České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Jakub Chalupa

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Implementace vyhledávače spojů veřejné dopravy nad grafovou databází Neo4j

Pokyny pro vypracování:

V rámci bakalářské práce "Využití grafových databází pro vyhledávání spojů veřejné dopravy" bylo navrženo a implementováno jádro algoritmu pro vyhledávání spojů nad grafovou databází Neo4j. Cílem této diplomové práce je rozšířit tento algoritmus o dosud neimplementované funkce (např. možnost vyhledání trasy "přes" další zastávky) a zejména pak implementace komplexní webové a mobilní aplikace pro vyhledávání spojů. Webová aplikace bude umožňovat mimo jiné import a export jízdných řádů ve formátu GTFS. Součástí práce bude analýza požadavků, návrh, implementace a nasazení webové aplikace do veřejného cloudu a vhodné srovnání s podobnými aplikacemi. Během implementace i po dokončení bude realizováno uživatelské a zátěžové testování obou aplikací nad reálnými daty veřejné dopravy.

Seznam odborné literatury:

[1] Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications.
[2] Sherif Sakr – Mohamed Gaber: Large Scale and Big Data: Processing and Management.
[3] Erich Gamma - Richard Helm - Ralph Johnson - John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software.

Vedoucí: Ing. Tomáš Černý, Ph.D.

Platnost zadání do konce letího semestra 2017/2018

L.S.

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 31.10.2016

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Implementace vyhledávače spojů veřejné dopravy nad grafovou databází Neo4j

Jakub Chalupa

Vedoucí práce: Ing. Tomáš Černý, MSc., Ph.D.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

10. května 2017

Poděkování

Rád bych tímto poděkoval Ing. Tomášovi Černému, MSc., Ph.D. za vedení práce a poskytnuté cenné rady. Dále děkuji doc. RNDr. Ireně Holubové, Ph.D. za poskytnuté konzultace, participantům, kteří se účastnili uživatelského testování, a všem mým blízkým, kteří mě po celou dobu studia podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Nové Bystřici dne 10. 5. 2017 ....

// Abstract

The aim of this work is an analysis, design, implementation, testing and deployment of both web and mobile applications allowing public transport connections searching over graph database Neo4j. For this purpose a detailed analysis of existing implementations of respective applications is processed and thereafter own solution is proposed. The main contribution of this work is ready to use web and mobile application allowing connections searching using actual data of public transport. Both applications can be instantly used by a wide range of users.

Abstrakt

Práce se zabývá analýzou, návrhem, implementací, testováním a nasazením webové a mobilní aplikace pro vyhledávání spojů veřejné dopravy nad grafovou databází Neo4j. Pro tyto účely práce nejprve detailně analyzuje existující implementace vyhledávačů a následně navrhuje vlastní řešení. Hlavním výstupem práce je finálně nasazená webová a mobilní aplikace, která umožňuje vyhledávání nad reálnými daty veřejné dopravy a která je dostupná širokému spektru uživatelů k okamžitému využití.

Obsah

1 Úvod **1**
1.1 Motivace . . . 2
1.2 Cíle . . . 2
1.3 Přínos . . . 2

2 Analýza **3**
2.1 Současný stav . . . 3
2.1.1 IDOS . . . 3
2.1.1.1 IDOS pro lokální počítače a sítě . . . 4
2.1.1.2 IDOS pro mobilní telefony . . . 4
2.1.1.3 IDOS pro internet . . . 4
2.1.1.4 IDOS pro mobilní zařízení . . . 5
2.1.2 Aplikace jednotlivých dopravců . . . 6
2.1.2.1 Student Agency / RegioJet . . . 7
2.1.2.2 České dráhy . . . 7
2.1.3 Další aplikace . . . 7
2.2 Parametry vyhledávače . . . 8
2.3 Standardizovaný formát dat o jízdních řádech . . . 9
2.3.1 Slovníček pojmů GTFS . . . 9
2.4 Uživatelské role . . . 10
2.4.1 Nepřihlášený uživatel . . . 10
2.4.2 Správce jízdního řádu . . . 10
2.4.3 Administrátor . . . 10
2.5 Funkční požadavky . . . 11
2.5.1 Požadavky z pohledu nepřihlášeného uživatele . . . 11
2.5.2 Požadavky z pohledu správce jízdního řádu . . . 12
2.5.3 Požadavky z pohledu administrátora . . . 15
2.6 Nefunkční požadavky . . . 15
2.6.1 Nefunkční požadavky na webovou aplikaci . . . 15
2.6.2 Nefunkční požadavky na mobilní aplikaci . . . 16

3 Analýza vyhledávání spojení v grafu **17**
3.1 Struktura grafu . . . 17
3.1.1 Time-dependent model . . . 17
3.1.2 Time-expanded model . . . 18

3.1.2.1 Fully time-expanded model . . . 19
3.1.3 Zvolená struktura jádra grafu . . . 19
3.1.4 Kompletní schéma grafu . . . 21
3.1.4.1 Atributy vrcholů grafu . . . 22
3.2 Vyhledávací algoritmus . . . 23
3.2.1 Nalezení výchozích zastavení . . . 24
3.2.2 Traverzování do cílové stanice . . . 25
3.2.2.1 Omezující podmínky . . . 26
3.2.2.2 Optimalizace vyhledávacího algoritmu . . . 27
3.2.2.3 Finální náhled na vyhledávací algoritmus . . . 27

4 Návrh **29**
4.1 Datové úložiště . . . 29
4.2 Rozdělení dat do schémat . . . 30
4.3 Datový model relační databáze . . . 31
4.4 Architektura . . . 33
4.4.1 REST . . . 33
4.4.2 Popis REST rozhraní . . . 34
4.4.2.1 Formát dat . . . 34
4.4.2.2 Zabezpečení . . . 35
4.4.2.3 Ukázky rozhraní . . . 36
4.5 Připojení k databázi Neo4j . . . 38
4.5.1 Serverové rozšíření databáze . . . 39
4.6 Zamýšlené nasazení . . . 40
4.7 Návrh vzhledu . . . 41

5 Implementace **45**
5.1 Zvolené technologie . . . 45
5.1.1 Java . . . 45
5.1.2 Spring . . . 45
5.1.3 Správa závislostí a sestavení aplikace . . . 46
5.1.4 Relační databáze PostgreSQL . . . 46
5.1.5 Hibernate . . . 46
5.1.6 Klientská aplikace . . . 46
5.1.6.1 Angular 2 . . . 47
5.1.7 Mobilní aplikace . . . 47
5.2 Detaily implementace . . . 48
5.2.1 Struktura aplikace . . . 48
5.2.2 REST API a Spring MVC . . . 49
5.2.3 Zabezpečení REST API . . . 51
5.2.4 Výběr konkrétního schématu databáze . . . 52
5.2.4.1 Výběr schématu relační databáze . . . 54
5.2.4.2 Připojení k požadované Neo4j databázi . . . 55
5.2.5 Vyhledávání spojů . . . 56
5.2.6 Angular 2 aplikace . . . 58
5.2.7 Android . . . 60

5.3 Ukázky UI . . . 61
5.4 Kód a statistiky . . . 63

6 Testování 65
6.1 Testování kódu . . . 65
6.2 Zátěžové a výkonnostní testování . . . 65
6.2.1 Testovací data . . . 65
6.2.2 Testovací prostředí . . . 66
6.2.3 Testování rychlosti importu a exportu jízdních řádů . . . 66
6.2.4 Testování rychlosti vyhledávání . . . 67
6.2.5 Zátěžové testování . . . 70
6.2.5.1 Konfigurace . . . 70
6.2.5.2 Průběh . . . 71
6.3 Uživatelské testování . . . 73
6.3.1 Výběr participantů . . . 73
6.3.1.1 1. participant . . . 73
6.3.1.2 2. participant . . . 73
6.3.1.3 3. participant . . . 73
6.3.2 Testovací prostředí . . . 74
6.3.3 Testovací scénáře . . . 74
6.3.4 Nálezy . . . 75
6.3.4.1 Komponenty pro výběr času (nízká priorita) . . . 75
6.3.4.2 Filtrování položek v administraci (vysoká priorita) . . . 75
6.3.4.3 Nutnost vkládání ID (nízká priorita) . . . 75
6.3.4.4 Zobrazování chybových hlášek (střední priorita) . . . 76
6.3.4.5 Vytváření a editace jízdy (vysoká priorita) . . . 76
6.3.4.6 Další nálezy (nízká priorita) . . . 76
6.3.5 Shrnutí uživatelského testování . . . 77

7 Nasazení 79
7.1 Nasazení webové aplikace . . . 79
7.1.1 Relační databáze . . . 79
7.1.2 Grafová databáze . . . 79
7.1.3 Aplikace . . . 80
7.1.4 Load balancer . . . 80
7.1.5 Doména a HTTPS . . . 80
7.2 Nasazení mobilní aplikace . . . 81
7.3 Výsledné schéma nasazení . . . 81

8 Závěr 83
8.1 Analýza . . . 83
8.2 Analýza vyhledávání spojů v grafu . . . 83
8.3 Návrh . . . 83
8.4 Implementace . . . 84
8.5 Testování . . . 84
8.6 Nasazení . . . 84

8.7 Zhodnocení výsledků a budoucí rozšíření 84

A Obsah přiloženého CD **91**

Seznam obrázků

3.1 Time-expanded (vlevo) a time-dependent model grafu se třemi stanicemi A, B, C. Tři jízdy spojují stanici A se stanicí B (jízdy u, v, w), jedna jízda spojuje stanici B a A (y) a dvě jízdy spojují stanici C a B (x, z). U time-expanded modelu si všimneme cyklického propojení vrcholů v rámci všech stanic. Zdroj: [29] 18
3.2 Zastavení v rámci jedné stanice. Čísla u vrcholů značí C_{pz} (vlevo) a C_{vz} (vpravo). Pro všechny C_z nyní platí $C_z = C_{vz}$. 20
3.3 Základní struktura grafu. Stejnou barvou jsou obarvena zastavení v rámci jízdy a jsou propojena jednosměrnými *přejezdovými* hranami. Pod sebou jsou zaznamenána zastavení v rámci jedné stanice a jsou propojena obousměrnými *přestupními* hranami. 21
3.4 Struktura grafu s přidanými vrcholy jízd (vrcholy J), intervalů platností (vrcholy I) a výjimek z intervalů platností (vrcholy V). Hrana znázorněná čárkovaně značí příslušnost zastavení k jízdě. Hrana znázorněná tečkovaně značí příslušnost jízdy k intervalu platnosti. Hrana znázorněná čerchovaně značí návaznost výjimky z intervalu platnosti na interval platnosti. 22

4.1 Schéma dat o jízdních řádech. Každý jízdní řád bude uložen v samostatném schématu. 32
4.2 Schéma dat o uživatelích. 33
4.3 Model očekávaného nasazení 40
4.4 Vyhledávací formulář. 42
4.5 Výsledky vyhledávání. 42
4.6 Seznam a vyhledávání stanic. 43
4.7 Detail a editace stanice. 43

5.1 Struktura aplikace - rozdělení do modulů. 48
5.2 Architektura a vrstvy aplikace. Zdroj: [19] 48
5.3 Vyhledávací formulář webové aplikace. 62
5.4 Vyhledávací obrazovka mobilní aplikace (Android). 62

7.1 Model skutečného nasazení. 81

Seznam tabulek

4.1 Definice vytvoření stanice. 36
4.2 Definice získání stanice dle ID. 37
4.3 Definice smazání stanice dle ID. 37
4.4 Definice updatu stanice dle ID. 37

6.1 Rychlost importu a exportu jízdního řádu o přibližné velikosti 2 400 000 řádků dat. 67
6.2 Rychlost vyhledávání s maximálně jedním přestupem. 68
6.3 Rychlost vyhledávání s maximálně třemi přestupy. 68
6.4 Rychlost vyhledávání s maximálně čtyřmi přestupy. 69
6.5 Rychlost vyhledávání s maximálně třemi přestupy a zvolenou stanicí *přes*. . . 69
6.6 Průměrná a maximální doba odpovědi serveru při přístupu na hlavní stránku aplikace. 71
6.7 Vyhledávací trasy pro první zátěžový test vyhledávání. 71
6.8 Průměrná a maximální doba odpovědi ze serveru pro první testovací sadu. . . 72
6.9 Vyhledávací trasy pro druhý zátěžový test vyhledávání. 72
6.10 Průměrná a maximální doba odpovědi serveru pro první a druhou testovací sadu. 72

Kapitola 1

Úvod

Aplikace pro vyhledávání spojů veřejné dopravy se celosvětově těší stále rostoucí oblibě. Umožňují naplánování kompletní cesty z místa A do místa B bez nutnosti studování jednotlivých jízdních řádů, čímž jsou schopné uživatelům ušetřit nemalé množství času. I v České republice tak existuje celá řada aplikací, které vyhledávání v jízdních řádech umožňují.

Tyto aplikace můžeme obecně rozdělit do dvou kategorií. První kategorii tvoří vyhledávače konkrétních dopravců, které slouží zpravidla pouze pro vyhledávání spojů tohoto dopravce. Reálně využívat je tak mohou pouze lidé, kteří již vědí, že daný dopravce na jimi požadované trase jezdí. Vyhledávání v takových aplikacích pak může sloužit primárně k účelu přímého zakoupení jízdenky, což tyto aplikace většinou umožňují. V České republice se k takovýmto dopravcům řadí například společnosti Student Agency[1] (respektive dceřiná společnost RegioJet[2]) a Leo Express[3], které poskytují jak vlastní vyhledávač, tak právě i možnost přímého zakoupení jízdenky na nalezený spoj.

Druhou kategorii pak tvoří obecné vyhledávače, které umožňují vyhledávání spojů napříč více dopravci i dopravními prostředky. K zajištění jejich funkčnosti je nutné seskupit a propojit data od jednotlivých dopravců. V České republice byl speciálně k tomuto účelu založen *Celostátní informační systém o jízdních řádech* (CIS JŘ)[4]. Provozovatelem tohoto systému je od roku 2001 společnost CHAPS s.r.o[5], která i sama vyvíjí rodinu aplikací IDOS[6], které slouží zejména právě pro vyhledávání nad kompletními daty z CIS JŘ. Do rodiny aplikací IDOS patří mimo jiné aplikace IDOS pro internet[7], která je v České republice pro vyhledávání spojů bezkonkurenčně nejpoužívanější [23].

IDOS poskytuje mimo jiné i rozhraní, které umožňuje provádět vyhledávání nad daty z CIS JŘ externím aplikacím. Mezi takovéto aplikace patří například Pubtran[8], MHDApp[9]

[1] https://www.studentagency.cz
[2] https://www.regiojet.cz
[3] https://www.le.cz
[4] http://www.chaps.cz/cs/products/CIS
[5] http://www.chaps.cz/
[6] https://www.chaps.cz/cs/products
[7] http://www.chaps.cz/cs/products/IDOS-internet
[8] https://play.google.com/store/apps/details?id=cz.fhejl.pubtran&hl=cs
[9] http://www.mhdapp.cz/

nebo také například vyhledávače Dopravního podniku hlavního města Prahy (DPP)[10] či Českých drah[11].

1.1 Motivace

V rámci bakalářské práce *Využití grafových databází pro vyhledávání spojů veřejné dopravy* z roku 2015 [3] (dále bude uváděno jako *BPJCH*) byly detailněji zkoumány vyhledávací algoritmy vybraných dopravců a aplikací v České republice. Výsledkem zkoumání bylo zjištění, že vyjma aplikací soukromých dopravců řeší drtivá většina aplikací vyhledávání delegováním vyhledávacích požadavků na rozhraní IDOS, což bylo nastíněno již v úvodu této práce.

Dalším výstupem zmíněné bakalářské práce *BPJCH* [3] bylo navržení schématu grafu, do kterého je možné data o jízdních řádech, která jsou ze své podstaty grafová, uložit. Schéma grafu bylo navrženo zejména s ohledem na možnost přímého vyhledávání. Hlavním přínosem práce pak bylo navržení vyhledávacího algoritmu, který umožňuje vyhledávání spojů z místa A do místa B s volitelným počtem přestupů a dle data odjezdu nebo příjezdu. Ukládání dat a následné vyhledávání bylo demonstrováno na grafové databázi Neo4j[12], která je jednou z nejrozšířenějších a nejpokročilejších grafových databází [5].

1.2 Cíle

Cílem této diplomové práce je vytvořit komplexní webovou aplikaci, která bude umožňovat nahrát či ručně vložit data libovolných jízdních řádů a následně nad nimi bude schopná provést vyhledávání. Součástí aplikace bude administrační rozhraní, ve kterém bude možné jízdní řády efektivně editovat. Pro ukládání a zejména následné vyhledávání bude využito grafové databáze Neo4j. Struktura ukládaných dat a jádro vyhledávacího algoritmu pak bude z větší části vycházet z výsledků bakalářské práce *BPJCH* [3]. Současně je cílem této práce vytvořit také mobilní aplikaci, která bude umožňovat jednoduché vyhledávání nad vloženými jízdními řády.

1.3 Přínos

Hlavním přínosem této práce je vytvoření obecné platformy pro správu jízdních řádů a následné vyhledávání nad nimi. Aplikace přitom bude umožňovat nahrát zcela libovolné jízdní řády ve standardizovaném formátu, čímž bude na jednom místě umožněno vyhledávání nad jízdními řády libovolných dopravců. Vzhledem k zlepšující se situaci s uvolňováním dat o jízdních řádech z CIS JŘ [30] [31] by pak aplikace výhledově měla sloužit jako alternativa k rodině aplikací IDOS, kterou dnes, ať už přímo, nebo přes využívání vyhledávacího rozhraní, musí využívat drtivá většina aplikací třetích stran, které chtějí vyhledávat nad jízdními řády z CIS JŘ.

[10] http://spojeni.dpp.cz/
[11] http://www.cd.cz/
[12] https://neo4j.com

Kapitola 2

Analýza

V této kapitole se budeme věnovat analýze již existujících aplikací pro vyhledávání spojů. Na jejich základě dále provedeme analýzu požadavků, které by měly obě naše aplikace - webová a mobilní - splňovat. Tyto požadavky budou i základem budoucích případů užití obou aplikací. V neposlední řadě si definujeme také zamýšlené role, ve kterých budou moci uživatelé aplikace vystupovat.

2.1 Současný stav

V úvodu práce již bylo zmíněno několik existujících aplikací pro vyhledávání spojů v České republice. Na tyto aplikace se nyní podíváme blíže zejména z uživatelského hlediska, čímž si přiblížíme aktuální stav na poli vyhledávačů. Analýzou konkrétních implementačních detailů vyhledávacích algoritmů se již zabývala bakalářská práce *BPJCH* [3], a tyto detaily zde tak nebudeme znovu rozebírat.

2.1.1 IDOS

IDOS[1] je rodina aplikací pro vyhledávání v jízdních řádech vlakové, autobusové, letecké a městské hromadné dopravy, jejímž provozovatelem je společnost CHAPS s.r.o.[2]. Jak již bylo řečeno v úvodu této práce, společnost CHAPS je také provozovatelem *Celostátního informačního systému o jízdních řádech* (CIS JŘ)[3]. Ten dle platného zákona o silniční dopravě (zákon číslo 111/1994 Sb. a prováděcí vyhláška 388/2000 Sb.) a zákona o dráhách (zákon číslo 266/1994 Sb.) obsahuje "*schválené jízdní řády linek veřejné vnitrostátní linkové dopravy (včetně městské autobusové dopravy), schválené jízdní řády linek veřejné mezinárodní linkové dopravy, které mají na území ČR zastávku pro nástup nebo výstup cestujících a schválené jízdní řády veřejné drážní osobní dopravy na dráze celostátní, regionální, tramvajové, trolejbusové, speciální a lanové provozované na území ČR*".

Dle vyjádření jednatele společnosti CHAPS Tomáše Chlebničana z roku 2014 je systém CIS JŘ z hlediska propracovanosti legislativy, ale i rozsahu dat, která jsou od jednotlivých

[1] https://www.chaps.cz/cs/products
[2] https://www.chaps.cz
[3] http://www.chaps.cz/cs/products/CIS

dopravců centrálně sbírána, věcí přinejmenším v Evropě naprosto ojedinělou. Existují sice určité zárodky podobných systémů, ale většinou na úrovni geograficky menších celků než na celostátní úrovni [32].

Díky exkluzivnímu přístupu k datům z CIS JŘ, která do dnešní doby nejsou kompletně volně dostupná ve strojově zpracovatelném formátu [30] [31] - ať už kvůli neschopnosti nebo nedostatku vůle státního aparátu - mají aplikace IDOS a společnost CHAPS na trhu prakticky neotřesitelnou pozici a jejich konkurence je velmi omezená [25].

Kompletní portfolio aplikací IDOS si nyní představíme.

2.1.1.1 IDOS pro lokální počítače a sítě

Jedná se o desktopovou aplikaci určenou výhradně pro operační systém Windows. Aplikace je zpoplatněná a umožňuje různorodou práci (vyhledávání, získávání podrobných informací o spojích) s kompletními jízdními řády z CIS JŘ. Pro účely této práce pro nás není tato aplikace z důvodu cílové platformy a zpoplatnění příliš zajímavá, nebudeme se jí tedy blíže věnovat. Více informací je dostupných přímo na stránkách produktu[4].

2.1.1.2 IDOS pro mobilní telefony

Ve spolupráci s operátory sítí mobilních telefonů je možné si přes tuto aplikaci vyžádat informace o dopravním spojení například za pomoci odeslání SMS zprávy v konkrétním znění či za uskutečnění hlasového hovoru. Toto řešení pro nás opět není z důvodu jeho zaměření příliš zajímavé, dále se mu tedy věnovat nebudeme. Více informací je dostupných přímo na stránkách produktu[5].

2.1.1.3 IDOS pro internet

IDOS pro internet je webová aplikace dostupná na http://www.idos.cz. Jedná se o kompletní plánovač cest pro Českou republiku a Slovensko, poskytuje ale také další přeshraniční autobusová a vlaková spojení po Evropě [4]. V České republice je to bezkonkurenčně nejpoužívanější vyhledávač dopravních spojení [23], který mimo jiné již v roce 2012 vyhrál soutěž Evropské komise o nejlepší evropský přeshraniční multimodální[6] plánovač dopravního spojení [4].

Vyhledávač umožňuje zvolit jízdní řád, ve kterém bude následně uživatel vyhledávat. Jízdní řády přímo odpovídají jednotlivým jízdním řádům z CIS JŘ, na výběr má tak uživatel mimo jiné kompletní jízdní řády železniční, autobusové a letecké dopravy. V neposlední řadě pak může uživatel volit z více než stovky jízdních řádů městské hromadné dopravy větších českých měst. Obrovskou výhodou je možnost kombinovat při vyhledávání jednotlivé jízdní řády, takže není problém vyhledávat například v kombinaci jízdních řadů vlaků a autobusů.

Samotné vyhledávání je už v základní verzi do velké míry parametrizovatelné. Uživatel tak může mimo jiné vyhledávat dle času odjezdu nebo příjezdu, přidat až tři průjezdní místa

[4] https://www.chaps.cz/cs/products/IDOS-lokal
[5] https://www.chaps.cz/cs/products/IDOS-mobil
[6] Multimodální přepravní systém je druh dopravy využívající více dopravních oborů při přepravě nákladu nebo osob.

či vyžadovat pouze bezbariérová spojení. Aplikace také umožňuje zadat nástupní i výstupní místo kliknutím do mapy a automaticky pak dohledá nejbližší stanice pro nástup a výstup.

V pokročilém módu jsou pak možnosti vyhledávání ještě výrazně rozšířené. Uživatel může navolit maximální počet přestupů (0 - 10), navolit vlastní minimální počet minut nutných na přestup v rámci konkrétního jízdního řádu či preferovat, nebo naopak vyloučit z dopravy zvolené dopravce a spoje.

Samotné vyhledávání spojení je zpravidla velmi rychlé a nalezené výsledky v drtivé většině případů opravdu ty nejlepší možné. Autorovi této práce se i při každodenním využívání aplikace stalo zatím jen naprosto ojediněle, že aplikace nalezla výsledek, který byl pro přesun z místa A do místa B naprosto nevhodný, zejména s ohledem na další nalezené výsledky. Velkou přidanou hodnotou aplikace je dostupnost dat o propojení blízkých stanic, kdy se ve výsledcích vyhledávání může zobrazovat přestup mezi dvěma stanicemi s rozdílným jménem, které jsou ale pro pěší přesun stále dostatečně blízko. Průměrnou délku takového přesunu aplikace také zobrazuje. Aplikace dále zohledňuje případné výluky v jízdních řádech nahlášené jednotlivými dopravci a také umí například zobrazovat aktuální zpoždění spojů, které jsou již na cestě.

Aplikace vždy umožňuje zobrazení kompletního detailu nalezených spojení. Tato spojení je většinou možné zobrazit i na mapě. Mimo jiné aplikace také disponuje rozsáhlou databází fotografií vybraných stanic. V neposlední řadě aplikace umožňuje přímé zakoupení jízdenky na vybrané spoje, většinou se ale jedná o přesměrování na stránky konkrétního dopravce.

2.1.1.4 IDOS pro mobilní zařízení

Pod pojmem *IDOS pro mobilní zařízení* se skrývá řada mobilních aplikací určených pro mobilní platformy Android[7], iOS[8] a Windows Phone[9]. Ačkoliv jsou zde aplikace uváděné jako aplikace IDOS, jedná se v některých případech o aplikace třetích stran, které nevyvíjí společnost CHAPS, ale pouze je pro vyhledávání na mobilních zařízeních doporučuje. Jednotlivé aplikace si nyní představíme.

- **CG Transit**[10] - Aplikace od vývojářů Circlegate[11] je dostupná na všechny tři uvedené platformy. Vyhledávání dopravních spojení probíhá přímo na cílovém zařízení, pro vyhledávání tedy není nutné připojení k internetu. Po instalaci aplikace si uživatel může stáhnout zvolené jízdní řády z vybraných měst Kanady a USA, zejména má ale na výběr z velkého množství jízdních řádů evropských dopravců. Pro účely této práce je nejzajímavější kompletní balík jízdních řádů České republiky (vlaky, autobusy i jízdní řády MHD z CIS JŘ). Ačkoliv je aplikace nabízena ke stažení zdarma, na všechny jízdní řády je po uplynutí měsíční zkušební doby nutné zakoupit příslušnou licenci. Například poplatek za kompletní jízdní řády ČR činil dle stránek aplikace v době psaní této práce 3,59 €.

  Aplikace podporuje stejně jako webová aplikace *IDOS pro internet* rozsáhlou parametrizaci vyhledávání. Umožňuje tak vyhledávat dle času příjezdu i odjezdu, přidávat velké

[7] https://www.android.com/
[8] http://www.apple.com/cz/ios/ios-10/
[9] https://www.microsoft.com/en-us/windows/windows-10-mobile-upgrade
[10] http://www.circlegate.com/cs/cgt
[11] http://www.circlegate.com/cs

množství průjezdních nebo přestupních stanic, zadat maximální možný počet přestupů nebo například vyžadovat jen určité dopravní prostředky.

Vyhledávání spojení je rychlé a kvalita nalezených výsledků je srovnatelná s výsledky, které vrací webová aplikace *IDOS pro internet*. U nalezených výsledků je možné zobrazit detaily jednotlivých spojů včetně zobrazení stanic i celých tras spojů na mapě. Častá vyhledávání je možné uložit do oblíbených, čímž při opakovaném vyhledávání odpadá nutnost znovuzadávání výchozí a cílové stanice.

- **Jízdní řády IDOS**[12] - Jedná se o oficiální mobilní aplikaci jízdních řádů IDOS. Aplikace je k dispozici pro platformy iOS a Android. Aplikace je k dispozici zdarma a zpoplatněny žádným způsobem nejsou ani jízdní řády, které odpovídají jízdním řádům z CIS JŘ. V aplikaci se bohužel zobrazují poměrně velké bloky reklam, a to i mezi nalezenými výsledky. Tyto je nicméně možné vypnout uhrazením jednorázového poplatku (v době psaní této práce činil poplatek 4,99 €).

