

Gymnázium, Praha 6, Arabská 14

Programování
Ročníkový projekt
Jízdní řády



Prohlašujeme, že jsme jedinými autory tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V Praze dne 30.dubna 2018

Obsah

Úvod	5
Vstup.....	5
Parametry pro určení optimální cesty.....	5
Finální verdikt	6
Serverová backend aplikace	7
Použité technologie	7
Neo4j	7
SQL Server	8
SQL v JAVA	8
GTFS formát dat.....	9
Struktura grafu	12
Spring framework	15
@BEANS.....	15
@Value	16
@Service a @Autowired	16
@Scheduled.....	17
REST API.....	17
Java neo4j traversal api a algoritmus na vyhledávání trasy	19
Aplikace pro android	22
Použité technologie	22
Android	22
XML.....	22
SQLite	22
OkHttp	22
Osmdroid	22
Websocket.....	22
Základní ustanovení pro tvorbu aplikace	24
Funkce aplikace	24
Vyhledávání	24
Mapa.....	25
Oblíbené trasy	26
Komunitní prvky	26
Plány do budoucna	Chyba! Záložka není definována.

Závěr	28
Bibliografie	29
Seznam obrázků:	30

Úvod

Cílem této práce bylo vymyslet, vytvořit a následně zprovoznit systém pro vyhledávání a upravování spojů PID. Původní plán vznikl při zkoumání různých hotových řešení pro vyhledávání spojů, kdy naprostá většina řešení pracuje s relačními databázemi, což vzhledem k podstatě problému, který přímo vybízí k řešení pomocí grafu připadalo nelogické. Při použití grafové databáze se eliminuje přechod mezi mnoha tabulkami a podle prvotního průzkumu se signifikantně krátí čas potřebný k traverzování jednotlivými trasami. Jako další výhodu tato implementace se jeví jednoduchost implementace komunitních prvků, viz kapitola „komunitní prvky“

V této kapitole se zamyslíme nad podstatou vyhledávacích spojů a určíme strukturu, podle které by měl náš vyhledávací program fungovat. Určíme také, jaké základní požadavky by měla naše implementace uspokojovat.

Vstup

Analýzou konkurenčních programů, kdy se parametry vyhledávání hodně podobají anebo jsou úplně totožné jsme došli k závěru, že by náš vyhledávač měl integrovat následující parametry:

1. Výchozí stanice
2. Cílová stanice
3. Maximální počet přestupů
4. Vyhledávání podle času nejen odjezdu, ale i příjezdu

K podonému závěru došla i *práce analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití¹*.

Ovšem pro usnadnění práce jsme se rozhodli integrovat jen prvky 1-3, kdy budeme vyhledávat vždy jen podle času odjezdu. Implementace algoritmu, který by hledal cestu podle času příjezdu by byla podobná, jen by se za počáteční stanici brala stanice cílová a algoritmus by běžel „po zpátku“.

Parametry pro určení optimální cesty

Jako optimální cestu považujeme cestu takovou, která

1. Je časově nejméně náročná, tzn. Její průjezd zabere nejméně času
2. Má k času jízdy také úměrně nízký počet přestupů
3. Nízké riziko „zameškatelnosti“ spoje

Parametry 1 a 2 se dají relativně snadno ohlídat. Určení délky spoje je triviální, stejně tak náš algoritmus bude implementovat zámeček na počet přestupů, aby se nestalo že sice trasa bude nejrychlejší, ale počet přestupů bude enormní. Ovšem problém nastává u rizika zmeškání spoje, kdy je tato informace reálně neměřitelná bez delších experimentů na velké masě lidí. V praxi se to řeší implementací časového údaje do přestupové hrany. To ovšem vyžaduje, jak už jsme zmínili delší pozorování časů potřebných na přestup na každé individuální zastávce a graf by se pro naše účely zbytečně větvil a komplikoval. Proto zavedeme fixní čas na přestup, kdy $x = 2$ min. Tento parametr je naprosto nespecifický a může se nám stát, že spoj tudíž nelze nijak stihnout, ovšem v měřítku

¹ KALLA, T. Analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití. Master's thesis, Masarykova univerzita, Fakulta informatiky, Česká republika, 2015

naší práce je to bohatě dostačující. Díky tomuto usnadnění se náš okruh sledování sníží pro nás na dva, nyní již snadno ohlídatelné faktory 1 a 2.

Finální verdikt

Díky těmto definovaným hodnotám lze výsledky vyhledávání mezi sebou porovnávat a poté z toho vyvodit „lepší spoje“. Jako například když bude absolutní čas dvou spojů stejný, jako lepší se bere spoj s menším počtem přestupů a vice versa.

Serverová backend aplikace

Použité technologie

Neo4j

Neo4j je systém pro správu databází grafů. Byl vyvinut společností Neo4j Inc., která byla založena v roce 2000. Je popsána jako nativní grafová databáze vytvořená pro použití datových vztahů pro složitější a inteligentnější aplikace. Je vytvořena v programovacím jazyce java.

První verze byla vytvořena v roce 2002. Verze 1.0 byla vydána v roce 2010 a nejnovější verze (4.0.3) měla datum vydání 30.3.2020.

Neo4j je využívána mnoho velkými společnostmi jako aukční síť eBay, americký řetězec obchodních domů Walmart nebo americká vládní agentura NASA.

Existují tři verze: veřejná, firemní a Neo4j Aura. Veřejná je zdarma a je určena pro osobní a menší projekty, má lehčí ovládání některé funkce jsou omezené nebo nedostupné. Firemní verze je placená, ale za to lze použít všechny funkce které Neo4j nabízí. Nejnovějším typem je Neo4j Aura který je automatizovaný a funguje na online úložišti, aby vaši databázi mohli spravovat přímo zaměstnanci Neo4j, Inc. Firma také uvádí že systém je pak 1000x rychlejší než u konkurenčních firem.

Jakožto standalone verze ještě funguje 4tá – neo4j embedded. Tato verze běží nativně v naší aplikaci. To má určité výhody, lze použít vcelku zdařilé neo4j traversal API, které slouží k propracovanějšímu traversování grafem.

Cypher

Pro komunikaci s Neo4j se používá jazyk cypher. Ten je obzvláště dobře strukturovaný a v mnohém se podobá SQL, odkud má i značnou míru převzatých klíčových slov.

Základní patern pro komunikaci cypheru je:

```
(a:NejakaBunka {nejakyParametr: "foo"}) -  
[rel:ODKAZUJE_NA_DRUHOU_BUNKU] -> (b:NejakaJinaBunka  
{nejakyJinyParametr: "foo"})
```

Pomocí jazyku cypher však lze tvořit i komplikované query, jako:

```
:auto USING PERIODIC COMMIT 50000 LOAD CSV WITH HEADERS FROM  
"file:///D:/Onedrive/Plocha/jizdnirady/guide_transfers.txt"  
AS row FIELDTERMINATOR ',' Match (a:TripStop) where  
id(a)=tointeger(row.id) Match(b:TripStop) where  
id(b)=tointeger(row.previous_id) create (a)<-[r:Prestup]-(b),  
(b)<-[x:Prestup]-(a)
```

Cypher je skvělý a mocný nástroj. Ovšem při práci jsme narazili na nějaké jeho limity, které jsme se rozhodli řešit pomocí nativní komunikace přes pro neo4j embedded nativní formě komunikace – Transaction, nebo implementací neo4j traversal API. Ovšem pro běžné účely je naprosto dostačující, přičemž v 95 % případů je to nejrychlejší a nejefektivnější volba komunikace s databází.

SQL Server

Je to relační databázový systém vyvinutý společností Microsoft a IBM převážně pro řešení velkých objednávek, účetnictví nebo skladů pro firmy.

Funguje na principu klient/server. Podpora jazyka XML, SQL a JSON je integrovaná. Podporován je také jazyk R, který se používá k pokročilým analýzám dat. Také ukládá události ze serveru, které lze později analyzovat.

Umožňuje správu síťové konektivity a konfiguraci síťových protokolů.

SQL Server sám je vyvinut v programovacích jazycích C, C++ a C#.

Verze 1.0 byla vydána 24. dubna 1989, nejnovější verze v roce 2019, podle toho i nese název SQL Server 2019.

