VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2019 Daniel Brát

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

VYHLEDÁVAČ OPTIMÁLNÍ CESTY MĚSTSKÉ HROMADNÉ DOPRAVY S ARCHITEKTUROU KLIENT-SERVER
CLIENT-SERVER SEARCH ENGINE FOR OPTIMAL PUBLIC TRANSPORT ROUTES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Daniel Brát**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Vojtěch Myška**
SUPERVISOR

BRNO 2019

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Daniel Brát ***ID:*** 186036
Ročník: 3 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Vyhledávač optimální cesty městské hromadné dopravy s architekturou klient-server

POKYNY PRO VYPRACOVÁNÍ:

V rámci této práce bude vytvořen vyhledávač optimální cesty městské hromadné dopravy s architekturou klient-server. Serverová část bude zodpovědná za získávání jízdních řádů příslušných dopravních podniků. Klientská část bude provádět výpočty a zabezpečeně komunikovat se serverovou částí. Cesta bude definována uživatelem aplikace. Každý bod této cesty bude doplněn o odhad doby pobytu v daném místě. Výstupem bude nalezení optimální trasy.

DOPORUČENÁ LITERATURA:

[1] Oracle. Java Platform, Standard Edition 8 API Specification [online]. Dostupné z https://docs.oracle.com/javase/8/docs/api/index.html

[2] BURGET, R. Teoretická informatika. Brno: Vysoké učení technické v Brně, 2012. s. 1-198. ISBN: 978-80-2-4-4897- 1. (cs)

Termín zadání: 1.2.2019 ***Termín odevzdání:*** 27.5.2019

Vedoucí práce: Ing. Vojtěch Myška
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Bakalářská práce se zabývá návrhem programu s architekturou klient-server pro hledání cesty v sítích městské hromadné dopravy. V rámci teoretického úvodu je popsán problém obchodního cestujícího a jsou zmíněny metody jeho řešení. Následuje teoretický návrh samotného programu. Další kapitola se věnuje konkrétně vybraným technologiím pro realizaci práce. Poslední pak už pojednává o samotné realizaci práce. Dosažené výsledky jsou pak hodnoceny v závěru.

KLÍČOVÁ SLOVA

Problém obchodního cestujícího, architektura klient-server, Kotlin, Java, JavaFX, Ktor, REST API, NoSQL, MongoDB

ABSTRACT

This bachelor thesis deals with the design and implementation of a client-server based program for finding the optimal path in public transport networks. Theoretical basics describe the Traveling salesman problem and mention some methods for finding its solution. This chapter is followed by a theoretical design of the application itself. Next chapter describes chosen technologies to be used. The last chapter describes the realization of the application itself. Reached goals are described in the conclusion.

KEYWORDS

Traveling salesman problem, client-server architecture, Kotlin, JavaFX, Ktor, REST API, NoSQL, MongoDB

BRÁT, Daniel. *Vyhledávač optimální cesty městské hromadné dopravy s architekturou klient-server*. Brno, 2019, 49 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Vojtěch Myška

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Vyhledávač optimální cesty městské hromadné dopravy s architekturou klient-server“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Vojtotěchu Myškovi za odborné vedení, konzultace a trpělivost.

Brno ....

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době je díky moderním technologiím možné čím dál více věcí zařídit z pohodlí našeho domova. Ať už mluvíme o on-line nakupování s dodáním přímo domů nebo o komunikaci s úřady pomocí datové schránky [24], můžeme obecně říci, že se jedná o činnosti, kvůli kterým jsme se v minulosti museli nutně fyzicky někam dostavit. Nehledě na stále přibývající alternativy však stále musíme mnohé záležitosti vyřídit osobně a díky hektickému životnímu stylu se mnohým z nás takovýchto povinností často nakupí na jeden den více. Je potom na nás si co nejefektivněji tyto "obchůzky"naplánovat tak, abychom toho nacestovali co nejméně, a hlavně abychom vše stihli vyřídit, pokud možno v jeden den. Toto plánování samotné nám však již zabírá notnou část již tak drahocenného času, který se tím paradoxně snažíme ušetřit, a navíc je velmi pravděpodobné, že pro větší počet destinací námi ručně nalezené řešení nebude to úplně nejlepší. Tato úloha je ještě poněkud zvládnutelná pro jedince, kteří mají k dispozici osobní automobil, což jim dává podstatnou flexibilitu při plánování posloupnosti návštěv jednotlivých destinací, ale začíná být velmi obtížná, a hlavně časově nákladná pro ty z nás, kteří musí spoléhat na využití hromadné dopravy. Již při zdánlivě malém počtu míst, která chceme navštívit, se úkol najít optimální trasu mezi nimi v komplexnější dopravní síti (např. Ostravy nebo Brna), kde musíme vzít v potaz velké množství linek, jejich, dle dne měnící se, jízdní řády a vzájemné návaznosti, již stává pro člověka v podstatě nemožný. A v tom nám právě informační technologie mohou značně pomoci.

Dnes již existuje spousta zdarma online dostupných vyhledávačů spojení, které většinou poskytují samotní dopravci, popřípadě třetí strany, které data od dopravců agregují (např. IDOS). Ty jsou pak dostupné ve formě interaktivní webové stránky a často i jako aplikace pro mobilní zařízení či PC. Jejich vstupem jsou souřadnice či jméno zastávky výchozího bodu a destinace (v některých případech i jednoho průchozího bodu), datum a čas požadovaného odjezdu, načež jejich výstupem je seznam vyhovujících spojení, a to buď přímo, popřípadě s přestupy. To z nich dělá užitečný nástroj, pokud hledáme cestu z bodu A do bodu B. Pokud však chceme najít nejrychlejší trasu přes několik bodů nezávisle na pořadí ve kterém je navštívíme, jsme odkázáni na zkoušení jednotlivých kombinací, jejichž počet neúnosně narůstá již pro malý počet bodů. To je ručně velmi zdlouhavé. Tato práce se snaží nabídnout řešení tohoto problému návrhem aplikace provádějící tyto výpočty.

1 Teoretický rozbor

Než bude podrobněji popsán a vysvětlen problém, který se snaží řešit tato práce, je třeba uvést základní pojmy teorie algoritmů.

1.1 Hodnocení algoritmů

Algoritmus je jednoznačně definovaná posloupnost konečného počtu kroků, která vede k řešení daného problému, přičemž splňuje následující vlastnosti[25]:

1. **Hromadnost a univerzálnost** - algoritmus produkuje řešení pro každou přípustnou kombinaci vstupních dat.
2. **Determinovanost** - v každém bodě výpočtu je jednoznačně určeno, jaký je výsledek aktuálního kroku a jaký je krok následující.
3. **Konečnost** - algoritmus v konečné době skončí.
4. **Rezultativnost** - pro každé vstupní data je vrácena výstupní hodnota.
5. **Korektnost** - výsledek vypočtený algoritmem musí být správný.
6. **Opakovatelnost** - stejná vstupní data produkují vždy stejný výsledek.

To, aby nám k něčemu algoritmus byl, musí být schopen řešit daný problém efektivně. Pro vyjádření efektivity algoritmů existuje tzv. složitost algoritmu, která sleduje, jak se algoritmus chová v závislosti na velikosti vstupních dat. To nám umožňuje kvalitativně porovnat různé algoritmy řešící ten samý problém. Podle sledované metriky definujeme složitost paměťovou, která sleduje velikost algoritmem používaných datových struktur a složitost časovou, odvíjející se od počtu vykonaných operací [23]. Každou z nich pak navíc můžeme určit ve dvou podobách, a to buď jako složitost absolutní nebo asymptotickou.

Absolutní složitost určuje konkrétní hodnotu pro každou danou velikost vstupu a často je uváděna také s jednotkou. Ta může v případě časové složitosti být např. počet procesorem vykonaných instrukcí nebo doba běhu programu v sekundách a v případě složitosti paměťové pak např. algoritmem využívaný počet bytů v paměti. Toto vyjádření se však téměř nepoužívá, protože je možné jej určit pouze u velmi jednoduchých případů a může se při použití konkrétních jednotek lišit podle platformy (např. počet instrukcí pro RISC procesor vs. pro CISC).

Oproti tomu asymptotická složitost charakterizuje chování algoritmu pouze přibližně. Pro její vyjádření existuje dle způsobu aproximace několik různých notací.

Mezi ty často používané patří:

- Notace O - udává horní asymptotický odhad. Jinak řečeno, neexistuje případ, kdy by ve skutečnosti složitost byla vyšší. Není sice sama o sobě vždy o daném algoritmu nejvíce vypovídající (mnohé algoritmy se ve většině případů této hranici vůbec nepřiblíží), ale díky tomu, že pokrývá všechny možné případy, je v praxi notací nejpoužívanější.
- Notace Θ - je průměrný odhad složitosti funkce. Algoritmus se může chovat v závislosti na vstupních datech hůře či lépe, ale střední hodnota jeho složitosti je dána touto notací.
- Notace Ω - vyjadřuje dolní odhad složitosti neboli nejnižší složitost které algoritmus dosahuje.

Pro zjednodušené dělení algoritmů jejich složitost nevyjadřujeme jako konkrétní funkci se všemi členy a konstantami, ale daný algoritmus zařadíme do jedné z několika kategorií podle podoby na některou ze základních složitostních funkcí.

Tab. 1.1: Základní složitostní funkce (seřazeny od nejmenší)

Funkce	**Název**
$f(1)$	konstantní
$f(\log n)$	logaritmická
$f(n)$	lineární
$f(nlogn)$	lineárně logaritmická (kvazilineární)
$f(n^2)$	kvadratická
$f(n^c), c > 1$	polynomická
$f(c^n)$	exponenciální
$f(n!)$	faktoriální

1.2 Dělení problémů

Problémy samotné můžeme obecně rozdělit do několika navzájem provázaných kategorií.

První z nich je problém rozhodovací. To je takový problém, který má na vstupu libovolný počet proměnných z libovolně velkého oboru hodnot, ale jeho výsledkem je hodnota z dvouprvkové množiny $\{Ano, Ne\}$. Rozhodovací problémy jsou tzv. „kompletní“, pokud existuje množina problémů, do které problém sám náleží a zároveň je možné prvky této množiny v polynomiálním čase redukovat na daný problém. Existují však i problémy, které rozhodnutelné nejsou.

Dalším druhem problémů je problém funkce, který již může na svém výstupu mít hodnotu z jakkoliv velké množiny prvků. Je analogický k matematické funkci, která

pro každou kombinaci vstupních parametrů z oboru hodnot produkuje určitý výstup. Všechny problémy funkce jsou převeditelné na rozhodovací problém a zároveň platí, že pokud je výsledný problém rozhodnutelný, je také řešitelný problém funkce, ze které byl odvozen. Tuto konverzi lze provést také opačně, kdy problém rozhodovací převedeme na problém funkce nalezením jeho charakteristické funkce.

Posledním typem jsou pak problémy optimalizační, které již řeší, jaké z několika platných řešení je pro daný vstup to nejlepší. Stejně jako problémy funkce, i tyto lze převést na problém rozhodovací, jehož výstupem je, zdali pro vstup existuje nějaké lepší řešení.

1.3 Třídy složitosti problémů

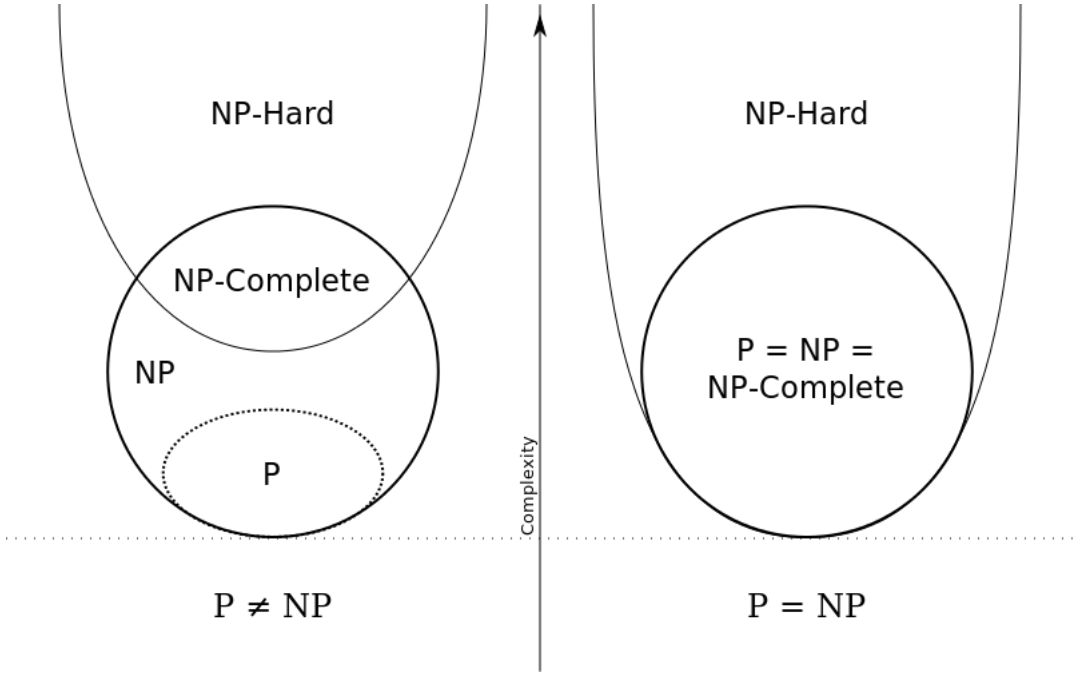
Stejně jako u algoritmů, i problémy samotné můžeme rozdělit do skupin podle existence a komplexity algoritmů, které je dokáží řešit. Vzhledem ke vztahu mezi typy problémů, který je popsán výše, je při kategorizování komplexity předpokládáno, že se jedná o problém rozhodovací.

