EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/I/2022/421000162361

Aplikácia využívajúca sémantické vyhľadávanie

Diplomová práca

2022 **Bc. Bohuslav Ondrúšek**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Aplikácia využívajúca sémantické vyhľadávanie

Diplomová práca

Študijný program:	Informačný manažment
Študijný odbor:	Ekonómia a manažment
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	Ing. Igor Bandurič, PhD.

Bratislava 2022 **Bc. Bohuslav Ondrúšek**

Čestné vyhlásenie

Čestne vyhlasujem, že som záverečnú prácu vypracoval samostatne a uviedol som všetku použitú literatúru.

Dátum: .....

Poďakovanie

Moje poďakovanie patrí môjmu vedúcemu bakalárskej práce Ing. Igorovi Banduričovi, PhD.. Chcem sa mu poďakovať za všetky cenné rady a námety na zlepšenie mojej diplomovej práce. Ďalej sa chcem poďakovať mojej rodine, kamarátovi a priateľke za oporu, ktorú mi poskytli nielen pri tvorbe bakalárskej práce.

Abstrakt

[ONDRUSEK, Bohuslav] : Aplikácia využívajúca sémantické vyhľadávanie – Ekonomická Univerzita v Bratislave , Fakulta Hospodárskej Informatiky, Katedra aplikovanej informatiky. Vedúci záverečnej práce : Ing. Igor Bandurič, PhD. - Bratislava : FHI EU 2022, počet strán : 70

Cieľom diplomovej práce bolo vytvoriť aplikáciu, ktorá využíva sémantické vyhľadávanie, v našom prípade sme pre tento účel použili grafovú databázu Neo4j. Na vytvorenie aplikácie bolo treba namodelovať databázu, vytvoriť databázu a následne naprogramovať jednotlivé vrstvy aplikácie až po prezentačnú vrstvu. Aplikačné rozhranie je naprogramované prostredníctvom jazyka Kotlin s využitím Spring boot technológie a prezentačná vrstva bola vytvorená s využitím rámca Angular. Aplikácie slúži pre vyhľadávanie receptov na základe názvu receptu, zvolených ingrediencií alebo typu receptu.

Prvá kapitola poskytuje detailný pohľad na skúmanú problematiku, s ohľadom na porozumenie konceptom, ktoré využívame v našej práci. Nachádza sa tu detailný opis grafových databáz s dôrazom na konkrétnu databázu Neo4j aj popis dopytovacieho jazyka Cypher, prostredníctvom ktorého sme vytvorili databázu.

Druhá kapitola popisuje cieľ nášho snaženia a metodiku, ktorú sme využili pri tvorbe aplikácie, teda stručne popísaný spôsob, ako sa dajú identifikovať potrebné funkcionality pred vytvorením aplikácie.

Posledná, tretia kapitola, sa zameriava na konkrétne výsledky našej práce, opisuje celý postup tvorby aplikácie, od ukážok jednotlivých modelov databázy až po ukážky zdrojových kódov a používateľského rozhrania.

Kľúčové slová: Neo4j, Kotlin, Cypher, Angular, SpringBoot

Abstract

[ONDRUSEK, Bohuslav]: An application that uses semantic search– University of Economics in Bratislava, Faculty of Business Informatics, Department of applied informatics. Head of Master Thesis: Ing. Igor Bandurič, PhD. - Bratislava: FHI EU 2022. Number of pages: 70

Our diploma thesis aims to create an application that uses semantic search, in our case we used the graph database Neo4j for this purpose. To create the application, it is needed to model and create a database with subsequent programming of individual layers of the application. The application interface is programmed in Kotlin language using Spring boot technology and the presentation layer was created by Angular framework. The application is used to search for recipes based on the name of the recipe, the selected ingredients or the type of recipe.

The first chapter provides a detailed look at the issue to understand the concepts that we use in our work. There is a detailed description of graph databases with emphasis on a specific database Neo4j and a description of the query language Cypher, through which we created the database.

The second chapter describes the goal of our effort and the methodology that was used to create the application. It briefly describes how to identify the necessary functionalities before creating the application.

The last, third chapter, is focusing on the specific results of our work, describing the whole process of creating the application, from examples of individual database models to examples of source code and user interface.

Keywords: Neo4j, Kotlin, Cypher, Angular, SpringBoot

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam skratiek

RDBMS - Relation database management system

RDF - Resource Description Framework

AWS - Amazon Web Services

GPS - Global Positioning System

ACID – Atomicity, Consistency, Isolation, Durability

ETL – Extract, Transform, Load

CSV- Comma-separated values

JDBC - Java Database Connection

APOC- Awesome Procedures on Cypher

OGM - Object Graph Mapper

JPA- Java Persistence API

HTTP - Hypertext Transfer Protocol

API - Application Programming Interface

JSON - JavaScript Object Notation

TCP/IP - Transmission Control Protocol/Internet Protocol

JVM - Java Virtual Machine

REST - Representational state transfer

CRUD – Create, Remove, Update, Delete

SVG - Scalable Vector Graphics

URL - Uniform Resource Locator

Zoznam zdrojových kódov

Úvod

Okrem relačných databáz, existujú aj grafové databázy, ktoré nie sú až tak známe. Relačné databázy sú vhodné pre tabuľkové údaje s konzistentnou štruktúrou a pevnou schémou, ak však databáza obsahuje vysoko prepojené dáta a štruktúra databázy sa postupom času prispôsobuje meniacim sa požiadavkám, relačné databázy sa stávajú veľmi zložité a neprehľadné, v dôsledku čoho sa zväčšuje aj čas odozvy dopytovania dát. Pre tieto prípady použitia sa odporúča zvoliť iný typ databázy a to grafové databázy, ktoré fungujú najlepšie, keď sú dáta, s ktorými pracujeme, veľmi prepojené, a ľahko v nich môžeme meniť schému, poprípade ich rozširovať o nové dáta a s nimi spojené vzťahy.

Grafové databázy sú prispôsobené na prácu s grafmi, takže neukladajú údaje v tabuľkách ale grafoch, ktoré obsahujú vrcholy a hrany. Toto je veľmi využiteľné v dátach, ktoré sú poprepájané vzťahmi, na základe ktorých potom vytvárame dopyty pre získanie potrebných informácií.

V našej práci si predstavíme grafové databázy, pričom si opíšeme aj momentálne najpoužívanejšiu platformu, ktorá obsahuje natívne úložisko grafových databáz a to Neo4j. Stručne si porovnáme implementácie ďalších grafových databáz a v poslednej kapitole navrhneme a implementujeme aplikáciu, ktorá bude využívať databázu Neo4j.

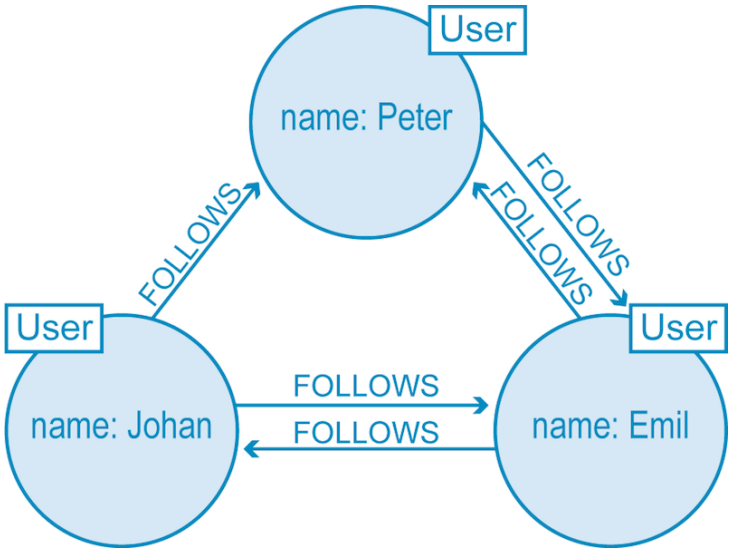
1 Súčasný stav riešenej problematiky

V prvej kapitole si predstavíme grafové databázy, ako fungujú aké majú prípady použitia a porovnáme ich s relačnými databázami. Ďalej sa budeme venovať Neo4j, čo je natívna grafová databáza, ktorá obsahuje rôzne typy nástrojov na prácu s grafmi a získavanie potrebných údajov z veľkého množstva dát. Na záver si stručne porovnáme implementácie iných grafových databáz a zdôvodníme výber databázy Neo4j.

1.1 Pojem grafová databáza

Grafová databáza, tiež označovaná ako sémantická databáza, je softvérová aplikácia určená na ukladanie, dopytovanie a úpravu sieťových grafov. Sieťový graf je vizuálna konštrukcia, ktorá pozostáva z vrcholov, hrán a vlastností. Každý vrchol predstavuje entitu (napríklad osobu) a každá hrana predstavuje spojenie alebo vzťah medzi dvoma vrcholmi. Grafové databázy už dlho existujú v určitých podobách. Napríklad rodokmeň je veľmi jednoduchá grafová databáza.

Grafové databázy sú vhodné na analýzu prepojení, a preto je veľký záujem o využitie grafových databáz pri získavaní údajov zo sociálnych médií, no sú veľmi užitočné aj v podnikovom prostredí, v ktorom sa nachádzajú zložité vzťahy a dynamické schémy ako napríklad riadenie dodávateľského reťazca, identifikácia zdroja problému alebo vytváranie odporúčaní pre zákazníkov na základe ich predchádzajúceho nákupu.



Obrázok 1-Ukážka grafu *[zdroj: https://neo4j.com/blog/why-graph-databases-are-the-future/]*

Grafy môžeme rozlišovať na orientované a neorientované, pričom orientáciu rozpoznávame pomocou hrán, ktoré nám v grafe určujú smer z daného vrcholu do iného vrcholu. Základné typy hrán sú:

- neorientovaná hrana – neusporiadaná dvojica vrcholov, ktorá nemá určený smer priechodu a hranou je možné prechádzať oboma smermi
- orientovaná hrana – usporiadaná dvojica vrcholov, ktorá má určený smer priechodu a hranou je možné prechádzať iba vyznačeným smerom
- násobné hrany – viac hrán spájajúcich rovnaké vrcholy
- slučka – hrana vedúca z vrcholu do toho istého vrcholu, teda do seba samotnej

Existuje viacero typov grafových databáz, ktoré môžeme deliť na základe úložiska alebo na základe dátového modelu. Podľa typu úložiska sú to:

- **Natívne úložisko grafov** — Úložisko, ktoré je špeciálne navrhnuté na ukladanie a správu grafov na disku, kde jednotka je vrchol a hrana. Je dobré pre analýzu grafov s hlbokými odkazmi (viacnásobné skoky). napr. TigerGraph, Neo4j.
- **Relačné úložisko** — Úložisko, ktoré používa relačný model na uloženie tabuľky vrcholov a tabuľky hrán, a za behu pomocou relačného príkazu JOIN spojí dve tabuľky. napr. GraphX.
- **Kľúč-Hodnota úložisko** — Úložisko, ktoré využívajú databázy NoSQL (Cassandra, HBase), kde ku každému kľúču patrí hodnota alebo hodnoty. Napr. JanusGraph.

V prípade delenia podľa dátového modelu, môžeme rozdeliť grafy do nasledových kategórií:

- **Graf vlastností** (napr. Neo4j, AWS Neptune) — Údaje z grafu vlastností sú usporiadané ako uzly, vzťahy a vlastnosti (údaje uložené v uzloch alebo vzťahoch). Uzly sú entity a môžu obsahovať ľubovoľný počet atribútov (párov kľúč-hodnota) nazývaných vlastnosti. Uzly môžu byť označené štítkami, ktoré predstavujú ich rôzne úlohy vo vašej doméne. Označenia uzlov môže tiež slúžiť na pripojenie metadát (ako sú informácie o indexe alebo obmedzeniach) k určitým uzlom. Vzťahy poskytujú riadené, pomenované, sémanticky relevantné spojenia medzi dvoma entitami uzla. Vzťah má vždy smer, typ, počiatočný uzol a koncový uzol. Podobne ako uzly, aj vzťahy môžu mať vlastnosti. Vo väčšine prípadov majú vzťahy kvantitatívne vlastnosti, ako sú váhy, náklady, vzdialenosti, hodnotenia, časové intervaly a pod. Vďaka efektívnemu spôsobu ukladania vzťahov môžu dva uzly

zdieľať ľubovoľný počet alebo typ vzťahov bez straty výkonu. Aj keď sú uložené v určitom smere, vzťahy sa dajú vždy efektívne navigovať oboma smermi.

- **Hypergraf** (napr. HyperGraphDB) — Hypergraf je dátový model grafu, v ktorom môže vzťah (nazývaný hyperedge) spájať ľubovoľný počet daných uzlov, pričom umožňuje ľubovoľný počet uzlov na oboch koncoch vzťahu. Je to užitočné, keď vaše údaje obsahujú veľké množstvo vzťahov „many-to-many“.
- **Trojitý ukladací priestor** (napr. AWS Neptune, AllegroGraph) — Trojitý ukladací priestor alebo RDF (Resource Description Framework) ukladá údaje vo formáte známom ako trojitá štruktúra údajov: subjekt-predikát-objekt. Model grafu RDF sa skladá z uzlov a oblúkov. Zápis RDF grafu predstavuje uzol pre subjekt, uzol pre objekt a oblúk pre predikát. Dáta spracované trojitými úložiskami bývajú logicky prepojené, preto sa trojité úložiská zaraďujú do kategórie grafových databáz. Trojité úložiská však nie sú natívne grafové databázy, pretože nepodporujú susedstvo bez indexu a ani nie sú ich ukladacie mechanizmy optimalizované na ukladanie grafov vlastností. Trojité ukladacie priestory ukladajú trojité položky ako nezávislé prvky, čo im umožňuje horizontálne škálovanie, ale bráni im v rýchlom prechode medzi vzťahmi. [1]

1.2 Využitie grafových databáz

Grafové databázy fungujú najlepšie, keď sú dáta, s ktorými pracujeme, veľmi prepojené a mali by byť reprezentované spôsobom, akým odkazujú na iné dáta, zvyčajne prostredníctvom vzťahov typu „many-to-many“. Na základe toho majú široké uplatnenie v rôznych oblastiach. V tejto podkapitole si predstavíme ich najčastejšie využitie a aké problémy vedia riešiť v jednotlivých prípadoch použitia:

Nástroje na odporúčanie – Ľudia, ktorí kupujú určitý sortiment produktov, často vyhľadávajú podobné produkty ako ostatní zákazníci. To znamená, že ak si jedna osoba kúpi šnúrky po zakúpení tenisiek, existuje šanca, že to urobia aj ostatní. Nástroje odporúčaní vyhľadajú ľudí, ktorých spojili ich nákupy, a potom v grafe vyhľadajú ďalšie úzko spojené produkty. [3]

Detekcia podvodov - Podvodníci sa často správajú podobnou schémou. Vytvorenie grafu transakcií môže identifikovať podvod podľa označenia podozrivých vzorcov, ktoré často nemajú súvislosť s legitímnymi transakciami. Systémy na zisťovanie podvodov používajú databázy grafov na objasnenie vzťahov medzi entitami, ktoré by sa inak mohli len ťažko nájsť. [3]

Strojové učenie - Pretože databázy grafov poskytujú spojivovú vrstvu medzi údajmi, ktoré sa nenachádzajú v iných databázových štruktúrach, sú vhodné na použitie v strojovom učení. Nástroje strojového učenia môžu využiť tieto spojenia na ďalšie vylepšenie a zrýchlenie procesu analýzy. To je obzvlášť výhodné, pokiaľ ide o odporúčanie scenára, kedy stroj skenuje databázu a poskytuje používateľovi východiskový bod alebo počiatočné body, cez ktoré má prejsť. [2]

Hľadanie cesty (GPS) - Ak sú križovatkami uzly a ulice medzi nimi sú prepojenia alebo hrany, potom je graf dobrým abstraktným zobrazením sveta. Výber cesty pre autonómne auto si vyžaduje iba postupnosť uzlov a väzieb medzi nimi. [3]

Spravovanie kmeňových dát v podniku – Kmeňové dáta sú tvorené základnými dátovými bodmi spoločnosti. Tieto údaje zvyčajne poskytujú prehľad súvisiaci s jadrom podnikania, vrátane zákazníkov, dodávateľov, účtov, zamestnancov, cieľov a operácií. O tom, čo sa považuje za kmeňové dáta, rozhodujú riadiace tímy a obchodné zainteresované strany. Po splnení týchto dátových štandardov môžu používatelia analyzovať údaje, ktoré potrebujú na identifikáciu kľúčových metrík, ktoré odhaľujú oblasti záujmu, aby bolo možné prijať príslušné opatrenia na zlepšenie prevádzky. [2]

1.3 Porovnanie grafových databáz s relačnými databázami

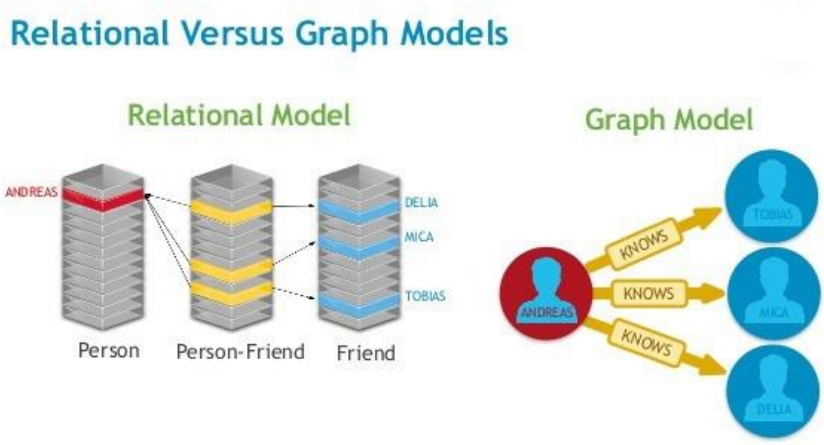
Nedávne rozšírenie databázových technológií je dôkazom toho, že relačné databázy nie sú tým správnym nástrojom pre každý typ a množstvo údajov. Samozrejme majú svoje využitie: Tabuľkové údaje s konzistentnou štruktúrou a pevnou schémou sa perfektne hodia pre relačné databázy (RDBMS). Ak však naša aplikácia vyžaduje flexibilitu alebo má vysoko prepojené údaje, potom je čas hľadať alternatívu k nášmu RDBMS. [4]

Slovo „relačný“ v relačných databázach pochádza zo súvisiacich stĺpcov v tabuľke, nie zo vzťahu informácií v rôznych tabuľkách. To je veľmi odlišné od reálneho sveta, kde existujú vzťahy medzi jednotlivými dátovými prvkami. [4]

V tradičnej relačnej databáze alebo databáze SQL sú údaje usporiadané do tabuliek. Každá tabuľka zaznamenáva údaje v konkrétnom formáte s pevným počtom stĺpcov, každý stĺpec má vlastný dátový typ (celé číslo, čas / dátum, voľný text atď.). Relačné databázy fungujú rýchlo a sú spoľahlivé v prípade, že máme konzistentné údaje, ktoré majú pevnú schému.