Pro komunikaci s databází se používá standardní dotazovací jazyk SQL (structured query language)

SQL je dotazovací jazyk. Každý takový dotaz má danou strukturu (zjednodušeně):

1. Co chceme provést
2. Kolik toho chceme provést
3. Kde to chceme provést

Příklad:

```
SELECT * FROM [TestDB].[dbo].[Sklad1] WHERE id=1
```

Obrázek 1. ukázka SQL příkazu

1. SELECT – dotaz, který vrátí položky z určitého místa
* vybere všechny buňky ve vybrané řádce
2. FROM – určí tabulku, ve které se bude vykonávat dotaz
V tomto případě je to v tabulce Sklad1 v databázi TestDB
3. WHERE – upřesňuje řádku v tabulce
V tomto případě vybere všechny řádky, kde je id=1

SQL v JAVA

Ke komunikaci s SQL serverem jsme zvolili knihovnu JPA, která nám značně usnadňuje migrace, dotazování a upravování tabulek ve Springu, přičemž je již z nějaké části ve Springu nativně implementovaná. Vzhledem k volbě grafové databáze, jako hlavní médium pro hledání a upravování spojení, budeme SQL databázi používat pouze pro přihlašování do API, tvorbu a schraňování aktuálních aktivních „komunitních eventů“ tedy událostí, které uživatel zadá a které určitým způsobem ovlivňují chod dopravy.

Spring JPA

Základem JPA je interface třída obsahující strukturu tabulky², na kterou budeme chtít vytvořit migraci. Ta se chová, jako šablona pro repositář

² *Spring.io* [online]. 2020. Dostupné z: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.core-concepts>


```

10
11 @Entity
12 @Table(name = "Users")
13 public class User {
14
15     @Id
16     @GeneratedValue(strategy= GenerationType.IDENTITY)
17     private int Id;
18
19     private String userName;
20     private String password;
21     private boolean active;
22     private String roles;
23
24
25     public int getId() { return Id; }
26
27
28     public void setId(int id) { Id = id; }
29
30
31     public String getUserName() { return userName; }
32
33     public void setUserName(String userName) { this.userName = userName; }
34
35
36     public String getPassword() { return password; }
37
38     public void setPassword(String password) { this.password = password; }
39
40
41     public boolean isActive() { return active; }
42
43     public void setActive(boolean active) { this.active = active; }
44
45
46     public String getRoles() { return roles; }
47
48     public void setRoles(String roles) { this.roles = roles; }
49
50     public List<GrantedAuthority> getAuthorities(){
51         return Arrays.stream(roles.split( REGEX: ";" )).map(SimpleGrantedAuthority::new).collect(Collectors.toList());
52     }
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

Obrázek 2. Ukázka třídy s implementovanou JPA

K vlastní komunikaci se používá interface **Repository**, který po implementaci tvoří sám SQL příkazy bez toho, aniž by jsme si je museli definovat. Příkladem je metoda `findByUserName`, která vyhledá v databázi záznam, který odpovídá `userName` parametru a naplní s ním `User` třídu.

