared.isConnectedToNetwork() {
        self.showWrongCredentialsAlert()
    } else {
        self.showNoConnectionAlert()
    }
```

Ukázka kódu 6 – ověření uživatele funkcí pro přihlášení

Jelikož se nejedná o nijak datově náročnou informaci, je možno tuto informaci pomocí JWT zaslat. Tato role je dekódovaná pomocí funkce *decode()* ve statické třídě *JWTManager*.

Role je v JWT uvedena formou klíč-hodnota a nachází se na prvních sedmi znacích JWT tokenu.

```
let token = UserDefaults.standard.userLoginToken
let range = { () -> Range<String.Index> in
    let start = token.index(token.startIndex, offsetBy: 7)
    let end = token.endIndex

    return start..<end
}()
```

Ukázka kódu 7 – oddělení prvních 7 znaků z JWT

Pomocná knihovna *JWTDecode*, která je určená pro práci s JWT, pomocí *try-catch* bloku dekóduje z našeho uloženého tokenu daný rozsah a pomocí funkce *claim(name: "role").string* získá požadovanou hodnotu role. Ta je poté uložena do lokální databáze pro pozdější zpracovávání.

```
do {
    let jwt = try JWTDecode.decode(jwt: String(token[range]))
    if let role = jwt.claim(name: "role").string {
        UserDefaults.standard.userRole = role == "user" ? .user : .admin
        return
    }
} catch {
    print("Failed to decode JWT: \(error)")
}
```

Ukázka kódu 8 – dekódování JWT a nastavení role

V případě neúspěchu dojde k nastavení defaultní role – user.

```
UserDefaults.standard.userRole = .user
```

Ukázka kódu 9 – nastavení výchozí role do lokální databáze

3.2 Kolekce senzorů

Po úspěšném přihlášení dojde k přesměrování na kolekci uživateli dostupných senzorů. Pro implementaci tabulky je v jazyce Swift připravena komponenta *UITableView*, která pomocí delegovaných funkcí zajistí chod tabulky. Autor však zvolil obyčejné okno, komponentu *UIView*, ve kterém tabulku vytváří programově. Toto zvolil z důvodu, že může nastat situace nedostupnosti dat nebo neexistují žádná data. Výchozí stav je takový, že okno obsahuje situaci prázdné tabulky, tzn. titulek a text informující o tomto stavu. Pokud dojde k načtení dat, je toto okno překryto tabulkou s daty.

3.2.1 Získání dat z databáze

Každá třída, která se starající o komponenty okna (*UIViewController*, *UITableViewController*, *UICollectionViewController*), obsahuje delegované funkce, které jsou spojeny s vznikem či zánikem okna. Tyto funkce jsou volány při první inicializaci okna, při jeho každém zobrazení či v moment zavření daného okna.

V kolekci senzorů je požadováno, aby při každém přesměrování došlo k obnově dat. K tomuto účelu je využita funkce *viewWillAppear()*, která je volána pokaždé, když dochází k přesměrování na dané okno. Jelikož bude vyslán asynchronní požadavek, autor zobrazil indikátor probíhající aktivity, který je popsán v kapitole 3.7.1, společně s efektem rozmazané pozadí, pro indikaci načítání dat.

Poté je pomocí statické třídy *SenzorAPI* a její funkce *requestSenzors()* kontaktován server s dotazem na data všech senzorů.

```
SenzorAPI.shared.requestSenzors { (senzors, error)
```

Ukázka kódu 10 – volání funkce pro získání dat všech senzorů

Pokud proběhl požadavek v pořádku a server nezaslal v odpovědi určitou chybu, jsou senzory uloženy do pomocné třídy, pro případ manipulace s originálními daty.

```
VodomerData.shared.senzors = senzors
```

Ukázka kódu 11 – uložení senzorů do pomocné třídy

Díky tomuto požadavku jsou získány základní informace o senzoru. Tabulka všech senzorů navíc obsahuje informaci o celkové spotřebě daného senzoru. Proto je potřeba zaslat další požadavek o doplňující informace. Toto je zajištěno pomocí statické třída *VodomerData*, která zajistí doplnění informace o celkové spotřebě u všech senzorů. Při doplnění informací jsou uložena získaná data do lokální proměnné, odebrán indikátor aktivity a je zobrazena tabulka se získanými daty.

```
VodomerData.shared.totalConsumption(completionHandler: { (error) in
    if error == nil {
        self.senzors = senzors
        self.view.removeBlurLoader()
    }
```

Ukázka kódu 12 – implementace funkce zajišťující naplnění spotřeby všem senzorům

V případě neúspěchu je uživateli zobrazeno upozornění, že vznikl problém při komunikaci se serverem. S tím je spojena situace, že neexistují data pro naplnění tabulky a je zobrazeno originální okno, indikující absenci dat.

```
self.present(
    ReachabilityManager.shared.noConnectionAlert(showLogin: false),
    animated: true,
    completion: nil
)
```

Ukázka kódu 13 – nativní funkce pro zobrazení komponenty pro upozornění uživatele

Funkce *totalConsumption()* je pomocnou funkcí, která prochází postupně všechny senzory, které jsou uloženy v této pomocné třídě a doplní je o data celkové spotřeby. Výstupním parametrem funkce je pouze *error*, neboť není potřeba hodnoty nijak předávat, ale přímo nastavit danému senzoru. Výstupem je buď *nil*, což lze interpretovat jako úspěch, či vzniklá chyba.

```
func totalConsumption(completionHandler: @escaping (_ error: Error?) -> ()) {
    for senzor in VodomerData.shared.senzors { }
}
```

Ukázka kódu 14 – definice funkce pro zajištění naplnění spotřeby všem senzorům

Parametrem této funkce je opět identifikační číslo senzoru, neboť je součástí URI požadavku. Pokud se nevyskytl problém při komunikaci se serverem, je navrácena hodnota spotřeby, která je nastavena atributu senzoru.

```
ConsumptionAPI.shared.requestTotalConsumption(id: senzor.idHw) { (consumption, error) in
    if error == nil {
        senzor.consumptionTotal = consumption
```

Ukázka kódu 15 – volání funkce pro naplnění spotřeby

3.2.2 Naplnění tabulky daty

Každá tabulka, reprezentována komponentou *UITableViewController*, obsahuje delegované funkce, které onu tabulku spravují. Funkce definují počet sekcí, počet řádků v sekci, výšku řádku či naplnění buňky daty. Jelikož autor zvolil přístup, že nemá třídu jako potomka komponentu spravující tabulku, ale je potomkem obyčejného okna – *UIViewController*, kde

vytváří tabulku programově, musí do definice třídy přidat protokoly, které umožní využívání těchto delegovaných hodnot. Těmito protokoly jsou *UITableViewDelegate* a *UITableViewDataSource*. Přidání se provede doplněním těchto klíčových slov za definici názvu třídy, jako je tomu i u jiných programovacích jazyků. Oba tyto protokoly dovolují chování oknu, jako by šlo o komponentu k tomu určenou – *UITableViewController*.

```
class SenzorsTableController: UITableViewDelegate, UITableViewDataSource
```

Ukázka kódu 16 – zajištění komunikace s nativními komponentami pomocí dědičnosti

Dalším podstatným krokem je definovat, programově vytvořené tabulce, objekt, který bude představovat zdrojovou třídu. Tato třída zajistí naplnění daty či doplňujícími informace, jako byl již zmíněný počet sekcí nebo řádků. Jelikož autor vytváří tabulku přímo ve výše zmíněné třídě, uvede se tato třída jako onen zdroj.

```
tableView.delegate = self
tableView.dataSource = self
```

Ukázka kódu 17 – zajištění komunikace komponent v GUI

Poté je již možné používat delegované funkce, které jsou vázány k práci s tabulkou. Jedinou změnou, v případě autorem vybrané metody, je, že implementovaná funkce neobsahuje modifikátor *override*. To je z důvodu, že hlavním předkem je klasické okno a funkce tabulky jsou implementovány pomocí protokolů.

V posloupnosti definování obsahu tabulky, je nejprve volána funkce, která určuje počet sekcí v tabulce.