  Samotné vyhledávání probíhá online, je tedy nutné mít zařízení připojené k internetu. Parametrizace vyhledávání je opět poměrně pestrá a srovnatelná s aplikací CG Transit. Je umožněno vyhledávat dle data odjezdu i příjezdu, přidat jednu průjezdní nebo přestupní stanici, volit maximální počet přestupů, vybírat požadované dopravní prostředky nebo například vyžadovat pouze bezbariérové spoje. U nalezených dopravních spojení je opět možné zobrazovat detaily jednotlivých spojů (včetně zobrazení na mapě) a například pro jízdní řády Pražské integrované dopravy (PID)[13] aplikace umožňuje snadné předvyplnění SMS pro zakoupení jízdenky.

2.1.2 Aplikace jednotlivých dopravců

V předchozí kapitole 2.1.1 jsme si představili řadu aplikací pro vyhledávání dopravních spojení napříč více dopravci zejména v rámci České republiky. Rádi bychom na tomto místě uvedli konkurenční alternativní nástroje, které by poskytovaly srovnatelné možnosti vyhledávání spojení, bohužel je jejich existence poměrně omezená. Jak jsme uvedli již dříve v kapitole 2.1.1, je to dáno zejména neveřejností kompletních dat z CIS JŘ ve strojově zpracovatelném formátu. V minulých letech vypadal velmi nadějně například projekt Bileto[14], který již umožňoval provádět vyhledávání nad jízdními řády celé řady soukromých dopravců (viz bakalářská práce *BPJCH* [3]), tuto možnost již Bileto nenabízí, ale místo toho se věnuje dodávce komplexních nástrojů pro prodej jízdenek a správu dat jednotlivých soukromých dopravců [2]. Příkladem je například rezervační systém dopravce Arriva Slovakia, a.s.[15].

Představíme si tak alespoň v ČR existující vyhledávače vybraných dopravců.

[12] https://play.google.com/store/apps/details?id=cz.mafra.jizdnirady
[13] https://ropid.cz
[14] http://www.bileto.com/cs/index.html
[15] https://listky.arrivaexpress.sk

2.1.2.1 Student Agency / RegioJet

Student Agency[16] je významným soukromým autobusovým dopravcem provozujícím mezinárodní dopravu. Společnost RegioJet[17] - dceřiná společnost Student Agency - provozuje osobní železniční dopravu a je největším soukromým železničním dopravcem v ČR [10].

Na stránkách obou společností je k dispozici vyhledávač, který umožňuje mimo jiné vyhledávat napříč jízdními řády vlakové i autobusové dopravy. Vyhledávání slouží primárně k účelu zakoupení jízdenky a oproti již dříve představeným vyhledávačům je nepoměrně jednodušší. Vyhledávač spojení umožňuje pouze zadání výchozí a cílové stanice a dne odjezdu, povinnou položkou je pak zadání počtu cestujících a jejich tarifního pásma.

Ve výsledcích vyhledávání je uživateli zobrazen seznam nalezených spojů včetně rozsahu ceny jízdenky pro navolený počet cestujících dle jejich tarifního pásma. Z této stránky je pak uživateli umožněno přejít přímo na nákupní proces jízdenky.

2.1.2.2 České dráhy

České dráhy jsou národní železniční společností v České republice a současně největším železničním dopravcem na území ČR [1]. Na svých webových stránkách[18] poskytují vyhledávač pro nalezení spojení a následné zakoupení jízdenky. Vyhledávač[19] je ve skutečnosti pouze upravenou verzí vyhledávače IDOS od společnosti CHAPS, která je vyvíjena speciálně pro účely Českých drah. Z tohoto důvodu je na vyhledávač nutné nahlížet jako na další vyhledávač z rodiny IDOS.

V pokročilém módu umí vyhledávač přidat až tři průjezdní nebo přestupní stanice, zvolit maximální možný počet přestupů (10), preferovat pouze vybrané dopravní prostředky nebo dopravce či definovat další požadavky na nalezené spoje (například dostupnost přepravy kol či dostupnost WiFi na palubě).

U nalezených výsledků jsou uživateli zobrazena případná omezení na trase a navíc, pokud už je spoj na cestě, zobrazuje se také případné zpoždění tohoto spoje. Z výsledků vyhledávání je možné přejít přímo na nákup jízdenky pro konkrétní spoje.

2.1.3 Další aplikace

Ačkoliv jsme si již uvedli celou řadu aplikací pro vyhledávání dopravních spojení v České republice, v žádném případě jsme nepokryli jejich kompletní výčet. Z mobilních aplikací jsme se kupříkladu nevěnovali aplikacím MHDApp[20] či Pubtran[21], které dle počtu stažení patří k velmi oblíbeným aplikacím pro vyhledávání dopravních spojení.

Z webových aplikací jsme se pak nevěnovali například aplikacím pro vyhledávání spojení městské hromadné dopravy (MHD) různých měst po celé ČR, ke kterým se řadí mimo jiné vyhledávač spojení Dopravního podniku hlavního města Prahy[22].

[16] https://www.studentagency.cz/
[17] https://www.regiojet.cz
[18] https://www.cd.cz/spojeni-a-jizdenka/
[19] https://www.chaps.cz/cs/products/vyhledavac-cd
[20] http://mhdapp.cz/
[21] https://play.google.com/store/apps/details?id=cz.fhejl.pubtran&hl=cs
[22] http://spojeni.dpp.cz/

Faktem je, že tyto aplikace nepřinášejí v rámci naší analýzy žádné převratné funkcionality, které by již dříve zanalyzované aplikace neumožňovaly, a všechny vyjmenované dokonce pro vyhledávání buď přímo využívají služeb IDOSu, nebo získávají data z CIS JŘ. Výjimkou budiž ještě plánování tras na portálu Mapy.cz[23] či Google Maps[24], které shodně také obsahují informace o veřejné dopravě vybraných dopravců a umožňují tak naplánovat trasu pomocí jejich spojů. V tomto případě ale nejde o primární funkcionalitu a na portálu Mapy.cz je plánování tras prostředky hromadné dopravy teprve ve verzi *beta*. Z tohoto důvodu se dalším aplikacím již věnovat nebudeme a pustíme se rovnou do analýzy požadavků na námi vytvářený vyhledávač.

2.2 Parametry vyhledávače

Rozhraní pro vyhledávání spojů bude bezpochyby nejpoužívanější částí aplikace a současně první věcí, kterou nově příchozí uživatel uvidí. Pro vytvoření konkurenceschopného vyhledávače je nutné správně definovat všechny možné vstupní parametry pro vyhledávání, které bude náš vyhledávač podporovat, aby byl využitelný co nejvyšším počtem uživatelů.

V kapitole 2.1 jsme si již představili celou řadu aplikací, které vyhledávání spojů umožňují. Jejich studiem jsme zjistili, že většina z nich má velmi podobné možnosti parametrizace vyhledávání spojů. Touto analýzou se také již zabývala bakalářská práce *BPJCH* [3] či podrobněji například také diplomová práce *Analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití* [11].

Analýzou výše uvedených aplikací a zdrojů vznikl seznam vstupních parametrů, které bude muset náš vyhledávač podporovat:

- Výchozí stanice
- Cílová stanice
- Datum a čas
- Výběr, zda chceme hledat podle data a času odjezdu z výchozí stanice, nebo příjezdu do cílové stanice.
- Maximální počet přestupů
- Možnost přidání minimálně jedné průjezdní či přestupní stanice
- Možnost vyhledávání pouze bezbariérových spojů

Oproti bakalářské práci *BPJCH* [3] se vstupní parametry rozšířily o poslední dva atributy. Pro úplnost zde uvedeme, že pod pojmem *Možnost vyhledávání pouze bezbariérových spojů* zde uvažujeme, že vyhledávač nalezne pouze takové spoje, kde je vozidlo bezbariérové a současně jsou bezbariérové i všechny stanice, na kterých je nutné nastoupit nebo vystoupit z vozu. Mezi takové stanice samozřejmě počítáme i stanice přestupní.

[23] https://mapy.cz/
[24] https://www.google.cz/maps/

2.3 Standardizovaný formát dat o jízdních řádech

Jedním z cílů této práce je umožnit uživatelům přímý import dat o jízdních řádech. K tomu je nutné specifikovat formát, který bude naše aplikace pro import očekávat. Celosvětově existuje několik více či méně rozšířených formátů pro přenos dat o jízdních řádech. Mimo jiné jsou to například formáty TRANSMODEL[25] či SIRI[26]. Česká legislativa také nedávno definovala vlastní formát JDF (Jednotný Datový Formát)[27] [31]. Nejvíce uznávaný a celosvětově pravděpodobně nejrozšířenější formát je však GTFS (General Transit Feed Specification) [9][33], na který se v rámci této práce zaměříme. Podpora dalších formátů by nicméně byla vítána, v českých podmínkách by určitě bylo vhodné přidat podporu i pro formát JDF, který je, jak uživatel prostudováním zjistí, formátu GTFS podobný.

GTFS je formát, který data o jízdních řádech ukládá v několika samostatných *CSV*[28] souborech, které mají příponu *.txt*. Formát definuje množinu povinných a nepovinných souborů, které by měl kompletní balík obsahovat. Každý ze souborů pak definuje množinu sloupců, které jsou opět buď povinné, nebo volitelné. V rámci implementace naší aplikace budeme pracovat se všemi povinnými položkami tak, jak je formát GTFS specifikuje. Současně si přibereme i řadu nepovinných údajů, které se pro účely naší aplikace budou hodit.

2.3.1 Slovníček pojmů GTFS

Nyní si představíme jednotlivé soubory z balíku dat GTFS, se kterými budeme v naší aplikaci pracovat. Názvy budeme uvádět v originálním anglickém znění a hned uvedeme i český ekvivalent, který budeme používat všude dále v této práci. Ke každému souboru navíc uvedeme vysvětlení, co konkrétně daná data reprezentují v rámci jízdního řádu, protože tato informace nemusí být nezasvěcenému čtenáři na první pohled zřejmá. Většina souborů byla představena již v bakalářské práci *BPJCH* [3], pro účely této práce je ale struktura jízdních řádů ve formátu GTFS natolik významná, že je nutné ji uvést.

- **Agency** (dopravci) - Soubor obsahuje údaje o jednotlivých dopravcích, kteří provozují spoje uvedené v daném balíku dat. Například *Dopravní podnik hl. m. Prahy*. Jedná se o povinný soubor.
- **Stops** (stanice) - Soubor obsahuje údaje o všech stanicích, které se objevují v jízdních řádech daného balíku dat. Například stanice *Náměstí Míru*. Jedná se o povinný soubor.
- **Routes** (spoje) - Soubor obsahuje údaje o spojích, které jezdí v rámci daného jízdního řádu. Jeden záznam zpravidla reprezentuje jeden spoj s konkrétním číslem. Například spoj *metro A*, či *tramvaj č. 26*. Soubor je povinný.
- **Trips** (jízdy) - Soubor obsahuje údaje o jednotlivých jízdách konkrétního spoje. Pokud tedy například spoj *tramvaj č. 26* jezdí pravidelně každých 20 minut, každá takováto nová jízda bude uvedena právě v tomto souboru. Soubor je povinný.

[25] http://www.transmodel.org/index.html
[26] http://user47094.vs.easily.co.uk/siri/index.htm
[27] https://www.chaps.cz/files/cis/jdf-1.10.pdf
[28] https://tools.ietf.org/html/rfc4180

- **Stop times** (zastavení jízdy) - Soubor obsahuje jednotlivá zastavení konkrétní jízdy v definovaných časech ve stanicích. Seřazená zastavení jízdy tvoří kompletní informace o jízdě, průjezdních stanicích a časech příjezdů a odjezdů z jednotlivých stanic. Například informace, že *jedna jízda tramvaje č. 26 staví ve stanici Náměstí Míru v 9:00* je uložena právě v tomto souboru. Soubor je povinný.
- **Calendar** (interval platnosti) - Soubor obsahuje definované intervaly platnosti, které jsou platné pro jednotlivé jízdy. Například údaj, že *jízda tramvaje č. 26 s výjezdem v 8:55* jezdí každý všední den od 13.5.2017 do 16.6.2017 je uložen právě v tomto souboru. Soubor je povinný.
- **Calendar dates** (výjimky z intervalu platnosti) - Soubor obsahuje výjimky z intervalu platnosti. Pokud například záznam v intervalu platnosti říká, že jízdy navázané na tento interval jezdí každý všední den od 13.5.2017 do 16.6.2017, tak záznam v tomto souboru může například definovat tyto výjimky - *15.5.2017 (pondělí) není tento interval platný*, nebo *14.5.2017 (neděle) je tento interval platný*. Soubor je nepovinný. Praktické využití může být například pro definování platnosti jízdních řádů o státních svátcích.
- **Shapes** (průjezdní body) - Soubor definuje geografické souřadnice bodů, kterými projíždí trasa jízdy. Soubor se využívá pro definování průjezdních bodů jízd, díky kterým je pak možné trasy jízd zobrazovat na mapě. Soubor je nepovinný.

2.4 Uživatelské role

Systém bude definovat tři základní uživatelské role. Dle příslušnosti uživatele aplikace k roli bude uživatel oprávněn provádět určité akce. Role v našem systému jsou definovány tak, že každá vyšší role automaticky přebírá všechna oprávnění rolí předchozích.

2.4.1 Nepřihlášený uživatel

Nepřihlášený uživatel bude moci vyhledávat spoje napříč dostupnými jízdními řády. Současně bude moci zobrazit detaily nalezených spojů.

2.4.2 Správce jízdního řádu

Správce jízdního řádu bude oproti nepřihlášenému uživateli oprávněn vkládat, editovat a mazat údaje o jízdním řádu, k němuž bude mít oprávnění. Současně bude moci importovat a exportovat jízdní řád ve formátu GTFS. Jeden *správce jízdního řádu* bude moci spravovat až *n* jízdních řádů. *Správce jízdního řádu* bude vkládat uživatel s rolí *administrátor*. Při založení účtu se uživateli vygeneruje jednorázové heslo do aplikace, které bude nutné při prvním přihlášení změnit. Tato role bude dostupná pouze přes webovou aplikaci.

2.4.3 Administrátor

Administrátor bude oprávněn provádět veškeré akce příslušící *správci jízdního řádu* nad libovolně zvoleným jízdním řádem. Současně bude moci zakládat nové uživatele v roli *správce*

jízdního řádu a všem těmto uživatelům přiřazovat oprávnění k úpravám jednotlivých jízdních řádů. Role *administrátor* bude uživateli přidělena přímým zásahem v databázi. Tato role bude dostupná pouze přes webovou aplikaci.

2.5 Funkční požadavky

V této kapitole definujeme požadavky na funkce aplikace. Požadavky seskupíme dohromady podle uživatelských rolí, kterých se týkají. Z kapitoly 2.4 nicméně již víme, že každá vyšší role automaticky přebírá všechna oprávnění rolí předchozích, například požadavky *nepřihlášeného uživatele* se tak fakticky týkají i *správce jízdního řádu* a *administrátora*.

Všechny požadavky jsou relevantní pro webovou aplikaci. Pro mobilní aplikaci jsou relevantní pouze požadavky 1 - 3 z pohledu nepřihlášeného uživatele z kapitoly 2.5.1.

Požadavky budeme uvádět podrobně tak, že budou současně definovat i zamýšlené případy užití aplikace jednotlivými uživateli. Z tohoto důvodu již práce neobsahuje vlastní kapitolu pro definici případů užití, které by z velké části pouze kopírovaly zde uvedené požadavky.

2.5.1 Požadavky z pohledu nepřihlášeného uživatele

1. Vyhledávání spojů
   (a) Zvolení jízdního řádu
   Aplikace bude umožňovat zvolit jízdní řád, ve kterém bude uživatel vyhledávat.
   (b) Vyhledávání dle parametrů
   Aplikace bude umožňovat provést vyhledávání dle vstupních parametrů uvedených v kapitole 2.2. Vyhledávací formulář bude uzpůsoben snadnému zadávání údajů.
2. Zobrazení následujících spojů
   Aplikace bude umožňovat zobrazit spoje, které přímo následují po již zobrazených spojích uskutečněného vyhledávání.
3. Zobrazení předchozích spojů
   Aplikace bude umožňovat zobrazit spoje, které přímo předcházejí již zobrazeným spojům uskutečněného vyhledávání.
4. Zobrazení detailu stanice
   Aplikace bude umožňovat zobrazit umístění stanice na mapě, pokud k dané stanici budou dostupné souřadnice GPS. Současně se na detailu stanice budou zobrazovat další v datech dostupné informace o stanici.
5. Zobrazení detailu spoje
   (a) Textové zobrazení spoje
   Aplikace bude umožňovat zobrazení kompletního detailu spoje z výsledků vyhledávání. Detailem se rozumí dostupné informace o všech zastaveních spoje, o dopravci, který spoj provozuje, a dnech, ve kterých je spoj v provozu.

   (b) Zobrazení spoje na mapě
   Pokud budou v datech k dispozici údaje o průjezdních bodech spoje, bude aplikace umožňovat zobrazit trasu tohoto spoje z výsledků vyhledávání na mapě. Na mapě budou taktéž zobrazeny všechny stanice daného spoje, pokud budou dostupné GPS souřadnice těchto stanic.

6. Přihlášení do administrace
   Aplikace bude umožňovat přihlášení do administrační části aplikace. Přihlášení bude uskutečněno pomocí uživatelského jména a hesla.

2.5.2 Požadavky z pohledu správce jízdního řádu

1. Správa jízdních řádů

   (a) Výběr jízdního řádu
   Aplikace bude uživateli zobrazovat všechny jemu dostupné jízdní řády pro administraci. Po výběru konkrétního jízdního řádu bude možné přejít na jeho detail.

   (b) Zobrazení a editace jízdního řádu
   Aplikace bude umožňovat zobrazit detailní informace zvoleného jízdní řádu. Jízdní řád bude možné zneplatnit, čímž se přestane nabízet uživatelům pro vyhledávání.

   (c) Import jízdního řádu
   Aplikace bude umožňovat importovat jízdní řád ve formátu GTFS. Očekávaný formát importního souboru bude *.zip* archiv, který bude obsahovat složku s jednotlivými soubory GTFS balíku dat s validními jmény dle kapitoly 2.3.1. Importem jízdního řádu dojde ke kompletnímu nahrazení dříve nahraných dat jízdního řádu. Během importu jízdního řádu se nebude tento nabízet uživatelům ve vyhledávacím formuláři a v administrační části nebude možné s tímto jízdním řádem provádět žádné akce.

   (d) Export jízdního řádu
   Aplikace bude podporovat export dostupných dat o jízdním řádu do formátu GTFS dle kapitoly 2.3.1. Výsledný exportní soubor bude *.zip* archiv, který bude obsahovat všechny validní soubory GTFS balíku zvoleného jízdního řádu. Volitelné soubory budou součástí exportního archivu pouze za předpokladu, že pro ně má jízdní řád dostupná data.

2. Správa dat jízdních řádů

   Aplikace bude umožňovat zobrazovat, vkládat, editovat a mazat konkrétní záznamy z jízdního řádu. Záznamy zde rozumíme jednotlivé entity GTFS balíku dat. Jednotlivé typy entit bude aplikace zobrazovat ve vlastní tabulce, která bude umožňovat filtrovat údaje dle příslušných atributů zvolené entity. Data v tabulkách bude možné řadit podle jednotlivých sloupců a všechny tabulky s entitami jednotlivých typů budou stránkovatelné. Konkrétní sloupce pro jednotlivé typy entit budou představeny v kapitole 4. Po zvolení konkrétního záznamu bude možné přejít na jeho detail a editaci.

3. Správa dopravců
    (a) Seznam dopravců
    Aplikace bude umožňovat zobrazit seznam všech dopravců zvoleného jízdního řádu dle požadavku 2.
    (b) Vložení dopravce
    Aplikace bude umožňovat vložit nového dopravce.
    (c) Detail a editace dopravce
    Aplikace bude umožňovat zobrazit detail dopravce a provést editaci jeho údajů.
    (d) Smazání dopravce
    Aplikace bude umožňovat smazat zvoleného dopravce. Smazání dopravce bude možné jen v případě, že k danému dopravci nebudou existovat žádné navázané spoje.
4. Správa spojů
    (a) Seznam spojů
    Aplikace bude umožňovat zobrazit seznam všech spojů zvoleného jízdního řádu dle požadavku 2.
    (b) Vložení spoje
    Aplikace bude umožňovat vložit nový spoj.
    (c) Detail a editace spoje
    Aplikace bude umožňovat zobrazit detail spoje a provést editaci jeho údajů.
    (d) Smazání spoje
    Aplikace bude umožňovat smazat zvolený spoj. Smazání spoje bude možné jen v případě, že na daný spoj nebudou navázané žádné jízdy.
5. Správa stanic
    (a) Seznam stanic
    Aplikace bude umožňovat zobrazit seznam všech stanic zvoleného jízdního řádu dle požadavku 2.
    (b) Vložení stanice
    Aplikace bude umožňovat vložit novou stanici.
    (c) Detail a editace stanice
    Aplikace bude umožňovat zobrazit detail stanice a provést editaci jejích údajů.
    (d) Smazání stanice
    Aplikace bude umožňovat smazat zvolenou stanici. Smazání stanice bude možné jen v případě, že na ni nebudou navázaná žádná zastavení jízdy.
6. Správa intervalů platnosti
    (a) Seznam intervalů platnosti
    Aplikace bude umožňovat zobrazit seznam všech intervalů platnosti zvoleného jízdního řádu dle požadavku 2.

   (b) Vložení intervalu platnosti
   Aplikace bude umožňovat vložit nový interval platnosti a navázat na něj případné výjimky z intervalu platnosti.
   (c) Detail a editace intervalu platnosti
   Aplikace bude umožňovat zobrazit detail intervalu platnosti a provést editaci jeho údajů včetně editace, mazání a vkládání případných výjimek z tohoto intervalu platnosti.
   (d) Smazání intervalu platnosti
   Aplikace bude umožňovat smazat zvolený interval platnosti. Smazání intervalu platnosti bude možné jen v případě, že na něj nebude navázaná žádná jízda. Smazáním intervalu platnosti dojde současně ke smazání případných navázaných výjimek z tohoto intervalu.

7. Správa jízd
   (a) Seznam jízd
   Aplikace bude umožňovat zobrazit seznam všech jízd zvoleného jízdního řádu dle požadavku 2.
   (b) Vložení jízdy
   Aplikace bude umožňovat vložit novou jízdu a definovat pro ni jednotlivá zastavení na zvolených stanicích v konkrétních časech.
   (c) Detail a editace jízdy
   Aplikace bude umožňovat zobrazit detail jízdy a provést editaci jejích údajů včetně editace, mazání a vkládání jednotlivých zastavení této jízdy.
   (d) Smazání jízdy
   Aplikace bude umožňovat smazat zvolenou jízdu. Smazáním jízdy dojde současně k odstranění všech navázaných zastavení na tuto jízdu.

8. Správa průjezdních bodů
   (a) Seznam průjezdních bodů
   Aplikace bude umožňovat zobrazit seznam všech průjezdních bodů zvoleného jízdního řádu dle požadavku 2.
   (b) Vložení průjezdních bodů
   Aplikace bude umožňovat vložit novou skupinu průjezdních bodů a definovat pro ni jednotlivé průjezdní body s konkrétními GPS souřadnicemi.
   (c) Detail a editace průjezdních bodů
   Aplikace bude umožňovat zobrazit detail skupiny průjezdních bodů a provést editaci jednotlivých průjezdních bodů z této skupiny.
   (d) Smazání průjezdních bodů
   Aplikace bude umožňovat smazat zvolenou skupinu průjezdních bodů.

9. Akce uživatele
   (a) Změna hesla
   Přihlášenému uživateli bude umožněno provést změnu hesla. Změna hesla bude podmíněna správným zadáním původního hesla uživatele.

(b) Odhlášení
Přihlášenému uživateli bude umožněno odhlášení z aplikace.

2.5.3 Požadavky z pohledu administrátora

1. Správa uživatelů

   (a) Vytvoření uživatele
   Aplikace bude umožňovat vytvoření nového uživatele. Při vytvoření uživatele dojde k vygenerování jednorázového hesla, které se administrátorovi zobrazí. Při prvním přihlášení uživatele bude nutné toto heslo změnit.

   (b) Přiřazování jízdních řádů uživatelům
   Administrátorovi bude umožněno přiřazovat jízdní řády jednotlivým uživatelům, kteří budou mít právo je spravovat.

   (c) Smazání uživatele
   Administrátorovi bude umožněno smazat libovolného uživatele vyjma sebe sama.

2.6 Nefunkční požadavky

V této kapitole jsou definovány obecné požadavky, které budou muset aplikace splňovat. Požadavky jsou rozdělené dle typu aplikace.

2.6.1 Nefunkční požadavky na webovou aplikaci

1. Technologie

   (a) Backend část aplikace bude napsána v programovacím jazyce Java za použití frameworku Spring.

   (b) Frontend část aplikace bude napsána ve frameworku Angular 2.

2. Požadavky na výkon

   (a) Aplikace bude plně škálovatelná - horizontálně i vertikálně. Ke snadnému zajištění obou těchto požadavků bude aplikace kompletně bezestavová.

3. Nasazení

   (a) Instance databáze Neo4j poběží na samostatném serveru a bude umožňovat libovolný počet připojení.

4. Zabezpečení

   (a) Aplikace bude ukládat uživatelská hesla v podobě *hash otisků*. Hesla nebudou v žádném případě a za žádných okolností ukládána jako prostý text.

   (b) Aplikace bude dostupná přes zabezpečený protokol HTTPS, aby nemohlo na nezabezpečených sítích dojít k přečtení hesla při přihlašování, případně k přečtení autorizačních tokenů při komunikaci se zabezpečeným rozhraním aplikace.

5. Funkční prostředí
   (a) Aplikace poběží a bude se správně zobrazovat v následujících prohlížečích:
      i. Mozilla Firefox (verze 35)
      ii. Google Chrome (verze 40)
      iii. Opera (verze 25)
      iv. Safari (verze 8)

   Vzhledem k velké rozličnosti implementací a neustálému vývoji jednotlivých prohlížečů není možné zaručit správnou funkčnost na všech budoucích verzích zde uvedených. Zaručená nicméně bude správná funkčnost přímo na verzích zde uvedených v době odevzdání této DP. Na vyšších verzích zde uvedených prohlížečů pak bude správné zobrazení velmi pravděpodobné.

2.6.2 Nefunkční požadavky na mobilní aplikaci

1. Cílová platforma
   (a) Aplikace bude napsána pro operační systém Android s podporou od verze 4.2.

Kapitola 3

Analýza vyhledávání spojení v grafu

V této kapitole se zaměříme na analýzu vyhledávacího algoritmu, který budeme pro vyhledávání používat. S tím je i přímo spojena analýza struktury grafu, ve které budeme data o jízdních řádech udržovat. Touto analýzou se již podrobněji zabývala zejména kapitola 6 bakalářská práce *BPJCH* [3], a některé zde uvedené údaje tak budou pouze prezentovat již představené závěry této práce. Nebudeme se zde tak již věnovat například jednotlivým aspektům optimalizace vyhledávacího algoritmu, které z uvedené práce přebíráme, ale pouze si představíme obecný pohled na ukládání dat do grafu a následné vyhledávání. Současně si pak zavedeme nové požadavky na vyhledávací algoritmus a rozšíříme schéma grafu o zatím neobsažené vrcholy, které budeme pro vyhledávání potřebovat vzhledem k požadavkům na vyhledávač z kapitoly 2.2.

