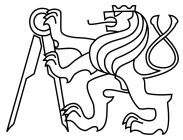
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Softwarové řešení pro správu GPS jednotek |
| **Student:** | Bc. Tomáš Neubauer |
| **Vedoucí:** | Ing. Pavel Dvořák |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce zimního semestru 2017/18 |

## Pokyny pro vypracování

Cílem práce je analýza, návrh a implementace softwarového řešení, které umožní vzdálenou hromadnou konfiguraci a správu GPS jednotek firmy Radium s.r.o.

Dílčí úkoly práce:
1. Vyjděte ze základních požadavků, jež jsou: uživatelské rozhraní použitelné na tabletech v terénu i v místech s horší rychlostí datové komunikace, asynchronní hromadné operace s GPS jednotkami, podpora uživatelských práv, historie operací nad GPS jednotkami, navázání na existující nízkoúrovňovou knihovnu pro komunikaci s GPS jednotkami.
2. Proveďte rešerši možných technologií a zvolte vhodné.
3. Navrhněte softwarovou architekturu kompletního řešení.
4. Návrh implementujte tak, že výsledkem bude framework zastřešující řízení a správu GPS jednotek (viz požadavky v bodu 1).
5. Na vhodně zvoleném dílčím projektu demonstrujte nasazení výše zmíněného frameworku.
6. Řešení zdokumentujte a zhodnoťte jeho přínosy.

## Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 29. září 2016

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

Diplomová práce

# Softwarové řešení pro správu GPS jednotek

***Bc. Tomáš Neubauer***

Vedoucí práce: Ing. Pavel Dvořák

10. ledna 2017

# Prohlášení

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

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

V Praze dne 10. ledna 2017 .....

### Odkaz na tuto práci

Neubauer, Tomáš. *Softwarové řešení pro správu GPS jednotek.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

# Abstrakt

Tato práce obsahuje popis kompletního procesu softwarového vývoje aplikace pro administraci GPS jednotek, včetně HTML 5 frontend knihovny sdílené mezi různými projekty. Serverová část aplikace je napsaná na platformě .NET, klientská část je tvořena v jazyce Typescript a frameworku Angular. Součástí práce je sofistikovaný systém verzování databáze, popis řízení projektu agilní technikou SCRUM a použití metodiky průběžné integrace (angl. Continuous Integration).

**Klíčová slova** .NET, Typescript, Angular, SCRUM, JIRA, Continuous Integration, MS SQL, Microsoft, Google, GPS, frontend, async, GIT, GIT Flow

# Abstract

The purpose of this thesis is to focus on the complete description of the software development process of application for the administration of GPS devices which includes a shared HTML front-end library among other projects. The server side of the application is written on a platform .NET, on the other hand the (fronted) client part is formed in Typescript language and Angular's framework. The thesis also includes a sophisticated system for database versioning and a description of project management by an agile technique called SCRUM supported by the methodology of continuous integration.

**Keywords** .NET, Typescript, Angular, SCRUM, JIRA, Continuous Integration, MS SQL, Microsoft, Google, GPS, frontend, async, GIT, GIT Flow

# Obsah

# Seznam obrázků

# Seznam tabulek

# Úvod

Moje práce se zabývá popisem kompletního procesu softwarového vývoje aplikace pro administraci GPS jednotek, včetně sdílené HTML 5 frontend knihovny pro společnost **Radium s.r.o.**

Tato společnost se věnuje zejména vývoji a výrobě systémů pro **Fleetmanagement**. Fleetmanagement je platforma pro sběr a vyhodnocení dat z vozidel, strojů a jiných prostředků.

V současnosti používaná hardwarová jednotka **CarPosition** ve své aktuálně čtvrté generaci se skládá z:

- GPS přijímač s interní nebo externí anténou
- GPRS/SMS modul
- Rozhraní (binární I/O, COM porty RS232/485, CAN sběrnice)
- Procesor ARM9 s podpůrnými obvody
- Paměti RAM a Flash
- Powermanagement

## Stávající konfigurační nástroj

Vozidlové jednotky **CarPosition** je potřeba konfigurovat a servisovat. Pro tyto účely byl vytvořen nástroj na konfiguraci jednotek, který slouží k editaci konfigurací, stahování dat pro servisní účely, aktualizace firmware atd. Aplikace je postavena na platformě Delphi jako desktopová aplikace instalovaná na koncové stanice. Aplikace nepoužívá databázi a z dnešního pohledu se jedná o zastaralý software.

## FleetwareMaster

Novou generací servisního nástroje je **FleetwareMaster**. Kompletní proces softwarového vývoje projektu je obsahem této práce.

Společnost Radium s.r.o. přechází u rodiny svých produktů na technologii **HTML5**. **FleetwareMaster** je průkopníkem této technologie ve společnosti. Cílem bylo sjednotit všechny produkty společnosti, jak technologicky, tak po stránce UX[1]. Proto byly společné obecné funkcionality vytvořeny v oddělené knihovně sdílené napříč celou společností. FleetwareMaster je prvním projektem používající tuto knihovnu. Tvorba této knihovny a samotný způsob sdílení kódů je také součástí této práce. V současné době je knihovna již využívána i jiným produktem firmy, webovou aplikací **FleetwareWeb 2**[2].

## Struktura práce

Na začátku této práce najdete specifikaci požadavků, které při zrodu projektu sloužily jako zadání. Dále je v kapitole **Volba technologie** popsáno období prototypování a výsledná volba technologie. Uvádím zde důvody a myšlenky, které mě nakonec vedly k volbě či zavrhnutí dané technologie. Poté jsem navrhl architekturu popsanou v kapitole **Návrh softwarové architektury**, která vycházela z vybrané technologie a specifikovaných požadavků.

V tuto chvíli se ustanovil tým a začala implementační část projektu popsaná v kapitole **Implementace**. Metodika řízení projektu a realizace implementační části je popsaná v třetí kapitole **Realizace a řízení projektu**. Součástí této práce je i systém nasazování a verzování databáze, který je spolu s popisem architektury DB obsahem kapitoly **Databáze**. V závěru naleznete stav splnění formálního zadání, moje shrnutí a několik statistik ze systémů GIT a JIRA .

V příloze jsou v kapitole **Zdrojové kódy** vloženy vybrané zdrojové kódy a nakonec kapitola **Ukázky z aplikace** obsahující nejzajímavější obrazovky klienta.

[1] **UX** - User Experience neboli uživatelský prožitek vyjadřuje celkovou míru spokojenosti uživatele s produktem

[2] **FleetwareWeb 2** - Webové prostředí pro sledování vozového parku

Kapitola 1

# Specifikace požadavků

V této kapitole popíši specifikaci požadavků, které u zrodu projektu tvořily zadání pro prototypy, volbu technologie a návrh architektury.

## 1.1 Cíle projektu

- Pilotní projekt přechodu firmy na technologii HTML5
- Sdílení frontend kódu mezi projekty firmy
- Centralizace správy hardwaru firmy

## 1.2 Funkční požadavky

- Napojení na komunikační knihovnu
- Čtení konfigurace
- Zápis konfigurace
- Kontrola kdo a co kde upravil
- Update firmware v jednotce
- Hromadná správa konfigurace v jednotkách
- Hromadný update firmwaru jednotek

## 1.3 Nefunkční požadavky

- Komfortní užívání na tabletu
- Responsivní design

- Nenáročné na kvalitu datového připojení v terénu

Aplikace je nástupcem desktopového jednovrstvého řešení. Při specifikaci jednotlivých scénářů užití jsme vycházeli ze zkušeností uživatelů předchozího řešení a snažili se navrhnout nové řešení podle jejich připomínek.

Kapitola 2

# Volba technologie

V kapitole volba technologie se zabývám zejména frontend části. Serverová a databázová část byla ovlivněna orientací společnosti na vývojovou platformu společnosti Microsoft. Platforma .NET je neustále rozvíjející se technologii a moje zkušenosti i zkušenosti kolegů s touto platformou na serveru jsou výborné a nebyla zde poptávka po změně. K rozhodnutí zůstalo pouze použití nejnovější multiplatformní technologie **.NET Core**.

Oproti tomu, výběr technologie klienta byl velkou výzvou. Zkušenosti firmy s webovým vývojem byly malé a přitom bylo jasné, že výběr ovlivní postupně všechny projekty společnosti na spoustu let dopředu.

## 2.1 .NET Core

.NET Core je nejnovější verze platformy .NET. Je multiplatformní open source frameworkem, umožnující hostovat Asp .NET server mimo prostředí Microsoft Serveru, například na MAC či nespočtu linuxových distribucí (RHEL, Ubuntu, Mint, Debian, Fedora, CentOS, Oracle, openSUSE).

Jeho použití jsem zavrhl, protože v době prototypování nebyla vydána stabilní release verze a rizika s použitím takto radikálně odlišné verze v raném stadiu převyšovala pozitiva.

## 2.2 Prototypy

Při hledání a volbě technologií klienta jsme si vytyčili několik faktorů, které jsme se snažili obsáhnout. Tyto faktory byly mimo jiné:

- Udržitelnost kódu ve větším projektu
- Možnost refaktoringu starých kódů
- Učící křivka pro vývojáře

- Perspektiva dané technologie do budoucna
- Adaptace ostatních vývojářů ve společnosti na tyto technologie

Na základě hledání na internetu, čtení článků na dané téma, konverzace s ostatními webovými vývojáři z jiných firem jsme se dohodli na následujícím seznamu technologií, které budeme prototypovat.

- Multi page application
  - ASP .NET MVC 5, Bootstrap
  - ASP .NET MVC 5 + Knockout MVC, Bootstrap
- Single page application
  - KnockoutJS, Typescript, Bootstrap
  - AngularJS, Typescript, Bootstrap
  - AngularJS, Typescript, Angular Material

V následujícím textu je výsledek mého hledání a podklady k následnému výběru.

### 2.2.1 Typescript

#### 2.2.1.1 Výhody

Typescript byl velkým překvapením. Díky třídnímu systému, typovosti, generice, podpoře rozhraní, překrývání atd. jsem byl schopen pro většinu standardních operací vytvořit bázové třídy ve stylu desktopové aplikace. Typescript se při buildu překládá do klasického javascriptu, a proto je kompatibilní se všemi moderními prohlížeči.

Díky definičním souborům pro knihovny třetích stran nejste limitování podporou vývojářů třetích stran. Pro již hotovou javascriptovou knihovnu jsou vytvořeny rozhraní, přes která se knihovny používají. Za celý rok jsem se nesetkal ani s jednou knihovnou, která by definiční soubor neměla. Existuje webová stránka GitHub DefinitelyTyped repository[2], kde jsou tisíce definičních souborů pro různé JS knihovny.

Později jsme ke kladům této technologie přidali i to, že Angular ve verzi 2.0 doporučuje Typescript jako hlavní jazyk[3] (umožňuje používat i jiné jazyky ale Typescript je prezentován v tuto chvíli jako hlavní). Výhody Typescriptu později předvedu v kapitole 5 sekce 5.3.6, 5.3.7 a 5.3.8.

#### 2.2.1.2 Nevýhody

Nevýhody nejsou téměř žádné. Snad jen menší problém je při použití některých starších JS knihoven, které způsobem jak jsou napsány a jak se používají, nejdou ruku v ruce s typovým prostředím.

Další problém byl v nesourodosti technologií dohromady. Většinou jsou tutoriály pro různé knihovny psány v čistém JavaScriptu, což ještě stěžovalo bolestivý proces učení u frameworků jako je například Angular. Zde jsem musel kódy z tutoriálů přepisovat do Typescriptu, abych je mohl rozumně použít ve svém prototypu.

### 2.2.2 Bootstrap

#### 2.2.2.1 Výhody

- Opravdu etablovaný framework na který existuje miliony šablon za žádný nebo drobný poplatek.
- Existuje spousta různých kontrolek používající Bootstrap.
- Bootstrap používá většinou CSS třídy (generované z LESS souborů). Přesto že Bootstrap používá JavaScript, většina věci se nastavuje pomocí CSS tříd, což je v kontrastu například s Angular Material.
- Lze použít „out of box“.

#### 2.2.2.2 Nevýhody

- Bez použití šablony třetích stran bude výsledná stránka velmi podobná tisíci ostatním stránkám napsané v Bootstrapu. Je potřeba zvolit chytře šablonu, která odliší vaší stránku od ostatních.
- Velká knihovna, která se dlouho načítá (například na mobilním telefonu).

### 2.2.3 Angular Material

#### 2.2.3.1 Výhody

- Spoustu funkcí „out of box“.
- Velmi zajímavý grafický styl ve výchozí šabloně.

#### 2.2.3.2 Nevýhody

Příliš spjaté s Angularem, součástí frameworku jsou angular direktivy jako **md-button** a podobně.

### 2.2.4 SPA přístup

SPA[3] přístup znamená, že při prvním navštívení webové aplikace se nahraje celá webová aplikace (HTML, Javascript, CSS) a dále běží v prohlížeči. Nepřenačitá se, pouze interaguje pomocí javascriptu s uživatelem, donačítá data ze serveru na pozadí ve formátu JSON. SPA aplikace často tvoří dojem desktopové aplikace bežící v prohlížeči.

#### 2.2.4.1 Výhody

- Jakmile se webová aplikace načte, není poté tak citlivá na rychlost internetového připojení. To je způsobeno hlavně tím, že pro další běh aplikace není potřeba donačítat HTML, skripty a CSS styly. Jediné co aplikace donačítá, jsou JSON data ze serveru, která jsou úsporná.

- V kombinaci s Typescriptem a MVVM[4] (MVC[5]) frameworkem se vývoj webové aplikace mnohem více přibližuje vývoji desktopové aplikace a je zde mnohem méně potřeba řešit určité workaroundy specifické webovému vývoji.

- Server není závislý na technologii klienta, takže lze použít stejné Web API pro webového klienta a například nativní mobilní aplikaci.

#### 2.2.4.2 Nevýhody

- Velmi pozvolně stoupající učící křivka. Pochopit a začít chápat věci v kontextu chvíli trvá a je to bolestný proces.

- Nedostatek vývojářů, kteří rozumí SPA a zároveň backend vývoji na platformě .NET.

### 2.2.5 ASP .NET MVC

Jedná se o člena rodiny MPA[6]. HTML stránka se renderuje na serveru. Pro každý request se vygeneruje stránka nová, která se posílá zpět na klienta.

#### 2.2.5.1 Výhody

- Velmi zaběhlá a osvědčená technologie.

- Je potřeba méně programování v Javascriptu.

[3] **SPA** - Single Page Application
[4] **MVVM** - Model - View - ViewModel
[5] **MVC** - Model - View - Controller
[6] **MPA** - Multi page application

#### 2.2.5.2 Nevýhody

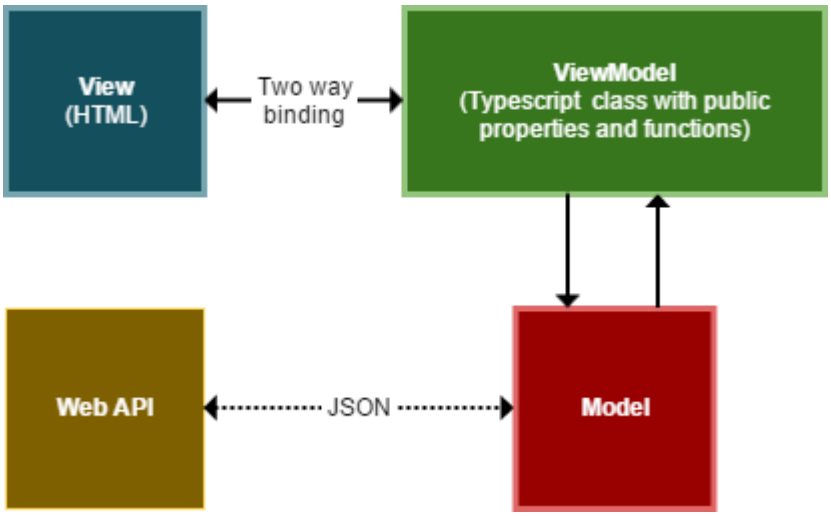
- Stránka se renderuje na serveru. Pro každou větší interakci uživatele tak dojde k přenačtení celé stránky. Proto tato technologie není vhodná pro použití na mobilních telefonech a tabletech, protože vyžaduje kvalitní internetové připojení.

- Velmi problematické udržování stavu komponent na stránce.

### 2.2.6 KnockoutJS

Framework **KnockoutJS** pro mně byl úvodem do světa SPA. Po úvodních potížích a nesnázích jsem byl velice pozitivně překvapen **MVVM** přístupem (který jsem znal z WPF[7] a Silverlightu) a velice rychle jsem se touto technologii nadchnul. Vytvořil jsem první prototyp, který obsahoval CRUD operace, stránkování a řazení. V průběhu psaní prototypu pro KnockoutJS jsem začal obracet a roli mého původního favorizovaného MPA, začal nahrazovat SPA přístup. V této části se prototypování velmi osvědčilo, protože bez něj bych pravděpodobně přistoupil k MPA přístupu.

Ze začátku byl pro mně KnockoutJS jednoduší kvůli čistému **MVVM** zaměření viz obr. 2.1, které důvěrně znám. Angular kombinuje MVC a MVVM a proto ze začátku působil více komplikovaně.




Obrázek 2.1: MVVM v KnockoutJS

KnockouJS používá pro hlídání změn binded properties **Observable wrapper** třídy. Ty zajistí, že při změně hodnoty dojde k promítnutí změn do HTML.

[7] **WPF** - Windows presentation foundation (Framework pro desktopové aplikace na platformě .NET)

#### 2.2.6.1 Výhody

- Jednoduchý začátek
- Přehledný tutoriál přímo na oficiální stránce
- Přímočařejší systém bindování a sledování změn

#### 2.2.6.2 Nevýhody

- Menší uživatelská základna
- Žádná velká firma na pozadí
- Menší počet článků a diskuzí s problémy a řešeními

### 2.2.7 AngularJS

Z důvodu porovnání jsem po dokončení prototypu KnockoutJS celý projekt přepsal do **AngularJS**. Začátek byl určitě složitější na pochopení, zvláště **scope**[8], tvorba **kontrolerů**, **direktivy** atd.

#### 2.2.7.1 MVC vs MVVM

Další nepříjemností bylo, že AngularJS byl původně koncipován jako MVC framework, a až později dostal funkce, které usnadňují použití MVVM. Původně kontroler sloužil k naplnění dat do scope. Scope je poté viditelný z HTML pro bindování viz obr. 2.2.

Tento přístup se mi ale nelíbí. Angular přidal funkci **controller as**[4] viz obr. 2.3, která automaticky vytvoří na scope proměnou kontroleru. To umožní přistupovat přímo k propertám a funkcím na kontroleru, podobně jako v případě MVVM v KnockoutJS viz obr. 2.1. Pokusím se následující dva přístupy demonstrovat na diagramech viz obr. 2.2 a obr. 2.3.

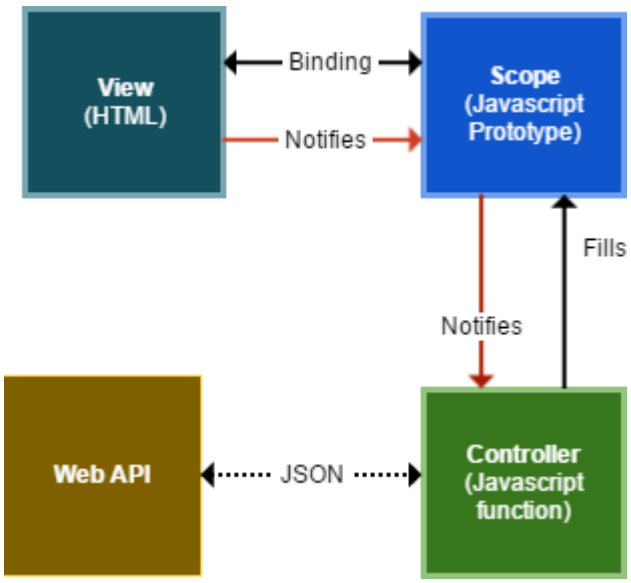
V další verzi Angular2 je scope úplně odstraněn. Moje rozhodnutí používat scope co možná nejméně by mi mělo usnadnit migraci projektu do Angular2.

