

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5382-52598

**VYUŽITIE GRAFOVEJ DATABÁZY V PRAXI  
BAKALÁRSKA PRÁCA**

**2017**

**Juraj Kubričan**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5382-52598

**VYUŽITIE GRAFOVEJ DATABÁZY V PRAXI  
BAKALÁRSKA PRÁCA**

Študijný program: Aplikovaná informatika  
Číslo študijného odboru: 2511  
Názov študijného odboru: 9.2.9 Aplikovaná informatika  
Školiace pracovisko: Ústav informatiky a matematiky  
Vedúci záverečnej práce: Ing. Maroš Čavojský

**Bratislava 2017**

**Juraj Kubričan**



## ZADANIE BAKALÁRSKEJ PRÁCE

Študent: **Juraj Kubričan**  
ID študenta: **52598**  
Študijný program: **aplikovaná informatika**  
Študijný odbor: **9.2.9. aplikovaná informatika**  
Vedúci práce: **Ing. Maroš Čavojský**  
Miesto vypracovania: **Ústav informatiky a matematiky**

Názov práce: **Využitie grafovej databázy v praxi**

Jazyk, v ktorom sa práca vypracuje: **slovenský jazyk**

Špecifikácia zadania:

V dnešnej dobe sa okrem tradičných zaužívaných relačných databáz, využívajú aj menej známe grafové databázy, v ktorých sú dáta uložené odlišným spôsobom ako v relačných databázach. Cieľom práce je oboznámiť sa s jednotlivými predstaviteľmi grafových databáz, vybrať jedného a navrhnúť a implementovať využitie vybranej grafovej databázy na reálnom príklade.

Úlohy:

1. Naštudujte si literatúru ohľadom jednotlivých predstaviteľov grafových databáz
2. Vyberte jedného predstaviteľa grafových databáz
3. Navrhnite reálny príklad pre implementáciu grafovej databázy
4. Implementujte reálny príklad pre implementáciu grafovej databázy
5. Zhodnoťte a uveďte výhody použitia grafovej databázy oproti iným typom databáz (relačné, dokumentové,...) v implementovanom reálnom príklade

Zoznam odbornej literatúry:


1. Ian Robinson, Jim Webber, and Emil Eifrem: Graph Databases, O'Reilly Media, Inc. USA 2015, p.224, ISBN: 978-1-491-93200-1


Riešenie zadania práce od: **19. 09. 2016**

Dátum odovzdania práce: **19. 05. 2017**

  
**Juraj Kubričan**  
študent

SLOVENSKÁ TECHNICKÁ UNIVERZITA  
V BRATISLAVE  
Fakulta elektrotechniky a informatiky  
Ústav informatiky a matematiky  
Ilkovičova 3, 812 19 Bratislava

  
**prof. RNDr. Otokar Grošek, PhD.**  
vedúci pracoviska

  
**prof. Dr. Ing. Miloš Oravec**  
garant študijného programu

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Juraj Kubričan
Bakalárska práca:	Využitie grafovej databázy v praxi
Vedúci záverečnej práce:	Ing. Maroš Čavojský
Miesto a rok predloženia práce:	Bratislava 2017

Bakalárska práca sa zaoberá oboznámením sa s rôznymi predstaviteľmi grafových databáz a naimplementovaním aplikácie ktorá bude využívať jedného predstaviteľa grafovej databázy. V prvej časti sa nachádza teoretický úvod do problematiky grafových databáz, ich výhody a nevýhody oproti relačným databázam z hľadiska štruktúry a výkonu. V druhej časti sa nachádza špecifikácia a návrh našej aplikácie s využitím UML diagramov. V časti implementácia sa nachádza prehľad ostatných použitých technológií, a popísaný proces implementácie našej aplikácie

Kľúčové slová: Využitie grafovej databázy v praxi

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Juraj Kubričan
Bachelor Thesis:	Graph database use in a real world application
Supervisor:	Ing. Maroš Čavojský
Place and year of submission:	Bratislava 2017

abstractEN

Keywords: Graph database use in a real world application

## Vyhlásenie autora

Podpísaný(á) Juraj Kubričan čestne vyhlasujem, že som bakalársku prácu Využitie grafovej databázy v praxi vypracoval(a) na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Vedúcim mojej bakalárskej práce bol Ing. Maroš Čavojský.

Bratislava, dňa 13.5.2017

.....

podpis autora

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Analýza problému</b>	<b>2</b>
1.1 Grafové databázy . . . . .	2
1.2 Výkon grafovej databázy pri dotazoch na vzťahy . . . . .	2
1.3 Vzťahy v relačných databázach . . . . .	2
1.4 Vzťahy v dokumentových databázach . . . . .	4
1.5 Výber grafovej databázy . . . . .	4
1.5.1 Neo4J . . . . .	5
1.5.2 OrientDB . . . . .	5
1.5.3 Titan . . . . .	6
1.6 Dopytovací jazyk Cypher . . . . .	6
<b>2 Špecifikácia</b>	<b>8</b>
2.1 Funkcionálne požiadavky . . . . .	8
2.2 Nefunkcionálne požiadavky . . . . .	9
<b>3 Návrh</b>	<b>10</b>
3.0.1 Prípady použitia . . . . .	10
3.0.2 Štruktúra databázy . . . . .	10
<b>4 Implementácia</b>	<b>15</b>
4.1 Použité technológie . . . . .	15
4.1.1 Laravel framework . . . . .	15
4.1.2 NeoEloquent OGM . . . . .	15
4.1.3 GraphAware Neo4j PHP Client . . . . .	15
4.1.4 Mapbox.js . . . . .	15
4.1.5 Handlebars.js . . . . .	16
4.1.6 Rome2Rio Api . . . . .	16
4.2 Inštalácia a konfigurácia Laravel Framework-u . . . . .	16
4.3 Inštalácia a konfigurácia databázy neo4j . . . . .	16
4.4 OGM . . . . .	17
4.5 Autentifikácia . . . . .	18
4.6 Databázové dopyty pomocou OGM . . . . .	20
4.7 Databázové dopyty pomocou Cypher klienta . . . . .	21

4.8	TSP . . . . .	22
4.9	Rome2rio API . . . . .	23
4.10	Mapa . . . . .	25
4.11	Vykresľovanie tabuliek . . . . .	25
4.12	Vzhľad . . . . .	26
<b>5</b>	<b>Výsledky práce</b>	<b>29</b>
5.1	Testy rýchlosti . . . . .	29
<b>6</b>	<b>Záver</b>	<b>32</b>
	<b>Záver</b>	<b>33</b>
	<b>Zoznam použitej literatúry</b>	<b>34</b>
	<b>Prílohy</b>	<b>I</b>



## Zoznam obrázkov a tabuliek

Obrázok 1	Ukážka riešenia vzťahov v relačnej databáze . . . . .	3
Obrázok 2	Ukážka riešenia vzťahov v dokumentovej databáze . . . . .	4
Obrázok 3	Prípad použitia - Registrácia, autentifikácia a nastavenia . . . . .	11
Obrázok 4	Prípad použitia - Registrácia, autentifikácia a nastavenia . . . . .	11
Obrázok 5	Prípad použitia - dashboard . . . . .	12
Obrázok 6	Prípad použitia - tsp . . . . .	12
Obrázok 7	Štruktúra databázy . . . . .	13
Obrázok 8	Príklad dát v databáze . . . . .	14
Obrázok 9	Ukážka hlavnej obrazovky používateľa . . . . .	27
Obrázok 10	Ukážka obrazovky TSP . . . . .	28
Obrázok 11	Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií . . . . .	30
Tabuľka 1	Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií . . . . .	29

## **Zoznam skratiek a značiek**

TSP - The Travelling Salesman Problem

UML - Unified Modeling Language verzie 2.5

GDBMS - Graph Database Nanagement System

RDBMS - Relational Database Management System

ORM - Object to Relational Mapping

OGM - Object to Graph Mapping

URL - Uniform Resource Locator

SSH - Secure SHell

CDN - Content Delivery Network

UX - User Experience

AJAX - Asynchronous Javascript and XML

REST - Representational State Sransfer

MVC - Model View Controller

## Zoznam algoritmov

1	Príklady dopytov v jazyku Cypher . . . . .	7
2	Inštalácia a konfigurácia Frameworku Laravel pre prácu z Neo4J . . . . .	17
3	Ukážka triedy neoEloquent . . . . .	19
4	Ukážka autentifikovateľnej triedy . . . . .	20
5	Pseudokód algoritmu na získavanie dát pre zoznam obľúbených destinácií pomocou OGM . . . . .	21
6	Dopyt na dáta pre zoznam obľúbených destinácií v jazyku Cypher . . . . .	22
7	Ukážka riešenia TSP pomocou dopytovacieho jazyka Cypher . . . . .	24

# Úvod

V dnešnej dobe, keď

# 1 Analýza problému

## 1.1 Grafové databázy

Modelovanie grafovej databázy prirodzene zapadá do spôsobu akým bežne abstrahujeme problémy pri vývoji softvéru. Pri návrhu softvéru objekty opisujeme obdĺžnikmi alebo kruhmi a súvislosti medzi nimi šípkami, či čiarami. Moderné grafové databázy sú viac ako akákoľvek iná databázová technológia vhodná na takúto reprezentáciu, lebo to, čo namodelujeme na papier vieme priamo implementovať v našej grafovej databáze. [2]

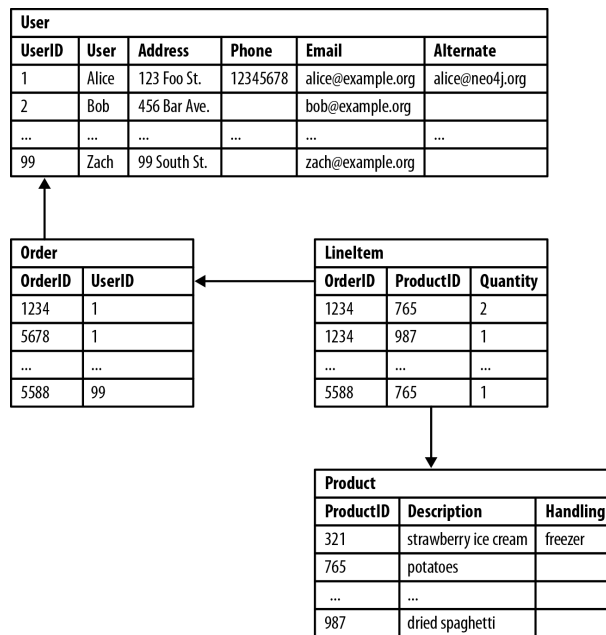
Grafové databázy využívajú model ktorý pozostáva z vrcholov, hrán, atribútov a značiek. Vrcholy obsahujú atribúty, a sú označené jednou alebo viacerými značkami. Tieto značky zoskupujú vrcholy, ktoré zastávajú rovnakú rolu v rámci aplikácie. Hrany v grafových databázach spájajú vrcholy a budujú štruktúru grafu. Hrana grafu má vždy smer, názov, východzí vrchol, cieľový vrchol a cieľový vrchol. Fakt, že hrany musia mať smer a názov pridáva to sémantickú prehľadnosť do grafu, ak zvolíme správne názov vieme rýchlo identifikovať štruktúru grafu a identifikovať význam vzťahov. Hrany môžu rovnako ako vrcholy obsahovať aj atribúty. Atribúty v hranách môžu byť praktické na pridanie kvalitatívnych dát (napr. váha, vzdialenosť) ku vzťahom, tieto dáta sa potom môžu použiť pri prehľadávaní grafu.