3.1 Struktura grafu

Pro ukládání dat o jízdních řádech do grafu a zejména následné vyhledávání máme na výběr prakticky ze dvou nejrozšířenějších modelů, které jsou výsledkem celé řady studií, které již byly na toto téma zpracovány. Oba modely byly vytvořeny tak, aby na již uložených datech bylo možné provést co nejrychlejší vyhledání všech možných spojení, která jedou ze zadané počáteční stanice do cílové stanice s co nejdřívějším časem příjezdu, a detailně byly představeny a porovnány například v diplomové práci [28]. My si je nyní představíme pouze v krátkosti a zdůvodníme výběr toho z nich, který pro naše účely použijeme.

3.1.1 Time-dependent model

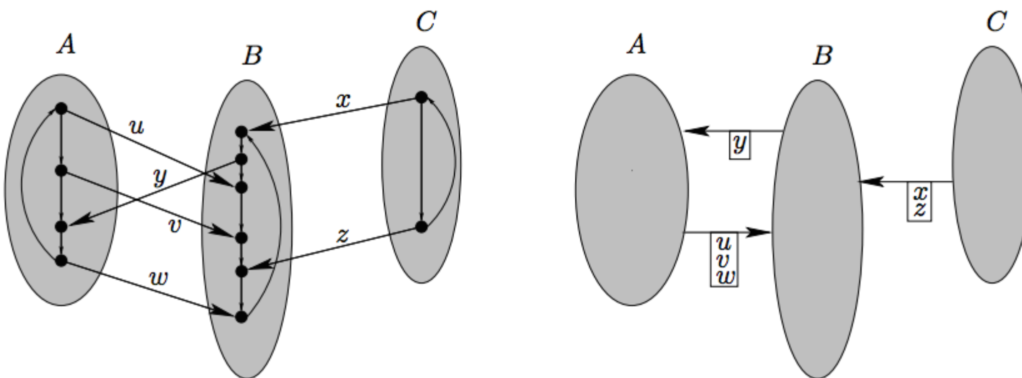
Time-dependent model byl poprvé představen počátkem 90. let 20. století v publikacích [20] a [21]. Vrcholy grafu reprezentují stanice jízdního řádu a mezi dvěma vrcholy existuje právě jedna přejezdová hrana, pokud existuje alespoň jedno bezprostřední spojení mezi těmito stanicemi.

Z popisu modelu je ihned jasné, že graf bude vždy obsahovat relativně málo vrcholů i hran, a to i pro jízdní řády velkého rozsahu. Na jedné stanici se totiž v rámci dne mohou odehrát i tisíce příjezdů a odjezdů různých spojů, které v time-dependent modelu všechny schováme pod jeden vrchol grafu. Jednotlivá zastavení (příjezdy a odjezdy) pak musíme k vrcholu zaznamenávat formou atributů.

3.1.2 Time-expanded model

Time-expanded model byl představen mimo jiné v publikacích [22] a [29]. V základní verzi modelu tvoří vrcholy grafu tzv. *události* na stanici, což není nic jiného, než příjezd nebo odjezd ze stanice. Alternativně je možné vrcholy příjezdu a odjezdu, které se týkají jednoho konkrétního spoje, sloučit do jednoho vrcholu, který pak bude mít dva atributy - čas příjezdu a čas odjezdu. Mezi vrcholy pak existují dva typy hran. První je hrana *přejezdová*, která zaznamenává možnost přejezdu z jedné stanice do druhé. Tyto hrany tak existují pouze mezi vrcholy, které se nacházejí na různých stanicích. Současně je zřejmé, že všechny tyto hrany jsou pouze jednosměrně orientované, a to po směru jízdy daného spoje. Druhým typem hran jsou hrany *přestupní*, které zaznamenávají možnost přestupu z jednoho spoje na druhý. Tyto hrany jsou tak přítomné pouze na vrcholech v rámci jedné stanice. Navíc nejsou mezi všemi vrcholy, kde je možný přestup, ale jedna hrana spojuje vždy pouze dvě bezprostředně po sobě následující události, které získáme seřazením událostí dle jejich času.

Není náhoda, že v právě popsané alternativní variantě modelu grafu odpovídají vrcholy přímo *zastavením*, která jsme definovali již při představování GTFS balíku dat v kapitole 2.3.1. U těch, jak už víme, máme vždy zaznamenaný pouze *čas* příjezdu a odjezdu, ale nikoliv *datum*. Data platnosti jednotlivých zastavení získáváme z intervalu platnosti, který je navázaný na jízdu, ke které se zastavení váže. Time-expanded model je definován stejně - jednotlivé události nemají datum, ale pouze čas. Ve skutečnosti je tak každá událost, která se například opakuje pravidelně každý den, v grafu zaznamenána pouze jedním vrcholem. Při vyhledávání je pak nutné vědět, pro jaký den vyhledáváme, a pro každý navštívený vrchol (událost) kontrolovat, zda je tato událost platná pro aktuální den vyhledávání. Dále je nutné si uvědomit, že pro uvedený model je navíc nutné propojit poslední událost dne s první událostí dne přestupní hranou, abychom při vyhledávání mohli přestupovat mezi spoji také přes půlnoc.



Obrázek 3.1: Time-expanded (vlevo) a time-dependent model grafu se třemi stanicemi A, B, C. Tři jízdy spojují stanici A se stanicí B (jízdy u, v, w), jedna jízda spojuje stanici B a A (y) a dvě jízdy spojují stanici C a B (x, z). U time-expanded modelu si všimneme cyklického propojení vrcholů v rámci všech stanic. Zdroj: [29]

3.1.2.1 Fully time-expanded model

Speciálním případem právě představeného time-expanded modelu je tzv. fully time-expanded model, který byl detailněji představen například v diplomové práci [28]. V tomto modelu má každá událost kromě času i datum, a pokud se periodicky opakuje, je v grafu uložena pro každý den opakování. V tomto modelu tedy nemusíme uchovávat někde bokem informace o platnosti konkrétních událostí, protože je tato informace přímo součástí vrcholu události, ale je očividné, že právě kvůli tomuto faktu bude mít fully time-expanded model mnohonásobně vyšší počet vrcholů i hran než standardní time-expanded model. Platnost jízdních řádů je totiž zpravidla známá minimálně na několik měsíců dopředu. Pro platnost jízdního řádu měsíc dopředu tak například rozdíl ve velikosti modelu time-expanded a fully time-expanded může být i třicetinásobný (Pokud by všechny události byly platné každý den v měsíci a měsíc měl právě třicet dní.).

3.1.3 Zvolená struktura jádra grafu

V bakalářské práci *BPJCH* [3], kde již byly v dřívějších kapitolách uvedené modely také zkoumány, byl zvolen time-expanded model jako základ struktury grafu. Důvodů výběru právě tohoto modelu je celá řada. Jednak umožňuje přímo ukládat zastavení jako jednotlivé vrcholy. Navíc v našem případě budeme chtít mít vyhledávací algoritmus parametrizovatelný například maximálním počtem přestupů nebo minimálním počtem minut na přestup a autoři studie [24] dokázali, že v těchto případech je výhodnější využívat právě time-expanded model.

Ukládání jednotlivých *zastavení* tedy máme prakticky vyřešené. Jednotlivá zastavení v rámci jízdy budeme ukládat přímo jako vrcholy do grafu. Mezi zastaveními budou jednosměrné přejezdové hrany po směru jízdy spoje. Před uložením do grafu tedy jízdy konkrétního spoje seřadíme dle jejich pořadí v rámci jízdy a mezi každou dvojici bezprostředně následujících zastavení vložíme přejezdovou hranu. Mírná komplikace nastává s *přestupními* hranami. Jak jsme si už řekli, přestupní hrana by měla být vždy jen mezi dvěma bezprostředně následujícími událostmi. To je jednoduché, protože každá událost má právě jeden *čas*, a tak stačí události seřadit právě dle času a poté propojit přestupními hranami. V našem případě ale za vrcholy máme *zastavení*, která mají buď pouze čas výjezdu (počáteční zastavení jízdy), nebo pouze čas příjezdu (konečná zastavení jízdy), nebo oba časy - tedy čas příjezdu i výjezdu (všechna ostatní zastavení jízdy). Nastává tedy otázka, podle kterého z časů je potřeba zastavení seřadit, abychom je následně propojili přestupními hranami.

Již v bakalářské práci *BPJCH* [3] jsme navrhli řešení, které si nyní představíme. Uvažujme zastavení z, jeho čas příjezdu C_{pz} a čas výjezdu C_{vz}. Definujeme *čas zastavení* (C_z) takto:

$$C_z = exists\ C_{vz}\ ?\ C_{vz}\ :\ C_{pz}$$

Tedy *čas zastavení* bude odpovídat času výjezdu, pokud tento čas existuje, jinak bude odpovídat času příjezdu. Uvědomíme si, že ve validním jízdním řádu platí pro každé zastavení $C_{vz} >= C_{pz}$. Pokud nyní všechna zastavení v rámci jedné stanice seřadíme dle *času zastavení* a propojíme přestupní hranou, tak přestup v rámci pohybu grafem *po směru seřazení zastavení* bude bez uvažování minimálního potřebného času na přestup vždy možný, protože pro každou dvojici zastavení z_1, z_2, kde z_1 předchází po seřazení z_2, platí:

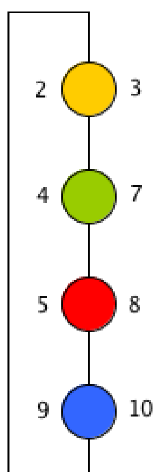
$$C_{pz_1} <= C_{vz_1} = C_{z_1} <= C_{vz_2} = C_{z_2}$$

Tato rovnice platí za dvou předpokladů:

1. z_1 má čas příjezdu, a nejde tedy o první zastavení jízdy. Pokud by šlo o první zastavení jízdy, nemá cenu uvažovat *z* tohoto zastavení o přestupu na jiný spoj a vrchol by pak v rámci stanice byl seřazen dle C_{vz_1} pouze z toho důvodu, aby naopak bylo možné přestoupit *na* tento spoj (respektive zastavení na spoji). Naopak z_1 nemusí mít čas výjezdu a v tom případě bychom C_{vz_1} jednoduše z rovnice vypustili.

2. z_2 má čas výjezdu, a nejde tedy o poslední zastavení jízdy. Pokud by šlo o poslední zastavení jízd, nemá cenu uvažovat o přestupu *na* tento spoj (respektive zastavení na spoji), protože v této stanici spoj končí. V tom případě by zastavení v rámci stanice bylo seřazené dle C_{pz_1} z toho důvodu, aby bylo možné přestoupit *z* tohoto spoje na další.

Popsanou situaci můžeme pozorovat na obrázku 3.2, kde vidíme znázorněná zastavení v rámci jedné stanice. Směrem *dolů* (po směru seřazení) je v rámci stanice přestup vždy možný, protože jsou zastavení seřazena dle C_z. Zatím žádným způsobem nepracujeme s minimálním časem na přestup.

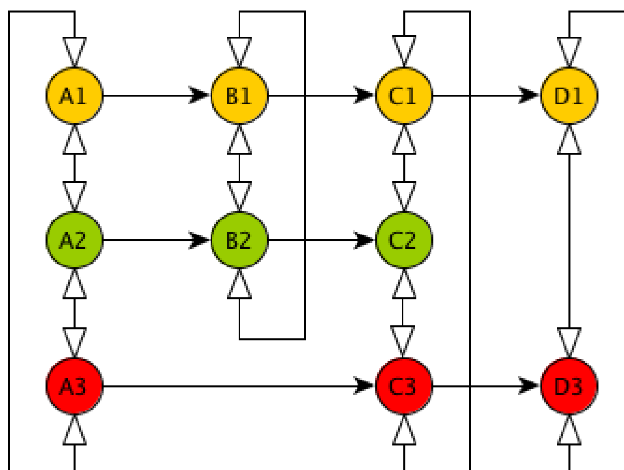


Obrázek 3.2: Zastavení v rámci jedné stanice. Čísla u vrcholů značí C_{pz} (vlevo) a C_{vz} (vpravo). Pro všechny C_z nyní platí $C_z = C_{vz}$.

Po bližším zkoumáním obrázku 3.2 si nicméně všimneme, že přestup je ve skutečnosti možný také mezi červeným a zeleným vrcholem. Tento přestup jde ovšem *proti* našemu seřazení dle C_z. V jízdních řádech jde sice o ne příliš častou, ale možnou situaci, na kterou musíme být připraveni. Řešením bude udělat všechny přestupní hrany obousměrné a až při vlastním vyhledáváním spojení dle parametrů kontrolovat, zda je přestup po *přestupní* hraně

možný. Koneckonců zkoumat, zda je přestup možný, budeme muset na jednotlivých zastaveních vždy, protože do vyhledávání zavedeme parametr *minimální čas na přestup* (MCP). Pokud bychom například měli $MCP = 5$, tak v situaci z obrázku 3.2 by nebyl možný přestup ze zeleného na červený vrchol.

Výslednou strukturu grafu obsahujícího všechna zastavení a potřebné hrany mezi nimi můžeme pozorovat na obrázku 3.3.

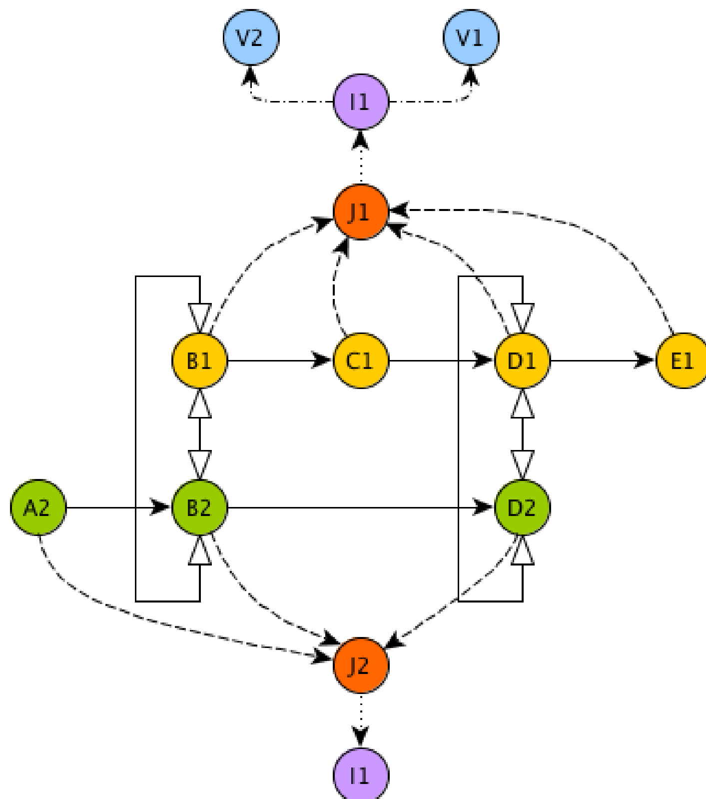


Obrázek 3.3: Základní struktura grafu. Stejnou barvou jsou obarvena zastavení v rámci jízdy a jsou propojena jednosměrnými *přejezdovými* hranami. Pod sebou jsou zaznamenána zastavení v rámci jedné stanice a jsou propojena obousměrnými *přestupními* hranami.

3.1.4 Kompletní schéma grafu

Ačkoliv už v grafu máme uložena všechna *zastavení* včetně nutných hran pro vyhledávání, chybí nám podstatné informace, které budeme pro vyhledávání také potřebovat. Zejména z konkrétního zastavení nevíme, ve kterých dnech je v spoj, kterému zastavení náleží, v provozu. Je tedy nezbytné doplnit do grafu *intervaly platnosti*, které se vážou na *jízdy*. Ty tedy budeme muset přidat také, abychom nezavedli do grafu přímé závislosti ze *zastavení* na *intervaly platnosti*, které v reálných datech nejsou. A nakonec budeme ještě potřebovat *výjimky z intervalu platnosti*, které přidáváme nově oproti bakalářské práci *BPJCH* [3].

Výslednou strukturu grafu, která bude pro naše účely vyhledávání považována za finální, můžeme pozorovat na obrázku 3.4. Z velké části je převzata z bakalářské práce *BPJCH* [3] a na předcházejících stranách této práce jsme si zejména v rychlosti ukázali, jak bylo výsledného schématu dosaženo. Současně jsme si ukázali, že zvolené schéma je založeno na vyzkoušeném a osvědčeném modelu, který byl popsán již v mnoha odborných publikacích. Pro naše účely jsme model pouze mírně upravili a rozšířili o další nutné vrcholy, které budeme potřebovat pro vyhledávání spojení.



Obrázek 3.4: Struktura grafu s přidanými vrcholy jízd (vrcholy J), intervalů platností (vrcholy I) a výjimek z intervalů platností (vrcholy V). Hrana znázorněná čárkovaně značí příslušnost zastavení k jízdě. Hrana znázorněná tečkovaně značí příslušnost jízdy k intervalu platnosti. Hrana znázorněná čerchovaně značí návaznost výjimky z intervalu platnosti na interval platnosti.

3.1.4.1 Atributy vrcholů grafu

Pro úplnost je ještě nutné zadefinovat si minimální množinu atributů jednotlivých typů vrcholů grafu, které budeme potřebovat pro vyhledávání:

- Zastavení
  - ID zastavení
  - ID stanice
  - Název stanice
  - Čas příjezdu
  - Čas výjezdu
  - Pořadí v rámci jízdy

  - Bezbariérovost zastavení (Zastavení označíme jako bezbariérové, pokud je bezbariérová stanice, na které se zastavení nachází, a současně je bezbariérové vozidlo jízdy, které toto zastavení náleží.)

- Jízdy
  - ID jízdy
- Intervaly platnosti
  - ID intervalu platnosti
  - Datum platnosti intervalu *OD*
  - Datum platnosti intervalu *DO*
  - Vlastní atributy pro jednotlivé dny v týdnu, které budou nabývat hodnot *true* nebo *false*, tedy atributy *Pondělí* - *Neděle*
- Výjimky z intervalu platnosti
  - ID výjimky
  - Datum výjimky
  - Typ výjimky (Buď *výjimečně jede*, nebo *výjimečně nejede*.)

3.2 Vyhledávací algoritmus

První verze vyhledávacího algoritmu nad grafovým schématem podobným schématu z kapitoly 3.1.4 byla již představena v rámci bakalářské práce *BPJCH* [3]. V této kapitole si tak vyhledávací algoritmus pouze v krátkosti představíme a rozšíříme o zatím neimplementované části. V průběhu představování algoritmu také zavedeme řadu změn a vylepšení, která v původní verzi algoritmu popsaném v *BPJCH* [3] nejsou a díky kterým budeme moci dosáhnout jak výrazného zrychlení algoritmu, tak i jeho lepší parametrizace o zatím neimplementované funkcionality.

Algoritmus bude prezentován pouze ve verzi vyhledávání dle času odjezdu z výchozí stanice. Verze algoritmu pro vyhledávání dle času příjezdu do cílové stanice je totiž velmi podobná, pouze ve své podstatě obrácená - traverzování probíhá opačným směrem. Současně budeme prezentovat pouze algoritmus pro vyhledávání v rámci jednoho dne, tedy nebudeme uvažovat vyhledávání přes půlnoc. To totiž do algoritmu přinese celou řadu výjimek, které by zde akorát algoritmus znepřehlednily.

Abychom v příštích kapitolách nemuseli opakovaně definovat uzly pro vyhledávání, zavedeme si hned zde v úvodu této sekce vyhledávací požadavek, kterým si budeme v následujících sekcích pomáhat pro znázornění funkčnosti algoritmu. Uvažujme tedy následující vstupní parametry pro vyhledávání, které již byly rozšířeny o parametry, které uživatel nezadává, ale ze zadaných parametrů je možné je snadno získat:

- Výchozí stanice (S_v) ... *Bělocerkevská*
- Cílová stanice (S_c) ... *Červeňanského*

- Datum výjezdu (D_v) ... 1.4.2017
- Čas výjezdu (C_v) ... 11:00
- Den výjezdu (Day_v) ... sobota
- Maximální počet přestupů (MPP) ... 3
- Minimální čas na přestup (MCP) ... 3 minuty
- Maximální datum příjezdu ($MaxD_p$) .. 1.4.2017
- Maximální čas příjezdu ($MaxC_p$) ... 15:00
- Maximální počet nalezených výsledků ($MaxResultsCount$) ... 3
- Pouze bezbariérová spojení ... ano

3.2.1 Nalezení výchozích zastavení

Samotný algoritmus je ve své podstatě poměrně jednoduchý. Jako u každého vyhledávání v grafu je nejprve nutné najít výchozí (počáteční) vrcholy, ze kterých vlastní traverzování započneme. Hledáme tedy vrcholy zastavení, pro které platí:

- Stanice = S_v
- Čas zastavení >= C_v *and* Čas zastavení < $MaxC_p$
- Zastavení je v platnosti ve dni D_v. K tomu je nutné dotraverzovat přes jízdu k intervalu platnosti a porovnat, zda je interval platný pro D_v. Tedy zkontrolovat, zda je datum v rozsahu *OD - DO* daného intervalu platnosti a zda je interval platný ve dni Day_v. Kontrola na den v týdnu navíc vyžaduje proiterovat i navázané výjimky z intervalu platnosti a zjistit, zda pro daný den náhodou není interval výjimečně platný, nebo výjimečně neplatný.
- Zastavení je bezbariérové

Všechna nalezená počáteční zastavení uložíme do prioritní fronty [14], ze které budeme v dalších krocích vrcholy vybírat. Detaily prioritní fronty si přiblížíme v dalších kapitolách. Poslední skutečností, kterou si musíme uvědomit, je, že traverzování ze všech výchozích zastavení bude nutně muset probíhat pouze přes *přejezdovou* hranu. Přes hrany přestupní bychom totiž pouze znovu dotraverzovali na zastavení, která jsou již obsažena v množině výchozích zastavení.

3.2.2 Traverzování do cílové stanice

Po nalezení množiny výchozích zastavení můžeme spustit traverzování grafem pro nalezení zastavení, která přísluší cílové stanici S_c. Přitom budeme chtít nalézt taková spojení, která nás dostanou z S_v do S_c:

- V co nejdřívějším čase příjezdu
- S co nejmenším počtem přestupů

Tento požadavek můžeme charakterizovat jako problém hledání nejkratších cest v grafu, kdy délka cesty je definována jako rozdíl času výjezdu z výchozího vrcholu a času příjezdu do aktuálního vrcholu. Protože navíc chceme preferovat cesty s co nejmenším počtem přestupů, zavedeme tzv. *penalizaci za přestup*, což bude určitý časový objem, který připočteme k aktuální délce cesty za každý uskutečněný přestup. Tím se cesta virtuálně prodlouží, a stane se tak méně výhodnou. Pokud pak mezi vrcholy x a y budou existovat dvě cesty a jedna z nich bude mít vyšší počet přestupů, bude z hlediska algoritmu považována za delší, protože k rozdílu času výjezdu z x a času příjezdu do y připočteme vyšší časový objem za více přestupů.

Pro hledání nejkratších cest v grafu slouží například Dijkstrův algoritmus, což je v současné době nejrychlejší známý algoritmus pro hledání všech nejkratších cest ze zadaného uzlu do všech ostatních uzlů grafu, který neobsahuje záporně ohodnocené hrany [12]. Pro naše účely není možné tento algoritmus přímo použít už jen z toho důvodu, že v našem grafu nemáme vůbec ohodnocené hrany, což je pro Dijkstrův algoritmus jeden ze základních předpokladů. V bakalářské práci *BPJCH* [3] byl ale navržen algoritmus, který je Dijkstrově algoritmu velmi podobný a pracuje s vlastní implementací prioritní fronty.

Nejprve si zavedeme funkci pro porovnání dvou vrcholů, kterou následně použijeme v prioritní frontě. Již jsme si řekli, že hledáme vrcholy cílové stanice s nejdřívějším časem příjezdu. Právě čas příjezdu bude naším prvním porovnávacím kritériem. Nicméně existují dvě výjimky. První výjimkou jsou výchozí vrcholy, ze kterých začínáme vyhledávání. U těch pro nás totiž nehraje čas příjezdu žádnou roli, naopak nás zajímá čas výjezdu. Druhou výjimkou budou všechna zastavení, která čas příjezdu nemají. U těch taktéž budeme nuceni sáhnout po času výjezdu. Obecně tento první parametr budeme značit C_{pvz}.

Pro porovnávání zavedeme ještě druhý atribut. Často se stane, že do jednoho zastavení dotraverzujeme různými cestami z rozdílných výchozích zastavení. V tom případě je pro nás velmi důležité ukončit další traverzování po zatím méně výhodné cestě a naopak pokračovat pouze v traverzování po cestě výhodnější. Tímto ukončováním traverzování méně výhodných cest ve výsledku nalezneme pouze všechny tzv. pareto-optimální cesty, což je přesně naším cílem. Pareto-optimální cestou rozumíme v tomto kontextu takovou cestu, která je *optimální* (v našem případě s nejkratší dobou jízdy mezi výchozí a cílovou stanicí a také co nejmenším počtem přestupů), ale její existence nevylučuje existenci cesty jiné, která je také *optimální.* Jinak řečeno, pareto-optimální je taková cesta mezi vrcholy A a B, která není *dominována* žádnou jinou cestou. Detailní a odborný náhled na pareto-optimální cesty můžeme nalézt například v práci [26].

Z výše uvedeného nám tedy vyplývají dvě základní kritéria, pomocí kterých budeme porovnávat vrcholy v rámci cesty a která budou součástí implementace naší prioritní fronty:

1. C_{pvz} (čas příjezdu do aktuálního vrcholu, případně čas výjezdu dle podmínek definovaných výše)
2. Délka cesty mezi výchozím a aktuálním vrcholem ve vteřinách navýšená o penalizace za každý uskutečněný přestup

3.2.2.1 Omezující podmínky

Další oblastí, kterou se při vyhledávání musíme zabývat, jsou podmínky, které musí nacházené cesty splňovat. Detailně jsou některé tyto podmínky uvedené v bakalářské práci *BPJCH* [3], zde je tak již nebudeme detailně popisovat, ale uvedeme jen ty z hlediska vyhledávání nejdůležitější a také změny a vylepšení, která jsme provedli oproti zmíněné bakalářské práci. V případě zájmu o detaily původních omezujících podmínek odkazujeme čtenáře na uvedenou publikaci.

Traverzování po *přejezdových hranách* je vždy bezproblémové, protože jakmile jsme se již jednou v rámci procházení grafem dostali na nějaký konkrétní spoj, tak pokračovat po jeho zastaveních dále je možné vždy. Problém tvoří traverzování po *přestupních hranách*. Před nebo po každém kroku přes takovou hranu je nutné kontrolovat řadu podmínek, mimo jiné například:

- **Platnost spoje, kterému zastavení náleží oproti dni vyhledávání** - Pokud po přesunu přes *přestupní* hranu zjistíme, že spoj, kterému patří aktuální zastavení, není v platnosti pro den vyhledávání, není možné traverzovat z tohoto zastavení *přejezdovou* hranou, protože spoj není v provozu. Naopak jedinou možností dalšího traverzování je použít *přestupní* hranu.
- **Překročení maximálního počtu přestupů** - Pokud v rámci cesty překročíme definovaný maximální počet přestupů, není již možné k traverzování využívat *přestupních* hran.
- **Dodržení minimálního počtu minut na přestup** - Pokud v rámci traverzování po *přestupní* hraně zjistíme, že čas příjezdu do aktuální stanice navýšený o minimální počet minut nutných na přestup je vyšší než čas výjezdu z aktuálního vrcholu, pak není možné pro další traverzování použít *přejezdovou* hranu, protože cestující by na daný spoj nestihl přestoupit. Naopak musíme dále traverzovat pouze po *přestupní* hraně.
- **Kontrola dodržování časového kontinua** - Předchozí podmínku je nutné navíc rozšířit o případy, kdy se po *přestupní* hraně v rámci stanice pohybujeme proti směru uspořádání zastavení na stanici. V tomto případě se velmi často stane, že čas odjezdu aktuálního zastavení bude dokonce menší než čas příjezdu do aktuální stanice. V tom případě je nutné traverzování po *přestupních* hranách proti směru uspořádání zastavení na stanici úplně ukončit, protože bychom traverzovali grafem do minulosti.
- **Kontrola bezbariérovosti zastavení** - Pokud vyhledáváme pouze bezbariérová spojení, je nutné před traverzováním po *přestupní* hraně zkontrolovat, zda je zastavení bezbariérové. Pokud by totiž nebylo, není možné z vozu vystoupit, a tudíž ani přestoupit na jiný spoj. Současně, pokud se na cestě již nacházíme ve vrcholu, do kterého jsme

se dostali *přestupní* hranou, je nutné kontrolovat, zda je aktuální zastavení bezbariérové. Pokud není, není možné pro další traverzování použít *přejezdovou* hranu, protože na daný spoj není možné nastoupit.