#### 2.2.7.2 Dirty checking [4]

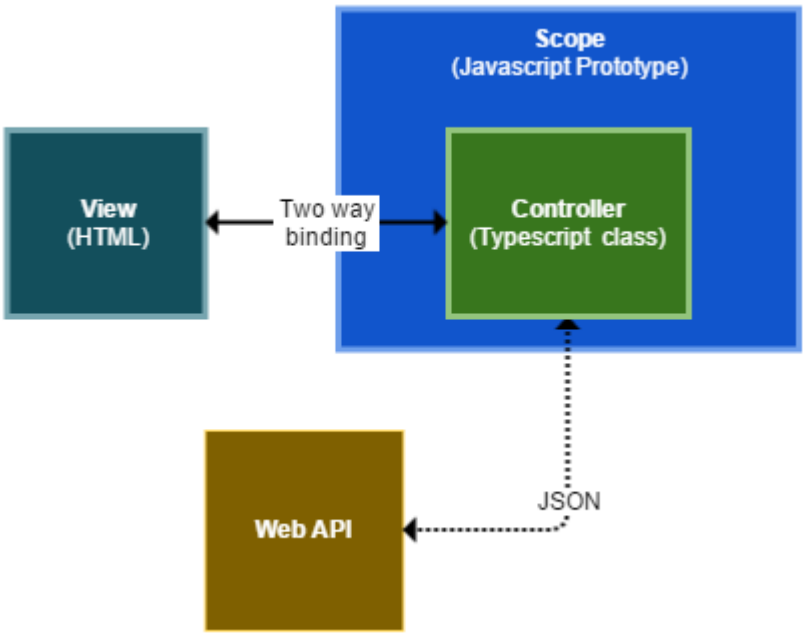
V angularu nejsou observable wrapper třídy jako v KnockoutJS. Místo toho vývojář používá standardní datové typy JavaScriptu a hlídání zajišťuje Angular sám. To je zajištěno pomocí tvz. **dirty checking**, který v cyklech kontroluje property na scope a kontroleru a pokud se nějaká property změní, promítne tuto změnu do view.

Postup dirty checkingu viz obr. 2.4 je následující:

[8] **Scope** - exekuční kontext pro expressions. Je tvořen hierarchickou strukturou kopírující strukturu DOM elementů aplikace [4].

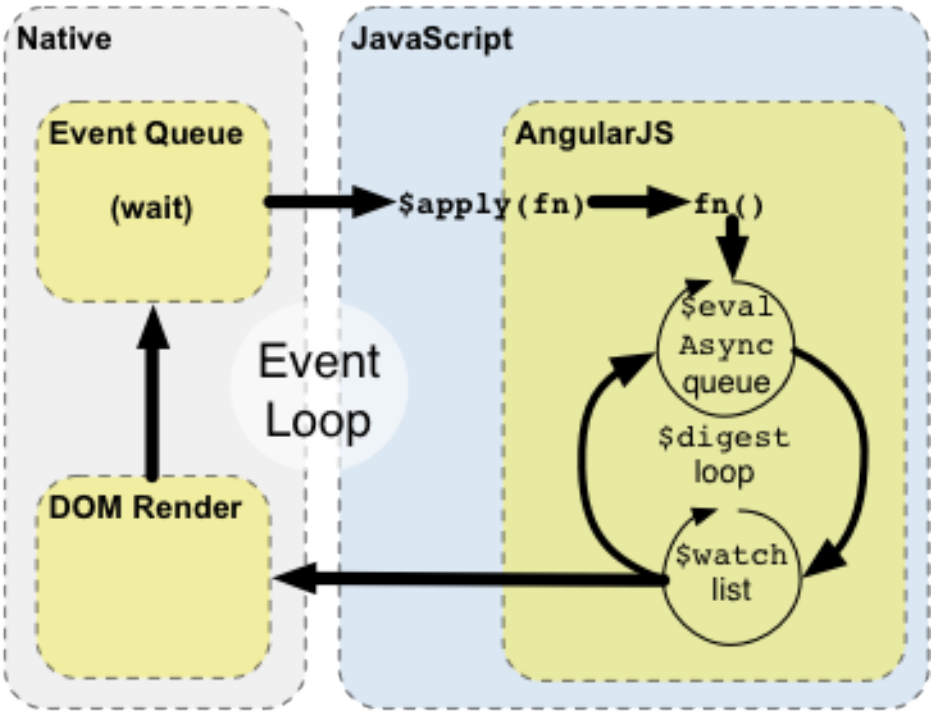



Obrázek 2.2: MVC v AngularJS




Obrázek 2.3: MVVM v AngularJS

1. V prohlížeči se čeká ve smyčce na novou událost (např. uživatelská interakce, timer atd.)
2. Dojde k události
3. Angular provede několikrát tvz. **$digest**, kde kontroluje změny na scope a kontroleru
4. Angular promítne změny do view




Obrázek 2.4: Dirty checking v Angularu [1]

#### 2.2.7.3 Výhody

- Velká uživatelská základna
- Spousta článků, diskuzí apod.
- Dobrá dokumnentace (dle mého názoru, někteří jí kritizují)
- Velká firma na pozadí
- Velmi aktivní vývoj, v současnosti startuje Angular2 s připravenou migrací z verze 1.5

#### 2.2.7.4 Nevýhody

- Dirty checking může při neopatrném použití způsobovat výkonnostní problémy
- Nerovnoměrně stoupající učící křivka[9], po slibném začátku mohou přijít problémy

## 2.3 Výběr technologie

Jak jsem již zmínil, při tvorbě SPA prototypů jsem se začal velice rychle přiklánět k SPA technologii, takže na konci bylo rozhodování více méně spíš o tom, jaký SPA framework použít. Angular se jevil jako sofistikovanější framework s více funkcemi a velice slibným výhledem do budoucna. Finální výběrem tedy byly tyto technologie:

- Databáze - Microsoft SQL Server 2016
- Server - Asp .NET 4.6 + Web API
- Klient - TypeScript, AngularJS 1.5, Bootstrap

[9] **Učící křivka** - potřeba zkušeností k ovládnutí nové technologie

Kapitola 3

# Realizace a řízení projektu

Pro účely projektu se ustanovil tým, který se postupem času rozšiřoval. V této kapitole nejdříve vysvětlím role všech členů týmů v projektu a poté popíši použité metodiky řízení softwarového vývoje.

## 3.1 Role v projektu

Role v týmu byly následující:

### 3.1.1 Moje role

- **Softwarový architekt** - měl jsem na starosti návrh architektury, výběr technologie a ostatní činnosti spojené s pracovní náplní softwarového architekta
- **SCRUM Master** - vedl jsem porady v 14 denním cyklu, plánoval práci pro ostatní členy týmu a tvořil zápis z porad.
- **Databázový architekt** - řídil jsem a navrhoval architekturu DB a tvorbu systému verzování DB.
- **Vývojář**

### 3.1.2 Role ostatních členů týmu

- **Vedoucí projektu a Product Owner** - Ing. Pavel Dvořák. Definuje vizi projektu, priority, rozhoduje co se bude dělat dřív, co později a co vůbec. Osoba zodpovědná za výběr a utvoření týmu.
- **HW programátor** - osoba zodpovědná za tvorbu komunikační knihovny, na kterou se napojuje aplikační vrstva. Jedná se o autora firmwaru v GPS jednotkách, který poskytoval v průběhu projektu cenné konzultace.

- **Databázista** - osoba zodpovědná za vývoj a administraci DB, tvorbu migračních skriptů, vytváření testovacích dat a jiné práce spojené s údržbou DB. Dále autor pomocné aplikace pro úpravu testovacích dat a autor skriptů pro nasazování DB.
- **Tester** - osoba zodpovědná za testování aplikace
- **Uživatelé předchozí verze aplikace** - v pozdější fázi jsme konzultovali s uživateli aplikaci FleetwareMaster jejich zkušenosti s předchozí verzí.

## 3.2 Řízení projektu

V implementační části projektu jsem ustanovil procesy, které se skládaly s těchto částí.

### 3.2.1 SCRUM

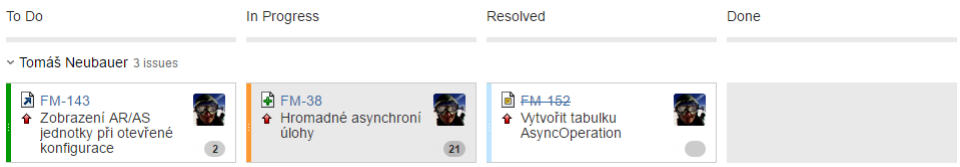
K řízení vývoje jsem použil pozměněnou variantu agilní metodiky **SCRUM**. Jeden sprint měl délku 14 dnů. Každému sprintu předcházel workshop, kde jsme ohodnotili úkoly a naplánovala se práce všem členům týmu na dalších 14 dnů. K ohodnocování úkolů jsme používali **storypointy**.

#### 3.2.1.1 Storypointy

Storypointy jsou imaginární jednotka, která by měla reprezentovat složitost daného úkolu. Zakomponovává v sobě několik věcí:

- Komplexnost
- Náročnost
- Počet neznámých v úkolu. Jak moc velkou část úkolu nevíme z hlavy a budeme muset bádat a zkoumat.

Storypointy nejsou přímo závislé na čase. Jak si tým zvykne na stupnici (v našem případě **Fibonacciho posloupnost**), časem se ustálí počet storypointů u každého člena týmu za jeden sprint. Tím se umožní plánování na delší časový horizont. Idea je taková, že víme přibližně kolik storypointů tým za sprint umoří a pokud máme úkoly oceněné dopředu, můžeme je naplánovat do budoucích sprintů. Tato metodika zmírňuje jednotlivé nepřesnosti v odhadech a jiné proměné, které způsobují nepřesnosti v klasických hodinových odhadech. Další výhodou je, že vývojář není tlačen do splnění časového kvanta přiděleného k úkolu a nemá tendenci úkol dodělat za každou cenu v přiděleném čase i za cenu horší kvality kódu.




Obrázek 3.1: JIRA Agile work board

### 3.2.2 JIRA

K plánování úkolů a evidenci chyb (bug/issue tracking) jsme používali placený software **JIRA** od firmy **Atlassian** se zakoupeným pluginem **Agile** viz obr. 3.1. Jedná se o velmi kvalitní software se spoustou různých pluginů a reportů. JIRA je stále v aktivním vývoji a Atlassian neustále přidává nové funkce.

### 3.2.3 Confluence

Pro účely dokumentace, zápisu know-how, zápisu porad atd. jsme použili placený software od Atlassianu **Confluence**. Jedná se o wiki s integrací do systému JIRA.

#### 3.2.3.1 Testování úkolů

Ve většině případů probíhá testování úkolů tímto způsobem:

1. V momentě, kdy vývojář dokončí úkol, změní jeho stav na **Resolved** a předá ho testerovi.
2. Tester otestuje funkčnost související s úkolem.
   a) Pokud najde nesrovnalosti mezi zadáním a aplikací a nebo najde chybu, předá úkol zpět autorovi a změní stav úkolu na **Reopened**.
   b) Pokud je úkol v pořádku, změní jeho stav na **Closed**.

## 3.3 Continuous Integration

Pro Continuous Integration jsme použili placený program **Teamcity**. Na Teamcity jsem nastavil tyto profily viz obr. 3.2:

### 3.3.1 FleetwareMaster.Build Including Unit Testing

Jedná se o hlavní buildovací profil. Spouští se automaticky při změně v GITu včetně změny v testovacích datech a struktury DB, která je také uložena v GITu. Tento profil je zdrojem artefaktů[10]

[10] **Artefakty** - soubory vzniklé buildem aplikace, které se nasazují na server

1. Build serveru
    a) Instalace Nuget balíčků
    b) Build .NET kódu VS2015 kompilátorem
2. Build klienta
    a) Node.js
    b) Gulp
        i. LESS build
        ii. CSS zabalení
        iii. CSS minifikace
        iv. Zkopírování assets (HTML, obrázky atd.)
        v. Zkopírování fontů
        vi. Zkopírování i18n resource souborů
    c) Bower instalace
    d) Webpack instalace
3. Vytvoření DB s testovacími daty pro účely testů
    a) Spuštění dávkového souboru pro vytvoření DB skriptů
    b) Nastavení názvu DB z aktuálního profilu (FleetwareMasterBuild)
    c) Spuštění DB skriptu, který:
        i. Ukončí všechny připojení k DB
        ii. Provede drop aktuální DB
        iii. Vytvoří DB verze 1
        iv. Spustí postupně všechny migrační skripty až na nejnovější verzi DB
        v. Vloží do DB testovací data
4. Spuštění unit testů
    a) Integrační testy (Testy Web API kontrolerů)
    b) Testy business logiky

### 3.3.2 FleetwareMaster - nightly - new DB

V momentě, kdy dojde k úspěšnému buildu větve **develop** 3.6.2 (může být dočasně nastavená jiná větev), můžeme použít tento profil pro nasazení vývojové verze na testovací prostředí **Nightly**. Tento profil se automaticky pouští každý den o půlnoci. Profil mimo jiné provede:

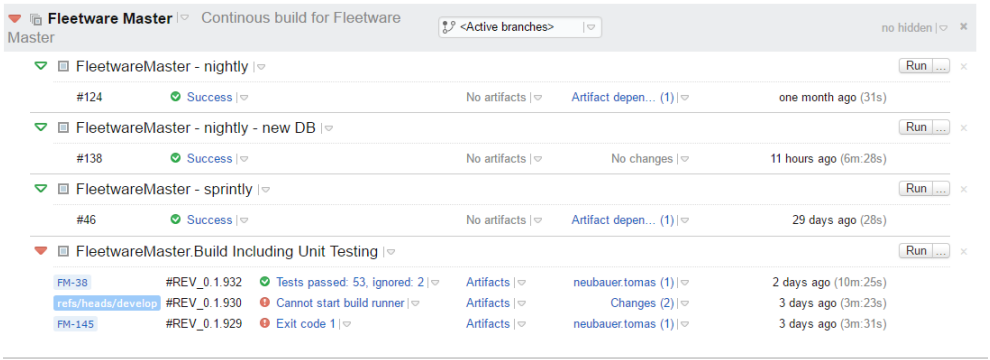
1. Nakopíruje artefakty serveru a klienta na testovací prostředí **Nightly**
2. Vymaže FleetwareNightly DB a vygeneruje jí znovu

### 3.3.3 FleetwareMaster - nightly

Podobný profil jako v sekci 3.3.2. Nemaže DB - ponechá ji v původním stavu.

### 3.3.4 FleetwareMaster - sprintly

Vyhází z profilu popsaném v sekci 3.3.3. Výchozí zdroj artefaktů je nastaven na **release**[11] větěv v GITu.




Obrázek 3.2: Teamcity projekt FleetwareMaster

Verzování a nasazování databáze je součástí nasazovacího a build procesu. Díky tomu se nemůže stát, že někdo zapomene zanést změnu v DB do balíčků. **Veškeré testování a ověřování se děje nad testovací DB, která se pokaždé přegeneruje z GITu**. Jakákoliv ad-hoc změna v DB nezanesená do GITu je tak okamžitě smazána a proto je odhalena nefunkčností programu nebo unit testů. Více viz sekce 6.2.

## 3.4 Testovací prostředí

Testovací prostředí se skládá ze dvou prostředí. Nasazovací proces je popsán v předchozích odstavcích.

### 3.4.1 Nightly

Nestabilní prostředí, které se nasazuje každý den o půlnoci z vývojové větve GITu. Slouží testerům k testování práce vývojářů. V momentě, kdy vývojář úkol zavře, provede merge do vývojové větve. Tester může očekávat, že nejpozději druhý den (pokud nedojde k chybě v buildu), je funkcionalita připravená k otestování na Nightly.

[11] **Release větev** - branch v GITu, reprezentující určitou verzi aplikace určenou do produkce

Na Nightly se DB maže každý den, aby úkoly byly testovány proti stabilním datům. Pro přidávání nových testovacích dat je vytvořený nástroj viz sekce 6.2.3.1.

### 3.4.2 Sprintly

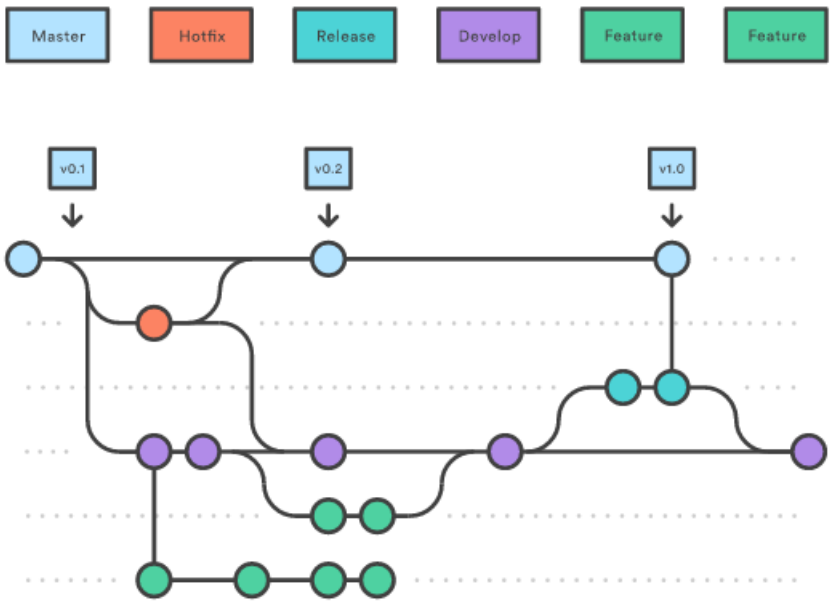
Stabilnější prostředí, které se zpravidla nasazuje až po doladění nové funkcionality a ukončení sprintu. Na konci sprintu se vytvoří nová verze aplikace a s ní související branch v GITu. Tato branch se poté nasadí na Nightly, kde se pomocí testerů doladí. V momentě, když je verze doladěna, nasadí se na sprintly.

Při nasazení se ručně nasadí migrační skripty, které vznikly mezi verzí, která je aktuální na sprintly a kterou nasazujeme. To slouží i jako kontrola, že při nasazování produkční verze nedojde k chybě při migraci DB na novější verzi. Data na sprintly databázi se tedy automaticky nemažou a zůstávají.

## 3.5 Source control

Jako source control jsme použili cloudové úložiště GIT **Atlassian BitBucket**.

## 3.6 GIT Flow




Obrázek 3.3: GitFlow diagram

Jako pracovní metodiku v GITu jsme použili **GIT Flow** [5] viz obr. 3.3. Idea je taková, že se vytvoří dvě hlavní branche:

- **master** - Pouze merge release větví běžící v produkci
- **develop** - Vývojová verze (ekvivalent k trunku v SVN)

### 3.6.1 Master

Při vytváření nové verze se z **develop** větve nejdříve vytvoří nová větev **release\x.x.x**, která je podrobena testování. V momentě, kdy je verze připravena do produkce, provede se merge do větve master. Merge je otagován verzí aplikace, která je viditelné uživateli a verze je nasazena.

### 3.6.2 Develop

Vývojová verze. Pokud vývojář (nebo databázista) začne pracovat na novém úkolu, vytvoří větev z develop větve a nazve jí **Feature\JIRA-TASK**.

### 3.6.3 Pull request

Po ukončení práce na úkolu se vytvoří pro merge větve (feature\JIRA-TASK) do větve **develop** tvz. **pull request**. To je možné provést například z aplikace **SourceTree**[12] viz obr. 3.4.

V následujícím kroku se automaticky otevře webová stránka na bitbucket.org, kde vývojář/databázista nastaví cílovou větev **develop** a přidá posuzovatele viz obr. 3.5.

### 3.6.4 Hotfix

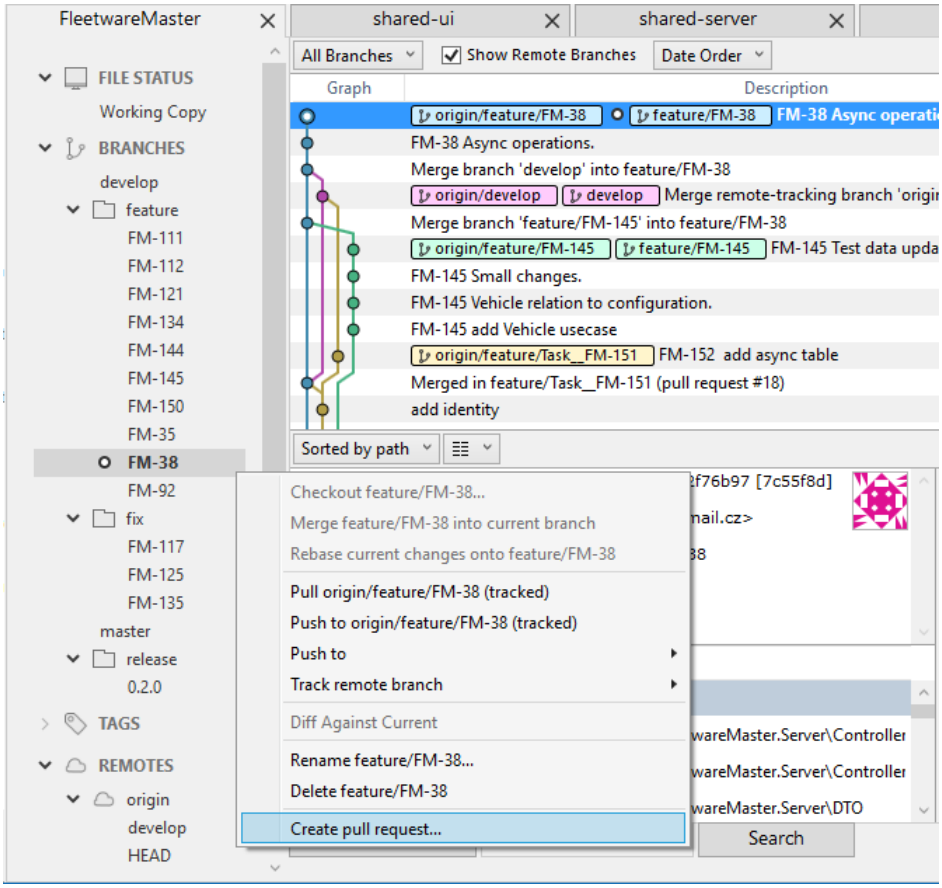
**Hotfix** se vytváří v případě kritické chyby v produkci, která musí být okamžitě opravena a není čas na to, aby prošla standardním procesem. Hotfix se vytváří z komitu **master** produkční větve, kde byla chyba nalezena. Poté, co je chyba opravena, dojde k merge zpět do produkční větvě **master** a k nasazení opravy. Současně dojde k merge do vývojové větve, aby se chyba opravila i v budoucích verzích.