Ak však máme obrovské množstvo rôznych typov dát, kde dáta majú medzi sebou veľa vzťahov, pričom vieme, že dátový model sa bude flexibilne rozširovať o nové typy dát

aj vzťahy, tak sa relačná databáza stáva veľmi zložitá a neprehľadná, pričom každé nové rozšírenie má dopad na zložitosť ale aj rýchlosť dopytovania dát. V takýchto prípadoch je vhodné začať využívať databázu grafov, ktorá je prehľadnejšia a je oveľa jednoduchšie do nej pridávať nové entity s novými vzťahmi.



***Obrázok 2- Ralation vs Graph model**, [zdroj: https://medium.com/dev-genius/graph-database-vs-relational-database-70f6156f7415]*

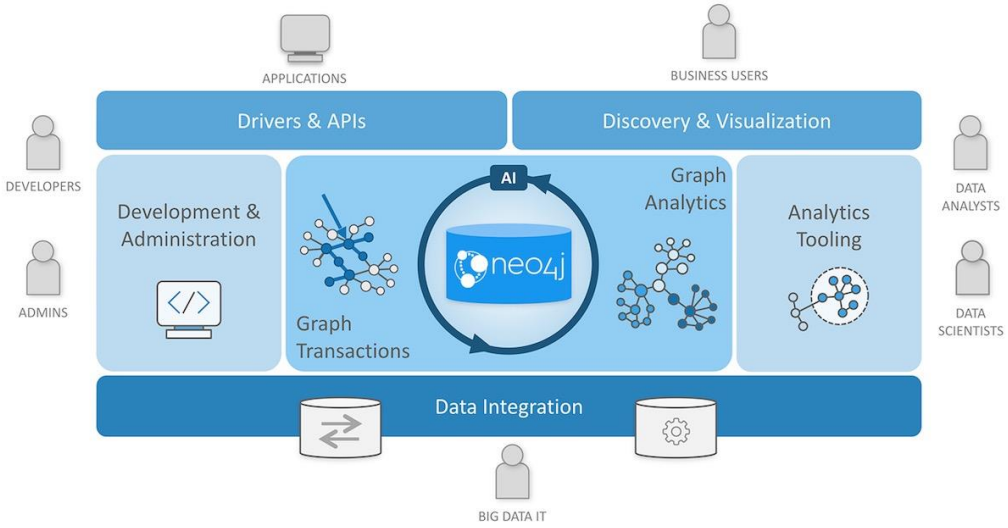
Ako môžeme vidieť na obrázku 2 tak v prípade relačnej databázy musíme vložiť medzi entity „Person“ a „Friend“ medzi-tabuľku „Person-Friend“, v ktorej sa nachádzajú primárne kľúče obidvoch entít.

Na rozdiel od relačnej databázy je grafová databáza štruktúrovaná výlučne podľa dátových vzťahov. Databázy grafov pracujú so vzťahmi, nie ako so štruktúrou schémy, ale ako s údajmi, podobne ako s inými hodnotami. Inými slovami, v grafovej databáze nám logický model zobrazuje spôsob, akým uvažujeme o probléme. [4]

1.4 Neo4j platforma

Neo4j je open-source, NoSQL, natívna grafová databáza, ktorá poskytuje transakčný backend kompatibilný s ACID transakciami pre našu aplikáciu, ktorý je verejne dostupný od roku 2007. Implementuje grafový model až po úroveň úložiska. Údaje sa neukladajú ako „grafová abstrakcia“ nad inou technológiou ale ukladajú sa presne tak, ako by sme ich nakreslili na tabuľu. To je dôležité, pretože je to hlavný dôvod, prečo Neo4j je poprednou grafovou databázou. Okrem základného grafu poskytuje Neo4j to, čo by ste od databázy očakávali; ACID transakcie, podpora klastrov a núdzové prepnutie za behu aplikácie. Táto stabilita a vyspelosť sú dôvodom, prečo sa už roky používa v produkčných scenároch pre veľké podniky. [5]

Vďaka komunite Neo4j existujú ovládače pre takmer každý populárny programovací jazyk, z ktorých väčšina napodobňuje existujúce prístupy ovládačov k databázam. Tieto ovládače sú napríklad pre jazyky ako Java, Python, JavaScript, Go, Ruby, PHP, Perl, C, C++, R a ďalšie.



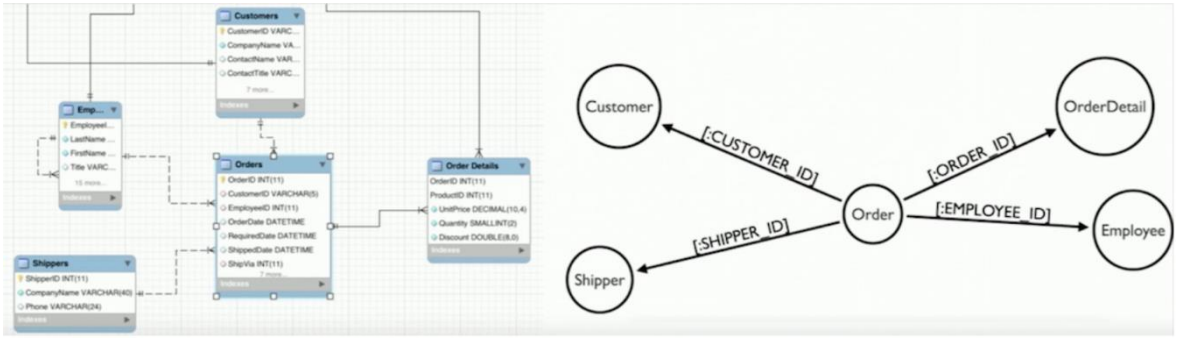
***Obrázok 3 - Neo4j**, [zdroj: https://neo4j.com/developer/graph-platform/]*

Medzi hlavné vlastnosti Neo4j patria:

- **Flexibilná schéma:** Neo4j sa riadi dátovým modelom nazývaným grafový model. Graf obsahuje uzly a uzly sú navzájom prepojené. Uzly a vzťahy ukladajú údaje do párov kľúč-hodnota známych ako vlastnosti.
- **Vlastnosť ACID:** Neo4j podporuje všetky vlastnosti ACID (atomicita, konzistencia, izolácia a trvanlivosť).
- **Škálovateľnosť:** Neo4j nám uľahčuje škálovanie databázy zvýšením počtu čítaní/zápisov a objemu bez ovplyvnenia integrity údajov a rýchlosti spracovania dotazov.
- **Spoľahlivosť:** Neo4j poskytuje replikáciu pre bezpečnosť a spoľahlivosť údajov.
- **Cypher Query Language:** Neo4j poskytuje výkonný deklaratívny dopytovací jazyk s názvom Cypher Query language. Používa sa na vytváranie a získavanie vzťahov medzi údajmi bez použitia zložitých dotazov.

Neo4j taktiež obsahuje nástroje, ktoré pomáhajú developerom pri tvorbe grafových databáz alebo vizualizácií grafov, čo ocení hlavne manažment podniku pri prezentácií údajov zákazníkovi . Tieto nástroje sú:

- **Cypher Shell CLI** sa používa na spúšťanie dotazov a vykonávanie administratívnych úloh. V predvolenom nastavení je shell interaktívny, ale môžeme ho použiť aj na skriptovanie zadaním skriptu priamo na príkazovom riadku alebo prepojením súboru s príkazmi skriptov (vyžaduje PowerShell v systéme Windows). Komunikuje cez protokol Bolt. [6]
- **Neo4j Browser** je nástroj zameraný na vývojárov, ktorý nám umožňuje vykonávať dotazy Cypher a vizualizovať výsledky. Je to predvolené vývojárske rozhranie pre Enterprise aj Community edíciu Neo4j. [7]
- **Neo4j Desktop** je klientska aplikácia, ktorá nám pomôže pracovať s Neo4j, či už len začíname, alebo máme predchádzajúce skúsenosti. Je navrhnutý tak, aby nám ako novému používateľovi pomohol učiť sa a experimentovať s Neo4j. Je lokálnym vývojovým prostredím pre projekty, kde budeme Neo4j používať. Pomocou Neo4j Desktop môžeme vytvoriť ľubovoľný počet lokálnych databáz, ktoré podporujú zdroje nášho počítača. [8]
- **Neo4j Bloom** je aplikácia na vizuálnu interakciu s údajmi z grafu. Graf dáva akúkoľvek informáciu do kontextu a spája všetky uzly. Ľudia, miesta a veci, produkty, služby, transakcie, identity a udalosti. Neo4j Bloom zobrazuje vzory, o ktorých intuitívne vieme, že existujú v našich údajoch, a odhaľuje nové vzory, ktoré sme možno neočakávali. Toto zobrazenie údajov otvára nové spôsoby myslenia, nové spôsoby práce a nové možnosti pre obchodný pohľad na graf. [9]
- **Neo4j ETL Tool** – slúži na importovanie údajov z relačných systémov do Neo4j. Nástroj Neo4j ETL bol vyvinutý, aby bol počiatočný import jednoduchý. Extrahuje schému z akejkoľvek relačnej databázy a umožňuje nám ju premeniť na grafovú schému. Následne sa postará o import údajov do nášho grafu, buď hromadne alebo v online móde. Na používanie tohto nástroja nepotrebujeme poznať jazyk Cypher. Neo4j ETL Tool zahŕňa napríklad funkcionalitu :
  - spravuje viacero pripojení RDBMS
  - automaticky extrahuje databázové metadáta z relačnej databázy
  - odvodzuje model grafu
  - vizuálne upravuje štítky, typy vzťahov, názvy vlastností a typy atribútov
  - získava relevantné údaje CSV z relačných databáz
  - spúšťa hromadný alebo online import[10]



***Obrázok 4-Porovnanie modelov**, [zdroj: https://neo4j.com/labs/etl-tool/]*

Na uvedenom obrázku môžeme vidieť ako vyzerá model grafovej databázy po transformácií z modelu relačnej databázy. Aj na základe týchto modelov je vidieť veľké zjednodušenie modelovania databázy, kedy na prvý pohľad vieme skorej identifikovať entity aj vzťahy medzi nimi.

- **Neo4j APOC Library -** APOC je skratka pre Awesome Procedures on Cypher. Vývojári potrebovali napísať svoje vlastné postupy pre bežné funkcie, ktoré Cypher alebo databáza Neo4j ešte neimplementovali a každý vývojár môže napísať svoju vlastnú verziu týchto funkcií, čo spôsobí veľa duplikácií. Neo4j vytvorila knižnicu APOC ako štandardnú knižnicu nástrojov pre bežné postupy a funkcie. To umožnilo vývojárom naprieč platformami a odvetviami používať štandardnú knižnicu pre bežné procedúry a písať iba vlastnú funkčnosť pre obchodnú logiku a potreby, špecifické pre jednotlivé prípady použitia. Knižnica APOC je považovaná za najväčšiu a najpoužívanejšiu knižnicu Neo4j. Zahŕňa viac ako 450 štandardných postupov, ktoré poskytujú funkcie pre pomocné programy, konverzie, aktualizácie grafov a ďalšie. Sú dobre podporované a veľmi ľahko sa spúšťajú ako samostatné funkcie alebo sa dajú zahrnúť do dopytov Cypher.[11]

Spôsob použitia procedúry APOC je realizovaný cez príkaz Cypher. Používa sa nasledovná syntax:

CALL <package>.<subpackage>.<procedure>(<argument1>,<argument2>,...);

Príkaz CALL sa používa na volanie procedúry, za ktorým nasleduje názov balíka, ktorý označuje, ktorý balík sa volá . Ako v mnohých iných programovacích jazykoch, aj tu sa nachádzajú pod balíky, aby sme zúžili rozsah toho, čo hľadáme.

1.5 Vyhľadávacie algoritmi v Neo4j

Grafové algoritmy poskytujú jeden z najúčinnejších prístupov k analýze prepojených údajov, pretože ich matematické výpočty sú špeciálne zostavené tak, aby fungovali na vzťahoch. Popisujú kroky, ktoré je potrebné vykonať pri spracovaní grafu, aby sa objavili jeho všeobecné kvality alebo špecifické veličiny.

Grafové algoritmy sú podmnožinou nástrojov na analýzu grafov. Grafová analýza je použitie akéhokoľvek prístupu založeného na grafoch na analýzu prepojených údajov. Môžeme použiť rôzne metódy: môžeme sa dotazovať na údaje z grafu, použiť základné štatistiky, vizuálne skúmať grafy alebo začleňovať grafy do našich úloh strojového učenia. Dotazovanie, založené na grafoch, sa často používa na lokálnu analýzu údajov, zatiaľ čo grafové výpočtové algoritmy zvyčajne odkazujú na globálnejšiu a iteratívnu analýzu.

Grafové algoritmy používame v oblasti výpočtovej analýzy a vedy o údajoch. Na základe matematickej teórie grafov používajú grafové algoritmy vzťahy medzi uzlami na odvodenie organizácie a dynamiky zložitých systémov. Sieťoví vedci používajú tieto algoritmy na odhaľovanie skrytých informácií, testovanie hypotéz a predpovede správania.[12]

1.5.1 Typy grafových algoritmov

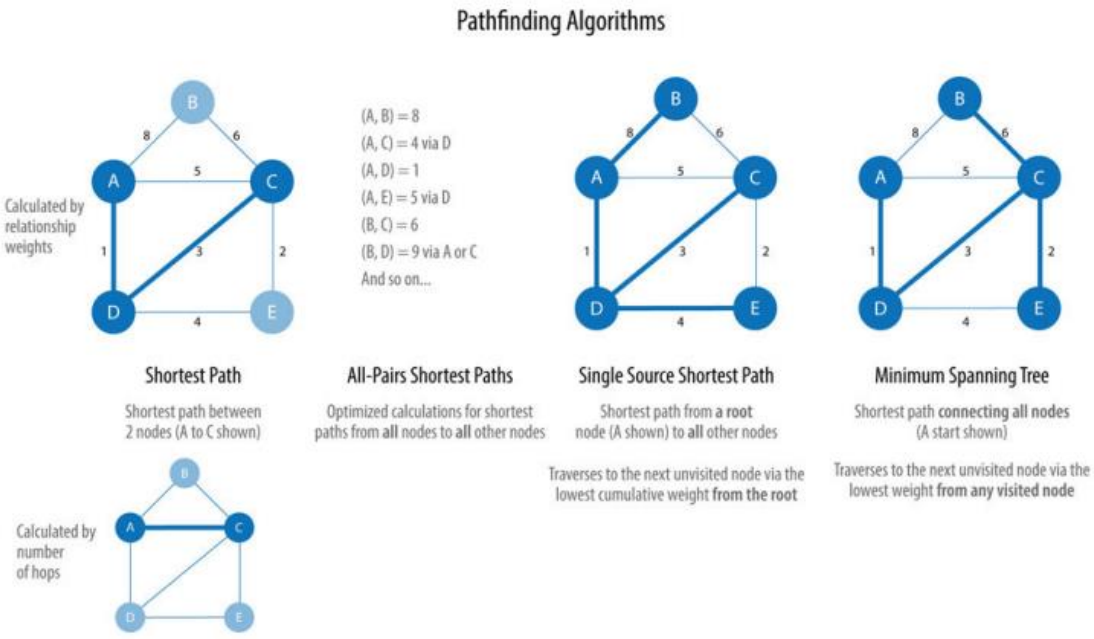
V tejto podkapitole si stručne predstavíme tri hlavné oblasti analýzy, ktoré sú jadrom grafových algoritmov.

Hľadanie cesty/trasy (Pathfinding) – Algoritmy hľadania cesty sú založené na algoritmoch vyhľadávania grafov a skúmajú trasy medzi uzlami, začínajú v jednom uzle a prechádzajúce cez vzťahy, kým sa nedosiahne cieľ. Tieto algoritmy sa používajú na identifikáciu optimálnych trás prostredníctvom grafu, ako je plánovanie logistiky, najmenej nákladné procesy alebo smerovanie IP. Algoritmy hľadania cesty sú základom analýzy grafov. Hľadanie najkratších ciest je pravdepodobne najčastejšou úlohou vykonávanou pomocou grafových algoritmov a je predchodcom niekoľkých rôznych typov analýz. [12]

Najčastejšie používané algoritmy hľadania cesty sú:

- Hľadania najkratšej cesty medzi dvomi uzlami
- Hľadanie najkratšej cesty medzi všetkými pármi uzlov, to znamená zo všetkých uzlov do všetkých ostatných uzlov
- Hľadanie najkratšej cesty z počiatočného uzla do všetkých ostatných uzlov
- Hľadanie najkratšej cesty na prepojenie všetkých uzlov v grafe

- Náhodné prechádzanie - je to užitočný pred proces, kedy získame počiatočné dáta, aby sme zistili efektívnosť aplikovaného algoritmu a



Obrázok 5- Hľadanie najkratšej cesty, *[zdroj. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j_Graph_Algorithms.pdf]*

Na uvedenom obrázku môžeme vidieť ako sa odlišuje hľadanie najkratšej cesty rôznymi upravenými algoritmami.