- **Kontrola překročení maximálního času příjezdu** - Tuto podmínku je nutné kontrolovat i po přesunu pomocí *přejezdové* hrany. Pokud má aktuální vrchol čas zastavení vyšší, než je $MaxC_p$, tak ukončíme traverzování touto cestou, protože výsledek v požadovaném čase již není možné na této cestě nalézt.

3.2.2.2 Optimalizace vyhledávacího algoritmu

Pro zrychlení vyhledávacího algoritmu je možné sáhnout k celé řadě optimalizačních kroků. Mimo jiné to jsou kontroly na:

- Vícenásobné přestupy mezi dvěma jízdami
- Traverzování mezi shodnými stanicemi

Tyto byly z větší části popsány v již zmíněné bakalářské práci *BPJCH* [3] a zde nebudou znovu uváděny. Jedinou výjimku tvoří postup pro hledání pouze všech pareto-optimálních cest, který je pro rychlé fungování algoritmu naprosto nezbytný a byl oproti zmíněné bakalářské práci upraven.

Již víme, že vrcholy budeme postupně procházet v pořadí udaném prioritní frontou, která bude implementovat vlastní porovnávání vrcholů. V průběhu vyhledávání si každý již navštívený vrchol označíme jako *navštívený*. Pokud v budoucnu navštívíme takovýto vrchol znovu, víme, že další traverzování danou cestou můžeme ukončit, protože tato cesta již zcela určitě nebude lepší než cesta, se kterou jsme vrchol navštívili dříve. Tento fakt přímo plyne z definice námi představené prioritní fronty - dříve totiž zpracujeme vrchol na výhodnější cestě, což je v tomto případě ta, která je z výchozího vrcholu cesty rychlejší po započtení penalizací za přestupy.

Díky pevně danému pořadí zpracovávání uzlů přes námi definovanou prioritní frontu navíc víme, že první nalezený výsledek je ten nejlepší možný, druhý nalezený výsledek je druhý nejlepší možný atd. Tento fakt přímo plyne z toho, že nejlepší výsledek je pro nás takový, který je v cílové stanici co nejdříve, kdy délku cesty navyšujeme o penalizace za přestup. Z tohoto zjištění pro nás vyplývá, že jakmile nalezneme takový počet výsledků, který odpovídá $MaxResultsCount$, tak můžeme vyhledávání již ukončit, protože k nalezení lepších výsledků v budoucnu už zcela jistě dojít nemůže.

3.2.2.3 Finální náhled na vyhledávací algoritmus

Se znalostmi z předchozích sekcí si konečně můžeme představit vyhledávací algoritmus kompletně. V algoritmu jsou vynechány optimalizační metody, které nebyly v rámci této práce popsány a jsou k dispozici pouze v bakalářské práci *BPJCH* [3].

1. Najdi všechna počáteční zastavení a vlož je do prioritní fronty.

2. Vyber vrchol z prioritní fronty. Pokud ve frontě již není žádný vrchol, ukonči algoritmus.

3. Pokud je vrchol označený jako *navštívený*, ukonči traverzování touto cestou a jdi na krok 2.

4. Označ vrchol jako *navštívený*.

5. Pokud je čas zastavení aktuálního vrcholu vyšší než je čas $MaxC_p$, ukonči traverzování po této cestě a pokračuj krokem 2.

6. Pokud je poslední hrana na cestě *přestupní* a aktuální čas zastavení je nižší, než je čas příjezdu na aktuální stanici, ukonči vyhledávání touto cestou a pokračuj krokem 2.

7. Pokud se aktuální vrchol (zastavení) nalézá na cílové stanici, ukonči traverzování aktuální cestou a ulož výsledek. Pokud již počet uložených výsledků odpovídá požadovanému počtu výsledků, ukonči celý algoritmus. Jinak pokračuj dále krokem 2. Pokud hledáme pouze bezbariérová spojení, je možné tento krok provést pouze za předpokladu, že je aktuální zastavení bezbariérové.

8. Dále následuje rozhodnutí, po kterých hranách je možné dále traverzovat. Výběr hran pro traverzování odpovídá první možnosti, která pro aktuální vrchol platí:

   (a) Pokud je aktuální zastavení úplně první v rámci cesty, je možné dále traverzovat pouze po *přejezdové* hraně.

   (b) Pokud je poslední hrana v rámci cesty *přestupní* a aktuální vrchol zastavení není pro dnešní den v platnosti, je možné dále traverzovat pouze po *přestupní* hraně.

   (c) Pokud je poslední hrana v rámci cesty *přestupní* a aktuální vrchol zastavení má čas zastavení nižší, než je čas příjezdu do aktuální stanice navýšený o minimální počet minut na přestup, pak je možné dále traverzovat pouze po *přestupní* hraně.

   (d) Pokud byl v rámci aktuální cesty již překročen maximální počet přestupů, je možné dále traverzovat pouze po *přejezdové* hraně.

   (e) Pokud hledáme pouze bezbariérová spojení a poslední hrana byla *přejezdová* a současně aktuální zastavení není bezbariérové, je možné pokračovat pouze po *přejezdové hraně.*

   (f) Neplatí ani jedno z rozsahu (a) - (e). Traverzování tedy může pokračovat jak po hraně *přestupní*, tak po hraně *přejezdové.*

9. Proiteruj všechny vrcholy dostupné přes hrany určené v kroku 8 a přidej je do prioritní fronty. Dále pokračuj krokem 2.

Právě představená zjednodušená verze vyhledávacího algoritmu nalezne celkově nejvýše $MaxResultsCount$ prvních pareto-optimálních výsledků z výchozí do cílové stanice při splnění všech požadavků na vyhledávání, jako jsou *maximální počet přestupů*, *minimální čas na přestup* či *maximální čas příjezdu.*

Kapitola 4

Návrh

V minulých kapitolách jsme provedli detailní analýzu nároků a požadavků na námi konstruovanou webovou i mobilní aplikaci. V této kapitole se o trochu více přiblížíme vlastní konstrukci aplikací a provedeme komplexní návrh všech důležitých součástí aplikací tak, aby následná implementace v konkrétně zvolených technologiích byla co nejpřímočařejší.

4.1 Datové úložiště

Z analýzy již víme, že pro vyhledávání spojů budeme data o jízdních řádech ukládat v grafové databázi Neo4j, která je k danému účelu vhodná, a již jsme si i představili kompletní schéma, které budeme v této databázi využívat. Pozornému čtenáři ale jistě neuniklo, že databáze nepočítá s ukládáním kompletního výčtu entit z datového balíku GTFS a současně zatím nemáme kam ukládat ani například informace o uživatelích a jejich rolích.

Nyní máme na výběr z několika možností, jak kompletní data ukládat. Jednou možností by bylo ukládat data kompletně do grafové databáze Neo4j. Každá entita by byla reprezentována vrcholem příslušného typu a vazby mezi entitami bychom řešili hranami konkrétního typu. Tento přístup by byl samozřejmě možný, ale je nutné si uvědomit, že grafovou databázi budeme chtít využívat zejména pro vyhledávání, kvůli kterému ji do našeho projektu vůbec chceme integrovat. Tento typ databáze naopak není vhodné používat pro případy, kdy nám mnohem lépe poslouží databáze relační[1]. Kdybychom aplikaci stavěli bez požadavku na vyhledávání, jistě bychom právě po relační databázi sáhli nejdříve, protože v jejím schématu můžeme velmi jednoduše a transparentně ukládat data, která získáváme ve formátu GTFS.

Je očividné, že jednotlivé soubory GTFS balíku dat je možné přímo namapovat na tabulky v relační databázi. Pokud navíc budeme ukládat data o jízdních řádech do takovéto databáze, bude pro nás velmi jednoduché importovat i exportovat data ve formátu GTFS. Import bude možné provádět soubor po souboru, kdy při importu jednoho souboru nám bude stačit zapisovat pouze do jedné tabulky bez nutnosti čtení dat z tabulky jiné. Při exportu naopak bude postačovat číst data pouze z konkrétní tabulky, jejíž data zrovna budeme exportovat do *csv* souboru datového formátu GTFS. Je nutné si uvědomit, že tato výhoda by při použití výhradně grafové databáze zcela odpadla. Při propojování vrcholů hranou je totiž

[1] https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

nutné vrcholy nejdříve z databáze vytáhnout a teprve poté je mezi ně možné hranu vložit [15]. Naproti tomu cizí klíč v relační databázi můžeme vložit přímo, pokud již samozřejmě existuje jeho vlastník v příslušné tabulce. U exportu by byl problém podobný - v relační databázi můžeme cizí klíče entity exportovat rovnou, vrchol grafu ale žádné cizí klíče neobsahuje a je nutné po příslušných hranách nalézt navázané entity a z nich teprve příslušný klíč přečíst [18]. Cizí klíče bychom si sice mohli na vrcholy ukládat přímo, ale tím bychom se dopustili de facto duplikace existujících hran, které mají vazby mezi entitami primárně zaznamenávat.

Naopak nevýhoda ukládání všech dat do relační databáze je také zřejmá - data, která potřebujeme i pro vyhledávání, což jsou ve skutečnosti prakticky pouze *zastavení jízdy*, budeme mít najednou uložená na dvou místech. To může vést k případným nekonzistencím, pokud nesprávně vyřešíme synchronizaci mezi oběma databázemi. Tento přístup ale i přes možné problémy použijeme, protože se jeví jako nejlepší možný. Nejenže výrazně usnadní a zrychlí import a export dat o jízdních řádech, ale umožní nám i snadnější implementaci správy jízdních řádů přirozeně ukládaných v souladu s formátem GTFS. Využití různých typů datových úložišť pro účely, ke kterým jsou vhodné, je navíc i již dobře známý a popsaný přístup, viz například *polyglot persistence* [7].

4.2 Rozdělení dat do schémat

Data o jízdních řádech tedy budeme primárně ukládat do relační databáze, a navíc data nutná pro vyhledávání v jízdních řádech budeme ukládat v již dříve specifikovaném formátu do grafové databáze Neo4j. Protože naše aplikace musí umožňovat ukládat data z více jízdních řádů a následně v nich i vyhledávat, je nutné určit způsob, kterým toho docílíme.

Jednou z možností by bylo do každé tabulky (respektive ke každému vrcholu v grafu) přidat identifikátor jízdního řádu, ke kterému daný záznam patří. Tento identifikátor by se pak pravděpodobně stal součástí primárního klíče záznamů. Každý dotaz do databáze by pak tento identifikátor vyžadoval, aby bylo možné vybrat data ze zvoleného jízdního řádu. V grafové databázi by nám pak pro každý nový jízdní řád vznikla samostatná *komponenta souvislosti*[2] (bez ohledu na orientaci hran), která by s jinými jízdními řády nebyla propojena žádnou hranou. Uvedený přístup by byl zcela validní, v rámci naší aplikace nicméně navrhneme zcela jiné řešení, které bude mít sice několik nevýhod, ale současně pro i nás velmi důležité výhody.

Hlavní motivací zvolit jiný přístup, než jsme právě nastínili, je možnost lepšího *škálování*[3] zejména grafové databáze. Chceme-li být schopni v aplikaci ukládat a zejména vyhledávat nad velkým počtem různých jízdních řádů, musíme začít uvažovat o výpočetních nárocích, které to s sebou přináší, a to zejména s ohledem na možnost budoucího navýšení výpočetní kapacity. Neo4j sice poskytuje možnosti pokročilého škálování, tzv. *High availability*[4], ale ty jsou dostupné pouze ve zpoplatněné enterprise verzi, a v rámci této práce je tak není možné využívat.

Podrobnější analýzou je ale možné nalézt uspokojivé řešení. Uložením každého jízdního řádu do vlastní databáze, která může být nasazena na dedikovaném serveru, dosáhneme

[2] http://algoritmy.eu/zga/pruchod-grafu/komponenty-souvislosti/
[3] https://msdn.microsoft.com/cs-cz/library/aa292203(v=vs.71).aspx
[4] https://neo4j.com/blog/neo4j-scalability-infographic/

toho, že bude možné přidávat nové jízdní řády prakticky neomezeně. Přidáváním nových jízdních řádů se nám tak nebude zvyšovat celková velikost jedné centrální databáze a žádným způsobem nebude ovlivněna ani rychlost vyhledávání nad ostatními jízdními řády právě díky tomu, že každý bude mít vyhrazenu vlastní databázi a vlastní výpočetní výkon. Velmi efektivně pak budeme schopni reagovat i na vytíženosti konkrétních jízdních řádů. K tomuto kroku nás vede i fakt, že databáze Neo4j má poměrně vysoké nároky na výkon a v případě, že se celý graf vejde do operační paměti, je následné vyhledávání v něm nepoměrně rychlejší, než když se musí jednotlivé části grafu načítat do operační paměti postupně v průběhu času z pevného disku[5].

Při použití uvedeného postupu současně nebude nutné u všech typů záznamů v databázi ukládat atribut s identifikátorem jízdního řádu, ke kterému entita patří. Jelikož už budeme mít data rozdělená po jízdních řádech v grafové databázi, nic nám nebrání rozdělit je i v databázi relační. K tomuto účelu použijeme *schémata*[6], která většina rozšířených relačních databází podporuje. K ukládání dat do zcela oddělených databází v tomto případě neshledáváme důvod, protože nad relační databází neočekáváme zvýšené požadavky na výpočetní výkon.

Výhody zvoleného přístupu jsme již uvedli, v rámci zachování objektivity je ale nutné zmínit i nevýhody zvoleného řešení. Navržený postup netriviálně zvyšuje nároky na údržbu aplikace, protože každý jízdní řád s sebou ponese vlastní grafovou databázi, respektive vlastní schéma relační databáze. Současně je očividné, že přidání nového jízdního řádu bude vyžadovat vytvoření nové grafové databáze a její nasazení, čímž se přidávání jízdních řádů do aplikace poměrně znepříjemňuje. I tento proces by bylo možné teoreticky automatizovat, v rámci této práce ale nebudeme tuto funkcionalitu implementovat.

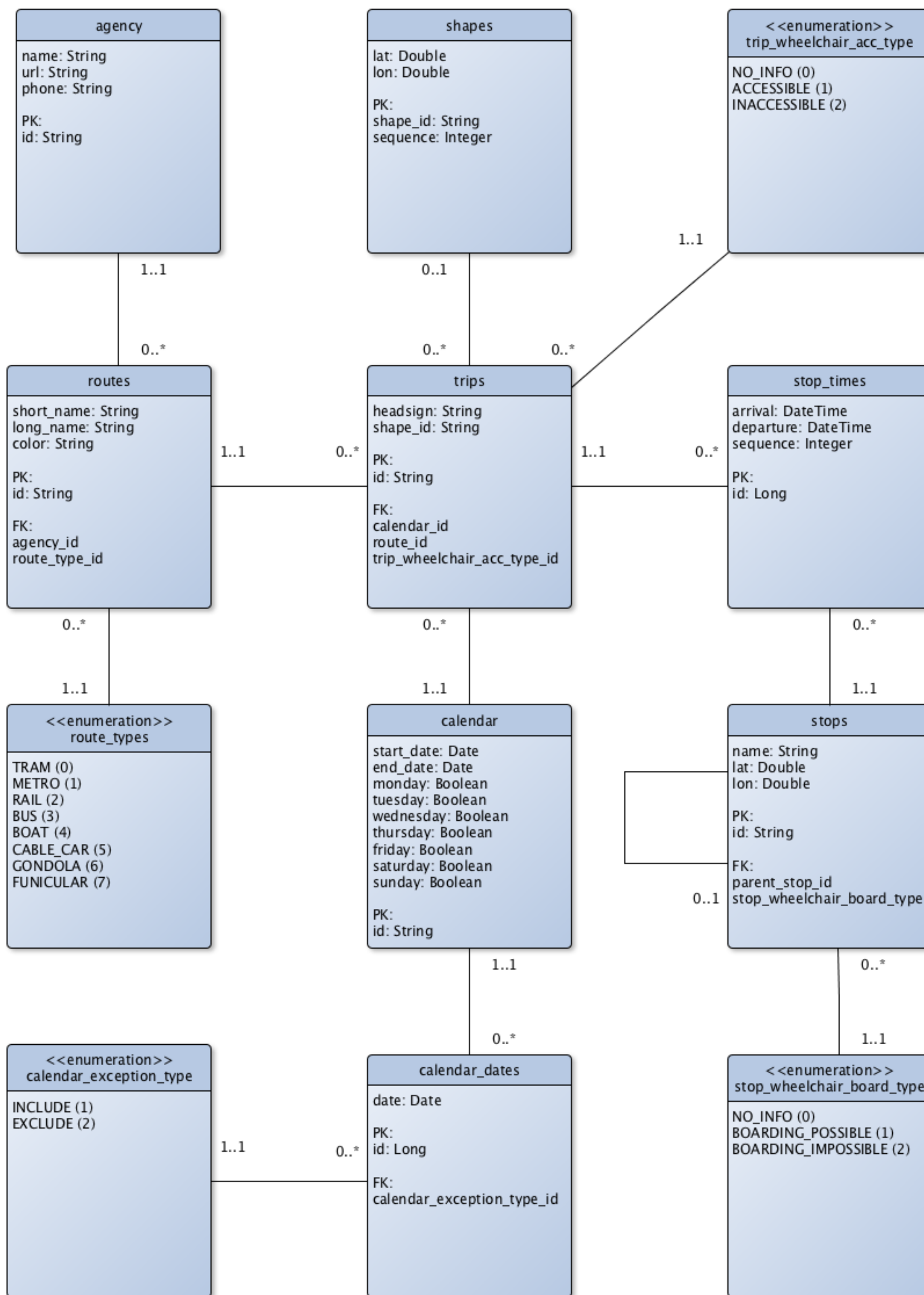
4.3 Datový model relační databáze

Protože jsme si již detailně navrhli jednotlivé aspekty ukládání dat, je načase navrhnout kompletní schéma relační databáze, které bude naše aplikace využívat. Schéma bude kopírovat strukturu GTFS formátu dat, a to v rozsahu, který v naší aplikaci budeme využívat. Jak jsme si již řekli dříve, díky dodržování GTFS formátu se nám jednak velmi usnadní import a export jízdních řádů, současně je ale schéma velmi vhodné pro celkový koncept aplikace.

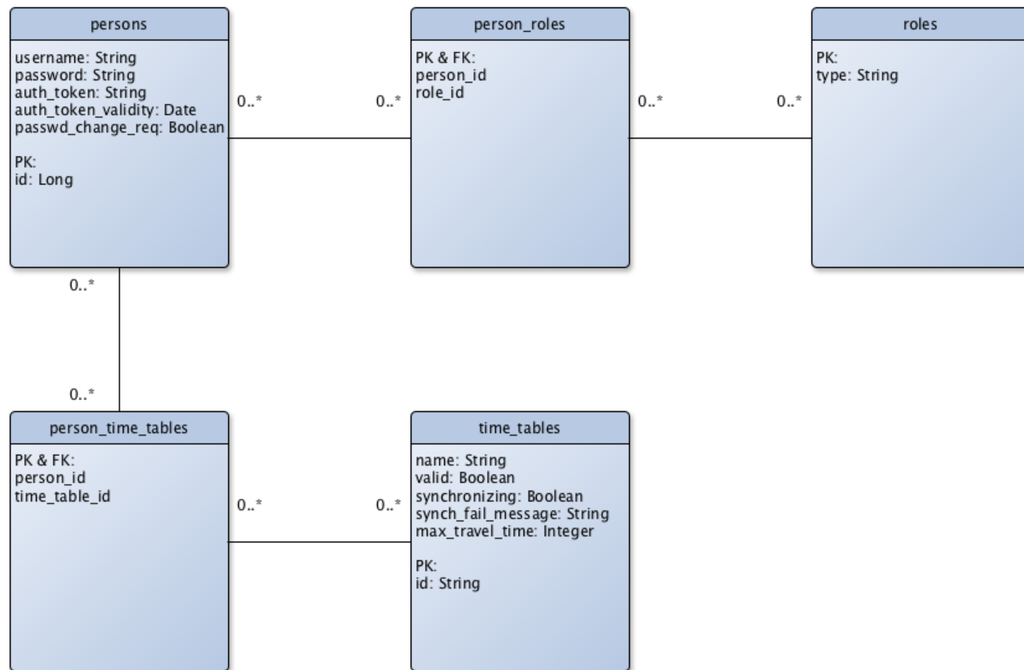
Kromě vlastních dat o jízdních řádech, která budeme ukládat do vlastních schémat, je nutné ještě ukládat data o uživatelích, jejich rolích a jízdních řádech, které budou mít uživatelé právo spravovat. Tyto údaje budeme ukládat do speciálního schématu.

Schéma databáze pro ukládání dat o jízdních řádech můžeme pozorovat na obrázku 4.1. Schéma pro ukládání ostatních údajů poté můžeme pozorovat na obrázku 4.2.

[5] https://neo4j.com/docs/operations-manual/current/installation/requirements/
[6] https://docs.oracle.com/cd/B19306_01/server.102/b14220/schema.htm



Obrázek 4.1: Schéma dat o jízdních řádech. Každý jízdní řád bude uložen v samostatném schématu.



Obrázek 4.2: Schéma dat o uživatelích.

4.4 Architektura

Naší aplikaci budeme budovat striktně oddělenou na serverovou a klientskou část. Server bude poskytovat *REST API (rozhraní)*, za kterým bude schována kompletní business logika aplikace. S tímto rozhraním poté bude komunikovat jednak statický javascriptový klient (dle požadavků psaný ve frameworku Angular 2) a také mobilní aplikace pro operační systém Android.

4.4.1 REST

V poslední době velmi rozšířeným přístupem pro návrh distribuovaných prostředí typu klient-server je použití *REST (Representational State Transfer)*[7]. Jedná se o sadu principů a postupů, jejichž hlavním cílem je popsat metodu přístupu k datům. REST rozhraní zpravidla využívá protokol HTTP[8], ačkoliv to není podmínkou a REST obecně není na tento protokol striktně vázán. Správně navržené REST rozhraní by mělo mimo jiné být kompletně *bezestavové*. Každý požadavek tedy musí obsahovat všechny informace nutné k jeho vykonání. Jinými slovy server si neuchovává žádné informace mezi jednotlivými požadavky.

REST definuje čtyři základní metody pro přístup k tzv. *resourcům* (zdrojům). Zdrojem v kontextu této práce budeme rozumět zejména data, která jsou zpravidla reprezentována ve

[7] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[8] https://www.w3.org/Protocols/HTTP/1.1/rfc2616bis/draft-lafon-rfc2616bis-03.html

formátu JSON[9] nebo XML[10]. Každý zdroj má vlastní identifikátor, který označujeme jako *URI*. Zmíněné čtyři základní metody přístupu k datům jsou:

- **GET** - metoda pro přístup k datům (získání zdroje)
- **POST** - metoda pro vytvoření dat (vytvoření zdroje) - V těle požadavku se zpravidla odesílají data, která se mají uložit.
- **PUT** - metoda pro úpravu dat (úpravu zdroje) - Na rozdíl od metody POST posíláme zpravidla v těle požadavku data, která se mají uložit k již existujícímu zdroji.
- **DELETE** - metoda pro smazání zdroje

4.4.2 Popis REST rozhraní

Pro možnost implementace REST rozhraní je nutné navrhnout jednotlivé zdroje, které budeme poskytovat pro práci s daty. Zejména tedy navrhnout jejich identifikátor (*uri*) a také definovat formát dat, který budeme pro komunikaci využívat.

Velmi dobrým zvykem při tvorbě každého API je hned od začátku zavést jeho verzování. Je očividné, že veškeré vytvořené rozhraní by mělo být vždy zpětně kompatibilní, a není tak možné bez varování kdykoliv změnit například formát očekávaných dat nebo identifikátory zdrojů. Na ty už totiž může být navázaná celá řada externích systémů, které by v případě takové změny přestaly okamžitě fungovat. Správný postup v případě nutné změny rozhraní je vytvořit novou verzi. Číslo verze rozhraní tedy bude přímo součástí všech identifikátorů zdrojů. Základní identifikátor tak v námi budované první verzi aplikace vždy bude:

`/api/v1/`

Z kapitoly 4.2 již víme, že každý jízdní řád bude uložen v samostatném schématu relační databáze. Pro každý dotaz na data je nutné nějak předat informaci, se kterým jízdním řádem chceme pracovat. Tuto informaci umístíme také přímo do *uri* zdrojů ve formě ID jízdního řádu. Aby bylo zcela očividné, že se opravdu jedná o ID jízdního řádu, umístíme před vlastní ID vždy prefix *x-*. Například pro jízdní řád s *ID = "pid"* tak bude základní identifikátor všech zdrojů ve tvaru:

`/api/v1/x-pid/`

4.4.2.1 Formát dat

V rámci komunikace přes REST rozhraní budeme používat výlučně formát JSON zejména kvůli jeho jednoduchosti a možnosti přímého mapování do javascriptových objektů. Vlastní implementaci je ale nutné provést tak, aby bylo do budoucna v případě potřeby možné jednoduše přidat podporu i pro jiné formáty, zejména XML. Volba požadovaného formátu dat by pak měla probíhat pomocí HTTP hlavičky *Content-type* dle principu *Content Negotiation*[11].

[9] https://www.w3schools.com/js/js_json_intro.asp
[10] https://www.w3schools.com/xml/xml_whatis.asp
[11] https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html

4.4.2.2 Zabezpečení

Při návrhu REST rozhraní je velmi důležité správně navrhnout také zabezpečení jednotlivých zdrojů z důvodu prevence neoprávněných přístupů k operacím s daty. Pro zabezpečení rozhraní máme na výběr z několika možností.

Nejjednodušší je pravděpodobně použití tzv. *Basic access authentication*[12]. Autentizace uživatele a jeho následná autorizace[13] na serveru probíhá pomocí HTTP hlavičky *Authorization*, ve které se při každém požadavku pošle uživatelské jméno a heslo zakódované pomocí algoritmu *Base64*[14]. Kódování je nutné zejména s ohledem na ošetření nepovolených znaků pro přenos pomocí protokolu HTTP a v žádném případě neslouží například k šifrování hesla. Naopak řetězec zakódovaný pomocí algoritmu Base64 je možné velmi snadno dekódovat a získat tak heslo uživatele. Z tohoto důvodu je bezpodmínečně nutné míst REST rozhraní vystavené přes zabezpečený protokol HTTPS. To už ale máme vynucené v nefunkčních požadavcích z kapitoly 2.6, žádný problém zde tedy nenastává.

Na tomto místě je obecně dobré si uvědomit, že i kdybychom heslo již posílali v zašifrované podobě (například v podobě *hash* otisku), tak v případě přenosu dat přes nezabezpečený protokol HTTP sice útočník nebude moci přečíst uživatelovo heslo v otevřeném formátu, ale stejně bude schopný se v rámci aplikace za uživatele vydávat, protože bude moci velmi snadno zopakovat požadavek se stejnou autorizační hlavičkou. REST rozhraní by tedy mělo být vystavené vždy výhradně přes zabezpečený protokol HTTPS.

