FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Aplikace pro konfiguraci dispečerských terminálů
Student:	Bc. Jiří Mantlík
Vedoucí:	Ing. Petra Pavlíčková, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem diplomové práce je navrhnout a vytvořit aplikaci pro konfigurační server, který zajistí nastavení připojených drážních dispečerských terminálů. Aplikace bude udržovat přehled o verzích všech připojených terminálů a zajišťovat distribuci nových verzí.

1. Prostudujte dostupnou literaturu k problematice konfiguračních aplikací.
2. Zanalyzujte funkční a nefunkční požadavky na aplikaci.
3. Navrhněte řešení serverové aplikace a administračního rozhraní.
4. Implementujte vybrané funkcionality a otestujte je.
5. Zhodnoťte dané řešení a doporučte další rozvoj.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 12. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Aplikace pro konfiguraci dispečerských terminálů

Bc. Jiři Mantlík

Katedra softwarového inženýrství
Vedoucí práce: Ing. Petra Pavlíčková, Ph.D.

9. ledna 2019

Poděkování

Rád bych poděkoval především vedoucí práce paní Ing. Petře Pavlíčkové, Ph.D. za cenné rady, podněty a její čas při psaní této práce. Velké díky patří také všem, kteří mi poskytli konzultaci či pomohli s korekturou textu. V neposlední řadě chci poděkovat své rodině, přátelům a kolegům za podporu a trpělivost nejen během tvorby diplomové práce, ale i během celého studia.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 9. ledna 2019 .....

Odkaz na tuto práci

Mantlík, Jiří. *Aplikace pro konfiguraci dispečerských terminálů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a implementací serverové aplikace pro konfiguraci drážních dispečerských terminálů. Aplikace by měla sloužit administrátorům Správy železniční dopravní cesty ke správě konfigurací, firmware a telefonních seznamů používaných terminály v jednotlivých stanicích. Přínosem práce je zjednodušení údržby terminálů, která je nyní prováděna ruční instalací a konfigurací každého stroje.

Klíčová slova serverová aplikace, konfigurační server, dispečerské terminály, konfigurace, firmware, telefonní seznamy, klient-server, RESTful API, Java, Spring, Vaadin

Abstract

This master thesis deals with analysis, design and implementation of server application for configuration of railroad dispatcher terminals. The application should serve administrators of Správa železniční dopravní cesty for management of configurations, firmware versions and phone books used by terminals in individual stations. The main beneficial outcome of this thesis is simplification of terminals maintenance, which is nowadays made by manual installation and configuration of each machine.

Keywords server application, configuration server, dispatcher terminals, configuration, firmware, phone books, client-server, RESTful API, Java, Spring, Vaadin

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Cestování vlakem je v naší zemi velice oblíbeným typem dopravy. Aby však mohlo bez problémů probíhat, musí být železnice v dobrém stavu, což samozřejmě vyžaduje velké množství logistiky. Zaměstnanci ve stanicích musí mít neustále možnost být v kontaktu s techniky, kteří kontrolují a opravují tratě, strojvedoucími železničních společností a dalšími lidmi. To je samozřejmě náročné zvládnout bez patřičných systémů, které by zajišťovaly a usnadňovaly hlasovou komunikaci.

V České republice se o udržování železničních tratí stará SŽDC (Správa Železniční Dopravní Cesty), která je jejich vlastníkem a provozovatelem. Tato společnost používá ve stanicích dispečerské terminály, které řeší výše zmíněný problém komunikace a usnadňují pracovníkům této společnosti jejich práci. Aplikace na těchto terminálech jsou komplikované a vyžadují množství konfigurace, která se navíc může lišit podle pracovníka, jenž zrovna terminál obsluhuje. Kromě toho je také nutné tyto aplikace čas od času aktualizovat na novější verze. A jaká by to byla hlasová komunikace bez možnosti ukládání telefonních kontaktů?

Všechny tyto problémy nyní musí řešit technici na každém terminálu zvlášť, a to i přesto, že se proces a data na jednotlivých strojích příliš nemění. Jistě by bylo pro techniky pohodlnější, kdyby mohli definovat konfiguraci či telefonní seznamy na jednom místě. Je zde tedy patrná potřeba jedné konfigurační serverové aplikace, která by tento problém řešila.

Na výše zmíněném předpokladu staví tato diplomová práce, jejíž téma bylo zvoleno především proto, že se jedná o zajímavý problém z reálného prostředí, jehož vyřešení skutečně usnadní mnoha lidem jejich práci. Druhou motivací pak byla příležitost naučit se nové postupy a technologie, které jsou používány ve firmě, jež dodává terminály Správě železniční dopravní cesty, a vůbec možnost tvorby práce v rámci praxe ve firmě.

Kapitola 1

Cíl práce a její struktura

Hlavním cílem této práce je návrh aplikace pro konfigurační server společně s jejím administračním rozhraním a následná realizace vybraných funkcionalit. Tato aplikace bude sloužit ke konfiguraci dispečerských terminálů v síti SŽDC. Pro dosažení tohoto cíle je nutné splnit několik důležitých bodů, jimž se věnují jednotlivé kapitoly.

Kapitola 2 je věnována vysvětlení teoretických pojmů a postupů, které jsou pro vývoj systému důležité.

V kapitole 3 je pak čtenář seznámen s firemními projekty, s nimiž by měla vytvořená aplikace spolupracovat a doplňovat je.

Sběrem a analýzou požadavků na tento systém se zabývá kapitola 4. Zmíněná analýza je důležitá pro pochopení procesů, jež by aplikace měla podporovat.

Obsahem následující kapitoly 5 je návrh zásadních součástí aplikace, aby nedošlo ke komplikacím během implementace.

Předposlední kapitola 6 se věnuje samotné implementaci a otestování vybraných funkcí, které budou moci být později doplněny o další navržené prvky.

Závěrečná část pak obsahuje zhodnocení výsledků a doporučení dalších kroků, které by měly být podniknuty pro dokončení celé aplikace.

Kapitola 2

Teorie

2.1 Unifikovaný proces vývoje softwaru

Proces vývoje softwaru je metodikou, která definuje způsob, jak jsou uživatelské požadavky převedeny na vytvořený software. USDP (Unified Software Development Process) je průmyslovým standardem této metodiky, jelikož se jedná o pragmatickou a ověřenou metodu vývoje. Unifikovaný proces vývoje je založen na iterativním přístupu, kdy každá iterace obsahuje pět základních pracovních fází:

- Požadavky
- Analýza
- Návrh
- Implementace
- Testování

Pro jednoduchý popis jednotlivých částí především ve fázi analýzy a návrhu se obvykle používá modelovací jazyk UML (Unified Modeling Language), který byl navržen pro spojení nejlepších modelovacích principů a softwarového inženýrství. Správně vytvořené diagramy psané v tomto jazyce jsou srozumitelné jak pro člověka, tak i pro automatizované nástroje [1].

2.2 Požadavky

Prvním krokem vývoje by vždy mělo být stanovení požadavků na vytvářený systém. Tato oblast by neměla být podceňována, protože chyby, kterých se zde vývojáři dopustí, ovlivní celý zbytek vývoje. Špatně definované požadavky mohou vést k neuspokojení zákazníků a finančním ztrátám. Proto by do sběru,

konkretizace a analýzy požadavků měli být co nejvíce zapojeni budoucí uživatelé systému.

Požadavkem se rozumí popis určité funkce či vlastnosti vyvíjeného systému. Při sběru požadavků se rozlišují dva jejich typy:

- **Funkční požadavky** určují, jaké chování bude systém nabízet.
- **Nefunkční požadavky** se zaměřují na vlastnosti či omezující podmínky fungování celého systému. Do této kategorie spadají požadavky na výkon, technologie a podobně. Tyto požadavky se pak promítnou do technické architektury systému.

Požadavky sjednocují a zpřesňují představy vývojářů a zákazníků. Každý požadavek určuje, co by mělo být dodáno. Při jejich specifikaci by měl být kladen důraz na to, co bude systém nabízet, a ne na to, jak se toho dosáhne [2].

2.3 Analýza

Analýza přímo souvisí se sběrem požadavků. Jejím cílem je jejich ujasnění a případné odhalení požadavků chybějících. Toho je dosaženo tvorbou modelů, které se opět zaměřují pouze na to, co by systém měl dělat, přičemž se abstrahují od konkrétních postupů, jak toho dosáhnout [1]. Analýza může sestávat z mnoha různých druhů modelů. V následujících sekcích budou popsány ty nejzákladnější z nich.

2.3.1 Analýza procesů v aplikaci

Modelování procesů aplikace je podstatné pro správné pochopení činnosti zákazníka, která by měla být systémem realizována. Tím je umožněno identifikovat problémová místa a navrhnout vylepšení těchto procesů, což vede k přesnější specifikaci požadavků. Procesem se rozumí uspořádaná množina aktivit, které transformují zadané vstupy na výstup, a je třeba sledovat jeho důležitost, četnost provádění a časovou či finanční náročnost [3].

V této části analýzy se často využívají diagramy aktivit, jelikož umožňují proces modelovat jako kolekci aktivit a přechodů mezi nimi. Jedná se tedy určitým způsobem o variantu stavového diagramu, a proto používají stejnou terminologii. Výhodou těchto diagramů je jejich schopnost modelovat i paralelní procesy, což umožňuje jejich zefektivnění často jen odstraněním zbytečných sekvenčních částí [2].

2.3.2 Model případů užití

Případy užití (Use Case) jsou nástrojem pro zachycení funkčních požadavků. Každý případ užití by měl popisovat jeden způsob, jakým bude systém použí-

ván. Z případů užití se vychází při návrhu a implementaci funkcí systému, proto by měly být co nejvíce kompletní [2].

Model případů užití se skládá z těchto komponent:

- **Aktéři** – role, které zastávají uživatelé systému. Tyto role mohou být přiděleny lidem, předmětům i jiným aplikacím.
- **Případy užití** – sada činností, které mohou uživatelé vykonávat v systému.
- **Hranice systému** – ohraničení oddělující aktéry od případů užití realizovaných systémem. Jsou to tedy hranice funkčnosti systému.
- **Relace** – vztahy mezi aktéry a případy užití

Use Case model dále poskytuje jeden ze zdrojů objektů a tříd pro jejich další modelování [1].

2.3.3 Doménový model

Posledním a zásadním krokem analýzy je přesné určení problémové domény. K tomu slouží doménový model, jehož úkolem je zachycení entit, popis vztahů mezi nimi a poskytnutí základu pro databázový model či model tříd. Doménový model používá jazyk problémové domény, třídy a vztahy bere z reálného světa a neměl by se nikdy zaobírat implementačními detaily [4].

Doménový model se skládá z těchto komponent:

- **Doménové třídy** – Entity, které modelují problémovou oblast. Obvykle jsou identifikovány při analýze procesů a případů užití. Tyto třídy mohou obsahovat základní atributy či operace, ale především by měly mít jasný a výstižný název.
- **Atributy a operace** – Blíže specifikují doménové třídy, avšak stále v jazyce domény. Neobsahují tedy žádné informace o datových typech, tabulkových klíčích, či implementačně závislých operacích.
- **Vztahy** – Popisují, jak spolu jednotlivé třídy souvisí. Zde je podstatné sledovat především násobnosti účastníků relace.

2.4 Návrh

S výstupy získanými při analýze se následně pracuje ve fázi návrhu. Cílem této části vývoje je přesně specifikovat způsob, jak budou zanalyzované funkce implementovány. Během návrhu se již berou v potaz i nefunkční požadavky a je potřeba zde rozhodnout o strategických otázkách, protože z této fáze pak přímo vychází implementační část vývoje [1]. Mezi nejdůležitější kroky, které

je zde třeba podniknout, patří výběr vhodných technologií, návrh architektury systému a vytvoření databázového modelu.

2.4.1 Architektura

Návrh architektury se zabývá rozhodováním o organizaci systému a přímo navazuje na analýzu nefunkčních požadavků. Přitom je podstatné určit takové faktory, které by ji mohly ovlivnit, pochopit je a navrhnout řešení. Zároveň je dobré se zamyslet nad různými variacemi problémů, které již navrhovaný systém obsahuje nebo by se mohly v budoucnu objevit. Při správném návrhu architektury je pak možné se vyhnout mnohým nepříjemnostem při následné implementaci, jako jsou třeba:

- Vynaložení zbytečného úsilí na nedůležité problémy.
- Riziko vynechání klíčových prvků v návrhu systému.
- Nesoulad produktu s obchodními cíli.

Při návrhu architektury systému se obvykle používají diagramy komponent. Jednotlivé části systému jsou modelovány jako uzavřené komponenty, které navenek vystavují pouze svá rozhraní. Zde je podstatné si uvědomit, že každá z komponent by měla být modulární, soběstačná a nahraditelná, přičemž rozhraní jsou opravdu důležitá [5]. Architektura aplikace je potom modelována jako skupina komponent, které jsou propojeny přes svá rozhraní.

Pro usnadnění návrhu architektury bylo vytvořeno značné množství architektonických návrhových vzorů, které přinášejí doporučení, jak sestavit systém v závislosti na jeho účelu. Použití návrhových vzorů obvykle zajišťuje uvolnění vztahů mezi jednotlivými částmi aplikace, čímž zpřehledňuje a usnadňuje jejich vývoj a údržbu. Jednotlivé vzory se také mohou vzájemně vnořovat, pro docílení ještě lepšího rozdělení systému [6]. Několik vzorů bude popsáno v následujících sekcích.

2.4.1.1 Klient-server

Tento architektonický vzor nabyl zásadního významu při návrhu systémů, jež běží na počítačové síti. V takovémto systému se zodpovědnost dělí mezi server a jednoho či více klientů. Ti spolu komunikují právě přes počítačovou síť.

Server pasivně vyčkává na požadavky klientů, které následně zpracovává a odesílá klientům odpovědi. Obvykle se tímto způsobem stará o aplikační logiku a ukládání dat. Zároveň je vždy na této straně potřeba zajistit kontrolu posílaných dat, aby nedošlo k narušení integrity systému. Server by ideálně měl poskytovat standardizované rozhraní – API (Application Programming Interface), které by nemělo zatěžovat klienty specifikami serverového systému.

Klient zajišťuje přímou komunikaci s uživatelem. Posílá požadavky na server a čeká na jejich zpracování, aby mohl následně získaná data zobrazit uživateli [7]. V praxi se používají klienti dvou typů:

- **Tenký klient** se stará pouze o zobrazování dat uživateli a o převedení jeho požadavků na server.
- **Tlustý klient** zajišťuje další funkce nad rámec tenkého klienta, a to například z důvodu, aby nebylo nutné se stále dotazovat serveru na zdánlivé maličkosti.

2.4.1.2 Třívrstvá architektura

Vzor třívrstvé architektury je specifickým příkladem vrstvené architektury. Tento přístup rozděluje systém na vrstvy, které se na sebe postupně nabalují. V ideálním případě by každá vrstva měla být závislá nejvýše na jedné jiné vrstvě, a to té, kterou obaluje. Jednotlivé vrstvy pak plní svůj vlastní účel a při zachování neměnných rozhraní mezi nimi je možné je nezávisle na sobě vyvíjet i měnit [8].

Jak již plyne z jejího názvu, třívrstvá architektura rozděluje systém na základní tři vrstvy:

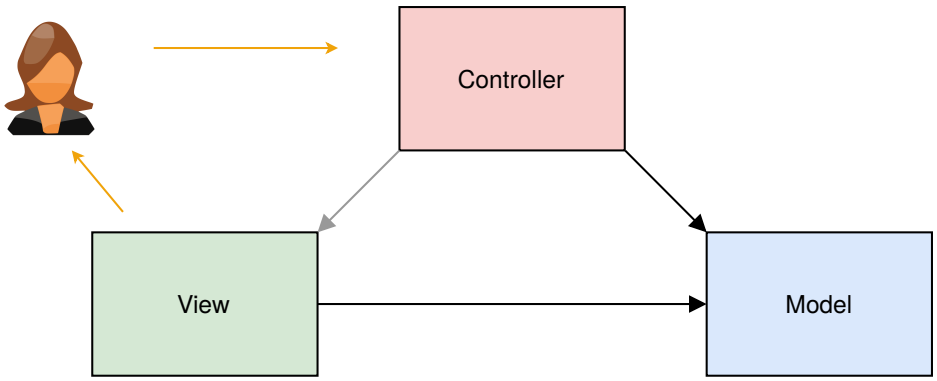
- **Prezentační vrstva** zajišťuje vstup požadavků, které předává vrstvě aplikační, a výstup výsledků. Může proto poskytovat GUI (Graphical User Interface), což ji činí závislou na dané platformě.
- **Aplikační vrstva** deleguje požadavky od vrstvy prezentační na vrstvu datovou, přičemž na nich provádí vlastní výpočty a operace.
- **Datová vrstva** má na starosti ukládání a načítání dat i další perzistenční operace.

2.4.1.3 MVC

Architektonický návrhový vzor MVC (Model-View-Controller) dělí aplikaci do třech částí:

- **Model** představuje data a business logiku aplikace.
- **View** se stará o zobrazení uživatelského rozhraní a dat z modelu.
- **Controller** zajišťuje úpravu modelu v reakci na uživatelské vstupy. Dále má na starosti provázání celé aplikace. Způsob, jakým toho dosáhne, pak odlišuje jednotlivé specifikace tohoto vzoru.

Schéma propojení těchto částí zobrazuje diagram na obrázku 2.1. Požadavek uživatele přichází do controlleru, který na základě toho upraví model. View obsahuje odkaz na model, takže je o změnách od controlleru informováno a může uživateli zobrazit upravená data. Také je poměrně časté přímé spojení mezi controllerem a view, ale to už záleží na konkrétní variantě MVC. Podstatné však je, aby model nikdy neobsahoval přímý odkaz na ostatní dvě komponenty [9].



Obrázek 2.1: Schéma propojení komponent v MVC architektuře [9].

Vzor MVC se vyvinul v sedmdesátých letech dvacátého století, tedy ještě v době sálových počítačů, čemuž také odpovídá jeho struktura. Controller sloužil pro příjem příkazů z klávesnice, zatímco view vykreslovalo data na monitor. Proto zde byla zásadní pouze vazba controlleru a view na model. Se sjednocením vstupu a uživatelského rozhraní však ztratil controller v původním smyslu význam, což vedlo ke vzniku různých nových variant této architektury. MVC se však stále používá, a to především v prostředí webových aplikací, kde se stará například o překlad HTTP (HyperText Transfer Protocol) požadavku na konkrétní view [10].

2.4.1.4 MVP

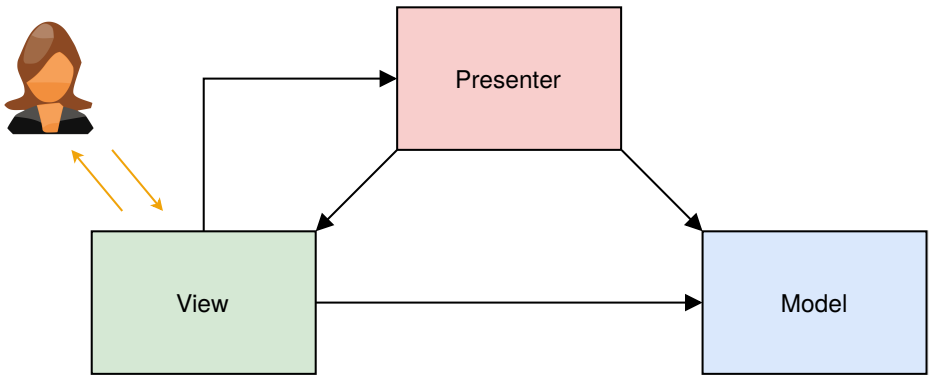
Z architektury MVC vychází mimo jiných variant také architektonický návrhový vzor MVP (Model-View-Presenter). Dnešní uživatelská rozhraní obvykle sama obsahují interaktivní prvky, a proto je tento vzor vhodnější především z toho důvodu, že přesouvá příjem požadavků uživatele přímo do view.