## 1.2 Výkon grafovej databázy pri dotazoch na vzťahy

Natívne grafové databázy používajú bezindexovú príslušnosť [2]. Absencia cudích kľúčov praxi znamená, že pri prehľadávaní vzťahov v je výpočtová zložitosť na  $O(1)$ . Táto rýchlosť je dosiahnutá tak, že všetky hrany sú uložené s priamymi ukazovateľmi na vrcholy, ktorých vzťah reprezentujú. Tak isto vo vrcholoch sú uložené priame ukazovatele na všetky hrany vychádzajúce z a mieriace do dotyčného vrcholu. Takáto štruktúra poskytuje už spomínanú výpočtovú zložitosť  $O(1)$  v oboch smeroch hrany, takže nielen v smere z východzieho bodu do cieľového ale aj opačným smerom. Pri relačnej databáze by toto muselo byť riešené reverzným vyhľadávaním v cudzích kľúčoch.

## 1.3 Vzťahy v relačných databázach

Relačná databáza je databáza, v ktorej sú údaje uložené podľa relačného databázového modelu[?] podľa E. F. Codd a z roku 1970. Relačný model je založený na matematickom aparáte relačných množín na predikátovej logiky. Databázová relácia sa od matematickej líši tým, že využíva pomocný aparát nazvaný schéma. V databáze schéma definuje názov relácie (tabuľky) a koľko obsahuje stĺpcová aký dátový typ obsahujú.



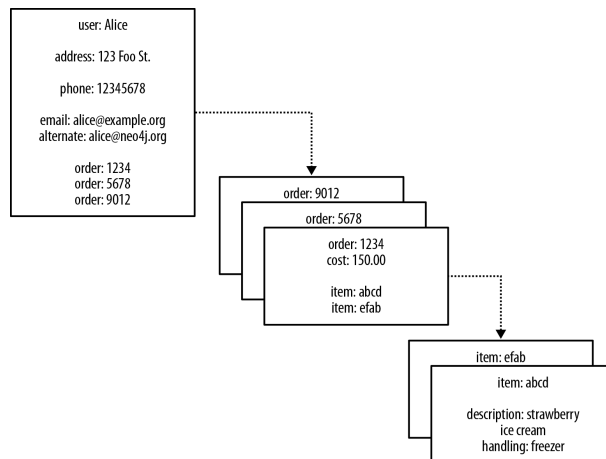
Obrázok 1: Ukážka riešenia vzťahov v relačnej databáze [2]

Vývojári databáz sa už desaťročia snažia relačné databázy prispôsobiť na to aby boli schopné efektívne pracovať s husto prepojenými dátovými sadami. Keďže však štruktúra relačných databáz bola pôvodne navrhnutá na ukladanie formulárov a tabuľkových štruktúr, majú problém s keď sa snažia poňať ad-hoc vzťahy, ktoré sa vyskytujú v dátach v reálnom svete. [2]

Vzťahy sa vyskytujú v modely relačných databáz no iba vo fáze návrhu, v reálnej implementácii to vyzerá tak, že keď pridávame vzťahy do relačnej databázy, veľmi rýchlo narastá komplexita a znižuje sa prehľadnosť dátového modelu a pridáva sa záťaž na databázu vznikom veľkých spájacích tabuliek a tabuliek s množstvom buniek s prázdnyimi cudzími kľúčmi. [2]

Pri vyhľadávaní každé prepojenie pridáva výpočtovú komplexitu. v každej ďalšej prepojenej tabuľke treba vyhľadať záznam s požadovaným kľúčom  $O(\log(n))$ . Používané relačné databázové systémy riešia tento výpočtový problém použitím indexov a inými optimalizáciami, no pokiaľ je dátová sada štruktúrovaná s veľkým množstvom vzťahov systém sa spravidla spomalí.

Na obrázku 1 vidíme prístup riešenia vzťahov v relačnej databáze.



Obrázok 2: Ukážka riešenia vzťahov v dokumentovej databáze  
[2]

## 1.4 Vzťahy v dokumentových databázach

Dokumentové databázy patria rovnako ako grafové do kategórie NoSQL databáz. V dokumentovej databáze sa údaje ukladajú ako štruktúrované objekty ktoré sú adresované pomocou unikátnych kľúčov, sémanticky vieme dokumenty rozlíšiť do kolekcií podľa ich významu v databáze. Štruktúra grafovej databázy je primárne určená na dopyty podľa kľúča a neposkytuje výhodu pri prehľadávaní cez vzťahy oproti relačným databázam. Rôzne implementácie dokumentových databáz implementujú odlišné prístupy k riešeniu vzťahov.

Bežný spôsob na pridanie vzťahov do štruktúry dokumentovej databázy je pridanie atribútu do dokumentu, ktorý bude odkazovať na kľúč a kolekciu dokumentu, s ktorým chceme vytvoriť vzťah. Týmto spôsobom vieme použiť koncept cudzích kľúčov z relačných databáz. Tento prístup sa však stretá s rovnakými problémami v rýchlosti ako pri relačných databázach s pridanou komplexitou v tom, že tento dopyt musíme často implementovať aplikačne.

Na obrázku 2 vidíme prístup riešenia vzťahov v dokumentovej databáze.

## 1.5 Výber grafovej databázy

Vybrali sme si troch najpopulárnejších predstaviteľov grafových databáz podľa rebríčka DB-Engines.com[1]. V nasledujúcej časti v skratke priblížime históriu každej grafovej databázy ich výhody, nevýhody a nevýhody.

### 1.5.1 Neo4J

Prvá verzia Neo4J bola vydaná v roku 2007 od vtedy sa stala dlhodobo najpoužívanejšou grafovou databázou. Je vyvíjaná Neo Technology, Inc. Neo4j je ponúkaná v dvoch variantoch: Neo4j Community je open source (GPLv3) grafová databáza obsahujúca všetky základné funkcie (Ďalej budeme spomínať len túto verziu). A Neo4j Enterprise edícia, ktorá má rozšírené funkcie ako shardovanie cache pamäte, rozšírené monitorovanie a zálohovanie za behu.

**Medzi hlavné výhody neo4j patrí:**

- Je dlhodobo najpopulárnejšia grafová databáza, podľa rebríčka db-rank[1]. Z toho vyplýva dobrá dokumentácia a vyššia podpora pre integráciu s populárnymi frameworkami.
- Dopytovací jazyk Cypher, ktorý táto databáza podporuje. Tento jazyk je vizuálne zrozumiteľný a navrhnutý primárne na prácu z grafovou databázou.

**Nevýhodami Neo4J sú:**

- Nižšia rýchlosť oproti OrientDB.
- Podporuje iba grafový model ukladania údajov.

### 1.5.2 OrientDB

OrientDB je vyvíjané od roku 2010 firmou OrientDB LTD. Databáza OrientDB rýchlo nabralo nabrala na popularite a v roku 2015 sa dostala na druhé miesto v rebríčku DB-engines. OrientDB sa rovnako ako Neo4j distribuuje v dvoch edíciách: Community - open source (Apache Licence 2.0) so základnými funkciami a Enterprise edíciou s podporou migrácie a synchronizácie na Neo4J a pridanými analytickými nástrojmi.

**Medzi hlavné výhody OrientDB patrí:**

- Podporuje okrem grafového modelu ukladania dát aj dokumentové a key-value úložisko
- Podľa nezávislých testov [?] je v niektorých testoch až desať krát rýchlejšie ako Neo4J.

**Nevýhodami OrientDB sú:**

- Menšia podpora pre populárne frameworky, konkrétne OGM pre Laravel nepodporuje ukladanie a čítanie atributov z hrán.
- Menej intuitívny dopytovací jazyk v kontexte grafových databáz.



### 1.5.3 Titan

Projekt Titan bol od 2012 vyvíjaný skupinou ThinkAurelius, no v roku 2017 bol odkúpený firmou Datstax a projekt Titan bol zastavený. Projekt je ďalej udržiavaný ako open source verzia pod menom JanusGraph. Titan je projekt určený na veľké distribuovane enterprise riešenia, je nasadzovaný na cloudové platformy ako napr. Apache Hadoop a Apache Spark podporuje rôzne distribuované úložné priestory ako napr. Apache HBase a Oracle BerkeleyDB. Ďalej podporuje rôzne vyhľadávacie enginy ako napr. Elasticsearch a Solr.

**Medzi hlavné výhody databázy Titan patrí:**

- Podpora rôznych zásuvných modulov vyžívaných v enterprise riešeniach.

**Nevýhodami databázy Titan sú:**

- Ukončenie vývoja po akvizícii firmou DataStax.
- Jej určenie na veľké distribuované systémy.