Pro zabezpečení naší aplikace sáhneme po mírně elegantnějším řešení, než je Basic access authentication. Autentizace a následná autorizace uživatele bude probíhat pomocí vlastní hlavičky *X-Auth*. Uživatel bude v hlavičce zasílat svůj token, který obdrží po přihlášení. Token bude mít platnost třicet minut a platnost se automaticky prodlouží při každém přijatém požadavku s tímto tokenem. V případě nového přihlášení uživatele pomocí jména a hesla dojde ke ztrátě dříve vygenerovaných (i platných) tokenů. Pokud tedy bude uživatel přihlášen na jednom zařízení a přihlásí se navíc na jiném, automaticky dojde k jeho odhlášení na prvním zařízení. Očekávaný proces přihlášení tedy bude následující:

1. Uživatel zašle na autorizační endpoint své validní uživatelské jméno a heslo.
2. Aplikace vygeneruje autorizační token, který vrátí v odpovědi klientovi. Token bude mít platnost třicet minut.
3. Uživatel nyní bude obdržený token zasílat v *X-Auth* hlavičce s každým požadavkem. V případě provedení požadavku s validním tokenem se jeho platnost automaticky prodlouží tak, že bude právě třicet minut od posledního požadavku.

Právě navržené řešení má pro naši aplikaci několik nezanedbatelných přínosů oproti použití *Basic access authentication* (dále jen *Base auth*), kterých chceme využít. Za prvé nám toto řešení díky nastavené platnosti autorizačního tokenu poskytne de facto automatické odhlašování, které je pro naši aplikaci žádoucí. Odhlášením v kontextu naší aplikace rozumíme zneplatnění tokenu - server si neuchovává žádný stav a token je nutné posílat s každým

[12] https://tools.ietf.org/html/rfc2617
[13] https://technet.microsoft.com/cs-cz/library/ff687657(v=ws.10).aspx
[14] http://www.herongyang.com/Encoding/Base64-Encoding-Algorithm.html

requestem. Za druhé nebude nutné v klientské aplikaci uchovávat uživatelovo heslo, které zadá v přihlašovacím formuláři, ale bude se uchovávat pouze token. V případě chybné implementace klienta a zneužití této chyby útočníkem je mnohem menším bezpečnostním rizikem získání náhodně vygenerovaného autorizačního tokenu s omezenou platností než hesla, které uživatel může používat pro přihlašování i do jiných aplikací. V neposlední řadě nám automatické zneplatnění starého tokenu v případě nového přihlášení poskytne také námi požadovanou funkcionalitu. Protože se uživatelé do administrace budou hlásit pouze z webové aplikace, není žádný důvod k tomu, aby byli přihlášeni z více zařízení současně. Toto nicméně nebude zcela vyloučené, pokud si uživatel bude vygenerovaný token spravovat manuálně.

4.4.2.3 Ukázky rozhraní

Nyní si představíme kompletní rozhraní pro obsluhu CRUD operací nad stanicemi jízdního řádu. Popis kompletního veřejného i zabezpečeného API je k dispozic na Apiary[15] [16], a nebude zde tak uváděn.

Popis	Vytvoření stanice
HTTP metoda	POST
URI	/api/v1/x-id/admin/stop

Tabulka 4.1: Definice vytvoření stanice.

```
+ Request [POST /api/v1/x-id/admin/stop]
    Content-type: application/json
    X-Auth: auth_token
    {
        "id": "XCZZR",
        "name": "Muzeum - A",
        "lat": 50.035375,
        "lon": 14.335876,
        "wheelChairCode": 0
    }
+ Response (201)
    Content-type: application/json;charset=UTF-8
    Location: /api/v1/x-id/admin/stop/XCZZR
    {
        "id": "XCZZR",
        "name": "Muzeum - A",
        "lat": 50.035375,
        "lon": 14.335876,
        "wheelChairCode": 0
    }
```

[15] http://docs.dpnssapi.apiary.io/
[16] http://docs.dpnssprivateapi.apiary.io/

Popis	Získání stanice
HTTP metoda	GET
URI	/api/v1/x-id/admin/stop/{id}

Tabulka 4.2: Definice získání stanice dle ID.

```
+ Request [GET /api/v1/x-id/admin/stop/XCZZR]
    Accept: application/json
    X-Auth: auth_token
+ Response (200)
    Content-type: application/json;charset=UTF-8
    {
        "id": "XCZZR",
        "name": "Muzeum - A",
        "lat": 50.035375,
        "lon": 14.335876,
        "wheelChairCode": 0,
        "parentStopId": null
    }
```

Popis	Smazání stanice
HTTP metoda	DELETE
URI	/api/v1/x-id/admin/stop/{id}

Tabulka 4.3: Definice smazání stanice dle ID.

```
+ Request [DELETE /api/v1/x-id/admin/stop/XCZZR]
    X-Auth: auth_token
+ Response (204)
```

Popis	Update stanice
HTTP metoda	PUT
URI	/api/v1/x-id/admin/stop/{id}

Tabulka 4.4: Definice updatu stanice dle ID.

```
+ Request [PUT /api/v1/x-id/admin/stop/XCZZR]
    Content-type: application/json
    X-Auth: auth_token
    {
        "name": "Muzeum - B",
        "lat": 50.035375,
        "lon": 14.335876,
        "wheelChairCode": 1
    }
+ Response (200)
    Content-type: application/json;charset=UTF-8
    {
        "id": "XCZZR",
        "name": "Muzeum - B",
        "lat": 50.035375,
        "lon": 14.335876,
        "wheelChairCode": 1,
        "parentStopId": null
    }
```

4.5 Připojení k databázi Neo4j

Pro připojení k databázi Neo4j z aplikace máme k dispozici prakticky tři možnosti:

- Spuštění databáze v tzv. *embedded* módu
- Komunikace pomocí REST rozhraní
- Komunikace pomocí protokolu *Bolt*

V rámci bakalářské práce *BPJCH* [3] byla vytvořena i jednoduchá aplikace, přes kterou bylo možné provést vlastní vyhledávání spojů nad databází Neo4j. Tato aplikace využívá právě spuštění databáze v embedded módu[17]. V tomto módu aplikace přistupuje přímo na souborový systém databáze, ze kterého čte veškerá data o vrcholech a hranách. Výhodou je velká rychlost přístupu k datům a zejména možnost použití *traversal frameworku*[18], který je nezbytný pro implementaci našeho vyhledávacího algoritmu. Bohužel v případě spuštění databáze ve standalone módu se databáze zamkne pro aplikaci, která ji spustila, a není možné se k databázi připojit z aplikace jiné. Toto je pro nás zcela nevhodné chování, protože k jedné databázi se budeme chtít připojovat až z n instancí aplikace (Protože webová aplikace má být dle požadavků z kapitoly 2.6 horizontálně škálovatelná.). I v případě použití sdíleného souborového systému tedy není kvůli zamčení databáze možné embedded módu využít.

Další možností je tedy spuštění databáze v tzv. *standalone módu* a komunikace pomocí REST rozhraní[19]. V tomto případě je databáze spouštěná například jako samostatný proces

[17] https://neo4j.com/developer/java/
[18] https://neo4j.com/docs/java-reference/current/
[19] https://neo4j.com/developer/java/

vystavující právě REST rozhraní, se kterým aplikace komunikuje. Ačkoliv je zatím možné přes REST rozhraní využívat *traversal frameworku*, šlo by v našem případě o velmi nešťastné rozhodnutí. Samotné traverzování grafem totiž probíhá postupně dle implementovaného algoritmu, a v případě navštívení vrcholu a zjišťování jeho sousedů propojených hranou tak vždy dojde k zavolání nového REST požadavku [17]. Protože databáze jízdních řádů mohou mít řádově i miliony vrcholů, byl by tento přístup velmi nevhodný kvůli zpoždění, které by při traverzování nutně nastávalo kvůli prodlevám při komunikaci přes protokol HTTP.

Poslední uvedenou možností je komunikace s databází Neo4j pomocí binárního protokolu Bolt[20]. V tomto případě je databáze také spuštěná ve *standalone módu*, ale pro komunikaci s aplikací se místo REST rozhraní přes protokol HTTP používá právě uvedený protokol Bolt. V případě komunikace pomocí tohoto protokolu ovšem není možné využívat *traversal framework* [17], který bezpodmínečně potřebujeme. Nemožnost využití *traversal frameworku* při komunikaci přes Bolt zavedli vývojáři databáze Neo4j schválně a to právě ze stejných důvodů, ze kterých je nevhodné používat *traversal framework* i přes rozhraní REST.

4.5.1 Serverové rozšíření databáze

Sice jsme si představili možnosti připojení k databázi Neo4j, bohužel pro nás není zatím ani jedna možnost použitelná. Neo4j nicméně naštěstí podporuje také tvorbu tzv. *server extensions* [17]. Jedná se vlastně o samostatný java archiv[21] (jar), který může obsahovat libovolné množství procedur, které se budou vykonávat přímo na serveru, na kterém běží databáze Neo4j ve *standalone módu*. Archiv je nutné nahrát do složky *plugins* souborové struktury Neo4j a každá procedura musí být označena pomocí anotace:

```
@Procedure(name = "example.procedure.name")
```

Pomocí jména uvedeného v anotaci je pak možné tuto proceduru zavolat například pomocí jazyka *Cypher*[22]:

```
CALL example.procedure.name()
```

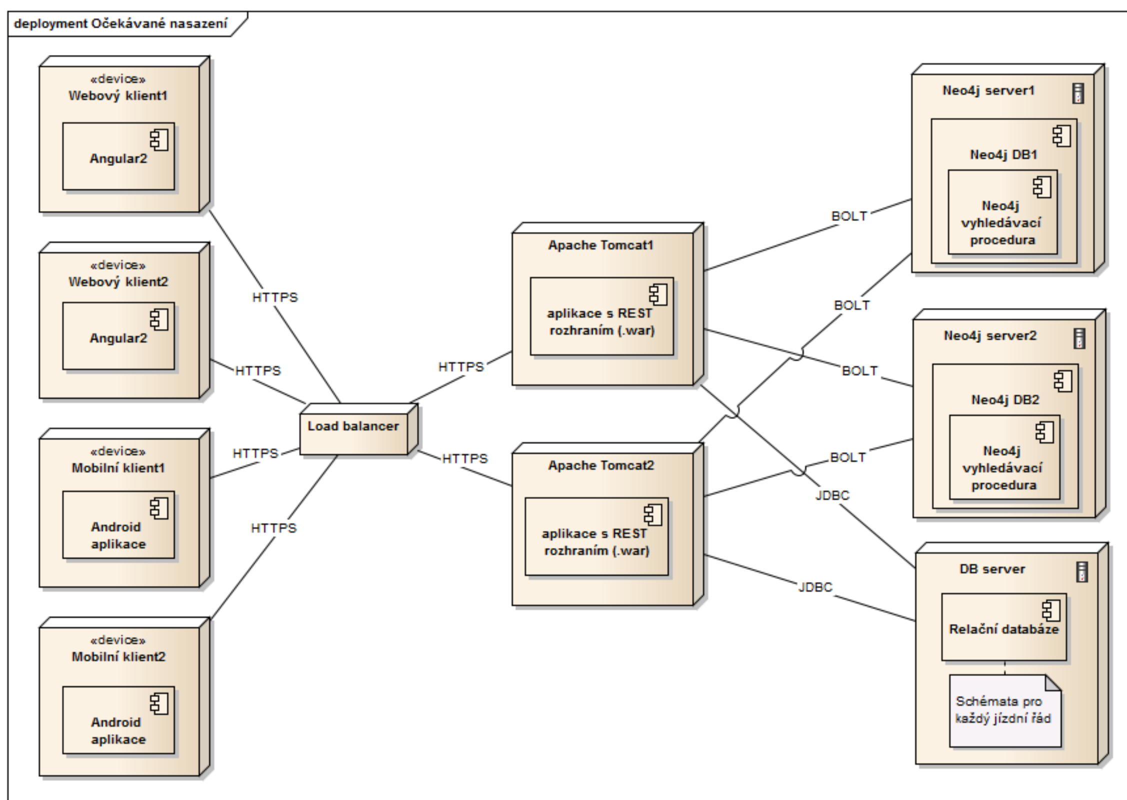
Toto je přesně řešení, které využijeme. Vlastní logika vyhledávání, a tedy využití *traversal frameworku* pro implementaci vyhledávacího algoritmu dle kapitoly 3, bude implementována jako *server extension*, a bude se tak provádět přímo na serveru, na kterém bude nasazena databáze Neo4j. Z naší aplikace pak budeme pouze přes protokol Bolt volat proceduru, která provede toto vlastní vyhledávání na serveru a vrátí nalezené výsledky. Díky tomuto přístupu budou splněny obě naše podmínky, a to nutnost použití *traversal frameworku* a současně možnost připojení více instancí aplikace na jednu instanci databáze Neo4j.

[20] http://neo4j.com/docs/java-reference/current/
[21] http://www.oracle.com/technetwork/java/archive-139210.html
[22] https://neo4j.com/developer/cypher-query-language/

4.6 Zamýšlené nasazení

Již dříve v této práci jsme uvedli, že současně bude moci běžet až n instancí aplikace (horizontální škálování), každý jízdní řád bude uložen v samostatné databázi Neo4j, která také poběží na samostatném serveru, a také, že budeme využívat i relační databázi se schématem pro každý jízdní řád. Na tomto místě je tak vhodné navrhnout celkové schéma zamýšleného nasazení hotové aplikace včetně popisu, jakým způsobem spolu budou jednotlivé komponenty komunikovat.

K tomuto účelu bylo vytvořeno očekávané schéma výsledného nasazení, které můžeme pozorovat na obrázku 4.3.



Obrázek 4.3: Model očekávaného nasazení

Load balancer[23] slouží pro rovnoměrné rozptýlení zátěže mezi jednotlivé instance aplikace. Každý klientský požadavek je tedy nejprve směřován na něj a následně přesměrován na uzel, který load balancer interně vybere dle algoritmu, který využívá.

Všechny instance databáze Neo4j i všechny instance aplikace by měly být umístěny v *Docker kontejneru*. Docker[24] je platforma, která umožňuje vytvářet *image*[25], které je ná-

[23] https://www.nginx.com/resources/glossary/load-balancing/
[24] https://www.docker.com/what-docker
[25] https://docs.docker.com/engine/getstarted/step_two/

sledně možné spustit jako *kontejnery*[26] na libovolném cílovém zařízení. Na rozdíl od virtuálního stroje[27] například využívají jádro hostitelského zařízení, díky čemuž jsou mnohem menší a startují rychleji než právě virtuální stroje.

V případě správné konfigurace docker image bude možné velmi snadno startovat nové instance databáze Neo4j i samotné aplikace. Databáze Neo4j poskytuje podporu pro docker standardně [16], v našem případě ale nebude možné jednoduše použít oficiální Neo4j Docker image a pouze jej nastartovat na požadovaném serveru, protože mimo jiné k databázi Neo4j potřebujeme přidat již dříve zmíněné rozšíření, které bude sloužit pro vyhledávání spojů. Image, který pak budeme na každém novém serveru pro Neo4j spouštět, tedy budeme muset vytvořit sami a toto rozšíření do něj přidat. Po vytvoření image jej pak bude nicméně možné spouštět fakticky jedním parametrizovaným příkazem *docker run*[28], a nasazování databází Neo4j pro nové jízdní řády tak bude velmi jednoduché, čímž alespoň částečně eliminujeme nevýhody řešení zvoleného v kapitole 4.2.

S ohledem na nasazování aplikace pak bude nutné vytvořit vlastní *docker image*, který bude obsahovat zejména aplikační server (případně pouze servlet kontejner jako například Apache Tomcat[29]) a vlastní aplikaci. Spuštěním tohoto image by pak měl server nastartovat a provést deploy aplikace dle konfigurace. Požadovaný cílový stav je takový, aby bylo možné image spouštět zcela automaticky, čímž bude zajištěna podpora pro automatické škálování aplikace, které bude reagovat na aktuální zatížení aplikace.

4.7 Návrh vzhledu

V rámci návrhu byla vytvořena i sada návrhů obrazovek (wireframy[30]) jak webové, tak mobilní aplikace, dle kterých by měla vzniknout výsledná implementace. Návrhy zachycují vybrané nejdůležitější obrazovky, což je zejména vyhledávací formulář a zobrazení výsledků vyhledávání. Mimo to ale zachycují i vybrané obrazovky z administrace jízdních řádů. Jednotlivé návrhy můžeme pozorovat na obrázcích 4.4 - 4.7.

[26] https://docs.docker.com/engine/getstarted/step_two/
[27] https://en.wikipedia.org/wiki/Virtual_machine
[28] https://docs.docker.com/engine/reference/run/
[29] http://tomcat.apache.org/
[30] http://jecas.cz/wireframe

NSS

https://dp-nss.cz

Jízdní řád:

Pražská integrovaná doprava (PID)

Odkud: Dejvická

Kam: Karlovo náměstí

Přes: Můstek

Datum: 17.4.2017

Čas: 12:00

Odjezd Příjezd

Maximální počet přestupů: 3

Pouze bezbariérové:

Hledat

Obrázek 4.4: Vyhledávací formulář.

NSS

https://dp-nss.cz

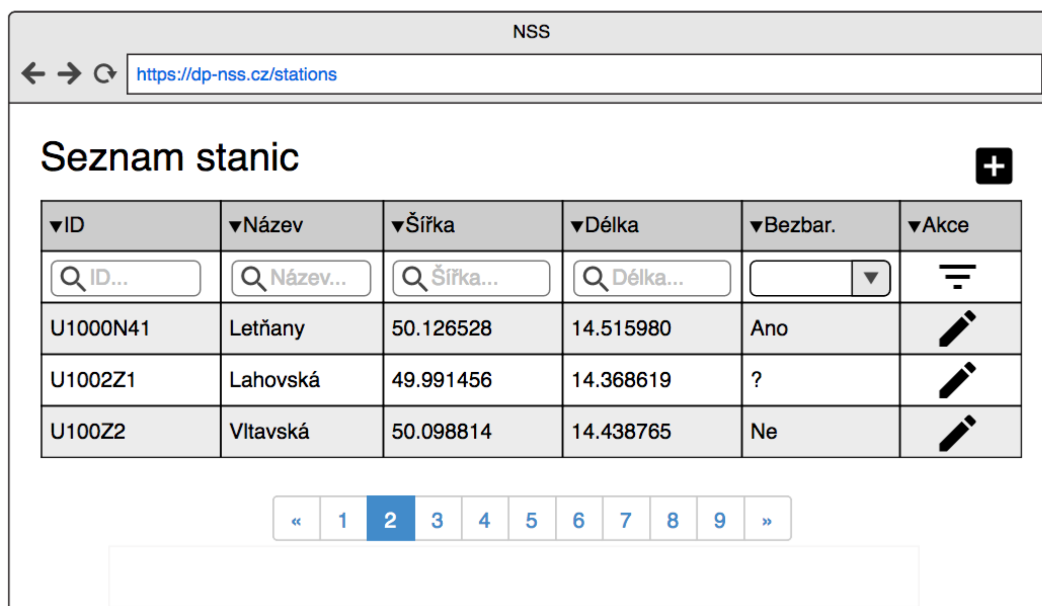
Nalezené výsledky

Předchozí

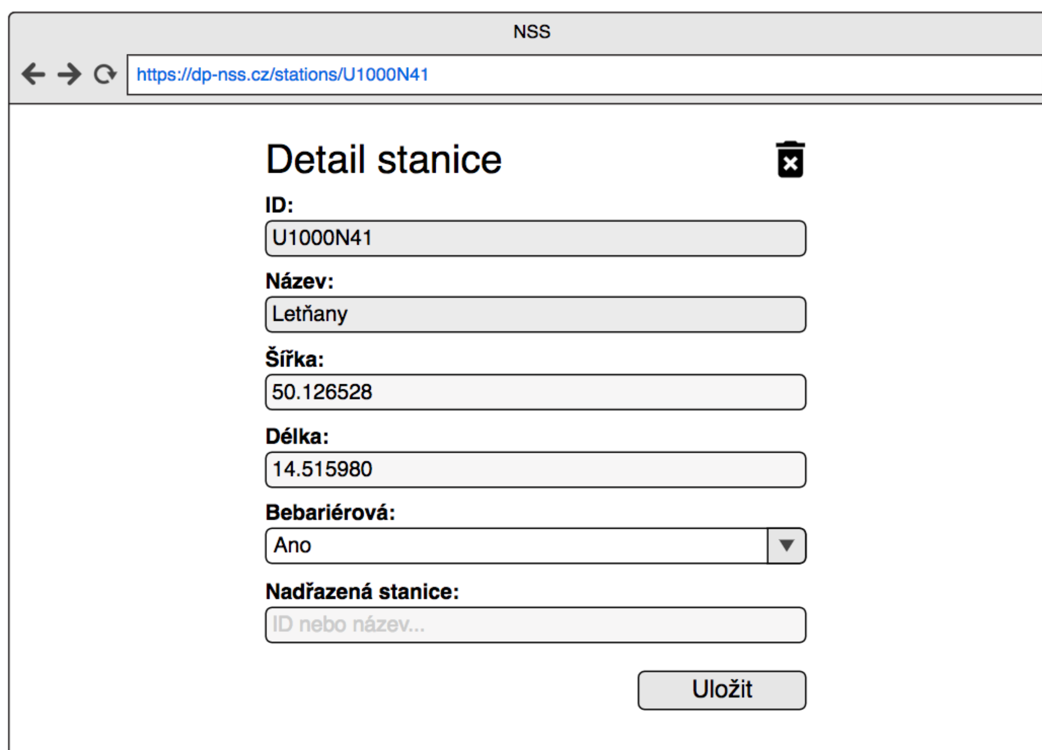
Datum	Stanice	Příjezd	Odjezd	Spoj
17.04.2017	Dejvická		12:06:15	Metro A
	Můstek - A	12:12		
	Můstek - B		12:16	Metro B
17.04.2017	Karlovo náměstí	12:18:15		

Další

Obrázek 4.5: Výsledky vyhledávání.



Obrázek 4.6: Seznam a vyhledávání stanic.



Obrázek 4.7: Detail a editace stanice.

Kapitola 5

Implementace

Díky provedené analýze a návrhu z předchozích kapitol můžeme nyní přejít k samotné implementaci webové i mobilní aplikace. V této kapitole si tak představíme zvolené technologie a zajímavé implementační detaily.

5.1 Zvolené technologie

Z nefunkčních požadavků z kapitoly 2.6.1 jsme vázáni tím, že serverová část webové aplikace bude psána v jazyce Java za použití frameworku Spring a klientská část aplikace bude psána ve frameworku Angular 2. Mobilní aplikace pak bude dle požadavků cílena na platformu Android. Na jednotlivé technologie se nyní podíváme.

5.1.1 Java

Jazyk Java[1] je velmi rozšířený silně typový programovací jazyk. Mimo jiné je díky kompilaci do bytekódu platformově nezávislý.

Speciálně pro tvorbu webových aplikací vznikla Java EE (enterprise edition)[2], což je rozšíření standardní edice Java SE (standard edition)[3], které podporuje tvorbu velmi robustních, škálovatelných, vysoce zabezpečených a vícevrstvých webových aplikací. Díky těmto vlastnostem se velmi hodí pro tvorbu velmi komplexních či kritických aplikací.

5.1.2 Spring

Spring[4] je velmi populární framework, který výrazným způsobem zjednodušuje tvorbu zejména webových aplikací v jazyce Java díky existenci velkého množství existujících komponent a knihoven, které může vývojář využít. V naší aplikaci budeme mimo jiné chtít využít Dependency injection[6] pomocí Spring IOC (inversion of control)[5] kontejneru. K implementaci dalších požadavků pak budeme moci využít jiné komponenty frameworku Spring.

[1] http://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html
[2] http://www.oracle.com/technetwork/java/javaee/overview/index.html
[3] http://www.oracle.com/technetwork/java/javase/overview/index.html
[4] https://spring.io/
[5] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html

Například pro tvorbu REST API se výborně hodí Spring MVC Framework[6] a pro zpracování velkého množství dat (import a export jízdních řádů) budeme moci využít například knihovnu Spring Batch[7]. Z těchto důvodů byl zvolen právě framework Spring, který se pro naše účely perfektně hodí.

5.1.3 Správa závislostí a sestavení aplikace

Pro správu závislostí a sestavení webové aplikace budeme využívat Apache Maven[8], což je nástroj pro správu softwarových projektů. Principem systému Maven je tvorba objektového modelu nad zdrojovým kódem, který řeší zejména způsob kompilace a parametrizaci sestavení aplikace do balíčků či správu externích závislostí aplikace (dotahování správných verzí externích knihoven apod.).

Vážným konkurentem systému Maven je zejména v poslední době buildovací nástroj Gradle[9]. Tento nástroj využijeme při implementaci mobilní aplikace pro platformu Android.

5.1.4 Relační databáze PostgreSQL

Při výběru konkrétní implementace relační databáze máme na výběr mezi řadou open-source řešení, jako je například databáze MySQL[10], SQLite[11] či PostgreSQL[12].

V aplikaci jsme se rozhodli využít právě relační databázi PostgreSQL, která je jednou z nejrozšířenějších open-source relačních databází vůbec a také je mezi těmito databázemi považována za nejstabilnější a nejvýkonnější [13].

5.1.5 Hibernate

Pro mapování dat mezi aplikací a databází budeme používat Hibernate ORM[13] (objektově relační mapování). Hibernate je pravděpodobně nejrozšířenější implementací Java Persistent Api (JPA), což je framework jazyka Java umožňující objektově relační mapování, díky kterému budeme moci po správné konfiguraci automaticky mapovat data z relační databáze na objekty programovacího jazyka, čímž si ušetříme spoustu práce a zbytečného kódu.

5.1.6 Klientská aplikace

Při tvorbě klientské části webové aplikace jsme vázáni tím, že musí být napojena na REST API, které jsme si již dříve popsali. Toho bychom mohli docílit například i přímým umisťováním JavaScriptového[14] kódu přímo do HTML šablon. Práci bychom si pak mohli

[6] https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html
[7] http://projects.spring.io/spring-batch/
[8] https://maven.apache.org/
[9] https://gradle.org/
[10] https://www.mysql.com/
[11] https://www.sqlite.org/
[12] https://www.postgresql.org/
[13] http://hibernate.org/orm/
[14] https://www.javascript.com/

trochu usnadnit například použitím velmi rozšířené knihovny jQuery[15], která výrazným způsobem ulehčuje zejména manipulaci s tzv. Document Object Modelem (DOM)[16] stránky, ale také usnadňuje volání Ajax[17] požadavků na server či zpracování událostí.

Nezanedbatelnou nevýhodou tohoto řešení je zejména skutečnost, že vývojář je zodpovědný za propagaci dat, která obdrží ze serveru, do HTML šablony, čímž se náročnost vývoje poměrně komplikuje. Vývojář totiž musí ručně manipulovat s DOMem HTML stránky, což může být v budoucnu velmi náchylné na změny struktury této stránky.

Naštěstí existují řešení, která vývoj javascriptových aplikací výrazným způsobem usnadňují. Je to zejména knihovna React[18] od vývojářů Facebooku[19] a framework Angular[20] vyvinutý společností Google[21]. V naší aplikaci jsme se rozhodli využít framework Angular, který byl při začátku implementace ve verzi 2.

5.1.6.1 Angular 2

Angular 2 je javascriptový webový framework určený pro tvorbu tzv. single-page[22] aplikací. Framework je založený na komponentové architektuře, která při správném použití prakticky odpovídá architektuře MVC[23]. Striktně jsou tak oddělena data (model), jejich grafická reprezentace (view) a vlastní logika aplikace (controller).

Angular 2 poskytuje celou řadu funkčností, které obrovským způsobem ulehčují vývoj. Například standardně vynucuje použití vlastního řešení Dependency injection[6] či poskytuje rozhraní pro volání HTTP požadavků a odpověď umí automaticky mapovat na javascriptové objekty. Klíčovou funkčností je pak například tzv. *two-way data binding*[24], který zajišťuje automatickou synchronizaci dat mezi modelem a view (HTML šablonou). Vývojář tak již nemusí ručně manipulovat s DOMem HTML stránky.