```

public interface UserRepository extends JpaRepository<User, Integer> {
    Optional<User> findByUserName(String userName);
}

```

Obrázek 3. Ukázka implementovaného interface `Repository` s metodou `findByUserName`

GTFS formát dat³

Obecná specifikace formátu GTFS je definovaná, jako společný formátů plánů veřejné dopravy a s tím souvisejícími geografickými informacemi. Vlastně jde o podrobné znázornění ulic pro vozidla i chodce.

³ Lenka ZAJÍČKOVÁ, Katedra geoinformatiky UPOL Patrik BŘEČKA, AssecoCentralEurope, a.s. DATOVÝ MODEL DOPRAVNÍ SÍTĚ PRO SPRÁVU DAT A ŘÍZENÍ VEŘEJNÉ HROMADNÉ DOPRAVY [online], 2013, dostupné z: <https://docplayer.cz/23511631-Datovy-model-dopravni-site-pro-spravu-dat-a-rizeni-verejne-hromadne-dopravy.html>

Plánování cesty je tvořeno kombinací dat z GFTS, OpenStreetMap a softwaru s otevřeným zdrojovým kódem OpenTripPlanner.

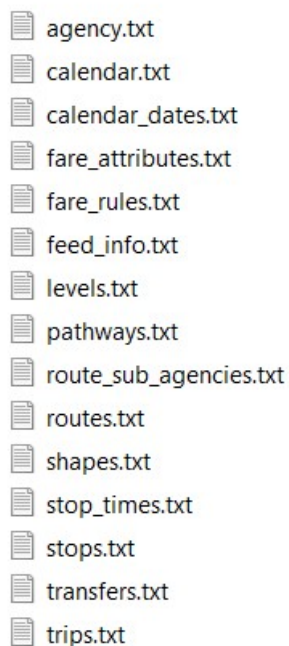
Standardní kolekce GFTS je tvořena 6 až 13 soubory CSV uložených v soubory .zip. Tabulky CSV popisují naplánované operace. Také používá kalendář pro znalost dnu v týdnu a s tím spojenými změnami.

GTFS bylo jako formát popsáno a „vydáno“ 27. září 2007 s názvem Google Transit Feed Specificationa roce 2009 byl název změněn na The General Transit Feed Specification.

Struktura PID GTFS

GTFS formát, který pro páci využíváme je dostupný z pid.cz v sekci otevřená data⁴, na webu lze také dohledat stručnou dokumentaci k jednotlivým položkám .zip souboru. Tento archiv souborů je updatován denně s minimálně desetidenní lhůtou platnosti. Strukturu našeho serveru jsme designovali tak, aby si každý den ráno (byla zvolena 3 hodina ranní kdy jezdí nejméně spojů, aby nevznikaly zbytečné komplikace se zpomalováním chodiú algoritmů) přemazal databázi s jízdami a nahradil jí novými daty. Z tohoto faktu vychází maximální počet dnů dopředu na které lze vyhledávat spojení- 10dní dopředu a zároveň z důvodu zjednodušení přemazávání dat- znemožnění hledání spojů do minulosti.

Náš GTFS balík má následující formát:



- agency.txt
- calendar.txt
- calendar_dates.txt
- fare_attributes.txt
- fare_rules.txt
- feed_info.txt
- levels.txt
- pathways.txt
- route_sub_agencies.txt
- routes.txt
- shapes.txt
- stop_times.txt
- stops.txt
- transfers.txt
- trips.txt

Obrázek 4, struktura PID GTFS balíku

Nejdůležitější části GTFS balíku a jejich stručný popis

⁴ Pid.cz: Pražská integrovaná doprava [online]. 2020. Dostupné z: <https://pid.cz/o-systemu/opendata/>

stops.txt

- V souboru jsou pouze zastávky, které jsou aktuálně využívány, tj. alespoň v některý den platnosti feedu v zastávce zastavují nějaké spoje. Pokud je zastávka (i dočasně) mimo provoz, nebude obsažena.
- Stanice metra mají dvě zastávky (pro každý směr jednu) sdružené do jedné stanice. stop_id této stanice je ve tvaru Učíslo_uzluIndex. U přestupních stanic metra jsou pak zastávky čtyři, rozdělené do dvou stanic, viz níže.

Struktura a příklad:

stop_id	stop_name	stop_lat	stop_lon	zone_id	stop_url	location_type	parent_station	wheelchair_boarding	level_id	platform_code
U50S1	Budějovická	50.04441	14.44879	P		1		1	1	
U52S1	Chodov	50.03167	14.49096	P		1		1	1	

Obrázek 5. struktura stops.txt

Stop_times.txt

- Nejdůležitější dokument pro náš program, obsahuje všechny informace potřebné ke tvorbě základní struktury námi modifikované verze time-expanded modelu
- Popisuje cestu jednoho spoje v rámci dne, čas příjezdu do zastávky, čas odjezdu a obsahuje také odkazy na dny platnosti tohoto popisu, díky tomuto lze poté jednoduše základní kostru grafu
- shape_dist_traveled popisující vzdálenost od počátku na trase má vždy svůj protějšek v shapes.txt (vždy existuje v shapes.txt záznam s identickou souřadnicí), aby bylo možné snadno dělit trasu na mezizastávkové úseky.

Struktura a příklad:

trip_id	arrival_time	departure_time	stop_id	stop_sequence	stop_headsign	pickup_type	drop_off_type	shape_dist_traveled
991_1398_180709	5:52:15	5:52:15	U953Z102P	1		0	0	0.00000
991_1398_180709	5:54:20	5:54:50	U713Z102P	2		0	0	1.38944

Obrázek 6. struktura stop_times.txt

calendar.txt

- service_id obsahuje řetězec sedmi nul a jedniček, které pro každý den v týdnu udávají, kdy spoj jede pro vyšší lidskou čitelnost
- Platnost je vždy minimálně 10 dní ode dne vygenerování, vlaky se generují na celý grafikon.
- Soubor používáme k ustanovení dnů platnosti dané jízdy

Struktura a příklad:

service_id	monday	tuesday	wednesday	thursday	friday	saturday	sunday	start_date	end_date
1111100-1	1	1	1	1	1	0	0	20200429	20200512
1111111-1	1	1	1	1	1	1	1	20200429	20200512

Obrázek 7. struktura calendar.txt

Tyto tři soubory jsou pro náš program nejdůležitější, proto pro jednoduchost dokumentace popíšeme jen je. Dokumentaci ke zbytku jak jsme již zmínili lze najít na pid.cz.

Struktura grafu

Po detailním nastudování grafové teorie a různých prací na téma grafové teorie a její implementaci v reálném světě se lze dozvědět, že existují dva zásadní typy grafů, které se ve své podstatě zásadně liší. Time-dependent⁵ a time-expanded model.

Time dependent graf

V tomto grafu jsou vrcholy jednotlivé zastávky. Mezi jednotlivými vrcholy existuje hrana tehdy, kdy existuje i spojení mezi těmito stanicemi. Pokud ovšem existuje více takových spojení, hrana zůstává vždy jen jedna. Díky této podmínce se drasticky sníží počet hran v grafu, ovšem za cenu faktu, že hrana nemůže mít jasně definovanou cenu (tedy čas v našem případě) a je nutno ji přepočítávat při traverzování grafem díky rozdílu času výjezdu a času příjezdu.

O bližším popisu struktury se lze dočíst například v práci **contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router**⁶ nebo v práci **Minimum weight paths in time-dependent networks**⁷

time-expanded graf

V tomto modelu jsou v roli vrcholů události na zastávkách. Událostmi na zastávkách je myšleno příjezd do stanice, odjezd ze stanice. Tyto vrcholy jsou spojeny dle pořadí konání tzn. Autobus nejdříve vyjede ze stanice A hrana na příjezd na stanici B hrana na výjezd z B atd. v Grafu je nutné definovat tedy dva typy hran, hrany traverzní a hrany přestupové. Hrany traverzní spojují zastávku s událostí přímo následující v časové linii. Přestupová umožňuje traverzování na jiný spoj v rámci jedné zastávky. Tím pádem může mít každá taková hrana definovanou exaktní cenovou hladinu. Za povšimnutí stojí také fakt, že díky způsobu jakým grafová struktura funguje je umožněno přestupovat jen v rámci jízd jednoho dne, proto je také potřeba spojit poslední událost v rámci jednoho dne s první událostí dne následujícího.

O bližším popisu struktury se lze dočíst například v práci **An approach for modelling time-varying flows on congested networks**⁸ nebo **System optimal dynamic assignment for electronic route guidance in a congested traffic network**⁹.

⁵ Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, Madhav Marathe

Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router

ESA 2003, Lecture Notes in Computer Science, volume 2461 (2002), pp. 126-138

⁶ Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, Madhav Marathe Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router ESA 2003, Lecture Notes in Computer Science, volume 2461 (2002), pp. 126-138

⁷ Ariel Orda, Raphael Rom: Minimum weight paths in time-dependent networks Networks: An International Journal, 21 (1991)

⁸ M. Carey and E. Subrahmanian. An approach for modelling time-varying flows on congested networks. *Transportation Research B*, 34:157–183, 2000.

⁹ H. S. Mahmassani and S. Peeta. System optimal dynamic assignment for electronic route guidance in a congested traffic network. In *Urban Traffic Networks. Dynamic Flow Modelling and Control*, pages 3–37. Springer, Berlin, 1995.