Algoritmy centrality (Centrality) - Algoritmy centrality sa používajú na pochopenie úloh jednotlivých uzlov v grafe a ich vplyvu na sieť. Sú užitočné, pretože identifikujú najdôležitejšie uzly a pomáhajú nám pochopiť dynamiku skupiny, ako je dôveryhodnosť, dostupnosť, rýchlosť, akou sa informácie šíria, a mosty medzi skupinami. Hoci mnohé z týchto algoritmov boli vynájdené na analýzu sociálnych sietí, našli využitie aj v iných odvetviach a oblastiach. Dôležitosť uzla môže znamenať že:

- má veľa priamych spojení
- je tranzitívne spojený s ďalšími dôležitými uzlami
- môže dosiahnuť iné uzly s niekoľkými „skokmi“
- nachádza sa na najkratšej ceste z párov uzlov. [12]

Algoritmy centrálnosti môžeme rozdeliť na:

- **Algoritmus stupňa centrality (The Degree Centrality algorithm)** – je možné ho použiť na nájdenie“ populárnych“ uzlov v grafe. Stupeň centrality meria počet prichádzajúcich alebo odchádzajúcich (alebo oboch) vzťahov z uzla v

závislosti od orientácie projekcie vzťahu. Dá sa použiť na ohodnotené alebo neohodnotené grafy. V ohodnotených prípadoch algoritmus vypočíta súčet všetkých kladných hodnôt susedných vzťahov uzla pre každý uzol v grafe. Nekladné ohodnotenia sa ignorujú. [13]
Použiť algoritmus môžeme napríklad pri zisťovaní podvodníkov na online aukciách, kde sa podvodníci snažia umelo navyšovať ceny alebo na zistenie kľúčových ľudí v sociálnej sieti.

- **Algoritmus Centrálnej blízkosti (The Closeness Centrality algorithm)** - je spôsob detekcie uzlov, ktoré sú schopné efektívne šíriť informácie cez graf. Meria priemernú vzdialenosť (inverznú vzdialenosť) medzi uzlom a všetkými ostatnými uzlami. Uzly s vysokým „skóre“ blízkosti majú v priemere najkratšie vzdialenosti od všetkých ostatných uzlov. [14]

Použitie tohto algoritmu slúži napríklad na zistenie jednotlivcov v organizačnej sieti, ktorí majú vysoký stupeň centrálnej blízkosti a sú teda vhodní na kontrolu a získavanie dôležitých informácií a zdrojov v rámci organizácie alebo na zistenie šírenia infekcie cez sociálnu sieť.

- **Algoritmus Centrálnej medzistrednosti (Betweenness Centrality algorithm)** - je spôsob zisťovania vplyvu, ktorý má uzol na tok informácií alebo zdrojov v grafe. Môže sa použiť na nájdenie uzlov, ktoré slúžia ako mosty z jednej časti grafu do druhej.

Používa sa napríklad na meranie sieťového toku v procesoch doručovania balíkov alebo v telekomunikačných sieťach. Tieto siete sa vyznačujú prevádzkou, ktorá má známy cieľ a vedie najkratšou možnou cestou. Ďalšie využitie je na pomoc mikro-blogerom šíriť ich dosah na sociálnej sieti pomocou nástroja odporúčaní, ktorý sa zameriava na influencerov, s ktorými by mali v budúcnosti interagovať. [14]

- **Algoritmus PageRank (The PageRank algorithm)** - meria dôležitosť každého uzla v grafe na základe počtu prichádzajúcich vzťahov a dôležitosti zodpovedajúcich zdrojových uzlov. Základným predpokladom je, že stránka je len taká dôležitá ako stránky, ktoré na ňu odkazujú.

Používa ho napríklad Twitter na prezentovanie odporúčaní iných účtov používateľom, ktoré by mohli chcieť nasledovať alebo sa využíva na hodnotenie verejných priestranstiev alebo ulíc, na predpovedanie plynulosti premávky a pohybu ľudí v týchto oblastiach. Algoritmus prebieha cez graf, ktorý obsahuje križovatky

spojené cestami, kde „skóre“ PageRank zobrazuje tendenciu ľudí zaparkovať alebo ukončiť svoju cestu na ulici. [14]

Detekcia komunity (Community Detection) - Prepojenosť je základný koncept teórie grafov, ktorý umožňuje sofistikovanú sieťovú analýzu, ako je vyhľadávanie komunít. Väčšina sietí reálneho sveta vykazuje pod-štruktúry, často kvázi fraktálne, viac-menej nezávislých pod-grafov. Konektivita sa používa na vyhľadávanie komunít a kvantifikáciu kvality zoskupení. Vyhodnotenie rôznych typov komunít v rámci grafu môže odhaliť štruktúry, ako sú centrá a hierarchie, a tendencie skupín priťahovať alebo odpudzovať ostatných. Všeobecným princípom pri hľadaní komunít je, že jej členovia budú mať viac vzťahov v rámci skupiny ako s uzlami mimo ich skupiny. Identifikácia týchto súvisiacich súborov odhalí zhluky uzlov, izolované skupiny a štruktúru siete. Tieto informácie pomáhajú odvodiť podobné správanie alebo preferencie rovesníckych skupín, odhadnúť odolnosť, nájsť vnorené vzťahy a pripraviť údaje pre ďalšie analýzy. [12]

Tieto algoritmy sa využívajú napríklad na nájdenie množiny firiem, v ktorých každý člen priamo a/alebo nepriamo vlastní akcie každého iného člena. Hoci má výhody, ako je zníženie transakčných nákladov a zvýšenie dôvery, tento typ štruktúry môže oslabiť konkurenciu na trhu alebo na výpočet konektivity rôznych konfigurácií siete pri meraní výkonu smerovania v bezdrôtových sieťach a v neposlednom rade aj na prvý krok v mnohých grafových algoritmoch, ktoré fungujú iba na silne prepojených grafoch. V sociálnych sieťach je skupina ľudí vo všeobecnosti silne prepojená (Napríklad študenti triedy alebo akéhokoľvek iného spoločného miesta). Mnohým ľuďom v týchto skupinách sa vo všeobecnosti páčia nejaké spoločné stránky alebo hrajú spoločné hry.

1.6 Implementácia Neo4j grafovej databázy so Spring Boot Neo4j Data

Spring Data Neo4j aplikuje koncepty na vývoj riešení pomocou grafového úložiska Neo4j. Repozitáre poskytuje ako vysokoúrovňovú abstrakciu na ukladanie a dopytovanie údajov pre všeobecný prístup k doméne alebo vykonávanie všeobecných dotazov. Všetky sú integrované s transakciami aplikácie Spring. Základná funkčnosť podpory Neo4j môže byť použité priamo, buď prostredníctvom Neo4jClient alebo Neo4jTemplate. Obe poskytujú integráciu s transakciami na úrovni aplikácií Spring. Na nižšej úrovni môžeme získať inštanciu ovládača Bolt, ale potom musíme spravovať svoje vlastné transakcie.

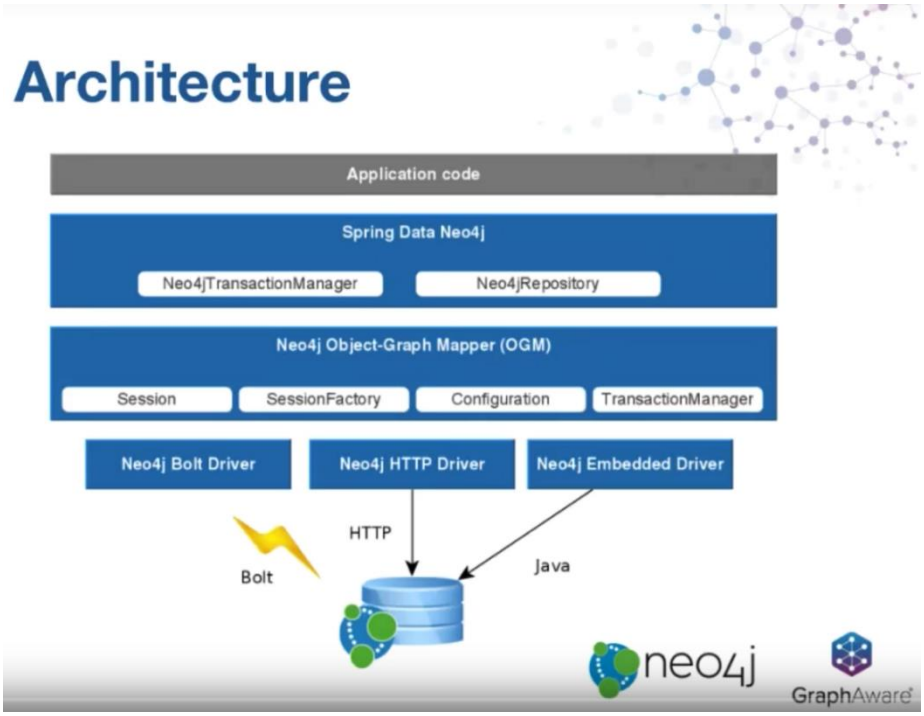
OGM mapuje uzly a vzťahy v grafe na objekty a referencie v modeli domény. Inštancie objektov sú mapované na uzly, zatiaľ čo referencie objektov sú mapované pomocou vzťahov alebo serializované do vlastností. OGM abstrahuje databázu a poskytuje

pohodlný spôsob, ako zachovať model domény v grafe a dotazovať sa naň bez toho, aby sme museli priamo používať ovládače nízkej úrovne a tiež poskytuje vývojárom flexibilitu pri poskytovaní vlastných dotazov tam. [15]

Spring Boot spracováva veľkú časť štandardného kódu pre konfiguráciu aplikácií a bootstrap, čo dáva vývojárom príležitosť preskočiť veľkú časť tejto práce a zamerať sa na obchodnú logiku a samotné dáta.

Spring Data Neo4j pomáha vývojárom písať kód tak, ako by to normálne robili v Springu, ale pracujú s databázou grafov. Robí to pomocou Neo4j objektového grafu mapovača (OGM), ktorý má podobnú úlohu ako Hibernate v JPA, na interakciu s ovládačmi a v konečnom dôsledku s databázou grafov Neo4j.

Súčasný Spring Data Neo4j je nástupcom Spring Data Neo4j + Neo4j-OGM. Samostatná vrstva Neo4j-OGM (Neo4j Object Graph Mapper) bola nahradená infraštruktúrou Spring, ale základné koncepty Object Graph Mapper (OGM) stále fungujú.



Obrázok 6 - Spring Neo4j data *[zdroj: https://miro.medium.com/max/1400/0*8JbCFKsPIibpjmkJ]*

Na lepšie znázornenie je nižšie uvedený model architektúry.

Na pripojenie k databáze sa používajú v súčasnosti ovládače, ktoré sú dostupné v troch variantoch: Embedded, HTTP a binárny protokol Bolt. Ten používa oficiálny ovládač Neo4j Java. Spring Data Neo4j poskytuje kód nad Neo4-OGM, ktorý pomáha rýchlo vytvárať aplikácie Neo4j založené na Spring.

1.7 Porovnanie grafových databáz

V tejto podkapitole si stručne predstavíme najznámejšie grafové databázy, ktoré sú v súčasnosti dostupné.

1.7.1 ArangoDB

ArangoDB je voľne dostupná natívna multi-modelová databáza. Podporuje grafy, dokumenty a dátové modely typu kľúč-hodnota, ktoré používateľom umožňujú voľne kombinovať všetky dátové modely v jednom dotaze. Databáza nám umožňuje rýchlo sa prispôsobiť požiadavkám na výkon a úložisko pomocou vertikálneho aj horizontálneho škálovania. Podporuje tiež nezávislé škálovanie rôznych dátových modelov a umožňuje nám rýchlo zmenšiť našu aplikáciu, aby sme ušetrili hardvérové a prevádzkové náklady. Systém spracovania vo vnútri ArangoDB je založený na: Pregel: čo je systém pre spracovanie veľkých grafov. Tento koncept nám umožňuje vykonávať distribuované spracovanie grafov bez potreby distribuovaného globálneho uzamykania. [16]

ArangoDB je implementované v jazyku C++, pričom tabuľka je kolekcia a jeden riadok v tabuľke je jeden dokument. Prístup k API má cez HTTP ale aj cez vlastný binárny protokol s názvom VelocyStream, ktorý možno použiť na lepšiu priepustnosť. Požiadavky HTTP sú ľahko mapovateľné na VelocyStream a neexistuje žiadna samostatná dokumentácia, pretože API je v podstate rovnaké pre oba sieťové protokoly. Serverová API očakáva, že klienti budú odosielať údaje vo formáte JSON alebo vo vlastnom binárnom formáte VelocyPack od ArangoDB. Využíva rôzne ovládače pre rôzne typy jazykov ako napríklad Java, Python, .NET, JavaScript a pod. Server beží na porte 8529 a pripája sa pomocou inštancie triedy ArangoDB.

1.7.2 OrientDB

OrientDB je prvý multi-modelový, voľne dostupný NoSQL DBMS, ktorý kombinuje silu grafov a flexibilitu dokumentov do jednej škálovateľnej, vysokovýkonnej operačnej databázy. Ako priama odozva na pretrvávanie polyglotov, multimodelové databázy podporujú viaceré dátové modely, pričom ich kombinujú, aby sa znížila prevádzková zložitosť a zachovala sa konzistencia dát. OrientDB bol navrhnutý od základov s výkonom ako kľúčovou špecifikáciou. Je rýchly pri operáciách čítania aj zápisu. Uloží až 120 000 záznamov za sekundu. S nulovou konfiguráciou multi-master architektúry je OrientDB ideálny pre cloudové riešenie. Stovky serverov môžu zdieľať pracovné zaťaženie a horizontálne škálovať naprieč distribuovanými modernými dátovými centrami.

OrientDB je napísaný výhradne v jazyku Java a môže bežať na akejkoľvek platforme bez konfigurácie a inštalácie. V databázovom systéme je implementované rozhranie Blueprints, čo znamená, že dátový model je orientovaný graf s vlastnosťami. OrientDB má možnosť udržiavať pevnú schému databázy pomocou obmedzenia a tried. Dopytovanie dát sa vykonáva pomocou jazyka Gremlin a rozšíreného jazyka SQL, pričom sú tieto operácie vykonávané v transakciách s ACID vlastnosťami pomocou MVCC25 mechanizmu. OrientDB podporuje 3 druhy ovládačov, a to natívny binárny diaľkový ovládač, ktorý komunikuje priamo s TCP/IP socketom pomocou binárneho protokolu, HTTP REST/JSON, ktorý komunikuje priamo s TCP/IP socketom pomocou protokolu http a „zabalenú“ Javu ako vrstvu, ktorá spája natívny ovládač Java, čo je celkom jednoduché pre jazyky, ktoré bežia pomocou JVM ako Scala, Groovy a JRuby [17]

OrientDB má nový bezpečnostný systém, ktorý inštaluje špecializovanú triedu OSecurityShared s názvom OSecurityExternal, ktorá podporuje externú autentifikáciu používateľov, čo znamená, že meno používateľa možno overiť mimo sféry lokálnej databázy. Typicky existuje „reťazec autentifikátorov“ špecifikovaný v konfiguračnom súbore „security.json“ v sekcii „autentifikácia“, ktorý sa skontroluje, či je možné overiť používateľské meno. [18]

1.7.3 Titan DB

Titan je distribuovaná databáza grafov optimalizovaná na ukladanie a dopytovanie grafov reprezentovaných cez klaster strojov. Klaster sa môže elasticky škálovať, aby podporoval rastúci súbor údajov a používateľskú základňu. Titan má pripojiteľnú architektúru úložiska, ktorá mu umožňuje stavať na overenej databázovej technológii, ako je Apache Cassandra, Apache HBase alebo Oracle BerkeleyDB, pričom každý ma k dispozícií iný druh škálovania, dostupnosti a udržania konzistencie dát. Okrem toho pripojiteľná architektúra indexovania podporuje elastické vyhľadávanie, Solr a Lucene. Titan podporuje globálnu grafovú analýzu, reportovanie a ETL prostredníctvom integrácie s Apache Spark, Apache Giraph a Apache Hadoop.

Titan natívne implementuje zásobník grafov Apache TinkerPop vrátane dotazovacieho jazyka grafov Gremlin. Titan Server vkladá Cassandru aj odľahčenú verziu Rexster do toho istého JVM. K Titanu sa môžeme pripojiť z akéhokoľvek programovacieho jazyka, pretože Titan Server má dva koncové body, HTTP RESTful dostupný na http://localhost:8182/graph a operácie CRUD je možné vykonávať cez HTTP a JSON a druhý koncový bod binárneho protokolu RexPro, dostupný na porte 8184, ktorý slúži pre ľubovoľne zložité operácie čítania/zápisu cez rámec Gremlin a je možné ich vykonávať

prostredníctvom MessagePack. Prvá operácia, s grafom, ktorá nie je súčasťou transakcie vytvorí novú transakciu, ktorá zavolá funkciu commit() alebo rollback(). [20]

Na konci tejto podkapitoly si stručne zhrnieme uvedené porovnania v tabuľke.

	Arango DB	Orient DB	Titan DB	Neo4j DB
Napísaná v jazyku	C++	Java	C	Java
Dátové modely	Kľúč/hodnota, graf, dokument	Multi-model: Graph, Document, Key/value, Object	Kolekcia vrcholov s ich zoznamom susedných vrcholov	graf, kľúč/hodnota
Využívaná spoločnosťami	Cisco, Refinitiv, Barclays, Kaseware	Eurotel, Ericsson, Flux, Activity Stream	Trailquest, Alpha Vertex, Theage, Allhomes, Freeth Technologie	Airbnb, Adidas, Adobe, eBay, NASA, Airbus a ďalšie
Škálovanie	vertikálne aj horizontálne	horizontálne, lineárne	Elastické a lineárne,	horizontálne
API	HTTP, Velocity Stream	HTTP REST/JSON, Binárny protokol	HTTP API through Apache TinkerPop's Gremlin Server	REST API/JSON, HTTP API
Ovládače	Java, Python, .NET, JavaScript	Java, Python, PhP, JavaScript, .NET, Scala, Groovy a ďalšie	Java, Kotlin, .NET, Python, JavaScript, PhP, Ruby, Rust, Scala a ďalšie	Java, Spring, .NET, JavaScript, Go, Python a ďalšie
Port	8529	2480	5432	7474

Table 1 - porovnanie grafových databáz, *[zdroj: vlastné spracovanie]*

V uvedenej tabuľke sme si stručne zhrnuli opísané grafové databázy do prehľadnej tabuľky, kde sú základné špecifikácie databáz aj so spoločnosťami, ktoré tieto databázy využívajúú.