V procese výberu databázy sme prihliadali na rôzne faktory. Posudzovali sme kompatibilitu, jednoduchosť použitia, rýchlosť a iné. Na koniec sme sa rozhodli pre Neo4J práve pre jeho rozšírenosť a použitie jazyka Cypher. Aj keď podľa rôznych nezávislých testov je OrientDB rýchlejšie ako Neo4J v kontexte porovnávania relačných a grafových databáz, tento fakt nezohral veľkú rolu, lebo obe databázy sú pri traverzovaní grafu rýchlejšie ako relačné. Databázu Titan sme diskvalifikovali z výberu z dôvodu ukončenia jej vývoja a určenia primárne na distribuované systémy.

## 1.6 Dopytovací jazyk Cypher

Dopytovací jazyk Cypher bol vyvinutý firmou NeoTechnology pre ich databázu Neo4J a prípadnú štandardizáciu tohto jazyka. Cypher bol navrhnutý tak, aby sa ľahko čítal a pochopil, preto používa intuitívnu "ASCII Art" notáciu. Cypher používa oblé zátvorky na reprezentáciu vrcholov, hranaté zátvorky a pomlčky na reprezentáciu hrán a znaky väčší a menší na upresnenie smerovania hrany. Medzi hlavný dopytovací príkaz v Cypher je MATCH, týmto príkazom podobne ako SELECT v SQL začína každý dopyt.

V nasledujúcom príklade 1 vidíme zopár príkladov dopytov. Na prvom príklade vidíme dopyt, ktorý vyhľadávame vrchol n, ktorý má značku Osoba a meno Alenka, z tohto vrcholu vychádzajúce (všimnime si smerovanie "ASCII ART šípky") hrany so značkou POZNA, ktoré vedú k cieľovým vrcholom so značkou Osoba. Všimnime si, že v zátvorkách reprezentujúcich vrcholy a hrany priradujeme premenným n, m a r kolekcie vrcholov a

hrán. Dedže je v dopyte "RETURN r,m" vrátia sa a vrcholy v týchto kolekciách, čiže nie pôvodný vrchol Alenka. Následujú klasické príkazy SQL pre zoradenie a obmedzenie počtu výsledkov.

Na druhom príklade vidíme dopyt, ktorý vráti všetky hrany s ľubovoľnými značkami ktoré sa nachádzajú medzi vrcholom Barbora a vrcholmi, ktorých atribút meno začína na M.

Na treťom príklade vidíme dopyt, ktorý vráti všetky vrcholy, ktoré sú vzdialené od vrchola s menom Lucia dve hrany typu POZNA

Na príklade číslo štyri je dopyt, ktorý vytvára vrchol a typu Osoba s atribútom meno a typ

Na poslednom príklade vidíme príkaz, ktorý najprv vyhľadá vrcholy s menom Patrik a Adam pridá hranu smerujúcu od Adama k patrikovi s atribútom váha.

---

**Algoritmus 1** Príklady dopytov v jazyku Cypher

---

```
1
2 1)
3 MATCH (n:Osoba)-[r:POZNA]->(m:Osoba)
4 WHERE n.meno = 'Alenka' RETURN r,m ORDER BY m.age LIMIT 42
5
6 2)
7 MATCH (n:Osoba)-[r]-(m:Osoba)
8 WHERE n.meno = 'Barbora' AND m.meno =~ "M.*" RETURN r
9
10 3)
11 MATCH (n:Osoba)-[:POZNA]->(:Osoba)-[:POZNA]->(m:person)
12 WHERE n.meno = 'Lucia' RETURN m
13
14 4)
15 CREATE (n:Osoba { meno: 'Jozef', typ: 'Absolvent' })
16
17 5)
18 MATCH (a:Osoba),(b:Osoba)
19 WHERE a.meno = 'Adam' AND b.meno = 'Patrik'
20 CREATE (a)-[r:POZNA { váha: 7 }]->(b)
21 RETURN r
```

---

## 2 Špecifikácia

Aplikácia ktorú sme sa rozhodli implementovať bude slúžiť na plánovanie a optimalizáciu trasy cestovateľa po svete. Bude umožňovať používateľovi sledovať ceny cesty z domáceho miesta, ktoré si používateľ zadá. Do destinácií, ktoré si pridá na zoznam obľúbených destinácií. Bude ďalej používateľovi ukazovať ostatných používateľov, ktorí si zvolili rovnaké destinácie, a bude vedieť používateľovi odporučiť ďalšie destinácie na základe zhody s ostatnými používateľmi.

### 2.1 Funkcionálne požiadavky

1. Aplikácia bude umožňovať registráciu a prihlásenie používateľa
2. Pri registrácii sa budú vyžadovať prihlasovacie údaje: e-mail, heslo. Okrem toho sa bude vyžadovať zdanie domácej destinácie.
3. Po prihlásení používateľa sa mu zobrazí obrazovka s mapou, zoznamom obľúbených destinácií, ktoré chce navštíviť a zoznam odporúčaných destinácií.
4. Na mape bude vyobrazená používateľova domáca destinácia a všetky destinácie ktoré má v zozname obľúbených destinácií.
5. V zozname obľúbených budú všetky destinácie, ktoré si používateľ pridal. Zoznam bude vo forme tabuľky ktorej riadok bude obsahovať meno destinácie a cenu najlacnejšej trasy z domáceho miesta do destinácie nachádza.
6. Budú sa dať zobrazíť informácie o všetkých dostupných trasách ku konkrétnej destinácii s ich cenami a spôsobmi dopravy.
7. V detailoch obľúbenej destinácie bude zoznam ostatných používateľov ktorí danú destináciu majú tiež na zozname obľúbených
8. V zozname odporúčaných destinácií budú destinácie ktoré majú na zozname ľudia s ktorými má prihlásený používateľ najviac spoločných destinácií.
9. V aplikácii bude obrazovka kde si bude používateľ vybrať niekoľko zo svojich obľúbených destinácií a nechať si vyrátať optimálnu trasu z domáceho miesta cez všetky zvolené miesta a potom späť. (TSP)

## 2.2 Nefunkcionálne požiadavky

1. Systém bude zrealizovaný na webovej platforme.
2. Aplikácia bude využívať natívnu grafovú databázu.
3. Aplikácia bude byť kompatibilná s webovými prehliadačmi Google Chrome, Mozilla Firefox, Microsoft Edge.
4. Užívateľské rozhranie systém musí byť plne použiteľné aj na mobilných telefónoch s OS Android a IOS.
5. Aplikácia bude implementovaný s použitím jazyka PHP a PHP frameworku.
6. Systém bude nasadený na virtuálnom serveri s operačným systémom Ubuntu 16.04.2 LTS poskytnutom Ústavom informatiky a matematiky FEI STU.

## 3 Návrh

V tejto kapitole si priblížime proces návrhu našej aplikácie, popíšeme prípady použitia ktoré pre našu aplikáciu očakávame. Ďalej priblížime návrh štruktúry databázy, ktorú plánujeme použiť.

### 3.0.1 Prípady použitia

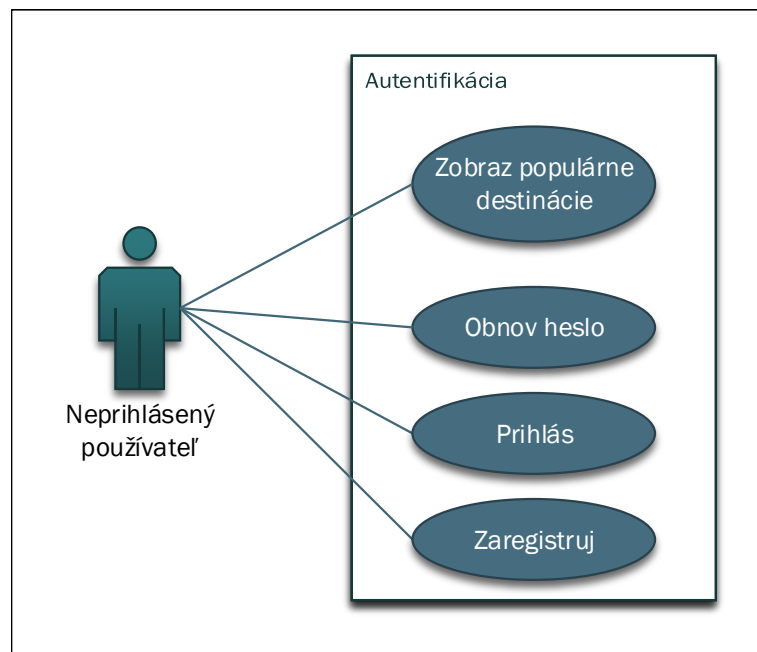
Prípady použitia popisujú interakcie medzi aplikáciou a používateľom, popisujú roly, ktoré rôzni aktéri hrajú v týchto interakciách.

1. V prvom prípade použitia 3 si bude môcť neprihlásený používateľ prezerat mapu a zoznam najpopulárnejších destinácií v našej aplikácii, ďalej sa môže prihlásiť zaregistrovať a požiadať o obnovu hesla.
2. V druhom prípade použitia 4 opisujeme nastavenia. Používateľ si bude môcť vyhľadať destináciu a zvoliť si ju ako svoje domáce miesto alebo si zmeniť heslo.
3. Tretí prípad použitia 5 opisuje akcie ktoré bude môcť používateľ vykonávať na hlavnej stránke. Bude môcť vyhľadávať destinácie a pridávať si ich do obľúbených. Destinácie ktoré má v obľúbených bude môcť mazať a zobrazíť detaily trasy, ktorá k destinácií vedie. Ďalej si bude môcť prezerat ostatných používateľov ktorý majú rovnaké miesto v obľúbených a prejsť na zoznam obľúbených destinácií jedného s týchto používateľov.
4. Posledný prípad použitia 6 popisuje obrazovku TSP. Na tejto obrazovke bude môcť používateľ prezerat svoju mapu optimálnej cestovateľskej trasy, pridávať a odoberať z trasy destinácie zo zoznamu obľúbených.