```
func numberOfSections(in tableView: UITableView) -> Int
```

Ukázka kódu 18 – delegovaná funkce pro počet sekcí v tabulce

Autor do této funkce navíc přidal rozhodnutí o zobrazení tabulky pomocí ternárního operátoru. Pokud lokální proměnná, obsahující pole senzorů, neobsahuje žádná data, je tabulce nastaven

atribut *isHidden* na *true*, čímž bude tabulka skryta a je zobrazeno okno s textem upozorňující na prázdnou tabulku.

```
tableView.isHidden = senzors.isEmpty ? true : false
```

Ukázka kódu 19 – definice viditelnosti tabulky

Hlavním výstupem funkce je však počet sekcí, kterými bude tabulka disponovat. Rozhodnutí je opět na základě obsahu lokální proměnné s informacemi o senzorech.

```
return senzors.isEmpty ? 0 : 1
```

Ukázka kódu 20 – počet sekcí na základě naplněnosti lokální proměnné

Dále je volána obdobná funkce, která definuje počet řádků v sekci. Pokud by tabulka obsahovala více sekcí, bylo by nutné, na základě indexu sekce, definovat její počet řádků. V tomto případě tabulka obsahuje jednu sekci a počet řádků je roven počtu záznamů v lokální proměnné.

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section:Int) -> Int {
    return senzors.count
}
```

Ukázka kódu 21 –implementace funkce pro určení počtu řádků v sekci

V neposlední řadě je nutné zvolit velikost řádku pro danou buňku. Je možné nastavit statickou hodnotu, pokud tabulka obsahuje ekvivalentní data. Pokud se jedná o dynamická data, u kterých nelze předpokládat jejich délku, je dobré zvolit buď variantu automatického určení dimenze nebo tuto velikost přepočítat na základě dat, které se mají zobrazit.

Určení automatické dimenze je metoda, která si sama, na základě definovaných závislostí mezi grafickými prvky v návrhu buňky, hodnotu dopočítá. Je však nutné nastavit odhadovanou výšku buňky, aby mohla dimenze vycházet z nějaké počáteční hodnoty. Autor si takto navrhl design buňky pro určitou velikost řádku a tu vložil do odhadované velikost.

```
tableView.estimatedRowHeight = 100
```

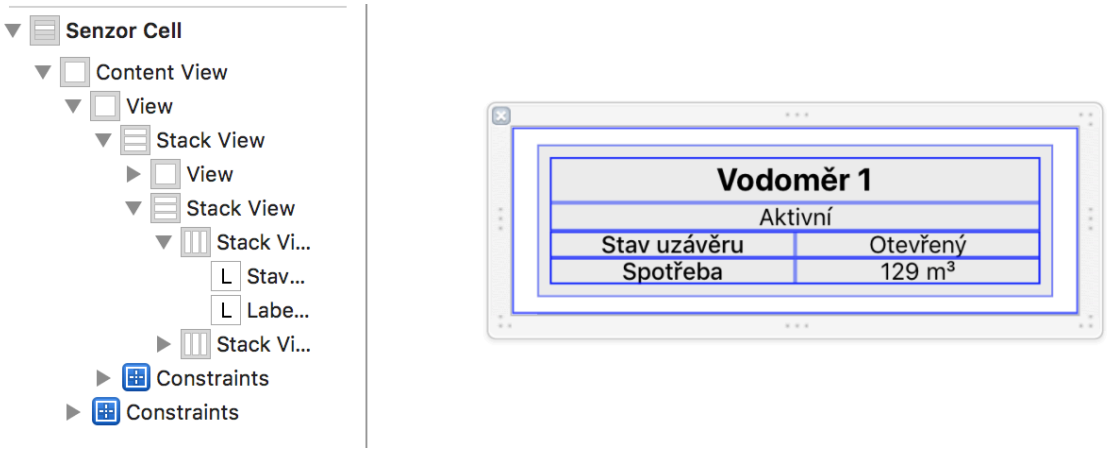
Ukázka kódu 22 – nastavení odhadované velikosti řádku

Poté je, pomocí funkce, rozměr sám dopočítán. Není však řečeno, že velikost řádku musí odpovídat odhadované výšce. Skutečnost se bude však pohybovat v blízkosti této hodnoty.

```
func tableView(_ tableView: UITableView, heightForRowAt indexPath IndexPath) -> CGFloat {
    return UITableViewAutomaticDimension
}
```

Ukázka kódu 23 – implementace funkce definující velikost řádku

Pravděpodobně nejdůležitější funkcí je naplnění buňky daty. To lze provést přímo z prostředí pro návrh grafického rozhraní, v takzvaném **Storyboardu**, programově nebo částečně graficky. Autor zvolil třetí způsob, který představuje vytvoření **Custom View**, které bude představovat danou buňku. Pro tento účel je nutné vytvořit soubor s příponou **.xib**.



Obrázek 14 – vytvoření vlastní buňky pro tabulku senzorů, zdroj: vlastní

Tento soubor se tváří podobně jako soubor s příponou .**storyboard**, avšak je nutné takové *Custom View* „zaregistrovat“, aby o jeho existuje měla tabulka povědomí. Poté je již možné v těle třídy s buňkou pracovat.

```
tableView.registerReusableNibCell(for: SenzorCell.self)
```

Ukázka kódu 24 – zaregistrování autorem vytvořeného okna

Pro práci s danou buňku, na určitém řádku tabulky, protokol obsahuje funkci *tableView(cellForRowAt:)*. V této funkci je nutné pro každý řádek buňku vytvořit, vložit data a vrátit, neboť tato buňka je návratovou hodnotou funkce. Datový typ proměnné bude třída, která je vytvořena pro práci s danou buňkou. Tato třída představuje model buňky.

```
let cell: SenzorCell =  tableView.dequeueReusableCell(for: indexPath)
```

Ukázka kódu 25 – instance buňky tabulky

Autor si poté, v rámci modelu buňky, vytvořil funkci *configure(for:)*, která zajistí naplnění příslušných komponent *UILabel* daty, které jsou příslušné senzoru, který má být na daném řádku zobrazen. Lokální proměnná se senzory je datový typ pole senzorů a na základě indexu řádku jsou hodnoty z pole vybírány a přiřazovány do buňky.

```
let senzor = senzors[indexPath.row]
```

Ukázka kódu 26 – instance senzoru z pole senzorů

Alternativou funkce *configure(for:)*, je naplnění hodnot přímo v popisované funkci, ale jelikož je nastavováno více položek, autor chtěl pro přehlednost kód rozčlenit. Přístup k prvkům by byl na základě vytvořené buňky, která obsahuje grafické komponenty jako svůj atribut.

```
cell.labelTitle.text = senzor.nameHw
```

Ukázka kódu 27 – nastavení textu textového štítku v těle funkce

Pokud je nastavení hodnot pro grafické komponenty zajištěn uvnitř modelu buňky, naplnění je provedeno jako nastavení atributu třídy.

```
labelTitle.text = senzor.nameHw
```

Ukázka kódu 28 – nastavení textu textového štítku v těle třídy buňky

Další delegovanou funkcí, kterou autor využil, je událost po kliknutí na řádek pro zobrazení detailu požadovaného vodoměru – *tableView(didSelectRowAt:)*. Parametrem této funkce je proměnná typu *IndexPath*, která obsahuje informace, v jaké sekci a na kterém řádku bylo v tabulce kliknuto.