## 3.7 Sdílení kódu ve společnosti

Framework pro klientskou část byl koncipován jako sdílený mezi více projekty ve firmě. V současnosti je tak kromě projektu **FleetwareMaster** sdílen ještě s hlavním projektem ve firmě, který je také přepisován do technologie HTML5.

Pro sdílení frontend kódu je vytvořen GIT repositář **shared-ui**, který je připojen jako submodule do hlavního GIT repositáře. Ve sdíleném repositáři

[12] **SourceTree** - GIT klient od firmy Atlassian.

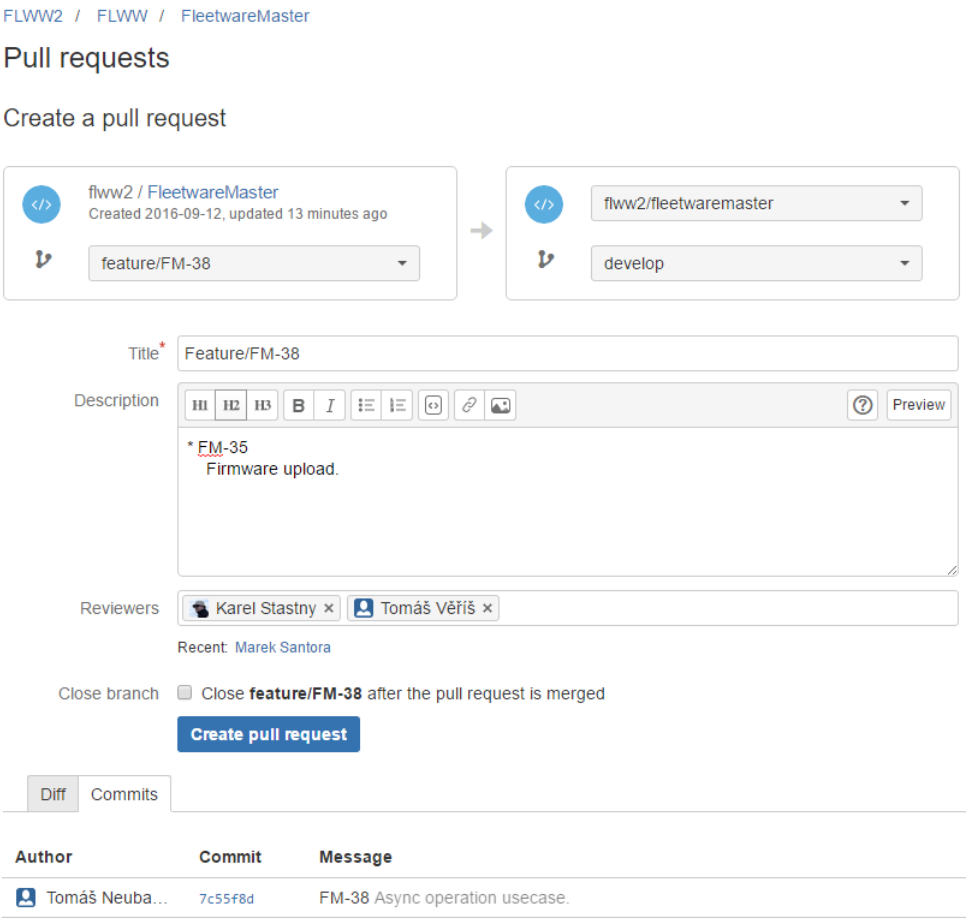



Obrázek 3.4: SourceTree Pull request

jsou bázové třídy v Typescriptu, například bázová třída pro Datový model viz 5.3.6, CrudMasterController viz 5.3.8 atd. Tyto třídy jsou abstraktní a každý projekt, který používá tuto sdílenou knihovnu je podědí a použije.

Pro balení JavaScriptových souborů používáme **Webpack**. Bázové třídy z modulu **shared-ui** se zabalí spolu se skripty FleetwareMasteru do velkého souboru ve správném pořadí.

Implementace různých služeb se registrují jako Angular služby až v konkrétních projektech. V bázových třídách jsou závislosti v kontruktorech definovány pomocí rozhraní, které v každém projektu reprezentují jiné implementace. Například služba `AutentificationService`, která přijímá závislost `ITokenService`. V následujícím kódu je ukázka této služby.




Obrázek 3.5: SourceTree Pull request

```
/**
 * Provides token authentication for WebAPI OWIN authorization.
 */
export class AuthenticationService
{
    /**
    * CTOR.
    * @param tokenDataModel DataModel for token acquisition.
    */
    constructor(
        private tokenAccessService: ITokenService,
        protected $q: angular.IQService,
        private $localStorage: IAuthenticationStorage)
    {
    }

    ...
}
```

Každá aplikace registruje vlastní implementaci `ITokenService`, od které je bázová služba odstíněna.

```
export class FMTokenService implements ITokenService
{
    constructor(
        private appConfigService: FMAppConfigService,
        private $http: angular.IHttpService)
    {
    }

    getToken(username: string, password: string): angular.IPromise<Token>
    {
        return this.appConfigService.config.then((conf) =>
        {
            return conf.server.fleetwareMasterApiUrl;
        }).then((fleetwareMasterApiUrl) =>
        {
            return this.$http(...);
        });
    }
}
```

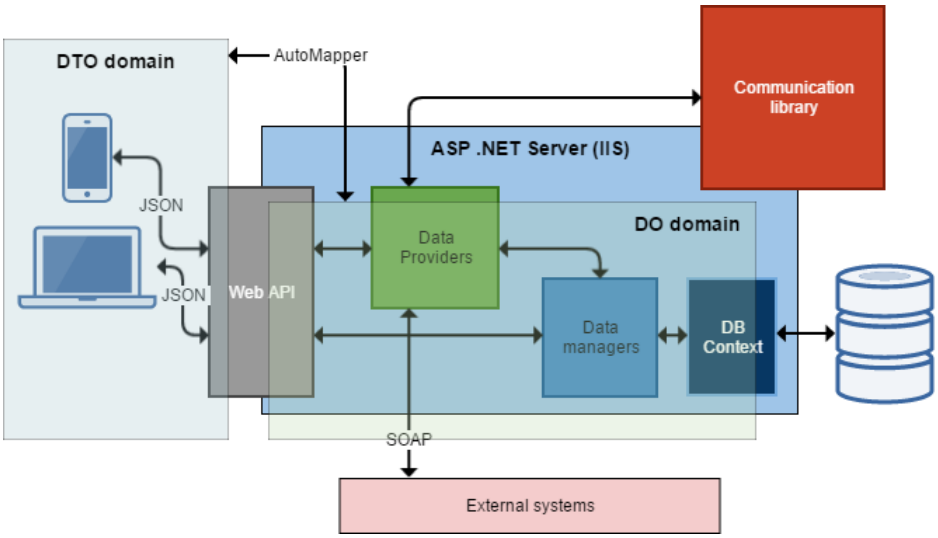
Dalším způsobem jak docílit podobné funkcionality je použití **abstraktních metod**.

Kapitola 4

# Návrh softwarové architektury

V této kapitole popíšu návrh softwarové architektury projektu FleetwareMaster. Některé komponenty představené v této kapitole dále rozeberu v kapitole 5. Navrhovaná architektura je znázorněna v několika pohledech v různých diagramech.

## 4.1 Návrh komponent




Obrázek 4.1: Architektura serverové části

### 4.1.1 DTO doména

DTO[13] doména viz průhledný obdélník vlevo na obr. 4.1 je část architektury, která není ve spojení s DB. V této doméně jsou objekty transportovány mezi klientem a serverem pomocí Web API kontrolerů.

### 4.1.2 DO doména

DO[14] doména viz průhledný obdélník uprostřed na obr. 4.1 je část serveru, kde aplikační logika pracuje přímo s databázovými objekty. Změny na těchto objektech jsou sledovány databázovým kontextem. Převod mezi DO a DTO objekty je ve většině případů realizován knihovnou **AutoMapper**.

### 4.1.3 Servisní vrstva

Servisní vrstvu ve FleetwareMasteru tvoří `Web API kontrolery`, které vystavují HTTP metody **GET**, **POST**, **PUT**, **DELETE** atd. pro webového klienta. Kontrolery komunikují ve formátu JSON[15]. Servisní vrstva nevystavuje DO objekty klientům. Při zavolání HTTP metody klientem dojde k obalení volání transakčním kontextem, který zajistí připojení databázového kontextu k DB a v případě chyby v půlce **PUT**/**POST** volání provede automatický **rollback** transakce.

### 4.1.4 Data managers

Data manager je bázová třída, která má na starosti práci s entitami mapovanými přes ORM do databáze. Jedná se o třídu s generickým parametrem reprezentující typ DO objektu, která poskytuje metody pro CRUD operace, kontroluje práva či implementuje jiné databázové operace spjaté s DO entitou.

### 4.1.5 DB Context

**DB Context** je třída připojující se k databázi a registrující DO třídy do databázového kontextu. Využívá `connection string` z konfigurace webového serveru.

### 4.1.6 Externí systémy

Některé entity jako například **Vozidlo** sdílí FleetwareMaster se systémem **FleetwareWeb**. Oba systémy musí synchronizovat tyto entity a udržovat je ve shodném stavu. FleetwareMaster používá některé informace ze systému **FleetwareWeb** jako například **registrační značka** nebo **název vozidla**.

[13] **DTO** - Data Transfer Object
[14] **DO** - Data object (ORM namapované třídy)
[15] **JSON** - JavaScript Object Notation

### 4.1.7 Klient

Klient je webová HTML5 Single page aplikace bežící ve webovém prohlížeči. Při prvním spuštění se uživateli do prohlížeče stáhnou všechny potřebné soubory a dále aplikace komunikuje pomocí HTTP dotazů na Web API serverové části.

## 4.2 Architektura komunikace s jednotkami

Aplikace FleetwareMaster se skládá z několika domén, které v této sekci popíši. Komunikace uživatele aplikace se SIM kartou v jednotce je znázorněna v doménovém diagramu viz obr. 4.2.

### 4.2.1 Komunikační knihovna

Aplikační server komunikuje s jednotkami pomocí komunikační knihovny. Knihovna dále komunikuje s **CGU servery** přes **ARNEP protokol**.

#### 4.2.1.1 ARNEP

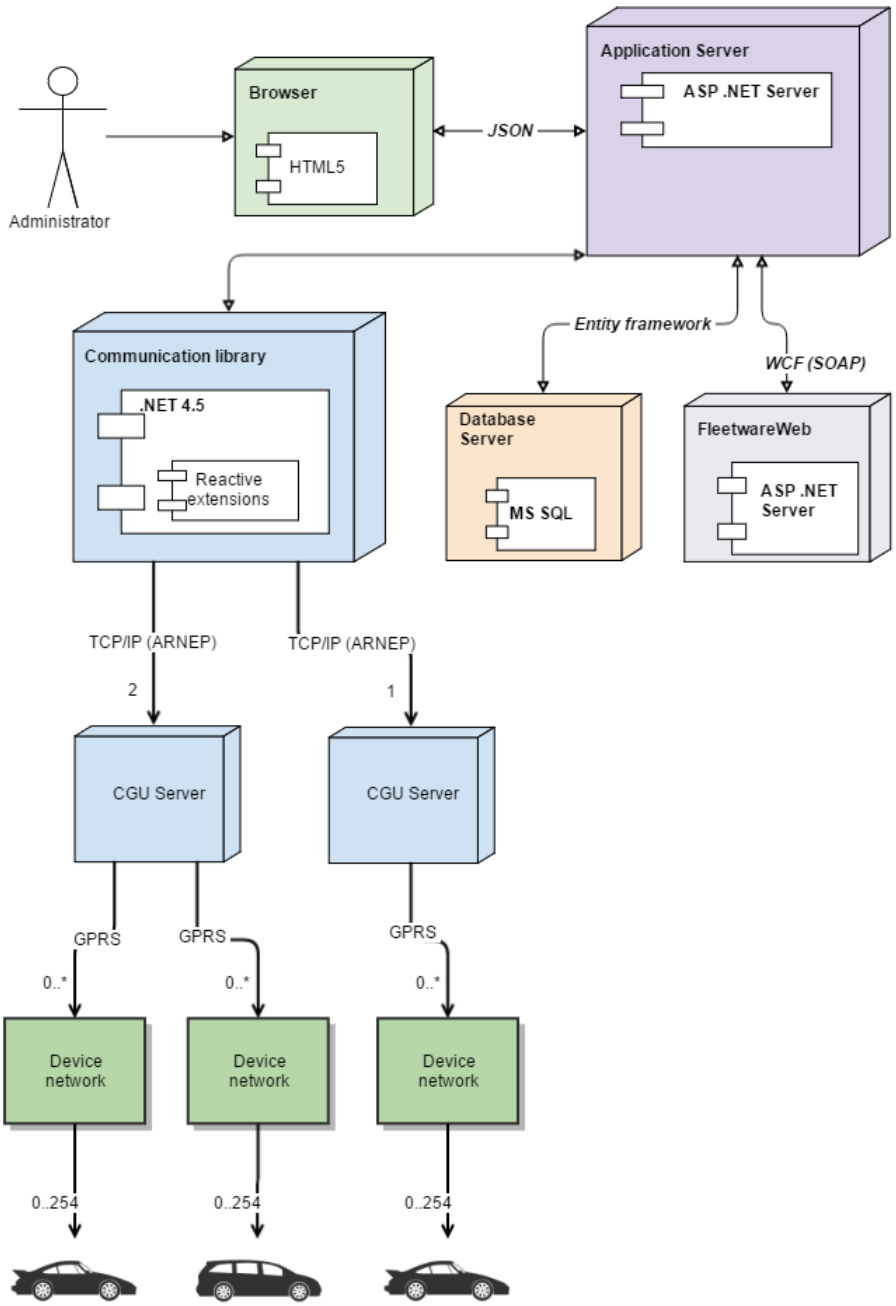
Protokol **ARNEP** je 8-bitový asynchronní sériový komunikační protokol používaný pro komunikaci s radiovými modemy dodávanými firmou Conel. Komunikace probíhá ve formě výměny paketů mezi připojeným zařízením a radiovým modemem [6].

### 4.2.2 CGU server

Komunikační knihovna je napojená na několik **CGU serverů**, kteří knihovně zpřístupňují **ARNEP** síť SIM karet. Připojení k CGU serverům se kontroluje v obrazovce **Připojení k CGU serverům** viz C.1.

### 4.2.3 Sítě jednotek

Každá síť s hexadecimální adresou může obsahovat až 254 jednotek se SIM kartou.



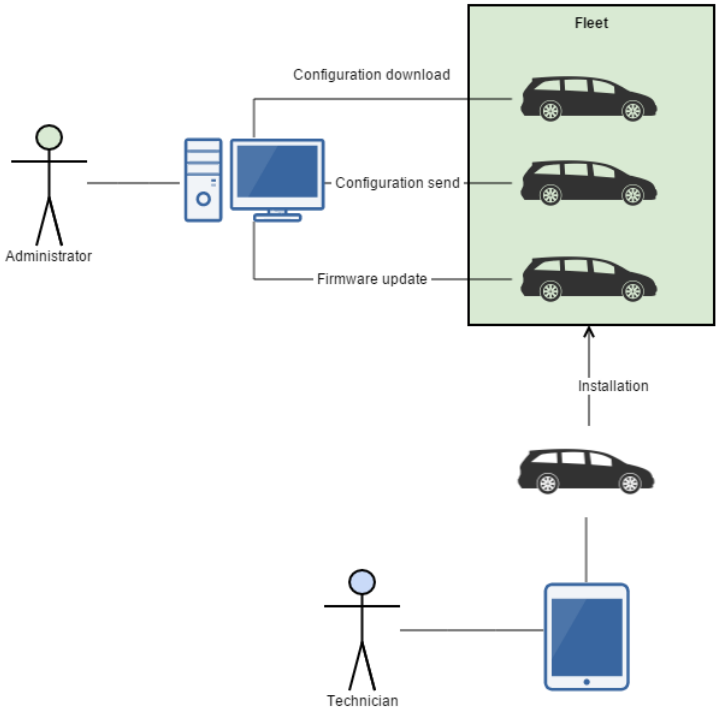

Obrázek 4.2: Doménový diagram

## 4.3 Příklad užití

Aplikace má dva hlavní typy uživatelů.

- **Administrátor** - spravuje kompletní flotilu vozidel a všechny jednotky, které společnost vlastní a provozuje
- **Technik** - osoba instalující nové jednotky do vozidel, konfigurující jednotku pomocí tabletu v terénu

Diagram užití viz obr. 4.3.




Obrázek 4.3: Diagram užití

## 4.4 Životní cyklus konfigurace jednotky

Návrh práce s konfigurací jednotky byl poměrně složitý proces, protože je potřeba ukládat všechny změny v konfiguraci každé jednotky. Dále je nutné umožnit uživateli pracovat s konceptem konfigurace, který si může uložit a poslat do jednotky později.