Existuje několik základních tříd složitosti. První z nich je třída P (z angl. Polynomial time), která obsahuje problémy, ke kterým známe algoritmy schopné je řešit v nejhorším případě v polynomiálním čase a stejně tak jsme schopni efektivně správnost nalezeného řešení ověřit. Patří sem problémy, které můžeme v porovnání s komplexitou jiných tříd označit jako „lehké“. Obecně platí, že pro jejich řešení v polynomiálním čase stačí deterministický Turingův stroj. Problémy, které spadají do této kategorie jsou například nalezení největšího společného dělitele nebo nalezení maximálního párování grafu.[22]

Další třídou složitosti je třída NP (z angl. Nondeterministic Polynomial time), obsahující problémy, ke kterým nejsou známy algoritmy schopné je v praxi vyřešit v polynomiálním čase, ale pokud bychom nějak k řešení došli, jsme schopni v polynomiálním čase ověřit jeho správnost. Teoreticky jsou tyto problémy řešitelné v polynomiálním čase, ale pouze za použití nedeterministického Turingova stroje (viz. níže). Podskupinou těchto problémů jsou pak problémy NP-kompletní. Obecně sem patří ty nejtěžší problémy z NP. Jejich specifickou vlastností je fakt, že pokud bychom někdy našli způsob, jak některý z nich řešit v polynomiálním čase, byli bychom pak schopni řešit každý problém z NP v polynomiálním čase, protože každý problém z NP lze na libovolný NP-kompletních problém efektivně redukovat. Podobné vlastnosti má i kategorie problémů NP-těžkých s výjimkou toho, že nemusí nutně do NP patřit, tudíž nemusí ani existovat způsob, jak v polynomiálním čase ověřit jejich řešení. Navíc sem můžou patřit i problémy nerozhodnutelné.

Přesná hierarchie a vztahy mezi jednotlivými třídami složitosti, konkrétněji platnost výroku $P = NP$, je i nadále nevyřešenou otázkou teorie výpočetní složitosti.

Pokud by se někomu podařilo jednoznačně prokázat platnost či i neplatnost tohoto tvrzení, krom odměny 1 miliónu dolarů [20] by jej taky čekalo zapsání do historie, neboť vyřešení tohoto dilema by mělo výrazný význam pro spoustu vědních odvětví. Dnes převládá spíše názor, že tyto dvě skupiny rovny nejsou, a proto ještě větší dopad by měl právě důkaz o opaku. To by znamenalo, že by bylo možné těžké problémy v polynomiálním čase převést na problémy efektivně řešitelné. Lze si představit, že tato skutečnost by umožnila nevídaný posun v mnoha odvětvích, ale neméně výrazně by negativně ovlivnila například dnes používané kryptografické funkce, jejichž výstup by najednou bylo možné efektivně převést zpět do vstupní podoby.



Obr. 1.1: Eulerův graf pro kategorie problémů pro případ rovnosti a nerovnosti P a NP By Behnam Esfahbod, CC BY-SA 3.0, Dostupné zde

1.4 Turingův stroj

Turingův stroj je abstraktní matematický model popisující hypotetický stroj pro provádění algoritmů popsaný v roce 1936 britským matematikem Alanem Turingem. Skládá se z nekonečně dlouhé pásky rozdělené na diskrétní „buňky“ (políčka), které mohou být prázdné nebo obsahovat nějaký symbol z konečné množiny, která je označována jako tzv. pásková abeceda. Tato páska je pamětí stroje. Nad páskou se pak pohybuje hlava, která může symbol v aktuálním políčku číst i zapsat a následně se posunout doprava nebo doleva, popřípadě ukončit běh stroje. Akce, kterou provede pak závisí na hodnotě právě čteného symbolu a aktuálním vnitřním

stavu. Toto chování je definováno přechodovou funkcí, která se dá označit za analog počítačového programu.

Typů Turingova stroje existuje více, které se mohou lišit počtem hlav, počtem pásek či vlastnostmi pásky. Za zmínku stojí i universální Turingův stroj, který je schopen jako vstup přijmout kód a data jiného Turingova stroje a ty vykonat. Tím je tak principiálně hodně podobný počítačům, které dnes známe. Zajímavé jsou pak také Turingovy stroje paralelní a kvantové. Nicméně hlavní dva obecné typy Turingova stroje, které je zde nutno uvést kvůli jejich vazbě na třídy složitosti, jsou deterministický a nedeterministický.[19][18]

1.4.1 Deterministický Turingův stroj

U deterministického Turingova stroje je při znalosti aktuálně hlavou čteného symbolu a vnitřního stavu možné přesně určit, jak se dále zachová, resp. jaká bude jeho další akce. Matematicky jej můžeme popsat jako

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_F) \tag{1.1}$$

kde Q je konečná množina vnitřních stavů, Σ je konečná množina vstupních symbolů mimo prázdného symbolu, Γ je konečná množina všech symbolů pásky včetně toho prázdného, $q_0 \in Q$ je počáteční stav, $q_F \in Q$ je stav koncový a δ je parciální funkce nazývaná jako přechodová funkce definována jako

$$\delta : (Q\{q_F\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R\}) \tag{1.2}$$

kde $\{L, R\} \notin \Gamma$ a L je symbol pro přesun hlavy doleva a R pro posun doprava.[25]

1.4.2 Nedeterministický Turingův stroj

U nedeterministického Turingova stroje již podle znalosti aktuálního symbolu a stavu nelze určit, jak se stroj zachová. Obecně v tuto chvíli může vykonat jakoukoliv akci z dané kolekce specifikovaných akcí. Matematicky jej můžeme popsat stejně jako ten deterministický, s jediným rozdílem v přechodové funkci, která je

$$\delta : (Q\{q_F\}) \times \Gamma \to Q \times 2^{(\Gamma \cup \{L,R\})} \tag{1.3}$$

Nedá se obecně říci, že by tento typ Turingova stroje byl nikterak lepší, než je jeho deterministický protějšek, co se týče schopnosti řešit jednotlivé problémy, ale zato dokáže řešit problémy z kategorie NP rychleji, a to v polynomiálním čase. [17] Proto také by sestavení takovéhoto typy výpočetního zařízení v praxi bylo stejně významné, jako je dokázání $P = NP$.

1.4.3 Problém zastavení Turingova stroje

Problém zastavení Turingova stroje řeší to, zdali, při znalosti fungování náhodného Turingova stroje, jeho abecedy a jeho vstupu, jsme schopní přesně určit, jestli bude cyklit do nekonečna nebo skončí. Už sám Alan Turing dokázal, že pro obecný Turingův stroj nelze toto určit a tento problém je proto nerozhodnutelný.[16] [15] Naopak „Church-Turingova teze nám říká, že každý algoritmus může být vykonáván Turingovým strojem.“ [25] Pokud tedy opravdu je problém zastavení nerozhodnutelný a Church-Turingova teze pravdivá, můžeme z toho v závěru vyvodit, že existují problémy algoritmicky neřešitelné. Reálné programovací jazyky mohou být tzv. Turingovsky kompletní, což znamená, že lze pomocí nich popsat veškeré algoritmy řešitelné Turingovským strojem.

1.5 Problém obchodního cestujícího

V neposlední řadě je nutno popsat pro tuto práci nejrelevantnější problém obchodního cestujícího (angl. Travelling Salesman Problem – TSP). Jedná se o optimalizační problém patřící do NP-těžkých, nicméně pokud je formulován jako rozhodovací, pak náleží do třídy problémů NP-kompletních. Jsou však známy některé variace se specifickými podmínkami, které jsou řešitelné v polynomiálním čase. Definovat ho můžeme jako: „Známe-li seznam měst a vzdáleností mezi nimi, jaká je nejkratší možná cesta abychom navštívili každé z nich přesně jednou a vrátili se zpět do výchozího bodu?“ [12]

Pokud si představíme města jako vrcholy grafu a cesty mezi nimi jako hrany, jejichž hodnota je vzdálenost mezi městy, pak řešení, které se snažíme najít, je tzv. Hamiltonovský okruh. Existuje spousta variací a podskupin tohoto problému, které najdou své uplatnění nejen v logistice ale třeba i při návrhu integrovaných obvodů. [14]

Pro základní formu TSP existuje spoustu způsobu hledání řešení. Pro stručný přehled uvedu několik z nich.

1.5.1 Exaktní algoritmy

Nejjednodušším způsobem, jak dojít k exaktnímu řešení, je hledání hrubou silou (angl. brute-force search, někdy taky nazýváno jako „naivní“). Výpočet probíhá tak, že jsou vyzkoušeny všechny možné permutace cest a jejich celkové délky jsou porovnány. Z nich je pak jednoduše vybrána ta s tou nejkratší. Tento způsob je sice relativně lehký na implementaci, ale díky nutnosti projít všechny možné kombinace je časově neefektivní. Pro n navzájem symetricky propojených měst existuje $\frac{(n-1)!}{2}$

kombinací a tudíž můžeme časovou náročnost tohoto algoritmu vyjádřit jako $\mathcal{O}(n!)$, což je ještě přijatelné pro malý počet měst, ale již pro několik desítek se stává nepoužitelný.

Další možností, jejíž výsledkem je řešení exaktní, je použití dynamického programování. Jedním z algoritmů ho využívajících je Held-Karpův algoritmus. Dynamické programování obecně funguje na principu rozdělení problému jako celku na menší sub-problémy, pro které je snadnější najít optimální řešení. V praxi je pro tyto algoritmy typická iterace nebo rekurze. V případě Held-Karpova algoritmu a n měst, existuje přesně na $n \cdot 2^n$ sub-problémů a každý z nich jsme schopni vyřešit v lineárním čase. Časová náročnost tohoto algoritmu je pak $\mathcal{O}(2^n \cdot n^2)$, ale na úkor vyšší paměťové náročnosti dané počtem sub-problémů. Časová komplexita tohoto algoritmu, v porovnání s hledáním hrubou silou, se může pro malý počet měst zdát horší či porovnatelná, nicméně již pro počet měst kolem deseti, můžeme pozorovat jeho výrazně menší náročnost.

1.5.2 Heuristické algoritmy

Ne vždy je třeba znát přesné řešení problému a vystačíme si s dostatečně kvalitním řešením sub-optimálním. K tomuto se právě využívají algoritmy heuristické, jejichž výhodou oproti těm exaktním je podstatně menší časová náročnost. I pro TSP existuje spoustu takovýchto algoritmů. Jedním z nich je například metoda nejbližšího souseda, která se snaží najít decentní řešení v co nejkratším možném čase. Funguje tak, že z výchozího bodu najde všechna sousední města a vybere to, s nejnižší vzdáleností. Z toho potom najde všechny jeho sousedy, krom již navštíveného výchozího a opět vybere to s tou nejmenší vzdáleností. Tento postup opakuje, dokud neprojde všechny města právě jednou a pak se vrátí zpět do výchozího bodu. Jeho časová náročnost je pak $\mathcal{O}(n^2)$. Nevýhodou je, že se jedná o tzv. „hladový“ algoritmus, protože bere v potaz pouze lokální minima. O výsledné přesnosti řešení pak nejsme schopni nic říct, protože se dle vybraného výchozího bodu a konkrétní kolekce měst, může výrazně lišit. Obecně se však výsledné řešení od toho optimálního liší průměrně do 25% a tak je tento algoritmus, díky své rychlosti, velmi užitečným nástrojem na získání dobrého odhadu.