Prvotní záměr byl použit z očividných důvodů (méně hran = rychlejší chod algoritmu) time-dependent model. Ovšem struktura zvoleného GTFS formátu se skoro rovná modelu time-expanded. Díky autorům práce *Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries*¹⁰, kteří dokázali, že za určitých situací je sice time-dependent model rychlejší, avšak při složitějším vyhledávání se rozdíly mezi nimi mažou, time-expanded graf je dokonce pro jisté typy traversování lepší než time-dependent model. Takže s kombinací toho že je mu GTFS formát nejbližší to byla zvolená cesta.

Základní struktura

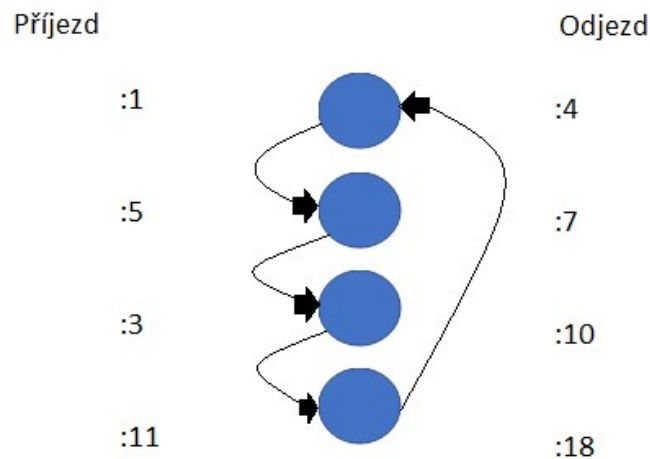
Zvolená struktura- time expanded model reprezentovaná vrcholy grafu, tedy jednotlivé události na zastávce, tzn. Příjezd, odjezd. Ovšem díky struktuře našeho balíku se mu můžeme ještě přiblížit a použít jako vrchol grafu kompletní zastavení jako souhrn a zmenšit tím tedy počet vrcholů grafu. Vrchol na zastávce (TripStop) obsahuje těchto 5 zásadních parametrů.

- Departure_time
- Arrival_time
- Trip_id
- Stop_id
- A pátý parametr svoje vlastní id, o ten se však stará databáze neo4j a nemusíme ho brát vůbec v potaz

Ted', když jsme vyřešili hlavní vrcholy musíme se postarat o hrany grafu. Ty se budou dělit na dva zásadní typy. Na hranu pokračující v rámci jedné jízdy POKRACUJE_DO a hranu přestupní PRESTUP. POKRACUJE_DO vyřešíme jednoduše, spojíme vždy zastávky se stejným trip_id s pořadím jdoucím striktně za sebou. Ovšem vzhledem k tomu že je počet zastavení poměrně velký a náš model počítá s pravidelnými updaty (aktuálně každé ráno se graf přepíše na nový GTFS balík je definovaný vždy na min. deset dní do předu) si musíme lehce hrát s query na update. Nejjednodušší způsob, kdy se vyzkouší všechny možnosti (složitost tudíž $O(N^2)$) při data setu o cca. 1 200 000 vrcholech by trval přibližně 5 h pro update jen POKRACUJE_DO hran (~166 000 000 000 jednotlivých úrovní vkládací procedury). Proto při tvorbě využíváme jednotlivých vlastností jazyka CYPHER kdy nejjednodušší (nejméně časově náročné) je vyhledávání podle id jednotlivé buňky, poté je přístup ke key hodnotám a až na konci k jednotlivým informacím.

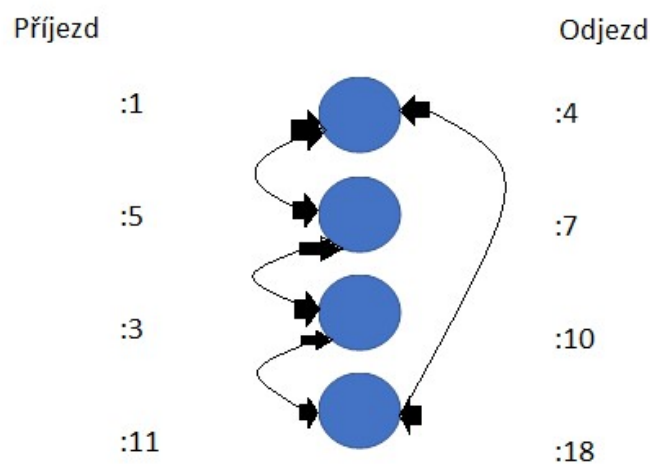
Při řešení přestupní hrany je to trochu složitější. První nápad byl ten, že seřadíme spoje v čase podle času odjezdu ze zastávky a spojíme je přestupní hranou, protože by se nám teoreticky nemělo vyplatit čekat na ten samý spoj, kterým jsme mohli jet už předtím. Ovšem tady vzniká problém, že někdy některé spoje nejezdí do stejných dep pokaždé a taky některé PID vlaky nefungují přímo podle tohoto modelu. Proto se jako další přestupový model nabízí následující model:

¹⁰ BRODAL, G. S. – JACOB, R. Time-dependent Networks as Models to AchieveFast Exact Time-table Queries.Electronic Notes in Theoretical Computer Science.2004, 92, 0, s. 3 – 15. ISSN 1571-0661. doi: <http://dx.doi.org/10.1016/j.entcs.2003.12.019>. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S1571066104000040> . Proceedings of {ATMOS} Workshop 2003.



Ovšem při lepším prozkoumání graf neobsahuje všechny možné přestupy, protože z buňky tři jde přestoupit i na buňku dva nejen na číslo 4. ovšem ani tohle není dostačující podmínka pro traversování na přestupu, protože ještě lze určitě přestoupit z buňky tři na buňku jedna. ovšem pokud spojíme buňku tři a jedna i „zpětnou“ hranou, bude graf obsahovat logický nesmysl, umožní přestup ze druhé buňky na buňku první, což zcela jistě v rámci jednoho dne zcela jistě možné není.

Z toho vyplývá myšlenka, že přestupní hrany budou obojetné a o traverzi se bude starat až výsledný algoritmus na vyhledávání trasy. Výsledný přestupní graf vypadá následně:

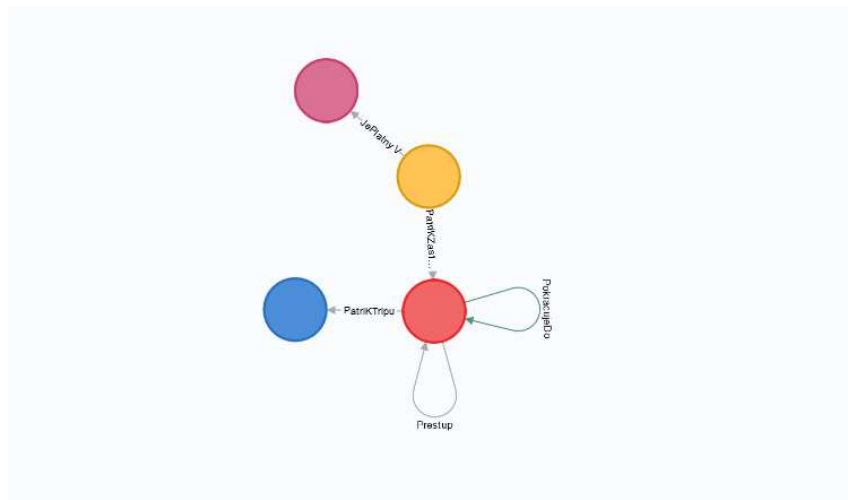


Nyní nám už chybí doplnit do grafu pouze vrcholy jednotlivých tripů, které spojují cestu jednoho spoje (tedy TripStopy v rámci jedné kompletní jízdy) do nadřazené množiny. Tento není přímo nutný, ovšem je přirozený z hlediska psaní algoritmu pro vyhledávání.

Stejný proces taky nastává ve spojení přestupů, díky jednomu Stop vrcholu. Ta spojuje všechny zastavení v rámci jedné zastávky a slouží čistě administrativním účelům, jako je tvorba přestupních hran a pomoc při hledání prvotní stanice na cestě.

Jako poslední věc v grafové části je nutnost identifikace, kdy je daná jízda platná. Toho lze docílit více způsoby, ovšem nejjednodušší a první logický je vytvořit nové vrcholy s datem splatnosti

(vrchol Calendar). Tyto vrcholy pak obsahují informace z calendar.txt, tedy sekvenci 7 booleanů, každá vyobrazující jeden den v týdnu a nabývající hodnotu podle platnosti jízdy v tomto dni. Tyto vrcholy pak už jen snadno navážeme na vrcholy tripů a máme hotovo. Tuto možnost používáme i přes větší počet vrcholů oproti ukládání parametrů přímo do tripStop vrcholu, z důvodu zmenšení dat, kdyby každý z vrcholů typu TripStop obsahoval navíc posloupnost 7mi booleanu, databáze by nám přetekla velikostně do obrovských měřítek a to pro denní update není přívětivé.



Obrázek 8. ukázka struktury finálního grafu – screenshot přímo z databáze

Na obrázku lze vidět tedy zvolenou strukturu. TripStop vrchol (červeně) odkazuje dvěma typy hranám na jiné svoje typy, přičemž je také připojený k dvěma typům jiných vrcholů: Trip (oranžově) a Stop (modře). Trip je také odpovídající k již vysvětlenému modelu vždy připojený ke Calendar vrcholu (purpurová).

Spring framework

Spring je aplikační rámec (framework) pro vývoj aplikací nad platformou Java. Je určen přede pro vším pro vývoj JEE aplikací... Framework je designovaný, jako open-source a patří mezi tzv. odlehčené J2EE kontejnery.

Celý systém Springu je navržen, jako kompletně modulární systém rozdělený do několika kategorií. Cílem této práce není popsat dopodrobna jednotlivé části, ale je nutno podotknout že porozumění Springu byla časově velmi náročná vzhledem k tomu, že nikdo z nás neměl původně žádné zkušenosti s vývojem JEE aplikací.

Spring má v sobě zabudované funkce na tagování tříd, proměnných, či metod, aby usnadnila určité procesy, bez nichž by byl vývoj značně komplikovanější. Některé použité v program si následně popíšeme.

@BEANS

Bean je základní kámen frameworku Spring, proto porozumění, co vlastně Bean je zásadní pro správný vývoj aplikace. Problém je ten, že Beans jako takový nemá úplně exaktní popis, který by obsahoval všechno, co potřebujeme. Definice z oficiální dokumentace Springu je:

„In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.“¹¹

~ “Ve Springu se objekty, které se starají o pozadí aplikace a jsou spravovány Spring IoC kontejnerem jmenují Beans. Beans je objekt, který je iniciován, sestaven a jinak spravován IoC kontejnerem.” ~

Což v principu odpovídá, ovšem odkazuje na IoC container. IoC container, též Inversion of Control container je jedno z modulů Spring frameworku. Jeho hlavním principem Inversion of Control je přesunout zodpovědnost za vytváření, inicializaci a propojení objektů z aplikace na framework. Zjednodušeně můžeme říct, že řeší těsné vazby mezi objekty aplikace, které jsou ve zdrojovém kódu těsně svázány. Jednotlivé objekty je tak možno získat pomocí tzv. svazování závislostí.

Zjednodušeně je Beans jednotlivá podsložka aplikace (například třída) označená příslušným tagem, o tuto třídu se náš program nestará a Spring framework si konstruktor vyvolá sám a přiřadí si k ní vše potřebné. Tím pádem nám, jako programátorům odpadá starost například s verzováním.

@Value

Tento tag se používá u proměnných. Slouží k importu hodnot z application.properties souboru přímo do vybrané proměnné bez nutnosti přepisování hodnot přímo ve zdrojovém kódu. Proměnná se jednoduše označí tagem s odkazem na daný záznam v application.properties a Spring IoC se postará o injektování dané hodnoty do proměnné. Je ovšem potřeba myslet na to, že Spring se o @Value tagy stará až po zavolání konstruktoru, takže je moudré je využít přímo v konstruktoru při inicializaci dané Třídy.