1.8 Výber grafovej databázy

V našom prípade sme si zvolili Neo4j, keďže má podporu „Springu“ a naše aplikačné rozhranie plánujeme vypracovať pomocou „Spring boot“ technológie, kde budeme vytvárať koncové body pomocou „“REST API“. Náš dátový set je vo formáte JSON, ktorý Neo4j plne podporuje a má dostupnú aj „APOC“ knižnicu, v ktorej sa nachádzajú procedúry na importovanie „JSON“ formátu s následným mapovaním jednotlivých vrcholov a hrán, čo nám značne uľahčí počiatočný import dát. Taktiež Neo4j skvele spolupracuje s jazykom Java, v ktorom je aj napísaný, keďže plánujeme písať aplikačné rozhranie v jazyku Kotlin, ktorý vychádza z jazyku Java a obsahuje aj potrebný ovládač.

Ďalšie dôvody sú, že je to voľne dostupná databáza (open-source), vývojové prostredie je veľmi užívateľsky prívetivé, je to momentálne najpopulárnejšia grafová databáza, ktorá má širokú komunitu, čiže v prípade, že by sme narazili na nejaký problém, vieme sa spýtať na pomoc na rôznych internetových fórach alebo hľadať rady priamo v dokumentácií Neo4j, ktorá je veľmi kvalitne a rozsiahlo spracovaná, pričom návody sú často

vysvetlené na reálnom príklade. Neo4j využíva dopytovací jazyk Cypher, ktorý si bližšie popíšeme v poslednej podkapitole 1.9. Tento jazyk je podobný SQL jazyku, takže po predchádzajúcich skúsenostiach s jazykom SQL, na predmetoch databázových systémov a bakalárskej práci, nebol prechod na Cypher veľmi náročný.

1.9 Dopytovací jazyk Cypher a jeho porovnanie s SQL

Jazyk Cypher je navrhnutý tak, aby bol ľahko čitateľný a zrozumiteľný pre vývojárov, databázových profesionálov aj obchodných partnerov. Jeho jednoduchosť použitia vyplýva zo skutočnosti, že je v súlade so spôsobom, akým intuitívne popisujeme grafy pomocou diagramov. Cypher umožňuje používateľovi (alebo aplikácii konajúcej v mene používateľa) požiadať databázu o nájdenie údajov, ktoré zodpovedajú špecifickému vzoru. Samotný dotaz zvyčajne nešpecifikuje algoritmus, ktorý sa má použiť na vykonanie vyhľadávania a Neo4j automaticky vypracuje najlepší prístup k nájdeniu počiatočných uzlov a zodpovedajúcich vzorov. [21]

Ako väčšina dopytovacích jazykov, aj Cypher sa skladá z klauzúl. Najjednoduchšie dotazy pozostávajú z klauzuly MATCH, za ktorou nasleduje klauzula RETURN, ale nachádzajú sa tu aj ďalšie klauzuly, ktoré si stručne popíšeme.

MATCH - je jadrom väčšiny dopytov Cypher. Umožňuje nám špecifikovať vzory, ktoré bude Neo4j hľadať v databáze. Toto je primárny spôsob získavania údajov do aktuálnej sady väzieb. Je často spojený s časťou WHERE, ktorá pridáva obmedzenia k vzorom MATCH, čím ich robí špecifickejšími. MATCH sa môže vyskytnúť na začiatku dopytu alebo neskôr, po WITH.

RETURN - táto klauzula určuje, ktoré uzly, vzťahy a vlastnosti v zhodných údajoch by sa mali vrátiť. Za RETURN môžeme použiť ešte klauzulu AS , čo nám spraví alias k hodnotám, ktoré sa nám vrátia alebo LIMIT klauzulu, ktorá nám limituje počet nájdených výsledkov.

WITH - umožňuje reťazenie častí dotazu, pričom výsledky z jednej sa dajú použiť ako východiskové body alebo kritériá v ďalšej.

UNWIND - rozdelí list alebo array do riadkov.

WHERE - pridáva obmedzenia k vzorom v klauzule MATCH alebo OPTIONAL MATCH alebo filtruje výsledky klauzuly WITH.

CALL - vyhodnocuje pod dotaz, ktorý sa zvyčajne používa na spracovanie alebo agregácie.

A veľa ďalších, ktoré poznáme aj z SQL jazyka, napríklad: ORDER BY, CREATE, DELETE, SET, FOREACH, MERGE, DROP, JOIN a pod.

Syntax zápisu dopytov v Cypher si ukážeme na jednoduchom príklade, kde hľadáme filmy, v ktorých hral „Tom Hanks“. Názov uzla sa nachádza v guľatých zátvorkách, pričom má v sebe aj názov vlastnosti, ktorá sa nachádza v kučeravých zátvorkách a konkrétnejšie nám špecifikuje, čo hľadáme. Potom nasleduje názov vzťahu, ktorý je v hranatých zátvorkách, kde z každej strany vedie pomlčka, pri ktorej môže byť znak „<“ alebo „>”, ktorý nám reprezentuje smerovanie vzťahu.

```
MATCH (:Person {name: 'Tom Hanks'})-[:ACTED]->(movie:Movie)
RETURN movie
```

Ukážka kódu 1 - Jednoduchý dotaz na zobrazenie filmov kde hral "Tom Hanks"
[zdroj: vlastné spracovanie]

Na rozdiel od relačných SQL dotazov sú dotazy na grafovú databázu jednoduché na písanie a pochopenie. Databázy grafov majú často svoju vlastnú syntax pre takéto dotazy. V prípade Neo4j sa táto syntax riadi Cypher, ktorý je účelovo vytvorený na prechádzanie dátových vzťahov. Cypher dotazy sú oveľa jednoduchšie ako SQL dotazy a v skutočnosti môže byť dlhý SQL dotaz často komprimovaný na oveľa menej riadkov v Cypher.

Ukážka a porovnanie Cypher dotazu s SQL dotazom, kde chceme pre každého zákazníka, ktorý si kúpil produkt, aby nástroj na odporúčane prehľadal produkty, ktoré si zakúpili partneri, a potom ich navrhol aktuálnemu používateľovi. Klauzula WHERE odstraňuje produkty, ktoré si zákazník už kúpil, pretože nechceme odporúčať niečo, čo si zákazník už kúpil.

Cypher dotaz:

```
MATCH (u:Customer {customer_id:’customer-one’})-[:BOUGHT]->(p:Product)
  <- [:BOUGHT]-(peer:Customer)-[:BOUGHT]->(reco:Product)
WHERE not (u)-[:BOUGHT]->(reco)
RETURN reco as Recommendation, count(*) as Frequency
ORDER BY Frequency DESC LIMIT 5;
```

Ukážka kódu 2 - Zložitejší dotaz na odporúčanie produktov zákazníkovi v jazyku Cypher *[zdroj: https://neo4j.com/blog/sql-vs-cypher-query-languages/]*

SQL dotaz:

```
SELECT product.product_name as Recommendation, count(1) as Frequency
FROM product, customer_product_mapping, (SELECT cpm3.product_id, cpm3.customer_id
    FROM Customer_product_mapping cpm, Customer_product_mapping cpm2,
        Customer_product_mapping cpm3
    WHERE cpm.customer_id = 'customer-one'
    and cpm.product_id = cpm2.product_id
    and cpm2.customer_id != 'customer-one'
  and cpm3.customer_id = cpm2.customer_id
  and cpm3.product_id not in (select distinct product_id
    FROM Customer_product_mapping cpm
    WHERE cpm.customer_id = 'customer-one')
 ) recommended_products
WHERE customer_product_mapping.product_id = product.product_id
and customer_product_mapping.product_id in recommended_products.product_id
and customer_product_mapping.customer_id = recommended_products.customer_id
GROUP BY product.product_name ORDER BY Frequency desc
```

Ukážka kódu 3- Zložitejší dotaz v SQL jazyku na odporúčané produkty zákazníkovi *[zdroj: https://neo4j.com/blog/sql-vs-cypher-query-languages/]*

2 Cieľ a metodika práce

Naša práca je zameraná na vytvorenie aplikácie, ktorá využíva sémantické vyhľadávanie, čo znamená, že aplikácia bude využívať NoSQL typ databázy, ktorá bude v našom prípade grafová.

2.1 Cieľ

Našim cieľom bude vytvorenie prototypu aplikácie, ktorá bude využívať ako úložisko dát grafovú databázu Neo4j. Aplikácia bude slúžiť na vyhľadávanie receptov podľa užívateľom zvolených kritérií, kde bude mať možnosť napríklad vyhľadať recepty, ktoré obsahujú určité ingrediencie, alebo vyhľadať recepty podľa typu jedla, ako napríklad obed, večera alebo dezert, pričom aplikačné rozhranie bude napísané v programovacom jazyku Kotlin. Výberom grafovej databázy namiesto relačnej chceme demonštrovať jednoduchosť písania zložitejších dotazov v jazyku Cypher, kedy využívame v jednom dotaze niekoľko entít aj vzťahov naraz, pričom si môžeme na výstupe upraviť všetky tieto dáta a dať ich do jedného objektu, čo by bolo v prípade SQL jazyka veľmi zložité.

2.2 Metodika práce

Na začiatku sme si zvolili „user stories“, na základe ktorých sme postupne identifikovali potrebnú funkcionalitu v našej aplikácií a následne vytvorili aj tzv. scenáre, pomocou ktorých sme identifikovali entity a vzťahy v našej databáze. Po vytvorení databázy sme vypracovali jednotlivé funkcionality aplikácie.

2.2.1 User stories (používateľské príbehy)

V agilnom vývoji softvéru je príbeh používateľa stručným a jasným jazykovým vysvetlením funkcie alebo funkcie napísaných z pohľadu používateľa. Mnoho agilných odborníkov tiež popisuje príbeh používateľa ako najmenšiu jednotku práce na vývoji produktu, ktorá môže viesť k úplnému prvku funkčnosti používateľa. [23]

Produktové tímy sa rozhodnú rozdeliť vývojovú prácu na príbehy používateľov namiesto funkcií produktu alebo požiadaviek na produkt z niekoľkých dôvodov.

- Sú ľahko zrozumiteľné pre každého
- Predstavujú jednotky, ktoré sa zmestia do sprintu
- Pomôžu tímu zamerať sa skôr na skutočných ľudí než na abstraktné prvky
- Budujú stimul vývojovým tímom prostredníctvom pocitu pokroku [23]

Väčšina produktových tímov používa podobnú šablónu príbehu používateľa:

Ako [opis používateľa] chcem [funkčnosť], aby [prospech]. [23]

2.2.2 Modelovanie a tvorba databázy

Pri navrhovaní grafového databázového modelu sme postupovali cez tzv. scenáre, ktoré nám pomohli identifikovať jednotlivé entity a vzťahy medzi nimi ešte pred samotným procesom modelovania grafového modelu. Potom, ako sme si vytvorili scenáre, mohli sme začať s modelovaním našej databázy a následnou realizáciou.

Uzol sa často používa na reprezentáciu entít, ale môže tiež predstavovať aj iné komponenty domény v závislosti od prípadu použitia. Uzly môžu obsahovať vlastnosti, ktoré obsahujú páry údajov názov-hodnota. Uzlom možno priradiť roly alebo typy pomocou jedného alebo viacerých označení.[24]

Vzťah spája dva uzly a umožňuje nám nájsť súvisiace uzly údajov. Má zdrojový uzol a cieľový uzol, ktorý ukazuje smer šípky. Aj keď musíme uložiť vzťah v určitom smere, Neo4j má rovnaký výkon prechodu v oboch smeroch, takže môžete dotazovať vzťah bez zadania smeru. Jediným základným a konzistentným pravidlom v databáze grafov je „Žiadne prerušené odkazy“, ktoré zaisťuje, že existujúci vzťah nikdy neukáže na neexistujúci koncový bod. Keďže vzťah má vždy počiatočný a koncový uzol, nemôžete odstrániť uzol bez toho, aby ste odstránili aj súvisiace vzťahy. [24

Vlastnosti sú páry názov-hodnota údajov, ktoré môžeme uložiť v uzloch alebo vo vzťahoch. Väčšina štandardných typov údajov je uložených ako vlastnosti. Vlastnosti nám umožňujú ukladať relevantné údaje o uzle alebo vzťahu s entitou, ktorú popisuje. Často ich možno nájsť podľa prípadu použitia, aké otázky sa vyskytujú k údajom v grafe. [24]

Po tom, ako sme identifikovali uzly, vzťahy a vlastnosti, začali sme vytvárať model s následnou realizáciou databázy. Modely aj vytvorenie databázy popisujeme v podkapitole 3.2.

3 Výsledky práce

V poslednej kapitole si predstavíme výsledky našej práce, kde sme na začiatku uviedli „user stories“ a scenáre. Nasleduje model architektúry aplikácie a modelovanie databázy s jej realizáciou, kde popisujeme jednotlivé kroky s ukážkami z databázy. Tiež sme si stručne popísali priebeh čistenia dát, vďaka ktorému sme odstránili duplicitné uzly. V podkapitole 3.3 sme popísali jednotlivé vrstvy aplikačného rozhrania a v poslednej časti kapitoly sme uviedli aj vypracovanie prezentačnej vrstvy, ktoré sme realizovali prostredníctvom rámca Angular aj s ukážkami už hotovej aplikácie.

User stories

- Ako používateľ si chcem pozrieť všetky dostupné recepty
- Ako používateľ si chcem pozrieť detail receptu s návodom na prípravu, ingredienciami aj gramážami
- Ako používateľ si chcem vyhľadať recepty podľa názvu receptu
- Ako používateľ si chcem vyhľadať recepty podľa typu receptu
- Ako používateľ si chcem vyhľadať recepty podľa jednej alebo dvoch zvolených ingrediencií

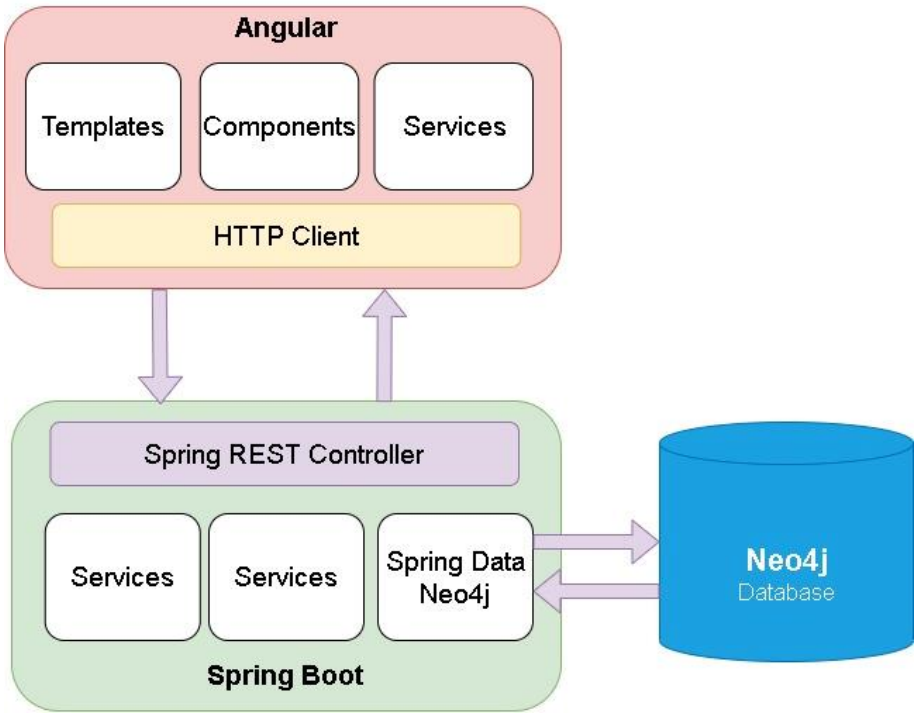
Scenáre

- Rizoto obsahuje ryžu a parmezán
- Rizoto má typ Obed/večera

Na základe týchto scenárov sme identifikovali uzly, ktorými sú „rizoto“, „ryža“, „parmezán“ a „obed-večera“. Ďalej sme zistili, že sa nám budú nachádzať v databáze dva vzťahy a to „obsahuje“ a „má_typ“. Uzly kategorizujeme a pridáme im tzv. štítky, takže rizoto bude „Recept“, ryža a parmezán bude „Ingrediencia“ a obed/večera bude "Typ_jedla“.

3.1 Architektúra aplikácie

Ešte pred začiatkom modelovania sme vybrali technológie, ktoré využijeme pri realizácií našej aplikácie. Na „back-end“ sme zvolili Spring Boot s využitím Spring Data Neo4j, ktoré poskytuje jednoduchú konfiguráciu a prístup k Neo4j grafovej databázy z aplikácií Spring a na „front-end“ sme použili Angular, ktorý posiela požiadavky na jednotlivé koncové body v REST Controlleri.



***Obrázok 7- Application architecture**, [zdroj: vlastné spracovanie]*

V nasledujúcej podkapitole si postupne prejdeme priebeh vytvorenia databázy aj s čistením dát.

3.2 Návrh databázy a jej realizácia

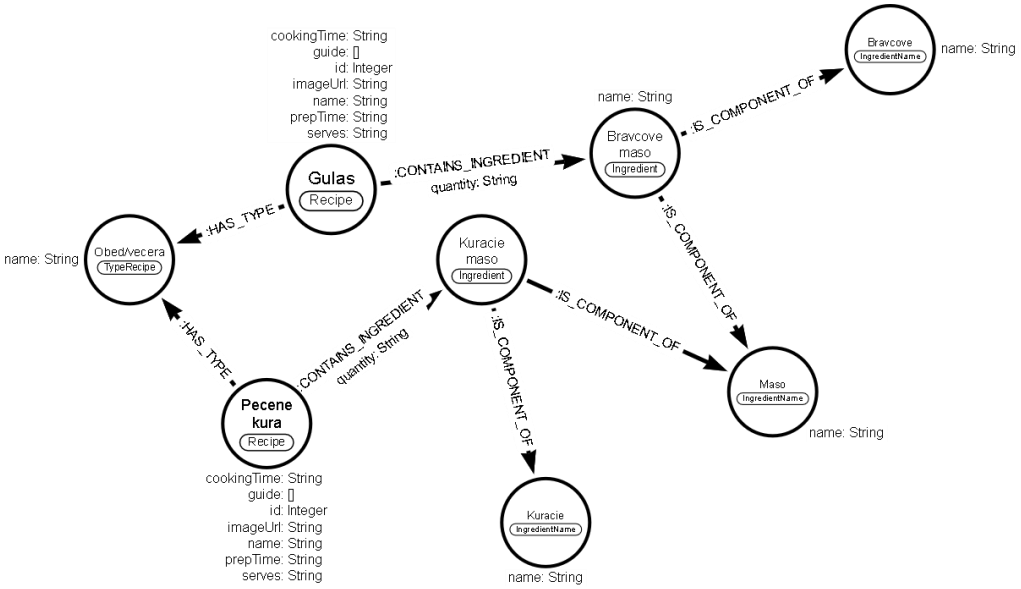
V nasledujúcej podkapitole postupne opisujeme priebeh vytvorenia databázy. Na začiatok si predstavíme modelovanie databázy, na základe ktorého potom vytvárame databázu, pričom je uvedený aj proces importovania údajov a aj následné čistenie importovaných údajov.

3.2.1 Modelovanie databázy

Modelovanie databázy sme realizovali cez desktopovú aplikáciu https://arrows.app/, ktorá nám veľmi uľahčila prácu svojou jednoduchosťou a prehľadnosťou. Z aplikácie si vieme exportovať nielen obrázok modelu, ale aj rôzne formáty ako napríklad Json, SVG

alebo Cypher dotaz, ktorý keď spustíme v prostredí Neo4j, tak nám vytvorí celú schému za nás.

Na nasledujúcom obrázku môžeme vidieť náš finálny databázový model.



***Obrázok 8 – Neo4j model**, [zdroj: vlastné spracovanie]*

V modeli môžeme vidieť uzly, ktoré sme si identifikovali na základe našich scenárov a to Recipe, Ingredient a TypeRecipe. Nachádza sa tu aj jeden ďalší uzol IngredientName, ktorý sme pridali do nášho modelu počas vývoja databázy na odstránenie duplicitných údajov, ale o tom si popíšeme v podkapitole 3.2.3. Do modelu sme zapracovali aj konkrétne dáta, aby bolo jasné ako medzi sebou interagujú uzly a na lepšie pochopenie modelu. Štítky jednotlivých uzlov sa nachádzajú pod konkrétnym názvom receptu alebo ingrediencie a podľa nich vieme rozlíšiť, ktoré uzly spadajú do rovnakej kategórie. Ďalej môžeme vidieť vzťahy medzi uzlami, ktoré sú napísané veľkými písmenami čo je Neo4j konvencia a aj šípku ich smerovania. Poslednou vecou v modely sú vlastnosti, či už uzlov alebo vzťahov, ktoré majú priradené aj dátové typy. Vzťah môže, alebo nemusí mať vlastnosť a v našom prípade môžeme vidieť, že vlastnosť má iba vzťah „CONTAINS_INGREDIENT“, a to konkrétne vlastnosť „quantity“, ktorá nám označuje gramáž alebo počet danej ingrediencie. Pri vlastnostiach môžeme vidieť, že vo väčšine prípadov máme dátový typ String, čo sa odporúča hlavne pred importovaním dát z neovereného zdroja, kde nemáme záruku, že sa napríklad na vlastnosti s dátovým typom Integer neobjaví nejaký znak.

3.2.2 Importovanie dát a vytvorenie databázy

Na začiatok sme potrebovali získať dátový set, ktorý importujeme do našej databázy. To sme realizovali cez „data scraping“, čo je vlastne extrahovanie dát. Na tento účel sme

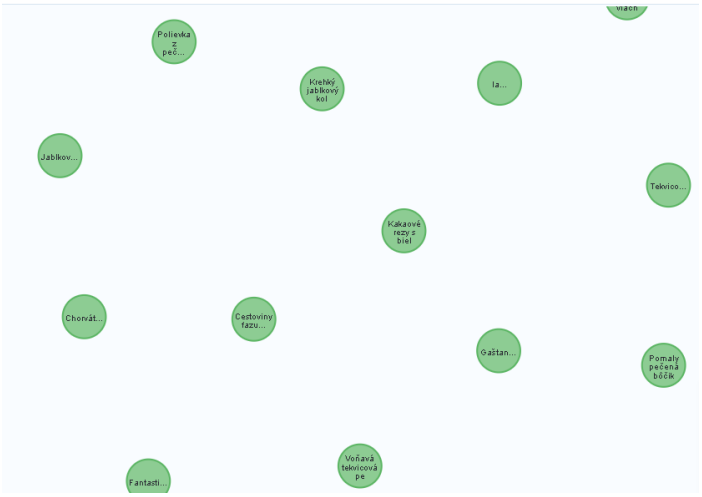
využili desktopovú aplikáciu „Parsehub“, kde sme si na nami vybranej stránke označili elementy, ktoré chceme extrahovať, nakonfigurovali automatické stránkovanie, finálny počet údajov a nakoniec aj formát, v ktorom sa nám vyextrahované údaje uložili. Neo4j podporuje načítanie CSV alebo JSON formátu, v našom prípade sme si zvolili načítanie JSON formátu.

Následne sme mohli začať s importovaním dát do databázy. Na importovanie dát má Neo4j k dispozícií knižnicu APOC (A Package Of Components), čo je súbor užívateľom definovaných funkcií a možno ich volať pomocou Cypher query. Na načítanie a na mapovanie údajov zo súboru použijeme procedúru „load.json“.

```
CALL apoc.load.json('file:///skusobne_3_recepty.json') YIELD value
UNWIND value.recipes as recipes
WITH recipes.id AS id,
recipes.name  AS title,
recipes.cooking_time AS cookingTime,
recipes.image_url AS imageUrl,
recipes.prep_time AS prepTime,
recipes.serves AS serves,
recipes.guide AS guide

MERGE (r:Recipe {id: id})
SET r.cookingTime = cookingTime,
r.imageUrl = imageUrl,
r.prepTime = prepTime,
r.serves = serves,
r.guide = guide,
r.name = title;
```

Ukážka kódu 4 - Vytvorenie uzlov "Recept" *[zdroj: vlastné spracovanie]*

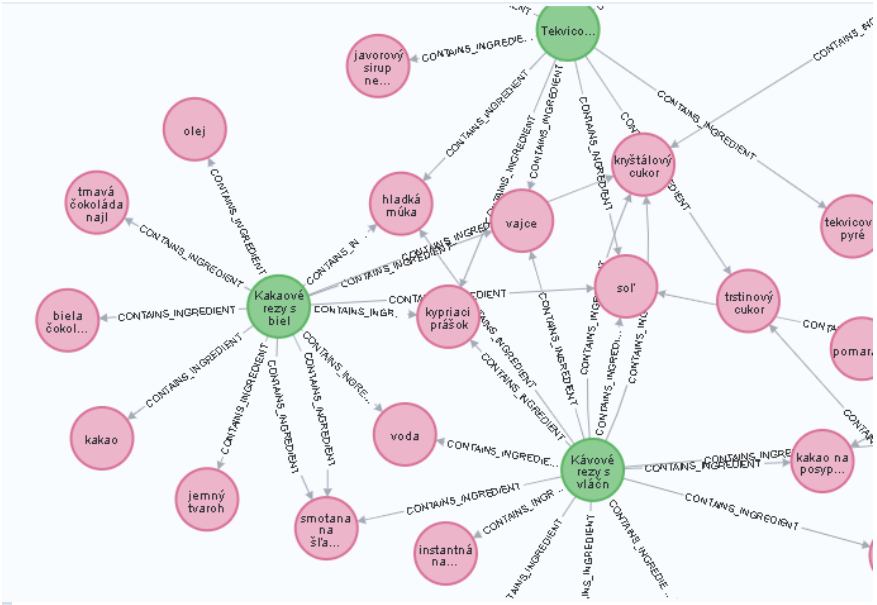


Obrázok 9- Recipe nodes, *[zdroj: vlastné spracovanie]*

V uvedenom Cypher dotaze môžeme vidieť na začiatku ako voláme procedúru „apoc.load.json“ cez kľúčové slovo CALL, pričom má jeden parameter a to cestu k Json súboru. Za cestou sa nachádza ďalšie kľúčové slovo YIELD, ktoré sa používa na explicitný výber dát, čo sú dostupné z výsledkových polí vrátia ako novo naviazané premenné z volania procedúry používateľovi, to znamená, že sa nám dáta zviažu s premennou „value“, ktorú ďalej využívame. V ďalšom riadku máme kľúčové slovo WIND pomocou ktorého transformujeme zoznam receptov na jednotlivé riadky. Nasleduje kľúčové slovo WITH pomocou ktorého môžeme manipulovať s výstupom predtým, ako bude odovzdaný nasledujúcim častiam dotazu, v našom prípade priraďujeme jednotlivé vlastnosti, ktoré obsahuje Recept k aliasom. V druhej časti dotazu sa nachádza kľúčové slovo MERG, ktoré nám spojí zhodné uzly k sebe, pričom v nasledujúcom riadku priraďujeme cez kľúčové slovo SET jednotlivé vlastnosti Receptu k aliasom, ktoré sme si vytvárali v prvej časti dotazu.

```
CALL apoc.load.json('file:///skusobne_3_recepty.json') YIELD value
UNWIND value.recipes as recipes
WITH recipes.id AS id,
    recipes.ingredients as ingredient
UNWIND ingredient as ingredientArray
MATCH (r:Recipe {id:id})
FOREACH (ingredient IN ingredientArray.name_ingredient |
 MERGE (i:Ingredient {name:ingredient})
 MERGE (r)-[:CONTAINS_INGREDIENT{quantity:ingredientArray.weight_ingredient}]->(i)
```

Ukážka kódu 5 - Vytvorenie uzlov "Ingredient" aj so vzťahmi "CONTAINS_INGREDIENT"*, [zdroj: vlastné spracovanie]*



***Obrázok 10 – Recipes with ingredients**, [zdroj: vlastné spracovanie]*

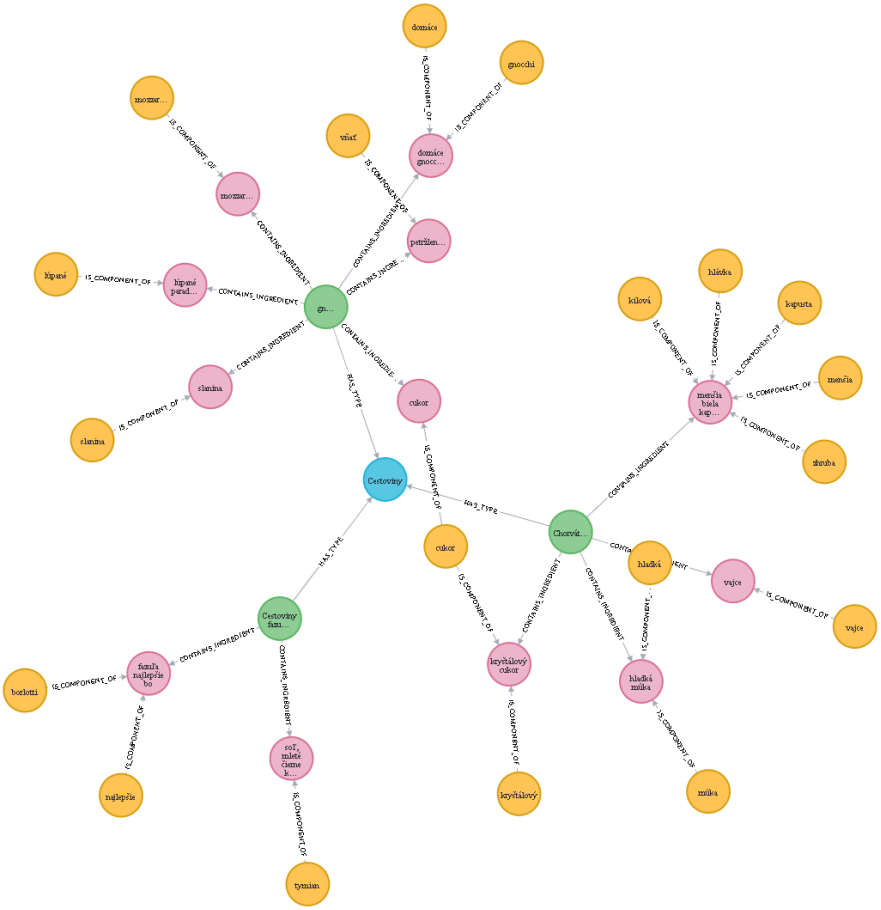
V ďalšom dotaze vytvárame uzly Ingrediencii. Znovu voláme procedúru so súborom a rozdeľujeme recepty do riadkov. V tomto dotaze už budeme používať aj uzly Receptov, čo znamená, že si pri WITH vložíme do aliasu aj vlastnosť „ID“ Receptu s ingredienciou. Nasleduje druhý WIND, ktorý potrebujeme na rozdelenie ingrediencii do riadkov, keďže sú vnorené v objekte Recept. Nasleduje kľúčové slovo MATCH, ktoré nám hľadá podľa „ID“ zhodné Recepty a následne použijeme FOREACH cyklus, ktorý nám pre každý záznam v liste „ingredientArray.name_ingredient“ zviaže dáta s vlastnosťou „name“ v uzly „Ingredient“ a taktiež vytvorí vzťah „CONTAINS_INGREDIENT“ s vlastnosťou „quantity“, ku ktorej priradí hodnotu z „ingredientArray.weight_ingredient“.

```
CALL apoc.load.json('file:///skusobne_3_recepty.json') YIELD value
UNWIND value.recipes as recipes
WITH recipes.id AS id,
    recipes.type_recipe as type_recipe
MATCH (r:Recipe {id:id})
FOREACH (type_recipe IN type_recipe |
 MERGE (t:TypeRecipe {name: type_recipe})
 MERGE (r)-[:HAS]->(t)
```

***Ukážka kódu 6 - Vytvorenie uzlov "Typ_recipe" aj so vzťahmi "HAS"** [zdroj: vlastné spracovanie]*

***Obrázok 11- Recipe type**, [zdroj: vlastné spracovanie]*

V poslednom dotaze v tejto podkapitole vytvárame uzly „TypeRecipe“ aj so vzťahmi „HAS“, ktoré sú prepojené s uzlami „Recipe“, pričom dotaz je podobný ako pri vytváraní „Ingrediecia“ uzlov.



***Obrázok 12 – Celá databáza** [zdroj: vlastné spracovanie]*

Na uvedenom obrázku môžeme vidieť ako vyzerá štruktúra vytvorenej databázy. Nastavili sme tam LIMIT na počet zobrazených uzlov kvôli prehľadnosti. Modrý uzol je TypReceptu, zelený je Recept, žltý je NázovIngrediencie a ružový je Ingrediencia. Uzol „NazovIngrediencia“ pridávame v nasledujúcej kapitole z dôvodu čistenia dát a odstraňovania duplicít. Po tom, ako sme vytvorili databázu, môžeme začať s čistením dát a odstraňovaním duplicitných údajov.

3.2.3 Čistenie dát

Keď máme databázu vytvorenú a údaje naimportované, môžeme začať s čistením dát, ktoré zahŕňa odstránenie duplicitných uzlov a odstránenie niektorých znakov v názvoch uzlov.

```
MATCH  (i:Ingredient)
WHERE i.name CONTAINS('(') OR i.name CONTAINS(')')
SET i.name = replace(replace(i.name, "(", ""), ")", "");
```

Ukážka kódu 7 - Odstránenie zátvoriek v názvoch receptov *[zdroj: vlastné spracovanie]*

V prvom rade odstránime zátvorky v názvoch ingrediencií, kde v našom dotaze hľadáme ingrediencie, ktoré obsahuje v názve „(“ alebo „)“ a nahradíme ich prázdnym Stringom cez metódu replace.

```
MATCH (i:Ingredient)
WITH i, split(ToLower(i.name), ' ') AS names
FOREACH (n IN names|
 MERGE (in:IngredientName {name:n})
 MERGE (in)-[:IS_COMPONENT_OF]->(i)
   );
```

Ukážka kódu 8 - Vytvorenie uzlov "IngredientName" a vzťahov "IS_COMPONENT_OF" *[zdroj: vlastné spracovanie]*

Nasleduje tokenizácia ingrediencií. Zobrali sme každú ingredienciu s využitím FOREACH cyklu zmenili všetky názvy aby boli malými písmenami cez metódu „ToLower“ a rozdelili slová podľa medzery. Zo samostatných slov sme vytvorili nové uzly s názvom „IngredientName“ a aj nové vzťahy „IS_COMPONENT_OF“, ktoré sme prepojili s ingredienciami.

```
MATCH (i:IngredientName)-[:IS_COMPONENT_OF]->(in)
WHERE i.name CONTAINS '-'
WITH i, in, split(i.name, '-') AS names
FOREACH (n IN names|
 MERGE (i2:IngredientName {name:n})
 MERGE (i2)-[:IS_COMPONENT_OF]->(in)
   )
DETACH DELETE i;
```

Ukážka kódu 9 - Odstránenie pomlčiek z názvov uzlov "IngredientName" *[zdroj: vlastné spracovanie]*

V uvedenom dotaze robíme podobné operácie ako v predchádzajúcom, ibaže tu už hľadáme v nových uzloch „IngredientName „ slová, ktoré obsahujú pomlčku, ktoré rozdelíme a zviažeme s uzlami „Ingredient“, pričom na konci zmažeme pôvodné uzly s pomlčkou cez „DETACH DELETE“, ktoré nám zmaže uzly aj so vzťahmi, ktoré obsahujú.

Toto bola základná tokenizácia. Teraz vykonáme ďalšie dotazy nad „IngredientName“ uzlami, ktoré nám pomôžu s odstránením duplicít. Existuje viacero prístupov, ktoré môžeme použiť na čistenie, ako napríklad nástroj ETL; alebo programové čistenie dát. V našom prípade budeme čistiť dáta cez jednotlivé dotazy v Neo4j.

V nasledujúcom dotaze chceme zmazať spojky alebo predložky, ako napríklad „v“/ „a“/„k“ a pod. Aby sme si zjednodušili prácu, budeme predpokladať, že väčšina týchto spojok alebo predložiek má jeden alebo dva znaky a zmažeme teda všetky uzly „IngredientName“, ktoré majú menej ako tri znaky.

```
MATCH (i:IngredientName)
WHERE size(i.name) < 3
DETACH DELETE i;
```

Ukážka kódu 10 - Odstránenie uzlov "IngredientName" kde je menej ako 3 znaky *[zdroj: vlastné spracovanie]*

V ďalšom dotaze zmažeme spojky alebo predložky, ako napríklad „al.“/ „nad“/„alebo“, ktoré majú tri a viac znakov, čiže nám prepadli prvým čistením a musíme ich zmazať explicitne, kedy všetky tieto reťazce vložíme do poľa a následne zmažeme tieto uzly.

```
MATCH (i:IngredientName)
WHERE i.name IN ['al.', 'nad', 'alebo']
DETACH DELETE i;
```

Ukážka kódu 11 - Odstránienie uzlov "IngredientName", v ktorých sa nachádza reťazec z poľa *[zdroj: vlastné spracovanie]*

Posledný dotaz máme zložitejší. V prvom rade hľadáme uzly „IngredientName“, ktoré nie sú rovnaké, čo je zabezpečené v WHERE klauzule prostredníctvom znakov nerovnosti „<>“, aby sme neporovnávali rovnaké uzly. Aby sme zredukovali počet dát, ktoré ideme porovnávať, kvôli rýchlejšiemu dopytu sme použili procedúru „apoc.text.sorensenDiceSimilarity“, ktorá porovnáva uzly a podľa zhodných písmen vyhodnocuje ako veľmi sú podobné, pričom zhoda môže nadobúdať hodnoty od 0 do 1, pričom 1 je úplná zhoda, v našom prípade sme dali podmienku, aby nám procedúra hľadala zhodu, ktorá je väčšia ako 0.6. K procedúre sme použili aj druhú podmienku, aby prvé dve začiatočné písmená uzlov boli rovnaké, na čo sme použili funkciu „left“, ktorá má dva parametre, prvý je názov uzlu a druhý, koľko porovnáva znakov.

Nakoniec sme vybrali uzly, v ktorých sa nachádza vzťah „IS_COMPONENT_OF“ a zlúčili z duplicitných uzlov iba jeden a ostatné zmazali. Na uvedených obrázkoch je ukážka sorensenDiceSimilarity procedúry na príklade s vajcami aj ako vyzerá vzorec podľa ktorého vyhodnocuje zhodu.

```
neo4j$ RETURN apoc.text.sorensenDiceSimilarity("VAJCE", "VAJCIA") AS output;
```

	output
1	0.6666666666666666

Obrázok 13 – Sorense algorithm*, [zdroj: vlastné spracovanie]*

$$DSC = \frac{2|X \cap Y|}{|X| + |Y|}$$

Obrázok 14 - Sorense algorithm clausele, *[zdroj: vlastné spracovanie]*

```
MATCH (n1:IngredientName),(n2:IngredientName)
WHERE id(n1) <> id(n2)
WITH n1, n2,
    apoc.text.sorensenDiceSimilarity(n1.name,n2.name) as sorensenDS
WHERE sorensenDS > 0.6 AND left(n1.name,2)=left(n2.name,2)
with n1, n2
MATCH (n2)-[:IS_COMPONENT_OF]->(i)
MERGE (n1)-[:IS_COMPONENT_OF]->(i)
DETACH DELETE n2;
```

Ukážka kódu 12 - Odstránenie uzlov "IngredientName" na základe Sorensovho algoritmu zhody *[zdroj: vlastné spracovanie]*

Treba spomenúť, že z procedúr, ktoré vyhodnocujú určité zhody medzi uzlami, sa nachádza v apoc knižnici viacero, napríklad Jaccardov algoritmus, ktorý meria podobnosť medzi listami dát, čiže podobnosť medzi celými uzlami. Taktiež spôsobov ako čistiť dáta je viacero a Neo4j ponúka širokú škálu procedúr, funkcií a nástrojov, ktorými to vieme docieliť. V našom prípade nemáme počet údajov taký obrovský, ako býva v reálnych prípadoch použitia a na naše študijné účely postačuje spomenuté čistenie dát.

3.3 Aplikačné rozhranie

Aplikačné rozhranie sme sa rozhodli naprogramovať s využitím technológie Spring Boot, pričom sme ho napísali v jazyku Kotlin. V nasledujúcej kapitole si prejdeme postupne všetky vrstvy, ktoré obsahuje naše aplikačné rozhranie, od modelov až po koncové body na radiči.

```
spring.neo4j.uri=bolt://localhost:7687
spring.neo4j.authentication.username=neo4j
spring.neo4j.authentication.password=12345
```

Ukážka kódu 13 - Prístupové údaje k databáze *[zdroj: vlastné spracovanie]*

Ešte pred písaním aplikačného kódu, sme si nastavili prístupové údaje k databáze, keďže Neo4j Community Edition vyžaduje na prístup k nej prihlasovacie údaje. Tieto prihlasovacie údaje sme nakonfigurovali v „application.properties“.

3.3.1 Model

V nasledujúcej podkapitole si predstavíme jednotlivé modely v našej aplikácií, ktoré sú viac-menej identické s entitami v databázovom modeli.

```
@Node
data class Recipe (

  @Id
  var id: Long? = null,
  var cookingTime: String? = null,
  var guide: MutableList<String>? = null,
  var imageUrl: String? = null,
  var name: String? = null,
  var prepTime: String? = null,
  var serves: String? = null,
  var ingredients: MutableList<String>?  = null,
  var quantity: MutableList<String>?  = null,
  var type: String?  = null,

  @Relationship(type = "CONTAINS_INGREDIENT", direction =
  Relationship.Direction.OUTGOING)
  var ingredientObjects: Collection<Ingredient>? = null,

  @Relationship(type = "HAS_TYPE", direction =
  Relationship.Direction.OUTGOING)
  var typesObjects: Collection<TypeRecipe>? = null,
  )
```

***Ukážka kódu 14 - Recipe model** [zdroj: vlastné spracovanie]*

V prvom riadku sa nachádza anotácia @Node, ktorá sa používa na označenie triedy ako riadenej entity. Konfiguruje tiež štítok Neo4j, ktorý je štandardne nastavený na názov triedy, ale môžeme definovať aj vlastný. V druhom riadku je označenie triedy „data“ , čo je trieda, ktorá sa používa na uchovávanie údajov alebo stavov a obsahuje štandardné funkcie ako napríklad toString(), equals(), hashCode() a pod. Trieda „data“ sa používa na deklarovanie triedy ako triedy údajov a musí obsahovať aspoň jeden primárny konštruktor s argumentom vlastnosti (val alebo var). Nasleduje anotácia @Id, ktorá nám označuje atribút

„id“ ako primárny, jedinečný kľúč. Ak by sme nemali v databáze zahrnuté ID, môžeme použiť anotáciu @GeneratedValue na generovanie jedinečného kľúča. Ďalej sú vypísané všetky atribúty aj s ich dátovými typmi, pričom treba spomenúť, že sa im musí nastaviť inicializačná hodnota, v našom prípade „null“. Nasleduje anotácia @Relationship, ktorá nám označuje vzťah medzi uzlami, v našom prípade medzi „Recipe“ a „Ingredient“, pričom anotácia má dva parametre, prvý je typ vzťahu, aký mám názov a druhý je smerovanie vzťahu, čo je v našom prípade „OUTGOING“ a tak isto aj druhý vzťah, ktorý je medzi uzlami „Recipe“ a „TypeRecipe“.

```
@Node
data class Ingredient(
    @Id
    var id: Long? = null,
    var name: String? = null,
    @Relationship(type = "CONTAINS_INGREDIENT")
    var recipes: MutableList<Recipe>? = null,
)
@Node
class TypeRecipe(
    @Id
    var id: Long? = null,
    var name: String? = null,
    @Relationship(type = "HAS_TYPE")
    var recipes: MutableList<Recipe>? = null,
)
```

Ukážka kódu 15 - Ingredient a TypeRecipe model *[zdroj: vlastné spracovanie]*

Podobne ako entitu „Recipe“, sme vytvorili aj „Ingredient“ aj „TypeRecipe“, ktoré majú len dva parametre a vzťah ktorý je medzi nimi a „Recipe“.

3.3.2 Repository

V tejto podkapitole si predstavíme jednotlivé metódy, ktoré sa nachádzajú v balíku „repository“.

```
interface RecipeRepository : Neo4jRepository<Recipe, Long> {

        @Query("MATCH (r:Recipe)-[c:CONTAINS_INGREDIENT]-
>(i:Ingredient)," +
                "(r:Recipe)-[:HAS_TYPE]->(t:TypeRecipe) \n" +
                "WITH  collect(i.name) as ingredients,\n" +
                "r, collect(c.quantity) as quantity, \n" +
                "t.name as type \n" +
                "WHERE (replace(replace(replace(replace(replace" +
                "(replace(replace(replace(replace(toLower(\n" +
                "r.name),'š','s'),'č','c'),'ž','z'),'ý','y'),'á','a')" +
                "','í','i'),'ť','t'),'é','e'),'ú','u')) " +
                "CONTAINS \$inputName " +
        "RETURN r{.*, ingredients: ingredients, quantity: quantity, type } LIMIT
200")

        fun getRecipesByContainingName(inputName: String): Collection<Recipe>
```

Ukážka kódu 16 - Metóda "getRecipesByContainingName" z „RecipeRepository“[*[zdroj: vlastné spracovanie]*

RecipeRepository rozširujeme o rozhranie„Neo4jRepository“, ktoré dedí z „CRUDRepository“ , čo znamená že máme k dispozícií určité metódy, aj bez toho, aby sme písali vlastné dopyty, napríklad „findAll()“, „saveAll()“, „findById()“ a pod. V „Neo4jRepository“ sa nachádzajú dva parametre a to „Recipe“ a „Long“, ktorý je podľa dátového typu „Id“ v triede „Recipe“. Nasleduje anotácia @Query, ktorá nám spustí dotaz vložený v nej.

Prvá metóda ,“getRecipesByContainingName()“ slúži na vyhľadávanie receptov podľa názvu receptu, na základe vloženého reťazca užívateľom. Na začiatku si vyberieme

všetky uzly cez MATCH., potom agregujeme dáta pre „ingredient“ uzly cez kľúčové slovo COLLECT, ktoré nám vytvorí pole názvov ingrediencií. Tak isto agregujeme aj „quantity“, čo je atribút vzťahu „CONTAINS_INGREDIENT“, ktorý nám spraví zoznam jednotlivých množstiev ingrediencií ku konkrétnemu receptu. Nasleduje podmienka WHERE, kde normalizujeme vstupnú premennú cez vnorené metódy „replace()“, ktoré slúžia na nahradenie písmen v reťazci, čo nám umožnilo odstrániť diakritiku zo vstupu, pričom v poslednej metóde „replace()“ sme ešte zavolali metódu „toLower()“, ktorá nám zmení všetky veľké písmená na malé. Ďalej voláme metódu CONTAINS, ktorá slúži na konkrétne vyhľadanie reťazca, ktorý sa nachádza v názvoch receptov a vraciame objekt recept, v ktorom sa nachádzajú všetky atribúty uzla „Recipe“, listy názvov ingrediencií a množstiev a ešte názov typu receptu, pričom sme na koniec dali ešte kľúčové slovo LIMIT, ktoré nám ohraničuje počet nájdených uzlov, v našom prípade je to „200“, keďže databáza obsahuje okolo 160 receptov a chceme zobraziť všetky dostupné recepty. Samotná metóda „getRecipesByContainingName()“ má jeden parameter, vstupný reťazec a návratovú hodnotu typu „Collection“, ktorá dokáže uchovávať sady, zoznamy a mapy. Tento dátový typ slúži iba na čítanie, ak by sme chceli robiť zmeny, zmenili by sme ho na MutableCollection, čo by nám umožnilo robiť editáciu.

Druhá metóda „getRecipesByContainingIngredientsTwo()“ nám slúži na vyhľadávanie receptov podľa názvov ingrediencií. Dopyt je veľmi podobný prvej až na parametre, ktoré sa používajú v metóde. V tomto prípade hľadáme namiesto v názvoch receptov, v názvoch ingrediencií a funkcia obsahuje dva parametre „nameOfIngredient1“ a „nameOfIngredient2“, pričom návratová hodnota je rovnaká ako pri prvej metóde, teda kolekcia typu „Recipe“.

```
@Query("MATCH (r:Recipe)-[c:CONTAINS_INGREDIENT]->(i1:Ingredient), \n" +
        "(r:Recipe)-[:CONTAINS_INGREDIENT]->(i2:Ingredient),\n" +
        "(r:Recipe)-[:HAS_TYPE]->(t:TypeRecipe) \n" +
        "WITH  collect(i1.name) as ingredients,\n" +
        "r,i1,i2, collect(c.quantity) as quantity, \n" +
        "t.name as type \n" +
        "WHERE (replace(replace(replace(replace(replace\n" +
        "(replace(replace(replace(replace(toLower(\n" +
        "i1.name),'š','s'),'č','c'),'ž','z'),'ý','y'),'á','a')\n" +
        " ,'í','i'),'ť','t'),'é','e'),'ú','u'))\n" +
        "CONTAINS \$nameOfIngredient1 AND\n " +
        "(replace(replace(replace(replace(replace\n" +
        "(replace(replace(replace(replace(toLower(\n" +
        " i2.name),'š','s'),'č','c'),'ž','z'),'ý','y'),'á','a')\n" +
        ",'í','i'),'ť','t'),'é','e'),'ú','u'))\n" +
        "CONTAINS \$nameOfIngredient2\n " +
    "RETURN r{.*, ingredients: ingredients, quantity: quantity, type } LIMIT
200")
    fun getRecipesByContainingIngredientsTwo(
        nameOfIngredient1: String,
        nameOfIngredient2: String,
    ): Collection<Recipe>
```

Ukážka kódu 17 - Metóda "getRecipesByContainingIngredientsTwo" z „RecipeRepository“[*zdroj: vlastné spracovanie]*

Tretia metóda je rovnaká ako predchádzajúca iba s jedným rozdielom, že v tejto metóde vyhľadávame iba podľa jednej ingrediencie. Túto metódu sme vytvorili pre prípad, že by užívateľ neuviedol druhý parameter, teda druhú ingredienciu vo vyhľadávaní.

```
        @Query("MATCH (r:Recipe)-[:CONTAINS_INGREDIENT]->(i:Ingredient)
" +
            "WITH r, collect(i.name) as ingredients " +
            "RETURN r{.*, ingredients: ingredients} LIMIT 200")
    fun getAllRecipes (): Collection<Recipe>
```

Ukážka kódu 18 - Metóda "getAllRecipes" z "RecipeRecpository" *[zdroj: vlastné spracovanie]*

Ďalšia metóda vráti všetky recepty, čiže nemá žiadny parameter a slúži nám na počiatočný zoznam receptov, kedy užívateľ ešte nezadal žiadne kritéria vo vyhľadávaní.

```
        @Query("MATCH (r:Recipe)-[c:CONTAINS_INGREDIENT]-
>(i:Ingredient), (r:Recipe)-[:HAS_TYPE]->(t:TypeRecipe) " +
            "WITH  collect(i.name) as ingredients, " +
            " r, collect(c.quantity) as quantity, " +
            "t.name as type " +
            "WHERE t.name = \$typeOfRecipe " +
"RETURN r{.*, ingredients: ingredients, quantity: quantity, type } LIMIT 200"
    )
    fun getRecipesByType(typeOfRecipe: String): Collection<Recipe>
```

Ukážka kódu 19 - Metóda "getRecipesByType" z „RecipeRepository“ *[zdroj: vlastné spracovanie]*

Predposledná metóda nám slúži na vyhľadávanie receptov podľa typu receptu. Obsahuje jeden parameter „typeOfRecipes“ a tiež má návratový typ kolekciu typu „Recipe“.

```
        @Query("MATCH(r:Recipe)-[c:CONTAINS_INGREDIENT]-
>(i:Ingredient), (r:Recipe)-[:HAS_TYPE]->(t:TypeRecipe) " +
            "WITH  collect(i.name) as ingredients, " +
            "r, collect(c.quantity) as quantity, " +
            "t.name as type " +
            "WHERE r.id = \$id " +
            "RETURN  r{.*,  ingredients:  ingredients,  quantity:  quantity,  type  }
SKIP 0 LIMIT 200"    )
    @Override
    override fun findById(id: Long): Optional<Recipe>
```

Ukážka kódu 20 - Metóda "findById" z "RecipeRepository" *[zdroj: vlastné spracovanie]*

Posledná metóda v „RecipeRepository“ nám vracia konkrétny recept. Je označená anotáciou @Override, čo nám signalizuje, že je to jedna z metód, ktorá sa nachádza v rozhraní„Neo4jRepository“ a teda už bola vytvorená a my jej dávame iba iné „správanie“, keďže v našom prípade nám počiatočná metóda nevracala požadovaný set výsledkov.

3.3.3 Service

V podkapitole Service si popíšeme ďalšiu vrstvu aplikácie. Je to implementačná vrstva nášho projektu, čo znamená, že v nej využívame metódy, ktoré sme popísali v podkapitole Respository a dodávame im obchodnú logiku, ak je treba, čo zahŕňa napríklad rôzne typy validácií.

```
@Service
class RecipeService @Autowired constructor(private val recipeRepository: RecipeRepository) {
    fun String.replace(vararg pairs: Pair<Char, Char>): String =
        pairs.fold(this) { acc, (old, new) -> acc.replace(old, new, ignoreCase = true) }
    fun normalizationString (normalizedString: String) =
        normalizedString.replace('š' to 's','č' to 'c','ž' to 'z','ý' to 'y',
        'á' to 'a','í' to 'i','ť" to 't','é' to 'e','ú' to 'u').lowercase()
```

Ukážka kódu 21 - Metódy "String.replace" a "normalizationString" z "RecipeService" *[zdroj: vlastné spracovanie]*

Na prvom riadku sa nachádza anotácia @Service, ktorá slúži ako indikátor pre Spring, že sa v tejto triede nachádza obchodná logika aplikácie a automaticky ju rozpozná. Za názvom triedy sa nachádza ďalšia anotácia @Autowired s konštruktorom, ktorá nám umožňuje automatickú injekciu závislostí, teda v našom prípade nám sprístupňuje funkcionalitu triedu „RecipeRepository“. Nasledujú dve metódy, pričom prvá metóda je tzv. „rozšírenie“ knižničnej metódy „String.replace()“, rozšírenia sú jednou z hlavných výhod jazyka Kotlin, kedy môžeme napríklad napísať nové funkcie alebo prepísať stávajúce pre triedu z knižnice tretej strany. Takéto funkcie možno volať bežným spôsobom, ako keby to boli metódy pôvodnej triedy. V našom prípade sme vylepšili funkciu „replace()“, ktorá má vstupný parameter typu „Pair“ v ktorom sa nachádza mapa znakov, pričom kľúč v mape je pôvodný znak a hodnota v mape je znak, ktorým sa má nahradiť.

Druhá funkcia využíva našu rozšírenú funkciu, pomocou ktorej odstraňujeme diakritiku zo vstupného parametra a meníme veľké písmena na malé. Túto funkciu využívame na normalizovanie vstupných parametrov vo vyhľadávacích metódach v triede

„RecipeService“, aby sa užívateľovi pri vyhľadávaní zobrazovali požadované výsledky receptov aj bez toho aby vyhľadával presné názvy s diakritikou alebo veľkými, či malými písmenami.

```
    @Transactional(readOnly = true)
    fun findByTitle(name: String) =
recipeRepository.getRecipesByContainingName(normalizationString(name))
```

Ukážka kódu 22 - Metóda "findByTitle" z "RecipeService" *[zdroj: vlastné spracovanie]*

Nad uvedenou metódou môžeme vidieť anotáciu @Transactional, ktorá slúži na špecifikovanie, že je iba na čítanie a umožní aplikáciám používajúcim kauzálne klastrovanie smerovať tieto transakcie na replikované servery a sústrediť prevádzku zápisu na hlavné servery, čím sa zlepší výkon. V našom prípade použitia to je síce zbytočné ale je to “best-practice” V samotnej metóde „findByTitle()“ môžeme vidieť využitie uvedenej normalizačnej funkcie „normalizationString()“. Telo metódy je veľmi stručné, ktoré iba volá funkciu „getRecipesByContainingName()“ z „RecipeRepository“.

```
    @Transactional(readOnly = true)
    fun findByNameOfIngredients(nameOfIngredient1: String,
nameOfIngredient2: String) : Collection<Recipe>
    {
        return if (nameOfIngredient2.isEmpty() || nameOfIngredient1.isEmpty())
{

recipeRepository.getRecipesByContainingIngredientsOne(normalizationString(nameOf
Ingredient1))
        } else
recipeRepository.getRecipesByContainingIngredientsTwo(normalizationString(nameOf
Ingredient1), normalizationString(nameOfIngredient2))
    }
```

Ukážka kódu 23 - Metóda „findByNameOfIngredients“ z „RecipeService“ *[zdroj: vlastné spracovanie]*

Ďalšia metóda „findByNameOfIngredients()“ nám slúži na vyhľadávanie receptov podľa ingrediencií. Má dva vstupné parametre typu reťazec a návratovú hodnotu Kolekciu typu „Recipe“, ktorú sme využívali aj pri metódach v „RecipeRepository“. Telo funkcie je rozdelené do dvoch vetiev, podľa splnenia podmienky, či nie je jeden zo vstupných parametrov prázdny reťazec. V prípade, že nie je, zavolá metódu

„getRecipesByContainingIngredientsTwo()“, ak je, ide po druhej vetve a zavolá metódu „getRecipesByContainingIngredientsOne()“. V oboch vetvách môžeme taktiež vidieť, že znovu využívame našu normalizačnú metódu na odstránenie diakritiky a zmenu veľkých písmen na malé pre lepšie vyhľadávanie.

```
@Transactional(readOnly = true)
fun findById(id: Long) = recipeRepository.findById(id)
@Transactional(readOnly = true)
fun findAllRecipes() = recipeRepository.getAllRecipes()
```

Ukážka kódu 24 - Metódy "findById" a "findAllRecipes" z "RecipeService" *[zdroj: vlastné spracovanie]*

Metóda „findAllRecipes()“ je bez parametrov a slúži na zobrazenie všetkých receptov a metóda „findId()“ nám slúži na zobrazenie detailu receptu, kedy po kliknutí na konkrétny recept, sa na základe „Id“ zavolá príslušný recept.

```
@Transactional(readOnly = true)
fun findAllRecipesByTypeDezert()  =
    recipeRepository.getRecipesByType(typeOfRecipe = "Dezert"
@Transactional(readOnly = true)
fun findAllRecipesByTypeObedVecera()  =
    recipeRepository.getRecipesByType(typeOfRecipe = "Obed/Večera")
@Transactional(readOnly = true)
fun findAllRecipesByTypeSlovenskaKuchyna()  =
    recipeRepository.getRecipesByType(typeOfRecipe = "Slovenská kuchyňa")
```

Ukážka kódu 25 - Metódy "findAllRecipesByTypeDezert", "findAllRecipesByTypeObedVecera" a "findAllRecipesByTypeSlovenskaKuchyna" z "RecipeService" *[zdroj: vlastné spracovanie]*

Posledné tri uvedené metódy, „findAllRecipesByTypeDezert()“ , „findAllRecipesByTypeObedVecera()“ a „findAllRecipesByTypeSlovenskaKuchyna()“ nám slúžia na zobrazenie receptov podľa kategórie, ktorú si užívateľ zvolil. Všetky tri metódy sú bez parametrov a majú jednoduché telo, kde volajú rovnakú metódu „getRecipesByType()“, pričom jej dávajú vstupný parameter reťazec, podľa názvu danej kategórie a na základe tohto reťazca potom príslušná funkcia vyhľadáva v databáze recepty.

3.3.4 Controller

V podkapitole Controller si popíšeme poslednú vrstvu aplikačného rozhrania, kde sa nachádzajú koncové body, ktoré používame na získavanie údajov cez HTTP požiadavky. V prvom riadku môžeme vidieť anotáciu @RestController, čo je špecializovaná verzia

ovládača, ktorá zahŕňa anotácie @Controller a @ResponseBody, takže zjednodušuje implementáciu radiča. Každá metóda, ktorá spracováva požiadavky automaticky serializuje návratové objekty do „HttpResponse“. Ďalej nasleduje anotácia @RequestMapping, ktorá nám označujú počiatočný vstupný bod do našej aplikácie, čo je v našom prípade "/api/recipes".

```
@RestController
@RequestMapping("/api/recipes")
class RecipeController @Autowired constructor(private val recipeService: RecipeService) {
    @GetMapping("/search/{str}")
    fun findRecipeByTitle(@PathVariable(value = "str") name: String) = recipeService.findByTitle(name)
```

Ukážka kódu 26 - Metóda "findRecipeByTitle" z "RecipeController" *[zdroj: vlastné spracovanie]*

Za názvom triedy sa nachádza ďalšia anotácia @Autowired s konštruktorom, ktorá nám umožňuje automatickú injekciu závislostí, teda v našom prípade nám sprístupňuje funkcionalitu triedu „RecipeRepository“.

Ako prvý koncový bod, definujeme metódu „findRecipeByTitle()“, ktorá vyhľadáva recepty na základe hľadaného reťazca používateľom. Aby sme k nej mohli pristupovať z webového prehliadača alebo klienta, označíme ju anotáciou @GetMapping, v ktorej nastavujeme cestu adresy URL, v tomto prípade teda "/search/{str}", pričom na konci cesty môžeme vidieť premennú „str“, do ktorej sa vkladá hľadaný reťazec znakov. Táto premenná je v metóde označená anotáciou @PathVariable, ktorá nám extrahuje šablónovú časť URI, čo predstavuje premenná „str“.

```
    @GetMapping("/{id}")
    fun getRecipeById(@PathVariable(value = "id") id: Long) = recipeService.findById(id)
    @GetMapping
    fun getAllRecipes() : Collection<Recipe> = recipeService.findAllRecipes()
```

Ukážka kódu 27 - Metódy "getRecipeById" a "getAllRecipes" z "RecipeController" *[zdroj: vlastné spracovanie]*

Druhá metóda, „getRecipeById“, je veľmi podobná prvej, pričom vyhľadáva konkrétny recept na základe „id“ a môžeme vidieť aj zmenu v ceste adresy, kde sme iba pridali ku počiatočnému koncovému bodu premennú „id“.

Ďalšia metóda,“getAllRecipes“, nám vracia kolekciu všetkých receptov a môžeme vidieť, že nie je špecifikovaná cesta URI za anotáciou @GetMapping, čo znamená, že zobrazí všetky recepty v počiatočnom bode, ktorý je "/api/recipes".

```
    @GetMapping("/filter")
    fun findRecipesByFilter(
        @RequestParam("n1", defaultValue = "") nameOfIngredient1: String,
        @RequestParam("n2", defaultValue = "") nameOfIngredient2: String,
    ): ResponseEntity<Collection<Recipe>> {
        return ResponseEntity.ok(
            this.recipeService.findByNameOfIngredients(
                nameOfIngredient1,
                nameOfIngredient2
            )
        )
    }
```

Ukážka kódu 28 - Metóda "findRecipesByFilter" z "RecipeController" *[zdroj: vlastné spracovanie]*

Metóda „findRecipesByFilter()“ nám slúži pre filtrovanie receptov na základe ingrediencií. Môžeme vidieť, že sa v nej nachádzajú dva parametre „n1“ a „n2“, ktoré inicializujeme na počiatočnú hodnotu prázdneho reťazca. Oba parametre sú označené anotáciou @RequestParam, ktorá nám signalizuje požadovaný parameter. Ak sú teda oba parametre naplnené pri volaní tohto koncového bodu, môžeme vidieť vo webovom prehliadači ako sa doplnia do cesty URI, napríklad „/filter?n1=cesnak&n2=slanina“. Návratový typ metódy je typu „ResponseEntity()“ v ktorej sa nachádza kolekcia typu „Recept“. „ResponseEntity()“ predstavuje celú odpoveď HTTP, v ktorej sa nachádza stavový kód, hlavička a telo. Vďaka tomu ho môžeme použiť na úplnú konfiguráciu HTTP odpovede, kde môžeme priradiť rôzne odpovede rôznym scenárom, napríklad „HttpStatus.BAD_REQUEST“ alebo „HttpStatus.OK“. Keďže sme použili „ResponseEntity()“ v návratovom typu, tak ju musíme aj vrátiť, čiže v našom prípade jej priradíme status „ok“ a vložíme do nej metódu „findByNameOfIngredients()“ aj s jej parametrami.

```
    @GetMapping("/type/Dezert")
    fun getRecipesByTypeDezert() = recipeService.findAllRecipesByTypeDezert()
    @GetMapping("/type/ObedVecera")
    fun getRecipesByTypeObedVecera() = recipeService.findAllRecipesByTypeObedVecera()
    @GetMapping("/type/SlovenskaKuchyna")
    fun getRecipesByTypeSlovenskaKuchyna() = recipeService.findAllRecipesByTypeSlovenskaKuchyna()
```

Ukážka kódu 29 - Metódy "getRecipesByTypeDezert", "getRecipesByTypeObedVecera" a "getRecipesByTypeSlovenskaKuchyna" z "RecipeController" *[zdroj: vlastné spracovanie]*

Posledné tri koncové body sú určené pre typy receptov, kde môžeme vidieť v ceste URI "/type/ " a za ním názov konkrétneho typu receptov, pričom všetky metódy sú bez parametrov a každá vracia príslušnú funkciu z „RecipeService“.

V nasledujúcej kapitole si ukážeme vypracovanie poslednej, prezentačnej, časti našej aplikácie.

3.4 Prezentačná vrstva

Po tom ako sme dokončili aplikačné rozhranie v Spring Boot a naprogramovali celú funkcionalitu, môžeme pristúpiť k záverečnej fáze našej aplikácie a to k prezentačnej vrstve“, ktorú sme sa rozhodli naprogramovať v rámci Angular. V ňom sme postupne naprogramovali jednotlivé metódy, ktoré sa odkazujú na koncové body v našom radiči, ktorý sme opísali v podkapitole 3.3.4. Angulár projekt sme rozdelili do dvoch hlavných komponentov a to do „recipe-list“ a „recipe-details“, ktoré si v tejto kapitole predstavíme a postupne prejdeme ako sme vypracovali jednotlivé funkcionality.

3.4.1 Recipe-list komponent a Recipe-details komponent

Najprv sme vygenerovali komponent „recipe-list“, deklarovali ho v „app.modelu.ts“ a priradili do „app-routing.module.ts“, kde sme mu priradili cestu URI „recipes“ { path: 'recipes', component: RecipeListComponent }. Následne sme vytvorili jednoduchý model „Recipe“, keďže naša aplikácia bude získavať a uchovávať entitu Recipe v databáze.

```
export class Recipe {
 id?: number;
 name?: string;
 imageUrl?: string;
 cookingTime?: string;
 prepTime?: string;
}
```

***Ukážka kódu 31 - Model "Recipe" v Angulari** [zdroj: vlastné spracovanie]*

Môžeme vidieť, že v uvedenom modely máme všetky potrebné atribúty, ktoré chceme aby model uchovával. Nasleduje implementácie služby „recipe-service“, čo je strednou vrstvou medzi službou REST a prezentačnou vrstvou.

```
export class RecipeService {
 private baseUrl = "http://localhost:8080/api/recipes"
 constructor(private http: HttpClient) { }
```

***Ukážka kódu 30 - "RecipeService" v Angulari** [zdroj: vlastné spracovanie]*

V triede „RecipeService“ sa vykonávajú všetky požiadavky na koncový bod „http://localhost:8080/api/recipes“, ktorý sme si vložili do privátnej premennej „baseUrl“. Do konštruktora sme potom vložili inštanciu HttpClient ako závislosť triedy.

```
getRecipes(): Observable<Recipe[]>{
  return this.http.get<Recipe[]>(`${this.baseUrl}`);
 }
```

***Ukážka kódu 32 - Metóda "getRecipes" v "RecipeService" v Angulari** [zdroj: vlastné spracovanie]*

Metóda „getRecipes()“ je základnou metódou, ktorá vracia všetky recepty po otvorení stránky. Môžeme vidieť, že voláme metódu „Observable()“, ktorá poskytuje podporu pre doručovanie správ medzi časťami našej aplikácie a je zodpovedná za spracovanie viacerých hodnôt alebo procesy spracovania udalostí, inak povedané, „Observable“ objekty sú zaregistrované a ostatné objekty ich zavolajú cez metódu

```
getRecipesDezert(): Observable<Recipe[]>{
  return this.http.get<Recipe[]>(`${this.baseUrl}/type/Dezert`);
 }
 getRecipesObedVecera(): Observable<Recipe[]>{
  return this.http.get<Recipe[]>(`${this.baseUrl}/type/ObedVecera`);
 }
 getRecipesSlovenskaKuchyna(): Observable<Recipe[]>{
  return this.http.get<Recipe[]>(`${this.baseUrl}/type/SlovenskaKuchyna`);
 }
```

Ukážka kódu 33 - Metódy "getRecipesDezert", "getRecipesObedVecera" a "getRecipesSlovenskaKuchyna" z "RecipeService" v Angulari *[zdroj: vlastné spracovanie]*

„Subscribe()“ , na čo zareagujú určitým spôsobom. V návratovej hodnote voláme cez „get“ z „http“ inštancie objekt „Recipe“, pričom ako parameter vkladáme premennú „baseUrl“.

Ďalšie tri metódy sú veľmi podobné našej základnej, ibaže v nich pridávame cestu aby smerovali na príslušné koncové body na radiči.

```
getRecipe(id: string | null): Observable<RecipeDetails> {
  return this.http.get<RecipeDetails>(`${this.baseUrl}/${id}`);
 }
 findByTitle(title: any): Observable<Recipe[]> {
  return this.http.get<Recipe[]>(`${this.baseUrl}/search/${title}`);
 }
 findByIngredient(ingredient1: any,ingredient2: any, ): Observable<Recipe[]> {
  return
this.http.get<Recipe[]>(`${this.baseUrl}/filter?n1=${ingredient1}&n2=${ingredient2}
`);
 }
```

Ukážka kódu 34 - Metódy "getRecipe", "findByTitle" a "findByIngredient" z "RecipeService" v Angulari *[zdroj: vlastné spracovanie]*

Metóda „getRecipe()“ nám slúži na zobrazenie detailu receptu a teda môžeme vidieť, že obsahuje objekt „RecipeDetails“, ku ktorému sa dostaneme v nasledujúcej podkapitole. Ďalšie dve metódy „findByTitle()“ a „findByIngredient()“ nám slúžia na vyhľadávanie a môžeme v nich vidieť zmenené príslušné cesty url, aby sa dostali na požadované koncové body v radiči pričom obsahujú aj parametre. V prípade „FindByTitle()“ to je „title“

a v prípade „findByIngredient()“ to sú „ingredient1“ a „ingredient2“. V jednej aj druhej metódu sú parametre typu „any“.

Nasleduje trieda „recipe-list.components.ts“, čo je stredná vrstva, kde si stručne popíšeme ďalšie metódy.

```
  constructor(private recipeService: RecipeService) {
 }
 ngOnInit(): void {
   this.recipeService.getRecipes().subscribe((data: Recipe[]) => {
     console.log(data);
     this.recipes = data;
   })
 }
```

Ukážka kódu 35 - Metóda "ngOnInit" z "Recipe-list.components" v Angulari *[zdroj: vlastné spracovanie]*

V prvom riadku môžeme vidieť konštruktor, v ktorom je inštancia triedy „RecipeService“, z ktorej budeme používať jednotlivé metódy. Nasleduje základná bez parametrická funkcia na zobrazenie všetkých receptov ngOnInit() v ktorej voláme metódu „suscribe()“ nad metódou „getRecipes()“. Subscribe je metóda v Angulari, ktorá spája pozorovateľa s pozorovateľnými udalosťami , čo sú „observable“ metódy . Vždy, keď dôjde k akejkoľvek zmene v týchto pozorovateľných položkách, spustí sa kód a sleduje výsledky alebo zmeny pomocou metódy odberu „Subscribe()“, ktorá je z knižnice rxjs. Subscribe() berie ako parametre tri metódy: „next()“, „error()“ a „complete()“. V našom prípade, okrem posledných dvoch metód, napĺňame iba prvý parameter „next“, kde máme parameter „data“ typu „Recipe[]“ a následne ho vkladáme do premennej „recipes“.

```
  getDezerts(): void {
    this.recipeService.getRecipesDezert().subscribe((data: Recipe[]) => {
      console.log(data);
      this.recipes = data;
    })
  }
  getObedVecera(): void {
    this.recipeService.getRecipesObedVecera().subscribe((data: Recipe[]) => {
      console.log(data);
      this.recipes = data;
    })
  }
  getSlovenskaKuchyna(): void {
  this.recipeService.getRecipesSlovenskaKuchyna().subscribe((data:    Recipe[])
=> {
      console.log(data);
      this.recipes = data;
    })
  }
```

Ukážka kódu 36 - Metódy "getDezerts", "getObedVecera" a "getSlovenskaKuchyna" z ""Recipe-list.components" v Angulari *[zdroj: vlastné spracovanie]*

```
searchTitle(): void {
  this.recipeService.findByTitle(this.searchString)
    .subscribe({
      next: (data) => {
        this.recipes = data;
        console.log(data);
      },
      error: (e) => console.error(e)
    });
 }
 searchByIngredient(): void {
  this.recipeService.findByIngredient(this.ingredient1, this.ingredient2)
    .subscribe({
      next: (data) => {
        this.recipes = data;
        console.log(data);
      },
      error: (e) => console.error(e)
    });
 }
```

Ukážka kódu 37 - Metódy "searchByTitle" a "searchByIngredient" z "Recipe-list.components" v Angulari
[zdroj: vlastné spracovanie]

Posledné dve metódy „searchTitle()“ a „searchByIngredient()“ volajú parametrické funkcie z „recipeService“, takže sme v tomto prípade využili aj druhý parameter metódy subscribe() a to „error“.

Nasleduje komponent „recipe-details“, pre ktorý sme si najprv vytvorili rozhranie „RecipeDetails“, kde sa nachádzajú všetky atribúty receptu, ktoré chceme používateľovi zobrazovať.

```
export interface RecipeDetails {
  id: number;
  cookingTime: string;
  guide: (string) [] | null
  imageUrl: string;
  name: string;
  prepTime: string;
  serves: String;
  ingredients: (string) [] | null
  quantity: (string) [] | null
  type: string
```

Ukážka kódu 39 - Rozhranie "RecipeDetails" v Angulari *[zdroj: vlastné spracovanie]*

Pre komponent „recipe-details“ sme nevytvárali službu, keďže v nej používame iba jednu metódu a to „“getRecipe(), ktorú sme si spravili v „recipe-service“.

```
 getRecipe(): void {
   const id = this.route.snapshot.paramMap.get("id");
   this.recipeService.getRecipe(id).subscribe(data => {
     this.recipeDetails = data;
   });
 }
```

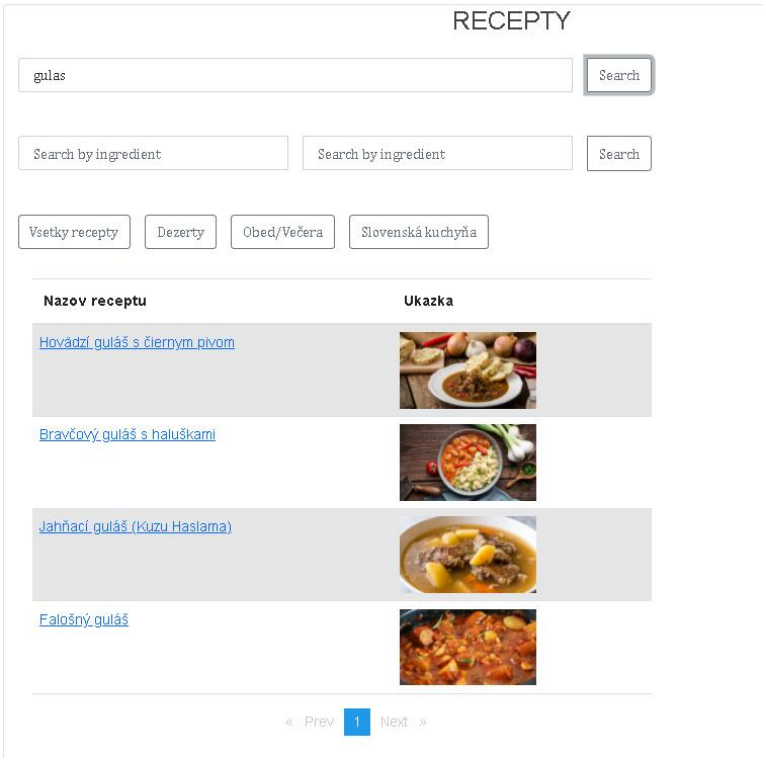
Ukážka kódu 38 - Metóda "getRecipe" v "RecipeDetails.component" v Angulari *[zdroj: vlastné spracovanie]*

V tejto, poslednej metóde „getRecipe()“ máme na začiatku vytvorenú konštantu „id“, do ktorej mapujeme konkrétne id receptu, pričom ho následne vkladáme ako parameter do funkcie „getRecipe()“ z triedy „RecipeService“. Nasleduje metóda „Subscribe()“, ktorú sme si už popísali a následne sa „data“ vložia do premennej „recipeDetails“, ktorá je typu „RecipeDetails“.

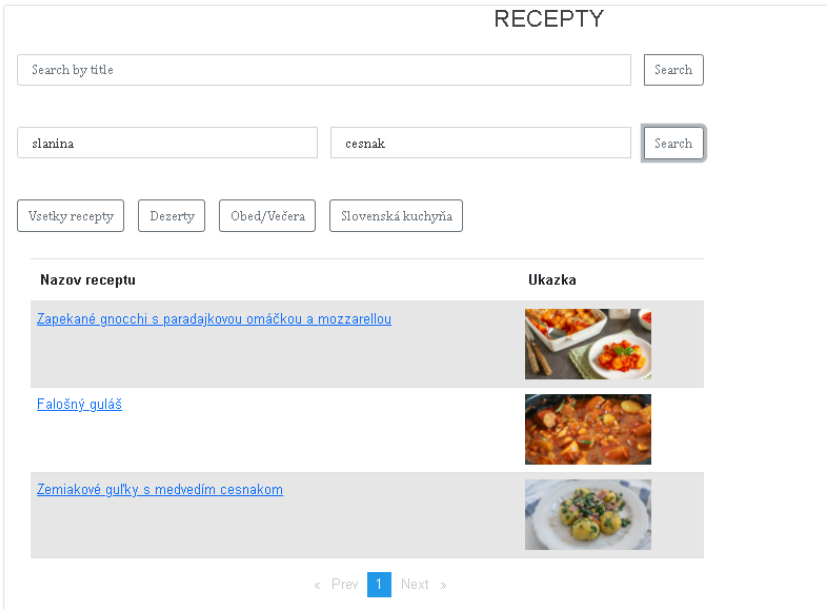
V tejto podkapitole sme si vytvorili dva komponenty „recipe-list“ a „recipe-details“, pričom sme podrobne popísali priebeh vypracovania každej z nich. Nasleduje posledná podkapitola v tejto kapitole, kde sú ukážky už hotovej aplikácie.

3.5 Ukážka aplikácia

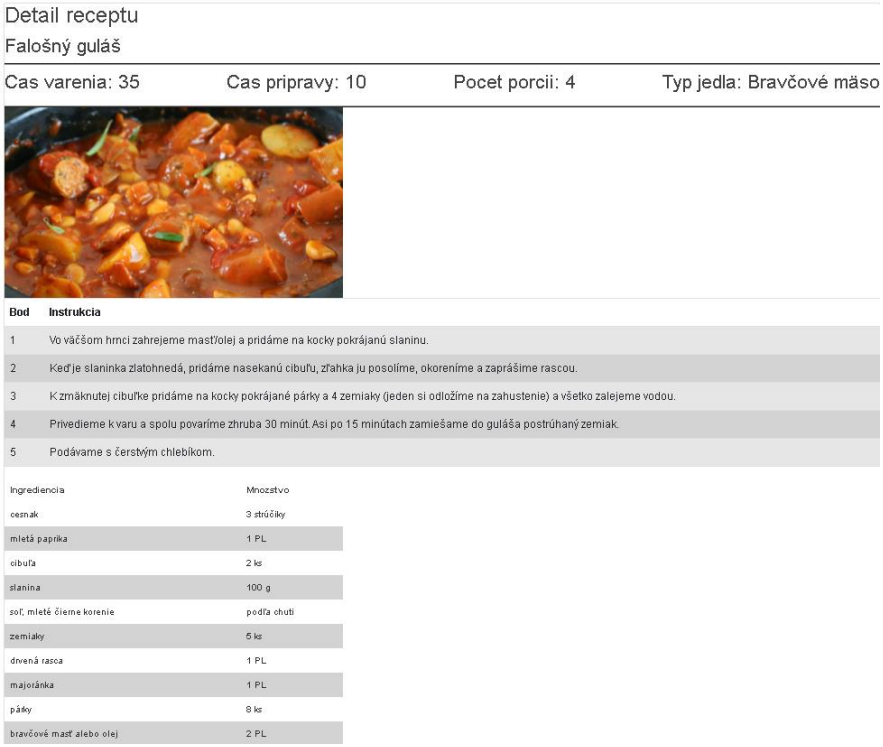
V poslednej podkapitole uvedieme ukážky vytvorenej aplikácie, kde môžeme vidieť zoznamy receptov, na základe zvolených kritérií aj detail receptu, ktorý sme si otvorili.



Obrázok 15 - Vyhľadanie receptov podľa názvu receptu *[zdroj: vlastné spracovanie]*



Obrázok 16 - Vyhľadanie receptov podľa zvolených ingrediencií *[zdroj: vlastné spracovanie]*



Obrázok 17 - Detail receptu *[zdroj: vlastné spracovanie]*

V poslednej podkapitole sme si ukázali ukážky aplikácie, kde máme na úvodnej stránke vyhľadávania receptov, či už podľa názvu receptu alebo ingrediencií. Môžeme taktiež vidieť tlačidlá, kde si môžeme vybrať kategóriu receptov „Dezert“, „Obed/Večera“ alebo „Slovenská kuchyňa“. V prípade, že by sme vyhľadávali recepty na základe určitých kritérií, kde by sme nevybrali žiadny recept a chceli by sme vidieť znova zoznam všetkých receptov, ktoré sa nachádzajú v databáze, tak máme tlačidlo zobraziť všetky recepty. Na obrázku č.17, môžeme vidieť detail konkrétneho receptu, ktorý sme si vybrali. Nachádzajú sa tu vlastnosti, ako napríklad „Čas varenia“, „Čas prípravy“, „Počet porcií“ alebo „Typ jedla. Nasleduje návod na uvarenie receptu a nakoniec aj potrebné ingrediencie aj s množstvami, ktoré si recept vyžaduje.

Záver

V našej práci sme vytvorili prototyp aplikácie, ktorej cieľom je vyhľadávanie receptov na základe zvolených kritérií.

Podarilo sa nám implementovať metódy, ktoré vytvorili funkcionalitu našej aplikácie. Medzi tieto funkcionality patrí napríklad vyhľadanie receptov pomocou zvolených surovín alebo podľa názvu receptov, čo nám zredukuje počet nájdených receptov na tie, o ktoré máme záujem

Aplikácia je veľmi jednoduchá a prehľadná pre bežného používateľa, a teda zľahčuje prácu pri vyhľadávaní receptov užívateľom.

Aplikáciu sme vypracovali s využitím technológie Spring Boot, o ktorom sme získali veľa nových poznatkov, čo je určite veľmi cenná skúsenosť do budúcnosti. Okrem toho sme si v práci osvojili aj používanie jazyka Cypher, ktorý slúži na písanie dopytov v grafovej databázy Neo4j.

V budúcnosti by sa dala aplikácia dala vylepšiť o sofistikovanejšie vyhľadávanie, v ktorom by sme predpokladali chybné vstupné reťazce od užívateľa, kde by aj pri chybných vstupoch dostal požadovaný set výsledkov. Momentálne sú vstupy ošetrené iba na diakritiku a veľké a malé písmená, čo je síce užitočné ale moderné vyhľadávacie nástroje by mali podľa nás mať aj nejakú pridanú hodnotu, ktorá zvyšuje komfort a efektívnosť vyhľadávania. Taktiež by bolo užitočné dopracovať funkcionalitu, kde si prihlásený užívateľ môže označiť obľúbené recepty a pri opätovnom prihlásení sa k ním jednoducho dostane.

Zoznam použitej literatúry

[1] SINGH, Naveen What are Graph databases and different types of Graph databases, [cit. 2021-11-29] Dostupné na internete: <https://singhnaveen.medium.com/what-are-graph-databases-and-different-types-of-graph-databases-369e5040a9d0>

[2] KING, Timothy. Four Common Graph Database Use Cases You Need to Know. In. solutionsreview.com, [cit. 2021-11-29]. Dostupné na internete: <https://solutionsreview.com/data-management/four-common-graph-database-use-cases-you-need-to-know/>

[3] WAYNER, Peter. What is a graph database?. In. venturebeat.com , [cit. 2021-11-29]. Dostupné na internete: <https://venturebeat.com/2021/02/08/what-is-a-graph-database/>

[4] Neo4j Staff, The Database Model Showdown: An RDBMS vs. Graph Comparison [cit. 2021-11-30] Dostupné na internete: <https://neo4j.com/blog/database-model-comparison/]>

[5] NEO4J, What is a Graph Database?, [cit. 2021-11-30] Dostupné na internete: < https://neo4j.com/developer/graph-database/>

[6] NEO4J, Cypher Shell, [cit. 2021-12-02] Dostupné na internete: <https://neo4j.com/docs/operations-manual/current/tools/cypher-shell/>

[7] NEO4J, Neo4j Browser, [cit. 2021-12-02] Dostupné na internete: <https://neo4j.com/docs/browser-manual/current>

[8] NEO4J, Neo4j Desktop, [cit. 2021-12-02] Dostupné na internete: <https://neo4j.com/docs/desktop-manual/current/>

[9] NEO4J, Neo4j Bloom, [cit. 2021-12-04] Dostupné na internete: <https://neo4j.com/docs/bloom-user-guide/current/>

[10] NEO4J, Neo4j ETL Tool - Interactive Relational Database Data Import, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/labs/etl-tool/>

[11] NEO4J, Neo4j APOC Library, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/developer/neo4j-apoc/>

[12] NEEDHAM, HODLER,Mark, Amy, Graph Algorithms, [cit.2021-12-04] Dostupné na internete: <https://go.neo4j.com/rs/710-RRC-335/images/Neo4j_Graph_Algorithms.pdf>

[13] NEO4J, Degree centrality, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/docs/graph-data-science/current/algorithms/degree-centrality/>

[14] NEO4J, Centrality Graph Algorithms, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/developer/graph-data-science/centrality-graph-algorithms/>

[15] SPRING, Spring Data Neo4j, [cit.2021-12-04] Dostupné na internete: <https://docs.spring.io/spring-data/neo4j/docs/current/reference/html/>

[16] ARANGODB, Distributed Iterative Graph Processing (Pregel), [cit.2021-12-04] Dostupné na internete: < https://www.arangodb.com/docs/stable/graphs-pregel.html>

[17] ORIENTDB, Overview, [cit.2021-12-04] Dostupné na internete: <https://orientdb.org/docs/3.0.x/misc/Overview.html>

[18] ORIENTDB, Users, Roles and Security, [cit.2021-12-04] Dostupné na internete: <http://orientdb.com/docs/3.0.x/gettingstarted/Users-Roles-Security.html>

[19] TITANDB, Overview, [cit.2021-12-04] Dostupné na internete: <http://espeed.github.io/titandb/>

[20] NEO4J, Neo4j APOC Library, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/developer/neo4j-apoc/>

[21] ROBINSON, WEBBER, EIFREM, Ian, Jim, Emil, HODLER,Mark, Amy, Graph Databases, [cit.2021-12-04] Dostupné na internete: <https://hura.hr/wp-content/uploads/2016/10/Graph_Databases_2e_Neo4j-5.pdf>

[22] HUNGER, BOYD, LYON, Michael, Ryan, Williem, RDBMS & Graphs: SQL vs. Cypher Query Languages, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/blog/sql-vs-cypher-query-languages/>

[23] PRODUCTPLAN, User Story, [cit. 2020-04-24] Dostupné na internete: <https://www.productplan.com/glossary/user-story/>

[24] NEO4J, Graph Modeling Guidelines, [cit.2021-12-04] Dostupné na internete: <https://neo4j.com/developer/guide-data-modeling/>