Jelikož má po kliknutí na buňku v tabulce dojít k přesměrování na detail vodoměru, je zobrazen indikátor probíhající aktivity a je uživateli zamezena interakce s tabulkou. Zamezení je z důvodu, aby uživatel neprovedl více kliknutí na buňku, čímž by došlo k více požadavkům o přesměrování a aplikace by přesun do detailu provedla tolikrát, kolikrát uživatel klikl.

```
view.showBlurLoader()
tableView.isUserInteractionEnabled = false
```

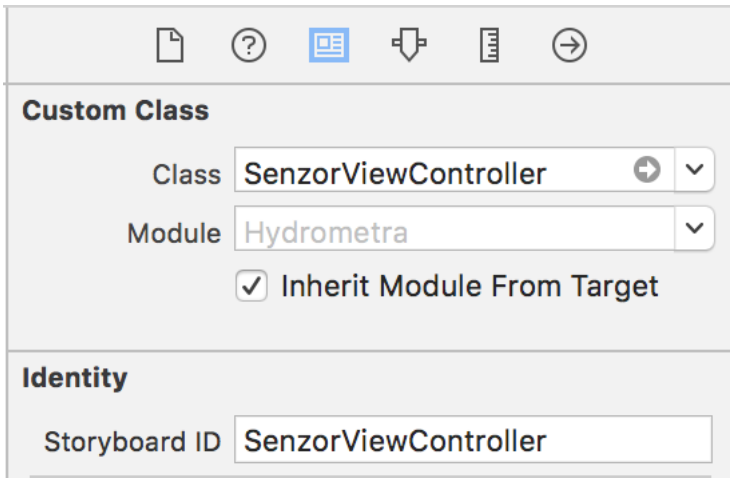
Ukázka kódu 29 – zobrazení efektu rozmazání a zamezení interakce

Poté je nutné předat vybraný senzor oknu, které zobrazuje detail. V tomto případě autor vybral způsob inicializace atributu dané třídy. Toho docílí pomocí vytvoření instance pro manipulaci a pozdější zobrazení dané třídy. Aplikace je pro přehlednost rozdělena na více souborů typu **storyboard**. Tento soubor reprezentuje grafické rozhraní aplikace. Pro zobrazení konkrétního okna, v grafickém rozhraní, je nutné provést zobrazení na konkrétním grafickém rozhraní. Toho lze docílit pomocí vytvoření instance s názvem daného rozhraní – *UIStoryboard(name: "SenzorDetail")*. S tím však přichází problém. Pokud si bude vývojář přát změnit název grafického rozhraní, musí tento název upravit v celé aplikaci, kde je využíváno. Autor si proto vytvořil rozšíření *UIStoryboard*, ve kterém si nadefinuje atributy tohoto rozšíření a ty poté používá v kódu. V případě změny názvu grafického rozhraní, je název změněn pouze v tomto rozšíření a není nutné procházet kód celé aplikace.

```swift
extension UIStoryboard {

    static let wizard                   = UIStoryboard(name: "Wizard")
    static let main                     = UIStoryboard(name: "Main")
    static let collection               = UIStoryboard(name: "Collection")
    static let edit                     = UIStoryboard(name: "SenzorEdit")
    static let service                  = UIStoryboard(name: "Service")
    static let menu                     = UIStoryboard(name: "SideMenu")
    static let detail                   = UIStoryboard(name: "SenzorDetail")
    static let valve                    = UIStoryboard(name: "ValveState")
}

extension UIStoryboard {

    convenience init(name: String) {
        self.init(name: name, bundle: nil)
    }

}
```

Obrázek 15 – Implementace rozšíření komponenty *UIStoryboard, zdroj: vlastní*

Instance třídy je vytvořena s identifikátorem okna v grafickém rozhraní, které je nutné nadefinovat, aby byl umožněn přístup k tomuto oknu. Tento identifikátor lze v editoru grafického rozhraní najít pod názvem **Storyboard ID**.



Obrázek 16 – Definování identifikátoru okna, zdroj: vlastní

Poté stačí nastavit objektu atribut *senzor* na uživatelem zvolený senzor, ke kterému je přistoupeno pomocí parametru *indexPath*.

```
let storyboard: UIStoryboard = .detail
let viewController = storyboard.
    instantiateViewController(withIdentifier: "SenzorViewController") as! SenzorViewController
viewController.senzor = self.senzors[indexPath.row]
```

Ukázka kódu 30 – nastavení atributu mimo aktuální třídu

V tento moment je vše potřebné nastaveno, proto je interakce s tabulkou opět aktivována a dojde k zobrazení detailu pomocí funkce *show()*. Parametr, který je obsažen v této funkci, je požadovaná třída spravující následující okno.

```
self.tableView.isUserInteractionEnabled = true
self.show(viewController, sender: nil)
```

Ukázka kódu 31 – obnovení interakce a zobrazení požadovaného okna

3.3 Detail senzoru

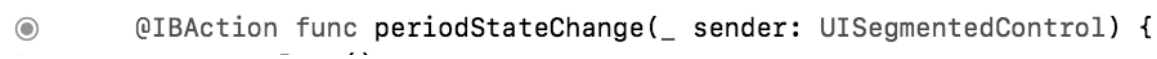
Okno detailu je nejobsáhlejším oknem aplikace. Obsahuje spoustu komponentů, komunikaci mezi nimi a spoustu operací, které jsou na ně vázány. V tomto okně je využit *Pod* pro vykreslování grafů, vytvořený dvojicí autorů – Daniel Cohen Gindi a Philipp Jahoda, který v této aplikace vykresluje koláčový graf, pro přehled aktuální denní spotřeby a měsíční spotřeby, a sloupcový graf, pro zvolené období uživatelem – den, měsíc a rok.

3.3.1 Změna zkoumaného období

Zkoumané období lze vyčíst z komponenty *UISegmentedControl*, díky které lze přepínat mezi spotřebou za určitý den, měsíc a rok. Přepnutí se projeví ve změně hodnot sloupcového grafu a jeho popisem, kde je také ono období popsáno, včetně konkrétního dne/měsíce/roku, který je vybrán.

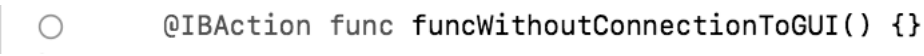
Změnu lze provést třemi způsoby – výběrem z komponenty *UISegmentedControl*, výběrem sloupce z grafu či pomocí tlačítka s ikonou „⇧“, která provede přechod o řád výš. To znamená, že pokud bude ve sloupcovém grafu vybrána měsíční spotřeba, dojde k zobrazení měsíční spotřeby ve vybraném měsíci. Změna je provedena stejným způsobem, pouze pomocí různých komponent.

Funkce, která zajišťuje zachycení interakci s komponentou *UISegmentedControl*, je akce této komponenty. Změna oproti obyčejné funkce je, že je implementována pomocí vytvoření tzn. akci *Outletu*. **Outlet** je reference komponenty do třídy, která je vytvořena skrze grafické rozhraní. Taková funkce pouze obsahuje v kódu identifikátor napojení do grafického rozhraní pomocí vyplněného modifikátoru *@IBAction*.