Jak je vidět na obrázku 2.2, skladbou komponent se MVP od svého předchůdce příliš neliší. Jednotlivé části však plní trochu jiné funkce. Akce uživatele jsou odchytávány ve view, které je následně posílá ke zpracování presenteru. Další postup pak závisí na tom, o jaký typ MVP se jedná:

- **Supervising Controller** staví presenter pouze do pozice dohlížejícího, který má na starosti pouze složitější prezentační logiku. View si data z modelu načte samo a od presenteru si nechá doplnit jen to nezbytně nutné. Tento typ zobrazuje právě obrázek 2.2.

- **Passive View** se snaží o co nejjednodušší view, kterému není do modelu umožněn přístup. Oproti obrázku 2.2 zde tedy chybí vazba mezi view a modelem, což přenáší mnohem více odpovědnosti na presenter, který se nyní musí postarat i o přenos dat z modelu do view.

Supervising controller MVP je vhodný v případě, kdy je žádoucí jednodušší kód, který není potřeba příliš testovat. Druhá varianta oproti tomu umožňuje snadnější automatické testování prezentační logiky, ovšem za cenu komplexnějšího kódu [10].



Obrázek 2.2: Schéma propojení komponent v MVP architektuře [10].

2.4.1.5 Architektura REST

REST (REpresentational State Transfer) představil v roce 2000 ve své dizertační práci [11] jeden ze spoluautorů protokolu HTTP, a je tedy správné předpokládat, že mají tyto dvě technologie mnoho společného. Tato architektura pro webové API totiž využívá zmíněného protokolu pro vzdálený přístup ke zdrojům, jako jsou data či stavy aplikace. REST je tedy oproti jiným technologiím, jako je například SOAP (Simple Object Access Protocol), orientován datově a nikoli procedurálně.

Architektura rozhraní REST definuje čtyři základní metody pro přístup ke zdrojům, jež jsou identifikovány svým URI (Uniform Resource Identifier). Tyto metody jsou realizovány pomocí odpovídajících HTTP metod a slouží pro vytvoření dat, jejich čtení, úpravu a smazání. Proto se označují pojmem

CRUD (Create, Read, Update, Delete) a odpovídají HTTP metodám POST, GET, PUT a DELETE [12].

API, která využívají architektury REST, jsou označována pojmem RESTful. K rozlišení, jak moc může být dané API tímto pojmem označeno, je využíván Richardsonův Maturity model, který poskytuje jednoduchý způsob, jak principy REST pochopit. Richardson v tomto modelu definuje 4 úrovně splnění principů REST:

- **Úroveň 0:** API využivá protokolu HTTP, ale nedefinuje zdroje a používá pouze jednu HTTP metodu.
- **Úroveň 1:** Přítomnost zdrojů, které jsou definovány jednoznačnou URI. Pro přístup k nim je však stále používána pouze jedna HTTP metoda.
- **Úroveň 2:** Specifikace metod použitelných pro jednotlivé zdroje v souladu s HTTP konvencemi. Jde tedy především o použití příslušných HTTP metod pro dané akce (GET pro čtení, PUT pro úpravu a podobně), práci s HTTP hlavičkami a využívání číselných HTTP status kódů.
- **Úroveň 3:** Použití HATEOAS (Hypertext As The Engine Of Application State), tedy specifikování možností interakce přímo v reprezentaci zdroje. V případě správného použití je klient méně závislý na podobě služby, neboť nemusí znát přesnou podobu URI a metody dopředu v době implementace, ale může si ji vyžádat od serveru za běhu.

Správně navržené RESTful API by mělo dosahovat třetí úrovně, ale v praxi se většina služeb spokojí pouze s dosažením úrovně druhé, a to především z důvodu obtíží s pochopením HATEOAS. To nemusí být nutně špatné, neboť pro určité případy může být použití HATEOAS zbytečně komplikované v porovnání s jeho přínosem [13].

2.4.2 Databázový model

Tento model vychází z původního návrhu relačního modelu databáze, který vytvořil doktor E. F. Chod v roce 1970. Jeho záměrem bylo popsat datové struktury jako matematické relace, které jsou dány svým jménem a množinou atributů. Relace pak symbolizují tabulky databáze, jejichž sloupce jsou dány atributy těchto relací [14].

Databázový model tedy zobrazuje strukturu dat tak, jak bude implementována konkrétní relační databází. Vychází z doménového modelu, který je vytvořen v rámci analýzy, a doplňuje jej o prvky podstatné při návrhu relační databáze [15]. Mezi tyto prvky patří především:

- **Atributy relací** včetně jejich datových typů, a to i takové, které již nevychází z problémové domény, ale jsou důležité pro implementaci.

- **Dekompozice m:n vazeb** mezi doménovými třídami na spojení přes vazební tabulky.
- **Primární a cizí klíče** pro vazby mezi jednotlivými relacemi.
- **Integritní omezení**, která zajišťují konzistenci dat vůči definovaným pravidlům.

Kapitola 3

Popis souvisejících projektů

Protože je aplikace v této práci vyvíjena v rámci firmy, věnuje se tato kapitola popisu firemních projektů, na které by měla výsledná aplikace navazovat a podporovat je.

3.1 Terminálové aplikace

Základním kamenem firmy jsou aplikace pro dispečerské terminály. Tyto aplikace umožňují hlasovou komunikaci mezi různými koncovými uživateli z řady technologicky rozdílných pevných i mobilních sítí. Tyto aplikace samozřejmě vyžadují obsáhlou konfiguraci, aby byly schopné poskytovat své služby. Tato konfigurace může být prováděna přímo na terminálech nebo hromadně pomocí aplikačního serveru.

Konfigurace v síti SŽDC jsou v tuto chvíli prováděny přímo na terminálech, jelikož pro aplikace použité v této síti není aplikační server vytvořen. Tento přístup je samozřejmě náročný na práci techniků, protože není možné terminály konfigurovat hromadně. Aplikace na drážních terminálech mají konfigurace uloženy v souborech formátu CCS (Configuration Control & Statistics), což je formát definovaný a používaný firmou právě pro tento účel. Pro úpravu těchto souborů slouží speciální webové rozhraní.

3.2 Aplikační server

Pro umožnění hromadné konfigurace terminálů již ve firmě existuje stávající projekt, který podporuje základní funkce, jako je třeba přihlášení uživatelů do administračního rozhraní či nastavení telefonních účtů, a dále různé funkce specifické pro různé zákazníky.

Tento projekt však nepodporuje konfiguraci a funkce, které vyžadují terminály v síti SŽDC, a některé další funkce nevyhovují jejich potřebám. Proto není možné tento server k jejich konfiguraci využít.

Přesto by bylo dobré využít již vytvořené funkce aplikačního serveru pro aplikaci vyvíjenou v rámci této práce a zároveň je žádoucí zahrnout nově vyvíjené unkce do stávajícího projektu. Proto bylo ve firmě v průběhu realizace této práce rozhodnuto, že bude tato aplikace vyvíjena do stávajícího projektu jako nový modul a bude tedy používat některé již hotové funkce.

Kapitola 4

Analýza

Tuto kapitolu tvoří analýza vyvíjené aplikace, jež je zahájena ujasněním funkčních a nefunkčních požadavků pomocí analýzy procesů v aplikaci. Dalšími kroky, které byly v této části učiněny, jsou pak vytvoření modelu případů užití a analýza problémové domény.

4.1 Analýza procesů v aplikaci

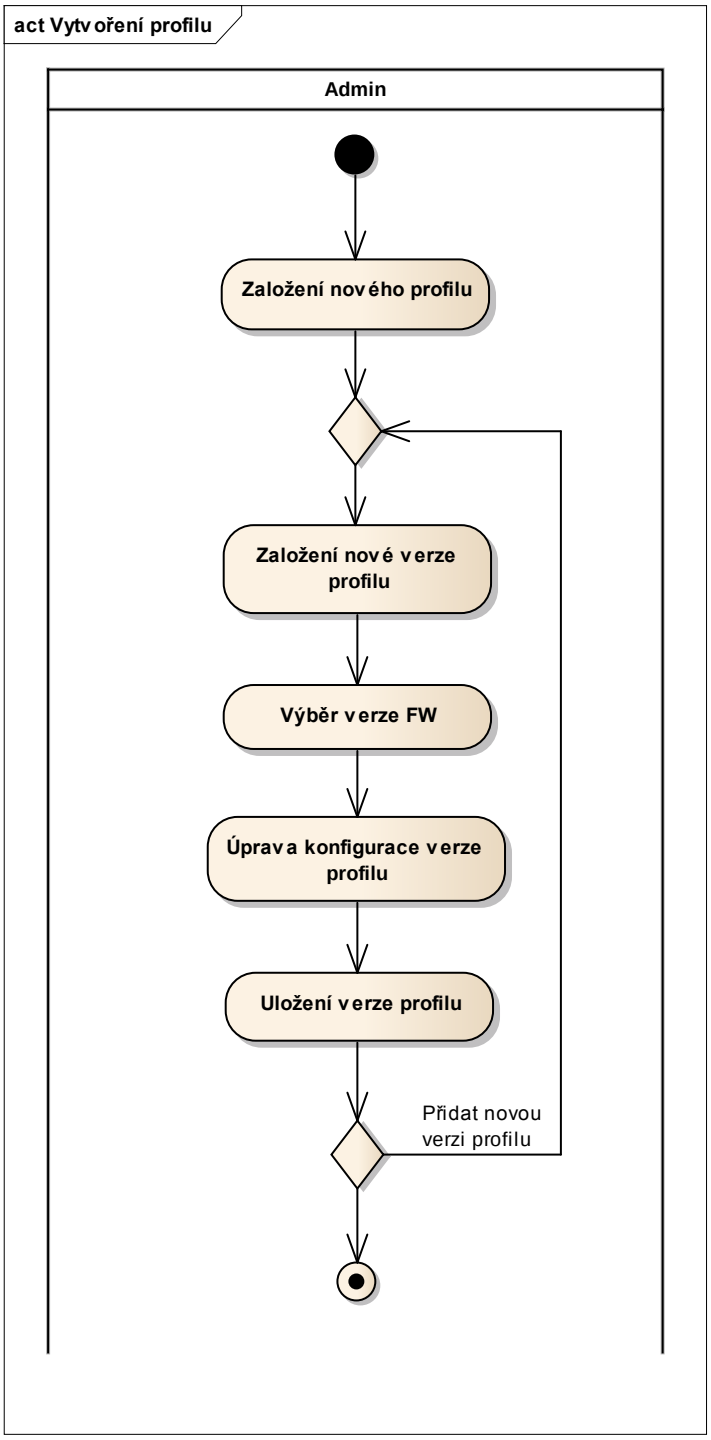
V této sekci jsou popsány nejdůležitější procesy, které by měla konfigurační aplikace podporovat. Analýza těchto procesů probíhala současně se sběrem požadavků, aby bylo možné funkční požadavky správně pochopit.

4.1.1 Vytvoření konfiguračního profilu

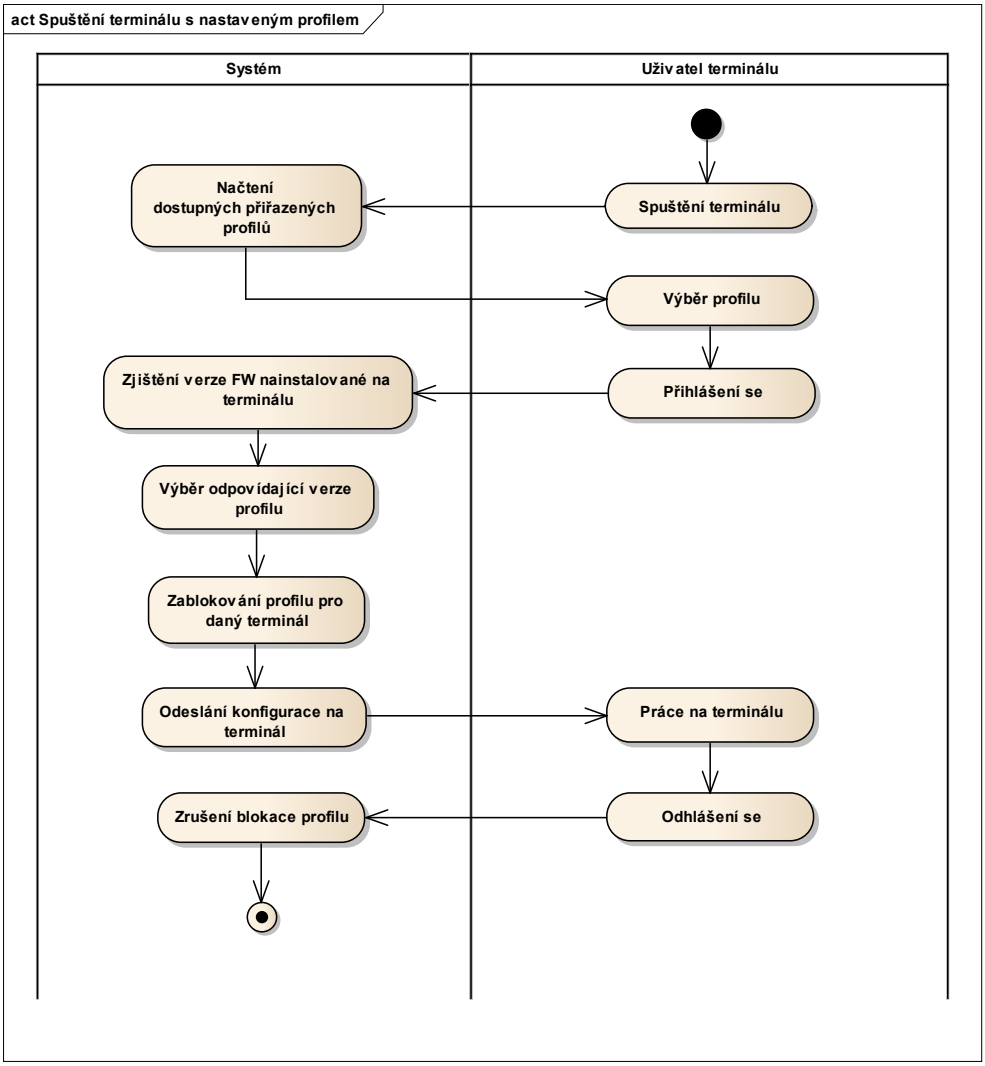
Diagram na obrázku 4.1 znázorňuje proces vytvoření nového konfiguračního profilu. Administrátor v systému založí nový profil. Pokračuje tím, že vytvoří jeho první verzi a přiřadí jí verzi FW (FirmWare), na níž by měla fungovat. Následně provede úpravu konfiguračního souboru a po uložení změn se rozhodne, zda vytvoří další verzi tohoto profilu. Proces končí v okamžiku, kdy administrátor již neplánuje vytvářet další verze profilu.

4.1.2 Spuštění terminálu s nastaveným profilem

Obrázek 4.2 obsahuje diagram popisující proces spuštění terminálu s daným konfiguračním profilem. Po spuštění terminálu uživatelem systém načte profily, které byly k tomuto terminálu přiřazeny a nejsou již spuštěny jinde. Uživatel si z těchto profilů jeden vybere a přihlásí se. Systém zareaguje výběrem verze profilu, která odpovídá verzi FW instalované na terminálu. Následně tuto konfiguraci odešle na terminál a označí profil za užívaný tímto terminálem. Uživatel pak na terminálu pracuje a po jeho odhlášení proběhne v systému uvolnění profilu pro ostatní uživatele.



Obrázek 4.1: Proces vytvoření nového konfiguračního profilu.



Obrázek 4.2: Proces spuštění terminálu s nastaveným profilem.

4.1.3 Nahrání nové verze FW

Diagram na dalším obrázku 4.3 představuje popis nahrání nové verze FW jak do systému, tak i na terminál. Administrátor v systému vytvoří záznam pro novou verzi FW a nahraje patřičná data z disku. Poté vybere terminál, na který má být nová verze FW nahrána, a označí jej k aktualizaci. Může pokračovat výběrem dalších terminálů.

Systém v reakci na označení od administrátora odešle firmware archiv na příslušný terminál a pak kontroluje stav aktualizace. V případě, že stále probíhá, upraví si informaci o jejím průběhu a pokračuje v kontrole. Jakmile vše v pořádku proběhne, systém si upraví uloženou aktuální verzi FW u daného terminálu.

4.1.4 Vytvoření telefonního seznamu

Posledním diagramem je vytvoření telefonního seznamu na obrázku 4.4. Administrátor založí prázdný telefonní seznam a pak se rozhodne, zda jej rovnou naplní daty ze souboru. V takovém případě pokračuje výběrem souboru z disku a nastavením patřičných parametrů pro import z CSV (Comma-Separated Values). Jakmile tyto parametry uloží, provede systém načtení hodnot ze souboru a jejich uložení do telefonního seznamu. Administrátor si pak nehledě na předchozí rozhodnutí zobrazí obsah telefonního seznamu, který následně dle libosti upravuje přidáváním, odebíráním či úpravou kontaktů, dokud není s výsledkem spokojen.

4.2 Funkční požadavky

V této sekci jsou přiblíženy požadavky na funkce aplikace. Požadavky byly formulovány na základě dat získaných od SŽDC a upřesněny za pomoci analýzy procesů provedené v sekci předchozí.

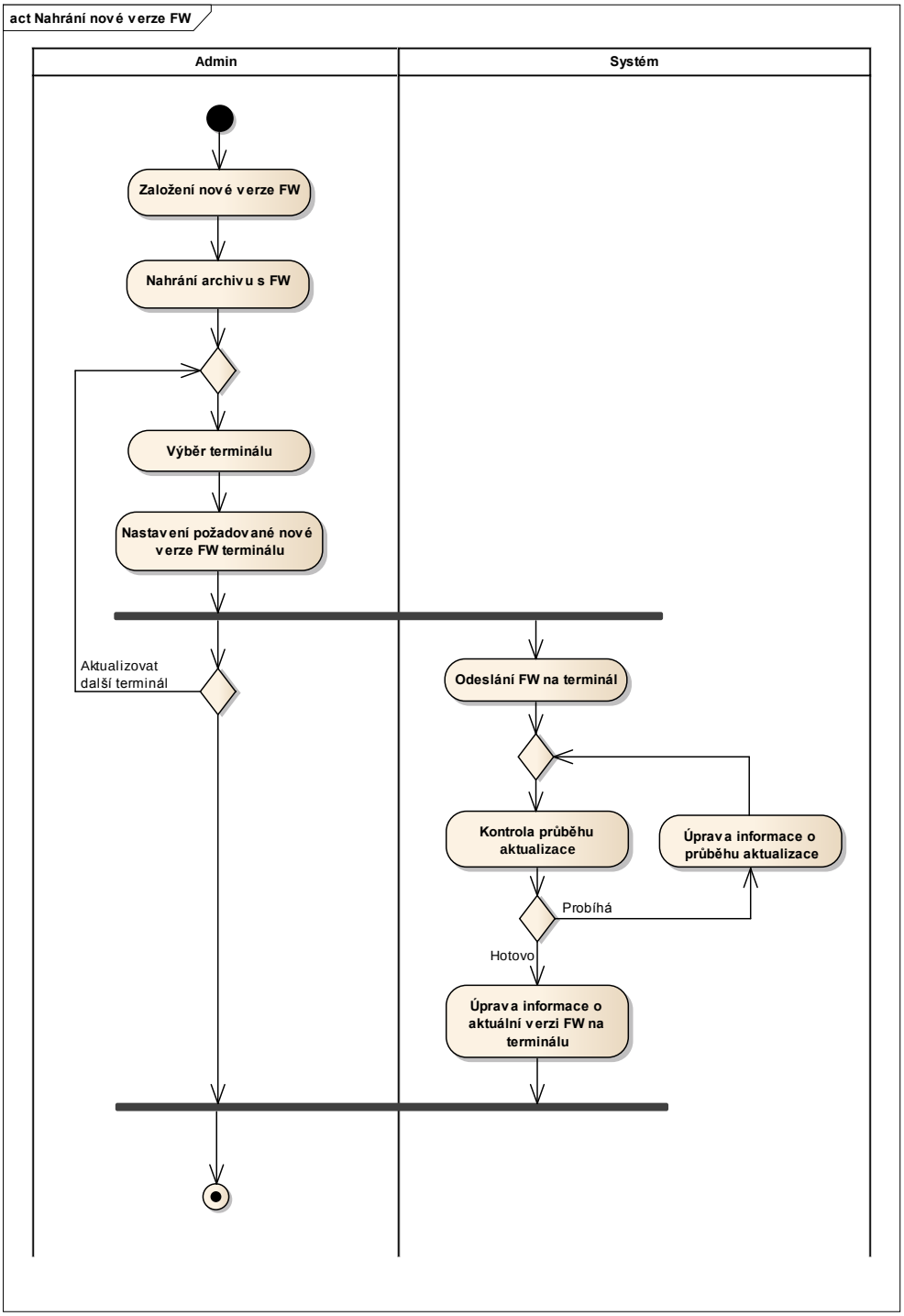
4.2.1 Organizační struktura

V aplikaci bude možné upravovat organizační strukturu obvodu, který je k danému serveru přiřazen. Pro účely SŽDC je tato struktura tvořena několika SSZT (Správa Sdělovací a Zabezpečovací Techniky), z nichž každá může obsahovat určité množství lokalit.