### 4.4.1 Stažení konfigurace

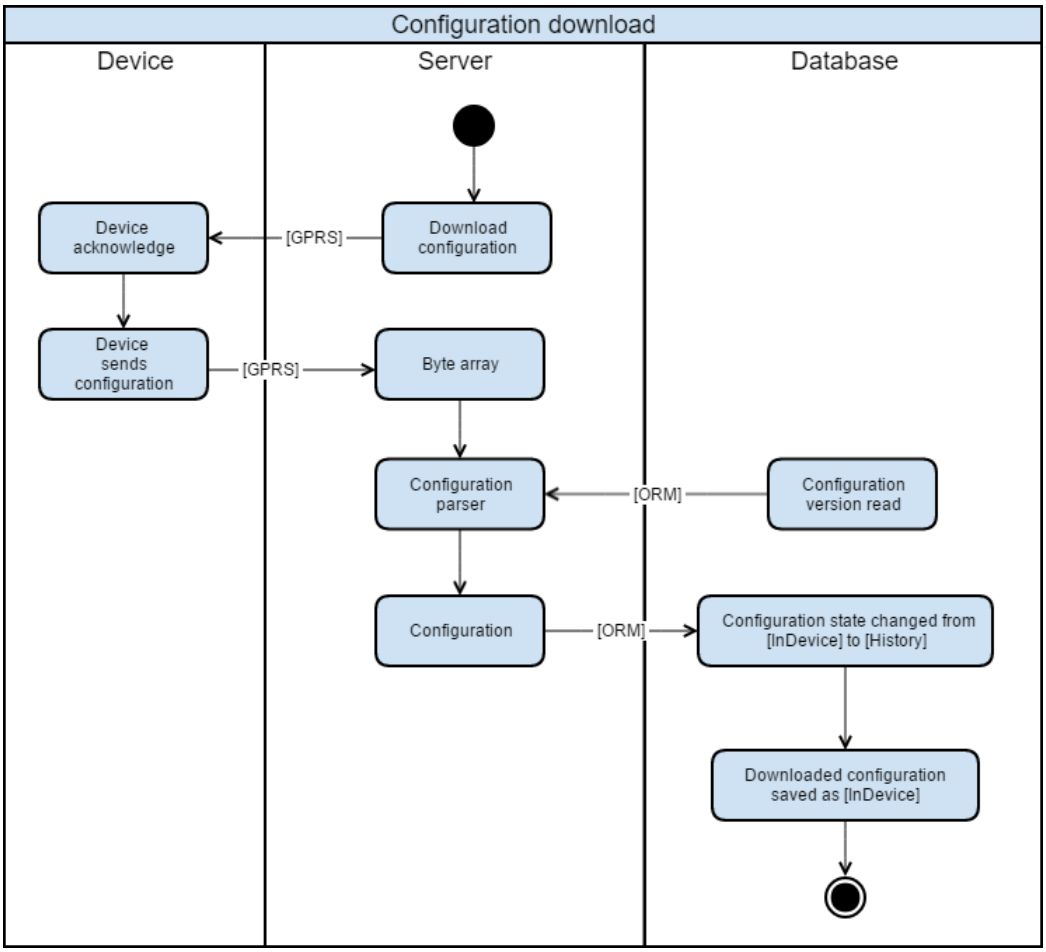
Stažení konfigurace (viz diagram 4.4) probíhá v tomto sledu:

1. Stažení bytového pole aktuální konfigurace z jednotky
2. Načtení verze konfigurace podle firmwaru jednotky
3. Na základě této verze konfigurace je bytové pole převedeno na položky konfigurace třídou `ConfigurationParser`
4. Původní konfigurace v jednotce je uložena v DB jako **historická**
5. Nově vytvořená konfigurace je uložena jako **konfigurace v jednotce**

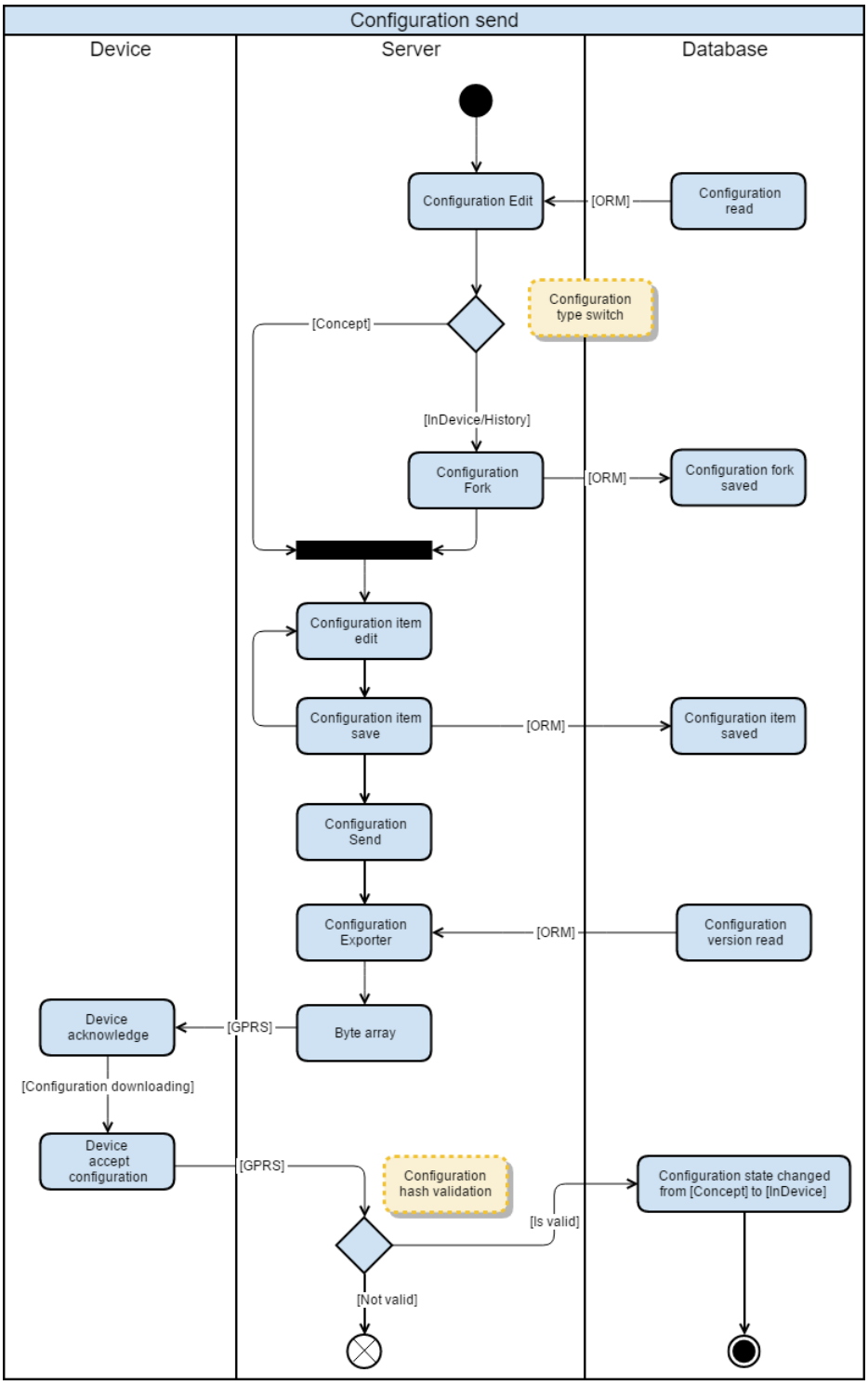
### 4.4.2 Editace konfigurace a odeslání do jednotky

Postup práce uživatele s konfigurací a její odeslání do jednotky (viz diagram 4.5) je následující:

1. Zvolení editace konfigurace
2. V závislosti na typu vybrané konfigurace:
   - **Koncept** - editace se otevře a uživatel pokračuje v konceptu
   - **Konfigurace v jednotce** nebo **Historie** - uživatel je dotázán, jestli chce vytvořit z vybrané konfigurace koncept
     - Jedná se o `fork` konfigurace, který uloží kopii konfigurace do DB
3. Editace položek konfigurace
4. Zvolení odeslání konfigurace do jednotky
5. Načtení verze konfigurace podle verze firmware v cílové jednotce
6. Exportování položek konfigurace do bytového pole na základě načtené verze konfigurace
7. Odeslání do jednotky
8. Kontrola hash vrácené jednotkou
   - **Hash je validní** - uložení odesílané konfigurace v DB jako **Konfigurace v jednotce**
   - **Hash není validní** - chyba aplikace zobrazená uživateli




Obrázek 4.4: Diagram stažení konfigurace




Obrázek 4.5: Diagram editace konfigurace a odeslání do jednotky

Kapitola 5

# Implementace

V této kapitole rozeberu podrobněji některé moduly obsažené v předchozí kapitole 4. Nastíním implementaci těchto modulů a přidám ukázky zdrojových kódů. V sekci Klient 5.3 popíši jak jsem využil možnosti Typescriptu při tvorbě sdílené knihovny.

## 5.1 Server

### 5.1.1 Data managers

Nejdůležitější metodou třídy **Data manager** je `GetAllItemsQuery`, která vrací `query`[16] dotaz do databáze. S takovým dotazem lze dále pracovat a obalovat ho dalšími dotazy. Nakonec je z `query` sestaven select, který je odeslán k vykonání do databáze příkazem `ToList()`, `FirstOrDefault()`, atd. Příklad použití GetAllItemsQuery:

```
var selectionSelectQuery = configurationItemSelectionManager
       .GetAllItemsQuery()
       .Where(w => w.SelectionType == "textvalue")
       .OrderBy(o => o.ConfigurationItem.Name)
       .GroupBy(g => g.ConfigurationItemId);

// Až zde dojde k dotazu do DB
var selections = selectionSelectQuery.ToList();
```

Zdrojový kód 5.1: LINQ dotaz s použitím GetAllItemsQuery.

[16] `Query` sestavený select dotaz k odeslání do DB

### 5.1.2 Entity Framework

Entity Framework je ORM[17] mapovací knihovna. Umožňuje mapovat například tabulky nebo relace na C# třídy. Základním stavebním kamenem pro tvorbu SQL dotazů je **LINQ**[18]. Jedná se o čistě typové dotazování, které lze kombinovat s lambda funkcionálním přístupem, viz zdrojový kód 5.1. Čistý LING viz 5.2.

```
var stringValues = from value in data
                join selection in selectionSelect
                on value.ConfigurationItemId equals selection.Key
                where value.ValueInt != null
                select new
                {
                    ValueId = value.ID,
                    Selection = selection.FirstOrDefault(f =>
                        f.DescriptionOrder == value.ValueInt)
                };
```

Zdrojový kód 5.2: Čistý LINQ dotaz.

Vygenerovaný SQL dotaz lze v **Debug módu** zkontrolovat. Objekt `IQuarable<T>` přetěžuje metodu `ToString()` a vrací vygenerované SQL.

Pro mapování tříd na tabulky používám atributovou notaci (je více možností, např. designér). Příklad viz 5.1.2

V příkladě viz zdrojový kód 5.3 je položka `Device` mapována skrz tvz. **lazy loading**. Díky tomu dojde k přidání **join** klauzule k dotazu až při přístupu k této položce. Pokud nedojde k přístupu k položce, join dotaz se nepoužije. Stejným způsobem funguje i mapování **N:M** relací.

Většina DO objektů dědí z `FMDataObject`, což je abstraktní bázová třída, která mapuje sloupce `ModifiedDate`, `ModifiedUser`, `InsertedDate` a `InsertedUser`. Příklad viz zdrojový kód 5.3. Tyto sloupce plní bázová implementace `FMDataManagerBase`.

Třída `FMDataObject` dále dědí od třídy `DataObject`, která obsahuje položku **ID**, se kterou počítá bázová implementace `DataManagerBase`.

### 5.1.3 Data Providers

Tato skupina tříd leží mezi **Web API** a **Data managers**. Obsahují **business logiku**.

Jako příklad uvedu `IOperationProvider`. Tento provider používá externí komunikační knihovnu a několik `data managers` tak, aby pro danou operaci

[17] **ORM** - Objektově relační mapování
[18] **LINQ** - Language Integrated Query

```
[Table("Configuration")]
public class ConfigurationDO : FMDataObject
{
    [Key]
    [Column("ConfigurationId")]
    public override long ID
    {
        get { return base.ID; }
        set { base.ID = value; }
    }

    [Column]
    public long? DeviceId { get; set; }

    [ForeignKey(nameof(DeviceId))]
    public virtual DeviceDO Device { get; set; }

    [Column]
    public DateTime? LastSavedToDevice { get; set; }
}
```

Zdrojový kód 5.3: Mapování tabulky na DO třídu.

našel komunikační údaje (IP adresu, APN, adresu sítě atd.) a zavolal vhodnou metodu na API komunikační knihovně.

### 5.1.4 Web API kontrolery

Protože většina kontrolerů obsahují hlavně CRUD[19] operace, vytvořil jsem `ICrudOperationHelper<TDto`[20]`, TDO`[21]`>`. Pro tyto operace se používá přímo tento helper, viz zdrojový kód 5.4.

#### 5.1.4.1 CrudOperationHelper

**CrudOperationHelper** je dvojitě generickou třídou používající shodně dvojitou generickou třídu IDataConverterBase<TDto, TDo>, která používá ke konverzi DTO a DO objektů **AutoMapper** viz 5.1.5. Dále se `CrudOperationHelper` stará o vhodné zpracování HTTP odpovědí (včetně chybových) a volá **Sig-**

[19] **CRUD** - Create, Read, Update a Delete
[20] **TDto** - Generický parametr DTO třídy
[21] **TDto** - Generický parametr DTO třídy

```
[HttpPut]
[Route("{id:long}")]
public HttpResponseMessage Update(long id, DeviceDTO item)
{
    return crudOperationHelper.Update(this, id, item);
}
```

Zdrojový kód 5.4: PUT metoda na Web API kontroleru

**nalR** notifikační službu o změnách. Tyto notifikace poslouchají klienti a obnovují si cache dat.

#### 5.1.4.2 Routing

Každý kontroler či metoda má vlastní routing určený atributem `Route`. Příklad ve zdrojovém kódu 5.5. V uvedeném příkladě lze pozorovat kombinaci url routování celého kontroleru a url routování `GET` metody `GetItem`.

Mějme položku s ID = 5, která je součástí konfigurace s ID = 10. Z uvedené definice url routování je pro vrácení této položky http dotaz následující:

```
GET "api/configuration/10/itemvalues/5"
```

Auto mapper automaticky překopíruje stejně nazvané položky třídy a to včetně vnořených objektů.

```
[Authorize]
[RoutePrefix("api/configuration/{configurationId:long}/itemvalues")]
public class ConfigurationValuesController : ApiControllerBase
{
    [HttpGet]
    [Route("{id:long}")]
    public HttpResponseMessage GetItem(long id, long configurationId)
    {
        ...
    }
}
```

Zdrojový kód 5.5: Routing v Web API kontrolerech

### 5.1.5 AutoMapper

Auto mapper je knihovna pro automatickou konverzi DTO a DO objektů. Dvojice se registují ve třídě `MappingRegistration` viz zdrojový kód 5.6. V uvedeném příkladě je potřeba dodefinovat rozdíl datových typů u položky `Disconnected`.

```
configuration.CreateMap<DeviceDTO, DeviceDO>()
    .ForMember(dest => dest.Disconnected,
               src => src.Disconnected.GetValueOrDefault() ? 1 : 0)
    .ForMember(dest => dest.DeviceType, src => src.Ignore());
```

Zdrojový kód 5.6: Příklad registrace dvojice tříd v **AutoMapperu**

## 5.2 Dependency injection

Na serverové části používám pro **Dependency injection** knihovnu **StructureMap**. Idea je taková, že ve třídách vkládám do konstruktorů rozhraní a **StructureMap** doplní podle nastavení jejich implementace (dle registrace novou instanci či singlton). Toho využívám například v testech, kde projekt s testy obsahuje vlastní registraci StructureMap a implementace pro komunikaci s jednotkami je nahrazena testovací implementací, která vrací předpřipravené odpovědi ze souboru.

Na ukázce zdrojového kódu 5.7 je vidět obecná registrace všech konkrétních instancí generického rozhraní pro `IDataObjectManager<T>`. Níže je registrace rozhraní `IVehicleManager`, které standardní rozhraní `IDataObjectManager<VehicleDO>` rozšiřuje o některé metody. Díky funkci `Forward` je zaručeno, že jak pro závislost `IDataObjectManager<VehicleDO>` tak `IVehicleManager` je vložena singlton instance `VehicleManager`.

```
For(typeof(IDataObjectManager<>)).Use(typeof(FMDataManagerBase<>));
For(typeof(IVehicleManager)).Use(typeof(VehicleManager));
Forward<IVehicleManager, IDataObjectManager<VehicleDO>>();
```

Zdrojový kód 5.7: Příklad registrace StructureMap

## 5.3 Klient

Klientská část aplikace je napsaná technologií HTML5. V této sekci se pokusím popsat základní technologie použité v architektuře a uvést příklady některých důležitých bázových tříd.

### 5.3.1 Typescript

**TypeScript** je typová nadmnožina jazyka JavaScript, která se kompiluje do čistého JavaScript kódu. Je proto kompatibilní se všemi prohlížeči, operačními systémy atd. TypeScript je šířen jako **open source**.

TypeScript podporuje všechny klíčové slova jazyka JavaScript a přidává i svoje vlastní. Díky výstupu v čistém JavaScript kódu lze libovolně kombinovat Typescript a JavaScript kód.

TypeScript obsahuje nejnovější funkce **ECMAScript 2015**, zároveň ale udržuje kompatibilitu se starší **ECMAScript 3** [7].

#### 5.3.1.1 Základní typy

V základu jsou v jazyce TypeScript k dispozici například tyto typy:

- `boolean`
- `number`
- `string`
- `Array`
- `Enum`

#### 5.3.1.2 Rozhraní

Jako příklad rozhraní v TypeScriptu uvedu `IDataModel<T>`. Ve ukázce zdrojovém kódu 5.9 je vidět generický parametr omezený dalším rozhraním IEntity. Proto musí generický parametr implementovat IEntity.

Rozhraní `IDataModel<T>` dále obsahuje položku a metodu. Metoda `getItemById` přijímá numerický parameter `id` a vrací `promise` typu `TEntity`, což je generický parametr rozhraní. Pokud zavoláme metodu `getItemById` na `IDataModel<DeviceDTO>`, vrátí promise typu `angular.IPromise<DeviceDTO>`.

#### 5.3.1.3 Dědičnost

V jazyce Typescript lze dědit a překrývat metody. V příkladě ve zdrojovém kódu 5.10 je bázová třída `CrudMasterControllerBase`, která implementuje metodu `getEntityRequest`. V případě, že je proměná `isTreeViewEnabled`

```
let isDone: boolean = false;

let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

let color: string = "blue";

let list: number[] = [1, 2, 3];

enum Color {Red, Green, Blue};
let c: Color = Color.Green;
```

Zdrojový kód 5.8: Příklady základních datových typů v jazyce TypeScript

```
export interface IDataModel<TEntity extends IEntity>
{
    ...

    /**
    * Occurs when items was removed from cache because was changed.
    */
    itemsUpdated: ILiteEvent<TEntity[]>;

    /**
     * Call this to obtain one entity by id.
     * @param id Id of entity.
     * @returns One entity that math id, null if entity not exist in DB.
     */
    getItemById(id: number): angular.IPromise<TEntity>;

    ...
}
```

Zdrojový kód 5.9: Příklad rozhraní v TypeScriptu

nastavena na `True`, se tato logika rozšíří. V opačném případě se zavolá bázová implementace.

```
export class ConfigurationValuesMasterController
    extends CrudMasterControllerBase<ConfigurationItemValueDTO>
{
    ...

    protected getEntityRequest(pageIndex: number, pageSize: number)
        : IRequest
    {
        if (this.isTreeViewEnabled)
        {
            var request = super.getEntityRequest(pageIndex, 10000)
            request.parentId = this.selectedTreeItemId;
            return request;
        }

        return super.getEntityRequest(pageIndex, pageSize);
    }

    ...
}
```

Zdrojový kód 5.10: Příklad dědičnosti v jazyce TypeScript

#### 5.3.1.4 Properties

Stejně jako v jazyce C# lze v jazyce Typescript vytvářet **Properties**. Zvenku se property jeví jako field, uvnitř třídy se jeví jako dvojice get/set method. Ukázka viz zdrojový kód 5.11.

#### 5.3.1.5 Výchozí hodnoty parametrů

V jazyce TypeScript stejně jako v jazyce JavaScript verze **ES6/ES2015** lze definovat výchozí hodnoty parametrů. V rozhraní lze poté daný parametr označit jako nepovinný. Příklad viz zdrojový kód 5.12.

### 5.3.2 Webpack

**Webpack** je modulární balíčkový systém pro moderní JavaScript aplikace [8]. V projektu FleetwareMaster s ním kompilujeme TypeScript soubory, které

```
public get itemModel(): TItemModel
{
    return this._itemModel;
}
public set itemModel(val: TItemModel)
{
    this._itemModel = val;
    this.fillControllerFromItemModel(val);
}
...

// Z venku
this.itemModel = new ItemModel();
```

Zdrojový kód 5.11: Příklad dědičnosti v jazyce TypeScript

```
export interface ITreeViewSource<TDto extends IDto>
{
   getChildren: (parentId?: number) => angular.IPromise<TDto[]>;
}

export class ConfigurationValuesMasterController
    implements ITreeViewSource<ConfigurationMenuDTO>

    public getChildren(parentId: number = null):
        angular.IPromise<ConfigurationMenuDTO[]>
    {
        ...
    }
}
```

Zdrojový kód 5.12: Příklad výchozích hodnot parametrů (nepovinných parametrů) v TypeScriptu

poté zabalíme a nakonec minifikujeme. Díky tomu prohlížeč stahuje jeden či dva soubory (vlastní skripty a skripty knihoven třetích stran).

Webpack není v projektu nasazen od začátku, původně byl použit **RequireJS**. Princip RequireJS je ten, že se v každém souboru pomocí syntaxe

```
import ConfigurationDTO from "../../dto/ConfigurationDTO"
```

odkazuje na ostatní JS soubory, na kterých je daný soubor závislý. To vyžaduje na místě nasazení udržovat stejnou strukturu, jako na disku při vývoji. To je problém, pokud sdílíte frontend kód ve společnosti.

Sdílení kódu je realizování pomocí sdíleného projektu (GIT repozitáře shared-ui) **Core.UI**. Jedná se o webový projekt, který se nasazuje jako IIS webová aplikace. Na tuto webovou aplikaci se z konkrétního projektu odkazuje (v mém případě **FleetwareMaster.UI**). Při použití **RequireJS** jsem musel nasazovat dvě webové aplikace což bylo nepraktické.

S knihovnou Webpack není struktura disku na nasazení vůbec závislá. Při vývoji se skripty zabalí do jednoho velkého souboru, který se nakopíruje do složky **dist**. Při nasazování se kopíruje pouze složka dist.

#### 5.3.2.1 HTML šablony

V aplikaci používám Webpack plugin pro zabalování HTML šablon do výsledného souboru. Pokud chci odkázat například v Angular direktivě na HTML šablonu, použiji tento zápis:

```
template = require("./crud-toolbar.html");
```

Webpack se postará o zbytek a při nasazení už nemusíme kopírovat všechny HTML soubory.