Obrázek 17 – funkce propojená s komponentou v GUI, zdroj: vlastní

Pokud kód obsahuje takovou funkci, která není napojena:



Obrázek 18 – funkce připravená pro napojení na GUI, zdroj: vlastní

Jak již bylo uvedeno výše, je možné přepínat mezi třemi stavy. Z toho důvodu autor implementoval přepínač, který na základě indexu vybraného segmentu změní zkoumané období.

```
switch sender.selectedSegmentIndex {
    case 0:
    case 1:
    case 2:
    default:
}
```

Ukázka kódu 32 – implementace přepínače na základě vybraného segmentu

V případě, že je vybrán například segment s indexem 0, tedy *case 0*, dojde k nastavení lokální proměnné, definované výčtový typem, na hodnotu, která reprezentuje denní spotřebu.

```
selectedType = .day
```

Ukázka kódu 33 – nastavení lokální proměnné na požadovaný typ

Nad sloupcovým grafem je komponenta *UILabel*, která informuje o datu aktuálně vizualizovaných hodnot. Každý segment má vlastní formát, proto je nutné při změně období tento datum přeformátovat. Na základě formátu je poté z lokální proměnné toto datum vytvořeno a vloženo do grafické komponenty. Nakonec je nastaven titulek příslušný k danému grafu. Tento text je lokalizovaný a pomocí klíče, který je spojený k hodnotě, je nastavena hodnota příslušná jazyku aplikace. Aplikace rozeznává dva jazyky – anglický a český.

```
dateFormatter.dateFormat = "dd. MM. YYYY"
let date: String = dateFormatter.string(from: storedDate)
tfDatePicker.text = date
labelBarChart.text = "DETAIL_BARCHART_DESCRIPTION_DAY".localized
```

Ukázka kódu 34 – nastavení popisných hodnot nad sloupcovým grafem

Následně je pomocí statické třídy *ConsumptionAPI* a její funkce *requestConsumption()* kontaktováno REST API s požadavkem o zaslání spotřeby za aktuální den. Aby byla spotřeba k aktuálnímu dni, je potřeba definovat parametr `offset` číslem 0. Pokud by byla například zadáno číslo -1, REST API zašle spotřebu za předchozí den. Dále je v parametru zasláno identifikační číslo senzoru, pro který chceme získat informace o spotřebě. Pomocným parametrem je období spotřeby, aby byl proveden požadavek právě pro denní spotřebu.

```
requestConsumption(id: senzor.idHw, offset: 0, timeUnit: .day)
```

Ukázka kódu 35 – zavolání funkce pro zjištění spotřeby za požadované období

Po přijetí hodnot z databáze, je upraven atribut aktuálně vybraného senzoru *consumptionDay*. Z tohoto atributu bude poté graf číst data pro naplnění svých hodnot k vizualizaci. Upravení je provedeno v uložených senzorech v pomocné třídě *VodomerData*, ze kterých si poté třída, naplňující graf, převezme hodnoty. Naplnění atributu, pro vybraný senzor, je pomocí funkce,

která vyhledá první vyskytující objekt v poli za definované podmínky – *first(where:)*. Tato podmínka je provedena porovnáním identifikačních čísel vybraného senzoru a zkoumaného objektu v poli.

```
VodomerData.shared.senzors.first(where: { $0.idHw == senzor.idHw }).consumptionDay
```

Ukázka kódu 36 – zjištění odpovídajícího senzoru na základě identifikačního čísla

Poté dojde k aktualizování sloupcového grafu pro vykreslení nově získaných hodnot.

```
self.updateBarChart()
```

Ukázka kódu 37 – zavolání pomocné funkce pro znovu vykreslení sloupcového grafu

V případě neúspěchu je zobrazeno upozornění o výskytu problému při komunikaci.

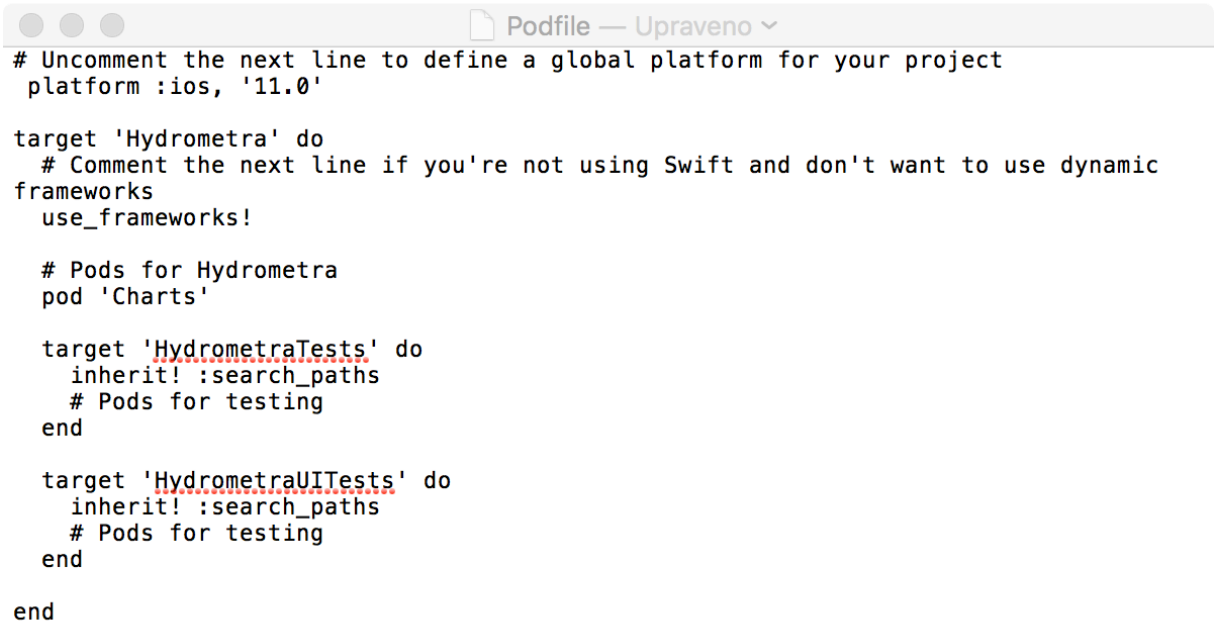
```
self.present(
    ReachabilityManager.shared.noConnectionAlert(showLogin: false),
    animated: true,
    completion: nil
)
```

Ukázka kódu 38 – zobrazení upozornění na absenci internetového připojení

3.3.2 Vykreslení grafu

Pro vykreslení grafu autor využil projekt **Charts**, v aktuální verzi 3.1.1, jako doplňující knihovnu. Tu vytvořil Daniel Cohen Gindi, který se inspiroval projektem, který vytvořil Philipp Jahoda. Ten provedl první řešení pro vykreslování grafů pro zařízení s operačním systémem Android.

Instalace této knihovny je pomocí manažeru závislostí CocoaPods. Popis tohoto manažeru je v kapitole 2.2. Poté stačí přejít do adresáře projektu a upravit soubor *Podfile*, který obsahuje seznam projektů, které budou díky manažeru importovány do hlavního projektu. Pro používání v kódu poté stačí do hlavičky u dané třídy doplnit *import Charts.*



Obrázek 19 – Importování projektu pomocí Cocoa Touch, zdroj: vlastní

Pro vytvoření koláčové grafu je nutné udělat tři základní kroky – vytvořit datový vstup, tzn. *PieChartDataEntry*, následně proměnnou typu *PieChartDataSet*, která představuje skupinu datových vstupů, a nadefinovat jeho vizuální stránku.