5.1.7 Mobilní aplikace

Mobilní aplikaci budeme dle nefunkčních požadavků z kapitoly 2.6.2 implementovat pro operační systém Android[25] od verze 4.2. Android je celosvětově bezkonkurenčně nejrozšířenější operační systém pro mobilní zařízení [27], rozhodnutí implementovat mobilní aplikaci právě pro tuto platformu je tak logické. Rozhodnutím podporovat systém od verze 4.2 pak pokryjeme naší aplikací cca 95 % všech zařízení vybavených tímto operačním systémem [8] a současně nebudeme muset podporovat zastaralé API dřívějších verzí, což považujeme za velmi dobrý kompromis.

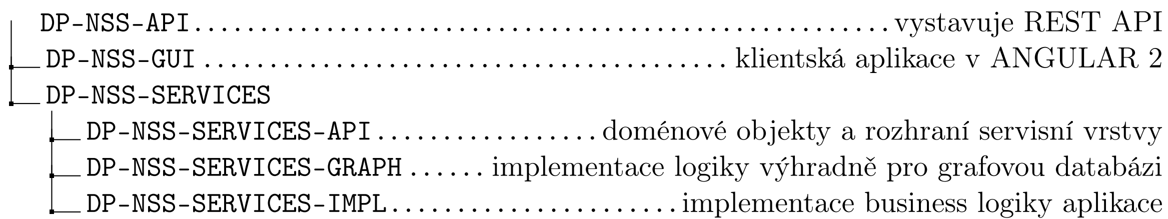
[15] https://jquery.com/
[16] https://www.w3schools.com/js/js_htmldom.asp
[17] https://www.w3schools.com/xml/ajax_intro.asp
[18] https://facebook.github.io/react/
[19] https://www.facebook.com/
[20] https://angular.io/
[21] https://www.google.com/
[22] http://jecas.cz/spa
[23] https://developer.chrome.com/apps/app_frameworks
[24] https://blog.thoughtram.io/angular/2016/10/13/two-way-data-binding-in-angular-2.html
[25] https://www.android.com/

5.2 Detaily implementace

V této kapitole si představíme vybrané části zdrojových kódů, které realizují některé definované požadavky na naši aplikaci či architektonická rozhodnutí z kapitol 2 - 4. Cílem není prezentovat kompletní implementovanou logiku, kterou si může čtenář dohledat přímo ve zdrojových kódech, ale ukázat zajímavé detaily z implementace.

5.2.1 Struktura aplikace

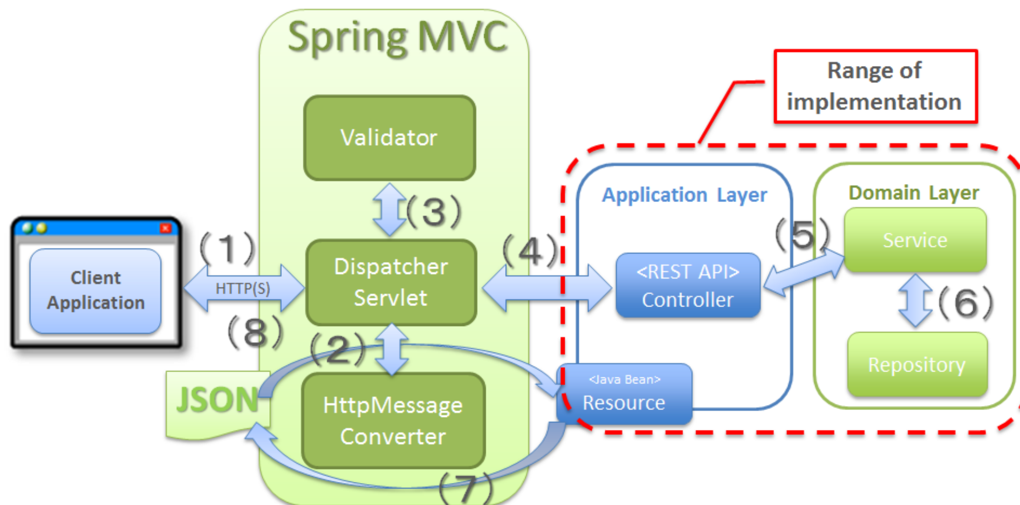
Aplikace je členěna do několika logicky oddělených samostatných modulů, které obsahují vždy konkrétní část aplikace. Díky modularizaci je dosaženo výrazně lepší udržovatelnosti a budoucí rozšiřitelnosti aplikace. Při sestavení aplikace pomocí nástroje Apache Maven se pak moduly *poskládají* do výsledného webového archivu, který bude možné nasadit na zvolený aplikační server či servlet kontejner, jako je například Apache Tomcat[26].



- DP-NSS-API .... vystavuje REST API
- DP-NSS-GUI .... klientská aplikace v ANGULAR 2
- DP-NSS-SERVICES
  - DP-NSS-SERVICES-API .... doménové objekty a rozhraní servisní vrstvy
  - DP-NSS-SERVICES-GRAPH .... implementace logiky výhradně pro grafovou databázi
  - DP-NSS-SERVICES-IMPL .... implementace business logiky aplikace

Obrázek 5.1: Struktura aplikace - rozdělení do modulů.

Architektura aplikace je pak tvořena několika vrstvami, které spolu komunikují. Tyto vrstvy můžeme pozorovat na obrázku 5.2.



Obrázek 5.2: Architektura a vrstvy aplikace. Zdroj: [19]

[26] http://tomcat.apache.org/

Proces zpracování požadavku je pak následující:

1. Klientská aplikace (webová nebo mobilní) pošle požadavek na konkrétní endpoint REST API pomocí protokolu HTTPS. Požadavek zachytí *Dispatcher Servlet*[27], což je Java servlet[28] z knihovny Spring MVC, který v našem případě slouží jako *Front controller*[29].
2. Pokud požadavek obsahuje tělo (body), knihovna Spring MVC za nás automaticky provede konverzi z formátu JSON na objekt programovacího jazyka Java.
3. Pokud máme definované validátory na příchozí objekt, tak je knihovna Spring MVC automaticky provede a v případě selhání validace vrátí příslušný chybový kód.
4. Dispatcher servlet přepošle zprávu na námi implementovaný konkrétní *Controller*[30], který poslouchá na adrese, na kterou přišel klientský požadavek.
5. Ve většině případů bude controller volat servisní vrstvu aplikace k provedení konkrétní business logiky.
6. V případě databázové operace zavolá servisní vrstva vrstvu datovou, která provede konkrétní logiku s daty v databázi (relační nebo grafové) nebo například data z databáze pouze vrátí ve formě doménového objektu.
7. Pokud z našeho controlleru vracíme klientovi v odpovědi i data (tělo), tak je za nás knihovna Spring MVC automaticky zkonvertuje do formátu JSON.
8. Odpověď se přes protokol HTTPS odešle zpět klientovi.

5.2.2 REST API a Spring MVC

Jak již bylo řečeno dříve, k implementaci REST API dle požadavků budeme využívat knihovnu Spring MVC, díky které budeme MVC architekturu i dodržovat. V kapitole 4.4.2.3 jsme si představili ukázky REST rozhraní, na tomto místě tak ukážeme jejich implementaci.

Pro CRUD operace nad stanicemi dle kapitoly 4.4.2.3 vytvoříme controller, který bude tyto operace implementovat:

```
@RestController
@RequestMapping(value = "/admin/stop")
public class AdminStopController {

    @Autowired
    private StopService stopService;

    ...
}
```

[27] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html
[28] http://docs.oracle.com/javaee/7/api/javax/servlet/Servlet.html
[29] https://martinfowler.com/eaaCatalog/frontController.html
[30] https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

Jednotlivé CRUD metody pak budou definovány pro odpovídající HTTP metodu:

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public StopWrapper getStop(@PathVariable("id") String id) {
    Stop stop = stopService.get(id);
    if(stop == null) throw new ResourceNotFoundException();

    return getStopWrapper(stop);
}

@RequestMapping(method = RequestMethod.POST)
public ResponseEntity<StopWrapper> createStop(
        @RequestBody StopWrapper wrapper) throws BadRequestException {

    Stop stop = getStop(wrapper);
    stopService.create(stop);

    return getResponseCreated(getStopWrapper(
        stopService.get(stop.getId())));
}

@RequestMapping(value = "/{id}", method = RequestMethod.PUT)
public StopWrapper updateStop(@PathVariable("id") String id,
        @RequestBody StopWrapper wrapper)
        throws ResourceNotFoundException, BadRequestException {

    Stop existingStop = stopService.get(id);
    if(existingStop == null) throw new ResourceNotFoundException();

    Stop stop = getStop(wrapper);
    stopService.update(stop, true);
    return getStopWrapper(stop);
}

@RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
public void deleteStop(@PathVariable("id") String id)
        throws BadRequestException {

    Stop stop = stopService.get(id);
    //ok, jiz neni v DB
    if(stop == null) return;

    if(!stopService.canBeDeleted(stop.getId())) {
        throw new BadRequestException("stop can not be deleted");
    }

    stopService.delete(stop.getId());
}
```

5.2.3 Zabezpečení REST API

V kapitole 4.4.2.2 jsme navrhli zabezpečení našeho REST API pomocí autorizačního tokenu, který se bude posílat s každým HTTP požadavkem v hlavičce *X-Auth*. K realizaci tohoto požadavku jsme se rozhodli implementovat vlastní anotaci, kterou budeme u zabezpečených metod používat. Parametrem anotace bude seznam uživatelských rolí, se kterými je možné zabezpečenou metodu zavolat. Implementace pak musí před vykonáním zabezpečené metody přečíst autorizační hlavičku z HTTP požadavku a zkontrolovat, zda uživatel, kterému patří, má požadovanou roli pro provedení metody.

Nejprve si ukážeme vlastní anotaci:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Inherited
@Documented
public @interface CheckAccess {

    /**
     * nutne role pro pristup
     */
    Role.Type[] value() default {Role.Type.USER};

}
```

A její předpokládané použití:

```
@CheckAccess(value = Role.Type.ADMIN)
@RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
public void deleteStop(@PathVariable("id") String id) {
    ...
}
```

Anotaci samozřejmě můžeme definovat pouze jednou v rámci třídy, čímž se stanou všechny metody zabezpečenými. Dokonce můžeme anotaci umístit na abstraktního předka všech zabezpečených controllerů, čímž na jednom místě můžeme zabezpečit celou administraci jízdních řádů. Tohoto řešení také v implementaci reálně využíváme, jak je zřejmé ze zdrojových kódů aplikace.

Zbývá si představit vlastní implementaci logiky zabezpečení. V našem případě jsme se rozhodli vytvořit *interceptor*[31], který před vykonáním těla každé zabezpečené metody provede kontrolu uživatelských rolí dle autorizační hlavičky. Zjednodušený kód můžeme pozorovat níže:

```
public class SecurityInterceptor implements HandlerInterceptor {

    @Autowired
    protected PersonService personService;

    public static final String SECURITY_HEADER = "X-Auth";

```

[31] http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html

```
    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object o) throws Exception {

        //ma tato metoda/trida anotaci CheckAccess?
        CheckAccess methodAnnotation =
            getMethodAnnotation((HandlerMethod) o, CheckAccess.class);

        //nezabezpecena metoda
        if(methodAnnotation == null) return true;

        String securityHeader = request.getHeader(SECURITY_HEADER);
        if(securityHeader == null || securityHeader.length() == 0) {
            throw new UnauthorizedException();
        }

        //pole roli, ktere jsou nutne pro tento resource
        Role.Type[] rolesNeeded = methodAnnotation.value();
        //aktualni uzivatele
        Person person = personService.getByToken(securityHeader);
        if(person != null) {
            //admin ma pravo automaticky na vsechno
            if(person.hasRole(Role.Type.ADMIN)) return true;

             //najdeme, jestli uzivatel nema pozadovanou roli
            for(Role.Type roleNeeded : rolesNeeded) {
                if(person.hasRole(roleNeeded)) return true;
            }
        }

        throw new UnauthorizedException();
    }

    ...
}
```

5.2.4 Výběr konkrétního schématu databáze

Jak jsme si řekli v kapitole 4.2, jednotlivé jízdní řády budeme ukládat do samostatných Neo4j databází, respektive do samostatných schémat relační databáze PostgreSQL. Dle návrhu REST rozhraní z kapitoly 4.4.2 již dále víme, že identifikátor požadovaného jízdního řádu bude přímo součástí URI jednotlivých zdrojů. Nyní si představíme řešení, pomocí kterého této požadované funkcionality dosáhneme.

V první řadě je nutné po obdržení REST požadavku zjistit, s jakým jízdním řádem chce uživatel pracovat, a tuto informaci si *někam* uložit tak, abychom ji mohli před vlastním čtením z databáze použít. Jedním z možných řešení by bylo všem metodám, které nějakým způsobem čtou či zapisují jízdní řády do databáze, přidat parametr *ID jízdního řádu.* V tom případě bychom pak tento parametr museli propagovat prakticky skrz všechny vrstvy aplikace, aby doputoval přes servisní vrstvu až do vrstvy datové, kde je vlastní práce s databází

řešena. Tento postup je možný, ale poměrně nepraktický, protože vyžaduje psaní spousty zbytečného kódu. Využijeme tedy jiné, lepší řešení.

Ihned po obdržení klientského požadavku přečteme z URI zdroje identifikátor jízdního řádu a uložíme jej do *Thread local*[32] proměnné. Hodnota této proměnné pak bude dostupná vždy jen z vlákna, ve kterém jsme hodnotu do proměnné uložili. Protože jeden požadavek budeme zpracovávat většinou pouze v jednom vlákně, můžeme tento přístup bez problémů použít. V případě, že v rámci požadavku budeme chtít volat další akci v novém vlákně, tak do něj hodnotu Thread local proměnné překopírujeme.

K uložení proměnné do thread local využijeme *servlet filter*[33], který bude odchytávat veškeré požadavky na naše REST rozhraní a bude se vykonávat před vlastním zavoláním konkrétního controlleru (viz dokumentace). Filtr definujeme v souboru *web.xml*:

```
<filter>
    <filter-name>schemaHandlerFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>schemaHandlerFilter</filter-name>
    <url-pattern>/api/v1/*</url-pattern>
</filter-mapping>
```

A níže můžeme pozorovat vlastní implementaci filtru ve zjednodušené formě:

```
public class SchemaHandlerFilter implements Filter {

    ...

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp,
            FilterChain chain) throws IOException, ServletException {

        HttpServletRequest httpRequest = (HttpServletRequest) req;
        //vytahnu si URI, na kterou prisel dotaz
        String requestURI = httpRequest.getRequestURI();

        //zjistim, jestli URI obsahuje identifikator existujiciho
        //jizdniho radu (napr. x-pid)
        String schema = getSchemaFromURI(requestURI);
        if(schema != null) {
            //nastavim schema do thread local
            SchemaThreadLocal.set(schema);

            //vytvorim URI, ktera jiz neobsahuje schema
            String newUri = getURIWithoutSchema(requestURI);

            //a provedu redirect (ten chytne prislusny controller)
            httpRequest.getRequestDispatcher(newUri).forward(req, resp);
```

[32] https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html
[33] http://www.oracle.com/technetwork/java/filters-137243.html

```
            //po provedeni requestu smazu schema z threadLocal
            SchemaThreadLocal.unset();
            return;
        }

        //k presmerovani nedoslo tak necham request bezet normalne
        chain.doFilter(req, resp);
    }

    ...
}
```

5.2.4.1 Výběr schématu relační databáze

Nyní zbývá zajistit, abychom četli a zapisovali do správné databáze (schématu) dle právě uloženého identifikátoru jízdního řádu. Jak již víme, pro práci s relační databází používáme framework Hibernate, který podporuje tzv. *Multi-tenancy*[34]. To je funkcionalita, kterou přesně potřebujeme, protože umožňuje připojovat se k více databázím, případně schématům databází, současně. Stačí tak implementovat rozhraní *CurrentTenantIdentifierResolver* a *MultiTenantConnectionProvider* a tyto implementace následně poskytnout v definici *SessionFactory*[35]. Zjednodušené implementace obou rozhraní si nyní ukážeme.

V implementaci rozhraní *CurrentTenantIdentifierResolver* stačí v metodě *resolveCurrentTenantIdentifier()* poskytnout ID schématu, které máme uložené v thread local proměnné:

```
public class CurrentTenantResolverImpl
        implements CurrentTenantIdentifierResolver {

    @Override
    public String resolveCurrentTenantIdentifier() {
        //vracime ID jizdniho radu z thread local promenne
        return SchemaThreadLocal.get();
    }

    @Override
    public boolean validateExistingCurrentSessions() {
        return true;
    }

}
```

V implementaci *MultiTenantConnectionProvider* pak musíme zejména po obdržení spojení s databází z tzv. *connection poolu*[36] nastavit požadované schéma, se kterým budeme chtít v rámci tohoto spojení pracovat. Toho docílíme provedením specifického SQL dotazu pro databázi PostgreSQL:

```
SET search_path TO {schema}
```

[34] https://docs.jboss.org/hibernate/core/4.2/devguide/en-US/html/ch16.html
[35] https://docs.jboss.org/hibernate/orm/5.1/javadocs/org/hibernate/SessionFactory.html
[36] https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch03.html

```
public class SchemaPerTenantConnectionProviderImpl
        implements MultiTenantConnectionProvider {

    ...

    //tenantIdentifier bude obsahovat ID jizdniho radu
    @Override
    public Connection getConnection(final String tenantIdentifier)
        throws SQLException {

        final Connection connection = getAnyConnection();
        try {
            connection.createStatement()
                .execute("SET search_path TO " + tenantIdentifier);
        } catch (SQLException e) {
            throw new HibernateException(...);
        }

        return connection;
    }

    ...
}
```

5.2.4.2 Připojení k požadované Neo4j databázi

Implementace připojení ke zvolené databázi Neo4j dle parametru uloženého v thread local proměnné je mírně složitější. Je nutné pro každou databázi vytvořit vlastní konfiguraci připojení, která bude obsahovat příslušnou adresu databáze přes BOLT protokol a následně při získávání připojení k databázi v rámci vlákna použít právě tu konfiguraci, která je uložená v proměnné thread local.

Pro komunikaci s databází Neo4j v aplikaci využíváme knihovnu Spring Data Neo4j[37], jež umožňuje definici připojení k databázi vytvořením konfigurační třídy, která dědí od třídy *Neo4jConfiguration*. Kromě samotného vytvoření jednotlivých konfigurací připojení k databázi je pak nezbytné správně implementovat metodu *getSession()*, což je ve skutečnosti *beana*[38] spravovaná Springovým kontejnerem, která vrátí požadované připojení k databázi.

Bohužel *Session Factory*[39] z knihovny Spring Data Neo4j standardně neumožňuje udržovat spojení k více než jedné databázi Neo4j, z toho důvodu bylo nutné ji přepsat a logiku doimplementovat. Velmi důležité je pak mít metodu a současně beanu *getSession()* v konfiguraci Neo4j ve správném *scopu*[40] tak, aby se v rámci požadavku vybrala vždy správná databáze dle hodnoty proměnné thread local. K tomuto účelu se výborně hodí *SimpleThreadScope*[41].

[37] https://projects.spring.io/spring-data-neo4j/
[38] http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html
[39] http://docs.spring.io/spring-data/neo4j/docs/current/reference/html/
[40] http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/annotation/Scope.html
[41] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/support/SimpleThreadScope.html

Základní ukázku konfigurace připojení k databázím Neo4j můžeme vidět níže:

```
@org.springframework.context.annotation.Configuration
@EnableNeo4jRepositories(basePackages = ...)
@EnableTransactionManagement
public class Neo4jConfig extends Neo4jConfiguration {

    ...

    @Bean
    public Map<String, SessionFactory> getSessionFactoryMaps() {
        Map<String, SessionFactory> map = new HashMap<>();

        //do mapy vlozime postupne spojeni ke vsem databazim Neo4j
        //klic je ID jizdniho radu
        ...

        return Collections.unmodifiableMap(map);
    }

    @Bean
    @Scope(scopeName = "thread", proxyMode = TARGET_CLASS)
    public Session getSession() throws Exception {
        //ziskame ID jizdniho radu
        String identifier = SchemaThreadLocal.get();

        //a vratime pripojeni k databazi s timto jizdnim radem
        SessionFactory sessionFactory =
            getSessionFactoryMaps().get(identifier);
        Assert.notNull(sessionFactory, ...);
        return sessionFactory.openSession();
    }

    ...
}
```

5.2.5 Vyhledávání spojů

Z kapitoly 4.5.1 víme, že implementace vyhledávacího algoritmu nad databází Neo4j je navržena jako serverové rozšíření této databáze. Jedná se tedy o samostatný projekt mimo hlavní aplikaci, který se sestavuje do java archivu (*jar*)[42], který se následně umístí do složky *plugins* databáze Neo4j.

Metodu pro vyhledávání spojů oanotujeme anotací *Procedure* a po vyhledání výsledků tyto vrátíme. Pro vlastní vyhledávání pak využíváme již zmíněný Traversal Framework[43]. Nejprve nalezneme výchozí zastavení, ze kterých vyhledávání započne, a následně nadefinujeme vyhledávání pomocí *Traversal description*[44]. Základní přístup můžeme pozorovat níže

[42] http://www.oracle.com/technetwork/java/archive-139210.html
[43] https://neo4j.com/docs/java-reference/current/
[44] http://neo4j.com/docs/java-reference/current/javadocs/org/neo4j/graphdb/traversal/TraversalDescription.html

a víceméně odpovídá přístupu zvolenému již v bakalářské práci *BPJCH* [3]. Pro úplnost si uvedeme alespoň základní kostru implementace vyhledávání:

```
//vstupni parametry jsou parametry vyhledavani
@Procedure(name = "cz.cvut.dp.nss.search", mode = READ)
public Stream<SearchResultWrapper> searchByDeparture(...) {

    ...

    //vybereme vychozi uzly, ze kterych zacne vyhledavani
    ResourceIterator<Node> startNodes = ...;

    ...

    //nadefinujeme predpis traverzovani grafem
    TraversalDescription td = db.traversalDescription()
        .order(new DepartureBranchSelector(...))
        .uniqueness(Uniqueness.NODE_PATH)
        .expand(new DepartureTypeExpander(...))
        .evaluator(new DepartureTypeEvaluator(...));

    List<SearchResultWrapper> results = new ArrayList<>();
    for(Path path : td.traverse(startNodes)) {
        //vysledek prevedu na pozadovany objekt
        ...
        results.add(...);
    }

    return results.stream();
}
```

Objekty definované v *traversal description* implementují rozhraní z Neo4j traversal frameworku:

```
public class DepartureBranchSelector implements BranchSelector {

    @Override
    public TraversalBranch next(TraversalContext traversalContext) {
        //vyber vrcholu z prioritni fronty pro zpracovani
        //vkladani novych uzlu do prioritni fronty
    }

}

public class DepartureTypeEvaluator implements Evaluator {

    @Override
    public Evaluation evaluate(Path path) {
        //kontrola aktualne navstiveneho uzlu
    }

}
```

```
public class DepartureTypeExpander implements PathExpander<?> {

    @Override
    public Iterable<Relationship> expand(Path path, BranchState<?> bs) {
        //vyber hran, po kterych budu z vrcholu dale traverzovat
    }

}
```

5.2.6 Angular 2 aplikace

Kostra javascriptové aplikace psané ve frameworku Angular 2 byla vygenerována pomocí rozhraní Angular CLI[45], a je tak spravovatelná pomocí příkazů tohoto rozhraní. Například příkazem

```
ng build --target=production
```

dojde k sestavení aplikace a tvorbě statické stránky *index.html* a několika javascriptových souborů, které jsou do stránky vloženy a které obsahují kompletní implementaci aplikace. Díky napojení na Maven je pak možné při sestavování celé aplikace právě pomocí nástroje Maven sestavit i tuto javascriptovou aplikaci a vygenerovanou stránku *index.html* včetně javascriptových souborů umístit do složky *target*. Z této složky se pak díky správné konfiguraci stránka *index.html* načte při příchodu uživatele na root nasazené a běžící aplikace (např. http://localhost:8080).

Na tomto místě se hodí prezentovat alespoň krátkou ukázku implementace nějaké zajímavé části aplikace. Představíme si tak základní kostru funkčnosti přihlašování do aplikace. Již víme, že po úspěšném přihlášení obdrží klient autorizační token, který musí poté posílat s každým požadavkem na server. Po úspěšném přihlášení je tedy nutné někam si tento token v rámci aplikace uložit.

Nejprve ukázka HTML stránky přihlašovacího formuláře:

```
<form name="form" (ngSubmit)="login()" #f="ngForm">
    <div>
        <label for="username">Username:</label>
        <input type="text" name="username" id="username"
            [(ngModel)]="model.username" #username="ngModel" required />
    </div>
    <div class="form-group">
        <label for="password">Password:</label>
        <input type="password" name="password" id="password"
            [(ngModel)]="model.password" #password="ngModel" required />
    </div>
    <div>
        <button>Login</button>
    </div>
</form>
```

[45] https://cli.angular.io/

Obsluhující komponenta přihlašovacího formuláře:

```
@Component({
    ...
})

export class LoginComponent implements OnInit {

  model: User = new User();

  ...

  login() {
    this.authService.login(this.model.username, this.model.password)
      .subscribe(person => {...},
      err  => {
        this.loginFailed();
      });
  }

  ...

}
```

Třída obsluhující přihlášení do aplikace (odeslání HTTP požadavku a přijetí odpovědi):

```
@Injectable()
export class AuthService {

  ...

  login(username: string, password: string): Observable<Person> {
    let user = new User();
    user.username = username;
    user.password = password;

    return this.http.post(...URI..., JSON.stringify(user))
        .map(response => {
          let loggedUser = response.json();
          this.userService.storeUser(loggedUser);
          return loggedUser as Person;
        })
        .catch(err => this.errorService.handleServerError(err));
  }

  ...

}
```

A nakonec třída *UserService* spravující údaje o aktuálně přihlášeném uživateli:

```
@Injectable()
export class UserService {

  ...

  isLoggedIn(): boolean {
      return sessionStorage.getItem(LOGGED_USER) != null;
  }

  getLoggedUser(): LoggedUser {
      return this.isLoggedIn() ?
        JSON.parse(sessionStorage.getItem(LOGGED_USER)) : null;
  }

  storeUser(user: LoggedUser): void {
      sessionStorage.setItem(LOGGED_USER, JSON.stringify(user));
  }

  removeUser(): void {
      sessionStorage.removeItem(LOGGED_USER);
  }

  ...

}
```

5.2.7 Android

Android aplikace je poměrně jednoduchá a sestává pouze ze dvou *aktivit*[46], které v tomto kontextu znamenají obrazovky. První obrazovka je vyhledávací formulář a druhá obrazovka slouží pro zobrazení nalezených výsledků. Z kódu si nyní představíme pouze krátkou ukázku definice části vyhledávacího formuláře (konkrétně políčka *odkud* a *kam*) a metodu, která provede prohození obsahu obou polí po kliknutí na příslušné tlačítko.