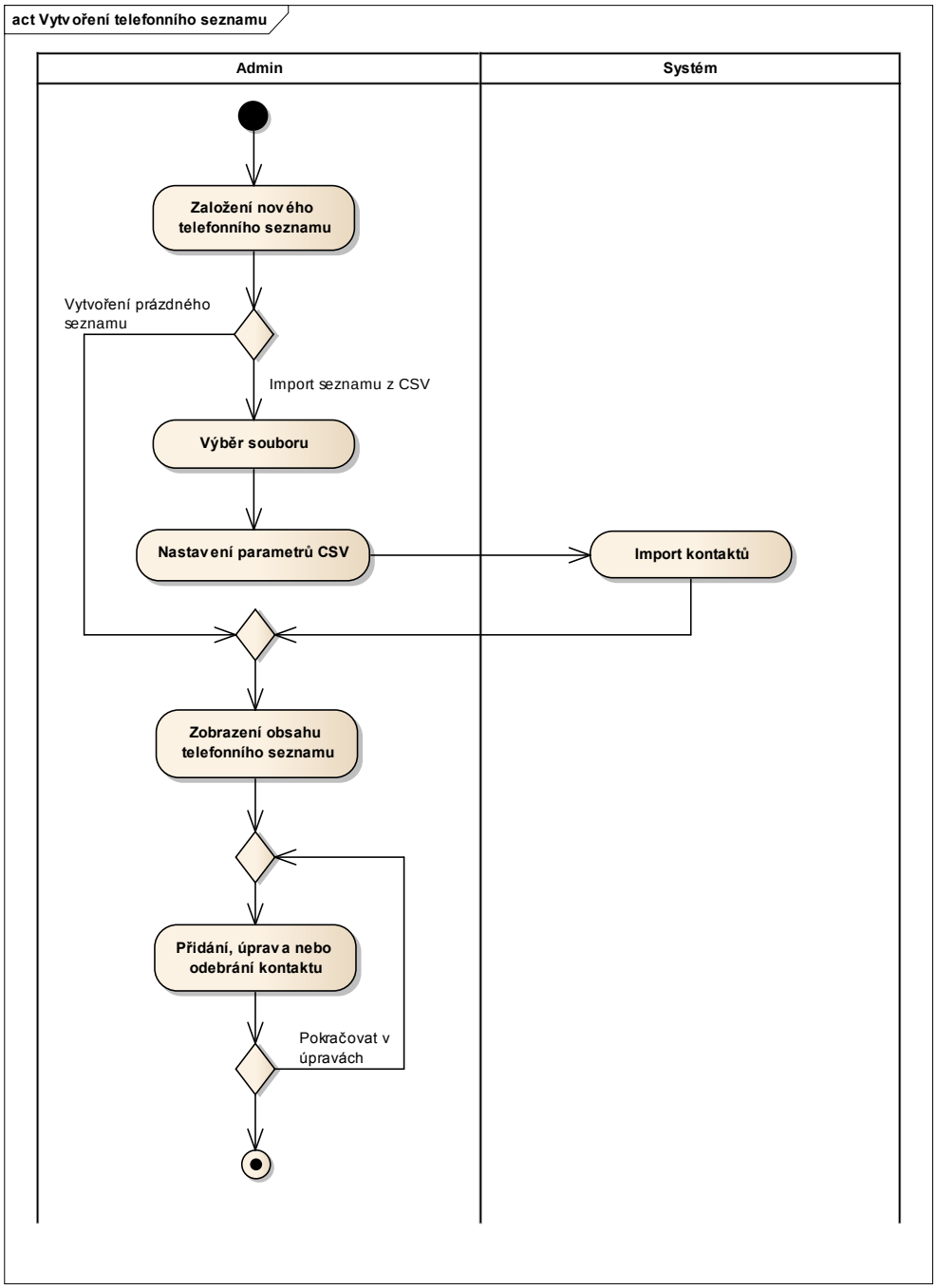
Do těchto lokalit budou administrátoři moci zařazovat terminály i konfigurační profily, díky čemuž je bude možné snadněji vyhledávat a případně k nim omezit přístup právě na základě přiřazení do určité lokality.

4.2.2 Správa terminálů

Aplikace bude umožňovat správu terminálů, které se k danému serveru mohou připojit a synchronizovat. Díky tomu bude možné libovolný terminál obnovit



Obrázek 4.3: Proces nahrání nové verze FW.



Obrázek 4.4: Proces vytvoření nového telefonního seznamu.

ze záloh na serveru.

Administrátoři si budou moci zobrazit informace o konkrétním terminálu a bude jim umožněno nastavit, jaké konfigurační profily na něm mohou být spuštěny, případně jaká je na daném terminálu požadována verze FW.

4.2.3 Správa konfiguračních profilů

Server bude obsahovat profily s konfigurací, které dosud byly ukládány pouze lokálně na terminálech. Pro konfiguraci těchto profilů, které jsou uloženy ve formátu CCS, bude použito stejného rozhraní, jako tomu bylo při lokálních úpravách.

Aplikace si tedy bude držet přehled o jednotlivých profilech a bude umožňovat přístup k jejich konfiguračnímu rozhraní. Při konfiguraci profilu na serveru bude znemožněno jej editovat na terminálu či jiným uživatelem konfigurační aplikace.

Konfigurační profily budou verzovány tak, aby bylo zajištěno jejich spuštění na různých verzích FW, a aplikace tedy bude poskytovat i přehled jednotlivých verzí každého profilu. Aplikace bude rovněž zodpovědná za spuštění správné verze profilu na daném FW.

Jeden profil nepůjde spustit na více terminálech zároveň.

4.2.4 Správa firmware

Aplikace bude umožňovat na server nahrát a následně distribuovat nové verze FW, případně mazat verze staré a nepoužívané. Dále bude aplikace zodpovědná za jejich distribuci na daný terminál, pokud to bude administrátor požadovat.

4.2.5 Centrální telefonní seznam

Server by měl umožňovat tvorbu telefonních seznamů společných pro všechny připojené terminály. Tyto telefonní seznamy bude možné přidávat, upravovat i mazat z administračního rozhraní. Terminálům bude umožněno seznamy jen zobrazovat.

Úprava kontaktů v telefonním seznamu bude v administračním rozhraní umožněna jednak přímou editací a dále pomocí importu ve formátu CSV. Pro případnou práci s kontakty v externím programu bude administrátor moci kontakty exportovat do zmíněného formátu.

4.2.6 Rozcestník

Aplikace si bude udržovat přehled o dalších konfiguračních serverech v síti SŽDC. Tím bude administrátorům umožněno přecházet mezi aplikacemi nasazenými v jednotlivých obvodech a konfigurovat je.

4.3 Nefunkční požadavky

Tato sekce se věnuje omezením, která jsou na aplikaci kladena. Zde však, kromě požadavků od zákazníka, vstoupily do hry také požadavky firmy, v níž byla práce vyvíjena.

4.3.1 Webové rozhraní

Aplikace bude mít přehledné a uživatelsky přívětivé webové GUI.

4.3.2 Integrace do stávajícího projektu

Aplikace bude začleněna do stávajícího projektu jako nový modul a bude využívat některé již vyvinuté funkce, jako například správu uživatelů a jejich přístupu k aplikaci. Z toho důvodu budou také zachovány použité technologie.

4.3.3 Zálohování

Důležitá data, jako třeba konfigurační profily, budou pro každý server zálohována na některém z ostatních serverů v síti SŽDC.

4.4 Model případů užití

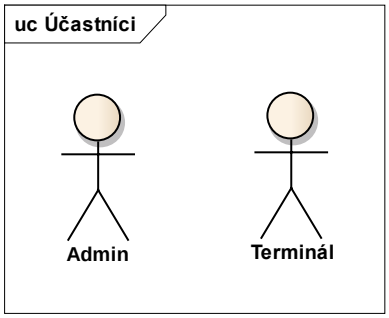
Pro přesné zdokumentování funkčních požadavků proběhlo v rámci analýzy také modelování případů užití, jemuž se věnuje tato sekce. Tvorba tohoto modelu výrazně přispěla k pochopení požadavků a dala základ pro analýzu problémové domény.

4.4.1 Účastníci

Účastníky případů užití analyzovaného systému zachycuje obrázek 4.5. Jak vyplývá z funkčních požadavků, do systému budou především přistupovat administrátoři s cílem konfigurace jednotlivých součástí. Druhým typem účastníků pak budou dispečerské terminály, které budou ze systému získávat data důležitá pro práci dispečerů.

4.4.2 Správa organizační struktury

Skupina případů užití zobrazená na obrázku 4.6 obsahuje případy, které se vážou k požadavkům na správu organizační struktury obvodu, který bude aplikace obsluhovat.



Obrázek 4.5: Účastníci jednotlivých případů užití.

Přidání uzlu do organizační struktury

Administrátor se rozhodne přidat nové SSZT či lokalitu. Aplikace jej nechá vyplnit potřebná data a záznam uloží. V případě, že aministrátor přidává SSZT, aplikace mu umožní do něj rovnou zařadit nové lokality.

Odebrání uzlu z organizační struktury

Aplikace zobrazí vzhled organizační struktury a administrátor vybere uzly (SSZT nebo lokality), které se ze struktury odstraní. V případě, že bude odstraněn uzel obsahující nějaké poduzly, postará se aplikace i o jejich odstranění. Pokud má některý z odstraňovaných uzlů přiřazené konfigurační profily či terminály, zařídí aplikace jejich odebrání z lokality.

Zařazení konfiguračního profilu do lokality

Administrátor si zobrazí seznam profilů a vybere ten, který chce zařadit do lokality. Vybere lokalitu, do které jej chce zařadit a pak nastavení uloží.

Zařazení terminálu do lokality

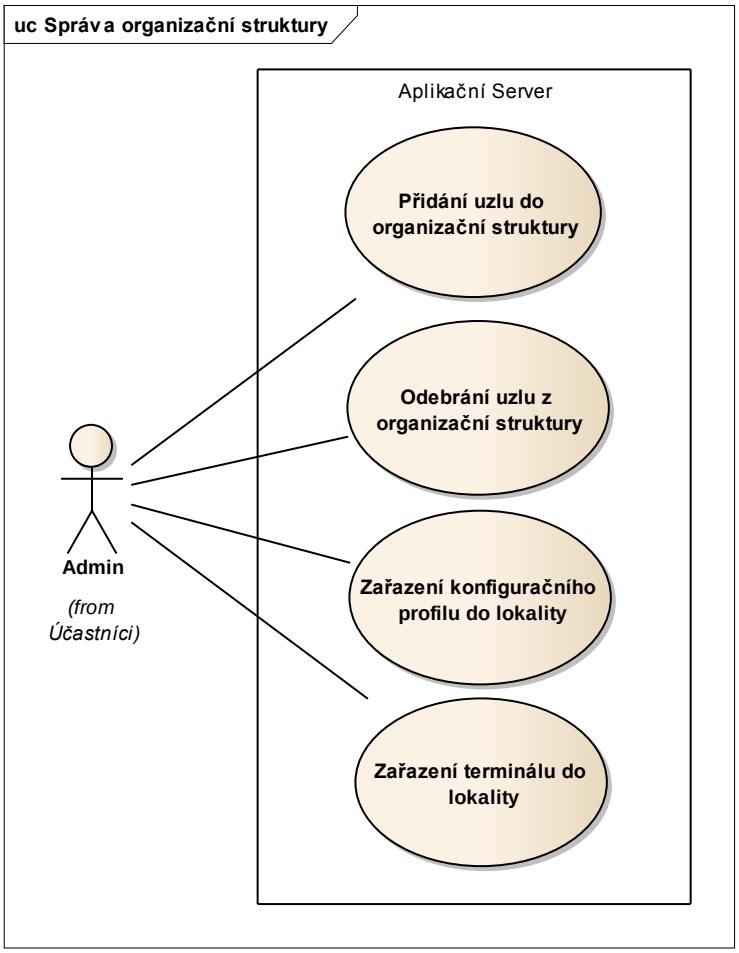
Administrátor vybere ze seznamu terminálů ten, který by měl být zařazen do lokality, přiřadí mu lokalitu a změny uloží.

4.4.3 Správa terminálů

Případy užití, které se týkají všeho kolem správy terminálů, zachycuje obrázek 4.7.

Přidání terminálu do systému

Má-li být do sítě SŽDC přidán nový terminál, založí pro něj administrátor nový záznam v systému. Po vyplnění potřebných údajů aplikace záznam uloží.



Obrázek 4.6: Případy užití, které se týkají správy organizační struktury.

Zobrazení informací o terminálu

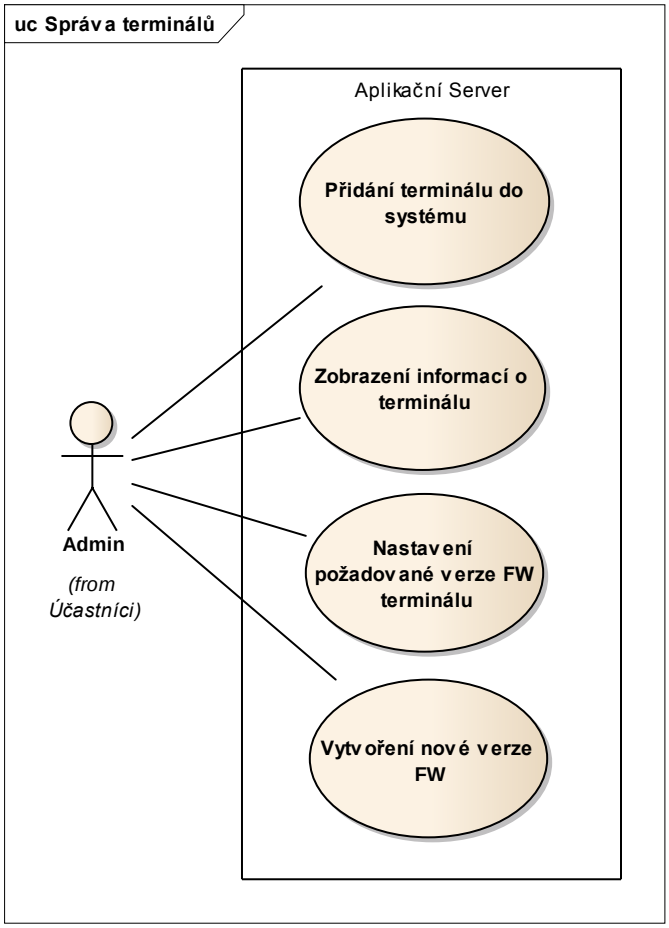
Aplikace zobrazí v přehledném výpisu seznam terminálů se základními informacemi. Administrátor následně může přejít na detail konkrétní položky, kde se mu zobrazí bližší informace, jako třeba seznam přiřazených konfiguračních profilů.

Nastavení požadované verze FW terminálu

Administrátor se rozhodne, že je třeba aktualizovat firmware na terminálu, a vyhledá si jej. Aplikace mu umožní výběr požadované verze FW z verzí uložených v systému a administrátor jednu zvolí. Aplikace změny uloží a zahájí proces aktualizace terminálu.

Vytvoření nové verze FW

Po získání nové verze FW od dodavatele administrátor založí v systému nový záznam pro tuto verzi. Poté nastaví všechny podstatné parametry a vybere z disku archiv s novou verzí. Aplikace jej nahraje na server a všechno uloží.



Obrázek 4.7: Případy užití, které se týkají správy terminálů.

4.4.4 Správa konfiguračních profilů

Diagram případů užití, jež spadají do správy konfiguračních profilů, je na obrázku 4.8.

Přiřazení či odebrání profilu terminálu

Scénář 1: Administrátor je požádán, aby umožnil spuštění určitého konfiguračního profilu na daném terminálu. Vyhledá tento terminál, zobrazí jeho

detail a ze seznamu dostupných profilů vybere ten požadovaný. Aplikace zařídí, že se v tomto seznamu zobrazí jen profily s verzí, která je kompatibilní s FW instalovaným na daném terminálu.

Scénář 2: Administrátor se rozhodne, že daný profil už na určitém terminálu nebude možné spustit. Najde tento terminál v systému a ze seznamu přiřazených profilů mu ten nežádoucí odebere.

Vytvoření nové verze profilu

Po nahrání nové verze FW do systému by byl Administrátor rád, aby na ní fungoval konkrétní konfigurační profil. Proto si tento profil vyhledá a založí pro něj novou verzi. Aplikace zkopíruje konfiguraci a umožní administrátorovi, aby ji podle libosti upravil.

Spuštění profilu

Aplikace terminálu poskytne seznam profilů, které jsou k němu přiřazeny a nejsou spuštěny na jiném terminálu. Terminál nechá uživatele vybrat profil a pak odešle systému žádost o jeho spuštění. Systém poté profil blokuje pro tento terminál tak dlouho, dokud se terminál z profilu neodhlásí.

Úprava konfigurace verze profilu

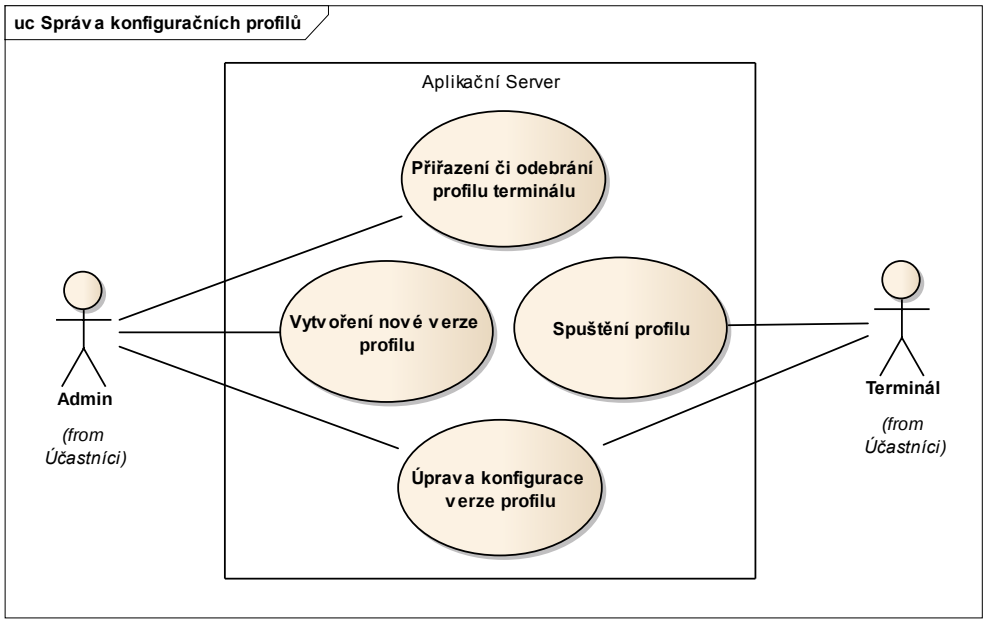
Scénář 1: Administrátor se rozhodne provést nějaké úpravy v konfiguraci verze profilu, což mu aplikace umožní a uzamkne verzi profilu proti editaci jinými uživateli. Administrátor provede své změny a konfiguraci opustí. Aplikace změny uloží a odemkne profil pro další editace.