Pro vytvoření grafu s denní spotřebou, je nutné vypočítat si celkovou spotřebu za dané období, které má být vizualizováno. Pro výpočet si autor implementoval pomocnou funkci *calcConsumption()*, které je předán typ období, pro které má být spotřeba vypočítána. Výpočet je proveden pomocí cyklu, který projde uložené záznamy o spotřebě za dané období, v záznamu vyhledá hodnotu a tu přičte do proměnné, která je poté funkcí navrácena. Tato hodnota je poté přidána do pole všech datových vstupů, které jsou v grafu požadovány.

```
func calcConsumption(_ type: TimeUnit, for senzor: Senzor) -> Double {
    var consumption = 0.0

    for rawVal in Utils.shared.typeOfCons(type, for: senzor) {
        let val = rawVal as! NSDictionary
        let value = val["value"] as! Double
        consumption += value
     }
    return consumption
}
```

Ukázka kódu 39 – implementace pomocné funkce pro výpočet celkové spotřeby

```
var pieChartEntry = [PieChartDataEntry]()
let consumption = Utils.shared.calcConsumption(type, for: senzor)
let val = PieChartDataEntry(value: consumption, label: "")
pieChartEntry.append(val)
```

Ukázka kódu 40 – nastavení datového vstupu koláčového grafu

U koláčového grafu je požadováno, aby byla aktuální spotřeba vizualizována společně s limitní hodnotou k danému období. Limit je realizován jako klasický datový vstup. Hodnota je vypočítána na základě atributem dané hodnoty pro limit za dané období a od té je odečtena vypočítaná spotřeba.

```
limitVal = senzor.limitDay! - consumption
limit = PieChartDataEntry(value: limitVal < 0 ? 0 : limitVal, label: "")
pieChartEntry.append(limit)
```

Ukázka kódu 41 – výpočet datového vstupu indikující nastavený limit

Z vytvořených datových vstupů je poté vytvořen datový set. U něho je možné nadefinovat, jak bude prezentován v grafu. Grafické rozhraní obsahuje komponentu *UILabel*, ve které bude uvedena hodnota aktuální spotřeby. Proto autor zamezil vykreslování hodnoty v rámci datového setu. Poté nastavil barevnou interpretaci pro datové vstupy. Přiřazení barev

k datovému vstupu je dáno pořadím, v jakém byly do pole přidány. Pro dodržení barevné palety zvolil autor, pro jeden datový vstup, hlavní barvu aplikace.

```
let dataSet = PieChartDataSet(values: pieChartEntry, label: "DETAIL_PIECHART_DESCRIPTION".localized)

dataSet.drawValuesEnabled = false
dataSet.colors = [.darkGray, .primaryColor]
```

Ukázka kódu 42 – nastavení barvy datového vstupu koláčového grafu

Pro naplnění grafu daty je nutné ze získaného setu dat vytvořit objekt typu *PieChartData*. Tímto objektem poté nastavíme atribut grafu a graf je připraven pro vykreslení.

```
let data = PieChartData()
data.addDataSet(dataSet)
pieChart.data = data
```

Ukázka kódu 43 – nastavení dat koláčovému grafu

Posledním, volitelným, krokem je nastavení vizuální stránky koláčového grafu. Tento graf umožňuje vykreslení určité hodnoty do středu onoho grafu. Autor zvolil tento prostor jako vhodné místo pro výpočet procentuální části k limitu.

```
let percent = (100 / (type == .day ? senzor.limitDay : senzor.limitMonth)!) * consumption
let attribute = [
    NSAttributedStringKey.font: UIFont(name: "Arial", size: 15.0)!
]
pieChart.centerAttributedText = NSAttributedString(
    string: "\(String(format: "%.1f", percent)) %",
    attributes: attribute)
```

Ukázka kódu 44 – definice vizuálních prvků koláčového grafu

Dále nadefinoval vykreslení grafu pomocí animace. Parametrem funkce *animate()*, je doba, po kterou se bude animace vykonávat. Jednotkou této doby jsou sekundy.

```
pieChart.animate(yAxisDuration: 2)
```

Ukázka kódu 45 – animované vykreslení koláčového grafu

Autor si poté nadefinoval pomocnou funkci *pieChartAppereance()*, která obsahuje vizualizaci prvků, které nejsou spojeny s daty. Tato funkce například definuje zamezení vykreslování legendy grafu.

Zdroj: [26]

3.4 Menu

Pro implementaci autor zvolil již vytvořenou komponentu SideMenu od vývojáře Jon Kent, která je veřejně k dispozici pod MIT licencí. Tato komponenta je do projektu zahrnuta pomocí manažeru závislostí CocoaPods, který je popsán v kapitole 2.2.

Zdroj: [6]

3.4.1 Implementace

Pro implementaci je nutné provést importování projektu v souboru, kde bude použit – *import SideMenu*. Dále je nutné nadefinovat chování této komponenty. Toto nastavení je nutné provést pouze jednou, proto se tato implementace provede ve funkci *viewDidLoad()*, která se v rámci dané třídy spouští pouze při inicializaci této třídy. Poté, pokud zůstane v hierarchii, již není tato funkce volána.

Je nutné definovat, zda má být menu umístěno v levé nebo pravé části okna. Autor zvolil druhou variantu, tedy menu se nachází na pravé straně okna.

```
SideMenuManager.default.menuRightNavigationController = viewController
```

Ukázka kódu 46 – implementace menu

Autor též zvolil volitelnou funkcionalitu a tou je zobrazení menu, pokud uživatel provede přejetí prstem z kraje obrazovky, pro kterou zvolil umístění menu, k protilehlé straně.

```
SideMenuManager.default.menuAddScreenEdgePanGesturesToPresent(toView: self.navigationController!.view)
```

Ukázka kódu 47 – nastavení gesta pro zobrazení menu

Poté již stačí, v rámci nadefinované akce, kterou aplikace zachytne, toto menu zobrazit. Jelikož je vše předem připraveno od vývojáře, stačí pomocí funkce *present()*, která je nativní funkcí každého okna pro zobrazení účelového okna, menu zobrazit. Syntax *if let* zajistí, že kód podmíněného bloku bude proveden pouze za podmínek, že lze do proměnné hodnotu přiřadit. To znamená, že hodnota musí existovat. V dalším kroku je pomocí mechanismu *NotificationCenter* zaslána notifikace, ve které je obsažen objekt senzoru, pro další práci v rámci položek v menu. Tento mechanismus je popsán v kapitole 3.4.3.

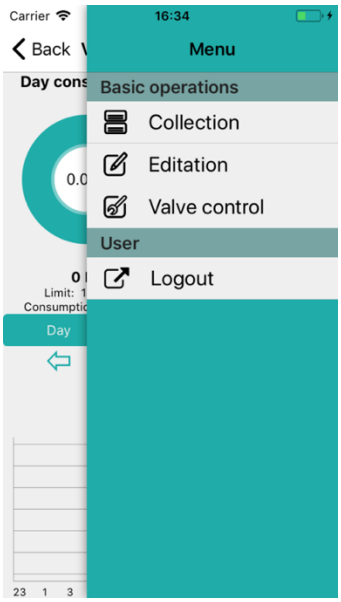
```
@IBAction func btnMenu(_ sender: UIBarButtonItem) {
    if let menu = SideMenuManager.defaultManager.menuRightNavigationController {
        present(menu, animated: true) {
        NotificationCenter.default.post(
            name: .menuWillAppear,
            object: nil,
            userInfo: ["senzor": self.senzor]
        )
    }
}
```

Ukázka kódu 48 – implementace události po kliknutí na tlačítko Menu

Zdroj: [6]

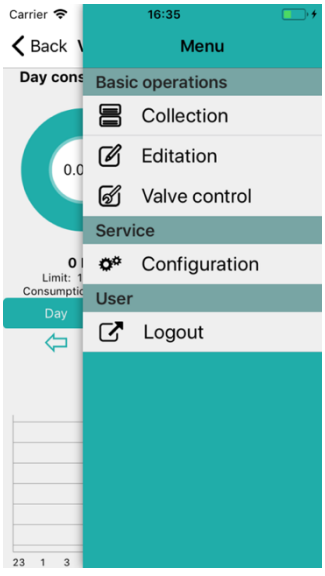
3.4.2 Role uživatele

Jak již bylo psáno dříve, aplikace umožňuje dvojí přístup na základě předem definovaných rolí. Pro lepší manipulaci autor implementoval výčtový typ *UserRole*, který obsahuje dva stavy – `admin a user`. Poté Toto se projeví právě v menu. Pokud je uživatel přihlášen jako uživatel, uvidí v první sekci základní položky a v druhé položku pro odhlášení.



Obrázek 20 – dostupné Menu položky pro uživatelská práva, zdroj: vlastní

Pokud je přihlášen uživatel s rolí administrátora, je mu navíc zobrazena položka *Konfigurace*, kde je možné například upravit celkovou spotřebu senzoru.