### 5.3.3 Gulp

Gulp je rozšiřitelný open source JavaScript nástroj, umožnující automatizaci build úkolů [9] jako jsou:

- Kompilace souborů z CSS preprocesorů (LESS, SASS atd.)
- Balení více CSS souborů do větších celků
- Minifikace CSS souborů
- Překopírování různých souborů (i18n resources, fonty, atd.)

Gulp umožňuje zapojovat pluginy takže jeho možnosti lze rozšiřovat.

### 5.3.4 Bower

Bower je balíčkový systém pro knihovny třetích stran. V konfiguračním souboru `bower.json` (viz zdrojový kód 5.13) se definují knihovny a jejich verze. Knihovny se poté stáhnou do složky `bower_packages`, kde se na ně odkazuje ve webpack import souborech.

```
{
  "name": "fleetware_master",
  "dependencies": {
    "ng-file-upload": "^12.2.13",
    "pace-ng-file-upload-inlinepatch": "^0.0.6",
    "angular-toastr": "^2.1.1"
  }
}
```

Zdrojový kód 5.13: Příklad konfigurace boweru

### 5.3.5 AngularJS

**Angular** je komplexní javascriptový framework, určený pro tvoření single page aplikací. Je šířen pod MIT licencí [10] a je vyvíjen pod záštitou firmy **Google**.

V projektu se snažím držet členění kódu do Angular modulů. Příklady názvů modulů v projektu `Authentication`, `ServerAccess`, `Paging`, `Crud` atd.

### 5.3.6 Komunikace s WebAPI

Pro komunikaci s Web API kontrolery jsem vytvořil `DataModelService`. Jedná se o Angular službu, která je tvořena generickou třídou. Generický paremetr musí implementovat rozhraní `IDto`, které poskytuje položku ID s kterou bázová implementace `DataModelService` pracuje. DataModelService poskytuje několik funkcí:

#### 5.3.6.1 GET operace a cache

DataModelService poskytuje `getItems` a `getItemsByFilter`. Metoda getItemsByFilter přijímá filter, který obsahuje například:

- Index stránky
- Velikost stránky
- Textový filtr

- Filtr nad sloupci
- Řazení

Na základě tohoto filtru server vrátí omezenou množinu jedné stránky seřazených dat. Tato metoda ukládá odpovědi serveru do dvou cache.

- **ID cache** - Každý DTO objekt, vrácený ze serveru, uložíme do cache podle ID klíče
- **Filter cache** - Filter request vrací textový řetězec, který slouží jako klíč pro filter cache. Pro každý request se uloží do cache ID všech entit, která byla vrácena.

Při volání `GetItem` nebo `GetItems` je zkontrolována ID cache a pokud je nalezeno požadované ID je vrácen objekt z cache.

V případě, že je zavolána `getItemsByFilter`, metoda zkontroluje filter cache. Pokud záznam najde, vybrané pole s ID všech entit načte z ID cache.

Technologii uložení cache lze měnit, `DataModelService` využívá pouze rozhraní cache služby `IEntityCacheService`.

#### 5.3.6.2 Notifikace o změnách

Při operacích `Add`, `Update` a `Delete` je potřeba promazat záznamy v cache a notifikovat kontrolery. K tomu je použita knihovna **SignalR**. Kontrolery na serveru při operaci PUT/POST/DELETE pošlou notifikaci o změně entity, která obsahuje:

- Název entity
- ID entity
- Uživatel, který vyvolal tuto událost
- Typ operace (Create, Edit, Delete, Invalidate)

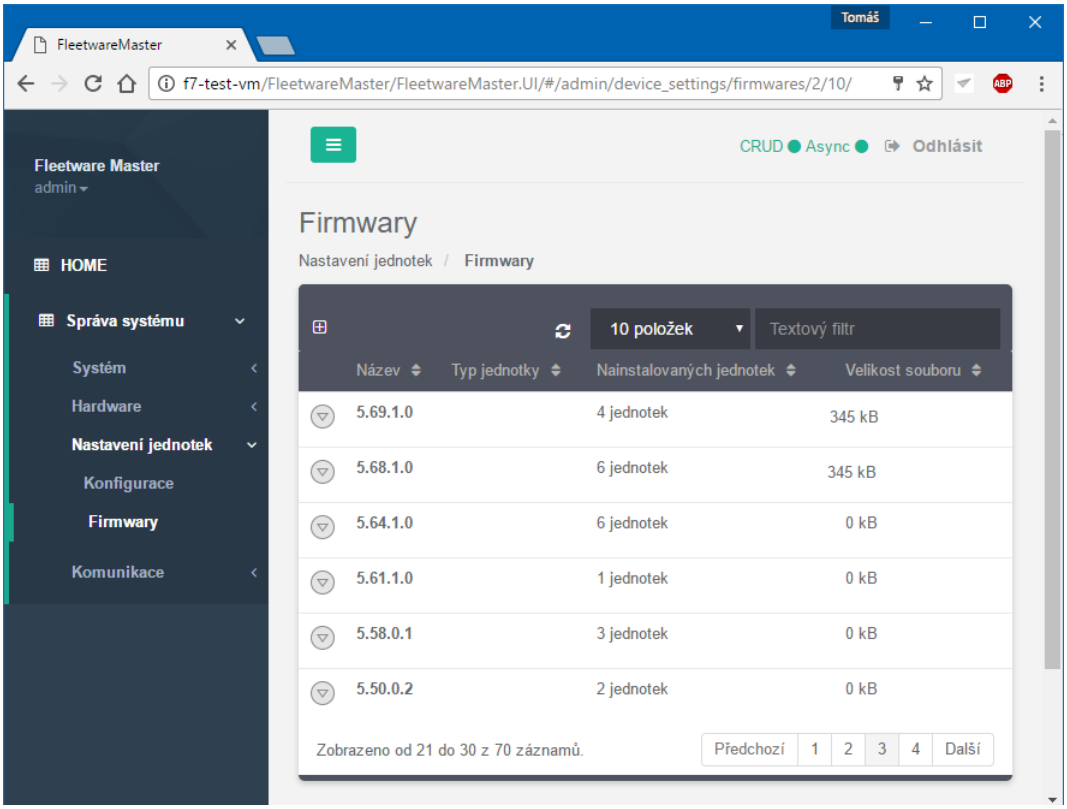
Každá instance `DataModelService` poslouchá příchozí zprávy notifikačního hubu. Pokud přijde zpráva od jiného uživatele, než je přihlášený (nechceme zpracovávat vlastní zprávy dvakrát), promaže cache dané entity a deleguje notifikaci pomocí událostí `ItemCreated`, `ItemUpdated` a `ItemDeleted`. Na tyto události jsou napojení kontrolery a pokud zaznamenají jednu z těchto událostí, vynutí obnovení aktuálního pohledu na data. Protože cache entity je promazaná, načtou se aktuální data ze serveru.

### 5.3.7 Stránkování

Protože ve vetšině aplikacích budeme potřebovat listovat ve zdroji dat čítající tisíce záznamů, je nutné vytvořit logiku podporující nějaký druh stránkování. Sdílená knihovna podporuje dva základní druhy stránkování:

- Stránkování po stránkách pomocí karuselu stránek
- Stealth paging[22]

Nastavená stránka se vybírá v karuselu stránek viz obr. 5.1 vpravo dole. Velikost stránky se nastavuje v liště nahoře vpravo. Toto nastavení se spolu s nastavením řazení, filtry, vybranými položkami atd. propisuje do url. Díky tomu uživatel může kdykoliv vzít url z adresního řádku a poslat ho druhé osobě, které se aplikace zobrazí v totožném stavu.

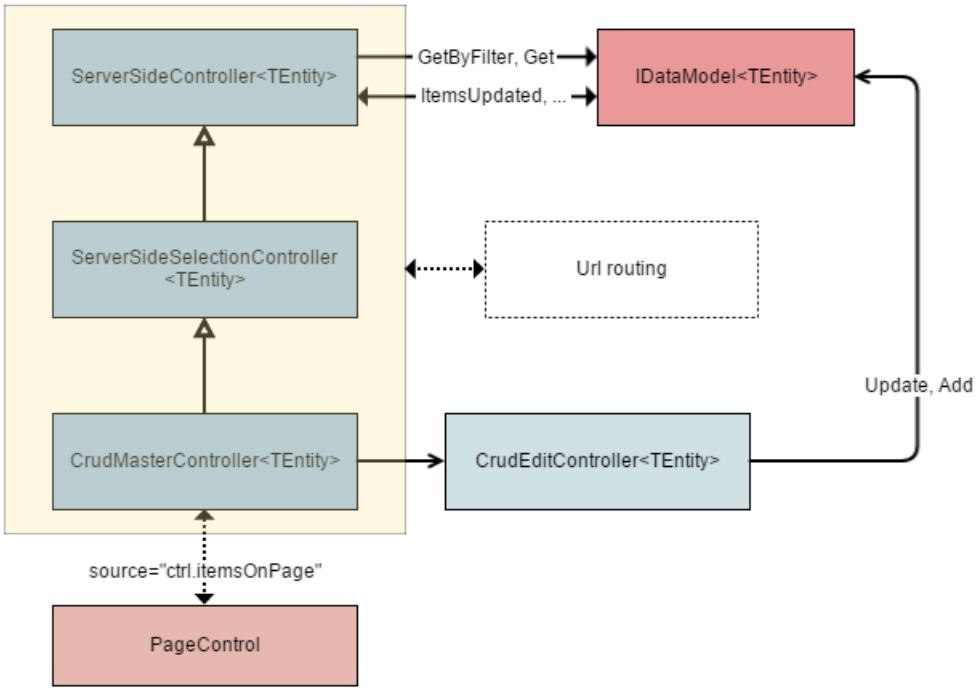



Obrázek 5.1: Stránkování v aplikaci

[22] **Stealth paging** - virtuální vertikální posuvník, který donačítá data tak, jak uživatel posouvá data

### 5.3.8 Bázové kontrolery

Protože velká část všech obrazovek obsahuje CRUD operace se stránkováním, vytvořil jsem několik bázových kontrolerů, kteří v bázi tyto operace poskytují a zabraňuji tím duplikaci kódu. Hierarchie tříd viz obr. 5.2.




Obrázek 5.2: Hierarchie tříd CRUD kontrolerů

#### 5.3.8.1 ServerSideController

`ServerSideController<TEntity>` má na starosti načítání stránkovaných dat ze serveru. K tomu mu slouží (dodaný přes DI[23]) `IDataModelService<TEntity>`. Kontroler pracuje se stránkováním, řazením, filtrováním, textovým filtrem a url. Pokud přijde notifikace o změně dat, na které se uživatel dívá, obnoví data na stránce.

#### 5.3.8.2 ServerSideSelectionController

`ServerSideSelectionController<TEntity>` rozšiřuje předka o logiku označení položek. Umožňuje mód označení jedné nebo více položek. ID označených položek propisuje do url.

[23] **DI** - Dependency injection

#### 5.3.8.3 CrudMasterController

`CrudMasterController<TEntity>` přidává navíc podporu editace položky. K tomu využivá `CrudEditController<TEntity>`. V případě editace propisuje ID editované položky do url. Díky tomu lze poslat url obrazovky s editovanou položkou viz obr. C.3.3.

#### 5.3.8.4 CrudEditController

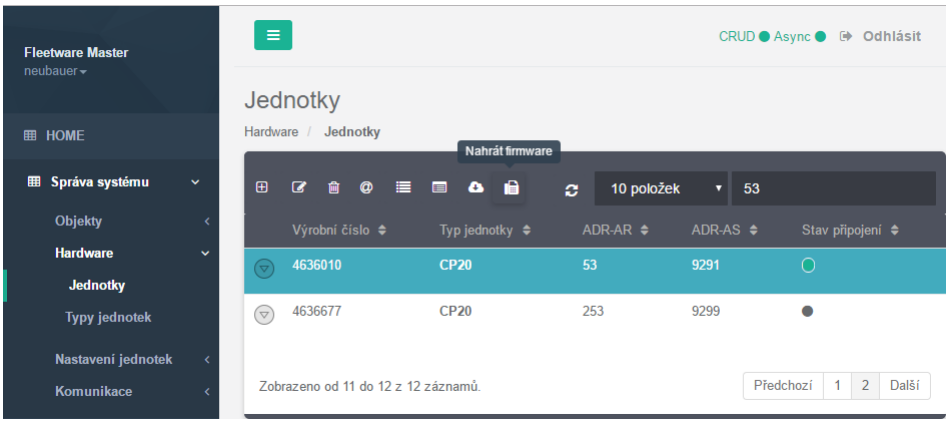
`CrudEditController<TEntity>` je abstraktní bází pro všechny edit kontrolery editačních dialogů viz obr. C.3.3. Umožňuje validaci a implementuje metodu `Save` a `Cancel`.

## 5.4 Asynchronií operace

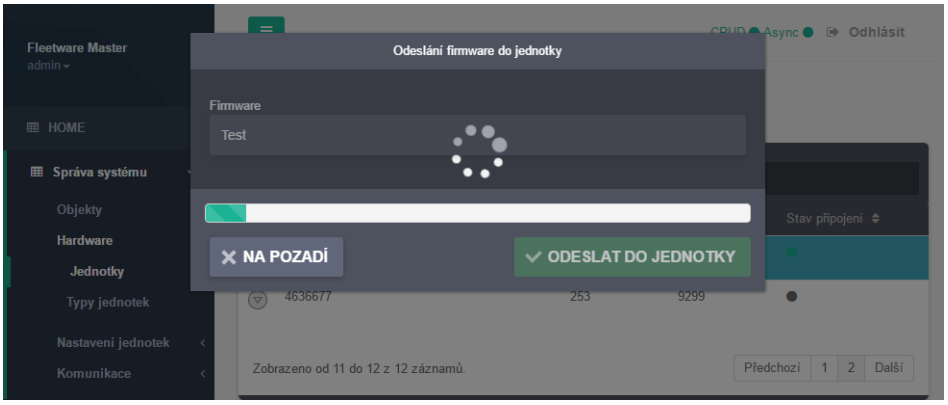
**Asynchronií operace** jsou důležitou částí projektu. Většina hlavních operací jsou časově náročné operace, je proto pohodlnější je provádět asynchronně než na ně čekat v dialogu. Při nahrávání firmwaru viz 5.4.1.1 a 5.4.1.2 je k dispozici tlačítko **Na pozadí**. Tímto tlačítkem uživatel zavře dialog a obdrží notifikaci, že asynchronní operace byla zaregistrována a může pokračovat dále v práci viz 5.4.1.3. Pokud by se uživatel chtěl podívat na stav všech asynchroniích operací, je mu k dispozici přehled viz 5.4.1.4.

### 5.4.1 Ukázka nahrátí nového firmware

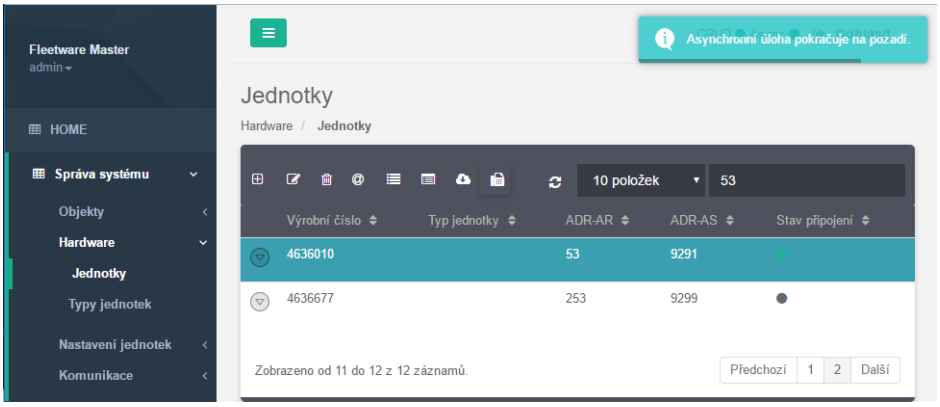
#### 5.4.1.1 Nalezení cílové jednotky



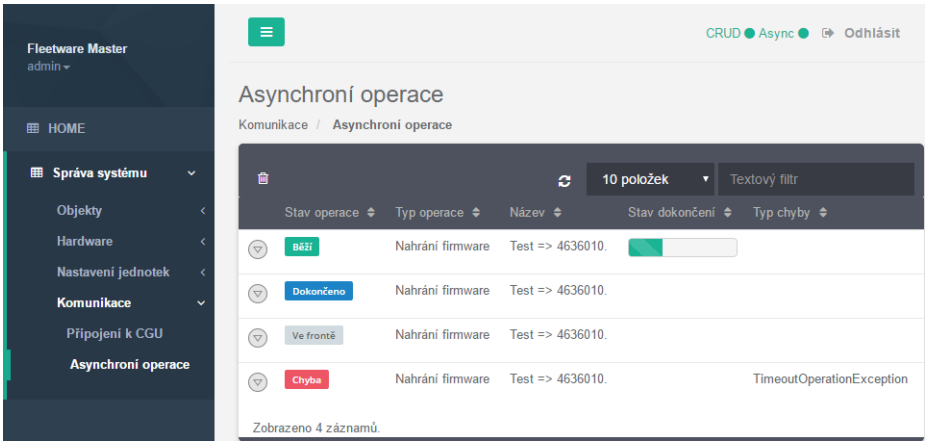
#### 5.4.1.2 Výběr zdrojové firmware




#### 5.4.1.3 Registrace operace na pozadí




#### 5.4.1.4 Přehled všech asynchronních operací



### 5.4.2 Implementace na serveru

Klíčové třídy jsou `AsyncOperationBase` a `AsyncOperationProvider`.

#### 5.4.2.1 AsyncOperationBase

`AsyncOperationBase` je abstraktní předek všech asynchroniích úloh. Zdrojový kód viz v příloze zdrovojé kódy B.1.1. Tento předek definuje abstraktní metody `BeforeOperation` a `AfterOperation`, které umožní operaci připravit si data a po dokončení data uložit. Obě metody jsou obaleny databázovým kontextem, tudíž v těchto metodách lze přistupovat k DB. `BeforeOperation` není obalena transakcí, protože tato metoda slouží k načtení dat. `AfterOperation` naopak obalena transakcí je, obvykle se využívá k uložení výsledků operace.

Metoda `AsyncOperation` slouží k zavolání asynchronní operace. Zde již není přístup k DB. Tato operace je spuštěna na pozadí a očekává se, že bude trvat delší časový úsek.

Příklad konkrétní implementace viz příloha zdrojové kódy B.1.2. V konstruktoru jsou dva určující parametry úlohy, zdrojový firmware a cílová jednotka. Dále v `BeforeOperation` dochází ke kompletnímu načtení těchto dvou entit z DB. V `AsyncOperation` je zavolána **komunikační knihovna** a spuštěna asynchronní operace nahrání nového firmware. Po dokončení operace je v metodě `AfterOperation` uložena změna firmware v DB.

#### 5.4.2.2 AsyncOperationProvider

`AsyncOperationProvider` organizuje asynchronií úlohy a stará se o jejich registrování. Zdrojový kód viz příloha zdrojové kódy B.1.3. V metodě `RunOperationAsync` se vytvoří záznam evidující asynchronií operaci v DB a operace je spuštěna na pozadí.

### 5.4.3 Hromadné operace

Další využití asynchronních operací jsou hromadné operace. Příkladem může být nahráti firmware či konfigurace na celou skupinu vozidel. Asynchronní operace se mohou sdružovat do stromu. Jsou řešeny `AsyncOperationGroup`, potomkem `AsyncOperationBase`. Spustí paralelně všechny podoperace a celkový postup (progress) je počítán podílově ze všech podoperací.

### 5.4.4 Plánování úloh

Přidanou funkcionalitou (která není v době psaní tohoto dokumentu kompletně hotová) je plánování asynchronních úloh. Uživatel má na výběr časové období a možnost opakování.