Nejprve si ukážeme layout definice formulářových políček:

```
...

<cvut.cz.dp.nss.view.DelayAutoCompleteTextView
     android:id="@+id/stopFrom"
     android:inputType="textFilter"
     android:hint="Odkud" ... />

<cvut.cz.dp.nss.view.DelayAutoCompleteTextView
     android:id="@+id/stopTo"
     android:inputType="textFilter"
     android:hint="Kam" ... />
```

[46] https://developer.android.com/reference/android/app/Activity.html

```
...

<ImageButton
    android:id="@+id/button2"
    android:background="@drawable/ic_swap_horiz_black_48dp"
    android:onClick="swapStops" ... />

...
```

A následně konkrétní třídu reprezentující aktivitu:

```java
public class SearchActivity extends AppCompatActivity {

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        //inicializace aktivity (obrazovky)
        ...
    }

    /**
     * prohodi stanice odkud a kam
     */
    public void swapStops() {
        //najdu jednotliva pole
        DelayAutoCompleteTextView stopFrom =
            (DelayAutoCompleteTextView) findViewById(R.id.stopFrom);
        DelayAutoCompleteTextView stopTo =
            (DelayAutoCompleteTextView) findViewById(R.id.stopTo);
        String stopFromName = stopFrom.getText().toString();

        //a prohodim jejich obsah
        stopFrom.setText(stopTo.getText());
        stopTo.setText(stopFromName);
    }

    ...

}
```

5.3 Ukázky UI

Abychom demonstrovali vzhled vytvořených aplikací (webové a mobilní), ukážeme si nyní vyhledávací formuláře z obou těchto aplikací. U webové aplikace by měl vyhledávací formulář pochopitelně odpovídat vzhledu navrženému v kapitole 4.7.

Obrázek 5.3: Vyhledávací formulář webové aplikace.

Obrázek 5.4: Vyhledávací obrazovka mobilní aplikace (Android).

5.4 Kód a statistiky

Zdrojový kód všech aplikací je veřejně dostupný v git[47] repozitáři prostřednictvím webové služby GitHub[48]. Odkazy na jednotlivé repozitáře včetně přibližné velikosti zdrojových kódů nalezneme v seznamu níže:

- Webová aplikace (server a klient)
  - https://github.com/chaluja7/dp-nss
  - cca 26 500 řádků kódu
  - popis veřejného i administračního REST rozhraní je k dispozici na Apiary[49]
    - http://docs.dpnssapi.apiary.io/
    - http://docs.dpnssprivateapi.apiary.io/
- Mobilní Android aplikace
  - https://github.com/chaluja7/dp-nss-android
  - cca 2 500 řádků kódu
- Vyhledávání nad databází Neo4j (server side extension)
  - https://github.com/chaluja7/dp-nss-search
  - cca 3 000 řádků kódu

[47] https://git-scm.com/
[48] https://github.com/
[49] https://apiary.io/

Kapitola 6

Testování

V této kapitole se budeme věnovat komplexnímu testování celé aplikace. Nejdříve se zaměříme na testování kódu, následně na zátěžové testování, do kterého zařadíme zejména testování rychlosti vyhledávacího algoritmu, a nakonec si představíme výsledky uživatelského testování, které bylo uskutečněno s několika participanty.

6.1 Testování kódu

K otestování kódu jsme využili zejména jednotkové a integrační testy, které pokrývají business logiku aplikace. Pro testování byl použit testovací framework JUnit[1], který je integrován s nástrojem Apache Maven tak, že se testy spouštějí volitelně v předem určených fázích sestavení aplikace.

V rámci implementace byly testy pokrývající business logiku aplikace psány ihned po napsání konkrétní metody, a právě u business logiky aplikace se tak dle analýzy vygenerované prostřednictvím vývojového prostředí IntelliJ IDEA[2] podařilo dosáhnout pokrytí cca 96 % tříd, 77 % metod a 70 % řádků kódu.

6.2 Zátěžové a výkonnostní testování

Pro zátěžové a zejména výkonnostní testování bylo nutné naplnit databáze reálnými daty, nad kterými by bylo možné testy provést. Zvolená testovací data si tedy nejprve představíme a pak se již budeme věnovat vlastnímu testování.

6.2.1 Testovací data

V bakalářské práci *BPJCH* [3] byla pro testování výkonnosti vyhledávacího algoritmu použita data od Dopravní podniku hlavního města Prahy (DPP) o velikost 1 215 stanic, 45 146 jízd a 883 568 zastavení, která byla od Dopravního podniku získána na základě žádosti podle zákona č. 106/1999 Sb. o svobodném přístupu k informacím. Se získáváním dat

[1] http://junit.org/junit4/
[2] https://www.jetbrains.com/idea/

je nyní situace výrazně jednodušší, protože Dopravní podnik začal jízdní řády ve formátu GTFS zveřejňovat dobrovolně na stránce http://opendata.praha.eu/. K testování jsme tak mohli využít reálná a aktuální data o jízdních řádech Pražské integrované dopravy mimo příměstských vlaků[3] z období cca. 1.4.2017 - 6.6.2017, která dosahují následující velikosti:

- 2 979 stanic s unikátním jménem (6 789 celkem)
- 85 025 jízd
- 1 739 389 zastavení

Všimněme si, že oproti testovacím datům použitým v bakalářské práci *BPJCH* [3] jsou tato data velikostně přibližně dvojnásobná.

6.2.2 Testovací prostředí

Všechny testy z této kapitoly byly provedeny na již finálně nasazené aplikaci ve službě Amazon AWS. Obě instance aplikace jsou nasazeny na typu instance *t2.micro*[4], na které běží operační systém Ubuntu 16.04 a jejíž hardwarová konfigurace zahrnuje jednojádrový procesor Intel Xeon Family s frekvencí 2.5 GHz, operační paměť 1 GB a SSD disk. Apache Tomcat má ponechanou výchozí velikost paměti pro JVM (Java Virtual Machine)[5], která činí 256 MB.

Grafová databáze Neo4j je nasazena na typu instance *t2.xlarge*, na které běží operační systém Ubuntu 16.04 a jejíž hardwarová konfigurace zahrnuje čtyřjádrový procesor Intel Xeon Family s frekvencí 2.4 GHz, operační paměť 16 GB a SSD disk. Tato konfigurace je dle dokumentace Neo4j[6] minimální doporučovaná.

6.2.3 Testování rychlosti importu a exportu jízdních řádů

Protože import a export jízdních řádů ve formátu GTFS je jednou z nejdůležitějších funkcionalit aplikace, je vhodné otestovat, jak dlouho tyto operace na testovacím prostředí trvají. K otestování jsme použili jízdní řády z kapitoly 6.2.1, které mají v archivačním formátu *.zip* velikost 13,3 MB a po rozbalení dokonce 90,3 MB. Upozorňujeme, že se jedná pouze o jednoduché textové soubory ve formátu *.txt*. Celkem se tak bude do databáze importovat více než 2 400 000 řádků.

Do doby importu je započítána i doba nahrávání souboru na server, která ale byla v našem případě zcela zanedbatelná. Stejně tak do doby exportu je započítána i doba, po kterou se již vyexportovaný jízdní řád ze serveru stahoval. V obou případech se jednalo maximálně o jednotky vteřin. Časy importu a exportu si konečně představíme:

[3] http://opendata.praha.eu/organization/cc93f650-b175-4f3f-844d-a246955165af?res_format=GTFS
[4] https://aws.amazon.com/ec2/instance-types/
[5] https://docs.oracle.com/javase/specs/jvms/se7/html/
[6] https://neo4j.com/docs/operations-manual/current/installation/requirements/

Typ operace	**Přibližný čas operace**
Import	1 hodina 12 minut
Export	38 vteřin

Tabulka 6.1: Rychlost importu a exportu jízdního řádu o přibližné velikosti 2 400 000 řádků dat.

Výsledné časy obou operací považujeme za velmi dobré. U importu je nutné si navíc uvědomit, že ve skutečnosti se zapisuje mnohem více záznamů, protože údaje o jízdách je nutné uložit také do grafové databáze Neo4j. Reálná rychlost zápisu dat do databází je tak více než 900 záznamů za vteřinu.

6.2.4 Testování rychlosti vyhledávání

Testování rychlosti vyhledávání bylo provedeno na řadě náhodně vybraných tras s různým maximálním počtem přestupů. Při výběru výchozí a cílové stanice jsme se snažili volit stanice tak, abychom pokryli jak krátké trasy, tak trasy, které jsou co možná nejdelší. Ačkoliv testování probíhalo v rámci webové aplikace, jsou naměřené časy platné i pro aplikaci mobilní, protože u obou probíhá vyhledávání zcela totožným způsobem, a to dotazem na REST API, které deleguje požadavek na vyhledávací plugin databáze Neo4j.

Čas výjezdu a maximální čas příjezdu byl pro všechna vyhledávání shodný a záměrně byl zvolen čas ranní špičky ve všední den, kdy jede nejvyšší počet spojů. Taktéž maximální požadovaný počet nalezených výsledků byl pro všechna vyhledávání zvolen shodně:

- *Datum a čas výjezdu* ... 26.4.2017 (středa) 08:00
- *Maximální datum a čas příjezdu* ... 26.4.2017 14:00
- *Požadovaný počet výsledků* ... 3

Okno na vyhledávání je tedy šest hodin, což je v rámci námi testovaných jízdních řádů zcela dostačující. Vyhledávání se navíc ukončí po nalezení prvních tří výsledků.

Jako čas vyhledávání budeme uvádět čas, po který trval kompletní požadavek na vyhledávání, tedy včetně doby odesílání požadavku na server a následného obdržení odpovědi. Výsledný naměřený čas tak bude obsahovat i případná zpoždění sítě, která jsou ale v našem případě vzhledem ke stabilnímu připojení zcela zanedbatelná. Výsledné časy díky tomuto způsobu měření budou zaznamenávat kompletní dobu, po kterou uživatel čekal na zobrazení výsledků po odeslání vyhledávacího formuláře.

Nejprve si představíme naměřené časy pro maximálně jeden přestup:

Odkud	**Kam**	**Doba hledání [s]**
Bělocerkevská	Dejvická	0.259
Dejvická	Karlovo náměstí	0.219
Černý Most	Hybšmanka	0.789
Černý Most	Terminál 1	1.610
Florenc	Anděl	0.242
Výstaviště Holešovice	Bělocerkevská	0.873
Zličín	Chodov	0.567
Florenc	Jinočany,Hlavní	6.650
Jinočany,Hlavní	Florenc	0.128
Roztoky,nádraží	Flora	0.150
Flora	Roztoky,nádraží	1.580

Tabulka 6.2: Rychlost vyhledávání s maximálně jedním přestupem.

Rychlost vyhledávání je u většiny případů dobrá, a uživateli se tak vyhledané výsledky zobrazí prakticky okamžitě. Výjimku tvoří zejména spojení z Florence do zastávky Jinočany,Hlavní, kde doba vyhledávání převýšila šest vteřin. Je to způsobené tím, že zatímco z Florence odjíždí během hodiny mnoho spojů, do Jinočan jezdí pouze jeden autobus za hodinu. Než algoritmus najde výsledky, musí probíhat vyhledávání až do 11:24 (vyhledávání začíná v 8:00), kdy do cílové stanice přijíždí autobus č. 352, který figuruje ve třetím nalezeném výsledku. Tento jev delšího vyhledávání pro tyto konkrétní typy spojů se projevil i v bakalářské práci *BPJCH* [3], ze které vyhledávací algoritmus vychází, proto nás nepřekvapí. U vyhledávání s prohozenou výchozí a cílovou stanicí je výsledek nalezen prakticky ihned, protože je situace přesně opačná.

Nyní se podíváme na naměřené časy pro maximálně tři přestupy, které v rámci testovaného jízdního řádu stačí pro většinu možných tras, a jde o výchozí hodnotu zvolenou ve vyhledávacím formuláři:

Odkud	**Kam**	**Doba hledání [s]**
Bělocerkevská	Dejvická	0.609
Dejvická	Karlovo náměstí	0.194
Černý Most	Hybšmanka	5.260
Černý Most	Terminál 1	12.310
Florenc	Anděl	0.181
Výstaviště Holešovice	Bělocerkevská	1.960
Zličín	Chodov	3.780
Florenc	Jinočany,Hlavní	29.170
Jinočany,Hlavní	Florenc	0.499
Roztoky,nádraží	Flora	0.513
Flora	Roztoky,nádraží	8.670

Tabulka 6.3: Rychlost vyhledávání s maximálně třemi přestupy.

Výrazně vzrostla doba vyhledávání spojení z Florence do Jinočan. Důvod už jsme si vysvětlili dříve, nyní se pouze čas vyhledávání výrazně navýšil kvůli mnohem většímu prohledávanému prostoru. Opět si povšimneme, že zpáteční spojení je nalezené prakticky ihned. Naopak výsledky, kde jsou výchozí i cílová stanice frekventované podobně, se podařilo nalézt ve slušném čase.

Dále si představíme naměřené časy pro maximálně čtyři přestupy, což je maximum, které může uživatel ve vyhledávacím formuláři zvolit:

Odkud	**Kam**	**Doba hledání [s]**
Bělocerkevská	Dejvická	0.633
Dejvická	Karlovo náměstí	0.207
Černý Most	Hybšmanka	5.490
Černý Most	Terminál 1	13.680
Florenc	Anděl	0.173
Výstaviště Holešovice	Bělocerkevská	2.140
Zličín	Chodov	3.970
Florenc	Jinočany,Hlavní	29.450
Jinočany,Hlavní	Florenc	0.431
Roztoky,nádraží	Flora	0.552
Flora	Roztoky,nádraží	9.300

Tabulka 6.4: Rychlost vyhledávání s maximálně čtyřmi přestupy.

Doba vyhledávání se oproti vyhledávání se třemi přestupy téměř nenavýšila, což je velmi pozitivní informace.

Nakonec otestujeme rychlost vyhledávání s nově implementovanou funkcí oproti bakalářské práci *BPJCH* [3], a to s využitím přestupní nebo průjezdní stanice. Výchozí a cílovou stanici ponecháme shodnou s předchozími vyhledáváními, ale navíc přidáme přestupní nebo průjezdní stanici, přes kterou se budeme chtít z výchozí do cílové stanice dopravit:

Odkud	**Kam**	**Přes**	**Doba hledání [s]**
Bělocerkevská	Dejvická	Želivského	1.000
Dejvická	Karlovo náměstí	Můstek	0.394
Černý Most	Hybšmanka	Karlovo náměstí	4.700
Černý Most	Terminál 1	Dejvická	3.330
Florenc	Anděl	Národní třída	0.362
Výstaviště Holešovice	Bělocerkevská	Muzeum	1.040
Zličín	Chodov	Florenc	24.210
Florenc	Jinočany,Hlavní	Zličín	30.110
Jinočany,Hlavní	Florenc	Zličín	0.356
Roztoky,nádraží	Flora	Nádraží Podbaba	0.538
Flora	Roztoky,nádraží	Nádraží Podbaba	3.950

Tabulka 6.5: Rychlost vyhledávání s maximálně třemi přestupy a zvolenou stanicí *přes*.

Rychlost vyhledávání zůstala u většiny testovaných tras podobná. K výraznému zlepšení nicméně došlo při vyhledávání z Černého Mostu na Terminál 1 a z Flory do Roztok. K výraznému zhoršení naopak došlo při vyhledávání ze Zličína na Chodov.

6.2.5 Zátěžové testování

Abychom otestovali připravenost aplikace na nápor většího množství uživatelů, vytvořili jsme testovací situaci, kdy se současně do aplikace připojí větší množství uživatelů a ti začnou postupně vyhledávat spojení mezi různými stanicemi. Měřenou jednotkou je pak průměrná a maximální doba odezvy serveru na všechny požadavky. Současně nám tento test také pomůže odhalit, zda se aplikace kompletně nepřetíží či nezačne požadavky na vyhledávání odmítat.

K vlastnímu testování jsme zvolili nástroj Apache JMeter[7], který umožňuje námi požadovaný test nasimulovat. Testování probíhalo zejména formou zasílání požadavků přímo na endpoint REST API, který slouží k vyhledávání, protože právě ten má neblahý potenciál tvořit úzké hrdlo. Otestovali jsme ale i nárazový přístup uživatelů na hlavní stránku aplikace.

6.2.5.1 Konfigurace

Nástroj Apache JMeter umožňuje velmi snadno vytvořit testovací situaci, ve které nadefinujeme HTTP požadavky, které budeme chtít provádět, zvolíme počet vláken (uživatelů), která je budou provádět, a zvolíme také časový interval, během kterého budou jednotlivé požadavky prováděny.

Základem každého testovacího plánu je tzv. *Thread Group*[8], v rámci které už můžeme nadefinovat jednotlivé HTTP požadavky[9], které budeme chtít provádět. V konfiguraci jsou pro nás důležité zejména dva atributy, které musíme správně nastavit. Konkrétně jimi jsou:

- **Počet vláken (uživatelů)** - počet virtuálních uživatelů, kteří budou na aplikaci přistupovat
- **Doba náběhu vláken (Ramp-Up period)** - počet vteřin, během kterých začne HTTP požadavky posílat kompletní počet zadaných uživatelů. Pokud tedy nastavíme počet vláken i dobu náběhu například na 10, tak během deseti vteřin začne posílat požadavky všech deset uživatelů, jinými slovy každou vteřinu začne posílat požadavky další uživatel. Pokud nastavíme *počet vláken = 100* a *doba náběhu = 10*, tak všech sto uživatelů začne posílat požadavky během deseti vteřin, tedy každý uživatel začne posílat požadavky se zpožděním jedné desetiny vteřiny po předchozím uživateli.

Současně je dobré si objasnit, jak testování probíhá, pokud má každé vlákno vykonat více než jeden HTTP požadavek. V tom případě začne každé vlákno vykonávat požadavky postupně a další požadavek z fronty vykoná až po obdržení odpovědi na požadavek předchozí, čímž poměrně věrně umožňuje simulovat chování reálného uživatele.

[7] http://jmeter.apache.org/
[8] http://jmeter.apache.org/usermanual/test_plan.html#thread_group
[9] http://jmeter.apache.org/usermanual/test_plan.html#controllers

6.2.5.2 Průběh

Nyní se už pustíme do vlastního testování. Nejprve otestujeme nárazový přístup uživatelů na hlavní stránku aplikace, kterou tvoří vyhledávací formulář. Upozorňujeme, že naměřené časy vyjadřují dobu odpovědi od serveru a nikoliv dobu náběhu javascriptové aplikace. Naměřené časy pro různé konfigurace můžeme pozorovat v tabulce níže:

Počet uživatelů	Doba náběhu [s]	Prům. odpověď [s]	Max. odpověď [s]
50	1	0.041	0.052
100	1	0.042	0.057
200	1	0.043	0.060
400	1	0.069	1.183
800	1	0.234	1.282
1000	1	0.600	2.264

Tabulka 6.6: Průměrná a maximální doba odpovědi serveru při přístupu na hlavní stránku aplikace.

Naměřené časy považujeme za velmi dobré, například při přístupu osmi set uživatelů během jedné vteřiny zvládne server obsloužit všechny uživatele během průměrné doby 0.234 s, což je výborné číslo.

Nyní se pustíme do zátěžového testování vyhledávacího algoritmu. Vyhledávat budeme se stejnými parametry jako v kapitole 6.2.4, navíc ale budeme vyhledávat vždy s maximálně třemi přestupy, což je výchozí možnost.

Nejprve zkusíme vyhledávat mezi stanicemi, u kterých netrvalo při testování rychlosti vyhledávacího algoritmu vyhledávání příliš dlouho. Konkrétně tak každého uživatele necháme postupně vyhledávat tato spojení:

Odkud	Kam
Bělocerkevská	Dejvická
Dejvická	Karlovo náměstí
Florenc	Anděl
Jinočany,Hlavní	Florenc
Roztoky,nádraží	Flora

Tabulka 6.7: Vyhledávací trasy pro první zátěžový test vyhledávání.

Každý uživatel (vlákno) tedy provede všech pět vyhledávání, jinými slovy pokud nastavíme *počet uživatelů = 10*, tak celkem bude zavoláno 50 HTTP požadavků na vyhledávání. Naměřené časy pro tuto konfiguraci můžeme pozorovat v následující tabulce:

Počet uživatelů	Doba náběhu [s]	Prům. odpověď [s]	Max. odpověď [s]
10	1	0.463	1.062
20	1	0.880	2.471
40	1	1.904	4.367
80	1	8.257	18.687
40	5	1.623	4.555
80	10	3.295	10.131

Tabulka 6.8: Průměrná a maximální doba odpovědi ze serveru pro první testovací sadu.

Při nastavení osmdesáti uživatelů, kdy všichni začnou posílat HTTP požadavky během jedné vteřiny, se již projevuje výraznější zpoždění odpovědi ze serveru. Pokud ale zaslání prvního požadavku těchto osmdesáti uživatelů rozmělníme do deseti vteřin, výsledný průměrný čas odpovědi se dá ještě považovat za přijatelný.

Nyní do testu začleníme i vyhledávání mezi stanicemi, která v kapitole 6.2.4 trvala delší dobu. Zcela logicky se tak prodlouží i průměrné a maximální časy navrácení odpovědí ze serveru. Pro nás bude ale důležité pozorovat, jestli prokládání dlouhotrvajícími vyhledávacími požadavky neovlivní jinak rychle provedená vyhledávání. Ponecháme tedy v platnosti vyhledávání z předchozího testu, a navíc je začneme prokládat požadavky na vyhledávání mezi stanicemi, které trvají delší dobu. Konkrétně budeme navíc vyhledávat mezi stanicemi:

Odkud	Kam
Černý Most	Hybšmanka
Černý Most	Terminál 1
Zličín	Chodov

Tabulka 6.9: Vyhledávací trasy pro druhý zátěžový test vyhledávání.

V tabulce nyní zobrazíme porovnání průměrné a maximální doby odpovědi jak na testovací případy z minulého testu, tak na právě uvedené testované případy, pokud spustíme oba testy současně. Data testovacích sad jsou uvedena ve formátu *Počet uživatelů / Doba náběhu [s]* a časy testovacích sad jsou uvedeny ve formátu *Průměrný čas odpovědi [s] / Maximální čas odpovědi [s]*:

Data 1. test. sady	Data 2. test. sady	Časy 1. test. sady	Časy 2. test. sady
10 / 1	2 / 1	0.774 / 1.533	9.871 / 16.283
10 / 1	4 / 1	1.032 / 2.319	16.330 / 27.138
20 / 1	2 / 1	1.877 / 4.454	12.604 / 16.474
20 / 1	4 / 1	1.731 / 4.319	17.957 / 28.467
40 / 1	4 / 1	3.892/ 8.889	22.234 / 30.421
40 / 5	5 / 1	3.259/ 8.254	26.888 / 41.369
80 / 10	5 / 1	6.360/ 15.314	34.288 / 52.902

Tabulka 6.10: Průměrná a maximální doba odpovědi serveru pro první a druhou testovací sadu.

Z naměřených výsledků je porovnáním s testováním pouze samotné první testovací sady zřejmé, že došlo i ke zpomalení vyhledávání dříve rychle nalezených výsledků. Propad rychlosti nalezení těchto výsledků je ale přijatelný. Naopak navyšováním počtu požadavků na vyhledávání, která trvají delší dobu, dochází k nezanedbatelnému propadu výkonnosti.

6.3 Uživatelské testování

Posledním typem provedených testů je testování uživatelského rozhraní. Toto testování probíhalo na papírových prototypech dokonce ještě před začátkem vlastní implementace a díky tomu se podařilo odhalit chyby v návrhu rozhraní hned v samotných počátcích. V této kapitole se budeme věnovat už pouze uživatelskému testování hotové a nasazené aplikace a slibujeme si od něj odhalení nedostatků, které by mohly a měly být opraveny v rámci budoucích úprav již mimo rozsah této diplomové práce.

Testování probíhalo v aplikaci s nahranými kompletními jízdními řády PID uvedenými v kapitole 6.2.1.

6.3.1 Výběr participantů

Pro testování byli zvoleni tři participanti z okruhu známých tak, aby zastupovali okruh lidí různého věku a různé úrovně počítačových dovedností. Při výběru participantů byl brán zřetel na to, aby měl každý alespoň částečnou znalost pražské MHD, která bude nezbytná pro testování vyhledávání. Jednotlivé participanty si v krátkosti představíme.

6.3.1.1 1. participant

První participantkou byla dívka ve věku osmnácti let. Jedná se o studentku střední školy, která má základní znalosti o počítačích a internetu. Vlastní chytrý mobilní telefon, který využívá denně.

6.3.1.2 2. participant

Druhým participantem byl muž ve věku dvaceti osmi let, který pracuje jako programátor. Díky své práci má velmi pokročilé znalosti počítačů, internetu i problematiky vývoje mobilních aplikací.

6.3.1.3 3. participant

Posledním participantem byl muž ve věku padesáti devíti let, který pracuje jako středoškolský učitel. Počítač využívá ke standardní kancelářské práci a má mírně pokročilé znalosti internetu. Tento participant není vlastníkem chytrého dotykového mobilního telefonu.

6.3.2 Testovací prostředí

Mobilní aplikace byla testována jedním participantem na zařízení Lenovo Vibe X2 a dvěma participanty na zařízení Samsung Galaxy Tab 3. Webová aplikace byla testována střídavě v aktuálních verzích prohlížečů Google Chrome, Mozilla Firefox a Opera na monitoru o rozlišení 1920 x 1080 pixelů.

6.3.3 Testovací scénáře

Nejdůležitější částí aplikace k otestování je bezpochyby proces vyhledávání spojů a zobrazování informací o vyhledaných spojích, který bude využívat převážná většina uživatelů. Naproti tomu administrační část je zamýšlena pro velmi úzký okruh obmyšlených uživatelů, u kterých se navíc očekává detailní znalost použité terminologie jízdních řádů. Před průchodem scénářů nad administračním rozhraním tak byl každý participant seznámen se základní terminologií jízdních řádů uvedenou v kapitole 2.3.1. Participantům ale v žádném případě nebyla terminologie vysvětlována nad vlastní aplikací - do té se mohli přihlásit až v průběhu testování tak, aby nemohlo dojít k ovlivnění výsledků testů.

Nyní si už můžeme představit testovací scénáře. Začneme scénáři pro mobilní aplikaci:

1. Najdi spojení z Hlavního nádraží na Anděl s časem odjezdu 13:00. Poté nalezni zpáteční spoj s časem odjezdu 16:00.

2. Najdi spojení mezi libovolně zvolenými stanicemi a po jejich nalezení najdi spoj, který odjíždí bezprostředně po spoji, který je v nalezených výsledcích zobrazený jako poslední.

Následovat budou testovací scénáře pro webovou aplikaci:

1. Najdi spojení z Hlavního nádraží na Anděl s časem odjezdu 13:00. Trasu jednotlivých nalezených spojů si zobraz na mapě a lokalizuj stanice, ve kterých budeš případně přestupovat. Poté nalezni zpáteční spojení s časem odjezdu 16:00.

2. Najdi spojení mezi libovolně zvolenými stanicemi a po jejich nalezení najdi spoj, který odjíždí bezprostředně po spoji, který je v nalezených výsledcích zobrazený jako poslední.

3. Pomocí uvedených přihlašovacích údajů se přihlas do administrace a zobraz si přehled dostupných jízdních řádů, které můžeš editovat. *Poznámka - všichni participanti měli vygenerované jednorázové heslo a po přihlášení do aplikace byli donuceni si jej změnit. Bez změny jednorázového hesla nebylo možné úkol dokončit, protože seznam jízdních řádů je k dispozici až poté, co uživatel jednorázové heslo změní.*

4. Proveď úpravu jízdního řádu PID tak, aby se přestal nabízet ve vyhledávacím formuláři pro vyhledávání spojů.

5. Najdi všechny stanice se jménem *Zličín* a uprav je tak, aby byly všechny bezbariérové.

6. Vlož nového dopravce libovolného jména a následně jej podle tohoto jména najdi v tabulce všech dopravců. Poté tohoto dopravce smaž.
7. Nejprve vlož libovolného nového dopravce a vytvoř nový spoj, který bude tento dopravce provozovat. Poté vytvoř interval platnosti, který bude zahrnovat pouze dnešní den. Dále vytvoř novou jízdu, která bude v provozu pro právě vytvořený interval platnosti a bude ji obsluhovat tebou vytvořený spoj. Následně přidej jízdě zastavení tak, aby jela ze stanice Zličín do stanice Chodov přes stanici Můstek.