Obrázek 21 – dostupné Menu položky pro administrátorská práva, zdroj: vlastní

Dekódování těchto rolí z JWT je popsáno v kapitole 3.1.1. Autor do třídy, která zajišťuje práci s položkami menu, nadefinoval privátní atribut *role*, do které, z lokální databáze, přiřadí

uloženou hodnotu. Programovací jazyk Swift nevyžaduje, aby vývojář přímo definoval datový typ. Ten je vyhodnocen na základě typu hodnoty, která je proměnné přiřazena. V tomto případě byl do lokální databáze, v předchozích krocích, uložen řetězec, proto se explicitně nastaví datový typ String.

```
private var role = UserDefaults.standard.userRole
```

Ukázka kódu 49 – uložení role z lokální databáze do privátní proměnné

Na základě této skutečnosti, je poté v kódu pomocí podmínky rozhodnuto, zda hodnota v proměnné nabývá jedné či druhé varianty. Například u funkce, která určuje počet sekcí v tabulce, pomocí ternárního operátoru. Ta v případě, že je role přihlášeného uživatele typu „user“, vrátí dvě sekce.

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return role == .user ? 2 : 3
}
```

Ukázka kódu 50 – implementace funkce pro definici počtu sekcí v tabulce Menu

3.4.3 Dočasné nastavení senzoru

Pro přenášení dat mezi pohledy lze využít různých způsobů. Jedním z nich lze vyslat notifikaci v určité funkci a tu na jiném místě přijmout. Notifikace obsahuje atribut *userInfo,* kam si lze uložit určitá data a ty pomocí notifikace předat. Právě tento způsob autor zvolit pro předání vybraného senzoru třídě, která spravuje menu. V té uživatel může vybrat takové okno, které právě tento senzor potřebuje.

Pro vytvoření takové notifikace je nutné nadefinovat si název, který bude zachytáván v rámci aplikace. Tohoto lze docílit přímo v konstruktoru notifikace. Avšak dle konvence a praktičnosti je lepší nadefinovat si atribut v rozšíření *Notification.Name* pro případ, že by autor chtěl změnit název notifikace. Pokud by toto neučinil, musel by, v případě změny, projít celou aplikaci a upravit ho.

```
extension Notification.Name {
    static var menuWillAppear = Notification.Name("menuWillAppear")
}
```

Ukázka kódu 51 – implementace rozšíření třídy *Notification*

Pro notifikaci je nutné nadefinovat dva směry – vysílač a přijímač. Vysílač již byl uveden v kapitole 3.4.1 – v moment, kdy autor zobrazuje uživateli menu, vysílá notifikaci, která sebou, nese vybraný senzor.

```
present(menu, animated: true) {
    NotificationCenter.default.post(
        name: .menuWillAppear,
        object: nil,
        userInfo: ["senzor": self.senzor])
}
```

Ukázka kódu 52 – zobrazení menu a vyslání notifikace nesoucí vybraný senzor

Přijímač se definuje pomocí funkce *addObserver*, kde je prvním parametrem objekt, který v rámci třídy bude naslouchat notifikaci. V tomto případě se jedná o celou třídu *SideMenuTableViewController*, proto se uvede do parametru klíčové slovo *self*.

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(captureSelectedSenzor(notification:)),
    name: .menuWillAppear,
    object: nil
)
```

Ukázka kódu 53 – definice naslouchající události notifikace

Dalším parametrem je akce, která se při výskytu provede. Je možné vložit řetězec představující název funkce v rámci třídy nebo pomocí selektoru zavolat danou funkci. Toto volání je pozůstatek jazyka Objective-C. Pokud se před modifikátor funkce vloží speciální modifikátor

@objc, lze využít přístup přímého volání funkce. Pokud by tento modifikátor nebyl použit, kompilátor nás na to sám upozorní a nedovolí projekt sestavit ani spustit.

Akce, která je vykonána na základě notifikace, musí obsahovat v parametru objekt notifikace. Ta obsahuje doplňující informace, kde je předáván vybraný senzor. Ačkoli víme, že jsme daný objekt v notifikaci poslali, je nutné ošetřit pomocí syntaxe *if let*, že opravdu hodnota existuje. Poté je nastavena lokální proměnná na notifikací zaslaný senzor.

```
@objc private func captureSelectedSenzor(notification: Notification) {
    if let capturedSenzor = notification.userInfo?["senzor"] as? Senzor {
        self.senzor = capturedSenzor
    }
}
```

Ukázka kódu 54 – implementace funkce na základě vyslané události

Druhý možný přístup je nastavení veřejného atributu třídy, která bude zobrazena, z aktuální třídy. Toho je docíleno pomocí vytvoření instance následující třídy a přímé nastavení námi vybraného senzoru.

```
let viewController = storyboard.
    instantiateViewController(withIdentifier: "ValveStateViewController") as! ValveStateViewController
```

Ukázka kódu 55 – inicializace třídy obsluhující okno pro správu ventilu

Poté již stačí jen okno zobrazit pomocí funkce *show()*.

```
viewController.senzor = self.senzor
show(viewController, sender: nil)
```

Ukázka kódu 56 – inicializace atributu v následující třídě

3.5 Editace senzoru

Před odesláním upravených hodnot dochází k jejich základnímu validování. Základní validování představuje u textových řetězců nenulový řetězec a u číselných naplnění kladnou hodnotou. Pro validaci autor vytvořil pomocnou funkci, která projde všechna textová pole a navrátí booleovskou hodnotu na základě validních či nevalidních vstupů. U okna editace

senzoru dochází k validaci například textového pole s názvem senzoru, cenou za m^3 či denním limitem.

```
if tfSenzorName.text == nil {
    return false
}
if tfPricePerM3.text == nil || priceVal < 0 {
    return false
}
if limitDayVal < 0 {
    return false
}
```

Ukázka kódu 57 – validace hodnot z textových polí

Po úspěšné validaci je uživateli zobrazeno upozornění, zda si opravdu přeje provést editaci. Při potvrzení editace je vytvořen objekt typu *Parameters*, díky kterému budou odeslány hodnoty jako parametr požadavku. Tento objekt představuje pole pro data typu klíč-hodnota. Tato data obsahující hodnoty z textových polí.

```
let parameters: Parameters = [
    "nameHw": self.tfSenzorName.text ?? "",
    "pricePerM3": self.tfPricePerM3.text ?? "",
    "implPerLit": self.senzor.implPerLit ?? 0,
    "countStop": self.senzor.countStop ?? 0,
    "countStopNight": self.senzor.countStopNight ?? 0,
    "nightStartHour": self.tfNightStartHour.text ?? "",
    "nightEndHour": self.tfNightEndHour.text ?? "",
    "limitDay": self.tfLimitDay.text ?? "",
    "limitMonth": self.tfLimitMonth.text ?? ""
]
```

Ukázka kódu 58 – nastavení parametrů pro požadavek HTTP

Tento objekt je poté zaslán jako parametr ve funkci *updateSenzor()*, která je implementována ve statické třídě *SenzorAPI*. Při úspěšném požadavku dojde k nastavení hodnot, které byly požadavkem odeslány, i pro aktuálně vybraný senzor. Současně dojde k přesměrování na detail

senzoru, který byl upraven. V případě neúspěchu dojde k zobrazení upozornění na výskyt chyby.

```
let storyboard: UIStoryboard = .detail
let viewController = storyboard.
    instantiateViewController(withIdentifier: "SenzorViewController") as! SenzorViewController
viewController.senzor = self.senzor

self.show(viewController, sender: nil)
```