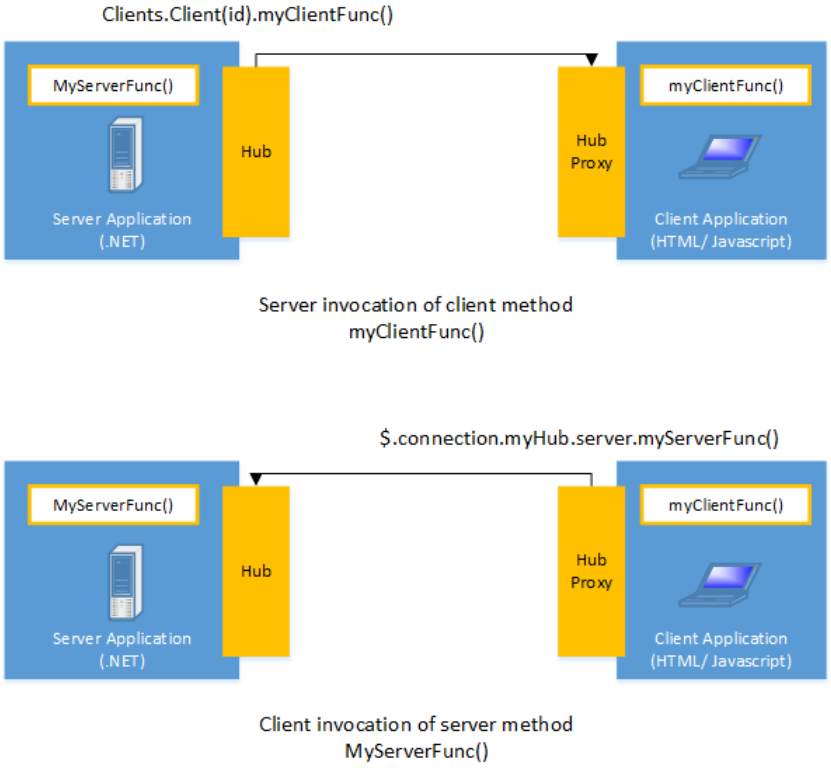
#### 5.4.4.1 Příklad užití

Vozidla flotily jsou organizována podle směn. První skupina jezdí v hodinách 8 - 16, druhá potom v rozmezí 16 - 24. Uživatel provádí asynchronní operaci podle toho, kdy auto jezdí a kdy ne. Označí skupinu a naplánuje operaci hromadně na dobu, kdy jednotka nebude (nebo bude) v provozu.

## 5.5 SignalR

ASP .NET SignalR je knihovna od společnosti Microsoft, která zjednodušuje proces přidávání real-time funcionality do webových aplikací [11]. Tato funkce umožnuje serveru poslat data webovým klientům ihned, když jsou k dispozici.

Existují dvě varianty použití SignalR. Volání klientských metod ze serveru nebo volání server metod z klientů. V naší aplikaci jsme využili pouze první variantu. Schéma tohoto volání viz obr. 5.3.




Obrázek 5.3: Schéma volání SignalR metod na serveru či klientu.

V tomto projektu je v tuto chvíli používán **SignalR** ve dvou případech.

- Notifikace o změnách entit (Add, Update, Delete)
- Změna stavu běžících asynchroniích operací

SignalR umožňuje pro komunikaci použít několik komunikačních protokolů. Vždy zvolí protokol podle aktuální situace. Odstiňuje tedy vývojáře od způsobu komunikace serveru s klienty. Komunikační protokoly jsou:

- HTML 5
  - WebSocket
  - Server Sent Events
- Comet
  - Forever Frame
  - Ajax long polling

Stav připojení SignalR hubů může uživatel v aplikaci sledovat pomocí semaforů vpravo nahoře viz obr. 5.1.

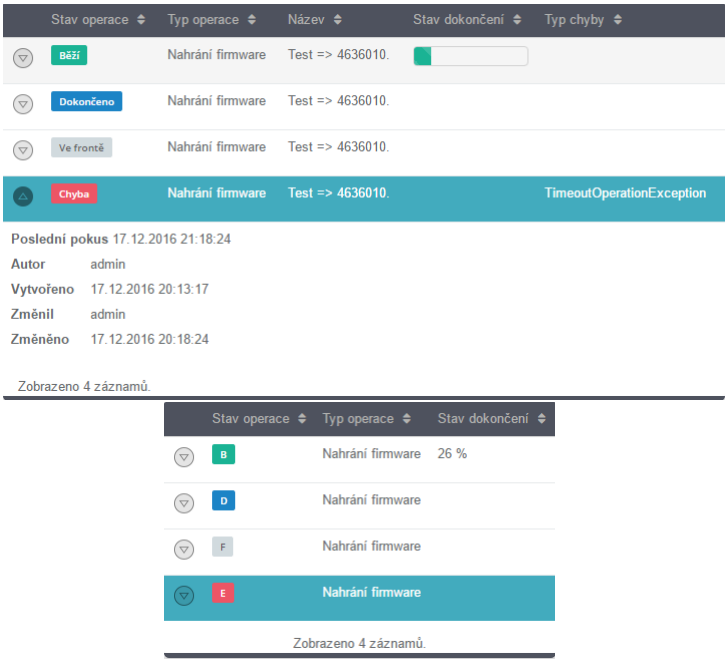
## 5.6 Komunikační knihovna

Komunikační knihovna je .NET projekt, jehož výstupem je DLL[24] soubor. Serverová část aplikace se napojuje na tuto knihovnu a používá jí pro komunikaci se SIM kartami v jednotkách. Autorem je stejný vývojář, jako u firmware v jednotkách. Používá nejnovější .NET 4.5, **ReactiveExtensions** a je kompletně napsaná asynchronně.

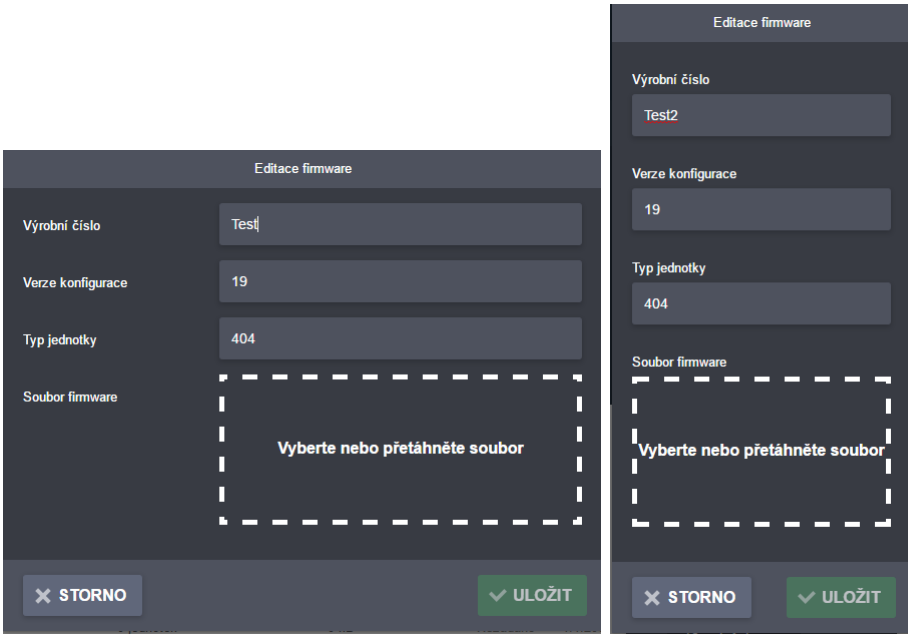
## 5.7 Responzibilní design

Protože aplikace bude používána na různých typech zařízení, design aplikace je od začátku koncipován responzivně. Toho je docíleno použitím gridového systému knihovny **Bootstrap**. Jako příklad uvedu dvě situace viz obr. 5.5 a obr. 5.5.

[24] **DLL** - Dynamic link library




Obrázek 5.4: Zobrazení dat v tabulce pro větší obrazovku

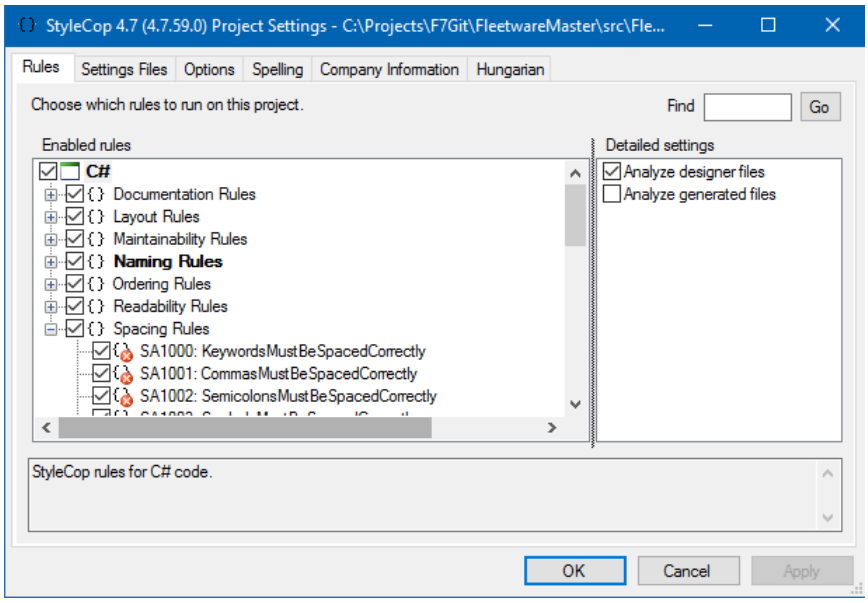



Obrázek 5.5: Responzibilní design v editačním dialogu

## 5.8 Kontrola čistoty kódu

Čistotu kódu zajišťuje **StyleCop**. Jedná se o konfigurovatelný plugin do Visual Studia, který hlídá konvence, komentáře, pořadí položek, konstruktorů, metod atd.

Pokud dojde k nedodržení některého pravidla, aplikace nelze zkompilovat. Pravidla lze vypínat, viz obr. 5.6.




Obrázek 5.6: Nastavení pravidel kontroly čistoty kódu v StyleCop pluginu ve Visual studiu

Kapitola 6

# Databáze

V této kapitole nejdříve představím architekturu databáze. Způsob návrhu architektury demonstruji na hlavní části DB - uložení konfigurace. Verzování databáze a udržování testovacích dat rozeberu v další sekci.

## 6.1 Architektura

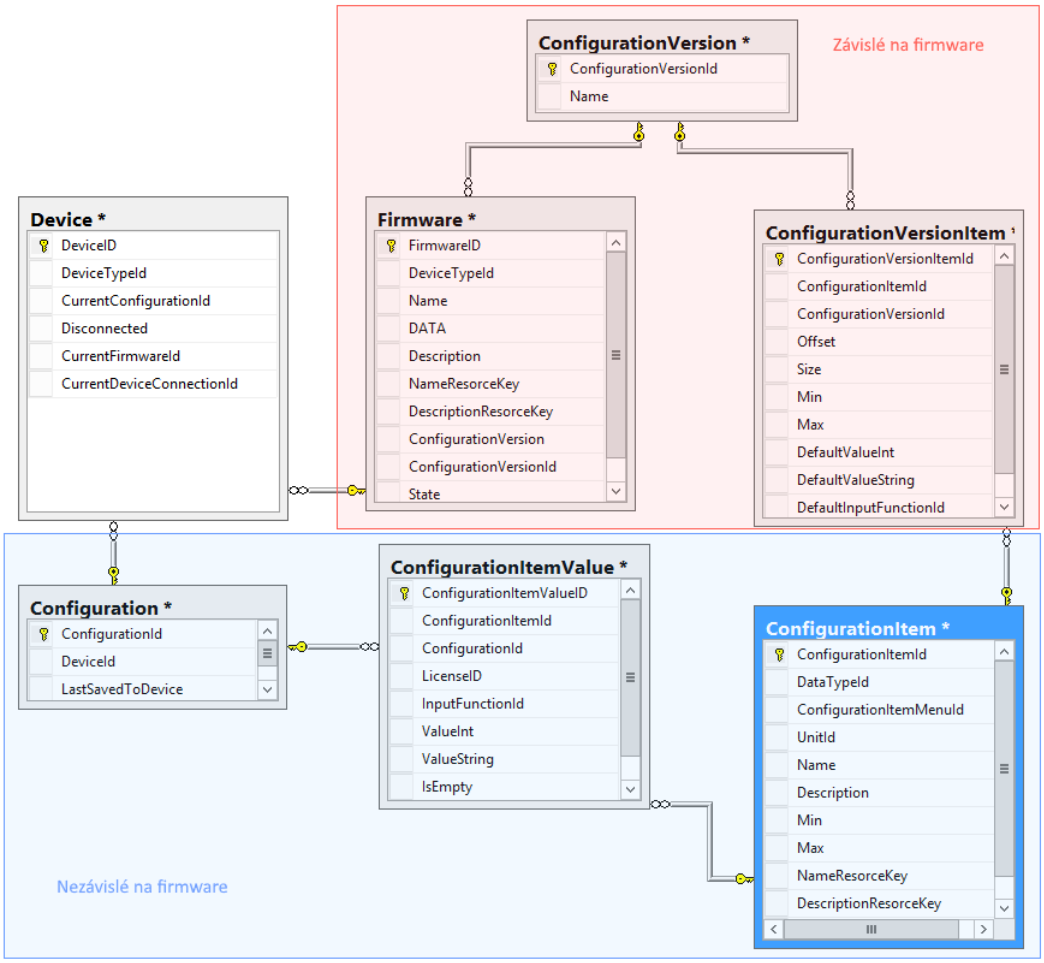
Při návrhu architektury databáze jsem se řídil doporučenými postupy pro návrh relačních databázi, které jsem se naučil v předmětech **BI-DBS** a **MI-DSP**. Data v tabulkách jsou uložena podle interpretace dat v reálném světě, nikoliv v podobě, jak přijdou z jednotek. Jedním z důvodů je možnost v budoucnu měnit podobu, jak data do jednotek posílíme a jak je přijímáme, aniž bychom museli konvertovat data v databázi.

### 6.1.1 Příklad

Konfigurace je přijímána a odesílána do jednotek v bytovém poli. Kde a jak jsou jednotlivé položky v bytovém poli uloženy, určuje verze konfigurace. Každá verze konfigurace má svůj definiční soubor, který určuje reprezentaci každého bytu v poli. Tento soubor je možné naimportovat do systému, takže se nemusí ručně přepisovat do systému FleetwareMaster. S každým firmwarem se tato verze může změnit, sémantický význam hodnot konfiguračních parametrů je ale stejný.

Bytové pole se při přijetí převede na hodnoty, které se uloží do databáze abstrahované od podoby jak přišly. Při odeslání konfigurace zpět do jednotky se tyto hodnoty složí zpět podle toho, jakou verzi konfigurace cílová jednotka používá.

Na obrázku 6.1 je modře odlišena část datového modelu, kde jsou konfigurace uloženy abstrahovaně od způsobu uložení v jednotce. Červeně odlišená část reprezentuje způsob, jak tyto hodnoty převést do konkrétní verze firmware.




Obrázek 6.1: DB architektura uložení konfigurace

Reprezentace jednotlivých tabulek jsou:

#### 6.1.1.1 Abstraktní část

- **Configuration** - reprezentuje jednu konfiguraci (v jednotce, historickou, koncept). Každá konfigurace má stovky hodnot.

- **ConfigurationItem** - Typ položky konfigurace, tedy její sémantický význam. Těchto položek je konečný počet (cca 500). Jako příklad uvedu **Timeout připojení GSM**, **Adresa sítě**, **APN** atd. Ke každé položce může připadat výchozí hodnota, minimální a maximální hodnota a název s nápovědou pro uživatele.

- **ConfigurationItemValue** - Reprezentuje hodnoty položek každé konfigurace. Podle datového typu je hodnota uložena v `ValueInt`, `ValueString` nebo `InputFunctionId`.

#### 6.1.1.2 Část závislá na firmware

- **ConfigurationVersion** - Verze konfigurace. Každý firmware používá jednu verzi konfigurace, která obsahuje informace o způsobu uložení položek konfigurace v bytovém poli.
- **ConfigurationVersionItem** - Informace o umístění konfigurační položky v bytovém poli. Nejduležitější sloupce jsou `Offset` a `Size`.

### 6.1.2 Příklad

V tabulce je popsaná vybraná položka konkrétní konfigurace v dané verzi konfigurace. V ukázce 6.14 je uveden LINQ dotaz, který pro uvedený příklad dohledá, v jakých bytech bude položka uložena. Následuje SQL varianta tohoto LINQ dotazu viz 6.15.

```
var configurationValues =
    configurationValuesManager.GetAllItemsQuery();

var configurationVersionItems =
    configurationVersionItemManager.GetAllItemsQuery();

from itemValue in configurationValues
join versionItem in configurationVersionItems
on itemValue.ConfigurationItemId equals versionItem.ConfigurationItemId
where itemValue.ConfigurationId == 381
where itemValue.ConfigurationItem.Code == "GPRSDelay"
where versionItem.ConfigurationVersion.Name == "19"
select new
{
    itemValue.ConfigurationItem.Name,
    itemValue.ValueInt,
    versionItem.Offset,
    versionItem.Size
}
```

Zdrojový kód 6.14: Příklad dotazu pro zjištění offsetu a velikosti pole bytů ve výsledném bytovém poli v LINQ dotazu jazyka C#

| | |
|---|---|
| **Configuration** | |
| ID | 381 |
| LastSavedToDevice | '2016-12-25' |
| **ConfigurationItem** | |
| ID | 10 |
| DateTypeId | 1 (Datový typ číslo) |
| Code | GPRSDelay |
| Name | Odložení GPRS komunikace |
| Min | 0 |
| Max | NULL |
| **ConfigurationItemValue** | |
| ID | 15 |
| ValueInt | 180 |
| ConfigurationId | 5 |
| ConfigurationItemId | 10 |
| **ConfigurationVersion** | |
| ID | 1 |
| Name | 19 |
| **ConfigurationVersionItem** | |
| ID | 30 |
| ConfigurationItemId | 10 |
| ConfigurationVersionId | 1 |
| Offset | 50 |
| Size | 32 |

Tabulka 6.1: Příklad konkrétní položky konfigurace v DB.

```
SELECT CI.Name, CIV.ValueInt, CVI.Offset, CVI.Size
FROM ConfigurationItemValue as CIV
JOIN ConfigurationItem as CI
    ON CIV.ConfigurationItemId = CI.ConfigurationItemId
JOIN ConfigurationVersionItem AS CVI
    ON CVI.ConfigurationItemId = CI.ConfigurationItemId
JOIN ConfigurationVersion as CV
    ON CV.ConfigurationVersionId = CVI.ConfigurationVersionId
WHERE CI.Code like 'GPRSDelay' AND
      CIV.ConfigurationId = 381 AND
      CV.Name like '19'
```

Zdrojový kód 6.15: Ekvivalent LINQ dotazu pro zjištění offsetu a velikosti pole bytů v SQL

## 6.2 Verzování a nasazování DB

Pro bezproblémové začlenění vývoje databáze do **Continious Integration** jsem navrhl proces verzování databáze a nasazovacích procesů tak, aby vývoj databáze nebyl oddělen od ostatního vývoje. Tento systém se skládá z několika částí.

### 6.2.1 Inicializační část

Tato část obsahuje soubory, které vytvoří postupně kompletní strukturu databáze, například tabulky, cizí klíče, triggery, stored procedury, atd. V této části se vytvoří databáze verze 1. Soubory ve složce jsou například:

- `010_CreateDatabase.sql`
- `050_Tables.sql`
- `051_Users.sql`
- `060_PrimaryKeys.sql`
- `070_ForeignKeys.sql`
- `089_Triggers`
- `095_SetNoCountOff.sql`

### 6.2.2 Migrační skripty

V této části jsou ve složkách číselně odlišené migrační skripty pro různé verze databáze. Každá verze řeší povýšení databáze o jednu verzi. Pokud je potřeba upgrade z verze 15 na verzi 19, spustí se skripty ze složek v pořadí 16, 17, 18 a 19.

V každé složce verze jsou skripty, které přidávají, upravují nebo odebírají různé prvky struktury databáze. Jako poslední soubor je v každé složce umístěn `999-version.sql`. Tento soubor provádí update tabulky `Version`, která slouží pro migrační skripty jako kontrola, že jsou spouštěny nad databází požadované verze. Jako příklad uvedu přidání tabulky `AsyncOperationType` ve verzi 6 viz zdrojový kód 6.16.