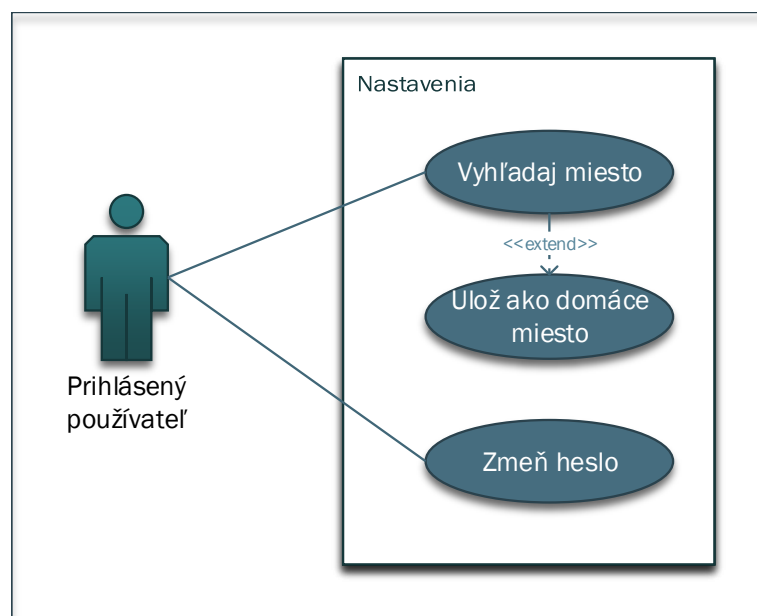
### 3.0.2 Štruktúra databázy

V našej databáze sa budú nachádzať dve hlavné entity: entita používateľa *appUser* a entita destinácie *appPlace*. Entita používateľa bude obsahovať iba základné atribúty ako meno a atribúty potrebné pre autentifikáciu používateľa email, a heslo. Entita destinácie bude obsahovať atribúty potrebné na identifikáciu tejto destinácie: unikátny názov a krátky názov vhodný na zobrazovanie a atribúty potrebné na lokalizáciu destinácie: zemepisná šírka a zemepisná dĺžka. Keďže použijeme grafovú databázu nebude potrebné pridávať cudzie kľúče reprezentujúce ich vzťahy.

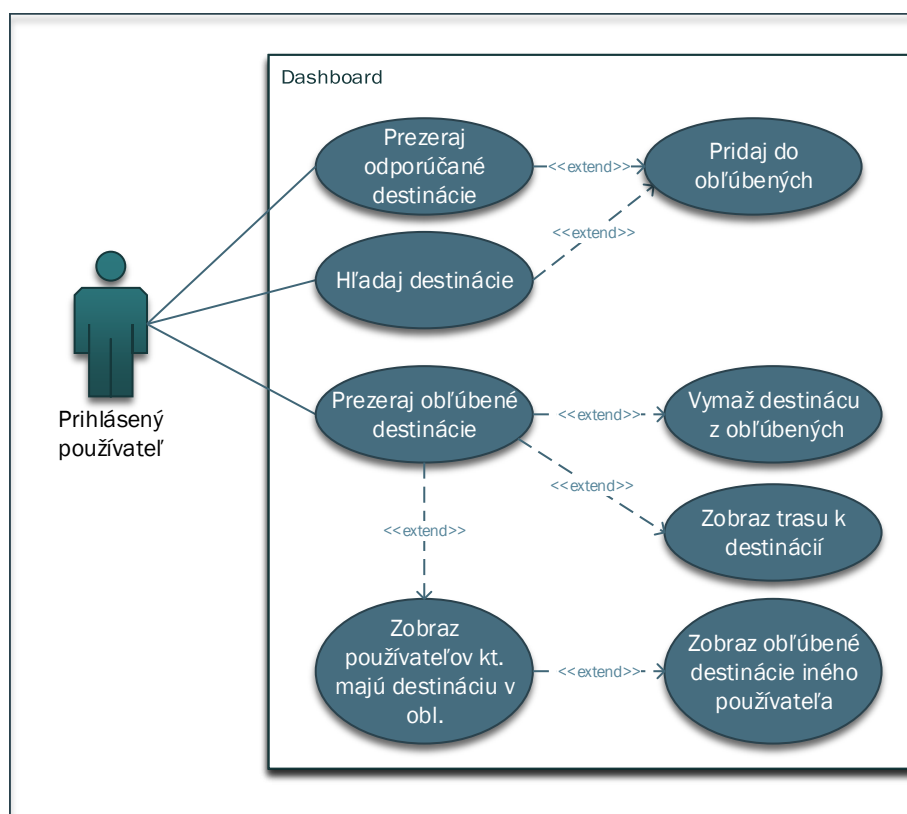
Ďalej bude naša databáza obsahovať hrany: trasa *ROUTES*, obľúbené *FOLLOWS*, a zoznam pre optimalizáciu trasy *TSP*. Hrana trasy bude vždy viesť od jedného miesta k druhému reprezentuje zoznam trás ktoré budú viesť od jednej destinácie k druhej.



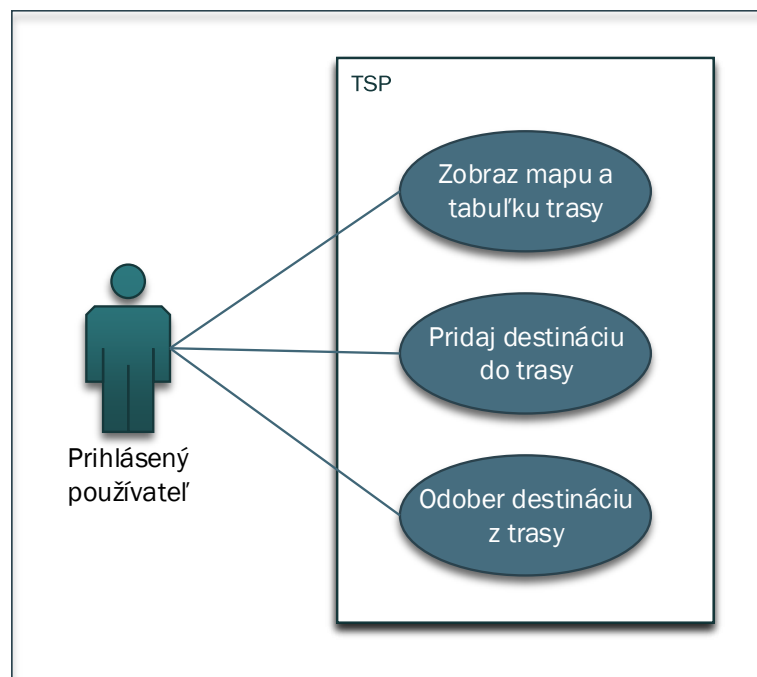
Obrázok 3: Prípad použitia - Registrácia, autentifikácia a nastavenia



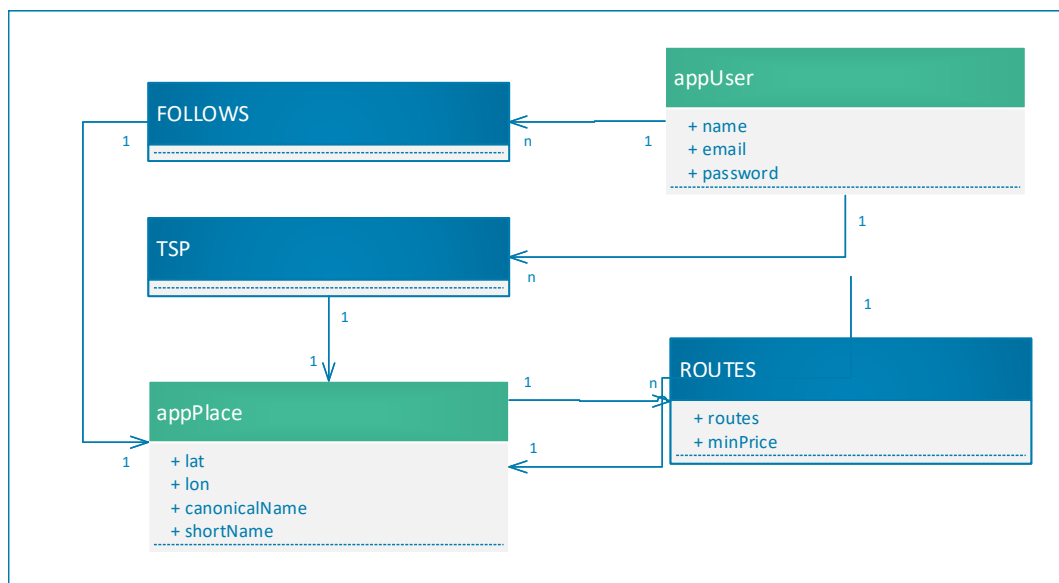
Obrázok 4: Prípad použitia - Registrácia, autentifikácia a nastavenia