Ukázka kódu 59 – inicializace a následné zobrazení okna detailu senzoru

3.6 Změna stavu uzávěru

Změna stavu, z pohledu uživatele, může nabývat dvou hodnot – uzavřít či otevřít uzávěr. Při kliknutí na příslušné tlačítko dojde nejprve ke zjištění, zda je k dispozici připojení k internetu. Pokud není, je zobrazeno upozornění na tuto skutečnost. Poté následuje změna stavu dle aktuálního stavu senzoru. Pokud je stav senzoru otevřený, dojde k přepnutí na stav „Požadavek k uzavření“. V opačném případě dojde k nastavení stavu na „Požadavek k uzavření“. Pro každý stav je definován vizuální stav tlačítka. Změna je provedena pomocí pomocné privátní funkce *btnAppereanceForState(_ state:).*

Parametrem této funkce je stav, pro který má být vizuální stav změněn. Dochází ke změně barvy pozadí, textu a zamezení uživatelské interakce s tlačítkem.

```
btnState.backgroundColor = .orange
btnState.setTitle("VALVE_CONTROL_BUTTON_TITLE_CLOSING".localized, for: .normal)
btnState.isEnabled = false
```

Ukázka kódu 60 – nastavení vizuální stránky tlačítka pro změnu stavu ventilu

Následně dojde k nastavení ostatních grafických prvků okna – komponenty *UILabel*, které obsahují informace o aktuálním stavu senzoru, kdo stav změnil a čas, kdy byla změna provedena.

```
self.labelState.text = Utils.shared.convertStateToString(4)
self.labelChanger.text = "Admin"
self.labelChangeDate.text = Utils.shared
    .formatDate(
        Utils.shared.getCurrentDateTime(),
        format: "dd.\u{00a0}MM.\u{00a0}YYYY HH:mm:ss"
    )
```

Ukázka kódu 61 – nastavení komponent informující o stavu ventilu

Statickou třídou *SenzorAPI* a funkcí *putValveState()* je zaslán požadavek o změnu stavu. Stav je odeslán parametrem této funkce. Při kladné odezvě ze strany serveru, dojde k aktualizaci stavu na základě parametrem zaslaného stavu, identifikačního čísla uživatele, který změnu provedl a atributu *stateModifiedDate*, který nese datum a čas změny stavu. To je zajištěno pomocí funkce *getCurrentDateTime()*, která naformátuje aktuální datum a čas dle určitého formátu.

```
senzor.state = state
senzor.stateModifierUserId = 1
senzor.stateModifiedDate = Utils.shared.getCurrentDateTime()
```

Ukázka kódu 62 – nastavení atributů senzoru na základě změny stavu ventilu

Předchozí kroky zajistí zaslání požadavku o změnu stavu a aktualizaci vizuální stránky pro tento stav. Autor implementoval funkci, která periodicky vysílá požadavek na server s určitým zpožděním, který zjišťuje stav senzoru. Pokud je senzor stále ve stavu požadavku o změnu stavu, funkce zavolá sama sebe responzivním způsobem. Pokud dojde k finální změně stavu, změní se vizuální stránka okna a je opět umožněna interakce s tlačítkem pro změnu stavu.

```
switch self.senzor.state {
// Otevřený
case 1:
self.senzor.state = senzor.state
self.btnAppereanceForState(self.senzor.state!)
self.labelState.text = Utils.shared.convertStateToString(senzor.state)
self.labelChangeDate.text = Utils.shared
    .formatDate(
        Utils.shared.getCurrentDateTime(),
        format: "dd.\u{00a0}MM.\u{00a0}YYYY HH:mm:ss")

return

// Čeká na oteření
case 2:
self.checkStateFor(self.senzor.state!)
```

Ukázka kódu 63 – přepínač zajišťující zjištění požadované změny stavu

3.7 Pomocné a doplňující funkce

Pro ulehčení práce si autor přidal pomocné třídy, které jsou využity například pro funkce, které jsou používány na více místech, určitým způsobem formátují data nebo rozšiřují komponenty jazyka Swift.

3.7.1 Zobrazení indikátoru probíhající aktivity

Programovací jazyk Swift umožňuje doimplementovat do již nativně definovaného prvku, například pro barvy *UIColor*, vlastní funkce či atributy. Tohoto je docíleno pomocí tzn. **Extension**. Programátor si takto může snadno nadefinovat hlavní barvy, které budou v aplikaci využívány.

Autor si takto vytvořil rozšíření pro *UIView*, ve kterém si implementoval funkci pro zobrazení indikátoru probíhající aktivity, tzn. **Activity Indicator**. Navíc je doprovázen efektem rozmazaného pozadí. Ten je využíván například v případě přechodu z přihlašovací obrazovky na kolekci dostupných senzorů. Indikátor je spuštěn při inicializaci požadavku pro získání senzorů na server.

Nejprve je nutné vytvořit si instanci komponenty *UIBlurEffect*, které se do parametru předává požadovaný styl. Tento efekt je poté zasazen do komponenty *UIVisualEffectView*, kde je parametrem vytvořený efekt.

```
let blurEffect = UIBlurEffect(style: .light)
let blurEffectView = UIVisualEffectView(effect: blurEffect)
```

Ukázka kódu 64 – vytvoření efektu rozmazání

Pro docílení rozmazání celého okna, se musí nastavit velikost vytvořené komponenty na velikost okna námi rozšiřovaného okna. Na dalším řádku je nastavení atributu automatického rozšiřování. Programovací jazyk Swift umožňuje nastavení atributu pomocí tečkové anotace. Jelikož je každá z očekávaných hodnot typu *UIViewAutoresizing*, Swift skrze tečkovou anotaci nabídne hodnotu tohoto výčtového typu a není nutné psát kompletní znění, které by v tomto případě představovalo:

```
blurEffectView.frame = self.bounds
blurEffectView.autoresizingMask = [
    UIViewAutoresizing.flexibleWidth,
    UIViewAutoresizing.flexibleHeight
]
```

Ukázka kódu 65 – nastavení atributu efektu rozmazání klasickým způsobem

Tečková anotace umožní zápis zkrátit na:

```
blurEffectView.frame = self.bounds
blurEffectView.autoresizingMask = [
    .flexibleWidth,
    .flexibleHeight
]
```

Ukázka kódu 66 – nastavení atributu efektu rozmazání tečkovou anotací

V neposlední řadě se provede vytvoření instance komponenty *UIActivityIndicatorView*, která obsahuje parametrem definovaný typ. Poté je vycentrována na střed a spuštěna animace pomocí komponentou definované funkce *startAnimating()*.

```
let activityIndicator = UIActivityIndicatorView(activityIndicatorStyle: .white)
activityIndicator.center = self.center
activityIndicator.startAnimating()
```

Ukázka kódu 67 – vytvoření komponenty pro indikaci probíhající aktivity

Posledním krokem je vložení indikátoru aktivity do okna, které obsahuje již efekt rozmazání obrazovky. Následně musí být toto okno vloženo do námi rozšiřovaného okna jako jeho potomka.

```
blurEffectView.contentView.addSubview(activityIndicator)
self.addSubview(blurEffectView)
```

Ukázka kódu 68 – zobrazení indikátoru probíhající aktivity

ZÁVĚR

Práce se se v teoretické části zaměřuje na čtvrtou průmyslovou revoluci. Detailněji rozebírá internet věcí v domácnostech a v průmyslu. Postupně přechází ke konkrétnímu řešení chytrého monitorování spotřeby vody. Popisuje existující chytrý vodoměr a aplikaci, která byla cílem praktické části, zajišťující interaktivní rozhraní pro uživatele. Poté byly popsány technologie umožňující vznik aplikace eVodoměr pro platformu iOS. Tento cíl se podařilo splnit dle specifikace, která byla definována před samotnou implementací. Určité aspekty této specifikace se postupem času rozšiřovaly k docílení co největší efektivnosti.