Dalším zástupcem heuristických algoritmů je Christofidova metoda. Její specifickou vlastností oproti metodě předchozí je to, že předem víme, že nalezené řešení je od toho exaktního maximálně 1,5krát delší. Tuto metodu však lze aplikovat pouze na grafy, kde jsou vzdálenosti mezi městy symetrické a splňují trojúhelníkovou nerovnost. Další nevýhodou je i fakt, že se jedná o postup složený z několika dílčích kroků, a tak jeho přesná efektivita záleží na kombinaci vybraných algoritmů, řešících dílčí kroky. Obecně však můžeme říct, že její časová náročnost je $\mathcal{O}(n^4)$. Detailnější popis

jednotlivých kroků je nad rámec této práce, ale obecný postup se skládá z určení minimální kostry grafu, nalezení vrcholů lichého stupně, nalezení jejich perfektního párování s nejmenší vzdáleností, vytvoření Eulerovského tahu a v posledním kroku jeho převedení na Hamiltonovský cyklus.

1.6 Analýza zadání

Problém, který má tato práce za úkol řešit, je variantou právě problému obchodního cestujícího, přesněji asymetrický časově závislý problém obchodního cestujícího, neboť ceny jednotlivých hran jsou dány dobou čekání na zastávce, spolu s dobou jízdy do destinace, a ty se mění nejen podle směru, ale jsou proměnné i v čase, v závislosti na již procestované době. To značně omezuje množství použitelných algoritmů, protože je třeba před samotným výpočtem zjistit existence a ceny jednotlivých cest mezi zadanými body, a to pro každou možnou posloupnost. Proto bylo pro potřeby této práce, navzdory své nízké efektivitě, vybráno řešení pomocí hledání hrubou silou, které umožňuje dynamické zjišťování ceny hran spojit přímo se samotným hledáním řešení. Pokud se při hledání spojů mezi dvojicemi bodů omezíme pouze na nalezení jediné hrany, můžeme očekávat, že při dostatečně efektivní implementaci bude časová náročnost výpočtu $\mathcal{O}(n!)$. Je ale více než pravděpodobné, že pro reálná data nebude mezi každou dvojicí míst nalezen spoj vyhovující parametrům, a tak nejspíš bude průměrná komplexita o něco nižší.

2 Návrh řešení

Zadáním této práce je aplikace s architekturou klient-server. Role serveru je v tomto případě agregace dat, konkrétně periodické získávání jízdních řádů z určitého seznamu poskytovatelů, jejich úprava do společné podoby a následné jejich poskytování klientům. Aby mohl server efektivně data dál poskytovat, je nutné je nejdřív uložit do vhodné databáze.

2.1 Databáze

Dnešní databázové systémy můžeme obecně rozdělit na relační a NoSQL databáze. Mezi ty relační patří všechny databáze ukládající data do tabulkové struktury s pevně danými sloupci. Navzdory svému stáří jsou stále hojně využívané. Patří sem obecně všechny databáze využívající jazyk SQL (Structured Query Language). Formát uložených dat musí být dát pevně předem a musí být dodržován, což je nevýhodou u aplikací, kde se datový model často mění. Co ztrácí na flexibilitě však dohání v robustnosti. Většina moderních SQL databází garantuje dodržení tzv. ACID (Atomicity, Consistency, Isolation, Durability) principů.

Ne vždy lze data jednoduše upravit do podoby vhodné pro relační databázi. To je jedním z motivátorů pro použití některého typu z NoSQL(Non SQL nebo Not Only SQL) databází. Jejich cílem není ty relační nahradit, ale spíše se snaží k nim poskytnout alternativu. Podle jimi využívaného datového modelu je můžeme dále rozdělit na:

- úložiště klíč-hodnota - řetězce odkazují na uložená data, které mohou být jakýkoliv datový typ nebo i objekt
- sloupcově orientované databáze - podobné těm relačním, ale řádky jsou kolekce odkazů na jednotlivé sloupce
- úložiště dokumentů - data uloženy jako dokumenty (soubor párů klíč-hodnota), společný mají jen klíč pro jejich adresaci, ale jinak může být každý dokument v kolekci jiný
- grafové databáze - data jsou vrcholy grafu a souvislosti mezi nimi jsou vyjádřeny pomocí hran
- objektově orientované databáze - navazuje na objektově orientované programování tím, že data jsou objekty a jejich struktura je dána třídami

Oproti těm relačním nutně nemusí garantovat dodržení ACID principů, ale mají BASE (Basically Available, Soft state, Eventual consistency) vlastnosti. To jim umožňuje být (v případech pro které jsou určené) rychlejší a efektivnější než databáze relační. Další jejich výhodou je možnost lehkého škálování a existence řešení umožňující použití distribuované architektury.[11]

V našem případě se dá očekávat, že server bude pracovat s větším objemem dat a víme, že data budou měněna s podstatně menší četností než budou čtena. Také je možné, že každý zdroj jízdních řádů bude poskytovat rozdílné podrobnosti, které můžeme do databáze uložit a proto se jako adekvátní hodí vybrat NoSQL databázi.

2.2 Komunikace

Při návrhů distribuovaných řešení, jako je právě třeba architektura klient-server, můžeme použít některý z již zažitých architekturálních stylů. Aplikovatelné pro náš případ webového rozhraní (API - Application Programming Interface) existují dva hlavní styly, a to REST a RPC.

REST (REpresentational State Transfer) navrhl v roce 2000 ve své diplomové práci Roy Fielding a jedná se o soubor doporučení a omezení pro návrh webového rozhraní. Tento styl je datově založený a jednotlivé unikátně adresovatelné koncové body, označovány jako "zdroje"(resources), je reprezentují. Implementace dodržující tyto principy pak můžeme označit jako tzv. "RESTful"protokoly. Přestože se jedná pouze o abstraktní požadavky, které protokol musí splnit, jde dnes ve většině případů pojem "RESTful API"ruku v ruce s volbou HTTP jako komunikačního protokolu. Ten totiž splňuje hlavních pět požadavků, kterými jsou:

1. **Architektura klient-server** - musí být jasně rozděleny role serveru a klienta
2. **Bezstavovost** - požadavky klienta obsahují veškeré potřebné informace pro jeho vykonání
3. **Cachovatelnost** - odpovědi na požadavky na statická data můžou být uchovávány v mezipaměti
4. **Vrstvená architektura** - mezi klientem a serverem může existovat několik, pro klienta transparentních, mezi-bodů (např. proxy servery, vyvažovač zátěže atd.)
5. **Jednotný interface** - existuje pevně definovaný set operací, které lze na jednotlivé zdroje aplikovat

Konkrétní operace vykonatelné na zdrojích definovány protokolem HTTP jsou:

- **GET** - čtení dat, vrací jednu položku nebo i jejich seznam
- **POST** - zápis, vytvoří novou položku na serveru
- **PUT** - přepis, přepíše celou adresovanou položku
- **PATCH** - úprava, změní konkrétní aspekt dané položky
- **DELETE** - smazání, smaže položku ze serveru
- **HEAD** - podobné GET, ale nevrací data, pouze záhlaví

Druhým přístupem k návrhu API může být pak architekturální styl RPC (Remote Procedure Call). Jeho hlavním rozdílem oproti RESTu je jeho procedurální zaměření. Dotazy na server adresují jednotlivé procedury, které jsou serverem vykonány a je

vrácen jejich výsledek. Mnohdy jsou dotazy posílány na server klientem využívanou knihovnou transparentně, takže nemusí být schopen ani rozlišit, jestli se jedná o lokální proceduru nebo vzdálenou. Existuje několik protokolů využívající tento styl architektury. Mezi často používané pro API jako alternativa k RESTu jsou hlavně JSON-RPC, XML-RPC a SOAP.

První z nich, JSON-RPC, využívá pro komunikaci formát JSON (JavaScript Object Notatation) a definuje podobu požadavků na vykonání procedury, odpovědí serveru o výsledcích i případných chybových hlášení. Jedná se o relativně moderní protokol a je koncipován jako jednoduchý a nenáročný na šířku pásma. To na druhou stranu znamená, že nedefinuje žádné pokročilé funkce. Definuje pouze formát přenášených dat, takže není nijak vázaný na konkrétní transportní protokol a lze jej použít například přímo přes TCP/IP sokety nebo i třeba HTTP či WebSocket.

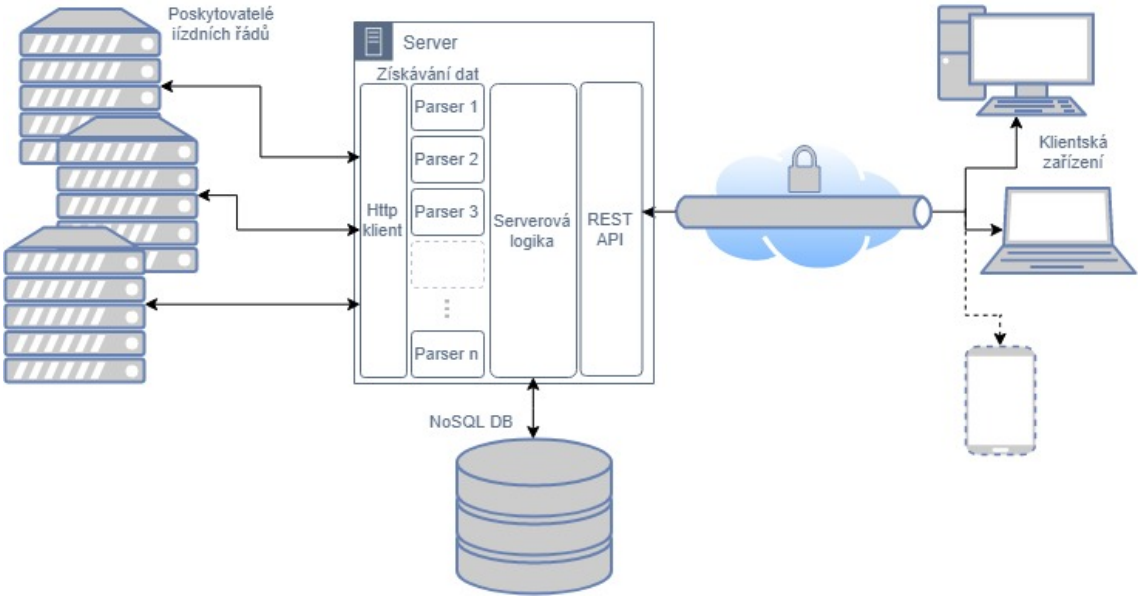
Hodně podobný je i XML-RPC, který, jak už jméno napovídá, používá pro výměnu zpráv formát XML (eXtensible Markup Language). Oproti JSON-RPC je pak přesně vázaný na protokol HTTP. Zprávy reprezentované pomocí XML jsou sice více čitelnější pro člověka, ale na druhou stranu obsahují spoustu nadbytečných dat, které je nutné přenášet a tak má tento protokol větší nároky na šířku pásma. V této své původní podobě se již skoro nepoužívá, ale přidáváním různých rozšíření a vylepšení se časem stal základem pro protokol SOAP.

SOAP (Simple Object Access Protocol) je hojně používaným protokolem zejména v interních firemních systémech. Jednotlivé zprávy jsou ve formátu XML a obsahují oproti XML-RPC navíc hlavičku s různými metadaty. SOAP pro samotnou komunikaci spoléhá na využití již stávajících protokolů, jako např. HTTP, SMTP, JMS (Java Messaging Services) nebo přímo TCP či UDP. Specifikace přihlíží i na rozšiřitelnost a dnes již existuje větší počet rozšíření, označovaných s předponou WS-*, které přidávají funkce jako např. zabezpečení, digitální podpisy nebo lepší přenos binárních dat. Podpůrným standardem, se SOAP spjatým, je WSDL (Web Services Description Language), který v XML poskytuje popis rozhraní webové služby, včetně výčtu volatelných procedur, jejich vstupních parametrů i dat, které vrací. To je výhodné hlavně při vývoji aplikace konzumující API třetích stran, kdy není třeba ručně napsat knihovnu dle poskytovatelem publikované dokumentace, ale můžeme si nechat programem vygenerovat podle WSDL kostru funkcí s již správnými signaturami. SOAP je standardizovaný a v praxi odzkoušený robustní (používaný i v bankovnictví) protokol, nicméně toho dosahuje na úkor větší komplexity a většího nároku na objem přenášených dat.[10][9]

S přihlédnutím na zadání práce, kdy rolí serveru je dále poskytovat agregovaná data, se jeví jako nejvhodnější použít k návrhu rozhraní REST API. Pro splnění požadavku na zabezpečenou komunikaci je pak nutné vybrat HTTPS protokol.

2.3 Shrnutí

Podle zadání je navrhovaný program rozdělen na serverovou a klientskou část. Serverová část se stará o stahování jízdních řádů potenciálně z více zdrojů, jejich parsování do společné podoby a uložení do databáze, ve které pak na žádost klienta provádí hledání dat vyhovujících jeho požadavkům. Pro ukládání dat byla vybrána NoSQL databáze a jako webové rozhraní REST používající HTTPS pro zabezpečenou komunikaci s klienty.