Obrázok 5: Prípad použitia - dashboard



Obrázok 6: Prípad použitia - tsp

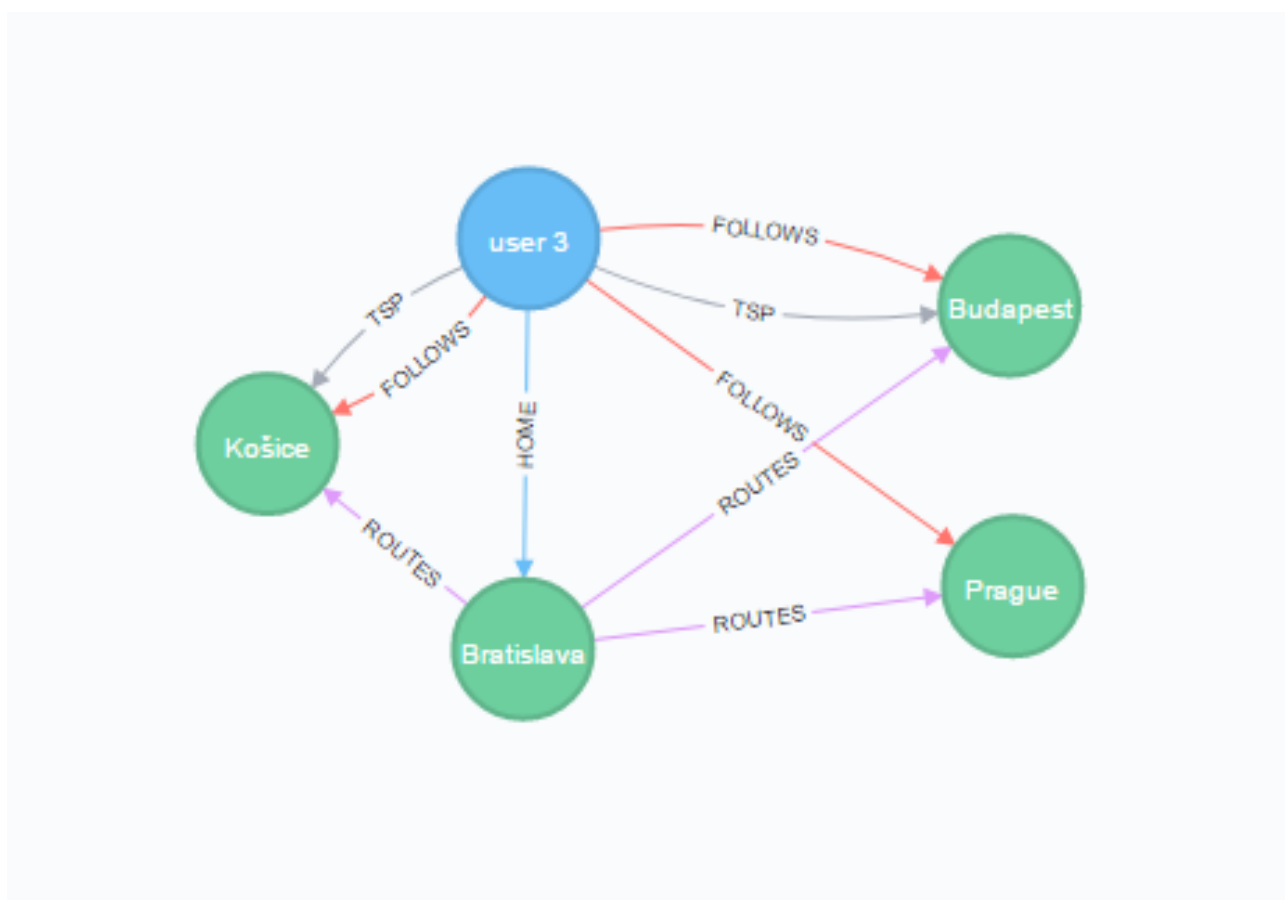


Obrázok 7: Štruktúra databázy

Bude obsahovať atribút *minPrice* hovoriaci o cene najlacnejšej trasy medzi spomínanými destináciami a atribút *routes* obsahujúci serializované detaily všetkých trás. Hrany *FOLLOWS* a *TSP* budú vždy smerovať od používateľa k destináciám a nebudú obsahovať žiadne atribúty.

Na obrázku č. 7 vidíme UML diagram štruktúry našej databázy. Na obrázku č. 8 vidíme reprezentáciu dát uložených v našej databáze. Používateľ *user 3* má ako domáce miesto (HOME) nastavené mesto *Bratislava*, v obľúbených destináciách (FOLLOWS) má mestá *Praha*, *Košice* a *Budapešť*, systém pridal hranu *ROUTES* smerom od domáceho miesta používateľa k cieľovým miestam. Používateľ má tiež pridané mestá *Budapešť* a *Košice* na zozname optimalizácie takže sme pridalí hranu *TSP*.





Obrázok 8: Príklad dát v databáze

## 4 Implementácia

V tejto kapitole si priblížime proces implementácie a evolúcie našej aplikácie. Najprv si opíšeme niektoré z použitých technológií, potom

### 4.1 Použité technológie

v tejto kapitole si priblížime niektoré z technológií ktoré sme využili na implementáciu našej aplikácie. V skratke opíšeme históriu technológie, jej využitie a licenciu pod ktorou je vydávaná.

#### 4.1.1 Laravel framework

Laravel je open source framework pre aplikácie v jazyku PHP. Laravel je od roku 2011 vyvíjaný Taylorom Otwellom ako framework určený na stavbu webových aplikácií s vývojovým vzorom MVC. Niektoré funkcie sú implementované ako zásuvné moduly, ktoré sa inštalujú pomocou vlastného správcu závislostí. Laravel je orientovaný na to, aby vývoj v ňom bol syntakticky čo najprehľadnejší a najjednoduchší. Toto je jeden z dôvodov prečo patrí od roku 2015 k najpopulárnejším PHP frameworkom. Kód frameworku je voľne dostupný pod licenciou MIT.

#### 4.1.2 NeoEloquent OGM

NeoEloquent OGM je voľne šíriteľná knižnica, ktorá umožňuje využívať grafovú databázu neo4j spolu s existujúcim dátovým modelom vo frameworku Laravel. Štruktúra NeoEloquent je modelovaná podľa natívneho Eloquent Modelu Laravel. Knižnica je vydávaná od roku 2014 pod licenciou MIT spoločnosťou Vinelab

#### 4.1.3 GraphAware Neo4j PHP Client

Neo4j PHP Client je klientská knižnica pre databázu Neo4J. Na rozdiel od OGM je táto schopná vykonávať programátorky zadané dopyty, poskytuje tým pádom prístup k plnému rozsahu Cypher API databázy Neo4j. Knižnica je vyvíjaná partnerskou firmou Neo Technology, firmou GraphAware od roku 2014. Je vydávaná pod licenciou MIT.

#### 4.1.4 Mapbox.js

Mapbox.js je knižnica na vytváranie projektov s interaktívnymi mapami. Je založená na voľne šíriteľnej knižnici Leaflet vyvíjanou Vladimírom Agafonkinom, rozširuje túto knižnicu o funkcie ako automatické zoskupovanie bodov do skupín a poskytuje prehľadnú dokumentáciu. My používame verziu zdarma, ktorá je obmedzená na 50000 zobrazení mapy na mesiac.

Túto knižnicu postupne nahrádza Mapbox-gl.js vyvíjaná tou istou firmou, ktorá

využíva technológiu WebGL na lepšiu akceleráciu vykresľovania, no v čase začiatku projektu táto knižnica ešte nepodporovala vyššie spomenuté zoskupovanie bodov.

#### 4.1.5 Handlebars.js

Handlebars.js je jednoduchá silná open source knižnica na prácu z šablónami, vyvíjaná je Yehudom Katzom a komunitou na GitHube od roku 2010 a vydávaná pod licenciou MIT.

#### 4.1.6 Rome2Rio Api

Rome2Rio je portál ktorý zbiera údaje o cenách dopravy po celom svete a umožňuje vyhľadať cenu cesty medzi dvomi ľubovoľnými destináciami. Rome2Rio taktiež poskytuje niekoľko otvorených a platených API. My využívame dve z nich: Search API a Autocomplete API. Autocomplete API slúži na vyhľadávanie destinácií z databázy rome2rio a nie je obmedzené na počet volaní. Search API slúži na vyhľadávanie trás medzi jednotlivými destináciami, je potrebné sa identifikovať API kľúčom a je obmedzené na 100 000 volaní za mesiac.

### 4.2 Inštalácia a konfigurácia Laravel Framework-u

Na inštaláciu frameworku Laravel sme použili nástroj pre správu PHP balíkov *Composer*. Pomocou tohto nástroja sme nainštalovali balík *laravel/installer* [5], následne sme požitím príkazu *laravel new projekt* vytvorili priečinok so základnou inštaláciou frameworku.

Jediné nastavenie ktoré bolo po tom potrebné bolo už len nastavenie názvu aplikácie, a databázy, o tom ďalej v sekcii 4.3.

### 4.3 Inštalácia a konfigurácia databázy neo4j

[4] Aby sme mohli nainštalovať databázu Neo4j musíme si najprv do systému pridať repozitár Neo Technology, potom je nám k dispozícii na inštaláciu balík neo4j. Po inštalácii je nám ihneď dostupné administračné rozhranie databázy na adrese: *localhost:7474*. Pri prvom prihasení sme vyzvaní na zmenu hesla.

Keďže základná inštalácia frameworku Laravel neobsahuje ovládač pre databázu neo4j museli sme použiť ovládač integrovaný v balíku NeoEloquent, to sa registráciou poskytovateľa služby. Po zaregistrovaní služby NeoEloquentServiceProvider sa automaticky zaregistruje ovládač pre databázu a pridajú sa nové možnosti pre konfiguráciu databázy. Následne stačí vykonať štandardnú konfiguráciu mena hostiteľa, port a prístupových údajov.

Na získanie vzdialeného prístupu k administračnému rozhraniu databázy bez otvorenia portu 7474 verejnosti využívame SSH tunel. Administračné rozhranie používa okrem portu 7474 ešte port 7687 lebo na komunikáciu s databázou využíva technológiu Web-

Socket.

---

**Algoritmus 2** Inštalácia a konfigurácia Frameworku Laravel pre prácu z Neo4J

---

```
1
2 #inštalácia databázy neo4j verzie 3.1.4
3 wget -O - https://debian.neo4j.org/neotechnology.gpg.key | sudo apt-key ad
4 echo 'deb https://debian.neo4j.org/repo stable/' | sudo tee /etc/apt/sourc
5 sudo apt-get update
6 sudo apt-get install neo4j=3.1.4
7
8 ...
9
10 //registrácia neoEloquent sevice rovider
11 $app->register('Vinelab\NeoEloquent\NeoEloquentServiceProvider');
12
13 ...
14
15 //nastvenie prístupových údajov do databázy
16 'default' => env('DB_CONNECTION', 'neo4j'),
17
18 'connections' => [
19 'neo4j' => [
20 'driver' => 'neo4j',
21 'host' => env('DB_HOST', 'neo4j'),
22 'port' => env('DB_PORT', 'neo4j'),
23 'username' => env('DB_USERNAME', 'neo4j'),
24 'password' => env('DB_PASSWORD', 'neo4j'),
25 ],
26 ],
```

---

## 4.4 OGM

Keďže framework Laravel natívne obsahuje len ovládače pre relačné databázové ovládače a nástroj na objektovo relačné mapovanie Eloquent Použili sme open source knižnicu NeoEloquent, ktorý obsahuje ovládač pre databázu Neo4J a zároveň rozširuje dátový model o prvky grafovej databázy. NeoEloquent umožňuje manipuláciu s vrcholmi aj hranami v Neo4J. Manipulácia s vrcholmi je rovnaká ako s entitami v relačnej databáze,

NeoEloquent umožňuje vytvárať perzistentné objekty, upravovať ich a vyhľadávať v nich. Pri práci s hranami je mierne odlišná. Najprv treba zadať ktorý objekt môže mať aké vzťahy, tieto vzťahy treba unikátne identifikovať ich značkou a kardinalitou. Tento vzťah vraciame ako návratovú hodnotu funkcie daného objektu. Vrátený objekt sa správa podobne ako štandardná entitná trieda Eloquent.