```
public Scheduling(  
    @Value("${update.unzip.source}") String source  
    , @Value("D:\\Onedrive\\Plocha\\jizdnirady\\GTFS") String destination
```

Obrázek 9. ukázka injektu hodnot do konstruktoru ve třídě Scheduling, odkaz (zeleně) a automaticky vyplněnou hodnotu díky IntelliJ pluginu (šedé pole)

@Service a @Autowired

Service tag a jemu podobné (například @Component) a s tím související Autowired tag jsou jednou z nejdůležitějších funkcí co základní práce s Beans ve Springu obnáší.

Service tag označuje komponentu programu, na kterou lze poté jednoduše odkázat z jiné části programu, aniž bychom museli třídu definovat. Po loptě je to takový ukazatel na třídu, která je schopna se „založit“ sama. K tomu nám právě slouží tag @Autowired, který z principu funguje, jako odkaz na třídu, která je importovaná pomocí Spring IoC.

¹¹Spring.io [online]. 2020. Dostupné z: <https://docs.spring.io>


```

@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        System.out.println(username + " username");
    }
}

```

Obrázek 10. Ukázka třídy, která je servisní třídou ve springu a obsahuje automaticky importovanou proměnnou typu `UserRepository`

@Scheduled

`Scheduled` tag není tak podstatný pro fungování Spring ekosystému jako takového, ale výborně se hodí k naší práci na automatizaci importu dat. Funguje totiž, jako autospouštěč určité metody. Buď se skvěle hodí k naplánování určité procedury, dejme tomu za patnáct minut od spuštění aplikace, nebo jako v našem případě použijeme cron formát pro plánování opakovaných procedur.

Cron ve stručnosti sestává z 5ti polí, které odkazují postupně na minuty, hodiny, dny v měsíci, měsíce, dny v týdnu, pokud chceme nějaký parametr vynechat, tak stačí doplnit `*` (stejně jako v SQL znak `*` značí všechno). Z tohoto vyplývá

0 0 * 8 * znamená **v 00:00 v srpnu**, nebo

0 6 5 8 4 znamená **v 06:00 pátého dne srpna a musí to být ke všemu čtvrtek**

Tedy lze si jednoduše představit, že se dá naplánovat v podstatě libovolně šílená procedura

```

@Scheduled(cron = "${update.cron}")
public void updateDatabase() throws InterruptedException {
    downloadFile();
    if (unzip() == true) {
        insertMainValues();
    } else {
        //v případě nepodaření unzipu program čeka 20 min a zkouší to znovu
        Logger.info("rescheduling data import in 20 mins");
        wait( timeoutMillis: 600000);
        updateDatabase();
    }
    insertMainValues();
}

```

Obrázek 11. Ukázka plánování metody

Za povšimnutí stojí, že můžeme u `@Scheduled` využívat stejného principu importu hodnot do proměnné, jako u `@Value`

REST API

Pro komunikaci s aplikací je nutno vytvořit nějaký komunikační proces. Původní plán byl použít WebSocket rozhraní, díky čemuž bychom byli schopni oboustranně komunikovat mezi serverem a aplikací. To by nám poskytlo velkou škálu možností na pozdější adice, v plánu jsou například směrované notifikace, real time mapa bez nutnosti znovu načítání pro aktuální data apod. Ovšem v práci jsme hodně zápasili s časem, takže jsme se rozhodli, že budeme používat pro implementaci jednodušší REST API, které nevyžaduje STOMP a speciálním message brokerem (v původním plánu byl vybrán RabbitMQ), jako WebSocket rozhraní se škálou možností, aby vůbec mělo důvod se této

možnosti věnovat. REST API bychom stejně psali, protože WebSocket rozhraní se také nehodí na všechna použití, takže to byl jen malý úskok.

Implementací existuje celá řada, avšak my si zvolili JWT a Spring Security knihovny. JWT (Jason web token) je komunikační standard určený k bezpečné komunikaci mezi serverem a cílovou aplikací. Používá se systém veřejného a privátního klíče, což znamená v podstatě, že aplikace má někde uložený klíč, respektive řetězec určitých hodnot, podle kterých se při tvorbě nějakého parametru, který chceme kontrolovat. Bývá to zpravidla heslo. Heslo při registraci program uloží do databáze společně se jménem, pro nás MSSQL a při přihlašování zkontroluje, jestli existuje dvojice jméno + heslo v databázi uložené. Pokud ano, vrátí jméno zašifrované pomocí privátního klíče – klíč veřejný. To se potom používá pro udržení komunikace... pokud veřejný klíč rozšifrovaný pomocí privátního klíče odpovídá zpátky jménu původnímu a není již token (veřejný klíč) expirovaný je jasné že člověk, který aktuálně komunikuje je přihlášený.

```
@RequestMapping(value = "/authenticate", method = RequestMethod.POST)
public ResponseEntity<> createAuthenticationToken(@RequestBody AuthenticationRequest authenticationRequest) throws Exception {
    try {
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(authenticationRequest.getUsername(), authenticationRequest.getPassword())
        );
    } catch (BadCredentialsException e) {
        return new ResponseEntity<>("Wrong Credentials", HttpStatus.FORBIDDEN);
    }

    final UserDetails userDetails = userService
        .loadUserByUsername(authenticationRequest.getUsername());
    final String jwt = jwtTokenUtil.generateToken(userDetails);

    return ResponseEntity.ok(new AuthenticationResponse(jwt));
}

@RequestMapping(value = "/register", method = RequestMethod.POST)
public ResponseEntity<> createAccount(@RequestHeader("username") final String username,
    @RequestHeader("password") final String password) throws Exception {
    if (userRepository.findByUserName(username).isPresent()) {
        return new ResponseEntity<>("userName used", HttpStatus.IM_USED);
    } else {
        String query = "INSERT INTO users (active, password, roles, user_name) values (1,?, 'USER_ROLE',?)";

        conn = sqlServerConnect.ConnectDB();
        pst = conn.prepareStatement(query);
        pst.setString(1, password);
        pst.setString(2, username);
        pst.executeUpdate();
    }

    AuthenticationRequest auth = new AuthenticationRequest();
    auth.setUsername(username);
    auth.setPassword(password);
}
```

Obrázek 12. ukázka autentikace, rozhraní, které parsuje hodnoty k jwt kontrole

Spring security se stará o vynucení přihlášení, o mapping doménových adres a zpracování levelů autorizace. Je zodpovědná o kontrolu přihlášení, některé její třídy tudíž musíme přepsat (@refactor), aby používali náš jwt kontrolní balíček, a ne standardní přihlášení skrz webové popup okno.

```

public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);
}

private Claims extractAllClaims(String token) {
    return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
}

private Boolean isTokenExpired(String token) {
    return extractExpiration(token).before(new Date());
}

public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    return createToken(claims, userDetails.getUsername());
}

private String createToken(Map<String, Object> claims, String subject) {
    System.out.println(subject+ "username generate token");
    return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact();
}