Scénář 2: Administrátor se rozhodne provést nějaké úpravy v konfiguraci verze profilu, ale zjistí, že se do konfigurace nedostane, protože je zamčená. Počká tedy se svými úpravami na okamžik, kdy jiný administrátor konfiguraci této verze profilu opustí.

Scénář 3: Uživatel terminálu se rozhodne provést nějaké úpravy v konfiguraci. Proto se terminál dotáže systému, zda je možné upravit konfiguraci verze profilu, která je na terminálu spuštěná. V reakci na odpověď systému pak uživateli úpravy konfigurace umožní nebo neumožní.

4.4.5 Centrální telefonní seznam

Poslední skupina případů užití se týká požadavků na centrální telefonní seznam. Tyto případy užití zobrazuje obrázek 4.9.



Obrázek 4.8: Případy užití, které se týkají správy konfiguračních profilů.

Import telefonního seznamu

Administrátor nechce ručně vyplňovat všechny kontakty telefonního seznamu, proto se rozhodne jej importovat z CSV. Nahraje tedy do systému soubor s daty, vyplní parametry CSV (jako třeba oddělovací znak) a namapuje sloupce z CSV na parametry kontaktu používané v systému. Aplikace se pak postará o import hodnot ze souboru do systému.

Export telefonního seznamu

Administrátor se rozhodne exportovat obsah telefonního seznamu, aby s ním mohl pracovat v externí aplikaci. Vybere konkrétní telefonní seznam a aplikace mu vrátí CSV soubor.

Zobrazení telefonního seznamu

Scénář 1: Aplikace administrátorovi zobrazí v přehledném výpisu seznam všech centrálních telefonních seznamů. Administrátor následně může přejít na detail konkrétního seznamu, kde se mu přehledně zobrazí kontakty, které tento telefonní seznam obsahuje.

Scénář 2: Terminál si nechá od aplikace poslat přehled všech telefonních seznamů či pouze jeden telefonní seznam s kontakty, aby je mohl zobrazit uživateli.

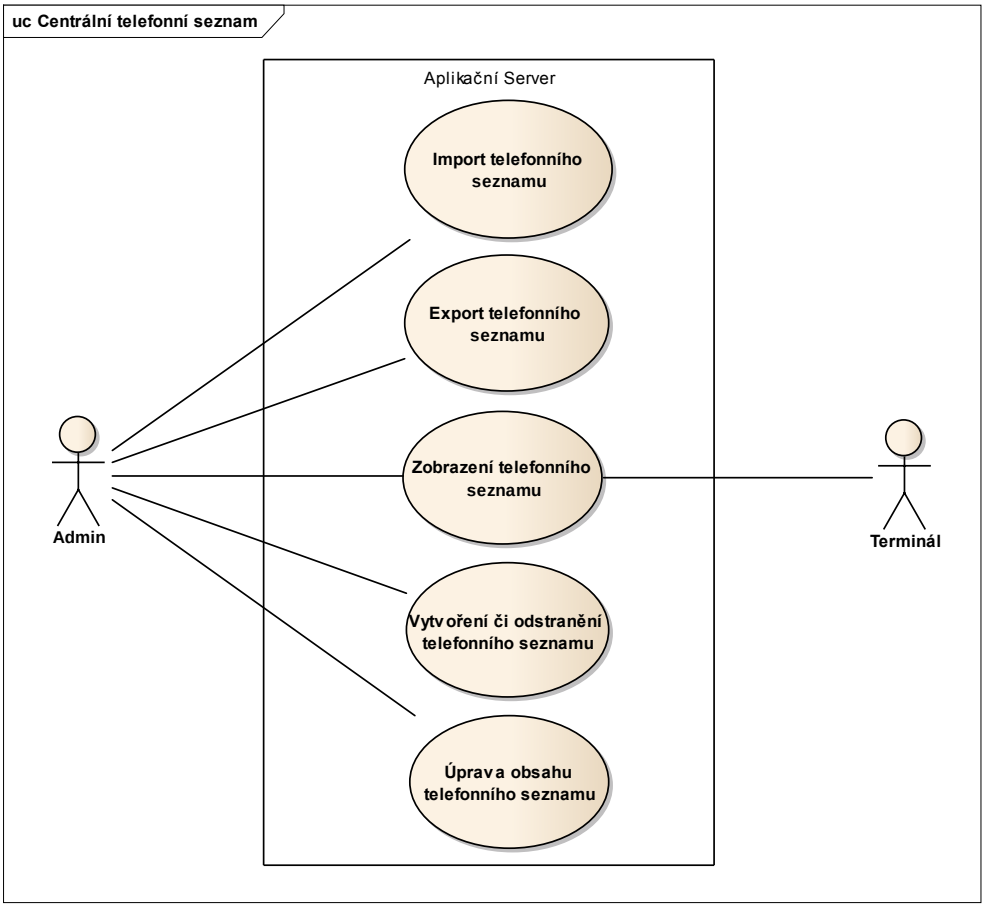
Vytvoření či odstranění telefonního seznamu

Scénář 1: Administrátor vytvoří prázdný telefonní seznam, nastaví jeho parametry a aplikace jej uloží.

Scénář 2: Administrátor se rozhodne telefonní seznam smazat. Aplikace se postará o odstranění všech kontaktů, které v tomto seznamu byly.

Úprava obsahu telefonního seznamu

Administrátor si zobrazí telefonní seznam a začne přidávat, upravovat a odebírat položky, na což bude aplikace ihned reagovat uložením změn a úpravou zobrazení.



Obrázek 4.9: Případy užití, které se týkají centrálního telefonního seznamu.

4.5 Doménový model

Doménový model je poslední částí této kapitoly a zachycuje všechny třídy, jež vzešly z analýzy požadavků a modelu případů užití.

4.5.1 Organizační struktura a rozcestník

Na obrázku 4.10 je zachycen doménový model popisující třídy nalezené při analýze požadavků na organizační strukturu, správu terminálů a konfiguračních profilů, firmware a rozcestník.

Lokalita

Tato třída reprezentuje jeden uzel organizační struktury určený svým názvem. Každý uzel může obsahovat libovolné množství poduzlů, což je naznačeno vztahem této třídy k sobě samé. Dále se lokalita váže ke třídám Terminál a Profil, které mohou být do lokality zařazeny.

Termiál

Třída reprezentující terminál obsahuje několik atributů, které terminál popisují – jeho název, adresu a jednoznačné jméno v rámci sítě, čas a stav poslední synchronizace a stav aktualizace FW. Podstatnější jsou však vazby této třídy k ostatním. Terminál se váže k lokalitě, do které může být přiřazen. Dále obsahuje dvě vazby na firmware. Terminál může mít nainstalován právě jeden firmware a nejvýše jeden firmware na něm může být požadován. Terminál se rovněž váže k třídě reprezentující konfigurační profil, a to tak, že mu může být libovolné množství profilů přiřazeno a nejvýše jeden na něm může být spuštěn.

Firmware

Pro reprezentaci verze FW slouží tato třída, jež obsahuje označení verze FW a informaci o tom, kde je možné ji na disku najít.

Konfigurační profil

Samotná třída konfiguračního profilu je definována pouze svým názvem, ale může mít několik verzí. Jak už bylo zmíněno, váže se profil také na třídy Terminál a Lokalita.

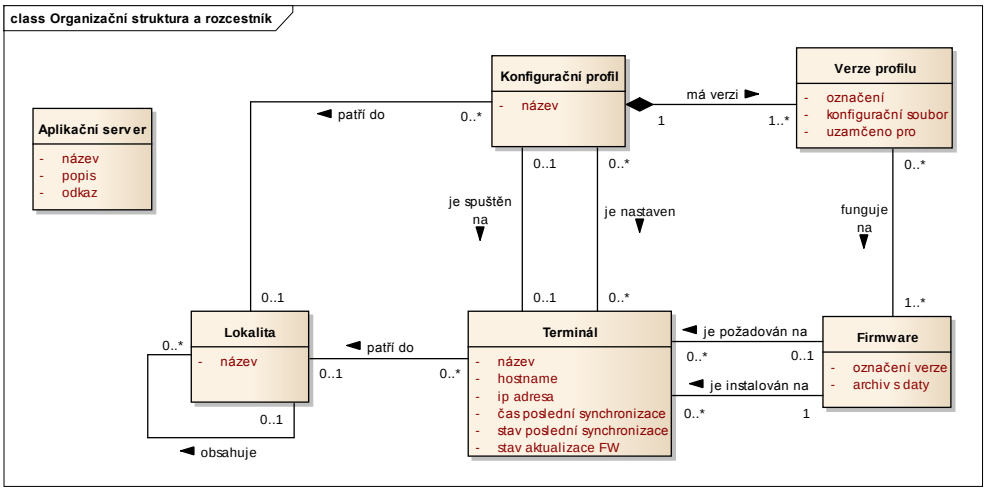
Verze profilu

Tato třída reprezentuje konkrétní verzi profilu danou svým označením a umístěním na disku. Třetím atributem této třídy je pak informace, že je tato verze někým upravována. Verze profilu nemůže existovat bez konfiguračního profilu,

proto je mezi těmito dvěma třídami vztah kompozice. Verze profilu dále musí být spustitelná alespoň na jedné verzi FW.

Aplikační server

Tato třída nemá žádné vazby na ostatní třídy, jelikož se jedná o pouhý nosič informace o dalších serverech v síti SŽDC. Každý server je dán svým názvem, popisem a adresou URL (Uniform Resource Locator).



Obrázek 4.10: Doménový model pro správu organizační struktury.

4.5.2 Telefonní seznamy

Obrázek 4.11 znázorňuje doménový model pro požadavek na centrální telefonní seznam.

Telefonní seznam

Telefonní seznam má několik základních atributů pro jeho popis a dále vazby, které budou vysvětleny v popisu tříd, kterých se týkají.

Konfigurace

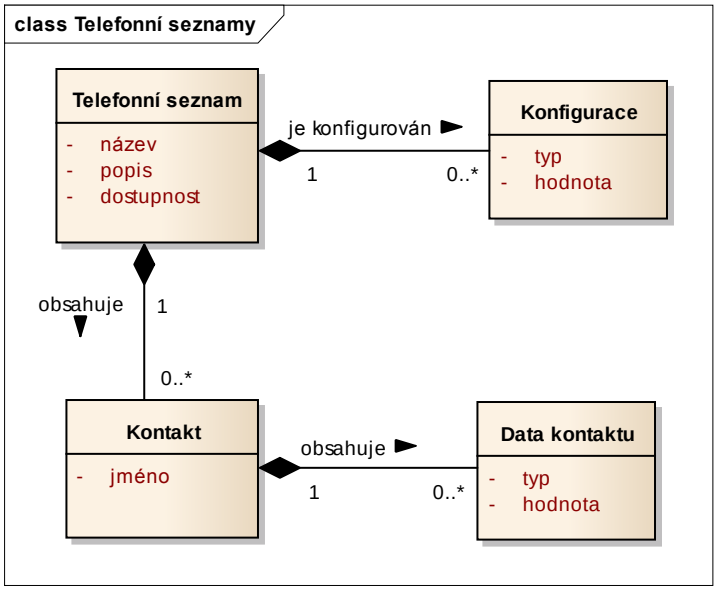
Tato třída představuje část konfigurace pro import telefonního seznamu z CSV, která je určena svým typem. Telefonní seznam nemusí obsahovat žádné konfigurace, a to v případě, že byl vytvořen ručně bez importu z CSV. Konfigurace oproti tomu bez telefonního seznamu nemůže existovat, proto je s ním ve vztahu kompozice.

Kontakt

Třída reprezentující kontakt se rovněž vždy musí vázat k telefonnímu seznamu. Kontakt v systému musí obsahovat jméno, avšak ostatní možné položky jsou nepovinné.

Data kontaktu

Tato třída reprezentuje nepovinnou položku kontaktu. Je určena svým typem a vždy musí být svázána s kontaktem.



Obrázek 4.11: Doménový model pro telefonní seznamy.

Kapitola 5

Návrh

V této kapitole je popsána návrhová část vývoje aplikace. V prvních sekcích jsou popsány použité technologie a architektura aplikace, které byly převedeny ze stávajícího projektu, což byl jeden z nefunkčních požadavků získaných během analýzy. Zbytek kapitoly je věnován návrhu perzistence dat a způsobu komunikace serveru s administračním rozhraním a terminály.

5.1 Výběr technologií

Jelikož bylo dáno, aby byly vyvíjené funkce integrovány do stávajícího projektu, nebyl výběr technologií přímo předmětem práce. Bylo však nutné zanalyzovat a pochopit technologie používané v projektu, aby na ně bylo možné snadno navázat, či navrhnout změny. Použité technologie se ukázaly pro potřebu tohoto systému jako vyhovující, a proto nebylo zapotřebí přijít s návrhem na jejich výměnu.

5.1.1 Technologie společné pro celou aplikaci

V této sekci jsou přiblíženy technologie, které jsou použité v serverové i klientské části aplikace. Jedná se o použitý programovací jazyk, framework a formát, v jakém si server a administrační rozhraní předávají data.

5.1.1.1 Java

Serverová i klientská část aplikace je napsána v jazyce Java ve verzi Java SE 8. Java je objektově orientovaný programovací jazyk založený na syntaxi jazyka C. Byl představen v roce 1995 a od té doby je stále upravován, aby co nejlépe posloužil potřebám vývojářů. Verze SE 8, která je použitá v aplikaci, například přišla s možností využití funkcionálního programování pro jednoduché úkony.

Java kód je kompilován do byte kódu spustitelného na JVM (Java Virtual Machine). Tím se odlišuje od jazyků, které se kompilují přímo pro daný systém a jež je nutné kompilovat (a někdy i programovat) pro každý systém zvlášť. Tuto úlohu zde zastupuje právě JVM, která se postará o spuštění programu na systému, pro který byla napsána. Tím je zajištěna snadná přenositelnost aplikace mezi různými platformami.

Oblíbenost Javy zvyšuje i přítomnost GC (Garbage Collector), který vývojáři usnadňuje práci s pamětí, a také JDK (Java Development Kit) obsahující značné množství standardních knihoven s často používanými funkcemi [16].

5.1.1.2 Spring

Společný pro obě části aplikace je také použitý framework Spring, který se stará o infrastrukturu aplikace a umožňuje vývojářům orientovat se pouze na aplikační logiku.

Spring stojí na principu IoC (Inversion of Control), tedy přebírá odpovědnost za průběh programu. Základem frameworku je Spring Container, který se stará o vytváření objektů, jejich propojování a konfiguraci. Container tedy sám vytváří funkční aplikaci, a to na základě dat poskytnutých vývojářem:

- **Java třídy** ve formě POJO (Plain Old Java Object), tedy třídy, jejichž podoba není frameworkem nijak omezena.
- **Metadata** pro konfiguraci kontejneru. Ta říkají především, o jaké objekty by se framework měl postarat. Tato metadata jsou frameworku dodávána buď v souborech XML (eXtensible Markup Language) nebo ve formě anotací u příslušných tříd.

Jedním ze způsobů, kterým Spring dosahuje IoC, je princip DI (Dependency Injection). Závislosti mezi třídami tedy nejsou dány přímo vývojářem, ale vkládány Spring Containerem na základě metadat. To výrazně usnadňuje údržbu a škálovatelnost aplikace a umožňuje snadnější testování komponent [17].

Kromě samotného IoC poskytuje Spring framework mnoho dalších služeb, jako je třeba podpora testování, webových služeb a přístupu k datům. Tyto technologie budou přiblíženy v dalších sekcích.

5.1.1.3 JSON

Pro přenos dat mezi klientem a serverem je použit formát JSON (JavaScript Object Notation), který byl definován standardem ECMA-404 [18]. Jedná se o kompaktní textový formát, který byl původně vytvořen pro jazyk JavaScript, ale v dnešní době je již podporován řadou programovacích jazyků a stal se jedním ze standardů při komunikaci přes rozhraní postavené na REST architektuře.

5.1.2 Backend (server)

Serverová část aplikace využívá pro přístup do PostgreSQL databáze technologii Hibernate. Pro přenos dat od klienta k databázi pak dále používá některé komponenty Spring frameworku. Všechny tyto technologie jsou postupně popsány v této sekci.

5.1.2.1 PostgreSQL

Pro ukládání dat na serveru je použita databáze PostgreSQL. Jedná se o open source objektově-relační databázový systém vyvinutý v osmdesátých letech na Kaliforské univerzitě v Berkeley. PostgreSQL využívá a rozšiřuje jazyk SQL a díky tomu, že se jedná o open source, je možné jej snadno rozšiřovat [19].

5.1.2.2 Hibernate

O přístup k databázi se zde stará ORM (Object-Relational Mapping) Hibernate. Jako každé jiné ORM je jejím úkolem převod objektů používaných v aplikaci na jejich reprezentaci v databázi, čímž odebírá vývojáři starost s tím, jak se přesně perzistence provede. Výhodou Hibernate je vysoký výkon, podporovaný například pomocí taktiky Lazy Initialization[1]. Hibernate také implementuje standardizované JPA (Java Persistence API), což usnadňuje její použití v libovolném systému či nahrazení za jiné ORM podporující JPA [20].

5.1.2.3 Spring Data

Aplikace využívá pro přístup k Hibernate jednu z technologií frameworku Spring. Cílem projektu Spring Data je redukování nutnosti psaní tzv. boilerplate[2] kódu pro přístup k datům. Jeho zásadní funkcí je schopnost automaticky vytvořit implementaci úložiště pouze na základě dodaného rozhraní. Vývojáři tedy ve většině případů stačí pouze definovat rozhraní a nemusí se dále o přístup k databázi starat. Samozřejmostí je i možnost toto kombinovat s vlastní implementací, a to například v případě náročnějších dotazů [21].

Pro doplnění automaticky generované implementace o vlastní dotazy je v aplikaci využíváno Spring Data podpory dotazovacího jazyka Querydsl. Tento jazyk při překladu aplikace generuje speciální třídy, jež reprezentují databázové objekty. Vygenerované třídy jsou opatřeny prefixem Q a Querydsl s jejich pomocí umožňuje snadnou tvorbu typově bezpečných dotazů [22].

[1] Instance objektů v asociaci se nevytváří při tvorbě jednoho z nich, ale až tehdy, kdy jsou skutečně potřeba.

[2] Text, který může být jen s drobnou obměnou znovu použit v novém kontextu.

5.1.2.4 Spring Web MVC

Další technologie frameworku Spring je využito pro implementaci přístupu administračního rozhraní či dispečerských terminálů k serverové části aplikace. Jak již název napovídá, Spring Web MVC (nebo zkráceně Spring MVC) poskytuje MVC architekturu a komponenty použitelné pro vývoj webových aplikací. Architekturu zde však netvoří čisté MVC, ale je doplněno o návrhový vzor Front controller. Framework tedy poskytuje Dispatcher Servlet, který hraje roli front controlleru a přijímá HTTP požadavky. Tento Dispatcher Servlet poté vyhledá controller, který je schopen požadavek zpracovat, a deleguje práci na něj. Jako odpověď dostává zprávu, jaké view by mělo být zobrazeno, společně s modelem obsahujícím data. Dispatcher Servlet pak na základě toho nechá vytvořit příslušné view a odešle jej HTTP odpovědí zpět tam, odkud přišel požadavek.

Díky přítomnosti Dispatcher Servletu je tedy na vývojáři opět pouze implementace vlastních tříd a nemusí se starat o komunikaci s klientem.

Spring MVC samozřejmě podporuje vše, co webová aplikace ke svému provozu potřebuje. Je zde možnost zpracování parametrů URL či přímo těla požadavku. Dále je zde snadné čtení i nastavení HTTP hlaviček a Dispatcher Servlet podporuje snadný způsob ošetřování výjimek a vracení chybové zprávy či view klientovi [23].