Knižnicu sme nainštalovali pomocou správcu balíkov *composer* pomocou príkazu *composer require vinelab/neoeloquent*.

V nasledujúcom príklade 4 vidíme implementáciu dvoch rôznych entitných tried. Trieda používateľa dedí od triedy NeoEloquent a teda sa stáva naviazanou na vrchol v našej databáze. Názov tejto entity v databáze je spojením menového priestoru v ktorom bol vytvorený a názvu triedy, takže v našom prípade *AppUser*. Trieda obsahuje verejné funkcie, ktoré vracajú objekty hrán. Objekty hrán dostávame volaním zdedených funkcií *hasMany*, *hasOne* a *belongsToMany*. Ako prvý argument funkcie berú názov triedy ktorou vzťah chceme vrátiť, ako druhý argument berie typ hrany, pomocou tohto typu je identifikovaný typ hrany v databáze.

Po spustení aplikácie a zadaní prvých dát sa v databáze vytvorili vrcholy a hrany podľa modelu ktorý sme si zadefinovali v kapitole 3.0.2.

Trieda NeoEloquent funguje vo väčšine prípadov presne ako Eloquent no v jednom prípade sme mali problém z CSRF tokenmi. CSRF token je bezpečnostný prvok, ktorý ochraňuje webovú stránku pred útokom falšovania požiadaviek z inej adresy. Laravel má tento bezpečnostný prvok vstavaný v sebe, je to 40 znakový reťazec, ktorý sa generuje každému používateľovi pri zobrazení formulára, tento istý reťazec sa zároveň uloží do databázy a keď Laravel prijme formulár overí či sa jeho token nachádza v databáze. Táto funkcionálnosť však po prejdení na grafovú databázu nefungovala. Pri každom odoslaní formulára vyhlasovalo nezhodu CSRF tokenu a v databáze sa neobjavila entita ktorá by tieto tokeny mohla obsahovať. Tento problém sme zatiaľ obišli deaktiváciou tohto bezpečnostného prvku.

Další problém na ktorý sme narazili počas implementácie bol problém kompatibility knižnice NeoEloquent verzie 1.4 s databázou Neo4J vo ve verzii 5.2. Pri ukladaní nového vrcholu vždy nastala chyba. Bolo potrebné znížiť verziu databázy.

## 4.5 Autentifikácia

Jednou zo silných stránok Frameworku Laravel je práve autentifikácia. Pre vytvorenie základnej funkcionality registrácie, prihlasovania a obnovenia zabudnutého hesla stačí použiť Artisan - konzolu frameworku príkaz *php artisan make:auth*, ktorá vytvorí URL

---

**Algoritmus 3** Ukážka triedy neoEloquent

---

```
1 namespace App;
2
3 class User extends \NeoEloquent implements Authenticatable {
4
5     // Jeden používateľ môže mať v obľúbených viac miest
6     public function follows() {
7         return $this->hasMany( 'App\Place', 'FOLLOWS' );
8     }
9
10    // Jeden používateľ má jedno miesto ako domáce
11    public function home() {
12        return $this->hasOne( 'App\Place', 'HOME' );
13    }
14    ...
15 }
16
17 ...
18 class Place extends \NeoEloquent {
19     // Inverzný vzťah – jedno miesto má v obľúbených viac používateľov
20     public function followers(){
21         return $this->belongsToMany( 'App\User', 'FOLLOWS' );
22     }
23     ...
24 }
```

---

cesty, obrazovky, triedu používateľa a triedy obsluhujúce túto funkcionálnosť. My sme potrebovali použiť vlastnú triedu používateľa, ktorá dedí od nášho balíka NeoEloquent, na implementáciu autentifikácie stačilo implementovať rozhranie Authenticatable, pridať do triedy používateľa pole skrytých a verejných atribútov, a použiť charakteristiku 'AuthenticableTrait' a autentifikácia fungovala rovnako ako s relačnou databázou vďaka tomu, že NeoEloquent implementuje podobnú funkcionálnosť ako natívne ORM Eloquent.

---

**Algoritmus 4** Ukážka autentifikovateľnej triedy

---

```
1
2 namespace App;
3
4 use Illuminate\Contracts\Auth\Authenticatable;
5 use Illuminate\Auth\Authenticatable as AuthenticableTrait;
6
7 class User extends \NeoEloquent implements Authenticatable {
8
9     use AuthenticableTrait;
10
11     // pole verejných atribútov
12     protected $fillable = [
13         'name', 'email', 'password', 'tspCache'
14     ];
15
16     // pole skrytých atribútov
17     protected $hidden = [
18         'password', 'remember_token',
19     ];
20     ...
21 }
```

---

## 4.6 Databázové dopyty pomocou OGM

Keďže sme sa rozhodli využiť javascriptové zobrazovanie dynamického obsahu museli sme vytvoriť interné API, aby sme mohli javascriptu poskytnúť údaje. Na tento účel sme vytvorili niekoľko ciest pomocou routovacieho nástroja frameworku. Router Laravel-u je relatívne jednoduchý, ale silný nástroj na vytváranie RESTful API. Na naše potreby sme potrebovali vytvoriť tri REST cesty: */place*, */placeapi* a */tsp*. Prvá je na pridávanie des-

tinácií, druhá na pridávanie, odoberanie obľúbených a zobrazovanie obľúbených destinácií, tretia na pridávanie odoberanie a zobrazovanie destinácií na zozname TSP.

API pre získanie dát na zobrazenie zoznamu obľúbených destinácií sme v prvej verzii naimplementovali pomocou OGM. Na príklade algoritmu č. 5 vidíme pseudokód tohto prístupu. Naprv naša PHP funkcia urobila dopyt na všetky miesta, ktoré má používateľ v obľúbených potom v cykle prešla všetky výsledky a pre každú destináciu poslala dopyt na trasu medzi domácim miestom na touto destináciu. Ďalej poslala dopyt na získanie všetkých používateľov ktorý majú destináciu tiež v obľúbených.

Pri testovaní rýchlosti sme zistili, že pri takomto prístupe priemerná latencia nášho API vzrástla až na 1,857 sekundy pri 200 položkách na zozname. Tento nárast latencie pripisujeme režií ktoru si vyžaduje diskkrétne volaní funkcií OGM v každom cykle. Ďalej sa testovaniu rýchlosti venujeme v kapitole ??.

---

**Algoritmus 5** Pseudokód algoritmu na získavanie dát pre zoznam obľúbených destinácií pomocou OGM

---

```
1  používateľ = dopytPoužívateľOGM ()
2  destinácie = používateľ->dopytOblubeneOGM ()
3  domov = používateľ->dopytdomovOGM ()
4
5  výsledok = []
6  FOR každú destinácia z destinácie
7      trasa = domov->dopytTrasaOGM ( destinácia )
8      ludia = destinácia->dopytLudiaOGM ()
9
10     výsledok << destinácia , trasa , ludia
11  ENDFOR
```

---

## 4.7 Databázové dopyty pomocou Cypher klienta

Aby sme sa vyhli problému z kapitoly 4.6 rozhodli sme sa implementovať dopyty na vaše dátové sady pomocou jedného dopytu v jazyku Cypher. Na vykonanie priameho dopytu sme použili knižnicu *PHP Neo4J Client*. Táto knižnica vykonáva dopyty v jazyku Cypher a teda nám poskytuje priamy prístup k Cypher API databázy. Na príklade 6 je dotaz ktorý nahradzuje všetkú funkcionálnu kódu z príkladu 5. Dopyt pozostáva vykoná nasledovné úkony (po riadkoch):

1. Získa vrchol typu AppUser s ID = 1 a uložení ho do premennej user, ďalej získa



všetkých vrcholov ku ktorým z vrcholu user vedie hrana typu FOLLOWS a uloží ich do premennej dest.

2. Získa vrchol ku ktorému vedie hrana typu home od vrcholu user a uloží ho do premennej home.
3. Získa všetky hrany, ktoré vedú od vrcholu home do jednotlivých vrcholov v premennej dest.
4. Získa všetky vrcholy od ktorých vedie hrana typu FOLLOWS k všetkým vrcholom v premennej dest, keďže aj vrchol user má hranu FOLLOWS do každého vrcholu v dest, vynechá vrchol user z výberu.
5. Definuje formát výstupu.

Po vykonaní dopytu sa nám vráti výstup vo forme tabuľky obdobnej ako pri vykonaní JOIN príkazu v relačnej databáze. Keďže sme na jeden dopyt získali všetky informácie, ktoré potrebujeme na zobrazenie tabuľky nepotrebujeme vykonávať už žiadne ďalšie dopyty. Tento prístup znížil priemernú latenciu nášho API na 37% (695,7ms) oproti prístupu pomocou OGM z kapitoly 4.6

---

**Algoritmus 6** Dopyt na dáta pre zoznam obľúbených destinácií v jayku Cypher

---

```
1 MATCH (user:AppUser) -[:FOLLOWS]->(dest) WHERE id(user)=1
2 MATCH (user) -[:HOME]->(home)
3 MATCH (home) -[:route:ROUTES]->(dest)
4 MATCH (follower:AppUser) -[:FOLLOWS]->(dest) WHERE follower <> user
5 RETURN dest, route, follower ;
```

---

## 4.8 TSP

V aplikácii sa nachádza obrazovka, na ktorej si môže používateľ vybrať destinácie, pre ktoré by chcel vyrátať trasu optimálnu vzhľadom na cenu. Táto trasa bude začínať v domovskej lokácii používateľa, prejde všetkými vybranými destináciami práve raz a vráti sa späť do domácej lokácie. Toto je klasický prípad NP-kompletný problému pocestného obchodníka.