Obr. 2.1: Architektura navrhovaného řešení, vytvořeno pomocí draw.io

3 Použité technologie

Předchozí kapitola se věnovala obecnému výběru architektury a typů komponent, tato se již věnuje popisu konkrétních použitých technologií a implementací.

3.1 Programovací jazyk

Jako hlavní jazyk pro realizaci této práce byl vybrán moderní programovací jazyk Kotlin. Jedná se o multiplatformní jazyk se statickým typováním, který není nikterak dogmatický a umožňuje použití objektově orientovaného tak i funkcionálního paradigmatu, případně kombinaci obou. Byl vytvořen společností JetBrains, která jé známá svými integrovanými vývojovými prostředími (IDE) jako třeba IntelliJ Idea pro Javu a jí podobné jazyky[1], CLion pro C a C++ nebo PyCharm pro jazyk Python. Kotlin je roku 2012 plně open-source a pro záštitu jeho dalšího vývoje byla vytvořeno Kotlin nadace, podporovaná JetBrains a Googlem. Ten taky od roku 2017 plně podporuje Kotlin pro vývoj aplikací pro jeho mobilní operační systém Android. To doposud bylo možné pouze v Javě[2]. Toho bylo možno dosáhnout díky tomu, že přestože Kotlin není s Javou syntakticky kompatibilní, jedním z možných výstupu kompilace je Java JVM kompatibilní bytecode. Kotlin byl dlouho jazykem pouze pro JVM, ale ve verzi 1.1 byla přidána možnost kompilovat do JavaScriptu (Kotlin/JS) a v aktuální verzi 1.3 je beta verze schopná kompilace do nativního kódu (Kotlin/-native). Ve zkušební verzi je zatím i podpora multiplatformních knihoven. Kotlin se nesnaží za každou cenu přinést něco nového a jeho snahou je spíše poučit se z nedostatků Javy a přidat užitečné vlastnosti, které se osvědčili u jiných moderních jazyků. Kotlin je plně kompatibilní s knihovnami napsanými v Javě a stejně tak Kotlinovské knihovny lze využít v programech napsaných v Javě. Podobně tomu je i v případě interoperability s JavaScriptem a C/C++/Swift pro Kotlin/native. Vývojářům zvyklých na jazyk Java může Kotlin nabídnout spoustu užitečných funkcí. Kotlin začalo používat již více větších firem, například český Seznam[4]

První a pro mnohé nejzásadnější výhodou Kotlinu je způsob, jakým pracuje s hodnotou *null*. Mnozí programátoři jistě znají z Javy nechvalně známou výjimku *NullPointerException* (NPE), kterou JVM vyvolá v případě, kdy se kód pokusí přistoupit k polím nebo zavolat nějakou funkci proměnné, která místo předpokládaného objektu obsahuje *null*. Mitigací tohoto problému v Javě jsou neustálé podmínky, které zjišťují, zdali proměnná *null* neobsahuje (tzv. *null checks*). Kotlin tento problém řeší hned na několika frontách. První z nich je rozdělení typů na ty, které

[1] včetně Kotlinu

[2] Pokud nepočítáme frameworky třetích stran, jako např. Xamarin umožňující použití C# nebo React native používající Javascript

mohou hodnotu *null* nabývat (nullable) a ty, které ne (non-nullable). Ty první z nich poznáme podle k nim přidaného otazníku (např. String?). To, zda se náhodou nepokoušíme propašovat *null* tam, kde nepatří, kontroluje při překladu automaticky kompilátor a většina vývojových prostředí na to upozorní již při psaní kódu. Když už však potřebujeme s proměnnou nabývající hodnoty *null* nějak pracovat (často při využití knihoven napsaných v Javě), můžeme využít několik dalších ulehčení. Při nutnosti použít proměnnou vícekrát můžeme využít klasicky podmínky testující jestli není rovna null. V tom případě je hodnota uvnitř těla podmínky automaticky přetypována na non-nullable. Pokud však chceme například pouze zavolat funkci objektu, nemusíme psát podmínku, ale stačí použít tzv. "safe call operátor"(?.), který funkci zavolá jen pokud není proměnná *null*. Ten lze pak výhodně kombinovat spolu s tzv. "Elvis operátorem"(?:), který funguje oproti Javě a jiným jazykům trochu jinak a umožňuje případný *null* nahradit výchozí hodnotou. Pro případy, kdy je parametr definován jako nullable a nevadí nám případné vyvolání NPE, můžeme použít "null assertion"operátor (!!).

```
1  // funkce s nullable parametrem typu String
2  fun udelejNeco(parametr: String?){
3      // Pokud není parametr null, vrátí jeho délku,
4      // jinak vrací -1
5      val promena = parametr?.length?:-1
6      // Pokud chceme provést více operací s parametrem,
7      // můžeme klasicky otestovat jeho hodnotu
8      // To můžeme také zkombinovat pomocí logických
9      // operací i s jinými výrazy, které budou vyčísleny
10     // pouze v případě platnosti výrazu prvního
11     if(parametr != null && parametr.length > 0){
12         // Uvnitř této větve je parametr automaticky
13         // přetypovaný na non-nullable String
14         // a můžeme s ním již normálně pracovat
15         jinaFunkce(parametr)
16         dalsiFunkce(parametr)
17     }
18     // Pokud by byl parametr null, vyvolá NPE
19     val promena2 = parametr!!.length
20 }
```

Výpis 3.1: Ukázka práce s nullable proměnnými

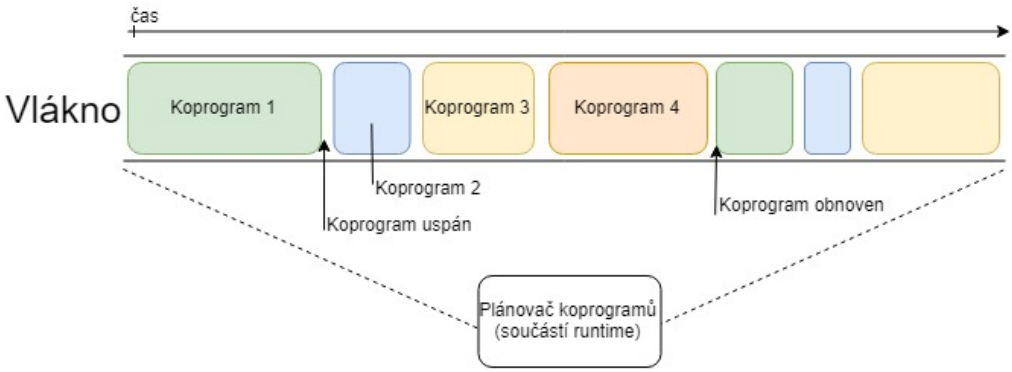
Další z užitečných funkcí jazyka je typová inference, kdy kompilátor ve většině situací dokáže při překladu určit typ proměnné z kontextu, v jakém je definována,

takže není třeba její typ explicitně uvádět. Java sice něco podobného přinesla ve své verzi 10, nicméně v jejím případě je inference limitována pouze na lokální proměnné. Kotlin také rozděluje proměnné podle klíčových slov *val* a *var* na neměnné a proměnlivé. Prvním z nich lze přiřadit hodnotu pouze jednou a jsou tak podobné použití modifikátoru *final* jazyka Java. Mnohé další zjednodušení se týkají definice tříd. Kotlin například umožňuje definovat tzv. primární konstruktor hned za jménem třídy a je v něm možné přímo specifikovat její atributy. To spolu s konceptem datových tříd, které mají kompilátorem automaticky generované metody *equals()*, *hashCode()*, *toString()* a *copy()* takže lze s nimi jednoduše definovat třídy, jejichž primárním úkolem je seskupení dat. Třídu, kterou bychom v Javě definovali na více než padesáti řádcích, jsme schopni v Kotlinu, díky datovým třídám, zapsat jedním řádkem.

```
1 data class Person(var name: String, var surname: String, var id: String)
```

Výpis 3.2: Ukázka použití datových tříd

Poslední důležitou vlastností, kterou je třeba zmínit, je nativní podpora koprogramů. Ty jsou odlehčenou alternativou ke konceptu vláken a umožňují psaní asynchronního neblokujícího kódu, který dokáže na více-jádrových procesorech běžet i paralelně. Jedná se o způsob kooperativního multitaskingu, kdy Kotlin runtime řídí, kdy bude který koprogram uspán a bude vykonáván jiný. Interně plánovač koprogramů využívá vlákno nebo fond vláken, na kterých jednotlivé koprogramy vykonává. Výhodné jsou koprogramy oproti přímému použití vláken zvláště v případech, kdy náš kód většinu svého času stráví čekáním na dokončení nějaké I/O operace. V případě koprogamů se v tomto bodě uspí a na daném vlákně může být vykonáván jiný koprogram. Koprogramy mají taky v porovnání s vlákny mnohonásobně menší otisk v paměti a je jich proto možné definovat větší počet.



Obr. 3.1: Ilustrace funkce koprogramů v Kotlinu pro případ jednoho vlákna, vytvořeno pomocí draw.io

3.2 Server

Technologie použité pro realizaci serverové části.

3.2.1 Framework

Pro snadnou tvorbu aplikací běžících na serveru (tzv. backend) existuje spoustu knihoven a frameworků. Přestože je jazyk Kotlin relativně mladý, i pro něj existuje již několik možností, a to jak řešení napsaných v Javě s přidanou podporou pro snadnější použití v Kotlinu (Spring, Vert.x), tak i těch nových, pro Kotlin přímo napsaných (http4k, kara). Ty z nich, které jsou již dobře zažité z Javy, jsou sice většinou sofistikovanější a mají více pokročilejších vlastností, ale některé z méně známých, napsaných od nuly, s co největším využitím silných stránek Kotlinu, jsou často ve výsledku výkonnější. Proto jsem se při výběru vhodných kandidátů omezil na ty napsané čistě v Kotlinu. Nakonec byl vybrán framework s názvem Ktor. Jedná se o open-source framework oficiálně podporovaný tvůrci jazyka. Je asynchronní a snaží se o co největší využití koprogramů, díky čemuž je výkonem srovnatelný spolu s mnohem vyspělejšími alternativami.[5] Framework ve své základní verzi poskytuje pouze ty nejzákladnější funkce a všechny ostatní je třeba přidat jejich deklarací při inicializaci. To umožňuje omezit výsledný server jen na to co zrovna potřebujeme. Výslednou serverovou část pak můžeme zkompilovat jako stand-alone JAR, který využívá některý z podporovaných webových enginů (Netty nebo Jetty), nebo může být výstupem soubor WAR (Web application ARchive), který můžeme spustit na některém z Java aplikačních serverů, podporujících tento formát (Jetty, Tomcat, Google App Engine). Součástí frameworku je také síťový klient. Toho lze použít jak v serverové části tak i v té klientské. Klienta lze, stejně jako server, rozšířit o dodatečné funkce, kdy k většině serverových rozšíření existuje jejich ekvivalent pro část klientskou. Nové verze frameworku se snaží o zavedení multiplatformní podpory klienta pro použití v prohlížečích a mobilních zařízeních (podpora pro Android již existuje).[7]

3.2.2 Databáze

V návrhu řešení bylo zmíněno odůvodnění volby databáze z rodiny NoSQL. Jako konkrétní implementace byla zvolena databáze MongoDB. Jedná se multiplatformní NoSQL databázi typu úložiště dokumentů (viz. Databáze). Data ukládá do formátu BSON (Binary JSON), který vycházející z formátu JSON. Jedná se o flexibilní řešení podporující snadnou škálovatelnost přidáním více serverů, které spolu dokáží spolupracovat. Podporuje pokročilé způsoby indexování záznamů, pro rychlejší vyhledávání dle různých kritérií. I samotné dotazy umožňují pokročilé funkce, jako třeba

vyhledávání podle regulárních výrazů, podle geografické lokace nebo lze i přímo v rámci dotazu zahrnout uživatelský JavaScript, který server v kontextu dotazu vykoná. Jednotlivé dotazy lze i zřetězit v rámci agregace.

Pro komunikaci s databází jsou dostupné synchronní i asynchronní ovladače pro širokou škálu programovacích jazyků. V případě této práce by šlo použít ovladač pro jazyk Java, nicméně byla nakonec vybrána knihovna KMongo napsaná v Kotlinu, která jej interně využívá a přidává podporu pro snadné mapování mezi dokumenty a objekty a nabízí i způsob, jak snadno sestavit typově bezpečné dotazy.

```
1 data class Person(val name: String, val age: Int)
2 collection.insertMany(Person("Adam", 16), Person("Vojta", 20))
3 val adam = collection.findOne(Person::name eq "Adam")
4 val dospeli = collection.find(Person::age gt 18)
```

Výpis 3.3: Ukázka použití knihovny Kmongo

3.2.3 Další knihovny

V rámci serverové části byly použity také jiné knihovny. Největší z nich je knihovna Krangl, která pomáhá při zpracování stažených jízdních řádů. Zjednodušuje parsování souborů CSV a jiných formátů tabulových dat, uložených v textových dokumentech. Nad nimi lze pak vykonávat operace podobně jako v databázi.