5.1.2.5 Spring MVC pro REST

Při vývoji RESTful API není záhodno vracet klientovi konkrétní view, ale pouze reprezentaci zdroje. Proto Spring MVC umožňuje úplně obejít tu část, kdy Dispatcher Controller vytváří view, a odeslat přímo reprezentaci objektu vráceného daným controllerem.

Zde je také nutno podotknout, že se Spring MVC sám stará i o překlad mezi serverovými objekty a formátem posílaným v těle HTTP požadavku či odpovědi. To opět ulehčuje vývojáři práci s psaním boilerplate kódu [23].

5.1.3 Frontend (administrační rozhraní)

Poslední část sekce, jež se zabývá použitými technologiemi, je věnována administračnímu rozhraní aplikace. Zde bylo důležité pochopit především způsob, jakým aplikace komunikuje se serverovou částí, a dále jak jsou data zobrazována uživateli.

5.1.3.1 Apache CXF

Pro komunikaci se serverem využívá administrační rozhraní frameworku Apache CXF (Celtix and XFire). Jedná se o open source projekt, který poskytuje snadný způsob vývoje webových služeb s podporou XML, JSON,

SOAP, REST a dalších. Ač je tento framework používán většinou ke specifikaci webových služeb na straně serveru, tak podporuje i tvorbu klientských protistran.

Pro účely tvorby RESTful služeb a klientů implementuje Apache CXF specifikaci JAX-RS (Java API for RESTful Web Services). Tato specifikace definuje anotace, výjimky a rozhraní, které slouží ke vkládání obsahu HTTP požadavků do Java metod. Framework pak tato rozhraní implementuje a využívá komponenty, aby umožnil nejen zpracování požadavků na straně serveru, ale také postup opačný, kdy je na klientovi, aby požadavek vytvořil a odeslal [24].

Apache CXF specifikuje pro tvorbu klientů dva přístupy:

- **Proxy-based API** využívá právě výše zmíněného JAX-RS a umožňuje využít rozhraní doplněná o anotace jako definice REST zdrojů. Vývojáři tedy stačí definovat rozhraní, která framework následně zapouzdří do zástupců (proxy). Proxy se poté v případě volání metod rozhraní postará na základě anotací o sestavení HTTP požadavku a předá jej frameworku ke zpracování. Výhodou tohoto přístupu může být i možnost znovupoužití rozhraní definovaných na serveru, tedy v případě, že i serverová část používá Apache CXF.
- **WebClient API** vyžaduje specifikaci každého dotazu v momentě, kdy jej aplikace volá. Vývojář specifikuje dotaz příkazy přes dané rozhraní frameworku a nakonec spustí odeslání požadavku příslušnou metodou tohoto rozhraní.

Framework se v obou případech postará o odeslání HTTP požadavku i následné přijetí odpovědi a její překlad na Java objekty využívané aplikací či na objekt reprezentující celou odpověď. Samozřejmostí je i reagování na problémovou odpověď serveru, které je zde řešeno pomocí výjimek specifikace JAX-RS [25].

5.1.3.2 Vaadin

O zobrazení dat uživateli ve webovém prohlížeči se stará framework Vaadin, což je open source Java framework pro vývoj webových uživatelských rozhraní. Tento framework umožňuje za použití pouze Javy implementovat interaktivní webové aplikace. Vývojář je tedy odstíněn od samotného HTML (HyperText Markup Language) či CSS (Cascading Style Sheet) a i tak je schopen tvořit rozhraní bohatá na komponenty a vizuálně uspokojivá.

Vaadin sestává ze serverového frameworku a klientského enginu. Engine běží jako JavaScript kód v prohlížeči, stará se o zobrazení uživatelského rozhraní a posílání uživatelské interakce serveru. Logika zobrazování uživatelského rozhraní a reakce na uživatelské vstupy je pak řešena ve frameworku na serveru. Komunikace enginu s frameworkem je vývojáři standardně skryta [26].

Velkou výhodou Vaadinu je především možnost vyvíjet klientskou i serverovou část webové aplikace v jednom jazyce. Dalšími výhodami je pak dobrá dokumentace, snadný vývoj jednoduchých aplikací a početná aktivní komunita, díky které existuje velké množství GUI komponent. Tyto komponenty však mohou být v některých případech překládány na ne moc přívětivý HTML kód, což následně znesnadňuje automatické testování GUI, a to samozřejmě může být nevýhodou. Vaadin také není příliš vhodný pro tvorbu aplikací s komplikovanějším GUI, které by vyžadovalo tvorbu nových vlastních komponent, či pro aplikace s vysokým počtem uživatelů. Pro potřeby aplikace, jež je obsahem této práce, je však dostačující.

5.2 Architektura

Podobně, jako u použitých technologií, i zde byla klíčová analýza stávajícího projektu pro bezproblémové zapojení vytvářených funkcí.

Jelikož je aplikace tvořena serverovou částí, ke které se mají připojovat dispečerské terminály, a administračním rozhraním, je zde logické použití architektury klient-server. Server vystavuje RESTful API, na které se pak mohou připojit terminály i administrační rozhraní. V důsledku toho je tedy aplikace tvořena dvěma různými programy, jejichž architektura je podrobně popsána v následujících podsekcích.

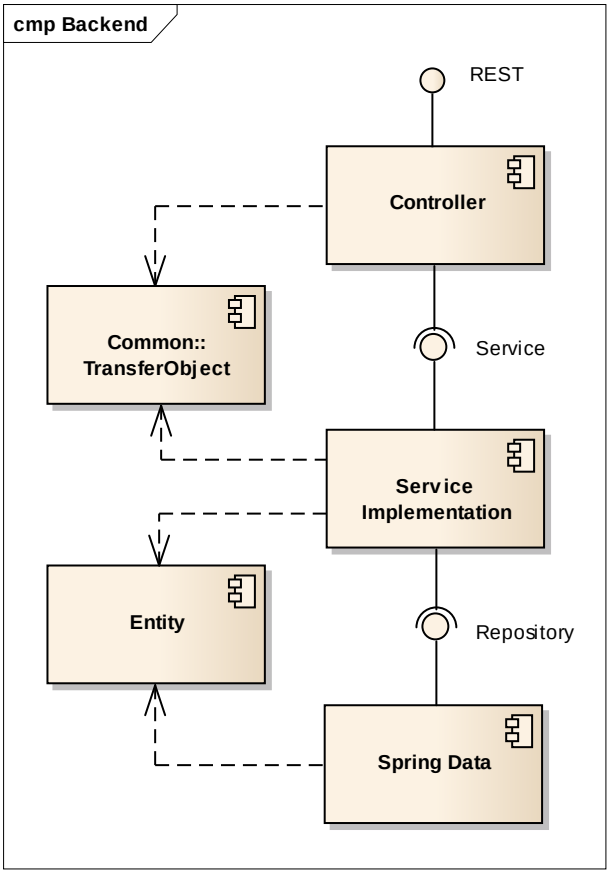
Aplikace využívá návrhového vzoru TO (Transfer Object), proto jsou obě části aplikace závislé na externích TO třídách, které zajišťují jednotnou reprezentaci dat na obou stranách RESTful API.

5.2.1 Architektura serverové části

Diagram na obrázku 5.1 znázorňuje třívrstvou architekturu serverové části aplikace. První vrstva (datová) je zde tvořena komponentami Spring Data zmíněného frameworku, které se starají o načítání dat z databáze a jejich mapování na entity používané aplikací. K těmto komponentám se přistupuje přes rozhraní resource, jež musí být vytvořeno pro každý typ entity, která má být z databáze načítána.

O aplikační logiku se starají implementace rozhraní service, jejichž úkolem je především překlad mezi serverovými entitami a třídami TO. V této vrstvě se také provádí validace příchozích dat a další složitější logické operace.

Prezentační vrstva je založena na Spring MVC architektuře v podobě pro RESTful API a tvoří ji tedy jen controllery bez view a modelů. Každý z controllerů pouze definuje podobu rozhraní, které má na starosti. Veškerou logiku pak přenechává výše zmíněným service třídám.



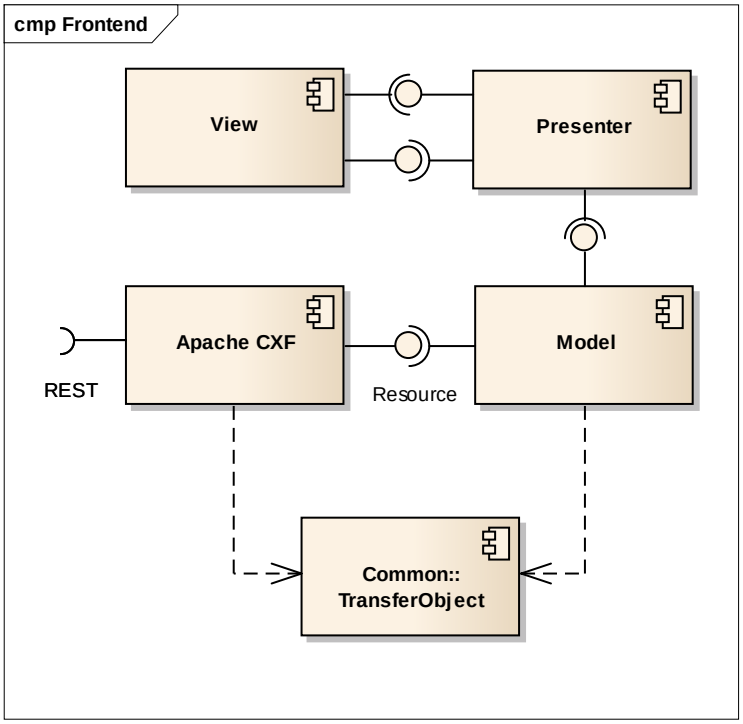
Obrázek 5.1: Architektura serverové části aplikace.

5.2.2 Architektura klientské části

Architektura klientské časti, jež je znázorněna na obrázku 5.2, je postavena podle vzoru MVP ve verzi passive view. Jedná se pouze o tenkého klienta, veškerá složitější logika je ponechána serveru. View používají komponenty frameworku Vaadin, které zajišťují validaci dat a jejich zobrazení v prohlížeči.

Každé view je rozhraním připojeno k presenteru, který se stará především o přenos dat mezi modelem a view. Presentery rovněž obsahují logiku reakcí na uživatelovu interakci s aplikací, jako je například stisk tlačítka a podobně. V případě potřeby je presenteru také umožněno přímo ovlivnit view, tedy třeba tehdy, kdy je potřeba zobrazit chybové hlášení uživateli.

Model se stará o konverzi mezi vnitřní reprezentací dat frameworku Vaadin a externími TO, které přichází ze serverové části aplikace. K získání těchto dat používá rozhraní resource, o jejichž implemetaci se stará framework Apache CXF, který zajišťuje komunikaci s RESTful API serveru.



Obrázek 5.2: Architektura klientské části aplikace.

5.3 Databázový model

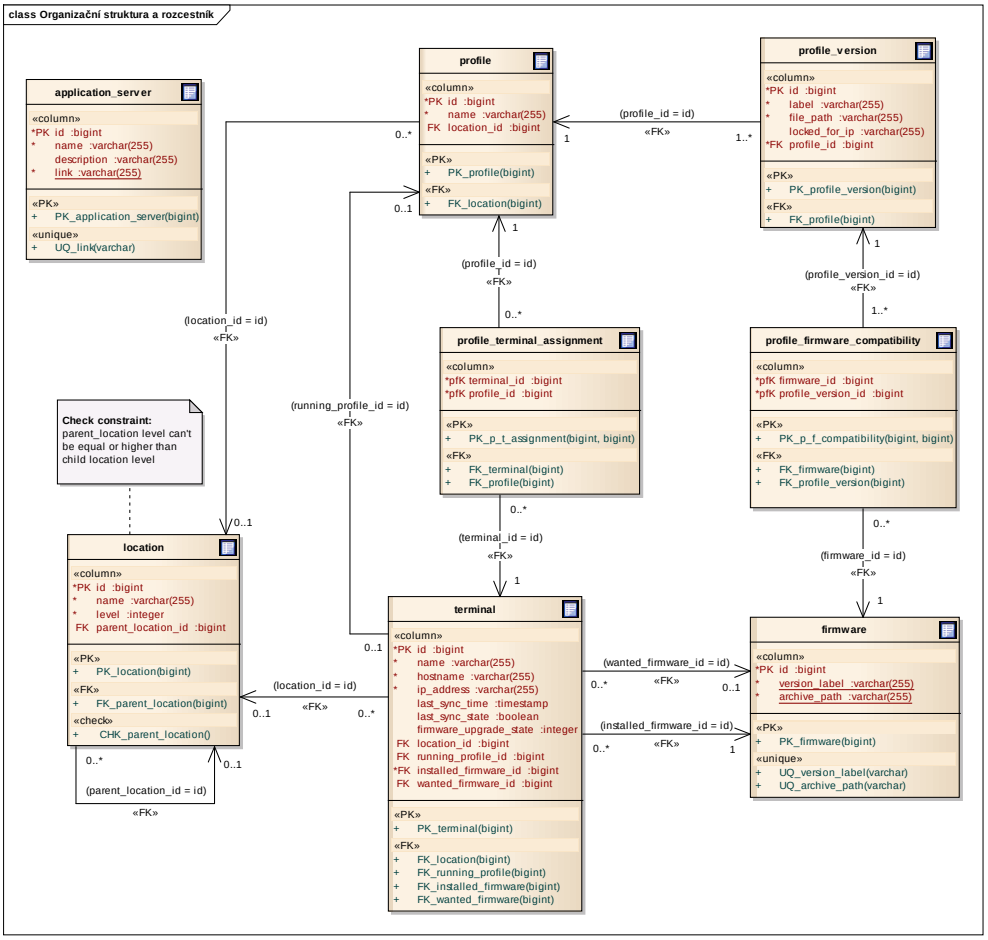
Databázový model aplikace vychází z analýzy problémové domény provedené v sekci 4.5. Přibyly zde datové typy jednotlivých atributů entit, omezení na neprázdnost či jedinečnost hodnot a samozřejmě také primární a cizí klíče specifikující vazby mezi tabulkami. Model byl také převeden do anglického jazyka, aby se předešlo míchání českého a anglického jazyka při implementaci, jež by mohlo vést ke zmatení vývojářů při budoucí údržbě aplikace.

Kromě databáze slouží tento model jako zdroj entit používaných v serverové části aplikace, jež jsou na tabulky databáze mapovány jedna ku jedné pomocí ORM Hibernate.

5.3.1 Organizační struktura a rozcestník

Model tabulek organizační struktury znázorňuje diagram na obrázku 5.3. Oproti příslušnému doménovému modelu v sekci 4.5.1 se liší především v dekompozici m:n vazeb reprezentujících přiřazení konfiguračního profilu k terminálu a spustitelnost verze konfiguračního profilu na konkrétní verzi FW. Kromě tabulek reprezentujících jednotlivé třídy doménového modelu tak vznikly dvě nové tabulky profile_terminal_assignment a profile_version_compatibility.

Další podstatnou částí návrhu tohoto modelu bylo přidání atributu level do entity reprezentující lokalitu. Tento atribut je využíván integritním omezením, jež omezuje možnost přiřazení lokality pouze do takových lokalit, které jsou na nižší úrovni než lokalita přiřazovaná.

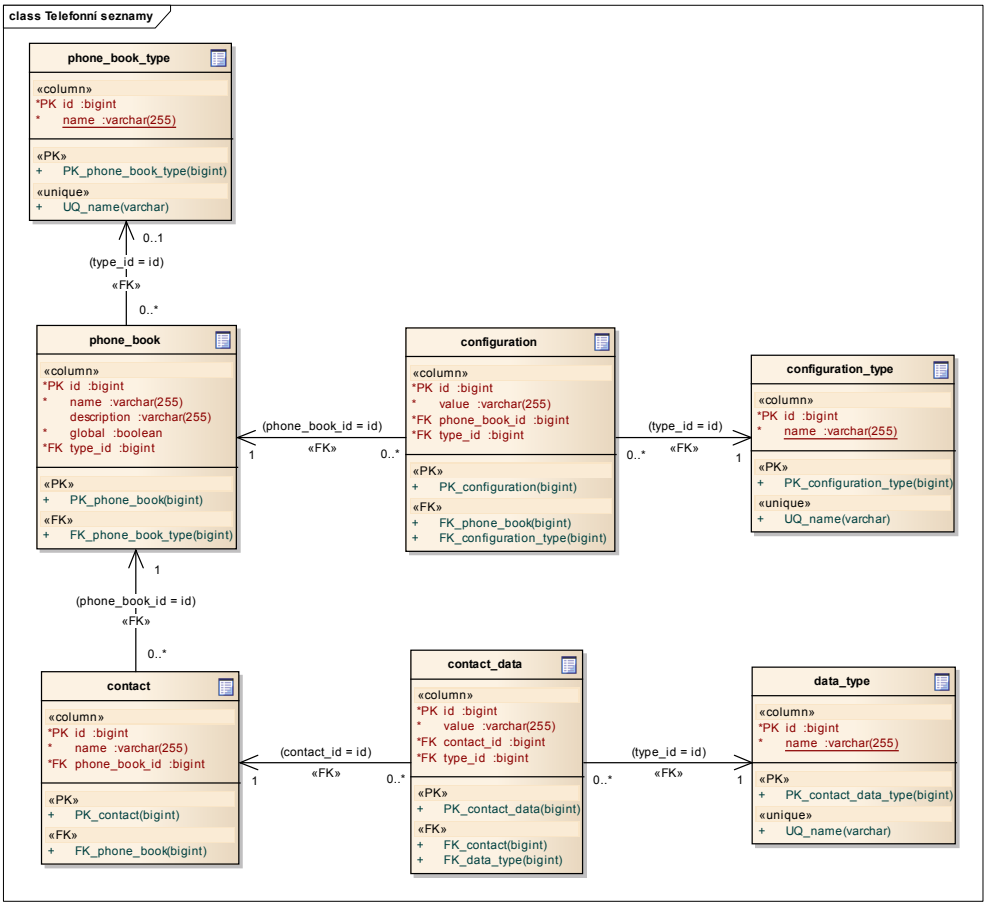


Obrázek 5.3: Databázový model pro správu organizační struktury.

5.3.2 Telefonní seznamy

Diagram na obrázku 5.4 představuje strukturu databáze pro telefonní seznamy. K původním entitám získaným z doménového modelu zde přibyly tři tabulky pro datové typy, které byly vyčleněny z příslušných entit, aby nedocházelo ke zbytečnému duplikování záznamů. Tabulka data_type tedy představuje původní atribut typ doménové třídy Data kontaktu a tabulka configuration_type dělá to samé pro doménovou třídu Konfigurace. Tabulka phone_book_type nemá v doménovém modelu svůj ekvivalent, vznikla pouze

pro účely implementace, aby bylo možné odlišit, zda pochází telefonní seznam z externího zdroje a jakého typu tento zdroj je. To je důležité zejména pro začlenění do stávajícího projektu, kde se předpokládá použití i jiných zdrojů, než je pouze CSV soubor.



Obrázek 5.4: Databázový model pro telefonní seznamy.