Pre jednoduchosť a spoľahlivosť sme naimplementovali prístup rekurzívneho prehľadania všetkých možností trás. Naimplementovali sme rekurzívnu funkciu v PHP, ktorá prejde všetkými možnými trasami medzi našimi destináciami a vyberie prechod grafom,

ktorý je optimálny vzhľadom na cenu celkovej trasy. Do argumentu naša funkcia berie východzie miesto, zoznam destinácií ktoré ešte neprehľadala a domáce miesto do ktorého sa má nakoniec vrátiť. V každom vnorení sa zoznam neprehľadaných vrcholov znižuje. Vyššie spomenuté argumenty sú perzistentné objekty resp. polia inšancií perzistentnej triedy *Place*.

Keď sa táto funkcia dopytuje na cenu trasy vedúcej z východzieho vrcholu do cieľového zavolá ktorá dá dopyt na našu databázu, ak sa táto trasa nenachádza v databáze vypýta si údaje o konkrétnej trase od API Rome2Rio. Keďže každý dopyt na API trvá cca 200ms prvé volanie pri väčšej trase trvá niekoľko násobne dlhšie ako každé nasledujúce.

Aj po načítaní všetkých potrebných trás však bolo naše riešenie príliš pomalé, pri siedmych miestach už trvalo rátanie TSP až 7 minút. Toto bolo spôsobené tým, že naša funkcia, ktorá počíta s výpočtovou komplexitou  $O(n!)$  beží na vrstve PHP a pri každom volaní funkcie invokes knižnicu NeoEloquent a dopytuje sa na hranu medzi dvoma destináciami. Keď funkcia v takomto režime bežala videli sme, že proces neo4j konzistentne bral cca 30% z procesorového času. Z tohto dôvodu sme naimplementovali cache na dopyt k databáze a priemerný čas sa znížil na 80 sekúnd.

Naša implementácia front endu vyžaduje dva dopyty na náš server, jeden na zobrazenie mapy a druhý na zobrazenie tabuľky s výsledkami, z tohto dôvodu sme pridali ešte jeden level cache na celé volanie funkcie pre rátanie TSP. Taktiež sme upravili implementáciu front endu tak, aby sa tieto dva dopyty vykonali vždy synchronne za sebou. Týmto sme znížili nároky na výpočtové kapacity nášho serveru.

Dlhý čas výpočtu pri dopytovaní sa na databázu pripisujeme tomu, že naše dopyty na databázu boli diskkrétne pri každom vnorení funkcie. Aj keď sme používali perzistentné objekty implementácia každého dopytu NeoEloquent pozostáva z jedného dopytu v jazyku Cypher na databázu toto volanie obsahuje vyhľadanie začiatočného a konečného bodu podľa id  $O(\log(n))$  a následne vyhľadanie hrany  $O(1)$ . Tento prístup nevyužíva hlavnú výhodu grafovej databázy, no s použitím NeoEloquent nie je možné postaviť komplexný dopyt, ktorý by vedel vyriešiť TSP za jeden beh. Takýto dopyt je v Neo4J možné implementovať pomocou dopytovacieho jazyka Cypher príklad takéhoto dopytu môžeme vidieť na ukážke 7. Aj tento dopyt však iba permutuje všetky riešenia a vráti optimálne a preto v je v reálnom prípade vhodnejšie použiť známe algoritmy, v programovej vrstve.

## 4.9 Rome2rio API

Všetky údaje o destináciách a trasách berieme z API Rome2Rio. Pomocou Autocomplete API umožňujeme používateľovi pridávať destinácie. Používateľ zadáva písmená do

---

**Algoritmus 7** Ukážka riešenia TSP pomocou dopytovacieho jazyka Cypher

---

```
1      MATCH (from:Node {name: "Source node" })
2      MATCH path = (from) -[:CONNECTED_TO*6]->()
3      WHERE ALL(n in nodes(path) WHERE 1 = length(filter(m in nodes(path)
4      AND length(nodes(path)) = 7
5      RETURN path ,
6      reduce(distance = 0, edge in relationships(path) | distance + edge
7      AS totalDistance
8      ORDER BY totalDistance ASC
9      LIMIT 1
```

---

autocomplete textového poľa na stránke toto pole posiela dotazy na Autocomplete API a ono vracia pole objektov s miestami. Tieto objekty obsahujú informácie o type destinácie(obec, mesto, región, štát, letisko,), geografickú polohu, názov v dlhom a krátkom tvare a kanonický názov. Používateľ si potom vyberie jenu z destinácií a príslušný objekt sa zašle na náš server. Na unikátnu identifikáciu objektu používame kanonický názov, ktoré je podľa dokumentácie unikátnym identifikátorom miesta.

Ak destináciu ešte nemáme v databáze, pridáme tento objekt do databázy. Toto riešenie nie je úplne ideálne z bezpečnostného hľadiska, lebo umožňuje zaslanie falošného miesta do našej databázy. AK by útočník vyrobil objekt s reálnym kanonickým názvom no falošnými údajmi napr. o zemepisnej šírke, dĺžke toto miesto by sa potom nesprávne zobrazovalo všetkým používateľom. Na vyriešenie tohto problému by postačilo urobiť ešte jeden dotaz z nášho servera na autocomplete API ktorým by sme si len vypýtali údaje k miestu za pomoci kanonického názvu.

Ďalšie údaje berieme z Rome2Rio Search API. Sú to údaje o možných trasách a ich cenách. Toto API je obmedzené na počet volaní preto sme na volanie toho API implementovali pamäť cache. Vždy keď voláme search API voláme ho na dve miesta ,ktoré už máme uložené v našej databáze ako vrcholy. Tento fakt sme využili a vytvorili sme ďalší typ hrany - CACHE. Keďže pri mestách ktoré sú dopravné uzly vystúpila veľkosť odpovede API až na rádovo 500kb a tento typ dopytu sa nerobí veľmi často rozhodli sme sa odpoveď servera neukladať priamo do databázy ale v nezmenenej podobe na disk a do databázy uložiť len vek cache súboru a referenciu na súbor na disku. Keďže v momentálnej podobe nevyužívame celú odpoveď tohto API mohli by sme zoptimalizovať využitie miesta na disku tým, že by sme najprv údaje zapracovali a uložili len tie, ktoré využívame.

Pri implementácií sme zistili, že pri niektorých trasách API vrátilo niektoré z trás s

nulovou cenou. Tieto trasy sme ignorovali.

## 4.10 Mapa

Na vizualizáciu destinácií a trás sme na rôznych miestach a aplikácií použili javascriptovú knižnicu Mapbox.js.

Keďže Mapbox.js rozširuje knižnicu Leaflet, všetky funkcie knižnice sa volajú z globálneho objektu *L*. Táto funkcia sa zavolá po načítaní stránky. Keďže sa mapy Mapbox.js sa sťahujú z CDN Mapbox musíme najprv aplikáciu identifikovať API kľúčom, ktorý sme si vygenerovali po registrácii. Následne inicializuje samotná mapa volaním funkcie *L.mapbox.map*. Táto funkcia berie ako prvý parameter id HTML elementu, do ktorého sa má mapa zobrazíť, ako druhý parameter berie textový identifikátor typu mapy, ktorý chceme zobrazíť. Po inicializácii sa vytvorí vrstva pre mapu ktorá bude obsahovať markery destinácií. Dáta ktoré obsahujú geografickú polohu destinácií sa vyžadajú vo formáte GeoJson z API našej aplikácie. Keďže pri malom priblížení mapy by sa nedali zreteľne rozlíšiť destinácie ktoré sú blízko pri sebe, po načítaní dát vytvoríme vrstvu clusterov pomocou funkcie *L.MarkerClusterGroup*. Táto vrstva spojí blízke body do clusterov, následne skryje značky týchto bodov a nahradí ich značkou clusteru. Na pridanie bodov do clusterov sa iteruje cez všetky body, ktoré sú po načítaní v mape a každý sa pridá do vrstvy clusterov. Následne sa vrstva clusterov pridá do objektu mapy.

Ako zdrojový formát dát pre všetky mapy v našej aplikácii používame štandardný formát GeoJson. GeoJson je dátová štruktúra zakódovaná do formátu JSON obsahujúca informácie potrebné na vykresľovanie kartografických veličín. V našej implementácii používame nasledovnú štruktúru GeoJson objektu. Hlavný objekt GeoJson obsahuje typ, u nás *FeatureCollection*, a pole objektov typu *Feature*. Každý z týchto objektov obsahuje pole koordinátov, a objekt dodatočných atribútov slúžiacich na upresnenie vizuálu vykresleného objektu. Spomínané pole koordinátov nám spôsobovalo mierne nedorozumenia, lebo v číslovanom poli je ako prvý prvok je uložená zemepisná dĺžka (longitude) a ako druhý zemepisná šírka (latitude) [3], čo je presne opačne ako všetky ostatné API ktoré sme používali.

## 4.11 Vykresľovanie tabuliek

Aby sme zlepšili UX stránky rozhodli sme sa na vykresľovanie dynamického obsahu využiť namiesto HTML obsahu renderovaného na serveri javascriptovú knižnicu na prácu so šablónami a, dynamický obsah načítavame z API našej aplikácie vo formáte JSON. Na vykresľovanie dynamického obsahu sme zvolili knižnicu Handlebars.js. Na príklade ??