Další je pak framework pro vkládání závislostí (angl. DI - Dependency Injection) Koin. Ten umožňuje použít při psaní návrhový vzor obráceného řízení (angl. IoC - Inversion of Control). Díky tomu lze snadno sdílet mezi jednotlivými komponenty aplikace například jednu instanci objektu klienta nebo připojení k databázi.

3.3 Klient

Technologie použité při realizaci klientské části.

3.3.1 Uživatelské rozhraní

Pro realizaci grafického uživatelského rozhraní (angl. GUI - Graphical User Interface) byla vybrána knihovna TornadoFX. Ta je postavená na platformě JavaFX a usnadňuje práci s ní pomocí Kotlinu. Také obsahuje několik rozšiřujících funkcí, jako je integrované vkládání závislostí, přidané primitivy pro snadnější implementaci často používaných návrhových vzorů MVC a MVP nebo třeba sběrnici pro komunikaci mezi komponenty.

3.3.2 Komunikace

Vzhledem k výběru frameworku Ktor pro část serverovou, byl i pro klientskou část vybrán http klient, který je jeho součástí. Alternativou, která se nabízela, bylo použít vestavěného klienta, který je součástí TornadoFX, nicméně ten je pouze velmi základní a nelze jej tak podrobně konfigurovat. Důležitými vlastnostmi vybraného klienta pro tento projekt je snadná serializace a deserializace objektů a také podpora autentifikace.

4 Realizace práce

Tato kapitola se již zabývá popisem vytvořené serverové a klientské aplikace. Aplikace byla vyvíjena v integrovaném vývojovém prostředí IntelliJ Idea ve verzi Ultimate, která je normálně placená, ale pro studenty je k dispozici zdarma[3]. Práce je koncipována jako jeden hlavní projekt nástroje Gradle (nástroj pro automatické sestavování a management závislostí), jehož součástí jsou pak tři pod-projekty. Jeden pro klientskou část, jeden pro serverovou část a jeden pro společný kód, který je mezi oběma částmi sdílený. Ten obsahuje třídy tvořící datový model pro komunikaci klienta se serverem (více popsáno později).

4.1 Server

Nejprve bude popsána realizace serverové části.

4.1.1 Zdroj dat

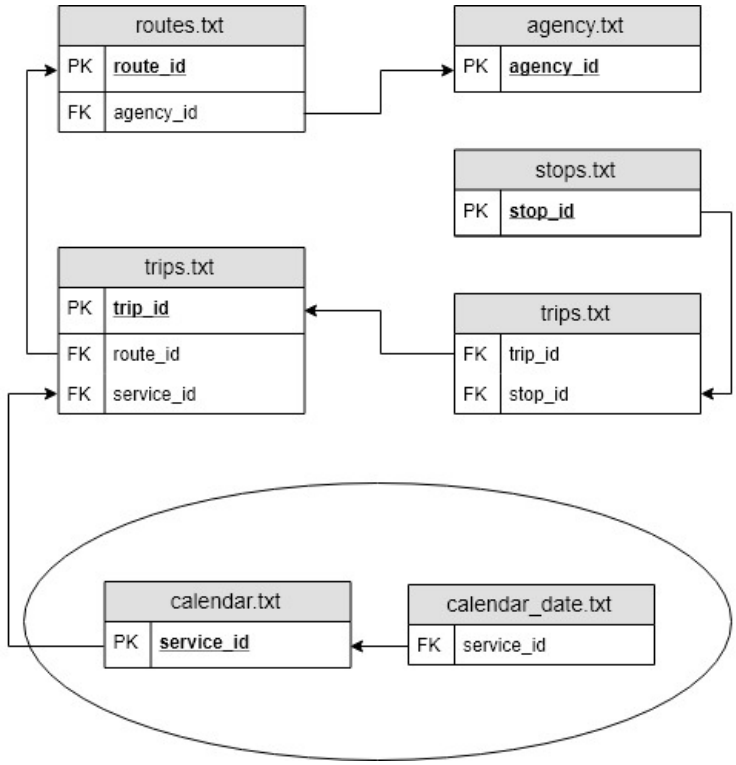
Jednou ze složitějších částí práce bylo paradoxně vůbec najít použitelný zdroj aktuálních jízdních řádů. V předchozích kapitolách byla jedna z popisovaných rolí serveru agregace jízdních řádu z více zdrojů, nicméně vytvořená aplikace pracuje pouze se zdrojem jedním, neboť jsem zjistil, že ze všech možných dopravních podniků svá data v otevřeném formátu poskytuje veřejně pouze Pražský dopravní podnik[2]. Ten nabízí ke stažení aktuální jízdní řády na 10 dní dopředu, které jsou denně aktualizovány. Jedná se o jeden velký archiv typu ZIP, který obsahuje textové soubory podle specifikace GTFS (General Transit Feed Specification). Data podléhají licenci CC-BY.

Formát, ve kterém jsou data poskytována (GTFS) naznačuje, že se jedná o redistribuci dat, které dopravní podnik poskytuje společnosti Google. Ta totiž stojí za vytvořením tohoto standartu, kdy původní význam písmena G bylo "Google"a až později, po adopci formátu více společnostmi, bylo změněno na "General". Tento formát se skládá z několika textových souborů, obsahující čárkou oddělené hodnoty. To je způsob zápisu dat použitý v souborech CSV (Comma-separated values), nicméně soubory mají podle standartu příponu txt. Standart GTFS definuje 5 povinných souborů. Dvojici, ze které musí být přítomen minimálně jeden soubor, a 8 souborů volitelných. Pro účely této aplikace stačí použít ty povinné a oba podmíněně povinné:

- `agency.txt` ..... Obsahuje informace o dopravcích
- `stops.txt` ..... Definuje jednotlivé zastávky, hlavně jejich jméno a polohu
- `routes.txt` ..... Definuje jednotlivé linky, jejich jméno a dopravce
- `trips.txt` ..... Definuje jednotlivé jízdy v rámci linky
- `stop_times.txt` .. Obsahuje posloupnost zastávek v rámci jízdy a jednotlivé příjezdy a odjezdy
- `calendart.txt` ..... Udává pro které dny v týdnu daná jízda platí
- `calendar_dates.txt` ..... Uvádí přímo data, kdy jízda platí platí nebo kdy naopak ne

Obr. 4.1: Seznam souborů ze standartu GTFS využívaných v programu

Jednotlivé soubory obsahují identifikátory různých jiných částí. Buď jsou v tom daném souboru definovány, nebo na id definované v jiném souboru odkazují. Vyjímkou jsou tak trochu soubory calendar.txt a calendar_dates.txt, které nemusí být přítomny nutně oba a tak není možné jednoznačně určit, který obsažené id definuje a který se na něj pouze odkazuje. Ve výsledku se v případě existence obou hodnoty z nich načtené zkombinují dohromady. Pokud budeme brát pouze jednotlivé identifikační sloupce, můžeme jejich vzájemné vztahy vyjádřit pomocí diagramu běžně používaného pro popis tabulek v relačních databázích (tzv. entity relation diagram).



Obr. 4.2: Diagram vazeb mezi identifikátory v jednotlivých souborech GTFS, vytvořeno pomocí draw.io

Po spuštění serveru vždy proběhne kontrola, zda databáze obsahuje data, případně jak jsou stará. Pokud jde o první spuštění a data v databázi nejsou, nebo pokud již server data jednou stáhl, ale jsou více než den staré, spustí se automaticky aktualizace. Data jsou stažena ze stránek dopravního podniku a jsou předána parseru. Ten ještě v paměti z archivu vybere potřebné soubory, které předá jako bytové pole jednotlivým částem parseru. Stažený soubor se tak vůbec nemusí dotknout disku, vše je vyřešeno přímo v paměti, což celý proces urychluje.

```
constructor(zipFile: ByteArray) {
    val zipInputStream = ZipInputStream(ByteArrayInputStream(zipFile))
    val fileMap = HashMap<String, ByteArray>()
    while (true) {
        val entry = zipInputStream.nextEntry ?: break
        fileMap[entry.name] = zipInputStream.readPacketAtLeast(entry.size)
            .readBytes()
    }
    if (fileMap.keys.intersect(conditionallyRequiredFiles.keys).isEmpty())
        throw GTFSParserException(
            "Does not contain calendar.txt nor calendar_dates.txt"
            )
    if (requiredFiles.keys.minus(fileMap.keys).isNotEmpty()) {
        throw GTFSParserException(
            "Some required files were not found:
            ${requiredFiles.keys.minus(fileMap.keys).joinToString(
                prefix = "[",
                postfix = "]"
            )}"
        )
    }
    dataMap = fileMap
      .filterKeys {
         requiredFiles.containsKey(it) or conditionallyRequiredFiles.containsKey(it)
      }.mapValues {
         DataFrame.readDelim(
            ByteArrayInputStream(it.value),
            colTypes = requiredFiles[it.key] ?: conditionallyRequiredFiles[it.key]!!
         )
      }
}
```

Výpis 4.1: Konstruktor GTFS parseru

Jednotlivé části parseru, která se každá věnuje svému souboru, jsou napsány jako koprogramy, které mohou běžet souběžně. Je zde využita dříve zmiňovaná knihovna Krangl, která umožňuje koprogram redukovat do jednoho funkčně zřetězeného výrazu. Nejdříve jsou dány ke zpracování soubory, kde jsou identifikátory definovány a na závěr jsou reference provázány. Data zpracovaná parserem jsou pak předána ke vložení do databáze.

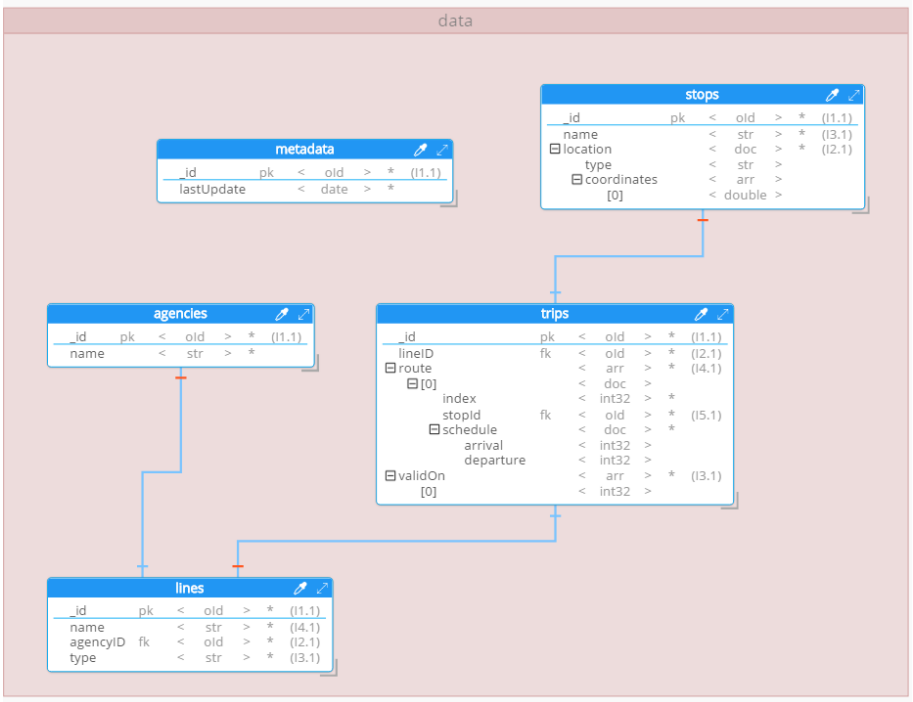
```
private fun CoroutineScope.parseStopsAsync(data: DataFrame) = async {
    data.select("stop_id", "stop_name", "stop_lat", "stop_lon")
        .rows.associate {
        it["stop_id"] as String to StopData(
            it["stop_name"] as String,
            it["stop_lat"] as Double,
            it["stop_lon"] as Double
        )
    }
  }
}
```

Výpis 4.2: Ukázka koprogramu pro parsování zastávek (stops.txt)

4.1.2 Datový model

Struktura a schéma do databáze ukládaných dat byly inspirovány formátem GTFS, ze kterého jsou získávány. Hlavním rozdílem je využití databází nabízené možnosti existence polí a vnořených dokumentů, pro sloučení spolu souvisejících údajů. Pro adresování jednotlivých dokumentů se využívají driverem automaticky generované identifikátory ObjectID. Ty se skládají z 12 bytů, kde 4 byty je aktuální čas vyjádřený jako počet sekund od epochy, 5 bytů náhodně vygenerovaných a 3 byty inkrementující čítač začínající od náhodného čísla. V případě použití pouze jednoho programu zapisujícího do databáze můžeme garantovat jeho globální unikátnost, nicméně i v případě současného zápisu více aplikacemi by byla šance na kolizi naprosto minimální.