### 6.2.3 Data

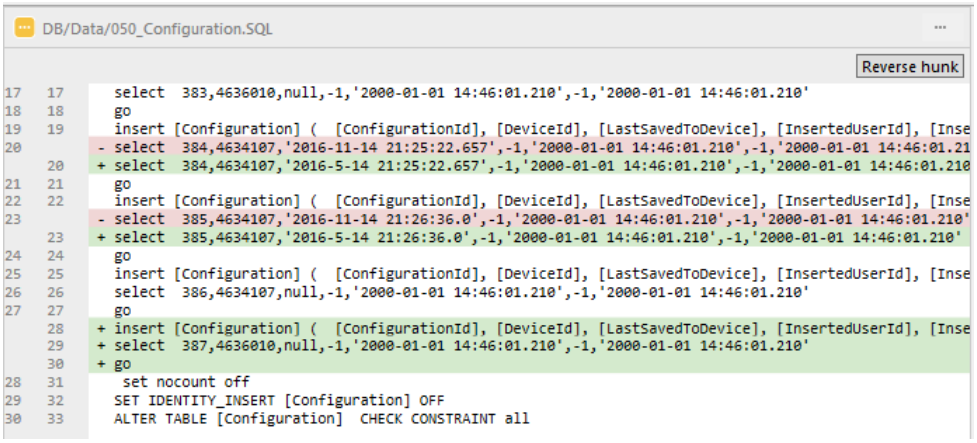
Složka data obsahuje sql insert skripty, které pro zvolený komit GITu nahrají testovací data. Unit testy spoléhají na tyto data, takže vždy testujeme nad konzistentními daty.

```
declare @Version int
select @Version=5

if(   not exists(select * from  [Version] where [VersionId] = @Version  )
 or       exists(select * from  [Version] where [VersionId] > @Version  )
)
begin
   raiserror( '50009; DB version %i is required.', @Version)
end
else
begin

create table AsyncOperationType
(
    AsyncOperationTypeId bigint not null  --nebude identity
   ,[Name]                           varchar(100)     not  null
   ,[NameResorceKey]                 varchar(30)      null
   )

ALTER table AsyncOperationType
ADD constraint PK_AsyncOperationType primary key (AsyncOperationTypeId)

ALTER TABLE [dbo].[AsyncOperationType]
ADD [ModifiedDate] [datetime] not  null

ALTER TABLE [dbo].[AsyncOperationType]
ADD [InsertedDate] [datetime] not null

ALTER TABLE [dbo].[AsyncOperationType]
ADD [ModifiedUserId] [bigint] not null

ALTER TABLE [AsyncOperationType]
ADD constraint  FK_AsyncOperationType_User_ModifiedUserId
FOREIGN KEY (ModifiedUserId) REFERENCES [User](UserId)

ALTER TABLE [dbo].[AsyncOperationType]
ADD [InsertedUserId] [bigint] not null

ALTER TABLE [AsyncOperationType]
ADD constraint  FK_AsyncOperationType_User_InsertedUserId
FOREIGN KEY (InsertedUserId) REFERENCES [User](UserId)
```

Zdrojový kód 6.16: Migrační skript kontrolující aktuální verzi DB

#### 6.2.3.1 Tvorba testovacích dat

Pro účely tvorby testovacích dat byla vytvořena pomocná aplikace, která přečte ze všech nesystémových tabulek data, a do zvoleného adresáře zapíše pro každou tabulku soubor s inserty řádků. GIT rozpozná změny v souborech s testovacími daty a v komitu jsou zahrnuty pouze řádky, kde se změnila testovací data. Ukázka viz obr. 6.2.




Obrázek 6.2: Změna v testovacích datech v GITu

### 6.2.4 Kontrola

Při každé změně sktuktury databáze nebo úpravě testovacích dat se automaticky spustí `FleetwareMaster.Build Including Unit Testing viz 3.3.1`, který provede drop databáze a nové kompletní sestavení. Dále se spustí všechny unit testy a dojde ke kontrole, jestli dané změny ve skriptech neobsahují chyby.

Pokud by build nebo unit testy skončily s chybou, potenciálnímu autorovi chyby je odeslán automaticky email.

Kapitola 7

# Splnění dílčích úkolů zadání

Aplikace je dnes ve funkčním stavu a většina funkcí zahrnutých do obsahu této práce je hotova. Kompletní výpis úkolů ze zadání a jejich ověření splnění:

1. *Vyjděte ze základních požadavků, jež jsou:*
   a) *Uživatelské rozhraní použitelné na tabletech v terénu i v místech s horší rychlostí datové komunikace*
      - Výsledná aplikace **FleetwareMaster** je použitelná na tabletech v terénu díky responzivnímu designu a SPA architektuře (viz 2.2.4) klientské části.
   b) *Asynchronní hromadné operace s GPS jednotkami.*
      - Implementace asynchronních operací viz 5.4.
   c) *Podpora uživatelských práv.*
      - Tato část je částečně hotova v podobě datového modelu v databázi, serverová a klientská část není hotova.
   d) *Historie operací nad GPS jednotkami.*
      - Tato část je splněna obrazovkou konfigurace v aplikaci v **Správa systému/Nastavení jednotek/Konfigurace**
   e) *Navázání na existující nízkoúrovňovou knihovnu pro komunikaci s GPS jednotkami.*
      - Splněno, viz celá kapitola 5
2. *Proveďte rešerši možných technologií a zvolte vhodné.*
   - Splněno, viz kapitola 2.
3. *Navrhněte softwarovou architekturu kompletního řešení.*
   - Splněno viz celá kapitola 5.

4. *Návrh implementujte tak, že výsledkem bude framework zastřešující řízení a správu GPS jednotek (viz požadavky v bodu 1).*

   - Splněno aplikací **FleetwareMaster** a sdílenou knihovnou **Core.UI**.

5. *Na vhodně zvoleném dílčím projektu demonstrujte nasazení výše zmíněného frameworku.*

   - Splněno aplikací **FleetwareMaster**.

# Závěr

Jak jsem podrobněji rozepsal v kapitole 7, všechny dílčí úkoly z formálního zadání práce byly splněny

Po více než roce práce nad projektem bych na závěr chtěl shrnout několik poznatků a pocitů, které jsem nabyl. Dále uvedu několik statistik, které jsem vyčetl z různých dostupných reportů.

V rámci této práce jsem si vyzkoušel spoustu teoretických poznatků a nápadů v praxi (ze studia na ČVUT FIT nebo vlastních). Spoustu vědomostí, které jsem během studia nabyl jsem mohl zúročit.

Za největší úspěch pokládám proces verzování a nasazování databáze, kde jsem měl možnost dlouho plánovaný systém uplatnit v praxi. Systém se ukázal jako velice efektivní a flexibilní a jeho nasazení je zvažováno i na budoucí projekty firmy.

Velkou novinkou byl pro mně webový vývoj Single page aplikací, který mě pohltil a nadchnul. Ze začátku to byla velká výzva. Webovému vývoji jsem se před tímto projektem 6 let nevěnoval, spíše jsem se pohyboval okolo vývoje desktopových aplikací či technologie Silverlight. Za tu dobu se ve webovém vývoji hodně věcí změnilo. Nastoupit do rozjetého vlaku dnešního frontend vývoje tak byla hlavně ze začátku velice nepříjemná zkušenost. Po čase jsem se ale rozkoukal a našel v kombinaci zvolených technologii (Typescript + Angular + Bootstrap) zalíbení.

Novou technologii byla pro mně také kombinace GITu, GITFlow a Bitbucket SourceTree. Tato technologie a metodika se mi velice zalíbily. V kombinaci s Teamcity a systémem verzování databáze se jedná o velice sofistikovaný model řízení vývoje software, který bych určitě doporučil.

Na úplný závěr bych chtěl říct, že poslední rok strávený na tomto projektu byl pro mně velice přínosným a jsem moc rád za tuto příležitost. Chtěl bych určitě poděkovat vedoucímu práce Pavlu Dvořákovi, díky kterému jsem tuto příležitost dostal a který mi dal volnou ruku realizovat moje myšlenky na poli softwarového inženýrství a řízení softwarového projektu. Dále bych poděkoval všem kolegům, kteří na projektu se mnou spolupracovali.

## Statistiky

- Na projektu **FleetwareMaster** je v systému **JIRA** zalogováno 1316 hodin práce.
- GIT repozitář **FleetwareMaster** obsahuje cca 30 000 řádků psaného kódu.
  - Z toho cca 20 000 řádků jsem autorem
  - Z toho cca 10 000 řádků je komunikační knihovna jenž nejsem autorem
- GIT repozitář **Shared-UI** obsahuje cca 20 000 řádků kódu.
  - Ze začátku čistě můj kód, později díky připojení projektu **FleetwareWeb2** více autorů.
- GIT repozitář **Shared-Server** obsahuje cca 50 000 řádků
  - Ačkoliv jsem do této knihovny přispěl spousty kódy, většina byla mimo rámec této práce.
  - Významnou změnou v této knihovně v rámci tohoto projektu bylo přidání podpory asynchroniích operací ve spolupráci s logikou databázového kontextu.

# Literatura


[1] Google: AngularJS API Docs Scope. 2016. Dostupné z: https://docs.angularjs.org/guide/scope

[2] The repository for high quality TypeScript type definitions. 2016. Dostupné z: https://github.com/DefinitelyTyped/DefinitelyTyped

[3] Wikipedie: AngularJS. 2016. Dostupné z: https://en.wikipedia.org/wiki/AngularJS#Angular_2

[4] Google: AngularJS API Docs. 2016. Dostupné z: https://docs.angularjs.org

[5] Attlasian: Comparing Workflows. 2016. Dostupné z: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[6] Conel: ARNEP. 2005. Dostupné z: http://www.ok1mjo.com/all/conel/arnep_2001-04-02.pdf

[7] Microsoft: TypeScript. 2016. Dostupné z: https://www.typescriptlang.org/

[8] Collective, O.: Webpack. 2016. Dostupné z: https://webpack.js.org

[9] Innovations, F.: Gulp. 2016. Dostupné z: http://gulpjs.com/

[10] Google: Angular. 2016. Dostupné z: http://stackoverflow.com/questions/39200309/what-is-code-over-configuration

[11] Microsoft: SignalR. 2014. Dostupné z: https://www.asp.net/signalr/overview/getting-started/introduction-to-signalr

Příloha **A**

# Seznam použitých zkratek

**DTO** Data Transfer Object

**DO** Data object (ORM namapované třídy)

**JSON** JavaScript Object Notation

**ORM** Objektově relační mapování

**LINQ** Language Integrated Query

**DLL** Dynamic link library

Příloha B

# Zdrojové kódy

## B.1 Asynchronní operace

### B.1.1 AsyncOperationBase

```
/// <summary>
/// Abstract base class for asynchronious operations.
/// </summary>
public abstract class AsyncOperationBase
{
    private readonly ITransactionContextHelper transactionContextHelpler;

    /// <summary>
    /// CTOR.
    /// </summary>
    protected AsyncOperationBase()
    {
        transactionContextHelpler =
            ServiceLocator.Current.GetInstance<ITransactionContextHelper>();
    }

    /// <summary>
    /// Operation progress report.
    /// </summary>
    public IProgress<ProgressValue> Progress { get; private set; }

    /// <summary>
    /// Operation state at the moment.
    /// </summary>
    public AsyncOperationState AsyncOperationState { get; protected set; }
        = AsyncOperationState.Pending;

    /// <summary>
    /// Async operation type. (Firmware load, ...).
    /// </summary>
    public abstract AsyncOperationType AsyncOperationType { get; }
```

```
/// <summary>
/// Message displayed to user when operation is completed succesfully.
/// </summary>
public abstract string CompleteMessage { get; }

/// <summary>
/// Name of operation.
/// </summary>
public abstract string Name { get; }

/// <summary>
/// Collection of child operations.
/// Every async operation could be a container of other async operations.
/// </summary>
public List<AsyncOperationBase> ChildOperations { get; protected set; }

/// <summary>
/// Call this to execute async operation in background.
/// </summary>
/// <param name="progress">This is provided by <see cref="AsyncOperationProvider"/></param>.
/// <returns>Task of async operation.</returns>
public async Task Run(IProgress<ProgressValue> progress)
{
    Progress = progress;

    AsyncOperationState = AsyncOperationState.InProgress;

    using (TransactionSynchronizationContextHelper.SetNewContext())
    using (transactionContextHelpler.GetOrCreateContext(false))
    {
        BeforeOperation();
    }

    await AsyncOperation(progress);

    using (TransactionSynchronizationContextHelper.SetNewContext())
    using (transactionContextHelpler.GetOrCreateContext(true))
    {
        AfterOperation();
    }

    AsyncOperationState = AsyncOperationState.Successful;
}

/// <summary>
/// This is called before async operation is executed.
/// You can prepare data or read something from DB.
/// </summary>
protected abstract void BeforeOperation();

/// <summary>
/// Here you should implement call of asynchronious job.
/// </summary>
```

```
    /// <param name="progress">
    /// Progress value reporter that you pass to <see cref="IOperationProvider"/>
    /// or you can report progress manually.</param>
    /// <returns>Task of async operation called in this method.</returns>
    protected abstract Task AsyncOperation(IProgress<ProgressValue> progress);

    /// <summary>
    /// This is called after async operation is completed.
    /// You can save the work to DB.
    /// </summary>
    protected abstract void AfterOperation();
}
```

### B.1.2 DeviceFirmwareLoader

```
/// <summary>
/// Firmware loader async operation.
/// </summary>
public class DeviceFirmwareLoader : AsyncOperationBase
{
    private readonly long deviceId;
    private readonly long firmwareId;

    private readonly IDataObjectManager<DeviceDO> deviceManager;
    private readonly IDataObjectManager<FirmwareDO> firmwareManager;
    private readonly IDataObjectManager<DeviceFirmwareDO> deviceFirmwareManager;
    private readonly IOperationProvider operationProvider;
    private DeviceDO device;
    private FirmwareDO firmware;

    /// <summary>
    /// CTOR.
    /// <param name="deviceId">Targed device ID.</param>
    /// <param name="firmwareId">Source firmware ID.</param>
    /// </summary>
    public DeviceFirmwareLoader(long deviceId, long firmwareId)
    {
        this.deviceId = deviceId;
        this.firmwareId = firmwareId;

        deviceManager =
            ServiceLocator.Current.GetInstance<IDataObjectManager<DeviceDO>>();

        firmwareManager =
            ServiceLocator.Current.GetInstance<IDataObjectManager<FirmwareDO>>();

        deviceFirmwareManager =
            ServiceLocator.Current.GetInstance<IDataObjectManager<DeviceFirmwareDO>>();

        operationProvider =
```

```
            ServiceLocator.Current.GetInstance<IOperationProvider>();
    }

    /// <summary>
    /// Async operation type. (Firmware load, ...).
    /// </summary>
    public override AsyncOperationType AsyncOperationType
        => AsyncOperationType.FirmwareLoad;

    /// <summary>
    /// Message displayed to user when operation is completed succesfully.
    /// </summary>
    public override string CompleteMessage =>
        string.Format(
            AsyncMessages.FirmwareSuccLoadedToDevice,
            firmwareManager.GetItemByID(firmwareId).Name,
            deviceId);

    /// <summary>
    /// Name of operation.
    /// </summary>
    public override string Name => string.Format(
        AsyncMessages.FirmwareLoadOperation,
        firmwareManager.GetItemByID(firmwareId).Name,
        deviceId);

    /// <summary>
    /// This is called before async operation is executed.
    /// You can prepare data or read something from DB.
    /// </summary>
    protected override void BeforeOperation()
    {
        device = deviceManager.GetItemByID(deviceId);
        firmware = firmwareManager.GetItemByID(firmwareId);
    }

    /// <summary>
    /// Here you should call asynchronious job.
    /// </summary>
    /// <param name="progress">
    /// Progress value reporter that you pass to <see cref="IOperationProvider"/>
    /// or you can report progress manually.</param>
    /// <returns>Task of async operation called in this method.</returns>
    protected override async Task AsyncOperation(IProgress<ProgressValue> progress)
    {
        await operationProvider.LoadFirmwareToDevice(
            device.DeviceConnection,
            firmware,
            progress);
    }

    /// <summary>
    /// This is called after async operation is executed.
    /// You can save the work to DB.
```

```
    /// </summary>
    protected override void AfterOperation()
    {
        device = deviceManager.GetItemByID(deviceId);
        firmware = firmwareManager.GetItemByID(firmwareId);

        var deviceFirmware = deviceFirmwareManager.GetAllItemsQuery()
                                .Where(w => w.DeviceId == deviceId)
                                .FirstOrDefault(o => !o.ValidTo.HasValue);

        if (deviceFirmware != null)
        {
            deviceFirmware.ValidTo = DateTime.Now.ToUniversalTime();
        }

        var newDeviceFirmware = new DeviceFirmwareDO
        {
            DeviceId = deviceId,
            FirmwareId = firmwareId,
            ValidFrom = DateTime.Now.ToUniversalTime(),
        };

        deviceFirmwareManager.Add(newDeviceFirmware);

        device.CurrentFirmwareId = firmware.ID;

        deviceManager.Update(device);
    }
}
```

### B.1.3 AsyncOperationProvider

```
/// <summary>
/// Async operation provider that accept <see cref="AsyncOperationBase"/>.
/// </summary>
public class AsyncOperationProvider : IAsyncOperationProvider
{
    private readonly IDataObjectManager<AsyncOperationDO> asyncOperationManager;
    private readonly IAsyncOperationHubService asyncOperationHubService;
    private readonly ITransactionContextHelper transactionContextHelpler;

    /// <summary>
    /// CTOR.
    /// </summary>
    public AsyncOperationProvider(
        IDataObjectManager<AsyncOperationDO> asyncOperationManager,
        IAsyncOperationHubService asyncOperationHubService,
        ITransactionContextHelper transactionContextHelpler)
    {
        this.asyncOperationManager = asyncOperationManager;
```

```
        this.asyncOperationHubService = asyncOperationHubService;
        this.transactionContextHelpler = transactionContextHelpler;

        RunningAsyncOperations = new ObservableCollection<AsyncOperationBase>();
    }

    /// <summary>
    /// All running async operations at the moment.
    /// </summary>
    public ObservableCollection<AsyncOperationBase> RunningAsyncOperations { get; }

    /// <summary>
    /// Run async operation immediately.
    /// </summary>
    /// <param name="asyncOperation">Async operation to run.</param>
    /// <param name="token">TODO: remove this.</param>
    /// <returns>ID of async operation.</returns>
    public long RunOperationAsync(AsyncOperationBase asyncOperation, string token)
    {
        var asyncOperationDataObject = new AsyncOperationDO
        {
            AsyncOperationStatus = (int)AsyncOperationState.InProgress,
            AsyncOperationType = (int)asyncOperation.AsyncOperationType,
            Name = asyncOperation.Name
        };
        asyncOperationManager.Add(asyncOperationDataObject);

        // If there are child operations, we execute them as well.
        if (asyncOperation.ChildOperations != null)
        {
            var childOperations = asyncOperation.ChildOperations.Select(s =>
                new AsyncOperationDO
            {
                AsyncOperationType = (int)s.AsyncOperationType,
                AsyncOperationStatus = (int)AsyncOperationState.InProgress,
                LastAttempt = DateTime.Now.ToUniversalTime(),
                Name = asyncOperation.Name,
                ParentId = asyncOperationDataObject.ID
            }).ToList();

            asyncOperationManager.AddRangeBulkInternal(childOperations);
        }

        // Progress notification reporter.
        var progress = new Progress<ProgressValue>(progressValue =>
        {
            // We notify clients about progress change in operation.
            asyncOperationHubService.AsyncOperationProgressChanged(
                token,
                asyncOperationDataObject.ID,
                progressValue.Progress,
                AsyncOperationState.InProgress);
        });
```

```
        // This will execute async operation in background.
        asyncOperation.Run(progress).ContinueWith(asyncTask =>
        {
            // After operation is completed (succesfully or not)
            // we call OnAsyncOperationCompleted in separate DB context
            // and save operation result to AsyncOperation table in DB.
            using (TransactionSynchronizationContextHelper.SetNewContext())
                transactionContextHelpler.ExecuteInNewContext<object>(
                    () =>
                    {
                        OnAsyncOperationCompleted(
                            asyncOperationDataObject.ID,
                            asyncOperation,
                            token,
                            asyncTask);
                        return null;
                    },
                    true);
        });

        RunningAsyncOperations.Add(asyncOperation);

        return asyncOperationDataObject.ID;
    }

    /// <summary>
    /// After async operation is completed,
    /// we execute this method in new context to allow provider
    /// save changes in DB tables like AsyncOperations etc.
    /// Then we send SignalR notifications about result (success or error).
    /// </summary>
    /// <param name="asyncOperationId">ID of async operation.</param>
    /// <param name="asyncOperation">Async operation implementation object.</param>
    /// <param name="token">TODO: Remove this.</param>
    /// <param name="task">Task of async operation execution.</param>
    private void OnAsyncOperationCompleted(
        long asyncOperationId,
        AsyncOperationBase asyncOperation,
        string token,
        Task task)
    {
        var asyncOperationDataObject = asyncOperationManager.GetItemByID(asyncOperationId);
        asyncOperationDataObject.LastAttempt = DateTime.Now;

        RunningAsyncOperations.Remove(asyncOperation);

        if (task.Exception != null)
        {
            asyncOperationHubService.AsyncOperationProgressChanged(
                token,
                asyncOperationDataObject.ID,
                0,
                AsyncOperationState.Failed,
                task.Exception.InnerException?.Message,
```

```
                AsyncMessages.AsyncOperationFailed);

            asyncOperationDataObject.ErrorType =
                task.Exception.InnerException?.GetType().Name;

            asyncOperationDataObject.ErrorMessage =
                task.Exception.InnerException?.Message;

            asyncOperationDataObject.ErrorStackTrace =
                task.Exception.InnerException?.StackTrace;

            asyncOperationDataObject.AsyncOperationStatus =
                (int)AsyncOperationState.Failed;
        }
        else
        {
            asyncOperationDataObject.AsyncOperationStatus =
                (int)AsyncOperationState.Successful;

            asyncOperationHubService.AsyncOperationProgressChanged(
                token,
                asyncOperationDataObject.ID,
                100,
                AsyncOperationState.Successful,
                asyncOperation.CompleteMessage,
                AsyncMessages.AsyncOperationCompleted);
        }

        asyncOperationManager.Update(asyncOperationDataObject);
    }
}
```

## B.2 TreeView

**TreeView** je obecná direktiva pro zobrazování stromové struktury s podporou **LazyLoading**[25].

### B.2.1 TreeViewController

**TreeViewController** je interní kontroler **TreeView** directivy. Ve zdrojovém kódu lze sledovat bindované property `source` a `expandedItems`.