Výsledkem je aplikace umožňující vizualizaci naměřených dat pomocí sloupcového a koláčových grafů. Tato data jsou též reprezentována cenově tak, aby byla uživatelsky co nejjasnější. Navíc umožňuje základní konfiguraci parametrů vodoměru. Aplikace, k docílení vizualizace pomocí grafů, využívá knihovnu Charts, dostupnou skrze manažer závislostí CocoaPods. Pro komunikaci s databází využívá REST API, které obsahuje předefinované funkce pro získání dat z databáze či jejich úpravu.

Při vývoji jsem se soustředil na vývoj pro zařízení iPhone 7, kde jsem využil architekturu MVC. Návrh grafického rozhraní mi byl do jisté míry nastíněn, avšak finální podoba byla v mých rukách. Velikou prioritou pro mne bylo udělat grafické rozhraní takovým způsobem, aby bylo co nejvíce uživatelsky přívětivé. Při vývoji jsem se seznámil s některými komponentami frameworku UIKit a znatelně jsem rozšířil své znalosti v programovacím jazykem Swift a s vývojem pro platformu iOS.

V dalším vývoji je možné zaměřit se na zařízení iPad. Již teď je možné aplikaci na tomto zařízení využívat, avšak iPad umožňuje implementaci grafického rozhraní speciálně pro jeho rozměry při horizontální orientaci, tzn. „landscape“. Pro plnohodnotné uplatnění této aplikace na trhu mobilních aplikací je nutné provést implementaci stejné aplikace na platformu Android. Toho by šlo docílit vývojem nativní aplikace pro Android nebo provést implementaci pomocí frameworku Xamarin, který umožňuje vývoj multiplatformních aplikací. Výstupem by tedy byla aplikace pro oba zmíněné operační systémy.

POUŽITÁ LITERATURA

[1] MALÝ, Martin. REST: architektura pro webové API. In: *Zdroják* [online]. (Česká Republika): Zdroják, 2009 [cit. 2018-04-12]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

[2] CASTELLS, Pau a David JONES. GB-wide smart meter roll out for the domestic sector. In: *Ofgem*[online]. London: Ofgem, 2010, 27-07-2010 [cit. 2018-04-12]. Dostupné z: https://www.ofgem.gov.uk/ofgem-publications/63551/decc-impact-assessment-domestic.pdf

[3] Karlis, D. Vasdekis, V. G. S. Banti, M. (2009) Hetero-sciatic semi-parametric models for domestic water consumption aggregated data, *Environ Ecol Stat*, 16, p.355–367

[4] Domestic water consumption monitoring and behaviour intervention by employing the internet of things technologies. *Procedia Computer Science* [online]. 2017, 2017(111), 367-375 [cit. 2018-04-12]. ISSN ISSN: 1877-0509. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1877050917312097

[5] Chytré monitorování spotřeby – cesta k velkým úsporám energií. In: *EnviWeb* [online]. Česká Republika: EnviWeb, 2011, 15.11.2011 [cit. 2018-04-12]. Dostupné z: http://www.enviweb.cz/88927

[6] KENT, Jon. SideMenu. In: *Github* [online]. San Francisco: Github, 2007- [cit. 2018-04-14]. Dostupné z: https://github.com/jonkykong/SideMenu

[7] JSON Web Token [online]. Bellevue: Auth0, 2018 [cit. 2018-04-14]. Dostupné z: https://jwt.io/introduction/

[8] POHANKA, Pavel. Internet věcí. In: *I2ot* [online]. Česká Republika: Pohanka, 2017 [cit. 2018-04-15]. Dostupné z: http://i2ot.eu/internet-of-things/

[9] MORAVEC, Petr. Google Brillo: Nový operační systém pro internet věcí. In: *Mobilizujeme*[online]. Česká Republika: Mobilizujeme, 2015 [cit. 2018-04-15]. Dostupné z: https://mobilizujeme.cz/clanky/google-brillo-novy-operacni-system-pro-internet-veci

[10] MCEWEN, Adrian; CASSIMALLY, Hakim. Designing the internet of things. John Wiley & Sons, 2013 [15-4-2018]. ISBN: 978-1-118-43062-0

[11] IoT devices (internet of things devices). In: *IoT Agenda* [online]. Atlanta: TechTarget, 2018 [cit. 2018-04-15]. Dostupné z: https://internetofthingsagenda.techtarget.com/definition/IoT-device

[12] Gartner [online]. Stamford: Gartner, 2018 [cit. 2018-04-15]. Dostupné z: https://www.gartner.com/

[13] Internet of Things For Dummies [online]. Hoboken: Wiley, 2017 [cit. 2018-04-15]. ISBN 978-1-119-34992-1. Dostupné z: http://media.wiley.com/assets/7348/78/Vol_1_9781119349891_Internet_of_Things_FD_QorvoSE.pdf

[14] SimpleCell [online]. Praha: SimpleCell, 2018 [cit. 2018-04-15]. Dostupné z: https://simplecell.eu/

[15] Sigfox [online]. Boston: Sigfox, 2018 [cit. 2018-04-15]. Dostupné z: https://www.sigfox.com/

[16] RŮŽIČKOVÁ, Veronika. Nová průmyslová revoluce – Průmysl 4.0. In: *Datamix* [online]. Olomouc: Datamix, 2017 [cit. 2018-04-15]. Dostupné z: http://www.datamix.eu/blog/nova-prumyslova-revoluce-prumysl-4-0/

[17] Hannover Messe 2017: Industrie 4.0 integrated – overview and news. In: *I-scoop* [online]. Belgie: I-scoop, 2017 [cit. 2018-04-15]. Dostupné z: https://www.i-scoop.eu/industry-4-0/hannover-messe-2017/

[18] Průmysl 4.0 (Industry 4.0). In: *ManagementMania.com* [online]. Wilmington (DE) 2011-2018, 15.06.2017 [cit. 15.04.2018]. Dostupné z: https://managementmania.com/cs/prumysl-40-industry-40

[19] What is smart metering?. In: *SMS Plc* [online]. Glasgow: SMS, 2018 [cit. 2018-04-15]. Dostupné z: https://www.sms-plc.com/insights/what-is-smart-metering/

[20] JAN, Fikejz. *Proposal of a smart water meter for detecting sudden water leakage*. Pardubice, 2018.

[21] MULLOY, Brian. *Web API Design: Crafting Interfaces that Developers Love*. London, 2014. Dostupné také z: https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf

[22] What is an API? (Application Programming Interface). In: *MuleSoft* [online]. San Francisco: MuleSoft, 2016 [cit. 2018-04-15]. Dostupné z: https://www.mulesoft.com/resources/api/what-is-an-api

[23] KOULA, Petr. *Hydrometra API* [online]. Česká Republika: Koula, 2017- [cit. 2018-04-15]. Dostupné z: http://petrkoula-001-site1.ctempurl.com/swagger/

[24] PERES, Rui. Model-View-Controller (MVC) in iOS: A Modern Approach. In: *Wenderlich* [online]. Virginia: Wenderlich, 2016 [cit. 2018-04-15]. Dostupné z: https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach

[25] *CocoaPods* [online]. CocoaPods, 2011 [cit. 2018-04-15]. Dostupné z: https://cocoapods.org/

[26] GINDI, Daniel Cohen. Charts. In: *GitHub* [online]. Izrael: Gindi, 2018 [cit. 2018-04-21]. Dostupné z: https://github.com/danielgindi/Charts

[27] *Concrete Sensors* [online]. Cambridge: Concrete Sensors, 2018 [cit. 2018-04-28]. Dostupné z: http://www.concretesensors.com

[28] FIERRO, Jordi. Clean Architecture in Django. In: *21 Buttons Engineering* [online]. Barcelona: 21 Buttons Engineering, 2017 [cit. 2018-04-05]. Dostupné z: https://engineering.21buttons.com/clean-architecture-in-django-d326a4ab86a9

PŘÍLOHY

PŘÍLOHA A – PŘILOŽENÉ CD

Přiložené CD obsahuje:

- Projekt aplikace,
- PDF verzi bakalářské práce