6.3.4 Nálezy

Nyní si představíme postřehy a problémy, na které participanti narazili při plnění testovacích scénářů. Nebudeme uvádět postřehy každého participanta zvlášť, ale uvedeme již seskupenou množinu nálezů, které z kompletního testování vyplynuly. Všechny nálezy kromě prvního uvedeného se týkají výhradně webové aplikace. U všech nálezů je v závorce uvedena priorita.

6.3.4.1 Komponenty pro výběr času (nízká priorita)

V mobilní aplikaci byla nalezena chyba u komponenty pro výběr data, která obsahovala některé údaje v angličtině. Po změně orientace zařízení se ale již zobrazilo vše správně v češtině. Toto chybné chování bylo zjištěno pouze na zařízení Lenovo Vibe X2.

U komponenty pro výběr času ve vyhledávacím formuláři webové aplikace si jeden participant stěžoval na nekonzistentní chování. Při klikání minut směrem dolů se od aktuálního času ubírá vždy patnáct minut. Při klikání minut směrem nahoru se ale čísla zaokrouhlují vždy na celé čtvrthodiny.

6.3.4.2 Filtrování položek v administraci (vysoká priorita)

Všichni participanti měli problémy nebo alespoň připomínky k filtrování tabulek administrační části. Jednomu participantovi nebylo zřejmé, že vyhledávat se začne i pomocí enteru a nemusí vyplňovat celý hledaný text. Jiný participant navrhl, že by vyhledávání mělo probíhat ihned při psaní vyhledávacího výrazu a navíc by mělo být možné filtrovat i vkládáním výrazů bez diakritiky, jako je tomu u vyhledávacího formuláře při našeptávání stanic.

Největším odhaleným nedostatkem, na který narazili všichni participanti, ale bylo neukládání filtru mezi zobrazením stránek. Pokud uživatel vyfiltruje záznam a přejde na jeho editaci, tak po návratu zpět na seznam dojde ke ztrátě dříve vyplněného filtru, a uživatel jej musí vyplňovat znovu.

6.3.4.3 Nutnost vkládání ID (nízká priorita)

Dva participanti se pozastavili nad nutností vkládat ID k dopravci, intervalu platnosti a jízdě, když všechny existující záznamy mají ID numerické, a mohlo by se tak generovat samo. V případě jízdního řádu PID je toto pravda, ale dle specifikace formátu GTFS nemusejí být ID nutně číselná, a naopak mohou obsahovat libovolný řetězec, který musí být unikátní v rámci daného jízdního řádu. Pokud tedy chceme zachovat možnost importovat libovolný

jízdní řád z formátu GTFS, nelze bohužel nechat zodpovědnost za přidělování ID databázi nebo jinému autoinkrementálnímu nástroji. Velmi užitečné by ale bylo do formuláře vkládání přidat uživatelský prvek, který by ID chytře vygeneroval. V případě, že by všechny záznamy v tabulce měly ID numerické, mohl by přiřadit další číslo v řadě, případně by mohl alespoň vygenerovat unikátní řetězec, který by jako ID mohl sloužit.

6.3.4.4 Zobrazování chybových hlášek (střední priorita)

Některé zobrazené chybové hlášky jsou matoucí. Jeden participant se pokusil uložit dopravce s ID, které již měl přiřazené jiný dopravce. Po uložení se zobrazila chybová hláška *org.hibernate.exception.ConstraintViolationException*, ze které participant nepoznal, v čem je problém a proč se nedaří dopravce uložit.

6.3.4.5 Vytváření a editace jízdy (vysoká priorita)

Při vytváření jízdy je nutné zadat ID spoje a ID intervalu platnosti, která platí pro tuto jízdu. Ačkoliv tyto entity všichni participanti vytvořili v předchozím kroku a ve formuláři vložení jízdy existuje pro vyhledávání spoje našeptávač a interval platnosti se vybírá ze seznamu, tak jeden participant měl problém si vzpomenout, jaké ID přiřadil jízdě, kterou vytvořil. Bohužel měl také problém jím vytvořenou jízdu dohledat i zpětně v seznamu jízd. K vyšší uživatelské přívětivosti by bylo vhodné umožnit vytvoření jízdy kliknutím na tlačítko z editace spoje (například tlačítko *Vytvořit jízdu tohoto spoje*). Po kliknutí na tlačítko by se pak otevřel formulář pro vložení jízdy s již předvyplněným spojem.

Bezkonkurenčně největší problémy ale všem participantům způsobovalo vkládání zastavení jízdy. Zastavení jsou totiž identifikována pomocí ID stanice, protože jméno stanice není napříč jízdním řádem unikátní. Aplikace bohužel momentálně nenabízí pro toto pole žádný našeptávač, všichni participanti tak nakonec museli otevřít novou záložku prohlížeče a v té na seznamu stanic dohledávat ID stanice dle jména. Tento postup je uživatelsky velmi nepříjemný. Pole *stanice* u zastavení by mělo poskytovat minimálně našeptávač, který by zobrazoval stanice dle zadaného jména. Ještě lepší by pravděpodobně bylo umožnit zvolit stanici kliknutím do mapy.

6.3.4.6 Další nálezy (nízká priorita)

Při zobrazování spoje na mapě je nastaveno pevné přiblížení mapy, a trasa delšího spoje tak není ihned po načtení stránky vidět celá. Přiblížení mapy by mělo být automatické a mělo by se určovat tak, aby celá trasa byla po načtení stránky vidět na mapě celá.

V administraci na záložce Jízdní řády není při zobrazených dvou jízdních řádech zřejmé, který z nich je aktuálně vybraný. Vybraný jízdní řád je třeba graficky zobrazit tak, aby bylo ihned zřejmé, že je vybraný.

V našeptávači se například u pole *ID nadřazené stanice* na editaci stanice po výběru zobrazuje ID stanice a její název, přičemž oddělovačem je mezera. Jednoho participanta toto zobrazení zmátlo a nebyl si jistý, zda tak může entitu uložit. Oddělení ID a názvu by v tomto případě mělo být zřejmější.

6.3.5 Shrnutí uživatelského testování

Uživatelské testování objevilo několik nedostatků, které bude třeba opravit, ale většina z nich se týká výhradně administrace jízdních řádů. Naopak u vyhledávání nebyly žádné podstatné nálezy zaznamenány.

Kapitola 7

Nasazení

7.1 Nasazení webové aplikace

Webová aplikace byla nasazena ve veřejném cloudu Amazon AWS[1] a je dostupná na adrese `https://dp-nss.org`.

Při nasazování aplikace jsme pochopitelně museli brát v úvahu zamýšlené schéma nasazení prezentované v kapitole 4.6, které jsme téměř přesně dodrželi. Amazon AWS poskytuje obrovské množství produktů a typů služeb, které je možné využít. Při nasazování naší aplikace jsme využili jen pár z nich a detaily si nyní představíme.

7.1.1 Relační databáze

Pro nasazení relační databáze v cloudu je nabízena služba *Amazon Relation Database Service (RDS)*[2]. Po správném vytvoření a nakonfigurování instance je tak databáze ihned k dispozici. V našem případě, kdy jsme se rozhodli využívat relační databázi PostgreSQL, jsme si konkrétně vytvořili instanci přímo pro tuto databázi[3].

7.1.2 Grafová databáze

Neo4j v dokumentaci přímo obsahuje pasáž o nasazení do služby AWS, konkrétně do instance EC2[4]. *Amazon Elastic Compute Cloud (EC2)*[5] je webová služba, která umožňuje vytvoření vlastního výpočetního prostoru (instance) v cloudu s možností prakticky libovolné konfigurace. V našem případě jsme tak vytvořili instanci s operačním systémem Ubuntu[6], ve které jsme již databázi Neo4j spustili jako proces na pozadí.

Pro minimální požadovanou hardwarovou konfiguraci, kterou databáze Neo4j vyžaduje[7], bohužel není možné spustit tuto instanci zdarma[8]. Z tohoto důvodu jsme se rozhodli, že pro

[1] `https://aws.amazon.com/`
[2] `https://aws.amazon.com/rds/`
[3] `https://aws.amazon.com/rds/postgresql/`
[4] `https://neo4j.com/developer/guide-cloud-deployment/#_hosting_on_aws_ec2`
[5] `https://aws.amazon.com/ec2/`
[6] `https://www.ubuntu.com/`
[7] `https://neo4j.com/docs/operations-manual/current/installation/requirements/`
[8] `https://aws.amazon.com/ec2/pricing/on-demand/`

účely naší práce na této instanci budeme nasazovat všechny databáze Neo4j (s různými jízdními řády), abychom nemuseli platit další instance. Každá databáze pak bude konfigurována pro komunikaci na jiných portech.

7.1.3 Aplikace

Samotná aplikace byla nasazena také na instanci EC2, která byla replikována, čímž jsme dosáhli statického horizontálního naškálování aplikace. Pro instanci jsme opět zvolili operační systém Ubuntu, na kterém je aplikace spouštěna jako proces v servlet kontejneru Apache Tomcat[9].

7.1.4 Load balancer

Protože každá z instancí aplikace má pochopitelně jinou IP adresu, bylo nutné zprovoznit i load balancer, který bude sloužit jako vstup do aplikace a bude směřovat provoz na jednotlivé instance aplikace dle zatížení. AWS k tomuto účelu nabízí službu *Elastic Load Balancig*[10]. Po správné konfiguraci služba umí zcela automaticky přeposílat požadavky na zaregistrované instance dle vnitřních algoritmů.

Zde je na místě si uvědomit, že load balancer je jediná instance, která musí být přístupná veřejně (*z internetu*). Ostatní již představené instance nemusí, a dokonce by pravděpodobně neměly mít veřejnou IP adresu, protože nechceme, aby se na ně někdo cíleně připojoval. Mezi sebou instance komunikují pomocí privátní IP adresy (adresy vnitřní sítě), kterou mají přidělenou v rámci VPC (Virtual Private Cloud)[11], ve kterém běží. Tyto IP adresy vnitřní sítě pak pochopitelně nejsou dostupné ze sítě vnější.

V našem případě všechny instance veřejnou IP adresu mají, ale kromě load balanceru jsou na všech instancích nastavená pravidla pro filtrování příchozích požadavků. Na instance tak není možné se připojit jinak než s IP adresou, která je explicitně povolená.

7.1.5 Doména a HTTPS

Aby nebylo nutné na aplikaci přistupovat pouze pomocí IP adresy, byl zakoupen záznam v systému DNS (Domain Name System)[12]. Pro zakoupenou doménu byl dále vygenerován SSL certifikát, a k doméně je tak možné přistupovat i pomocí zabezpečeného protokolu HTTPS.

Po zakoupení domény bylo nutné přesměrovat požadavky, které na doménu přijdou, na námi vytvořený load balancer. K tomuto účelu existuje v AWS služba *Amazon Route 53*[13], kde jsme po správné konfiguraci dosáhli přesně tohoto chování. Aplikace je tak dostupná na již uvedené adrese `https://dp-nss.org`.

[9] `http://tomcat.apache.org/`
[10] `https://aws.amazon.com/elasticloadbalancing/`
[11] `https://aws.amazon.com/vpc/`
[12] `https://www.verisign.com/en_US/website-presence/online/how-dns-works/index.xhtml`
[13] `https://aws.amazon.com/route53/`

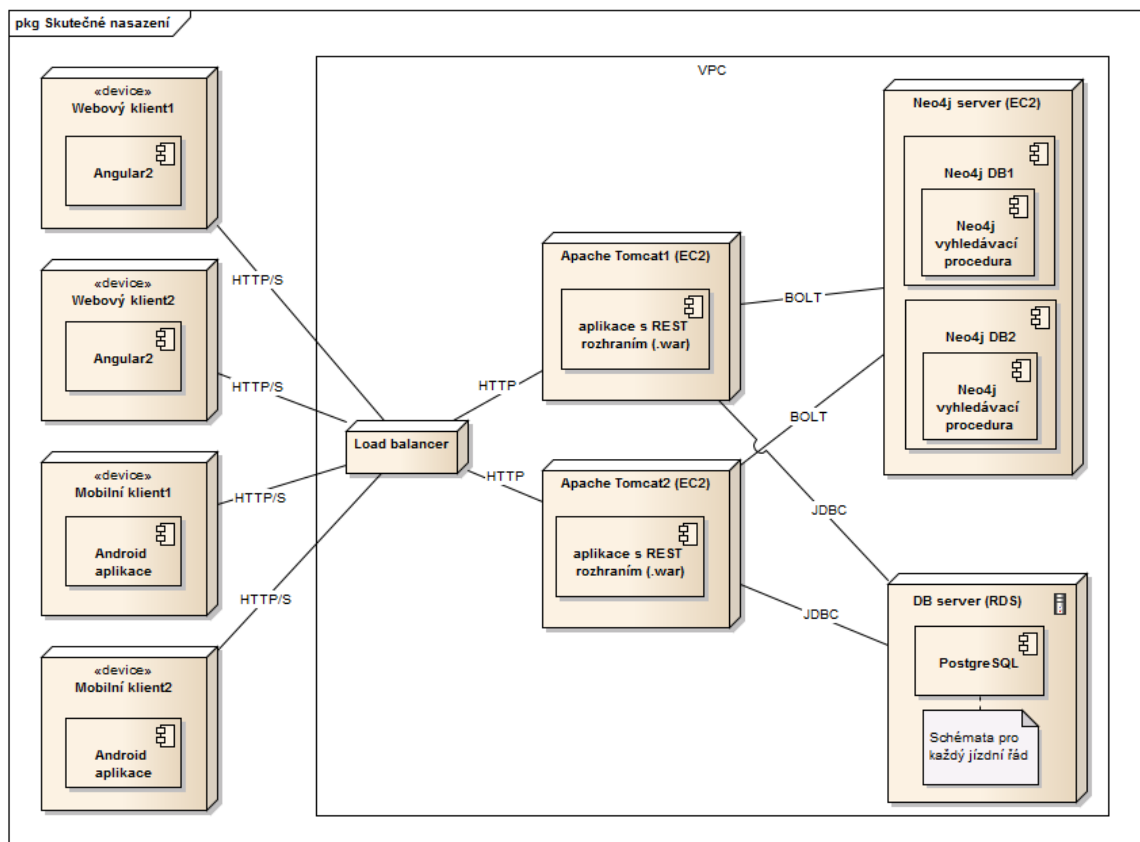
7.2 Nasazení mobilní aplikace

Mobilní aplikace pro platformu Android byla pod názvem DP-NSS[14] nahrána do distribuční služby Google Play[15], odkud je k dispozici ke stažení na libovolné podporované zařízení s operačním systémem Android.

Pro nahrání aplikace do služby Google Play je nutné si zaregistrovat vývojářský účet ve službě Play Console[16] a zaplatit jednorázový poplatek ve výši 25 $. Poté je již možné nahrávat do knihovny libovolný počet aplikací a proces uveřejnění aplikace je díky přehlednému uživatelskému rozhraní služby poměrně jednoduchý.

7.3 Výsledné schéma nasazení

Výsledné schéma nasazení dle popisu z této kapitoly můžeme pozorovat na obrázku 7.1.



Obrázek 7.1: Model skutečného nasazení.

[14] https://play.google.com/store/apps/details?id=cvut.cz.dp.nss
[15] https://play.google.com/store
[16] https://support.google.com/googleplay/android-developer/answer/6112435?hl=cs

Kapitola 8

Závěr

Cílem této práce bylo analyzovat, navrhnout, implementovat, otestovat a nasadit webovou a mobilní aplikaci pro vyhledávání spojů veřejné dopravy nad grafovou databází Neo4j. K tomuto účelu bylo převzato jádro vyhledávacího algoritmu nad databází Neo4j z bakalářské práce *BPJCH* [3], které ovšem muselo být upraveno a také rozšířeno o dosud neimplementované části.

8.1 Analýza

V analýze jsme se nejprve věnovali prozkoumání existujících podobných řešení a následně jsme analyzovali dostupné formáty pro ukládání dat o jízdních řádech. Díky těmto krokům jsme byli dále schopni definovat detailní požadavky, které budeme na náš vyhledávač klást. V neposlední řadě jsme definovali uživatelské role, které bude muset naše aplikace podporovat.

8.2 Analýza vyhledávání spojů v grafu

V této kapitole jsme se věnovali analýze existujících přístupů pro ukládání dat o jízdních řádech do grafové struktury. Na základě provedené analýzy jsme si představili námi zvolené řešení struktury grafu a zejména pak navrhli vlastní vyhledávací algoritmus, který bude sloužit pro vyhledávání spojů. Kapitola z velké části vycházela z bakalářské práce *BPJCH* [3], která se již tímto tématem zabývala. Strukturu grafu i vyhledávací algoritmus jsme ale zefektivnili a rozšířili o dosud neimplementované části.

8.3 Návrh

Zatímco v rámci analýzy jsme popsali aplikaci z pohledu *co budeme chtít implementovat*, v rámci návrhu jsme navrhli způsob, *jakým to budeme implementovat.*

Navrhli jsme tak způsob ukládání dat o jízdních řádech a kompletní doménový model pro ukládání dat do relační databáze. Dále jsme provedli návrh architektury, a zejména pak REST rozhraní, které bude serverová část aplikace vystavovat. Pokračovali jsme návrhem

způsobu, jakým spolu budou jednotlivé komponenty systému komunikovat a pro tyto účely i vytvořili zamýšlené schéma výsledného nasazení. V neposlední řadě jsme pak vytvořili řadu návrhů uživatelských obrazovek, dle kterých by měla vzniknout výsledná aplikace.

8.4 Implementace

Kapitola Implementace se nejprve zabývá výběrem a zdůvodněním zvolených technologií. Poté již přechází k popisu vlastní implementace, kdy se zaměřujeme zejména na popis zajímavých implementačních detailů, čímž čtenáři pomáháme přiblížit vlastní proces implementace. V závěru kapitoly jsou k dispozici ukázky uživatelského rozhraní obou implementovaných aplikací.

8.5 Testování

V kapitole Testování se věnujeme testování aplikací ze tří pohledů. Kromě testování samotného kódu pomocí jednotkových a integračních testů se tak věnujeme i zátěžovému a výkonnostnímu testování. V rámci výkonnostního testování jsme provedli otestování rychlosti vyhledávacího algoritmu pro různé vstupní parametry. Zátěžové testování naproti tomu otestovalo připravenost aplikace na nápor velkého množství uživatelů, kteří začnou vyhledávat spoje v jeden okamžik.

Posledním typem provedených testů bylo uživatelské testování, kterého se účastnilo několik participantů, kteří v mobilní i webové aplikaci postupně procházeli předem definované scénáře. Díky uživatelskému testování bylo identifikováno několik nálezů různých priorit.

8.6 Nasazení

Kapitola Nasazení popisuje proces nasazení webové aplikace do veřejného cloudu Amazon AWS a mobilní aplikace do knihovny Google Play.

8.7 Zhodnocení výsledků a budoucí rozšíření

Všechny požadavky, které jsme si vytkli v zadání práce, byly splněny a výstupem této práce je tak komplexní a reálně nasazená webová a mobilní aplikace, která je komukoliv dostupná k okamžitému využití například pro vyhledávání nad jízdními řády Pražské integrované dopravy.

V rámci uživatelského testování bylo odhaleno několik problémů, které bude nutné zejména s ohledem na přívětivost administračního rozhraní webové aplikace opravit. Právě administrační rozhraní totiž momentálně ještě není zcela uživatelsky přívětivé pro správu jízdních řádů velkého rozsahu, kterými jsou například jízdní řády Pražské integrované dopravy, a přidávání zejména nových zastavení jízd je tak problematické.

Ze zatím neimplementovaných funkcionalit by měla přibýt možnost postupného nahrávání jízdních řádů ve formátu GTFS. Aplikace totiž momentálně umí nahrát jízdní řád pouze

jednorázově, a pokud ke konkrétnímu jízdnímu řádu nahrajeme nová data ve formátu GTFS, dojde ke smazání dat starých. V neposlední řadě je stále prostor pro lepší optimalizaci vyhledávacího algoritmu. Ten sice pro většinu vyhledávání i nad kompletními daty Pražské integrované dopravy nalezne výsledky velmi rychle, pro určitá specifická vyhledávání ale trvá nalezení výsledků i několik desítek vteřin.

Literatura

[1] Zákon č. 77/2002 Sb., o akciové společnosti České dráhy, státní organizaci Správa železniční dopravní cesty a o změně zákona č. 266/1994 Sb., o dráhách, ve znění pozdějších předpisů, a zákona č. 77/1997 Sb., o státním podniku, ve znění pozdějších předpisů. *Sbírka zákonů*. 2002.

[2] Bileto.com. *Moderní technologie pro veřejnou dopravu* [online]. 2017. [cit. 8. 4. 2017]. Dostupné z: http://www.bileto.com/cs/features.html.

[3] CHALUPA, J. Využití grafových databází pro vyhledávání spojů veřejné dopravy. Bakalářská práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, Česká republika, 2015.

[4] EVROPSKÁ KOMISE. *Doprava: Vítězem první soutěže inteligentní mobility se stává ...* [online]. 2012. [cit. 6. 4. 2017]. Dostupné z: http://europa.eu/rapid/press-release_IP-12-233_cs.htm.

[5] FINLEY, K. *5 Graph Databases to Consider* [online]. 2011. [cit. 2. 4. 2015]. Dostupné z: http://readwrite.com/2011/04/20/5-graph-databases-to-consider.

[6] FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern* [online]. 2004. [cit. 26. 4. 2017]. Dostupné z: https://martinfowler.com/articles/injection.html.

[7] FOWLER, M. *Polyglot Persistence* [online]. 2011. [cit. 9. 4. 2017]. Dostupné z: https://martinfowler.com/bliki/PolyglotPersistence.html.

[8] Google, Inc. *Google developers: Dashboards* [online]. 2017. [cit. 26. 4. 2017]. Dostupné z: https://developer.android.com/about/dashboards/index.html.

[9] Google, Inc. *Google developers: What is GTFS?* [online]. 2012. [cit. 2. 4. 2017]. Dostupné z: https://developers.google.com/transit/gtfs/.

[10] JANČURA, R. *Jak to všechno začalo* [online]. 2017. [cit. 8. 4. 2017]. Dostupné z: https://www.regiojet.cz/o-nas/nas-pribeh.

[11] KALLA, T. Analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití. Master's thesis, Masarykova univerzita, Fakulta informatiky, Česká republika, 2015.

[12] KOLÁŘ, J. *Teoretická informatika.* V Praze : České vysoké učení technické, vyd. 1. edition, 2009. ISBN 9788001043318.

[13] MALÝ, M. *Školení: PostgreSQL efektivně* [online]. 2010. [cit. 26. 4. 2017]. Dostupné z: http://www.zdrojak.cz/zpravicky/skoleni-postgresql-efektivne/.

[14] MIČKA, P. *Java pro začátečníky (21) - Kolekce* [online]. 2011. [cit. 9. 4. 2017]. Dostupné z: http://www.algoritmy.net/article/34009/Kolekce-21.

[15] Neo Technology, Inc. *3.3.5. CREATE* [online]. 2017. [cit. 9. 4. 2017]. Dostupné z: https://neo4j.com/docs/developer-manual/current/cypher/clauses/create/.

[16] Neo Technology, Inc. *Neo4j with Docker* [online]. 2017. [cit. 15. 4. 2017]. Dostupné z: https://neo4j.com/developer/docker/.

[17] Neo Technology, Inc. *The Neo4j Java Developer Reference v3.1* [online]. 2017. [cit. 15. 4. 2017]. Dostupné z: http://neo4j.com/docs/java-reference/current/.

[18] Neo Technology, Inc. *From SQL to Cypher – A hands-on Guide* [online]. 2017. [cit. 9. 4. 2017]. Dostupné z: https://neo4j.com/developer/guide-sql-to-cypher/.

[19] NTT Data. *RESTful Web Service* [online]. 2015. [cit. 27. 4. 2017]. Dostupné z: http://terasolunaorg.github.io/guideline/5.0.0.RELEASE/en/ArchitectureInDetail/REST.html.

[20] ORDA, A. – ROM, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM (JACM).* 1990, 37, 3, s. 607–625.

[21] ORDA, A. – ROM, R. Minimum weight paths in time-dependent networks. *Networks.* 1991, 21, s. 295–319.

[22] PALLOTTINO, S. – SCUTELLA, M. G. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects, 1998.

[23] PEVNÝ, J. *Idos.cz - Jízdní řády na internetu snadno a rychle* [online]. 2010. [cit. 02. 4. 2015]. Dostupné z: http://www.swmag.cz/718/idos-cz-jizdni-rady-na-internetu-snadno-a-rychle/.

[24] PYRGA, E. et al. Efficient Models for Timetable Information in Public Transportation Systems. *J. Exp. Algorithmics.* June 2008, 12, s. 2.4:1–2.4:39. ISSN 1084-6654. doi: 10.1145/1227161.1227166. Dostupné z: http://doi.acm.org/10.1145/1227161.1227166.

[25] Přispěvatelé Wikipedie. *IDOS* [online]. 2014. [cit. 6. 4. 2017]. Dostupné z: http://cs.wikipedia.org/wiki/IDOS.

[26] SANDERS, P. – MANDOW, L. Parallel Label-Setting Multi-objective Shortest Path Search. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, s. 215–224, May 2013. doi: 10.1109/IPDPS.2013.89.

[27] SAVOV, V. *The entire history of iPhone vs. Android summed up in two charts* [online]. 2016. [cit. 26. 4. 2017]. Dostupné z: http://www.theverge.com/2016/6/1/11836816/iphone-vs-android-history-charts.

[28] SCHULZ, F. Timetable Information and Shortest Paths. Master's thesis, der Universität Fridericiana zu Karlsruhe (TH), der Fakultät für Informatik, Deutschland, 2005.

[29] SCHULZ, F. – WAGNER, D. – WEIHE, K. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *JOURNAL OF EXPERIMENTAL ALGORITHMICS*. 2000, 5, 12, s. 2000.

[30] SLÍŽEK, D. *Monopol na data z jízdních řádů končí. CHAPS se dohodl s ministerstvem* [online]. 2015. [cit. 2. 4. 2017]. Dostupné z: http://www.lupa.cz/clanky/monopol-na-data-z-jizdnich-radu-konci-chaps-se-dohodl-s-ministerstvem.

[31] SLÍŽEK, D. *Tohle je výsměch. Úplné uvolnění dat o jízdních řádech CHAPSem se zase nekoná* [online]. 2015. [cit. 2. 4. 2017]. Dostupné z: http://www.lupa.cz/clanky/tohle-je-vysmech-uplne-uvolneni-dat-o-jizdnich-radech-chapsem-se-zas-nekona.

[32] VYLEŤAL, M. *Tomáš Chlebničan (CHAPS): Data, která má Bileto a Seznam, pocházejí od nás* [online]. 2014. [cit. 6. 4. 2017]. Dostupné z: http://www.lupa.cz/clanky/tomas-chlebnican-chaps-data-ktera-ma-bileto-a-seznam-pochazeji-od-nas/.

[33] ZAJÍČKOVÁ, L. – BŘEČKA, P. Datový model dopravní sítě pro správu dat a řízení veřejné hromadné dopravy. [online], 2013. Dostupné z: http://gis.tuzvo.sk/tiki-download_file.php?fileId=140.

Příloha A

Obsah přiloženého CD

```
abstract ............................................ abstrakt práce
├── abstract-cz.txt
└── abstract-en.txt
readme.txt .......................................... popis obsahu CD
src
├── dp-nss.zip ......................... zdrojové kódy webové aplikace
├── dp-nss-search.zip ........ zdrojové kódy vyhledávacího pluginu Neo4j
├── dp-nss-android.zip ........ zdrojové kódy mobilní Android aplikace
└── chaluja7-diploma-thesis-2017.zip ... zdrojová forma práce ve formátu LaTeX
test
└── dp-nss-test-plan.jmx ...... definice zátěžových testů pro aplikaci JMeter
text
└── chaluja7-diploma-thesis-2017.pdf ........ text práce ve formátu PDF
```