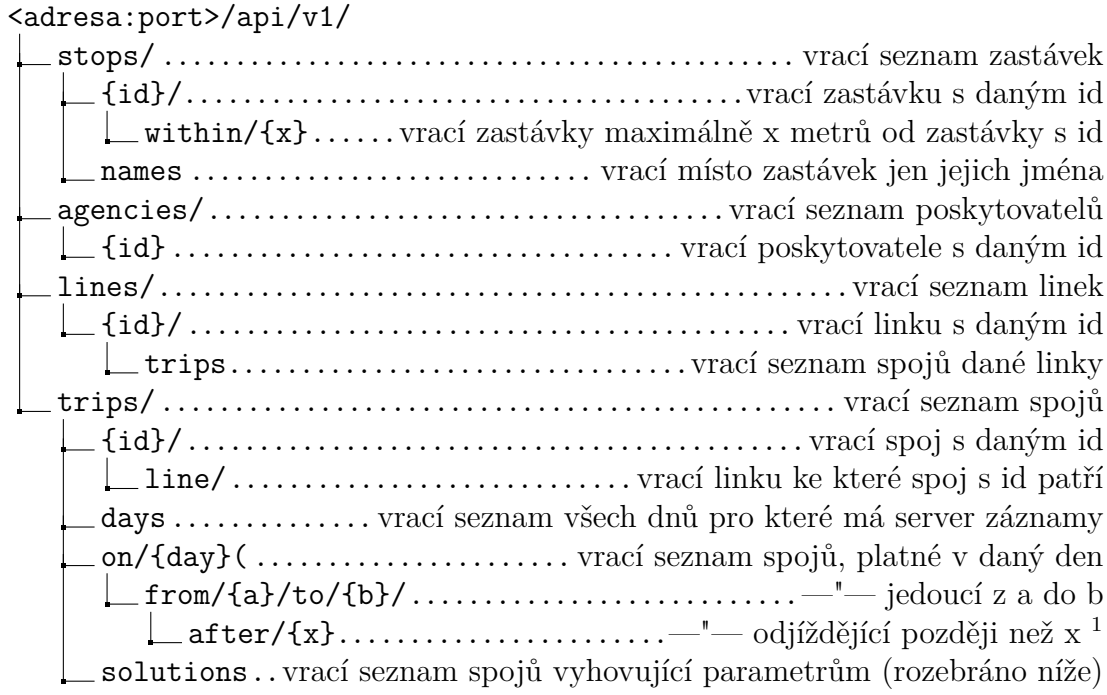
Podoba jednotlivých kolekcí je dána přímo datovými třídami, pomocí kterých jsou v kódu reprezentovány. O dodržení této podoby se stará přímo knihovna KMongo. Možnost databáze měnit agilně schéma byla při vývoji velmi užitečná. Pro přehled finální podoby dat v kolekcích pro účely dokumentace byl použit nástroj Hackolade, který umí z existující databáze schéma odvodit.



Obr. 4.3: Diagram kolekcí v databázi

4.1.3 Webové rozhraní

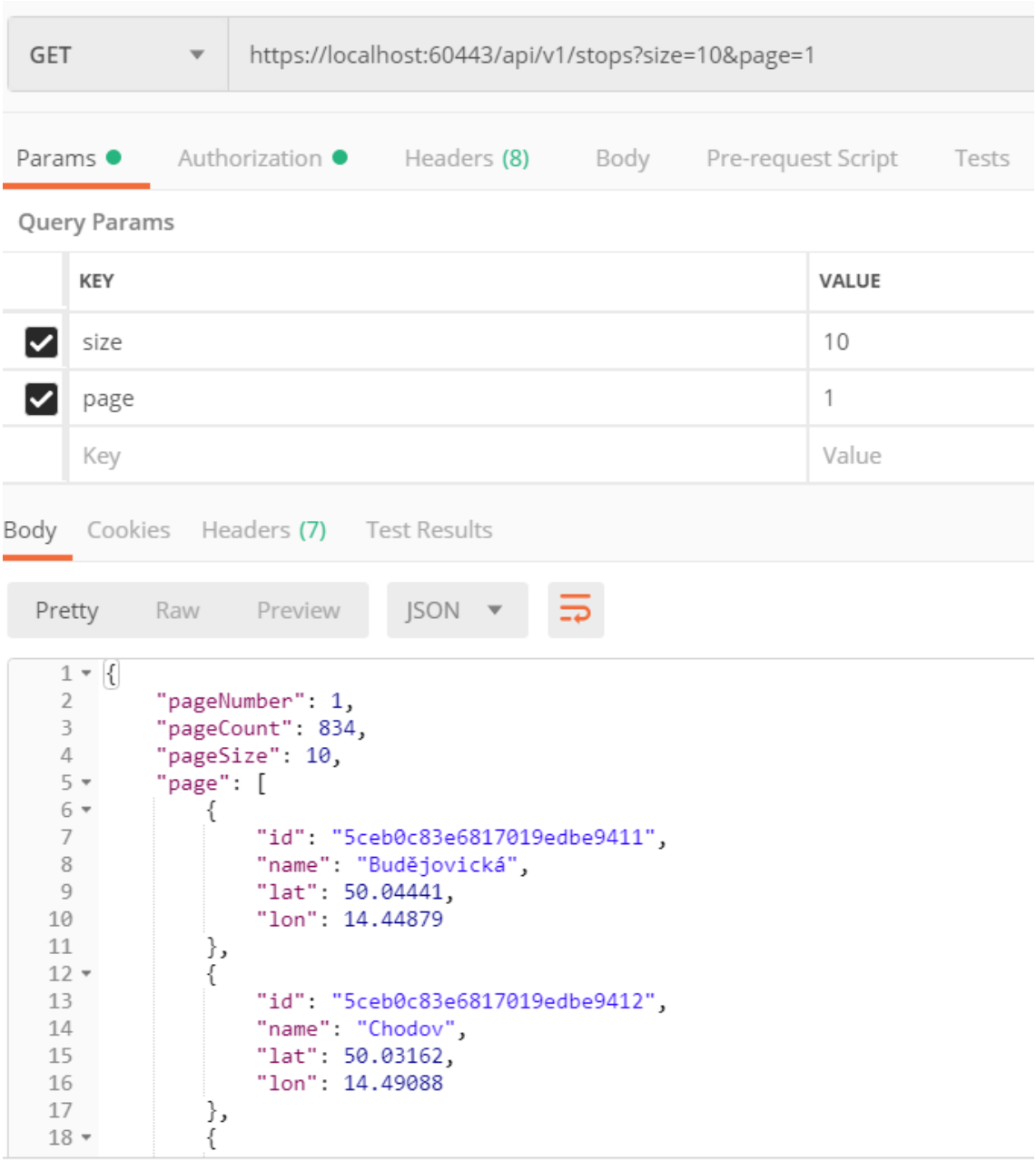
Webové rozhraní serverové části bylo realizováno jako RESTful HTTP API. Vzhledem k tomu, že server informace pouze redistribuuje klientům, byla implementována pouze metoda GET. Jednotlivé adresovatelné prostředky byly modelovány tak, aby zrcadlily strukturu uložených dat v databázi. Každá kolekce je tedy reprezentována jednou větví v hierarchii URI. Výjimkou je prostředek *solutions*, který provádí v databázi pokročilou agregaci na základě specifikovaných parametrů a výsledek vrací v objektu *solution*. Jejím důvodem existence je umožnění klientovi hledat spoje mezi skupinami zastávek přímo na serveru, což je mnohem rychlejší díky indexování databáze.



Obr. 4.4: Popis jednotlivých adresovatelných prostředků API

U většiny dotazů, které vracejí větší počet hodnot, je možné specifikovat parametr pro řazení *sort*, který může být jméno pole požadovaného objektu, podle kterého chceme řadit. Je taky možné specifikovat řazení sestupné a to uvedením znaménka mínus před dané jméno. Krom řazení lze pro tyto dotazy specifikovat také dvojici celočíselných parametrů *size* a *page*. Ty způsobí zabalení daného seznamu hodnot do objektu simulující stránku, jejíž interní seznam hodnot má maximální velikost rovnu parametru *size*. Vrácený objekt krom informací o jeho velikosti obsahuje jeho pořadové číslo a kolik stránek s touto velikostí ještě existuje.

[1] kde x je počet sekund během 1 dne (1..86400)



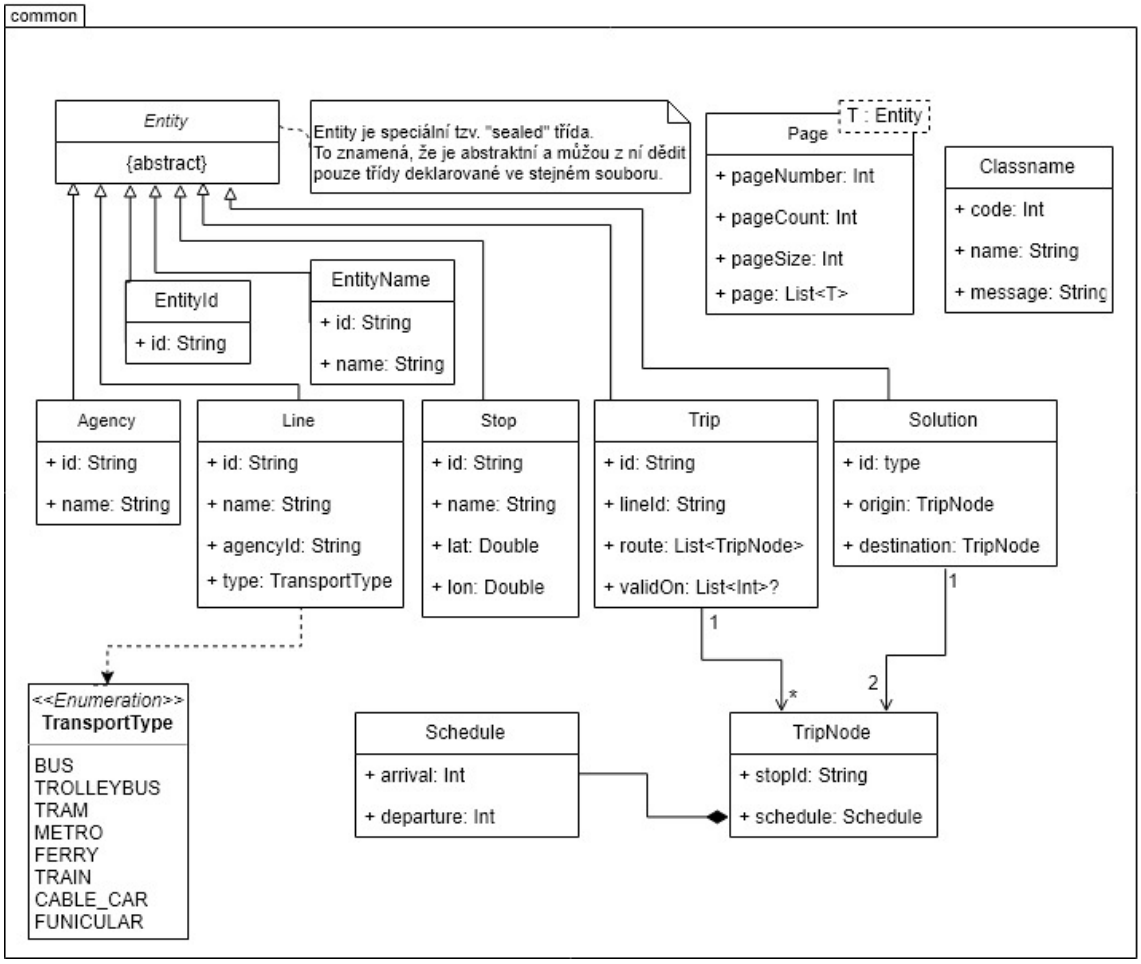
Obr. 4.5: Ukázka odpovědi serveru se stránkou o velikosti 10 položek

To umožňuje klientovi požádat o všechny výsledky po menších částech. To bylo nutné implementovat z důvodu, že podpora standardních HTTP řešení, jako částečných odpovědí (partial response) nebo stremování po kusech (chunked transfer encoding), není v Ktor frameworku doposud funkční v kombinaci s automatickou konverzí do používaného formátu. Toto řešení je podobné konceptu používaném často při tvorbě rozhraní přímo pro frontend a říká se mu stránkování (pagination). Krom těchto obecných parametrů je možné specifikovat u určitých požadavků i jiné.

- `stops/`
  - `name` ..... Regulární výraz, který musí jméno splňovat
  - `within` ..... Udává uzavřený polygon ze souřadnic podle specifikace GeoJson ve kterém musí dané zastávky ležet
- `stops/{id}/within/{x}`
  - `idOnly` .. příznak (true/false), udávající, jestli vrátit pouze seznam id nebo celé objekty
- `stops/names/`
  - `within` ..... Udává uzavřený polygon ze souřadnic podle specifikace GeoJson ve kterém musí dané zastávky ležet
- `agencies/`
  - `name` ..... Regulární výraz, který musí jméno splňovat
- `lines/`
  - `name` ..... Regulární výraz, který musí jméno splňovat
  - `agency` ..... Identifikátor poskytovatele, kterému je agencyID rovno
  - `type` ..... Seznam způsobů přepravy, které chceme zahrnout
- `trips/`
  - `days` ..... Seznam dní, kdy musí daný spoj platit
  - `stops` ..... Seznam zastávek, kterými musí spoj projíždět
- `trips/solutions/`
  - `day` ..... Den, kdy musí daný spoj jet
  - `origin` ..... Seznam id zastávek, ze kterých může odpovídající spoj vyjíždět
  - `dest` ..... Seznam id zastávek, z nichž aspoň do 1 spoj jede
  - `after` .. Čas v sekundách, kdy nejdřív může spoj vyjíždět ze zdrojové zastávky[2]

Obr. 4.6: Popis specifikovatelných parametrů

Tím se dostáváme k formátu komunikace. Pro komunikaci mezi serverem a klientem se používá formát JSON (JavaScript Object Notatation). Do tohoto formátu jsou automaticky serializovány posílané objekty. To se děje automaticky pomocí Ktor modulu *ContentNegotation*, který používá interně knihovnu *Jackson*. Ekvivalent této funkce je pak nainstalován i v klientské části pro deserializaci zpět do objektů. Pro vyhnutí se duplicitním definicím tříd posílaných tímto způsobem, jsou tyto definovány v projektu, který je společný serverové i klientské části.



Obr. 4.7: UML diagram tříd společných klientu i serveru

4.1.4 Zabezpečení

Vzhledem k tomu, že server redistribuuje již tak veřejná data a nejsou prováděny žádné zápisy do databáze ze strany klienta, bylo jako adekvátní řešení zabezpečené komunikace vybráno použití šifrovaného protokolu HTTPS v kombinaci s autentifikací HTTP basic. Její nevýhodou je, že uživatelské jméno a heslo posílá pouze zakódované podle Base64, což je způsob, jak vyjádřit případné speciální znaky ve jméně či heslu pomocí symbolů přenositelných protokolem HTTP a nejedná se v žádném případě o kryptografickou operaci. Celé zabezpečení pak ve výsledku spoléhá na šifrování samotného protokolu HTTPS a pokud by se podařilo útočníkovi podstrčit aplikaci jeho vlastní certifikát, byl by schopen figurovat jako prostředník (tzv. Man-in-the-middle - MITM) a odposlouchávat veškerou komunikaci včetně uživatelských jmen a hesel.

Na straně serveru bylo načítání jednotlivých uživatelských jmen a hesel realizo-

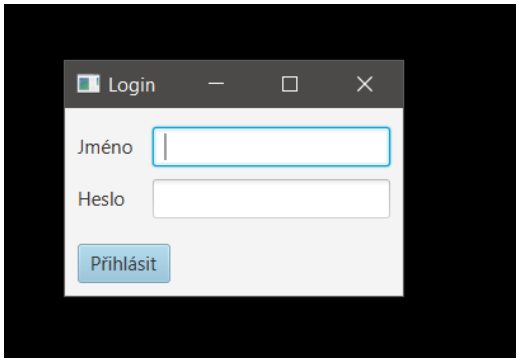
váno pomocí čtení ze souboru při jeho startu. To je dostatečné pro demonstrační účely. V reálném nasazení by ale nejspíše bylo zvoleno uložení těchto údajů v šifrované podobě do databáze.

4.2 Klient

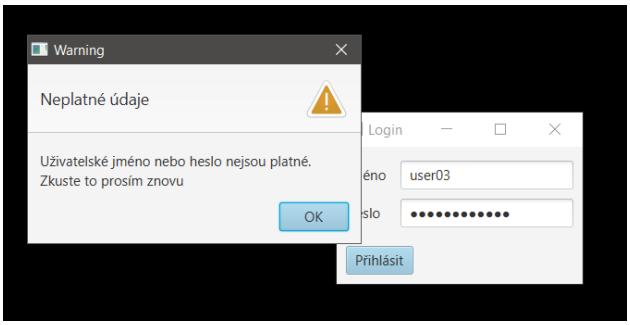
Tato kapitola se zabývá popisem realizace klienta.

4.2.1 Uživatelské rozhraní

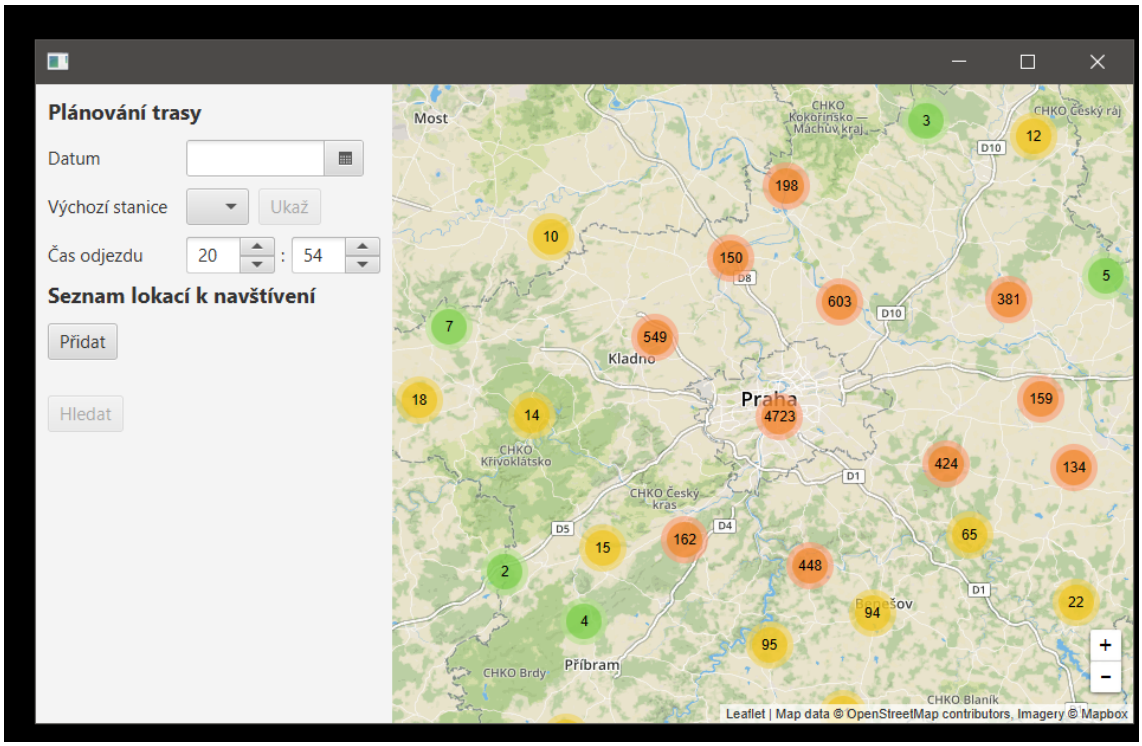
Jako první se po spuštění aplikace zobrazí přihlašovací okno, vyzývající uživatele k zadání jeho uživatelského jména a hesla. Obě dvě textová pole jsou v kódu specifikovaná jako povinná, takže se nevykoná žádná akce, pokud nejsou obě dvě vyplněna. Po zadání údajů a potvrzení tlačítkem *Přihlásit* jsou na pozadí tyto údaje ověřeny na serveru. Mezitím co se čeká na odpověď serveru, je celé přihlašovací okno překryto indikátorem načítání. Jakmile je kontrolérem vrácen výsledek, je na jeho základě buďto zobrazeno varování o nesprávných přihlašovacích údajích, hlášení o chybě serveru nebo jsou údaje uloženy do modelu a aktuální okno je nahrazeno hlavním oknem aplikace.



Obr. 4.8: Přihlašovací okno

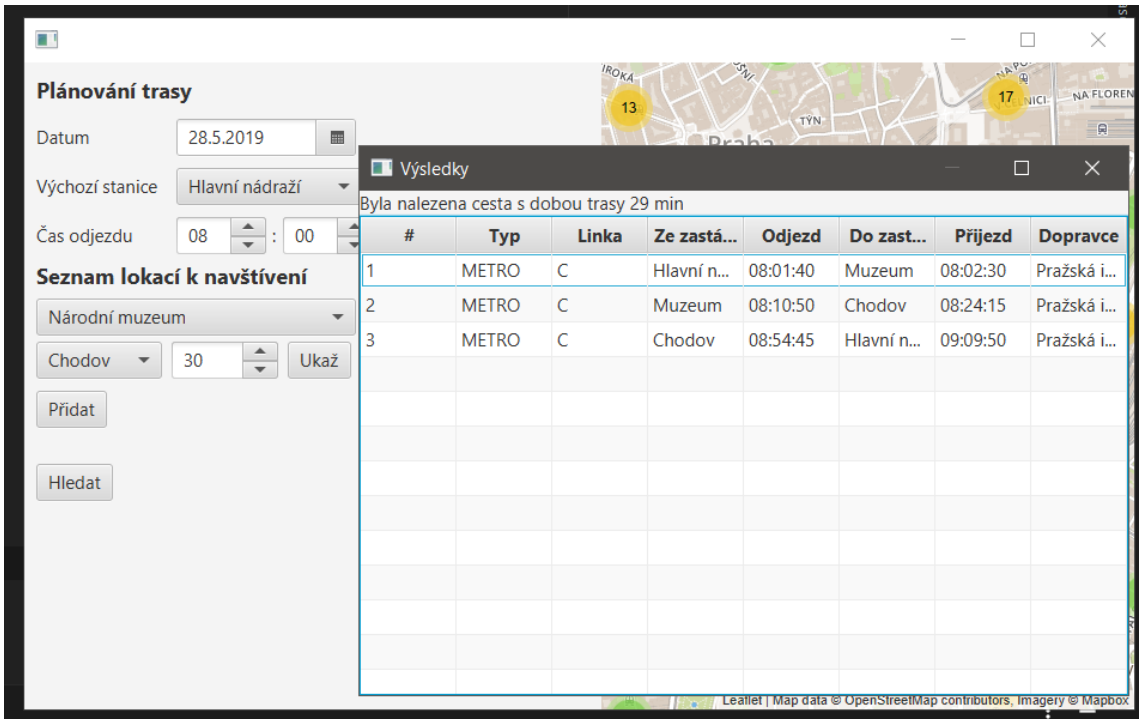


Obr. 4.9: Zobrazení varování v případě chybných údajů



Obr. 4.10: Hlavní okno programu hned po přihlášení

Hlavní okno programu je pomocí manažeru rozložení *BorderPane* rozdělena na hlavní část s mapou a postranní oblast pro zadání hodnot. To uživateli umožňuje zadat den, pro který chce hledat spojení, výchozí stanici, ze které chce trasu začít a čas, kdy chce nejdříve vyjet. Zastávku je možné vybrat z tzv. komboboxu. Při klepnutím na něj se vysune roletka s možnostmi, ze kterých můžeme vybrat námi požadovanou zastávku. V zobrazených možnostech lze vyhledávat napsáním části jména požadované stanice. Jakmile je v komponentě vybrána hodnota, povolí se vedle ní dříve deaktivované tlačítko *Ukaž*, po jehož klepnutí se vybraná zastávka zobrazí na mapě. To je výhodné pro ověření výběru správné zastávky, zvláště pokud mají stejná jména. Pod tím následuje oblast pro přidání jednotlivých lokací, které chceme navštívit. Výběr je vždy doplněn polem pro zadání číselné hodnoty maximální doby, po kterou máme v plánu se v dané lokalitě zdržet. Opět je zde tlačítko umožňující zobrazení vybrané zastávky na mapce. Jakmile jsou zadány veškeré potřebné vstupní údaje, je povoleno tlačítko pro zahájení hledání trasy. To se nachází dole pod seznamem lokalit. Všechny uživatelem zadávené pole jsou již před výpočtem kontrolovány, zda obsahují validní data a většina z nich ani zadat neplatné údaje uživateli neumožní. Po započetí výpočtu se přes vstupní pole, podobně jako u přihlášení, zobrazí indikátor načítání.Po dokončení výpočtu se uživateli zobrazí v dalším okně nalezená trasa.