public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}

```

Obrázek 13. výňatek z třídy JwtUtils, která pracuje s jwt tokenama

Java neo4j traversal api a algoritmus na vyhledávání trasy

Dijkstrův algoritmus

Nejrozšířenější a nejznámější algoritmus na vyhledávání trasy v grafu mezi vrcholy je Dijkstrův algoritmus. Tento algoritmus můžeme využít pro hledání té nejkratší cesty vždy, když neobsahuje záporně ohodnocené hrany. Originální verze algoritmu byla určena pro hledání nejkratší cesty mezi dvěma vrcholy, dá se ale snadno modifikovat takovým způsobem, že nalezne nejkratší cesty od výchozího vrcholu ke všem ostatním vrcholům v grafu.

stručný popis funkčnosti algoritmu:

1. Ke každému uzlu přiřaď aktuální vzdálenost od počátečního uzlu. V případě samotného výchozího uzlu je vzdálenost 0, v případě ostatních uzlů je to nyní nekonečno.
2. Označ všechny uzly jako nenavštívené. Označ výchozí uzel jako aktuální uzel. Vytvoř množinu nenavštívených uzlů, která bude obsahovat všechny uzly kromě výchozího. Nyní se dostáváme k třetímu bodu, který se bude opakovat ve smyčce.
3. Podívej se na všechny nenavštívené sousedy aktuálního uzlu a vypočítej jejich vzdálenost od počátečního uzlu. Pokud má např. aktuální uzel vzdálenost 3 a délka hrany mezi tímto uzlem a jeho sousedem je 2, potom vzdálenost k tomuto sousedovi bude $3 + 2 = 5$. Pokud je aktuální vzdálenost menší než dříve zaznamenaná vzdálenost tohoto souseda, pak ji přepiš lepší, kratší hodnotou. Poznamenejme, že i když jsme se na sousední uzly aktuálního uzlu podívali, stále zůstávají v množině nenavštívených.
4. Jakmile jsme se podívali na sousedy aktuálního uzlu, budeme považovat tento aktuální uzel za navštívený a vyjmeme ho z množiny nenavštívených uzlů. K tomuto uzlu už se nikdy nevrátíme.
5. Pokud byla destinace označena jako navštívená, nebo nejmenší aktuální vzdálenost mezi uzly v množině nenavštívených uzlů je nekonečno, ukonči algoritmus.

Časová složitost Dijkstrova algoritmu je přímo závislá na výběru struktury, kterou používáme pro ukládání nenavštívených uzlů. Pro tyto účely se výborně hodí prioritní fronta. S využitím prioritní

fronty implementované jako Fibonacciho halda je časová složitost algoritmu rovna $O(M+N \log N)$, kde M je počet hran a N je počet vrcholů v grafu, zkráceně se dá použít $O(n \log n)$).

Vlastní algoritmus

Nejprve si musíme rozmyslet, mezi kterými vrcholy bude vlastně naše úprava Dijkstrova algoritmu cestu vyhledávat, již dříve když jsme si definovali strukturu grafu jsme poukázali na to, že každá jízda má svůj definovaný termín platnosti, což znamená, že nelze traverzovat po všech vrcholech. Graf je strukturovaný tak, že pokud stojíme na vrcholu, který má legitimní dobu splatnosti pro naši dobu, při přejezdu po přejezdové hraně je vždy další vrchol přístupný bez nutnosti testu platnosti. Z toho vyplývá fakt, že podmínky testování jízd na platnost za pomoci Calendar vrcholu nastane jen při použití hrany přestupní. Co se ovšem stane, když při traverzování za pomoci přestupní hrany narazíme na vrchol mimo aktuální dobu platnosti? Traverzování grafem v dané větvi nekončí, ovšem nelze již použít hrany přejezdové a algoritmus je nucen přestoupit. Tímto jsme definovali jedinou nutnou podmínku pro platnost časových intervalů v rámci jednoho dne.

Pro začátek algoritmu je nutno najít počátek traverzování. Při lehce hlubším zamyšlení nebude pouze jeden. Jako počáteční vrchol budeme považovat kolekci TripStopů, které jsou všechny příslušné jedné zastávce a mají dobu platnosti \leq času počátku vyhledávání. Tuto kolekci si přiřadíme do prioritní fronty, která je řazena podle času zastavení a později i C_z (čas aktuální cesty)

Pro algoritmus bude nutno přiřadit nějaký parametr podle kterého se bude hodnotit čas. Již dříve při definování struktury jsme si popsali že jednotlivé vrcholy budou obsahovat pouze čas výjezdu a čas příjezdu a hrany nebudou nijak ohodnoceny. Avšak C_z se, pokud nepočítáme s penalizací za přestup, dá vypočítat jednoduše, $C_z(x) = |\text{čas výjezdu z počátečního vrcholu} - \text{čas příjezdu}(x)|$. Ovšem na začátku práce jsme si definovali, že pro jednoduchost budeme používat striktně danou hodnotu pro přestup. To pro nás znamená že si budeme muset vést někde separátně počet přestupů. Počtem přestupů se ovšem nemyslí počet použití přestupových hran, musíme si uvědomit, že díky naší struktuře grafu můžeme traverzovat několikrát po přestupové hraně, ovšem stále budeme na stejné zastávce. To znamená, že jako přestup budeme považovat využití přejezdové hrany a hrany přestupní striktně za sebou.

Díky těmto definicím je již samotná úprava Dijkstrova algoritmu relativně jednoduchá. Algoritmus probíhá ve smyčce:

1. Vyber z prioritní fronty vrchol aktuální(X), pokud nějaký existuje. Pokud ne, zastav algoritmus
2. Pokud již byl vrchol X zpracováván dříve v rámci této cesty, ukonči další traverzování touto cestou. Vrať se ke kroku 1
3. Pokud již byl vrchol X zpracováván algoritmem dříve v rámci jiné cesty, kdy čas příjezdu na vrchol původní $<$ čas příjezdu na vrchol (X), ukonči další traverzování touto cestou a vrať se ke kroku 1
4. Pokud je X zastavením v cílové stanici, ulož čas příjezdu a ukonči traverzování. Vrať se ke kroku 1
5. Prozkoumej všechny sousedy, které jsou dostupné pomocí hran. Typy hran, které mohou být využity pro hledání sousedů, jsou určeny první z možností, která pro aktuální vrchol X platí:

- a. Pokud je X prvním vrcholem v rámci cesty, sousedy uvažuj pouze po přejezdové hraně. Tím se vyhneme nekonečnému smyčkování na začátku.
- b. Pokud byla poslední hrana po cestě k X hranou přestupní a současně není vrchol X v platnosti pro aktuální datum vyhledávání, sousedy uvažuj pouze po přestupní hraně. Jak jsem definovali výše.
- c. Pokud neplatí podmínka (a) ani podmínka (b), všech dostupných hranách.

6. 7. Všechny zbylé sousedy v množině sousedů z kroku 5 vlož do prioritní fronty a pokračuj krokem 1

Výstupem našeho algoritmu je kolekce všech cest odpovídajících zadání. To znamená, že pole výsledků si seřadíme a prvních 5 od nejmenšího předáme pomocí API android aplikaci.

Aplikace pro android

Použité technologie

Android