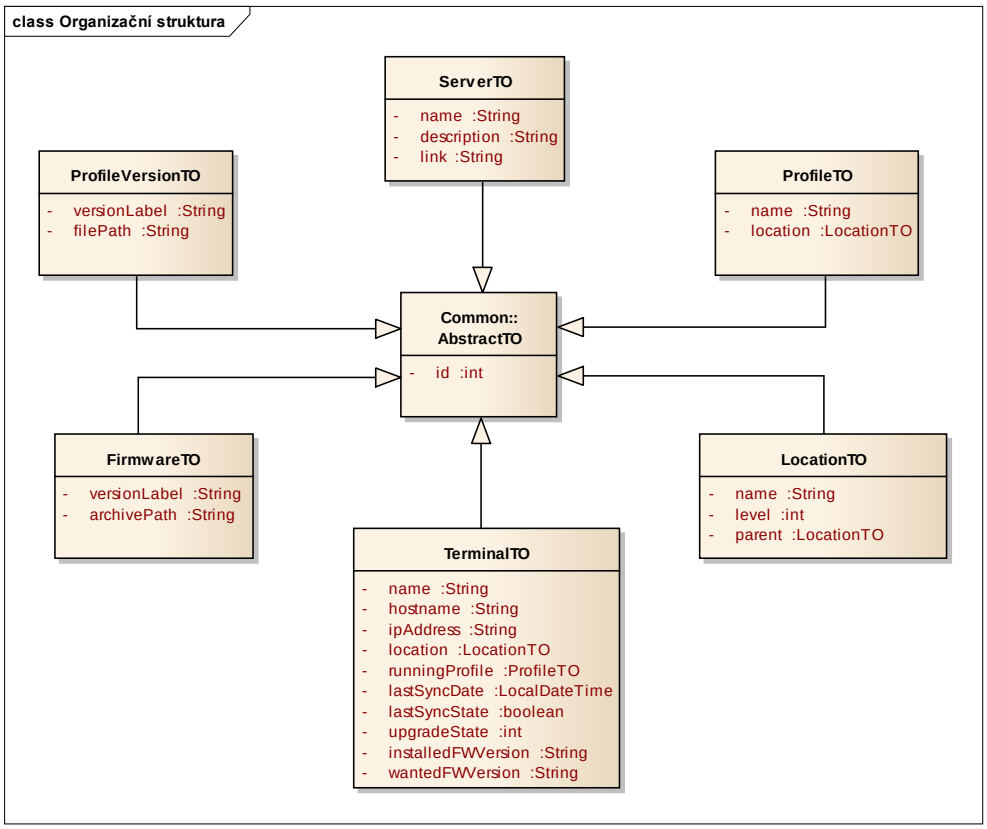
5.4 Návrh RESTful API

Poslední sekce této kapitoly je věnována návrhu RESTful rozhraní serveru. Cílem bylo navrhnout rozhraní co nejpřehledněji a aby co nejlépe splňovalo RESTful podmínky. Co se těch týče, tak bylo rozhraní záměrně navrhnuto jako REST druhé úrovně, jelikož by zakomponování HATEOAS v tomto případě nepřineslo více výhod než komplikací. Stále se však jedná o výrazné zlepšení oproti stávajícímu projektu, který vesměs dosahuje pouze nulté úrovně a není příliš přehledný.

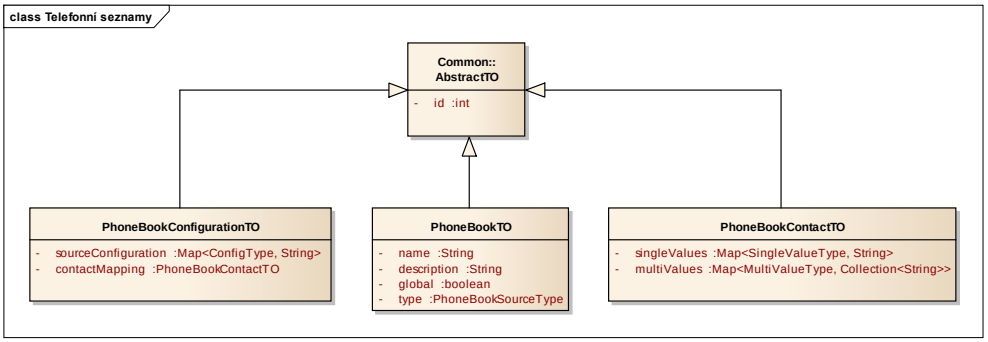
V následujících odstavcích jsou blíže popsány přenášené objekty, stavy, kterými aplikace může odpovědět na požadavek uživatele, a samozřejmě také adresy zdrojů a příslušné metody, které mohou klienti použít pro získání dat. Detailní specifikace tohoto API je pak sepsána v příloze B.

5.4.1 Přenášené objekty

Obrázky 5.5 a 5.6 zobrazují TO třídy, které serverová aplikace vyžaduje či odesílá v odpovědích. Tyto třídy víceméně odpovídají databázovému modelu, ale samozřejmě jsou zde kladeny nároky na jednoduchost přenášených objektů. API specifikace pak uvádí podobu těchto tříd v JSON formátu a převádí do textových řetězců komplexnější parametry, jako jsou například výčty typu enum či Java třídy reprezentující datum. Jde především o usnadnění jejich přenosu v HTTP odpovědi a následné jednoduché přeložení na třídy jazyka použitého na klientovi. Administrační rozhraní aplikace je ale také implementováno v jazyce Java, a proto i ono používá tyto TO. Kdyby však v budoucnu přišlo rozhodnutí o převedení klienta do jiného jazyka, bude stále možné toto rozhraní a definované JSON objekty použít.



Obrázek 5.5: Model TO tříd pro organizační strukturu.



Obrázek 5.6: Model TO tříd pro telefonní seznamy.

5.4.2 Stavové kódy a chybové zprávy

REST specifikace říká, že by měl být klient informován o úspěšnosti svých požadavků pomocí HTTP stavových kódů. Proto je tedy nutné zahrnout do návrhu API i je. Stavové kódy navržené pro konfigurační server odpovídají konvencím a indikují úspěch či neúspěch, který je dále vysvětlen v těle odpovědi.

5.4.2.1 Úspěch

Základními stavovými kódy jsou kódy 2xx, jež značí úspěšné zpracování požadavku. V aplikaci jsou použity tyto:

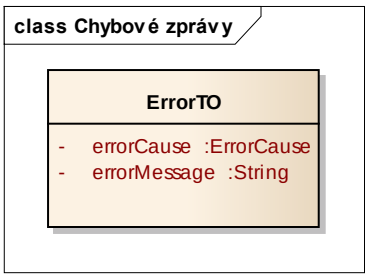
- **200 OK** značí, že byl GET či PUT požadavek úspěšně zpracován a v těle zprávy je vrácena reprezentace požadovaného či upraveného zdroje.
- **201 Created** je obdobou kódu 200 v případě POST požadavku a značí, že se podařilo zdroj úspěšně vytvořit.
- **202 Accepted** posílá aplikace v reakci na požadavek o přenos nové verze firmware na terminál a také jako odpověď na úspěšný pokus o import kontaktů do telefonního seznamu. Tento kód vyjadřuje přijetí požadavku a informaci, že bylo zahájeno jeho zpracování, které bude nějakou dobu trvat.
- **204 No Content** je vracen v takovém případě, kdy není smysluplné posílat v těle odpovědi nějaká data. Tento kód je proto vracen v odpovědi na všechny DELETE požadavky a na dotazy, zda je na terminálu požadován nový firmware.

5.4.2.2 Chyba

V aplikaci může snadno docházet k chybám, jako jsou například špatné uživatelské vstupy, a především ty je třeba předat klientovi v rozumné podobě. Proto jsou následující stavové kódy podstatnější, než kódy v předchozí sekci.

- **400 Bad Request** je obecný kód indikující chybu požadavku od klienta. V aplikaci je tento kód nastaven v případě, že nejsou vyplněna některá povinná pole pro vytvoření daného záznamu v databázi, a také tehdy, když jsou porušena omezení na unikátnost záznamu.
- **403 Forbidden** používá aplikace pro indikaci, že klientovi není povoleno žádaný krok provést. Tedy v okamžiku, kdy se terminál snaží spustit konfigurační profil, který již běží na jiném terminálu, a v případě že se některý z administrátorů pokouší editovat verzi konfiguračního profilu, kterou právě upravuje někdo jiný.
- **404 Not Found** vzniká tehdy, kdy není na serveru nalezena požadovaná entita, či se klient snaží o přístup k neznámým URI.
- **405 Method Not Allowed** je vyvolán přímo Spring frameworkem v případě, že se klient snaží použít pro přistup ke zdroji na známé URI metodu, která není pro tuto URI definována v API specifikaci.
- **500 Internal Server Error** značí nějakou neočekávanou chybu na serveru, tedy například problém s databází a podobně.

Jelikož samotný HTTP stavový kód obvykle neřekne o chybě mnoho, jsou tyto kódy v těle odpovědi doplněny o TO reprezentující chybu. Formát tohoto TO zachycuje obrázek 5.7 a skládá se ze stručného popisu příčiny chyby a doplňující detailní zprávy.



Obrázek 5.7: TO pro chybové HTTP odpovědi.

5.4.3 Zdroje a metody

Následuje přehled jednotlivých zdrojů, které aplikace poskytuje, a metod, jež je možné použít k manipulaci s nimi.

5.4.3.1 Organizační struktura

Pro přístup k jednotlivým prvkům organizační struktury je možné využívat rozhraní, která znázorňuje tabulka 5.1. Za zmínku zde stojí především zdroj pro profil spuštěný na terminálu, který slouží pro přihlašování a odhlašování se z konfiguračního profilu na terminálu. Další zajímavostí je seznam terminálů, kam je daný profil přiřazen. Tento zdroj slouží pouze pro čtení, úprava přiřazení profilu k terminálu je možná pouze ze strany terminálu.

URI	**Popis**	**Metody**
Ostatní servery		
servers	Seznam konfiguračních serverů	POST, GET
servers/{id}	Konkrétní konfigurační server	PUT, DELETE
Lokality		
locations	Seznam lokalit	POST, GET
locations/{id}	Konkrétní lokalita	PUT, DELETE
Terminály		
terminals	Seznam terminálů	POST, GET
terminals/{id}	Konkrétní terminál	PUT, DELETE
terminals/{id} /firmware	Firmware požadovaný na terminálu	GET
terminals/{id} /profiles	Seznam profilů přiřazených danému terminálu	GET, PUT
terminals/{id} /profiles/{pid}	Profil spuštěný na daném terminálu	POST, DELETE
Firmware		
firmware	Seznam verzí FW	POST, GET
firmware/{id}	Konkrétní verze FW	PUT, DELETE
Konfigurační profily		
profiles	Seznam konfiguračních profilů	POST, GET
profiles/{id}	Konkrétní konfigurační profil	PUT, DELETE
profiles/{id} /terminals	Seznam terminálů, kam je daný profil přiřazen	GET
Verze profilů		
profiles/{id} /versions	Seznam verzí profilu	POST, GET
profiles/{id} /versions/{vid}	Konkrétní verze profilu	PUT, DELETE
profiles/{id} /versions/{vid} /firmware	Seznam verzí FW na kterých je daná verze profilu spustitelná	GET, PUT

Tabulka 5.1: RESTful API pro správu organizační struktury.

5.4.3.2 Telefonní seznamy

Pro přístup k telefonním seznamům slouží metody a zdroje, které jsou popsány v tabulce 5.2. Je možné vidět, že konfigurace je oddělená od zdroje telefonních seznamů, a to především z důvodu použitelnosti stejného zdroje telefonních seznamů jak pro dotazy z administračního rozhraní, tak i v případě dispečerských terminálů, které se o konfiguraci telefonních seznamů nestarají.

URI	**Popis**	**Metody**
Telefonní seznamy		
phonebooks	Seznam telefonních seznamů	POST, GET
phonebooks/{id}	Konkrétní telefonní seznam	GET, PUT, DELETE
Kontakty		
phonebooks/{id} /contacts	Seznam kontaktů v telefonním seznamu	POST, GET
phonebooks/{id} /contacts/{cid}	Konkrétní kontakt	GET, PUT, DELETE
Konfigurace telefonních seznamů		
phonebooks/configs	Seznam konfigurací všech telefonních seznamů	GET
phonebooks/configs/{id}	Konfigurace pro daný telefonní seznam	GET, PUT
Synchronizace telefonních seznamů		
phonebooks/{id} /synchronization	Import či export telefonního seznamu	POST, GET

Tabulka 5.2: RESTful API pro telefonní seznamy.

Kapitola 6

Implementace a testování

V rámci této kapitoly jsou popsány detaily implementace vybraných funkcí aplikace a dále způsob, jakým byly tyto funkce testovány.

6.1 Implementace

Protože předmětem práce nebyla realizace celé aplikace, byl prvním krokem této části výběr funkcí, které budou implementovány. Na základě návrhu byly k tomuto účelu vybrány funkce týkající se telefonních seznamů. Hlavním důvodem tohoto výběru byla jejich nevětší využitelnost nejen v síti SŽDC, ale i ve stávajícím projektu a u dalších zákazníků.

Na serveru i v administračním rozhraní tedy vznikly funkce podporující CRUD operace nad telefonními seznamy a kontakty v nich. Dále byla implementována možnost konfigurace zdroje kontaktů pro telefonní seznam včetně namapování polí ze zdroje na jednotlivé typy kontaktních informací ukládané v aplikaci. Poslední implementovanou funkcí pak byl import kontaktů z CSV souboru, který by měl být v krátké době doplněn i o export.

V následujících podsekcích je přiblížen vývoj serverové i klentské části aplikace včetně ukázky zdrojových kódů pro klíčové funkce. Kompletní zdrojové kódy jsou pak k nahlédnutí na přiloženém médiu.

6.1.1 Serverová část

Podoba serverových tříd vychází především z návrhu databáze a API. Vývoj byl rozdělen podle jednotlivých částí API, které byly navrženy v sekci 5.4. Pro každou z těchto částí se vytvořily komponenty v souladu s navrženou architekturou.

V nasledujících odstavcích jsou kromě těchto komponent popsány další podpůrné funkce, které byly potřeba implementovat pro úspěšný běh aplikace. Mezi tyto funkce patří zpracování výjimek a import kontaktů telefonního seznamu ze souboru CSV.

6.1.1.1 Entity

Prvním krokem v implemetaci každé z částí bylo vytvoření entit na základě databázového modelu. Příklad takové entity pro kontakt v telefonním seznamu ukazuje výpis 6.1. Tyto entity jsou vytvořeny za pomoci JPA anotací, které umožňují definovat mapování entity a jejích parametrů na tabulku s jejími sloupci. Mezi nimi je především podstatná anotace @OneToMany, jež definuje vztah mezi tabulkami. Ta totiž pomocí svých parametrů cascade a orphanRemoval definuje, jakým způsobem se při uložení či smazání dané entity bude zacházet s kolekcí entit, jež je touto anotací opatřena.

Mimo tyto anotace je pak tato třída pouze klasickým POJO, který kromě svých parametrů obsahuje getter a setter metody a metodu toString.

```
1   @Entity
2   @Table(name = "phone_book_contact")
3   public class PhoneBookContact extends AbstractEntity {
4
5     @Column(name="name")
6     private String name;
7
8     @ManyToOne
9     @JoinColumn(name="phone_book_id")
10    private PhoneBook phoneBook;
11
12    @OneToMany(cascade=CascadeType.ALL, mappedBy="contact", ↩
          orphanRemoval=true)
13    private Set<PhoneBookContactData> contactData = new HashSet<>();
14
15    ...
16  }
```

Výpis 6.1: Entity třída reprezentující kontakt telefonního seznamu

6.1.1.2 Repository

Po definici entit je nutné vytvořit repository rozhraní pro framework Spring Data, které bude určovat metody pro přístup k datům. Základní CRUD metody jsou definovány již v rozhraní CrudRepository tohoto frameworku, a proto jej všechna vytvořená rozhraní rozšiřují a případně doplňují o vlastní metody.

V případě, že je však potřeba nějakých koplikovanějších přístupů do databáze či definování speciálních podmínek, je potřeba definovat druhé rozhraní a specifikovat jeho implementaci. Aby bylo možné tyto metody používat pomocí základního rozhraní, stačí jej definovat jako potomka rozhraní druhého a Spring Data automaticky zajistí provázání implementace a základního rozhraní.

Pro představu je na výpisu 6.2 uveden příklad těchto rozhraní pro telefonní seznamy spolu s jejich implementací. Zde je potřeba filtrovat telefonní seznamy na základě jejich typu, popřípadě získat i takové, které typ nemají. Přístup zvenčí je zde zajištěn pomocí rozhraní PhoneBookRepository, které rozšiřuje

nejen CrudRepository, ale také PhoneBookQDRepository. To je implementováno třídou PhoneBookRepositoryImpl, jež s pomocí Spring Data podpory pro Querydsl získává z ORM Hibernate telefonní seznamy podle jejich typu.

```
1  public interface PhoneBookQDRepository {
2    List<PhoneBook> findAllByTypeIn(List<String> types);
3  }
4
5  public interface PhoneBookRepository extends ↩
       CrudRepository<PhoneBook, Long>, PhoneBookQDRepository {
6  }
7
8  public class PhoneBookRepositoryImpl extends ↩
       QueryDslRepositorySupport implements PhoneBookQDRepository {
9    @Override
10   public List<PhoneBook> findAllByTypeIn(List<String> types) {
11     QPhoneBook phoneBook = new QPhoneBook("myPhoneBook");
12     QPhoneBookType type = new QPhoneBookType("myType");
13
14     BooleanExpression whereExpr = type.name.in(types);
15     if(types.contains("")) {
16       types.remove("");
17       whereExpr = whereExpr.or(phoneBook.type.isNull());
18     }
19
20     return from(phoneBook)
21           .leftJoin(phoneBook.type, type)
22           .where(whereExpr)
23           .list(phoneBook);
24   }
25 }
```

Výpis 6.2: Načítání telefonních seznamů z databáze

6.1.1.3 Service

Service třídy staví na rozhraních repository a zajišťují překlad mezi databázovými entitami a TO, které používají controllery. Při tvorbě service tříd bylo tedy potřeba tento překlad implementovat, zajistit validaci dat a na každý závažný prohřešek reagovat výjimkou.

Service třídy využívají @Autowired anotací frameworku Spring pro zajištění DI, takže nebylo potřeba nijak složitě implementovat či konfigurovat připojení repository objektů do instancí těchto tříd. Dále jsou některé z metod service tříd opatřeny anotací @Transactional (také ze Springu), která zajišťuje provedení celé metody v jedné databázové transakci a návrat databáze do původního stavu v případě, že se v průběhu zpracování metody objeví problém.

6.1.1.4 Controller

Výpis 6.3 ukazuje příklad controlleru a jeho metody. Opět je zde využito anotací frameworku Spring, které zajišťují definici RESTful rozhraní pro Spring

Dispatcher Servlet. Anotace @RestController zajišťuje použití Spring MVC pro REST, @RequestMapping pak definuje URI, metodu a další parametry požadavku, který daný controller a metoda obsluhuje.

Implementace controllerů přímo vychází ze specifikace API, jsou zde vraceny TO a nastaveny úpěšné HTTP status kódy (výchozí hodnota je 200, ale může být změněna pomocí anotace @ResponseStatus). Nastavení chybových status kódů uvedených ve specifikaci popisuje následující odstavec.

```
1  @RestController
2  @RequestMapping(value="/phonebooks/configs", ↩
     produces={"application/json;␣charset=UTF-8"})
3  public class PhoneBookConfigsController {
4
5    @Autowired
6    private PhoneBookConfigsService phoneBookConfigsService;
7
8    @RequestMapping(value="/{phoneBookId}", method=RequestMethod.PUT)
9    public PhoneBookConfigurationTO updatePhoneBookConfigurations( ↩
         @RequestBody PhoneBookConfigurationTO configTO,
10       @PathVariable Long phoneBookId) {
11     return phoneBookConfigsService ↩
           .updatePhoneBookConfigs(phoneBookId, configTO);
12   }
13
14   ...
15 }
```

Výpis 6.3: Controller pro konfigurace telefonních seznamů

6.1.1.5 Zpracování výjimek

Jak již bylo řečeno, service třídy posílají controllerům výjimky v reakci na chybné požadavky, ale controllery se o ně nestarají. Děje se tak díky tomu, že framework Spring umožňuje tvorbu vlastních globálních obsluh výjimek. Ty jsou připojeny k Dispatcher Servletu a starají se o odchycení a patřičné zpracování výjimek, které přijdou z controllerů.

Pro účely této práce byla tedy vytvořena jedna globální obsluha, jejíž část je zobrazena na výpise 6.4. Anotace u definice třídy zajišťují připojení obsluhy ke všem controllerům v aplikaci a specifikaci pro Dispatcher Servlet, že všechny metody v této třídě vrací přímo tělo HTTP odpovědi a nikoli specifikaci view.

Metody této třídy vrací v souladu se specifikací ErrorTO, jenž má být vracen v případě jakéhokoli problému. Anotace @ExceptionHandler specifikuje, jakou výjimku daná metoda obsluhuje. Druhá anotace pak zajišťuje nastavení správného HTTP status kódu, stejně jako tomu bylo u controllerů. Tato třída je pro aplikaci zásadní, protože výrazně ulehčuje vývoj controllerů a zabraňuje opakování kódu pro ošetřování stejných výjimek napříč aplikací.

```
@ControllerAdvice
@ResponseBody
public class GlobalControllerExceptionHandler {

  @ResponseStatus(NOT_FOUND)
  @ExceptionHandler(NoEntityException.class)
  public ErrorTO handleNotFound(NoEntityException ex) {
    return return new ErrorTO(ex.getErrorCause(), ex.getMessage());
  }

  ...
}
```

Výpis 6.4: Globální obsluha výjimek

6.1.1.6 Import kontaktů z CSV

Pro import kontaktů z libovolného zdroje podle typu telefonního seznamu byla implementována třída PhoneBookSynchronizationServiceImpl. Tato třída využívá pro získání dat PhoneBookProvider podle typu daného telefonního seznamu. V případě CSV je použit CsvProvider, který načítá data ze souboru pomocí knihovny implementované v jednom z dalších firemních projektů. Získaná data jsou pak zkonvertována na TO, se kterými si již umí poradit service třída obsluhující kontakty. Pro třídu PhoneBookSynchronizationServiceImpl je opět podstatná anotace @Transactional, která zajišťuje, že staré kontakty nebudou smazány, pokud se během nahrávání kontaktů nových vyskytne nějaký problém.

Import kontaktů probíhá asynchronně vůči požadavku, aby klient nebyl zdržován zdlouhavým zpracováváním většího objemu dat. Service tedy pouze zkontroluje platnost požadavku, při úspěšné kontrole naplánuje synchronizaci a volání metody ukončí. Následně controller, který metodu této service třídy zavolal, vrátí klientovi HTTP status kód 202 a o další průběh nahrávání se nestará.

6.1.2 Klientská část

Implementace klientské části se dá opět rozdělit na dílčí části, které jsou spolu s dalšími zásadními funkcemi popsány v následujících odstavcích.

6.1.2.1 Resource

Podobně, jako je na serveru potřeba začít přístupem k databázi, je zde potřeba začít u resource rozhraní, které slouží pro připojení k RESTful API serveru. Definice resource rozhraní pro telefonní seznamy je uvedena ve výpise 6.5. Podoba tohoto rozhraní vychází z API specifikace serveru a jeho podstatou jsou JAX-RS anotace. Framework Apache CXF na základě těchto anotací vytváří

proxy resource objekty, které zajišťují připojení na API serveru a vyhodnocování HTTP odpovědí.

```
1  @Produces(MediaType.APPLICATION_JSON)
2  public interface PhoneBooksResource {
3
4    @POST
5    @Path(value = "/phonebooks")
6    @Consumes(MediaType.APPLICATION_JSON)
7    PhoneBookTO createPhoneBook(@HeaderParam(PhoneBookTO phoneBook);
8
9    ...
10 }
```

Výpis 6.5: Rozhraní pro připojení ke zdroji telefonních seznamů

6.1.2.2 MVP

Implementace ostatních částí administračního rozhraní probíhala prakticky souběžně. Všechny komponenty MVP využívají prvků, které již byly v rámci firmy vytvořeny a stavějí na frameworku Vaadin.

Model využívá service třídu pro zajištění tvorby a komunikace s proxy resource rozhraním. V obou třídách tedy bylo potřeba implementovat základní CRUD metody. Model je také zodpovědný za překlad mezi TO a objekty používanými komponentami frameworku Vaadin, o což se ale ve většině případů stará již existující kód. Pouze pro kontakty bylo nutné tuto funkcionalitu implementovat znovu, protože se struktura jejich TO příliš neslučuje s existující generickou implementací.

Jelikož je většina tabulek v administračním rozhraní založena na podobném principu, je většina view a presenter funkcionalit již vytvořena ve stávajícím projektu, proto byly přepoužity. Bylo však nutné implementovat některé nové prvky, které tyto komponenty využívají. Především bylo nutné zavést možnost zobrazení tabulky kontaktů po stisku tlačítka v tabulce telefonních seznamů. Dále bylo potřeba upravit některé formuláře pro editaci záznamů, případně vytvořit nové FormFactory třídy pro tvorbu těchto formulářů. Poslední větší změnou v této části pak byla úprava menu administračního rozhraní v závislosti na obsahu tabulky telefonních seznamů, aby bylo možné mezi tabulkami kontaktů přecházet i přímo pomocí menu. Kvůli této poslední změně však bylo nutné provést refaktoring menu ve stávajícím projektu, což přibližuje následující odstavec.

6.1.2.3 Menu aplikace

Menu v administračním rozhraní je koncipováno tak, že po uživatelské interakci je daná položka zvýrazněna, dokud uživatel neklikne na položku další. To trochu znesnadňuje orientaci v případě, kdy uživatel používá tlačítka webového prohlížeče pro posun vpřed či zpět, nebo pokud se dostane na stránku,

která není reprezentována žádnou položkou v menu. Toto menu bylo ve stávajícím projektu tvořeno pouze generickou třídou typu Component, a proto jej nebylo možné programově za běhu aplikace upravovat.

Pro potřeby telefonních seznamů se však ukázalo, že by bylo potřeba buď automaticky měnit obsah menu a nebo mít možnost změnit, která z položek je zvýrazněna. Proto bylo vytvořeno a implementováno rozhraní Menu zobrazené na výpise 6.6. Toto rozhraní definuje nejen přidávání a odebírání prvků menu, ale také obsahuje metodu pro zvýraznění konkrétní položky. Tím se umožnilo přidání jedotlivých telefonních seznamů do menu, odkud se nyní může administrátor dostat k seznamu jejich kontaktů rychleji. Zvýšila se i orientace v aplikaci, protože je vždy zvýrazněna aktuálně navštívená položka.

```
1   public interface Menu extends Component {
2
3         boolean clickItem(String path);
4
5         void addItem(String parentPath, String path, String name);
6
7         void removeChildItems(String parentPath);
8
9   }
```

Výpis 6.6: Rozhraní pro programově modifikovatelné menu

6.1.2.4 Zpracování chybových zpráv serveru

V případě, že server zareaguje na požadavek administračního rozhraní odpovědí s chybovým HTTP status kódem, postará se o tuto odpověď sám framework Apache CXF odesláním výjimky typu některého z potomků třídy WebApplicationException. Tato výjimka však obsahuje pouze generickou zprávu o chybě, která není pro potřeby administračního rozhraní dostačující. Výhodou však je to, že tato výjimka obsahuje celou HTTP odpověď, ze které je možné vyčíst ErrorTO odeslané serverem.

Tuto výjimku je potřeba odchytávat při každém volání metod resource tříd a mapovat ji na interní výjimku administračního rozhraní. Aby se zamezilo opakování stejného kódu napříč aplikací, byla vytvořena metoda ve třídě ServiceHelper, která má tuto funkcionalitu na starosti. V případě, že dostane výjimku daného typu, získá z ní ErrorTO a použije jej pro vytvoření interní ServiceException. Zároveň se však může stát, že výjimka ErrorTO vůbec neobsahuje (například v případě požadavku na špatnou adresu) a nebo došlo ještě před odesláním požadavku v resource třídě k problému úplně jiného typu. Takové případy jsou v mapovací metodě samozřejmě ošetřeny a jejich výsledkem je ServiceException s vlastní generickou zprávou indikující neznámý problém se serverem. Zde popsanou metodu zobrazuje výpis 6.7.

Odchytávání výjimek a volání metody třídy ServiceHelper je nyní stále potřeba provádět při každém volání metod resource tříd, protože nebyl nalezen

jednoduchý způsob, jak to zajistit na jednom místě. Je však pozitivní, že se podařilo většinu funkcionality převést do jedné třídy. V budoucím vývoji aplikace by stálo za úvahu, zda nevyužít například AOP (Aspected Oriented Programming) pro odchytávání výjimek. Jistě by to zabránilo opakujícímu se kódu, avšak mohlo by to zhoršit jeho čitelnost pro neinformované vývojáře.

```
1  public static ServiceException mapServiceException(Exception e) {
2    if (e instanceof WebApplicationException) {
3      WebApplicationException wea = (WebApplicationException) e;
4      try {
5        ErrorTO error = wea.getResponse().readEntity(ErrorTO.class);
6        return new ServiceException(error.getErrorCause().toString(), ↩
             error.getErrorCause().getCode());
7      } catch (ProcessingException ex) {
8        LOG.warn("Server␣returned␣error␣response␣without␣ErrorTO");
9      }
10   }
11   return new ServiceException("SERVER_UNAVAILABLE", e);
12 }
```

Výpis 6.7: Zpracování chybových zpráv serveru

6.2 Testování

Poslední sekce praktické části se zabývá způsobem, jakým byly implementované funkce otestovány. Kompletní zdrojové kódy testů je opět možné najít na přiloženém médiu.

6.2.1 Serverová část

Serverová část aplikace byla pokryta automatickými integračními testy, a to jak controllerů, tak i service tříd. Pro účely testování byla vytvořena komponenta PhoneBooksTestHelper, která je využívána téměř všemi testovacími třídami a zajišťuje tvorbu testovacích dat v databázi, tvorbu testovacích TO a kontrolu dat, která jsou v databázi po provedení testované činnosti.

6.2.1.1 Controller

Všechny základní CRUD funkce serverové části byly testovány za pomoci frameworku Spring. Ten poskytuje třídu MockMvc, která slouží jako vstupní bod pro testování Spring MVC aplikací. V každém testu je tedy na této třídě volána některá z HTTP metod s daným URI a parametry a následně je kontrolováno, zda vrácená odpověď odpovídá očekáváním.

Výpis 6.8 ukazuje tento způsob testování na metodě pro úpravu telefonního seznamu příslušného controlleru. Jak je možné vidět, test na třídě MockMvc volá metodu PUT s danými parametry a pak kontroluje HTTP status kód a obsah odpovědi. Také je zde využit zmíněný PhoneBooksTestHelper, který

zajišťuje počáteční inicializaci databáze, vytvoření TO pro úpravu seznamu a nakonec kontrolu databáze po provedení metody PUT.

```
1  @Test
2  public void updatePhoneBookTest() throws Exception {
3    helper.createDummyPhoneBook("");
4
5    PhoneBookTO phoneBookTO = helper.createDummyPhoneBookTO("2");
6
7    mockMvc.perform(put("/phonebooks/" + DEFAULT_ID) ↩
         .contentType(contentType).content(json(phoneBookTO)))
8           .andExpect(status().isOk())
9           .andExpect(content().contentType(contentType))
10          .andExpect(jsonPath("$.id", is(1)))
11          .andExpect(jsonPath("$.name", is(NAME + 2)))
12          .andExpect(jsonPath("$.description", is(DESC + 2)))
13          .andExpect(jsonPath("$.global", is(true)))
14          .andExpect(jsonPath("$.type", ↩
                is(PhoneBookSourceType.CSV.name())))
15   ;
16
17   helper.checkDummyPhoneBook("2");
18 }
```

Výpis 6.8: Testování controlleru pro telefonní seznamy

6.2.1.2 Service

Funkce otestované při testování controllerů již na straně service tříd testovány nebyly. Místo toho zde bylo kontrolováno, zda příslušné metody reagují na chybové vstupy a také zda se správně pracuje s typovými tabulkami v databázi.

Výpis 6.9 zobrazuje jeden z těchto testů pro typ dat kontaktu. Na začátku se předpokládá, že je databáze prázdná. Následně je vždy při žádosti o nový typ testováno, že se databáze rozrostla, a při žádosti o již přidaný typ se kontroluje, že do databáze nebyl přidán podruhé.

```
1  @Test
2  public void fetchDataTypeTest() throws Exception {
3    helper.checkDataTypes(0);
4    service.fetchDataType(SingleValueType.SURNAME.name());
5    helper.checkDataTypes(1);
6    service.fetchDataType(MultiValueType.ADDRESSBOOK_CELL.name());
7    helper.checkDataTypes(2);
8    service.fetchDataType(MultiValueType.ADDRESSBOOK_CELL.name());
9    helper.checkDataTypes(2);
10 }
```

Výpis 6.9: Testování perzistence typů dat kontaktu

6.2.1.3 Zpracování výjimek

Protože bylo jedním z důležitých kroků implementace také globální ošetření výjimek, bylo jej samozřejmě potřeba také otestovat. Přímé jednotkové tes-

tování však nebylo možné, jelikož by neprokázalo, že jsou skutečně správně vytvářeny HTTP odpovědi. Proto byl pouze za tímto účel vytvořen ExceptionThrowingController, který na všechny své definované metody odpovídá vyhozením některé ze sledovaných výjimek.

Samotné testování pak probíhalo stejně, jako u každého jiného controlleru. Rozdíl zde byl pouze v tom, že se testovala přítomnost chybového kódu v odpovědi a obsah přeneseného ErrorTO. Otestování této funkce pomocí jediného controlleru také přineslo jednu další výhodu, a to usnadnění testování ostatních controllerů, u kterých již není nutné chybové status kódy kontrolovat, protože je díky frameworku Spring jasné, že se tato funkce napříč celou aplikací nezmění.

```
1  public class ExceptionThrowingController {
2
3    @RequestMapping(value = "/404", method = RequestMethod.GET)
4    public void getNotFound() {
5      throw new NoEntityException("Some␣really␣important␣message");
6    }
7
8    ...
9  }
10
11 public class GlobalControllerExceptionHandlerIT extends ↩
       RESTTestCase {
12
13   @Test
14   public void handleNotFoundTest() throws Exception {
15     mockMvc.perform(get("/exceptions/404"))
16            .andExpect(status().isNotFound())
17            .andExpect(content().contentType(contentType))
18            .andExpect(jsonPath("$.errorCause", ↩
                  is(RECORD_NOT_FOUND.name())))
19            .andExpect(jsonPath("$.errorMessage", is("Some␣really␣↩
                  important␣message")))
20     ;
21   }
22 }
```

Výpis 6.10: Testování ošetření výjimek na serveru

6.2.2 Klientská část

Tato část aplikace nebyla testována jinak, než pouhým jejím ručním procházením. Jelikož se jedná prakticky o tenkého klienta, není to až takový problém, protože většina funkcí byla otestována na serveru. Do budoucna se však počítá s použitím automatických testů, jež by simulovaly práci uživatele v aplikaci.

Závěr

Hlavními cíli této práce byla analýza a návrh serverové aplikace pro konfiguraci drážních dispečerských terminálů a následná implementace a otestování jejích vybraných funkcionalit. Součástí práce bylo rovněž objasnění teorie v této problematice a přiblížení projektů, na které tato aplikace navazuje.

Po úvodní části a stanovení cílů byla vysvětlena teorie, jež se týká procesu vývoje software. Byly zde popsány potřebné kroky analytické a návrhové části vývoje, včetně konkrétních příkladů modelů a architektonických vzorů.

Další kapitola ve zkratce objasnila firemní prostředí, kde je aplikace vyvíjena, a projekty, se kterými by měla spolupracovat či je rozvíjet.

Následovala analýza procesů, které by měla výsledná aplikace podporovat, spolu se sběrem funkčních i nefunkčních požadavků. Na jejich základě byly zjištěny oblasti požadovaných funkcí a vytvořeny modely případů užití a problémové domény. Oblasti funkcí, které byly objeveny, se týkají správy organizační struktury terminálů, konfigurací, verzí FW a telefonních seznamů.

Návrhová část navázala na analýzu. Byly zde představeny použité technologie a architektura systému. V této kapitole byl také navržen model databáze a podoba rozhraní, pomocí kterého bude k aplikaci přistupováno.

Nakonec byly implementovány vybrané funkce týkající se telefonních seznamů. Jmenovitě se jedná o vytváření, zobrazení, úpravu i mazání samotných telefonních seznamů a jejich kontaktů. Další z implementovaných funkcí je pak možnost konfigurace externího zdroje pro tyto seznamy a konečně samotný import telefonních seznamů ze souborů ve formátu CSV. Tyto funkce byly úspěšně otestovány a bylo tedy ověřeno, že bezpečně fungují.

Všechny cíle stanovené v úvodu práce byly tedy splněny. Analytická i návrhová část byly kompletně dokončeny a na jejich základě je tedy možné pokračovat ve vývoji. Předmětem práce pak byla implementace pouze vybraných funkcionalit, a proto je jasné, jakým směrem se bude další vývoj aplikace ubírat. Bude potřeba implementovat zbylé funkcionality, které se týkají distribuce konfigurací a aplikačního firmware. Do budoucna se také plánuje zavedení automatických testů administračního rozhraní aplikace. Samozřejmě také

bude potřeba upravit terminálové aplikace, aby se dokázaly připojit k API vytvořeného serveru. Zároveň je pravděpodobné, že bude tato práce využívána pro pochopení technologií a celkového návrhu serverové aplikace v případě potřeby dalších funkcí.

Literatura

[1] Arlow, J.; Neustadt, I.: *UML 2 a unifikovaný proces vývoje aplikací*. Computer Press, a.s., první vydání, 2007, ISBN 978-80-251-1503-9.

[2] Kanisová, H.; Müller, M.: *UML srozumitelně*. Computer Press, a.s., druhé vydání, 2006, ISBN 80-251-1083-4.

[3] Mlejnek, J.: Modelování obchodních procesů [přednáška]. letní semestr 2018.

[4] Mlejnek, J.: Analýza problémové domény [přednáška]. letní semestr 2018.

[5] Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Addison Wesley Professional, třetí vydání, 2004, ISBN 0-13-148906-2.

[6] Špaček, P.: Architecture and Design Patterns [přednáška]. zimní semestr 2017.

[7] Bernson, A.: *Client/Server Architecture*. McGraw-Hill, druhé vydání, 1996, ISBN 0-07-005664-1.

[8] Ramirez, A. O.: Three-Tier Architecture [online]. *Linux Journal*, 2000, [cit. 2018-12-10]. Dostupné z: `http://www.linuxjournal.com/article/3508`

[9] Borek, B.: Úvod do architektury MVC [online]. *Zdroják.cz*, 2009, [cit. 2018-12-15]. Dostupné z: `https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/`

[10] Borek, B.: Prezentační vzory z rodiny MVC [online]. *Zdroják.cz*, 2009, [cit. 2018-12-15]. Dostupné z: `https://www.zdrojak.cz/clanky/prezentacni-vzory-zrodiny-mvc/`

[11] Fielding, R.: *Architectural Styles and the Design of Network-based Software Architectures*. Disertační práce, University of California, Irvine, 2000. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[12] Malý, M.: REST: architektura pro webové API [online]. *Zdroják.cz*, 2009, [cit. 2018-12-30]. Dostupné z: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

[13] Richardson, L.: Act Three: The Maturity Heuristic [přednáška]. 2008. Dostupné z: https://www.crummy.com/writing/speaking/2008-QCon/act3.html

[14] Pokorný, J.: *Databázové systémy 2 [skripta]*. Nakladatelství ČVUT, první vydání, 2007, ISBN 978-80-01-03797-3.

[15] Mlejnek, J.: Návrhové třídy a přiřazení zodpovědností [přednáška]. letní semestr 2018.