Obr. 4.11: Zobrazení nalezené trasy

4.2.2 Výpočet trasy

Výpočet trasy probíhá v dedikovaném kontroléru. Ten má k dispozici instanci webového klienta pro komunikaci se serverem a fond 4 vláken, které využívá pro vykonávání koprogramů. Samotný koprogram je pak rekurzivně volán dokud nejsou vyzkoušeny všechny možnosti.

```
fun calculate(input: Input): TSPSolution = runBlocking {
        withContext(pool) {
            val expected = input.destinations.size.factorial()/2
            val solutions = mutableListOf<TSPSolution>()
            val jobs = mutableListOf<Job>()
            val channel = Channel<TSPSolution>(Channel.UNLIMITED)
            launch {
                for (solution in channel) {
                    solutions.add(solution)
                }
            }
            try {
                withTimeout(TIMEOUT.toLong()){
                    for (dest in input.destinations) {
                        jobs.add(
                            branchAsync(input.day, input.time, input.origin, input.
                                ↪ origin, dest, input.destinations.minus(dest.key)
                                ↪ , solutionChannel = channel)
                        )
                    }
                    jobs.joinAll()
                }
            }catch (ex: TimeoutCancellationException){
                logger.warn("Calculation ran longer then $TIMEOUT ms and was
                    ↪ cancelled.")
            }
            channel.close()
            logger.debug("Calculation used ${counter.get()} coroutines. Expected
                ↪ $expected")
            counter.set(0)
            return@withContext solutions.filterIsInstance<TSPSolution.Path>().minBy
                ↪ { it.cost } ?: TSPSolution.Nonexistant
        }
    }
```

Výpis 4.3: Ukázka koprogramu pro parsování zastávek (stops.txt)

```
1  private fun CoroutineScope.branchAsync(
2          day: Int,
3          time: Int,
4          origin: String,
5          from: String,
6          to: Map.Entry<String, Int>,
7          next: Map<String, Int>,
8          previous: List<Entity.Solution> = emptyList(),
9          solutionChannel: Channel<TSPSolution>
10     ):Job = launch {
11         val cid = counter.incrementAndGet()
12         logger.debug("CID $cid started (day=$day, time=$time, origin=$origin, from=
               ↪ $from, to=${to.key}@${to.value}, next=$next, previous=$previous)")
13         //find stops nearby
14         val _from = client.get<List<Entity.EntityId>>("$BASE_URL/$API_PATH/stops/
               ↪ $from/within/$DISTANCE?idonly=true").also{
15             logger.debug("CID $cid: Origins: ($from) -> ${it.joinToString()}")
16         }
17         val _to = client.get<List<Entity.EntityId>>("$BASE_URL/$API_PATH/stops/${
               ↪ to.key}/within/$DISTANCE?idonly=true").also{
18             logger.debug("CID $cid: Dests: (${to.key}) -> ${it.joinToString()}")
19         }
20         //get solution
21         val query = "day=$day&origin=${if (_from.size == 1) _from.first().id else
               ↪ _from.joinToString(separator = ",",transform = { it.id })}&dest=${if
               ↪  (_to.size == 1) _to.first().id else _to.joinToString(separator = ",
               ↪ ",transform = { it.id })}&after=$time"
22         val trips = client.get<List<Entity.Solution>>("$BASE_URL/$API_PATH/trips/
               ↪ solutions?$query").also{
23             logger.debug("CID $cid: Trips: ${it.joinToString()}")
24         }
25         //select best solution
26         if (trips.isEmpty())  logger.debug("CID $cid: No trip between $from and $to
               ↪  after $time was found.").also { return@launch }
27         val sol1 = trips.minBy { it.origin.schedule.departure }!!.also {
28             logger.debug("CID $cid: Selected trip: ${it.id}, ${it.origin.stopId}@${
                   ↪ it.origin.schedule.departure} -> ${it.destination.stopId}@${it.
                   ↪ destination.schedule.arrival}")
29         }
30         val t = (time + (to.value*60) + (sol1.destination.schedule.arrival - sol1.
               ↪ origin.schedule.departure)).also {
31             logger.debug("CID $cid: Time is now $it. ($time + ${to.value*60} + (${
                   ↪ sol1.destination.schedule.arrival} - ${sol1.origin.schedule.
                   ↪ departure}))")
32         }
```

Výpis 4.4: Kód pro hledání řešení

```
//check if next is not empty, if not, recurse
        if (next.isNotEmpty()) {
            next.map{
                branchAsync(
                    day,
                    t,
                    origin,
                    to.key,
                    it,
                    next.minus(it.key),
                    previous + sol1,
                    solutionChannel
                )
            }.joinAll().also {
                logger.debug("CID $cid: children finished.")
            }
            return@launch
        } else {
            //Calculate path back to origin
            logger.debug("CID $cid: all destinations visited, calculating route back to origin.")
            val ___from = client.get<List<Entity.EntityId>>("$BASE_URL/$API_PATH/stops/${to.key}/within/$DISTANCE?idonly=true").also{
                logger.debug("CID $cid: Return Origins: (${to.key}) -> ${it.joinToString()}")
            }
            val ___to = client.get<List<Entity.EntityId>>("$BASE_URL/$API_PATH/stops/$origin/within/$DISTANCE?idonly=true").also{
                logger.debug("CID $cid: Return Dests: ($origin) -> ${it.joinToString()}")
            }
            val q = "day=$day&origin=${if(__from.size==1) __from.first().id else __from.joinToString(separator = ",",transform = {it.id})}&dest=${if(__to.size==1) __to.first().id else __to.joinToString(separator = ",", transform = {it.id})}&after=$t"
            val trips = client.get<List<Entity.Solution>>("$BASE_URL/$API_PATH/trips/solutions?$q").also{
                logger.debug("CID $cid: Return trips: ${it.joinToString()}")
            }
            if(trips.isNotEmpty()){
                val sol2 = trips.minBy { it.origin.schedule.departure }!!
                val chain = previous+sol1+sol2
                val cost = chain.sumBy { it.destination.schedule.arrival - it.origin.schedule.departure }
                solutionChannel.send(TSPSolution.Path(chain, cost)).also {
                    logger.debug("CID $cid: found solutin with cost $cost !")
                }
            }else{
                solutionChannel.send(TSPSolution.Nonexistant).also { logger.debug("CID $cid: no route home found.") }
            }
            return@launch
    }
}
```

5 Závěr

V rámci této práce byla navrhnuta aplikace typu klient-server pro hledání optimální trasy v síti městské hromadné dopravy. Podařilo se najít pouze základní řešení, které je schopno najít nejkratší okruh procházející danými body pouze v případě, že mezi nimi existrují přímá spojení. To se ukázalo v praxi jako velmi limitující i ve velkých doprávních sítích jako je například Praha.

Literatura

[1] General Transit Feed Specification reference. *Google developers* [online]. [cit. 2019-05-27]. Dostupné z: https://developers.google.com/transit/gtfs/reference/

[2] Otevřená data PID. *Pražská integrovaná doprava* [online]. Praha, c2019 [cit. 2019-05-27]. Dostupné z: https://pid.cz/o-systemu/opendata/

[3] Free individual licenses for students and faculty members. *JetBrains* [online]. [cit. 2019-05-27]. Dostupné z: https://www.jetbrains.com/student/

[4] Proč použít Kotlin místo Javy. In: *SBlog* [online]. Seznam.cz, 2018 [cit. 2019-05-26]. Dostupné z: https://blog.seznam.cz/2018/10/proc-pouzit-kotlin-misto-javy/

[5] Http benchmarks. *Github* [online]. 2018 [cit. 2019-05-26]. Dostupné z: https://github.com/orangy/http-benchmarks

[6] *Kotlin* [online]. [cit. 2019-05-26]. Dostupné z: http://kotlinlang.org

[7] *Ktor* [online]. [cit. 2019-05-26]. Dostupné z: https://ktor.io

[8] Learn Kotlin. *Kotlin* [online]. [cit. 2019-05-26]. Dostupné z: https://kotlinlang.org/docs/reference/

[9] BIEHL, Matthias. *API Architecture: The Big Picture for Building APIs*. CreateSpace Independent Publishing Platform, 2015. ISBN 978-1-5086-7664-5.

[10] LUECKE, David. Design patterns for modern web APIs. In: *Feathers blog* [online]. 2018 [cit. 2019-05-24]. Dostupné z: https://blog.feathersjs.com/design-patterns-for-modern-web-apis-1f046635215

[11] Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems* [online]. **2013**(5.4), 16-19 [cit. 2019-05-20]. ISSN 2249-0868. Dostupné z: https://www.researchgate.net/profile/Dikshay_Poojary/publication/302557703_Article_Ty
Type-of-nosql-databases-and-its-comparison-with-relational-databases.pdf

[12] MATHEW, Roy, Divya CHERUKUPALLI, Kevin PUSICH a Kevin ZHAO. *Traveling Salesman Algorithms: From Naive to Christofide* [online]. [cit. 2019-05-20]. Dostupné z: https://cse442-17f.github.io/Traveling-Salesman-Algorithms/

[13] WEISSTEIN, Eric. Hamiltonian Cycle. *MathWorld* [online]. [cit. 2019-05-20]. Dostupné z: http://mathworld.wolfram.com/HamiltonianCycle.html

[14] WEISSTEIN, Eric. Traveling Salesman Problem. *MathWorld* [online]. [cit. 2019-05-20]. Dostupné z: http://mathworld.wolfram.com/TravelingSalesmanProblem.html

[15] Unsolvable Problems. *Old Dominion university: computer science department* [online]. Norfolk, 2013 [cit. 2019-05-20]. Dostupné z: https://www.cs.odu.edu/ toida/nerzic/390teched/computability/unsolv-1.html

[16] WEISSTEIN, Eric. Halting problem. In: *MathWorld* [online]. [cit. 2019-05-20]. Dostupné z: http://mathworld.wolfram.com/HaltingProblem.html

[17] Types of Turing Machines. *Old Dominion university: computer science department* [online]. Norfolk [cit. 2019-05-20]. Dostupné z: https://www.cs.odu.edu/ toida/nerzic/390teched/tm/othertms.html

[18] Introduction to Algorithms: Notes on Turing Machines. In: *Cornell University: Computer science department* [online]. New York, 2009 [cit. 2019-05-20]. Dostupné z: http://www.cs.cornell.edu/courses/cs4820/2010sp/handouts/turingm.pdf

[19] Introduction: What is a Turing machine?. *University of Cambridge* [online]. Cambridge, 2012 [cit. 2019-05-20]. Dostupné z: https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html

[20] The Millennium Prize Problems. *Clay Mathematics Institute* [online]. Oxford: Clay Mathematics Institute, 2019 [cit. 2019-05-20]. Dostupné z: https://www.claymath.org/millennium-problems/millennium-prize-problems

[21] Decision problem. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-05-20]. Dostupné z: https://en.wikipedia.org/wiki/Decision_problem

[22] P (complexity). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-05-19]. Dostupné z: https://en.wikipedia.org/wiki/P_(complexity)

[23] Časová a prostorová složitost algoritmů. *Mendelova universita v Brně* [online]. Brno: Mendelova universita v Brně [cit. 2019-05-19]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=376

[24] K čemu je datová schránka a jak ji využít. *Vimkamklikam.cz* [online]. Jihlava, 2017 [cit. 2019-05-19]. Dostupné z: https://www.vimkamklikam.cz/rady-a-tipy/k-cemu-je-datova-schranka-a-jak-ji-vyuzit

[25] BURGET, R. *Teoretická informatika*. Brno: Vysoké učení technické v Brně, 2012. ISBN 978-80-2-4-4897-1.

Seznam symbolů, veličin a zkratek

RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computing
NP	Nondeterministic Polynomial time
P	Polynomial time
TSP	Travelling Salesman Problem
SQL	Structured Query Language
ACID	Atomicity, Consistency, Isolation, Durability
NoSQL	Non SQL nebo Not Only SQL
BASE	Basically Available, Soft state, Eventual consistency
API	Application Programming Interface
REST	REpresentational State Transfer
RPC	Remote Procedure Call
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notatation
XML	eXtensible Markup Language
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SMTP	Simple Mail Transfer Protocol
JMS	Java Messaging Services
WSDL	Web Services Description Language
IDE	Integrated Development Enviroment
JVM	Java Virtual Machine
NPE	Null Pointer Exception
JAR	Java ARchive
WAR	Web application ARchive
BSON	Binary JSON
DI	Dependency Injection
IoC	Inversion of Control
MVC	Model-View-Controller
MVP	Model-View-Presenter
GTFS	General Transit Feed Specification
CSV	Comma-separated values
URI	Uniform Resource Identificator
MITM	Man-in-the-middle
GUI	Graphical User Interface

Seznam příloh