Android je open source mobilní operační systém založený na Linuxovém jádře, který byl vytvořen společností Android Inc. Společnost Android Inc. byla v roce 2005 odkoupena Googlem. Hlavním cílem společnosti bylo vytvořit open source systém, který bude možné spustit na jakémkoliv mobilním zařízení, což se společnosti podařilo, jelikož se dnes jedná o dominující systém na trhu (přibližně 86,8 % všech mobilních zařízení běží právě na Androidu). První verze Androidu se objevila v roce 2007 s označením 2.6. Necelý rok po představení Androidu byl uveden i první mobilní telefon s tímto systémem, který nesl označení T-Mobile G1 (HTC Dream). Dnes se systém nachází ve verzi 10.0.

XML

XML (Extensible Markup Language) je obecný značkovací jazyk, který byl vyvinut sdružením podnikatelů W3C. Umožňuje snadné vytváření konkrétních značkovacích jazyků (tzv. aplikací) pro různé účely a různé typy dat. První zmínky o XML se datují do roku 1996, kdy byl vyvinut z SGML. Dnes se jedná vedle JSONu o jednu nejpoužívanějších datových struktur, které se používají pro výměnu dat mezi aplikacemi. V androidu je XML použito pro tvorbu layoutů a komponentů, podle kterých se následně tvoří vzhled celé aplikace.

SQLite

SQLite je relační databázový systém napsaný v jazyce C, který byl vytvořen D. Richardem Hippem. První verze se objevila kolem roku 2000 ve verzi 1.0, k dnešnímu dni se SQLite nachází ve verzi 3.31.1. Velkou výhodou SQLite oproti jiným databázovým systémům je to, že se jedná o lokální databázový systém, tj. všechna uložená data se ukládají ve formě souborů přímo na zařízení uživatele. Jelikož se jedná pouze o velmi malou knihovnu nástrojů (celá knihovna zabere pouze něco málo přes 700KB) je SQLite zabudované i v některých programovacích jazycích. SQLite je zabudované i přímo do systému android.

OkHttp

OkHttp je open source knihovna, která usnadňuje tvorbu webových klientů. První verze byla vydána ve verzi 1.0.0 v květnu 2013. Dnes knihovna nachází ve verzi 4.6.0 a umožňuje širokou škálu různých typů webových klientů. Knihovna podporuje v dnešní době velmi šikovné WebSokety. Jedná se o jednu s nejvíce využívaných knihoven na tvorbu webových klientů pro Android.

Osmdroid

Jedná se o knihovnu usnadňující správu a konfiguraci při využívání OpenStreet map.

Websocket

Websocket je podobně jako HTTP počítačový komunikační protokol, ale na rozdíl od zmíněného HTTP poskytuje Websocket protokol plně obousměrný komunikační kanál přes jediný TCP

připojení, který aplikacím umožňuje real-time přenos dat. WebSocket byl poprvé standardizován komisí IETF v roce 2011. Websokety se hojně využívají u aplikací požadující odpověď v reálném čase. Mezi nejznámější aplikace využívající WebSocket protokol patří například chatovací aplikace (např. Messenger),

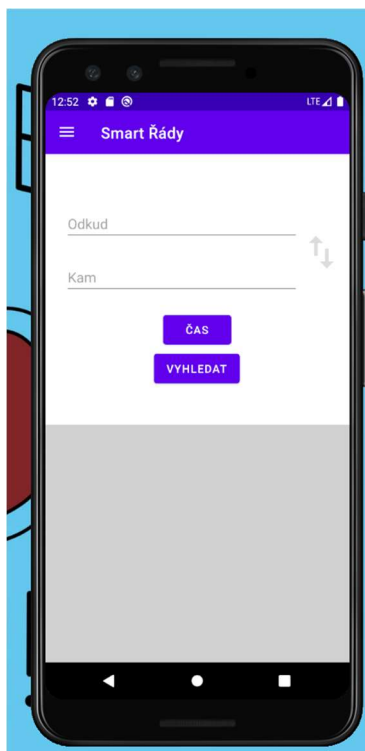
Základní ustanovení pro tvorbu aplikace

Před samotnou tvorbou aplikace je důležité ustanovit několik základních parametrů, podle kterých se budeme při tvorbě aplikace řídit. Hlavním cílem naší aplikace je usnadnit každodenní život pro každého, kdo plánuje využívat veřejnou dopravu. Proto musíme naši aplikaci uzpůsobit tak, aby jí byl schopen používat úplně každý člověk. Hned z prvního pohledu na aplikaci by měl být uživatel schopen poznat, jak aplikaci používat a jaké funkce aplikace poskytuje. Toho můžeme docílit tím, že se budeme snažit aplikaci vytvořit s co možná nejjednodušším vzhledem, tak aby byla aplikace co nejvíce přehledná, rychle responzivní a dobře ovladatelná.

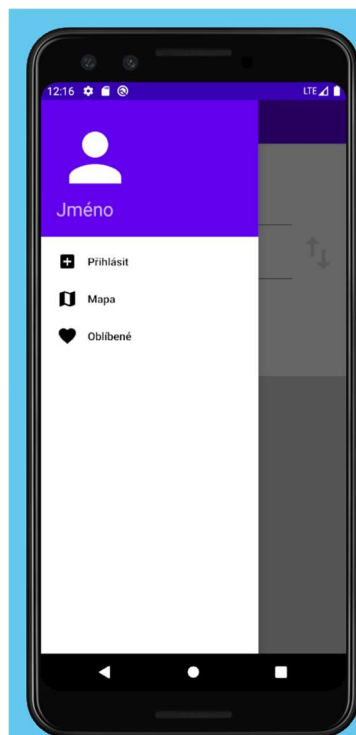
Funkce aplikace

Vyhledávání

Základní myšlenkou aplikace bylo uživateli poskytnout možnost vyhledat požadovaný spoj z bodu a do bodu b. Vyhledávací obrazovka je první věc, se kterou se uživatel po spuštění aplikace setká.



Obrázek14. vstupní layout aplikace

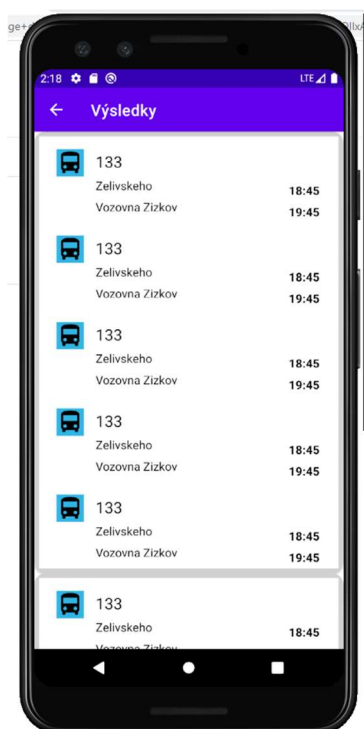


Obrázek 15. Navigační menu

Uživatel zde může zadat odkud a kam vede jeho požadovaná trasa a vybrat kdy chce z počáteční stanice odjet. Vyhledávání disponuje funkcí `AutoCopleteTextVlew`, které zjednodušuje uživateli zadávání požadované trasy a zároveň umožňuje ošetřit podmínku, při které by uživatel zadal neexistující zastávku. `AutoCopleteTextView` je funkce propojená s SQLite databází všech zastávek PID, díky které lze s pomocí `AutoCopleteTextView` adaptéru uživateli napovídat při zadávání zastávky do kolonky odkud, nebo kam, čímž se mu usnadní i samotné zadávání, a zároveň nedáte uživateli možnost zadat do kolonky jinou než existující zastávku. Layout vyhledávání také disponuje komponentou `NavigationView`, která uživateli umožňuje pohodlný a plynulý pohyb po aplikaci.

Nabídka se automaticky odemyká a zamyká funkce aplikace podle toho, jestli je uživatel přihlášen, nebo ne.

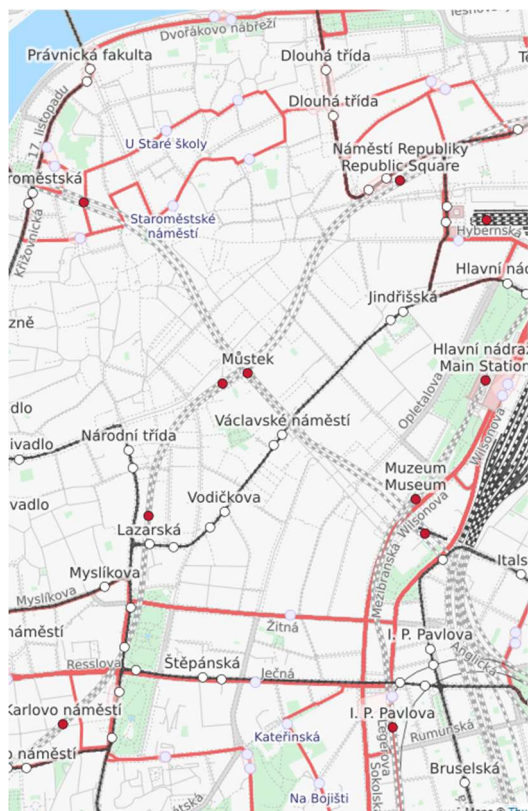
Po zadání času odjezdu, vyplnění kolonek odkud, kam a následném stisknutí tlačítka vyhledat se spustí aktivita, která má za úkol vykreslit požadované výsledky a uživateli zadané hodnoty se serializují do JSON soboru. Tento JSON soubor je následně odeslán na server, ze kterého se jako odpověď odešle další JSON soubor, který obsahuje všechny nalezené a optimální trasy. Tento soubor je dále deserializován a zpracován pomocí dvou RecyclerView adapterů, díky kterým se dynamicky vykreslují požadované odpovědi.



Obrázek 16. Ukázka testovního vykreslení výsledků

Mapa

Pro lepší orientaci ve městě byla pro uživatele do aplikace zaimplementována mapa. Tato funkce využívá volně dostupné knihovny osmdroid a volně dostupných map s API od organizace Thunderforest, které poskytují téměř dokonalou náhradu za Google maps. Thunderforest API bylo pro naši aplikaci zvolené, jelikož obsahuje pro nás perfektní balíček map se jménem Transport, který obsahuje všechny známé světové stanice a zastávky, a také se jedná o open source balík map.



Obrázek 17. Ukázka mapy zobrazující stanice

Aplikace využívá v dnešní době pro mobilní zařízení již standartní funkce GPS, díky které je schopna ve spojení s funkcemi knihovny osmdroid přesně vykreslit do mapy polohu uživatele a tím uživateli napomoci k lepší orientaci a vyhledání zastávek v dané lokalitě.

Celá mapa funguje díky jednoduchému API, které zprostředkovává výstřižky mapy, podle toho, kde se uživatel právě nachází při prohlížení mapy.

Oblíbené trasy

Třída `FavouritesFragment` nastavuje oblíbené trasy. Můžete si vytvořit oblíbené trasy, které často používáte (např. do školy, práce atd.) pro rychlejší vyhledávání spojů.

Komunitní prvky

Proto abychom uživateli umožnili co nejlepší a nejpresnější možný výsledek vyhledávání, jsme se rozhodli do aplikace zaimplementovat určité komunitní prvky. Tyto prvky by umožňovali vytvořit síť uživatelů, kteří by mohli v reálném čase zadávat a získávat informace o aktuálních nehodách, výukách a zpoždění spojů.

Celý princip by pro uživatele fungoval velice jednoduše. Každý uživatel by si mohl v záložce registrace vytvořit uživatelský účet, který by byl následně uložen v externí serverové databázi uživatelů. Dále by se uživatel musel přihlásit a tím odblokovat rozšířené uživatelské možnosti. Tímto přihlášením by také uživatel navázal trvalé spojení se serverem, díky protokolu `WebSocket`. Toto připojení by umožnilo síti uživatelů v reálném čase zadávat výuky nebo nehody v daných zastávkách, nebo získávat v reálném čase notifikace o daných nehodách ve všech stanicích, čímž by se mohl vyhnout zdržení.

Mezi uživateli by fungoval rankovací systém. Každý uživatel, který by zadal jakoukoliv nehodu, by následně mohl být ohodnocen ostatními uživateli podle pravdivosti a přesnosti jeho příspěvku. Uživatel by následně mohl získat svůj rank, čímž by ho následně mohli brát ostatní uživatelé více, nebo méně seriózně. Pokud by uživatel získal opravdu vysoký rank, tak by se uživatel dostal až na pozici „ověřeného uživatele“. Tato skupina ověřených uživatelů by následně svými příspěvky zasahovat do funkčnosti samotné databáze. Pokud by ověřený uživatel zadal svůj záznam o nehodě, tak by tento záznam změnil i určité parametry průjezdnosti v hlavní grafové databázi. Parametr průjezdnosti by následně mohl vypnout daný uzel na krátký čas, čímž by se mohl všichni ostatní uživatelé vyhnout zdržení.

Závěr

Bohužel na závěr se hodí zmínit, že jsme práci zásadně podcenili. Díky neschopnosti týmu se domluvit jsme neefektivně strávili náš čas. Z toho důvodu nás obrovsky tlačil termín a nestihli jsme velkou část věcí co by se bývala stihnout dala, kdyby se všichni naplno zúčastnili práce. Téma je zajímavé a myslím že můžeme objektivně zhodnotit naši rozmyšlenost nad daným tématem jako výbornou. Do designování struktury grafu, návrhu algoritmu a učení se s novými technologiemi byla dána obrovská míra úsilí a energie. Bohužel jsme selhali nad implementací, zabývat se takto těžkým tématem s sebou nese značné problémy, které nás stály drahocenný čas, protože pro velkou část práce jsme si museli vymyslet sami, neexistují návody, jak implementovat tu či onu část a speciálně na neo4j embedded a traversal api existuje velice málo dokumentace. To také vedlo k tomu, že jsme selhali při její implementaci. Na díle je ovšem odvedený velký kus práce a i když jsme ji nedokázali v termínu plně dokončit doufáme že na ní nebude pohlíženo s velkým mínusem. Vzali jsme si velké sousto se kterým jsme se jakž takž popasovali, ale také jsme si odnesli velké množství zkušeností, zjistili že teamové projekty nejsou nic pro nás a že si příště dáme větší časovou rezervu

Bibliografie

1. KALLA, T. Analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití. Master's thesis, Masarykova univerzita, Fakulta informatiky, Česká republika, 2015
2. Spring.io [online]. 2020. Dostupné z: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.core-concepts>
3. Spring.io [online]. 2020. Dostupné z: <https://docs.spring.io>
4. BRODAL, G. S. – JACOB, R. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. *Electronic Notes in Theoretical Computer Science*. 2004, 92, 0, s. 3 – 15. ISSN 1571-0661. doi: <http://dx.doi.org/10.1016/j.entcs.2003.12.019>. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S1571066104000040> . Proceedings of {ATMOS} Workshop 2003.
5. Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, Madhav Marathe Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router ESA 2003, *Lecture Notes in Computer Science*, volume 2461 (2002), pp. 126-138
6. Pid.cz: Pražská integrovaná doprava [online]. 2020. Dostupné z: <https://pid.cz/o-systemu/opensdata/>
7. Frank Schulz, Dorothea Wagner, Karsten Weißen Dijkstra's algorithm on-line: An empirical case study from public railroad transport *Algorithm Engineering, LNCS*, volume 1668 (1999), pp. 110-123
8. Ariel Orda, Raphael Rom: Minimum weight paths in time-dependent networks *Networks: An International Journal*, 21 (1991)
9. PALLOTTINO, S. – SCUTELLÀ, M. G. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects, 1998
10. M. Carey and E. Subrahmanian. An approach for modelling time-varying flows on congested networks. *Transportation Research B*, 34:157–183, 2000.
11. H. S. Mahmassani and S. Peeta. System optimal dynamic assignment for electronic route guidance in a congested traffic network. In *Urban Traffic Networks. Dynamic Flow Modelling and Control*, strany 3–37. Springer, Berlin, 1995.
12. Michael Szul [online]. 2015. Dostupné z: <https://codepunk.io/a-brief-history-of-markup-and-xml/>
13. John Callahan [online]. 2020. Dostupné z <https://www.androidauthority.com/history-android-os-name-789433/>
14. TchořoBot [online]. 2020. Dostupné z [https://cs.wikipedia.org/wiki/Android_\(opera%C4%8Dn%C3%AD_syst%C3%A9m\)](https://cs.wikipedia.org/wiki/Android_(opera%C4%8Dn%C3%AD_syst%C3%A9m))
15. Michal Martinek [online]. 2020. Dostupné z <https://www.itnetwork.cz/sqlite/sqlite-tutorial-uvod-a-priprava-prostredi>
16. *monsieurtanuki* [online]. 2020. Dostupné z <https://github.com/osmdroid/osmdroid>
17. Google, Inc. [online]. 2011. Dostupné z <https://tools.ietf.org/html/rfc6455>

Seznam obrázků:

Obrázek 1. ukázka SQL příkazu.....	8
Obrázek 2. Ukázka třídy s implementovanou JPA.....	9
Obrázek 3. Ukázka implementovaného interface Repository s metodou findByUserName.....	9
Obrázek 4. struktura PID GTFS balíku.....	10
Obrázek 5. struktura stops.txt.....	11
Obrázek 6. struktura stop_times.txt	11
Obrázek 7. struktura calendar.txt	11
Obrázek 8. ukázka struktury finálního grafu – screenshot přímo z databáze.....	15
Obrázek 9. ukázka injektu hodnot do konstruktoru ve třídě Scheduling, odkaz (zeleně) a automaticky vyplněnou hodnotu díky IntelliJ pluginu (šedé pole)	16
Obrázek 10. Ukázka třídy, která je servisní třídou ve springu a obsahuje automaticky importvanou proměnnou typu UserRepository.....	17
Obrázek 11. Ukázka plánování metody.....	17
Obrázek 12. ukázka autentikace, rozhraní, které parsuje hodnoty k jwt kontrole.....	18
Obrázek 13. výňatek z třídy JwtUtils, která pracuje s jwt tokenama	19
Obrázek 15. vstupní layout aplikace Obrázek 16. Navigační menu.....	24
Obrázek 17. Ukázka testového vykreslení výsledků.....	25
Obrázek 18. Ukázka mapy zobrazující stanice.....	26