[16] Perry, S.: Java language basics [online]. 2010, [cit. 2018-12-20]. Dostupné z: https://www.ibm.com/developerworks/java/tutorials/j-introtojava1/index.html

[17] Pivotal Software, Inc.: *Spring Framework [online]*. 2018, [cit. 2018-12-20]. Dostupné z: https://spring.io/projects/spring-framework

[18] Ecma International: *Standard ECMA-404: The JSON Data Interchange Format [online]*. 2013, [cit. 2018-12-20]. Dostupné z: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[19] The PostgreSQL Global Development Group: *About PostgreSQL [online]*. 2018, [cit. 2018-12-20]. Dostupné z: https://www.postgresql.org/about/

[20] Red Hat Developers: *Hibernate ORM: Your relational data. Objectively. [online]*. [cit. 2018-12-20]. Dostupné z: http://hibernate.org/orm/

[21] Pivotal Software, Inc.: *Spring Data [online]*. 2018, [cit. 2018-12-20]. Dostupné z: https://spring.io/projects/spring-data

[22] The Querydsl Team: *Querydsl [online]*. 2015, [cit. 2019-01-08]. Dostupné z: http://www.querydsl.com/

[23] Pivotal Software, Inc.: *Web on Servlet Stack [online]*. 2018, [cit. 2018-12-20]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

[24] Balani, N.; Hathi, R.: *Apache CXF Web Service Development*. Packt Publishing Ltd., první vydání, 2009, ISBN 978-1-847195-40-1.

[25] The Apache Software Foundation: *JAX-RS : Client API [online]*. [cit. 2018-12-28]. Dostupné z: `http://cxf.apache.org/docs/jax-rs-client-api.html`

[26] Vaadin Ltd.: *Vaadin Docs [online]*. 2018, [cit. 2018-12-28]. Dostupné z: `https://vaadin.com/docs`

Příloha **A**

Seznam použitých zkratek

AOP Aspected Oriented Programming

API Application Programming Interface

CCS Configuration Control & Statistics

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheet

CSV Comma-Separated Values

CXF Celtix and XFire

DI Dependency Injection

FW Firmware

GC Garbage Collector

GUI Graphical User Interface

HATEOAS Hypertext As The Engine Of Application State

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

IoC Inversion of Control

JAX-RS Java API for RESTful Web Services

JDK Java Development Kit

JPA Java Persistence API

JSON JavaScript Object Notation

JVM Java Virtual Machine

MVC Model–View-Controller

MVP Model-View-Presenter

ORM Object-relational Mapping

POJO Plain Old Java Object

REST REpresentational State Transfer

SOAP Simple Object Access Protocol

SSZT Správa Sdělovací a Zabezpečovací Techniky

SŽDC Správa Železniční Dopravní Cesty

TO Transfer Object

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

USDP Unified Software Development Process

XML eXtensible Markup Language

Příloha B

RESTful API specifikace

Lokality

Models

LocationTO

- **id** *(Int)* - Id of location *(not in POST request body)*
- **name** *(String)* - Name of location
- **level** *(Int)* - Level of location
- **parent** *(LocationTO)* - Parent location

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST `/locations`

Create a new location.

Request parameters

Body

- **location** *(LocationTO)* - Location to be created

Response

LocationTO

- **201** - New location was successfully created.

ErrorTO

- **400** - Request body is missing required values.
- **400** - Location has lower level than its parent.

GET `/locations`

Return a list of locations, that are on given level. Search for multiple levels is allowed. If no level is provided, all locations are returned.

Request parameters

Query

- **level** *(Int)* - Level of desired locations *(optional)*

Response

Array [LocationTO]

- **200** - List of locations was successfully returned.

PUT `/locations/{id}`

Update a location by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired location *(required)*

Body

- **LocationTO** *(LocationTO)* - Location to be updated

Response

LocationTO

- **200** - Location was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **400** - Location has lower level than its parent.
- **404** - Location was not found.

DELETE `/locations/{id}`

Delete a location by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired location *(required)*

Response

Empty

- **204** - Location was successfully deleted.

ErrorTO

- **404** - Location was not found.

Firmware

Models

FirmwareTO

- **id** *(Int)* - Id of firmware *(not in POST request body)*
- **versionLabel** *(String)* - Label of firmware version
- **archivePath** *(String)* - System path to archive with firmware

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST `/firmware`

Create a new firmware.

Request parameters

Body

- **firmware** *(FirmwareTO)* - Firmware to be created

Response

FirmwareTO

- **201** - New firmware was successfully created.

ErrorTO

- **400** - Request body is missing required values.
- **400** - Provided data are not unique.

GET `/firmware`

Return a list of firmware.

Request parameters

None

Response

Array [FirmwareTO]

- **200** - List of firmware was successfully returned.

PUT `/firmware/{id}`

Update a firmware by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired firmware *(required)*

Body

- **firmware** *(FirmwareTO)* - Firmware to be updated

Response

FirmwareTO

- **200** - Firmware was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **400** - Provided data are not unique.
- **404** - Firmware was not found.

DELETE `/firmware/{id}`

Delete a firmware by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired firmware *(required)*

Response

Empty

- **204** - Firmware was successfully deleted.

ErrorTO

- **404** - Firmware was not found.

Terminály

Models

TerminalTO

- **id** *(Int)* - Id of terminal *(not in POST request body)*
- **name** *(String)* - Name of terminal
- **hostname** *(String)* - Unique name of terminal in the network
- **ipAddress** *(String)* - IP address of the terminal
- **location** *(LocationTO)* - Location, where the terminal is assigned
- **runningProfile** *(ProfileTO)* - Profile currently running on terminal
- **lastSyncDate** *(String)* - Last date of terminal synchronization
- **lastSyncState** *(Boolean)* - Last state of terminal synchronization (OK or not)
- **upgradeState** *(Int)* - State of terminal firmware upgrade (percent)
- **installedFWVersion** *(String)* - Version of firmware installed on terminal
- **wantedFWVersion** *(String)* - Version of firmware wanted on terminal

LocationTO

- **id** *(Int)* - Id of location *(not in POST request body)*
- **name** *(String)* - Name of location
- **level** *(Int)* - Level of location
- **parent** *(LocationTO)* - Parent location

ProfileTO

- **id** *(Int)* - Id of profile *(not in POST request body)*
- **name** *(String)* - Name of profile
- **location** *(LocationTO)* - Location, where the profile is assigned

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST `/terminals`

Create a new terminal.

Request parameters

Body

- **terminal** *(TerminalTO)* - Terminal to be created

Response

TerminalTO

- **201** - New terminal was successfully created.

ErrorTO

- **400** - Request body is missing required values.

GET `/terminals`

Return a list of terminals.

Request parameters

None

Response

Array [TerminalTO]

- **200** - List of terminals was successfully returned.

PUT `/terminals/{id}`

Update a terminal by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired terminal *(required)*

Body

- **terminal** *(TerminalTO)* - Terminal to be updated

Response

TerminalTO

- **200** - Terminal was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **404** - Terminal was not found.

DELETE `/terminals/{id}`

Delete a terminal by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired terminal *(required)*

Response

Empty

- **204** - Terminal was successfully deleted.

ErrorTO

- **404** - Terminal was not found.

GET /terminals/{id}/firmware

Start transfer of the new firmware to given terminal if there is one wanted.

Request parameters

Path

- **id** *(Int)* - Id of terminal *(required)*

Response

Empty

- **202** - There is new firmware available and transfer successfully started.
- **204** - No new firmware is wanted for this terminal.

ErrorTO

- **404** - Terminal was not found.

GET /terminals/{id}/profiles

Return a list of profiles assigned to given terminal.

Request parameters

Path

- **id** *(Int)* - Id of terminal *(required)*

Response

Array [ProfileTO]

- **200** - List of profiles for terminal was successfully returned.

ErrorTO

- **404** - Terminal was not found.

PUT /terminals/{id}/profiles

Update list of assigned profiles for given terminal.

Request parameters

Path

- **id** *(Int)* - Id of terminal *(required)*

Body

- **profiles** *(Array [ProfileTO])* - Profiles to be assigned to terminal

Response

Array [ProfileTO]

- **200** - Assigned profiles for terminal were successfully updated.

ErrorTO

- **404** - Terminal was not found.

POST /terminals/{id}/profiles/{pid}

Log in to profile on given terminal.

Request parameters

Path

- **id** *(Int)* - Id of terminal *(required)*
- **pid** *(Int)* - Id of desired profile *(required)*

Response

Configuration file (CCS)

- **200** - Login completed successfully.

ErrorTO

- **403** - Profile is already running on another terminal.
- **404** - Terminal or profile was not found.

DELETE /terminals/{id}/profiles/{pid}

Log out of profile on given terminal.

Request parameters

Path

- **id** *(Int)* - Id of terminal *(required)*
- **pid** *(Int)* - Id of desired profile *(required)*

Response

Empty

- **204** - Logout completed successfully.

ErrorTO

- **404** - Terminal was not found or does not have given profile running on it.

Konfigurační profily

Models

ProfileTO

- **id** *(Int)* - Id of profile *(not in POST request body)*
- **name** *(String)* - Name of profile
- **location** *(LocationTO)* - Location, where the profile is assigned

LocationTO

- **id** *(Int)* - Id of location *(not in POST request body)*
- **name** *(String)* - Name of location
- **level** *(Int)* - Level of location
- **parent** *(LocationTO)* - Parent location

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

TerminalTO

- **id** *(Int)* - Id of terminal *(not in POST request body)*
- **name** *(String)* - Name of terminal
- **hostname** *(String)* - Unique name of terminal in the network
- **ipAddress** *(String)* - IP address of the terminal
- **location** *(LocationTO)* - Location, where the terminal is assigned
- **runningProfile** *(ProfileTO)* - Profile currently running on terminal
- **lastSyncDate** *(String)* - Last date of terminal synchronization
- **lastSyncState** *(Boolean)* - Last state of terminal synchronization (OK or not)
- **upgradeState** *(Int)* - State of terminal firmware upgrade (percent)
- **installedFWVersion** *(String)* - Version of firmware installed on terminal
- **wantedFWVersion** *(String)* - Version of firmware wanted on terminal

POST `/profiles`

Create a new profile.

Request parameters

Body

- **profile** *(ProfileTO)* - Profile to be created

Response

ProfileTO

- **201** - New profile was successfully created.

ErrorTO

- **400** - Request body is missing required values.

GET `/profiles`

Return a list of profiles.

Request parameters

None

Response

Array [ProfileTO]

- **200** - List of profiles was successfully returned.

PUT /profiles/{id}

Update a profile by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired profile *(required)*

Body

- **profile** *(ProfileTO)* - Profile to be updated

Response

ProfileTO

- **200** - Profile was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **404** - Profile was not found.

DELETE /profiles/{id}

Delete a profile by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired profile *(required)*

Response

Empty

- **204** - Profile was successfully deleted.

ErrorTO

- **404** - Profile was not found.

GET /profiles/{id} /terminals

Return a list of terminals where is given profile assigned.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*

Response

Array [TerminalTO]

- **200** - List of terminals for profile was successfully returned.

ErrorTO

- **404** - Profile was not found.

Verze profilů

Models

ProfileVersionTO

- **id** *(Int)* - Id of profile version *(not in POST request body)*
- **versionLabel** *(String)* - Label of profile version
- **filePath** *(String)* - System path to configuration file

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST /profiles/{id}/versions

Create a new version of profile.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*

Body

- **profileVersion** *(ProfileVersionTO)* - Profile version to be created

Response

ProfileVersionTO

- **201** - New version of profile was successfully created.

ErrorTO

- **400** - Request body is missing required values.
- **404** - Profile was not found.

GET /profiles/{id}/versions

Return a list of versions for given profile.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*

Response

Array [ProfileVersionTO]

- **200** - List of version for profile was successfully returned.

ErrorTO

- **404** - Profile was not found.

PUT /profiles/{id}/versions/{vid}

Update a version of profile by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*
- **vid** *(Int)* - Id of desired profile version *(required)*

Body

- **profileVersion** *(ProfileVersionTO)* - Profile version to be updated

Response

ProfileVersionTO

- **200** - Version of profile was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **403** - Profile version is being updated by other user.
- **404** - Profile was not found or does not have desired version.

DELETE /profiles/{id}/versions/{vid}

Delete a version of profile by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*
- **vid** *(Int)* - Id of desired profile version *(required)*

Response

Empty

- **204** - Version of profile was successfully deleted.

ErrorTO

- **403** - Profile version is being updated by other user.
- **404** - Profile was not found or does not have desired version.

GET /profiles/{id}/versions/{vid}/firmware

Return a list of firmware versions, where is given profile version executable.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*
- **vid** *(Int)* - Id of profile version *(required)*

Response

Array [FirmwareTO]

- **200** - List of firmware for profile version was successfully returned.

ErrorTO

- **404** - Profile was not found or does not have desired version.

PUT /profiles/{id}/versions/{vid}/firmware

Update a list of firmware versions, where is given profile version executable.

Request parameters

Path

- **id** *(Int)* - Id of profile *(required)*
- **vid** *(Int)* - Id of profile version *(required)*

Body

- **firmware** *(Array [FirmwareTO])* - Firmware versions, where profile version is executable

Response

Array [FirmwareTO]

- **200** - Assigned firmware for profile version was successfully updated.

ErrorTO

- **400** - List of firmware versions was empty.
- **404** - Profile was not found or does not have desired version.

Ostatní servery

Models

ServerTO

- **id** *(Int)* - Id of server *(not in POST request body)*
- **name** *(String)* - Name of server
- **description** *(String)* - Description of server
- **link** *(String)* - URL address of the server

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST /servers

Create a new server.

Request parameters

Body

- **server** *(ServerTO)* - Server to be created

Response

ServerTO

- **201** - New server was successfully created.

ErrorTO

- **400** - Request body is missing required values.
- **400** - Server address was not unique.

GET /servers

Return a list of servers.

Request parameters

None

Response

Array [ServerTO]

- **200** - List of servers was successfully returned.

PUT /servers/{id}

Update a server by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired server *(required)*

Body

- **server** *(ServerTO)* - Server to be updated

Response

ServerTO

- **200** - Server was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **400** - Server address was not unique.
- **404** - Server was not found.

DELETE /servers/{id}

Delete a server by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired server *(required)*

Response

Empty

- **204** - Server was successfully deleted.

ErrorTO

- **404** - Server was not found.

Telefonní seznamy

Models

PhoneBookTO

- **id** *(Int)* - Id of phone book *(not in POST request body)*
- **name** *(String)* - Name of the phone book
- **description** *(String)* - Description of the phone book
- **global** *(Boolean)* - Availability of the phone book
- **type** *(String)* - Type of the phone book

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST /phonebooks

Create a new phone book.

Request parameters

Body

- **phoneBook** *(PhoneBookTO)* - Phone book to be created

Response

PhoneBookTO

- **201** - New phone book was successfully created.

ErrorTO

- **400** - Request body is missing required values.

GET /phonebooks

Return a list of phone books, that are of given type. Search for multiple types is allowed. If no type is provided, all phone books are returned.

Request parameters

Query

- **type** *(String)* - Type of desired phone books *(optional)*

Response

Array [PhoneBookTO]

- **200** - List of phone books was successfully returned.

GET /phonebooks/{id}

Return a phone book by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired phone book *(required)*

Response

PhoneBookTO

- **200** - Phone book was successfully returned.

ErrorTO

- **404** - Phone book was not found.

PUT /phonebooks/{id}

Update a phone book by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired phone book *(required)*

Body

- **phoneBook** *(PhoneBookTO)* - Phone book to be updated

Response

PhoneBookTO

- **200** - Phone book was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **404** - Phone book was not found.

DELETE /phonebooks/{id}

Delete a phone book by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of desired phone book *(required)*

Response

Empty

- **204** - Phone book was successfully deleted.

ErrorTO

- **404** - Phone book was not found.

Kontakty

Models

PhoneBookContactTO

- **id** *(Int)* - Id of contact *(not in POST request body)*
- **singleValues** *(Array [SingleValue])* - Contact data of types, which should have only single value
- **multiValues** *(Array [MultiValue])* - Contact data of types, which can have multiple values

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

SingleValue

- **key** *(String)* - Type of value
- **value** *(String)* - Value for the type

MultiValue

- **key** *(String)* - Type of value
- **value** *(Array [String])* - Multiple values for the type

POST /phonebooks/{id}/contacts

Create a new contact in phone book.

Request parameters

Path

- **id** *(Int)* - Id of phone book to create contact in *(required)*

Body

- **contact** *(PhoneBookContactTO)* - Contact to be created

Response

PhoneBookContactTO

- **201** - New contact was successfully created.

ErrorTO

- **400** - Request body is missing required values.
- **404** - Phone book was not found.

GET /phonebooks/{id}/contacts

Return a list of contacts in phone book.

Request parameters

Path

- **id** *(Int)* - Id of phone book *(required)*

Response

Array [PhoneBookContactTO]

- **200** - List of contacts was successfully returned.
- **404** - Phone book was not found.

GET /phonebooks/{id}/contacts/{cid}

Return a contact by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of phone book *(required)*
- **cid** *(Int)* - Id of desired contact *(required)*

Response

PhoneBookContactTO

- **200** - Contact was successfully returned.

ErrorTO

- **404** - Contact was not found in given phone book.

PUT /phonebooks/{id}/contacts/{cid}

Update a contact by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of phone book *(required)*
- **cid** *(Int)* - Id of desired contact *(required)*

Body

- **contact** *(PhoneBookContactTO)* - Contact to be updated

Response

PhoneBookContactTO

- **200** - Contact was successfully updated.

ErrorTO

- **400** - Request body is missing required values.
- **404** - Contact was not found in given phone book.

DELETE /phonebooks/{id}/contacts/{cid}

Delete a contact by its unique id.

Request parameters

Path

- **id** *(Int)* - Id of phone book *(required)*
- **cid** *(Int)* - Id of desired contact *(required)*

Response

Empty

- **204** - Contact was successfully deleted.

ErrorTO

- **404** - Contact was not found in given phone book.

Konfigurace telefonních seznamů

Models

PhoneBookConfigurationTO

- **id** *(Int)* - Id of configured phone book *(not in POST request body)*
- **sourceConfiguration** *(Array [Configuration])* - Configurations of data source
- **contactMapping** *(PhoneBookContactTO)* - Mapping of contact fields from source to system

Configuration

- **key** *(String)* - Type of configuration
- **value** *(String)* - Configuration value

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

PhoneBookContactTO

- **singleValues** *(Array [SingleValue])* - Contact data of types, which should have only single value
- **multiValues** *(Array [MultiValue])* - Contact data of types, which can have multiple values

SingleValue

- **key** *(String)* - Type of value
- **value** *(String)* - Value for the type

MultiValue

- **key** *(String)* - Type of value
- **value** *(Array [String])* - Multiple values for the type

GET /phonebooks/configs

Return a list of configurations for all phone books, that are of given type. Search for multiple types is allowed. If no type is provided, configurations for all phone books are returned.

Request parameters

Query

- **type** *(String)* - Type of phone books to search for configurations *(optional)*

Response

Array [PhoneBookConfigurationTO]

- **200** - List of configurations was successfully returned.

GET /phonebooks/configs/{id}

Return configurations for phone book with given id.

Request parameters

Path

- **id** *(Int)* - Id of phone book *(required)*

Response

PhoneBookConfigurationTO

- **200** - Configurations were successfully returned.

ErrorTO

- **404** - Phone book was not found.

PUT /phonebooks/configs/{id}

Update configurations for phone book with given id.

Request parameters

Path

- **id** *(Int)* - Id of phone book *(required)*

Body

- **configurations** *(PhoneBookConfigurationTO)* - Configurations to be set to phone book

Response

PhoneBookConfigurationTO

- **200** - Configurations were successfully updated.

ErrorTO

- **400** - Given configurations did not match type of phone book or were incomplete.
- **404** - Phone book was not found.

Synchronizace telefonních seznamů

Models

ErrorTO

- **errorCause** *(String)* - Short description of error
- **errorMessage** *(String)* - Detailed error message

POST `/phonebooks/{id}/synchronization`

Import phone book contacts.

Request parameters

Path

- **id** *(Int)* - Id of desired phone book *(required)*

Response

Empty

- **202** - Import of contacts successfully started.

ErrorTO

- **404** - Phone book was not found.
- **404** - No configuration for source of contacts was found.

GET `/phonebooks/{id}/synchronization`

Export phone book contacts.

Request parameters

Path

- **id** *(Int)* - Id of desired phone book *(required)*

Response

File

- **200** - Contacts were successfully exported.

ErrorTO

- **404** - Phone book was not found.

Příloha C

Obsah přiloženého CD

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