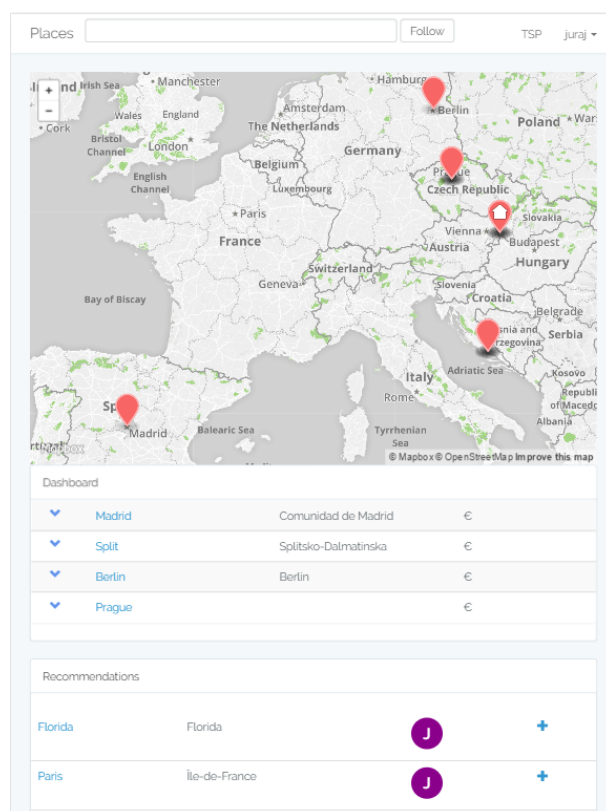
môžeme ukázať príklad našej implementácie dynamickej tabuľky pre hlavnú tabuľku zobrazujúcu destinácie a ich detaily.

Najprv je zadeklarovaná premenná v *template* ktorá bude neskôr obsahovať funkciu renderujúcu šablónu. Potom je zadeklarovaná funkcia *refreshPage*, ktorá bude volaná vždy, keď nejaká funkcia spustí event s menom *appRefresh*. Táto funkcia využije AJAX API knižnice jQuery a stiahne potrebné dáta, potom využije funkciu *template*, ktorá do argumentu berie dáta vo forme poľa objektov a vracia vygenerované HTML ktoré sa následne vkladá na stránku.

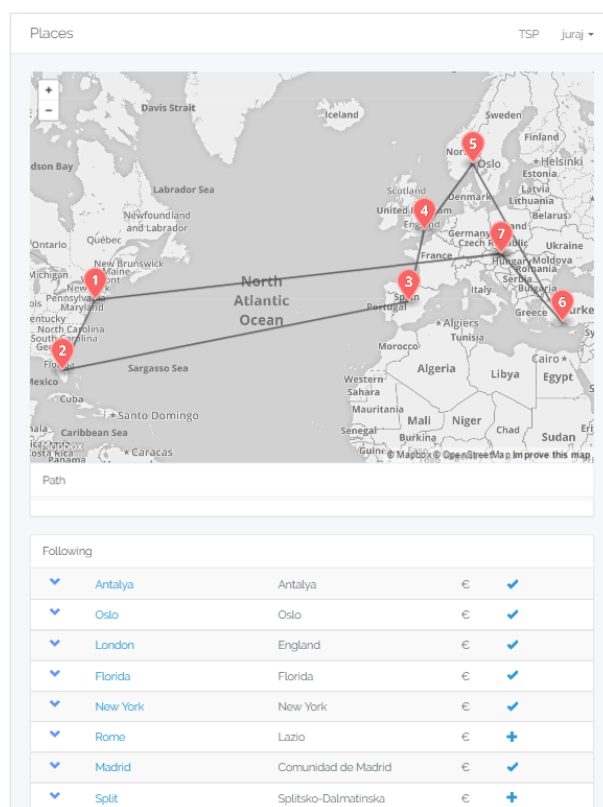
V druhej časti kódu sa najprv zavolá funkcia *Handlebars.compile* do argumentu zoberie šablónu ktorá je uložená v elemente s id *places-template*. Ako návratovú hodnotu vráti funkciu, ktorú uložíme pod menom *template*. Následne sa na event *appRefresh* naviaže volanie funkcie *refreshPage* a prvý krát sa spustí tento event.

Naša šablóna *places-template* obsahuje aj aktívne linky a v nasledujúcom bloku je príklad implementácie vymazania miesta z obľúbených. Na generálny event *click* je naviazaný filtrovaný event, ktorý spustí AJAX dotaz na vymazanie miesta z obľúbených ak element ktorý event spustil obsahuje triedu *delete* a atribút *data-id*.

## 4.12 Vzhľad



Obrázok 9: Ukážka hlavnej obrazovky používateľa



Obrázok 10: Ukážka obrazovky TSP

Tabuľka 1: Výsledky testu rýchlosti API pre zobrazenie zoznamu obľúbených destinácií

Počet záznamov	NeoEloquent	PHP neo4j client	cypher-shell
10	101,6ms	83,2ms	6,7ms
50	281,7ms	195ms	25,2ms
100	705,7ms	358,5ms	32ms
200	1857ms	695,7ms	68ms

## 5 Výsledky práce

### 5.1 Testy rýchlosti

Testy rýchlosti našej aplikácie sme vykonávali na našom serveri pri nulovej záťaži, test pozostával z desiatich volaní nášho API pre zobrazenie zoznamu obľúbených destinácií spomínaného v kapitole 4.6. Zaznamenávali sme čas od doručenia dopytu na server do začiatku sťahovania dát, týmto pádom sme eliminovali vplyv ktorý mala na čas dopytu latencia siete. Testovali sme rýchlosť aplikácie vo verzií s použitím OGM, s použitím OGM s obmedzenou funkcionalitou o dopyty na ostatných používateľov, s použitím PHP Neo4J klienta s dopytom z príkladu ?? a s použitím tohto istého dopytu priamo v bash konzole Neo4J.

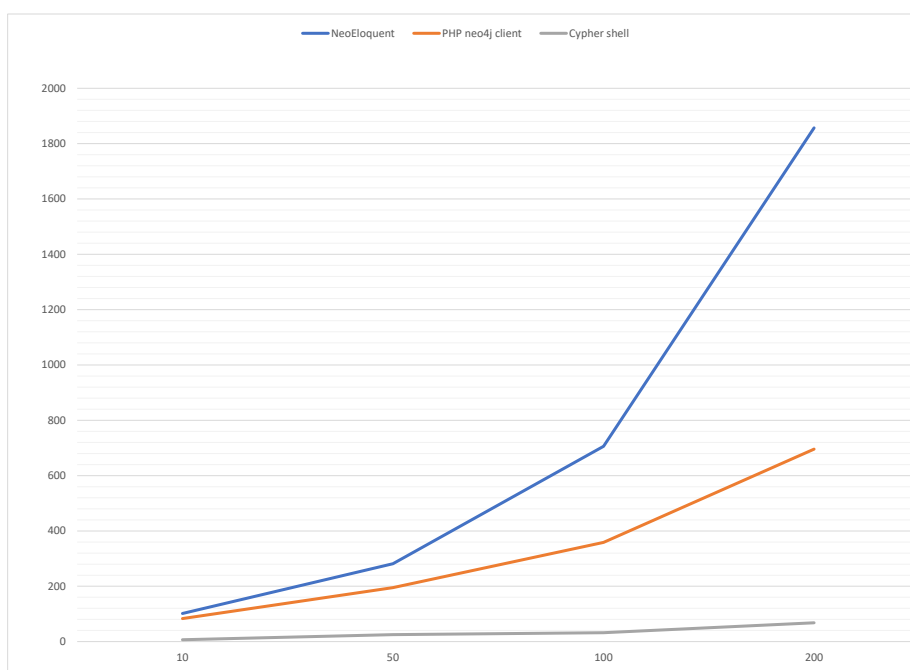
Výsledky tohto testu môžeme vidieť v tabuľke 1 a na grafe.

Na grafe 11 môžeme vidieť ako latencia nášho API narastala s počtom záznamov na zozname obľúbených s použitím dvoch prístupov: OGM a Cypher dopytu cez PHP Neo4J Client. Ďalší údaj je latencia manuálne spusteného dopytu rovnakého ako v druhom prípade ale cez neo4j-shell. Na údajoch vidíme, že latencia stúpa polynomiálne vo všetkých prípadoch aj napriek tomu, že teoretická výpočtová zložitosť prehľadávania hrán v grafe je  $O(1)$ , toto je spôsobené tým, že pre každú položku v zozname sme prehľadávali n ďalších hrán.

Napriek výhodám OGM pre rýchlosť a jednoduchosť implementácie sa ukazuje ako nevhodný pre zložitejšie dopyty, lebo kvôli obmedzenému prístupu k databáze OGM tieto musia byť impelentované cez viaceré dopyty čo sa ukazuje ako neefektívne.

Na zložitejšie dopyty je z dvoch testovaných spôsobov vhodnejšie použiť dopyt pomocou PHP Neo4J Client. Tento prístup však zďaleka nedosiahol rýchlosť priameho dopytu pomocou cypher-shell. Hlavný dôvod tohto rozdielu bude s najväčšou pravdepodobnosťou spôsob nášho testovania údaj o rýchlosti cypher-shell zahŕňa len čas ktorý trvalo vykonanie dopytu, zatiaľ čo ostatné údaje zahŕňali aj latenciu jednorazových operácií ako beh APACHE, PHP a štart frameworku Laravel, tiež zahŕňali latenciu prenosu dát medzi





Obrázok 11: Výsledky testu rýchlosti API pre zobrazenie zoznamu oblúbených destinácií

Neo4J, konverzie JSON dokumentov na PHP objekty a spravoanie údajov v príslušných knižniciach.

+ RELATIOSNSHIPS + NoSQL flexibility NeoEloquent

+ speed

## 6 Závěr

# Záver

Cieľom práce bolo oboznámiť sa s rôznymi predstaviteľmi gravoých databáz, vybrať jedného, navrhnúť a naimplementovať využitie tejto databázy na reálnom príklade.

# Zoznam použitej literatúry

- [1] AT, S. I. Db-engines ranking - popularity ranking of graph dbms, 2017.
- [2] Ian Robinson, Jim Webber, Emil Eifrem. *Graph Databases*. 2. O'Reilly Media, Inc. USA, 2015. s. 224. ISBN: 9781491932001.
- [3] (IETF), I. E. T. F. The gejson format, 2016.
- [4] NEOTECHNOLOGY. Neo4j debian packages, 2017.
- [5] OTWELL, T. Laravel - the php framework for web artisans, 2017.

# Prílohy