[25] **LazyLoading** - Potomci uzlů jsou donačítány ze serveru až když je potřeba

```
/**
 * Internal interface for each item.
 */
interface ITreeItem extends IDto
{
    /**
     * If item is expended in tree or not.
     */
    $expand: boolean;

    /**
     * Returns if item is in process of loading its children.
     */
    $loading: boolean;
}

export class TreeViewController
{
    /**
     * Internal dictionary.
     */
    private _expandedItemsDictionary: { [key: number]: number };

    private _expandedItems: number[];
    private menuTree = new collections.Dictionary<number, ITreeItem[]>();

    /**
     * Binded source model from controller.
     */
    public source: ITreeViewSource<IDto>;

    /**
     * Public property bindable from outside.
     */
    public get expandedItems(): number[]
    {
        return this._expandedItems;
    }
    public set expandedItems(val)
    {
        this._expandedItems = val;
        this._expandedItemsDictionary = [];
        Enumerable.from(val).forEach(f => this._expandedItemsDictionary[f] = f);
        this.refresh();
    }

    /**
     * Selected item in tree binded back to controller.
     * TODO: Implement two way binding support.
     * Note that this includes server support as well.
     */
    public selectedItem: ITreeItem;
```

```
/**
 * Selected item ID in tree binded back to controller.
 */
public selectedItemId: number;

/**
 * Only first server load is covered by this property. (Root level of tree).
 * Then each item has $loading property visualized differently by different
 * busy indicator.
 */
public isAsyncInProgress;

constructor(private $scope: angular.IScope)
{
    this.source.onTreeRefresh.on(() => this.refresh());
}

/**
 * Method called form plus/minus buttons on each item.
 * @param item Item associated with button.
 */
public expandOrCollapseItem(item: ITreeItem)
{
    if (this._expandedItemsDictionary == null)
        this._expandedItemsDictionary = [];

    item.$expand = !item.$expand;
    if (item.$expand)
    {
        this._expandedItemsDictionary[item.id] = item.id;
    } else
        delete this._expandedItemsDictionary[item.id];

    this._expandedItems = Object.keys(this._expandedItemsDictionary).map(s => +s);
}

/**
 * Returns children from binded source to html template.
 * @param parentId Children parent ID.
 */
public getChildren(parent: ITreeItem = null): ITreeItem[]
{
    var parentId = parent == null ? null : parent.id;

    // If this item has not been loaded yet, we execute server call.
    if (!this.menuTree.containsKey(parentId))
    {
        this.menuTree.setValue(parentId, null);

        // If this is root level call, we set root level async property
        // otherwise we set invidual item async property $loading.
        if (parent == null)
            this.isAsyncInProgress = true;
```

```
            else
                parent.$loading = true;

            // We call source getChildren method binded from controller.
            this.source.getChildren(parentId)
                .then((data: ITreeItem[]) =>
                {
                    if (parent == null)
                        this.isAsyncInProgress = false;
                    else
                        parent.$loading = false;

                    data.forEach(f =>
                    {
                        f.$expand = this._expandedItemsDictionary[f.id] != null;
                    });

                    this.menuTree.setValue(parentId, data);
                });

            return null;
        } else // If this item was already loaded, we use cache instead.
        {
            var menuTreeValue = this.menuTree.getValue(parentId);

            return menuTreeValue;
        }
    }

    /**
     * This returns from binded source if specified item has children or not.
     * @param id Item in question.
     */
    public hasChildren(item: IDto)
    {
        return this.source.hasChildren(item);
    }

    /**
     * This redirect selected item out to controller through binded property
     * @see selectedItem.
     * @param item
     */
    public selectTreeItem(item: any)
    {
        this.selectedItem = item;
        this.selectedItemId = item.id;
    }

    /**
     * If binded source model requested refresh, we clear cache
     * and reload tree from scratch.
     */
    private refresh(): void
```

```
    {
        if (this.menuTree != null)
            this.menuTree.clear();
    }
}
```

### B.2.2 ITreeViewSource

Každá třída, která chce sloužit jako zdroj dat pro direktivu **TreeView**, musí implementovat rozhraní `ITreeViewSource<TDto extends IDto>`.

```
/**
 * Interface to model binded from controller to @see TreeView as a source of data.
 */
export interface ITreeViewSource<TDto extends IDto>
{
    /**
     * This is delegate to method on source model, that for specified parent
     * returns children collection.
     * @param parentId Parent ID.
     */
    getChildren: (parentId?: number) => angular.IPromise<TDto[]>;

    /**
     * This is delegate to method on source model, that returns
     * if specified item has children or not.
     * @param item Item in question.
     * @returns Yes/No
     */
    hasChildren: (item: TDto) => boolean;

    /**
     * This could be triggered by source model to force refresh of tree view.
     */
    onTreeRefresh : ILiteEvent<void>;
}
```

### B.2.3 TreeView HTML šablona

```
<rd-busy-indicator class="treeview" is-busy="$ctrl.isAsyncInProgress">
    <ul>
        <li ng-repeat="child in $ctrl.getChildren()">
            <rd-tree-view-item child-template="childTemplate">
            </rd-tree-view-item>
        </li>
```

```
    </ul>
</rd-busy-indicator>
```

### B.2.4 Použití TreeView

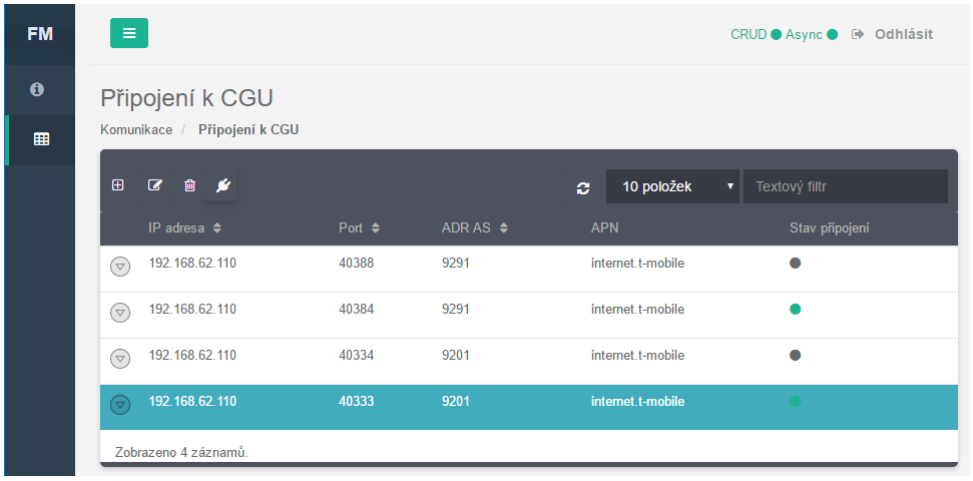
Uvnitř html tagu `<rd-tree-view>` je předána šablona, pro vykreslení každé položky.

```
<rd-tree-view source="configurationValuesMasterCtrl"
              expanded-items="configurationValuesMasterCtrl.expandedMenuItems"
              selected-item-id="configurationValuesMasterCtrl.selectedTreeItemId">
    <a>{{child.name}}</a>
</rd-tree-view>
```

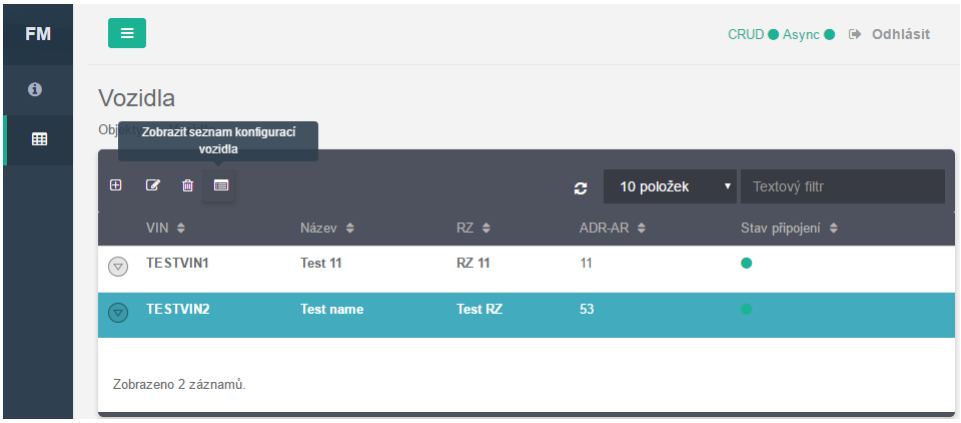
Příloha C

# Ukázky z aplikace

## C.1 Připojení k CGU serverům



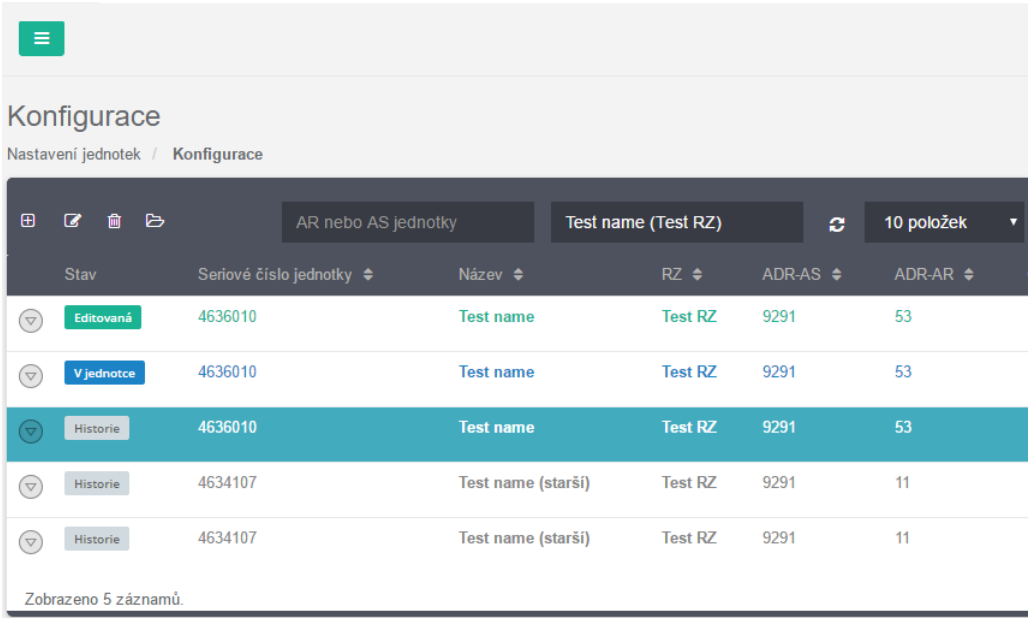

## C.2 Přehled vozidel



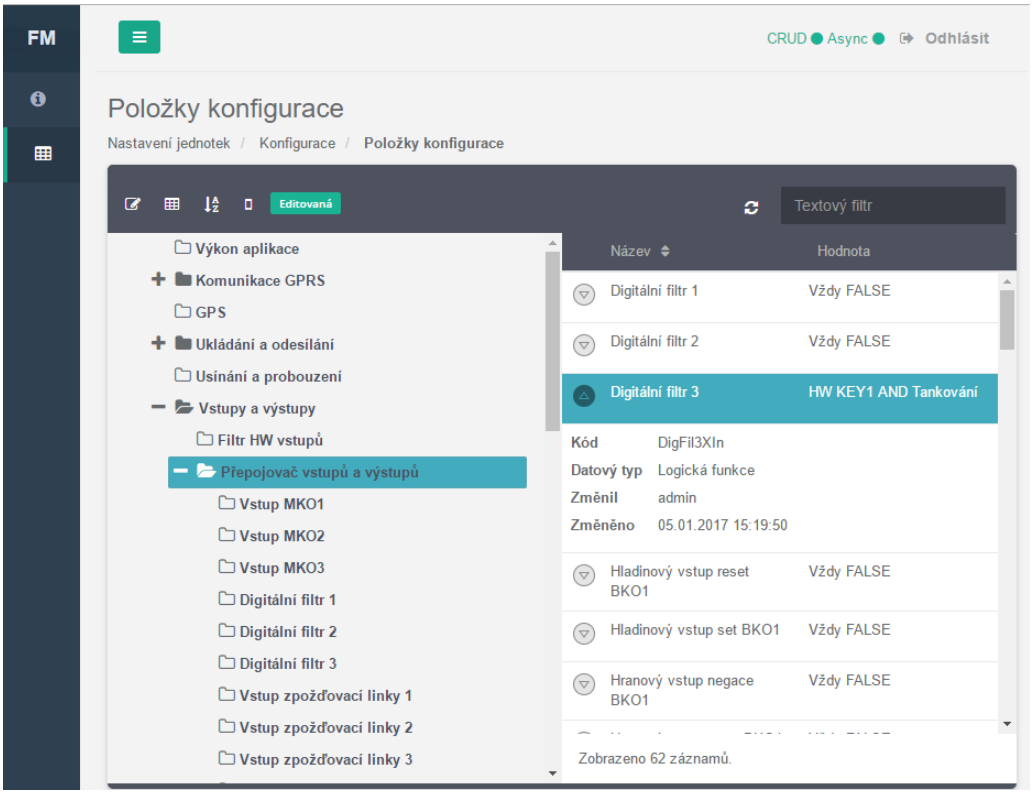
## C.3 Editace konfigurace

### C.3.1 Přehled konfigurací jednoho vozidla

Zde ve sloupci **ADR-AR** je vidět, že dané vozidlo mělo vyměněnu jednotku. Aplikační vrstva s tímto počítá a vrátí všechny konfigurace za celou životnost vozidla.

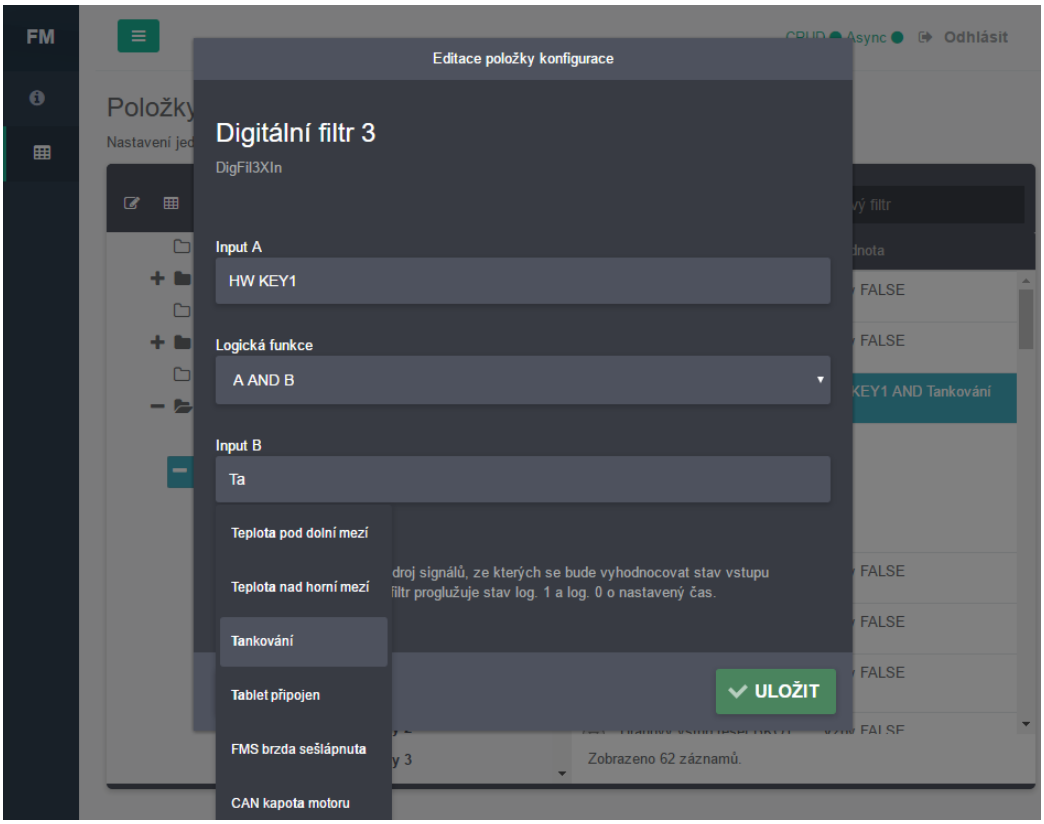


### C.3.2 Zobrazení konfiguračních hodnot ve stromové struktuře



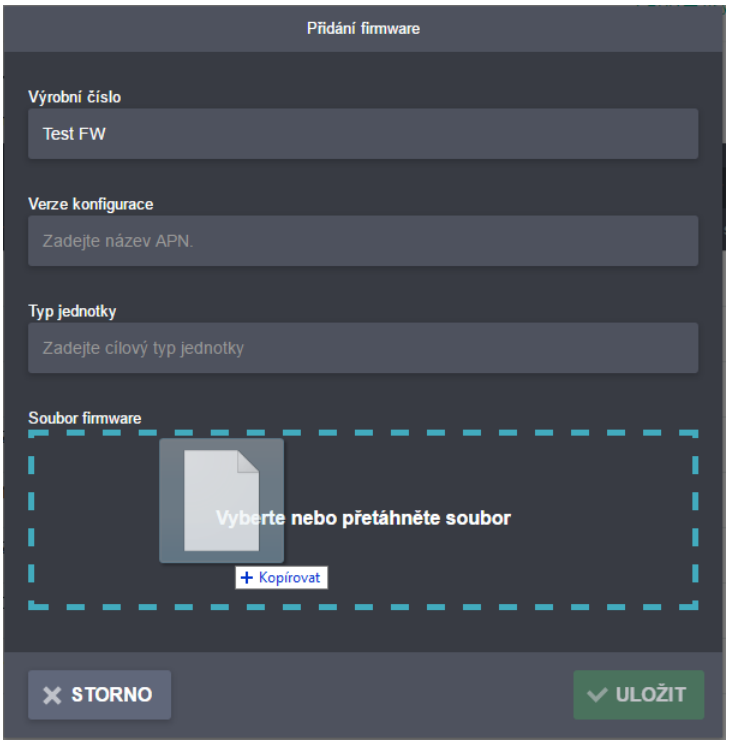
### C.3.3 Editace konfigurační položky

Editace položky typu **logická funkce**.

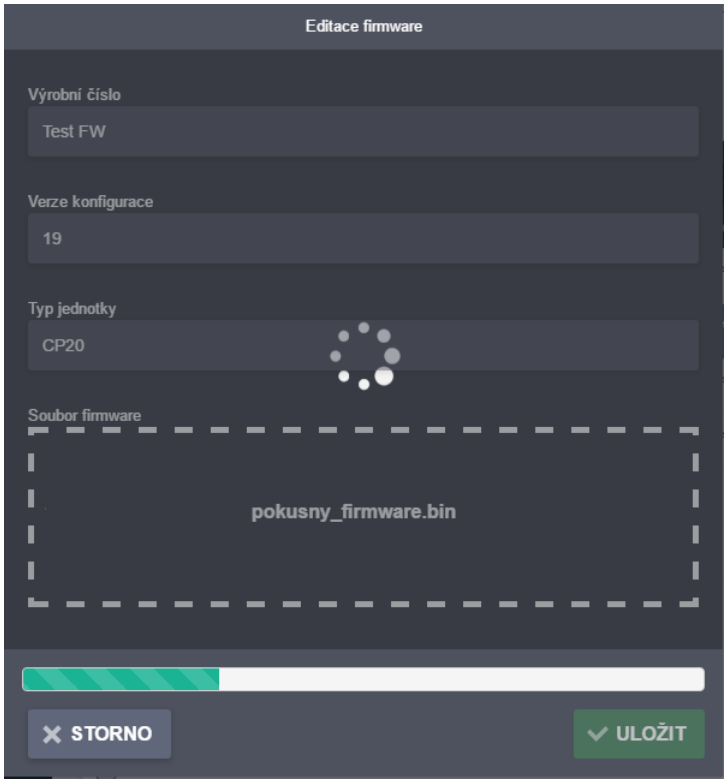


## C.4 Firmware

### C.4.1 Zvolení binárního souboru



### C.4.2 Nahrání binárního souboru



## C.5 Přihlášení

Příloha D

# Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF