\section{Analýza problému}

%Súčasný stav riešenej problematiky

\noindent

\subsection{Grafové databázy}

\noindent Grafové databázy patria do kategórie NoSQL databáz

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

Grafové databázy využívajú model ktorý pozostáva z vrcholov, hrán, atribútov a značiek. Vrcholy obsahujú atribúty, a sú označené jednou alebo viacerými značkami. Tieto značky zoskupujú vrcholy, ktoré zastávajú rovnakú rolu v rámci aplikácie.

Hrany v grafových databázach spájajú vrcholy a budujú štruktúru grafu. Hrana grafu má vždy smer, názov, východzí vrchol, cieľový vrchol a cieľový vrchol. Fakt, že hany musia mať smer a názov pridáva to sémantickú prehľadnosť do grafu, ak zvolíme správne názov vieme rýchlo identifikovať štruktúru grafu a identifikovať význam vzťahov. Hrany môžu rovnako ako vrcholy obsahovať aj atribúty. Atribúty v hranách môžu byť praktické na pridanie kvalitatívnych dát (napr. váha, vzdialenosť) ku vzťahom, tieto dáta sa potom môžu použiť pri prehľadávaní grafu.

\subsection{Výkon grafovej databázy pri dotazoch na vzťahy}

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

\subsection{Vzťahy v relačných databázach}

\label{relational}

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

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

Vzťahy sa vyskytujú v modely relačných databáz no iba vo fáze návrhu, v reálnej implementácií to vyzerá tak, že keď pridávame vzťahy do relačnej databázy, veľmi rýchlo narastá komplexita a znižuje sa prehľadnosť dátového modelu a pridáva sa záťaž na databázu vznikom veľkých spájacích tabuliek a tabuliek s množstvom buniek s prázdnymi cudzími kľúčmi. \cite{graphDBs}

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

Na obrázku \ref{relational-relationships} vidíme prístup riešenia vzťahov v relačnej databáze.

\begin{figure}

\centering

\includegraphics[width=8cm]{img/relational-relationships.png}

\caption{Ukážka riešenia vzťahov v relačnej databáze}

\cite{graphDBs}

\label{relational-relationships}

\end{figure}

\subsection{Vzťahy v dokumentových databázach}

Dokumentové databázy patria rovnako ako grafové do kategórie NoSQL databáz. V dokumentovej databáze sa údaje ukladajú ako štruktúrované objekty ktoré sú adresované pomocou kľúčov.

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

Na obrázku \ref{document-relationships} vidíme prístup riešenia vzťahov v dokumentovej databáze.

\begin{figure}

\centering

\includegraphics[width=8cm]{img/document-relationships.png}

\caption{Ukážka riešenia vzťahov v dokumentovej databáze}

\cite{graphDBs}

\label{document-relationships}

\end{figure}

%TODO TEORIA key-value

%TODO TEORIA dokumentové databázy

\section{Špecifikácia}

\noindent Aplikácia ktorú sme sa rozhodli implementovať bude slúžiť na plánovanie a optimalizáciu trasy cestovateľa po svete. Dáta našej aplikácie budú teda pozostávať z používateľov, destinácií a prepojeniami medzi nimi. Štruktúra grafovej databázy je obzvlášť vhodná na návrh a implementáciu prepojení medzi jednotlivými mestami, keďže tento model má presne štruktúru klasického grafu.

%TODO SPECIFIKACIA

\begin{enumerate}

\item Funkcionálne požiadavky

\begin{enumerate}

\item Aplikácia bude umožňovať registráciu a prihlásenie používateľa

\item Pri registrácií sa budú vyžadovať prihlasovacie údaje: e-mail, heslo. Okrem toho sa bude vyžadovať zdanie mena a domácej destinácie.

\item Po prihlásení používateľa sa mu zobrazí obrazovka s mapou, zoznamom obľúbených destinácií, ktoré chce navštíviť a zoznam odporúčaných destinácií

\item Na mape bude vyobrazené používateľove domáca destinácia a všetky destinácie ktoré ma v zozname obľúbených destinácií. Po kliknutí na destináciu sa používateľovi otvorí príslušný riadok v zozname obľúbených.

\item V zozname obľúbených budú všetky destinácie, ktoré si používateľ pridal. Zoznam bude vo forme tabuľky riadok bude obsahovať Meno destinácie a lokalitu v ktorej sa destinácia nachádza. V riadku tiež bude tlačidlo na vymazanie destinácie z obľúbených a tlačidlo na zobrazenie detailov.

\item V detailoch obľúbenej destinácie bude zoznam ostatných ľudí ktorí danú destináciu majú tiež na zozname obľúbených

\item V detailoch obľúbenej destinácie bude zoznam možných trás z domácej destinácie používateľa do zvolenej destinácie.

\item V zozname odporúčaných destinácií budú destinácie ktoré majú v obľúbených používatelia, ktorí majú v obľúbených rovnaké miesta ako miesta, ktoré má v obľúbených prihlásený používateľ. Vynechané budú miesta, ktoré už prihlásený používateľ má obľúbených. Každá položka z odporúčaných sa bude dať jednoducho pridať do obľúbených prihláseného používateľa.

\item V aplikácií bude obrazovka s nastaveniami, na ktorej si bude človek môcť zmeniť heslo, domáce miesto.

\item V aplikácií bude obrazovka kde si bude používateľ vybrať nakoľko zo svojich obľúbených miest a nechať si vyrátať najkratšiu trasu z domáceho miesta cez všetky zvolené miesta a potom spať. (TSP)

\end{enumerate}

\item Nefunkcionálne požiadavky

\begin{enumerate}

\item Systém bude zrealizovaný na webovej platforme.

\item Aplikácia bude využívať natívnu grafovú databázu.

\item Aplikácia bude byť kompatibilná s webovými prehliadačmi Google Chrome Mozilla Firefox, Microsoft Edge, Microsoft Internet Explorer.

\item Užívateľské rozhranie systém musí byť plne použiteľné aj na mobilných telefónoch s OS Android a IOS.

\item Aplikácia bude implementovaný s použitím jazyka PHP a PHP frameworku.

\item Systém bude nasadený na virtuálnom serveri s operačným systémom Ubuntu 16.04.2 LTS poskytnutom Ústavom informatiky a matematiky FEI STU.

\end{enumerate}

\end{enumerate}

\section{Návrh}

%TODO NAVRH PO KAPITOLE

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

\subsubsection{Prípady použitia}

\noindent Prípady použitia popisujú interakcie medzi aplikáciou a používateľom, popisujú roly, ktoré rôzny hráči hrajú v týchto interakciách.

\begin{enumerate}

\item V prvom prípade použitia \ref{usecase-login} si bude môcť neprihlásený používateľ prezerať mapu a zoznam najpopulárnejších destinácií v našej aplikácií, ďalej sa môže prihlásiť zaregistrovať a požiadať o obnovu hesla.

\item Ďalší prípad použitia \ref{usecase-settings} opisuje nastavenia. Používateľ si bude môcť vyhľadať destináciu a zvoliť si ju ako svoje domáce miesto alebo si zmeniť heslo.

\item Tretí prípad použitia \ref{usecase-dash} opisuje akcie ktoré bude môcť používateľ vykonávať na hlavnej stránke. Bude môcť vyhľadávať destinácie a pridávať si ich do obľúbených. Destinácie ktoré má v obľúbených bude môcť mazať a zobraziť detaily trasy, ktorá k destinácií vedie. Ďalej si bude môcť prezerať ostatných používateľov ktorý majú rovnaké miesto v obľúbených a prejsť na zoznam obľúbených destinácií jedného s týchto používateľov.

\item Posledný prípad použitia \ref{usecase-tsp} popisuje obrazovku TSP. Na tejto obrazovke bude môcť používateľ prezerať svoju mapu optimálnej cestovateľskej trasy, pridávať a odoberať z trasy destinácie zo zoznamu obľúbených.

\end{enumerate}

\begin{figure}

\centering

\includegraphics[width=10cm]{uml/usecase-login.pdf}

\caption{Prípad použitia - Registrácia, autentifikácia a nastavenia}

\label{usecase-login}

\end{figure}

\begin{figure}

\centering

\includegraphics[width=10cm]{uml/usecase-settings.pdf}

\caption{Prípad použitia - Registrácia, autentifikácia a nastavenia}

\label{usecase-settings}

\end{figure}

\begin{figure}

\centering

\includegraphics[width=12cm]{uml/usecase-dash.pdf}

\caption{Prípad použitia - dashboard}

\label{usecase-dash}

\end{figure}

\begin{figure}

\centering

\includegraphics[width=10cm]{uml/usecase-tsp.pdf}

\caption{Prípad použitia - tsp}

\label{usecase-tsp}

\end{figure}

\subsubsection{Štruktúra databázy}

\noindent Na obrázku č. \ref{database-diagram} našej databáze sa budú nachádzať dve hlavné entity: entita používateľa \textit{appUser} a entita destinácie \textit{appPlace}. Entita používateľa bude obsahovať iba základné atribúty ako meno a atribúty potrebné pre autentifikáciu používateľa email, a heslo.

Entita destinácie bude obsahovať atribúty potrebné na identifikáciu tejto destinácie: unikátny názov a krátky názov vhodný na zobrazovanie a atribúty potrebné na lokalizáciu destinácie: zemepisná šírka a zemepisná dĺžka.

Keďže použijeme grafovú databázu nebude potrebné pridávať cudzie kľúče reprezentujúce ich vzťahy.

Ďalej bude naša databáza obsahovať hrany: trasa \textit{ROUTE}, obľúbené \textit{FOLLOWS}, a zoznam pre optimalizáciu trasy \textit{TSP}.

Hrana trasy bude vždy viesť od jedného miesta k druhému reprezentuje zoznam trás ktoré budú viesť od jednej destinácie k druhý. Bude obsahovať atribút \textit{minPrice} hovoriaci o cene najlacnejšej trasy medzi spomínanými destináciami a tribút \textit{routes} obsahujúci serializované detaily všetkých trás.

%todo this is an optimalization since neo4j sasves oly one attribute in graph

Hrany \textit{FOLLOWS} a \textit{TSP} budú vždy smerovať od používateľa k destinácií a nebudú obsahovať žiadne atribúty.

\begin{figure}

\centering

\makebox[\textwidth]{\includegraphics[width=14cm]{uml/database.pdf}}

\caption{Štruktúra databázy}

\label{database-diagram}

\end{figure}

\subsection{Výber grafovej databázy}

\noindent Vybrali sme si troch najpopulárnejších predstaviteľov grafových databáz podľa rebríčka DB-Engines.com\cite{dbrank}. V nasledujúcej časti v skratke priblížime históriu každej grafovej databázy ich výhody, nevýhody pre naše použitie.

%TODO citations

\subsubsection{NEO4J}Prvá verzia Neo4J bola vydaná v roku 2007 od vtedy sa stala dlhodobo najpoužívanejšou grafovou databázou. Je vyvíjaná Neo Technology, Inc. Neo4j je ponúkaná v dvoch variantoch': Neo4j Community je open source (GPLv3) grafová databáza obsahujúca všetky základné funkcií (Ďalej budeme spomínať len túto verziu). A Neo4j Enterprise edícia, ktorá má rozšírené funkcie ako shardovanie cache pamäte, rozšírené monitorovanie a zálohovanie za behu.

Medzi hlavné výhody neo4j patrí to, že je to dlhodobo najrozšírenejšia grafová databáza, má dobrú dokumentáciu a natívne podporuje OGM (mapovanie objektov na graf) mapovanie dát na objekty. Práve táto natívna podpora OGM umožnila vzniku prehľadného a nástroja NeoEloquent, ktorý sa dá využiť na integrácie Neo4J do frameworku Laravel.

Ďalšou výhodou je dopytovací jazyk Cypher, ktorý táto databáza môže využívať. Tento jazyk má širokú, prehľadnú dokumentáciu.

Nevýhodou je, že podporuje iba grafový model ukladania údajov, nepodporuje ukladanie dokumentov ani key-value úložisko.

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

Výhodou OrientDB je podpora okrem grafového modelu ukladania dát aj dokumentové a key-value úložisko a podľa nezávislých testov \cite{dbspeed} je v niektorých testoch až desať krát rýchlejšie ako Neo4J.

Nevýhodou OrientDB je horšia dokumentácia a nástroj na integráciu to frameworku Laravel \textit{oriquent} \cite{oriquent} nepodporuje zapisovanie a čítanie z atribútov hrán.

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

Toto riešenie je pre naše použitie nevhodné, lebo Titan nepodporuje ovládače pre PHP, celé riešenie je skôr orientované na veľké distribuované enterprise riešenia.

\section{Implementácia}

\noindent V tejto kapitole si priblížime proces

\subsection{Použité technológie}

\subsubsection{Laravel framework}

\noindent Laravel Framework je komplexný voľne šíriteľný framework. Tento framework je od roku 2015 najpopulárnejší PHP framework. Medzi jeho hlavné výhody patrí použitie relatívne novej verzie PHP 5.4, ktorá obsahuje technológie, ktoré v minulosti v PHP chýbali ako menové priestory a anonymné funkcie. Ďalej obsahuje veľmi silný nástroj Eloquent ORM pre objektovo relačné mapovanie databázy, Blade šablónovací nástroj na rýchlu tvorbu dynamického obsahu.

Komplexita Frameworku Laravel je však aj jeho hlavnou nevýhodou, nie je vhodný na menšie projekty. V rýchlosti patrí medzi priemer PHP frameworkov.

\subsubsection{NeoEloquent OGM}

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

\subsubsection{Mapbox.js}

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

Túto knižnicu postupne nahrádza Mapbox-gl.js vyvíjaná tou istou firmou, ktorá využíva technológiu WebGL na lepšiu akceleráciu vykresľovania, no v čase začiatku projektu táto knižnica ešte nepodporovala vyššie spomenuté zoskupovanie bodov.

\subsubsection{Handlebars.js}

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

\subsubsection{Rome2Rio Api}

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

\subsection{Inštalácia a konfigurácia Laravel Framework-u}

\noindent Na inštaláciu frameworku Laravel sme použili nástroj pre správu PHP balíkov \textit{Composer}. Pomocou tohto nástroja sme nainštalovali balík \textit{laravel/installer} \cite{laravel}, následne sme požitím príkazu \textit{laravel new projekt} vytvorili priečinok so základnou inštaláciou frameworku.

Jediné nastavanie ktoré bolo po tom potrebné bolo už len nastavenie názvu aplikácie, a databázy, o tom ďalej v sekcii \ref{installneo}.

\subsection{Inštalácia a konfigurácia databázy neo4j}

\label{installneo}

\cite{neoinstall}

\noindent Aby sme mohli nainštalovať databázu Neo4j musíme si najprv do systému pridať repozitár Neo Technology, potom je nám k dispozícií na inštaláciu balík neo4j. Po inštalácií je nám ihneď dostupné administračné rozhranie databázy na adrese: \textit{localhost:7474}. Pri prvom prihasení sme vyzvaní na zmenu hesla.

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

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

\begin{algorithm}

\lstset{

language=PHP,

basicstyle=\small\sffamily,

frame=none,

numbers=left,

xleftmargin=5.0ex,

numberstyle=\tiny,

stepnumber=1,

showstringspaces=false,

keywordstyle=\color{blue}\bfseries

}

\lstset{emph={% Adjust any special keywords

printf%

},emphstyle={\color[rgb]{1,0,0}\bfseries}%

}%

\begin{lstlisting}

$app->register('Vinelab\NeoEloquent\NeoEloquentServiceProvider');

...

'default' => env('DB\_CONNECTION', 'neo4j'),

'connections' => [

'neo4j' => [

'driver' => 'neo4j',

'host' => env('DB\_HOST', 'neo4j'),

'port' => env('DB\_PORT', 'neo4j'),

'username' => env('DB\_USERNAME', 'neo4j'),

'password' => env('DB\_PASSWORD', 'neo4j'),

],

],

\end{lstlisting}

\caption{Ukážka algoritmu}

\label{euclid}

\end{algorithm}

\subsection{OGM}

\label{ogm}

\noindent Keďže framework Laravel natívne obsahuje len ovládače pre relačné databázové ovládače a nástroj na objektovo relačné mapovanie Eloquent Použili sme open source balík NeoEloquent, ktorý obsahuje ovládač pre databázu Neo4J a zároveň rozširuje dátový model o prvky grafovej databázy. NeoEloquent umožňuje manipuláciu s vrcholmi aj hranami v Neo4J. Manipulácia s vrcholmi je rovnaká ako s entitami v relačnej databáze, NeoEloquent umožňuje vytvárať perzistentné objekty, upravovať ich a vyhľadávať v nich. Pri práca s hranami je mierne odlišná. Najprv treba zadefinovať ktorý objekt môže mať aké vzťahy, tieto vzťahy treba unikátne identifikovať ich typom a kardinalitou. Tento vzťah vraciame ako návratovú hodnotu funkcie daného objektu. Vrátený objekt sa správa podobne ako štandardná entitná trieda Eloquent.

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

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

\begin{algorithm}

\lstset{

language=PHP,

basicstyle=\small\sffamily,

frame=none,

numbers=left,

xleftmargin=5.0ex,

numberstyle=\tiny,

stepnumber=1,

showstringspaces=false,

keywordstyle=\color{blue}\bfseries

}

\lstset{emph={% Adjust any special keywords

printf%

},emphstyle={\color[rgb]{1,0,0}\bfseries}%

}%

\begin{lstlisting}

namespace App;

class User extends \NeoEloquent implements Authenticatable {

// Jeden používateľ môže mať v obľúbených viac miest

public function follows() {

return $this->hasMany('App\Place', 'FOLLOWS');

}

// Jeden používateľ má jedno miesto ako domáce

public function home() {

return $this->hasOne('App\Place', 'HOME');

}

...

}

...

class Place extends \NeoEloquent {

// Inverzný vzťah - jedno miesto má v obľúbených viac používateľov

public function followers(){

return $this->belongsToMany('App\User','FOLLOWS');

}

...

}

\end{lstlisting}

\caption{Ukážka triedy neoEoquent}

\label{codeauth}

\end{algorithm}

\subsection{Autentifikácia}

\noindent Jednou zo silných stránok Frameworku Laravel je práve autentifikácia. Pre vytvorenie základnej funkcionality registrácie, prihlasovania a obnovenia zabudnutého hesla stačí použiť Artisan - konzolu frameworku príkaz \textit{php artisan make:auth} , ktorá vytvorí URL cesty, obrazovky, triedu používateľa a triedy obsluhujúce túto funkcionalitu. My sme potrebovali použiť vlastnú triedu používateľa, ktorá dedí od nášho balíka NeoEloquent, na implementáciu autentifikácie stačilo neimplementovať rozhranie Authenticatable, pridať do triedy používateľa pole skrytých a verejných atribútov, a použiť charakteristiku 'AuthenticableTrait' a autentifikácia fungovala rovnako ako s relačnou databázou vďaka tomu, že NeoEloquent implementuje podobnú funkcionalitu ako natívne ORM Eloquent.

\begin{algorithm}

\lstset{

language=PHP,

basicstyle=\small\sffamily,

frame=none,

numbers=left,

xleftmargin=5.0ex,

numberstyle=\tiny,

stepnumber=1,

showstringspaces=false,

keywordstyle=\color{blue}\bfseries

}

\lstset{emph={% Adjust any special keywords

printf%

},emphstyle={\color[rgb]{1,0,0}\bfseries}%

}%

\begin{lstlisting}

namespace App;

use Illuminate\Contracts\Auth\Authenticatable;

use Illuminate\Auth\Authenticatable as AuthenticableTrait;

class User extends \NeoEloquent implements Authenticatable {

use AuthenticableTrait;

// pole verejných atribútov

protected $fillable = [

'name', 'email', 'password', 'tspCache'

];

// pole skrytých atribútov

protected $hidden = [

'password', 'remember\_token',

];

...

}

\end{lstlisting}

\caption{Ukážka autentifikovateľnej triedy}

\label{codeauth}

\end{algorithm}

\subsection{API}

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

\subsection{Rome2rio API}

\noindent Všetky údaje o destináciách a trasách berieme z API Rome2Rio. Pomocou Autocomplete API umožňujeme používateľovi pridávať destinácie. Používateľ zadáva písmená do autocomplete textového poľa na stránke toto pole posiela dotazy na Autocomplete API a ono vracia pole objektov s miestami. Tieto objekty obsahujú informácie o type destinácie(obec, mesto, región, štát, letisko,), geografickú polohu, názov v dlhom a krátkom tvare a kanonický názov. Používateľ si potom vyberie jenu z destinácií a príslušný objekt sa zašle na náš server. Na unikátnu identifikáciu objektu používame kanonický názov, ktoré je podľa dokumentácie unikátnym identifikátorom miesta.

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

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

Pri implementácií sme zistili, že pri niektorých trasách API vrátilo jednu z možných trás s nulovou cenou. Túto trasu sme ignorovali.

%TODO KIWI.com ANOTHER SOURCES Problémom takejto integrácie Rome2Rio API do našej aplikácie je, že ak by sme chceli brať údaje z iných zdrojov boli by nutné zmeny v štruktúre ukladania dát aby boli

\subsection{TSP}

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

Pre jednoduchosť a spoľahlivosť sme naimplementovali prístup rekurzívneho prehľadania všetkých možnosti trás. Naimplementovali sme rekurzívnu funkciu v PHP, ktorá prejde všetkými možnými trasami medzi našimi destináciami a vyberie prechod grafom, ktorý je optimálny vzhľadom na cenu celkovej trasy. Do argumentu naša funkcia berie východzie miesto, zoznam destinácií ktoré ešte neprehľadala a domáce miesto do ktorého sa má nakoniec vrátiť. V každom vnorení sa zoznam neprehľadaných vrcholov zmenšuje. Vyššie spomenuté argumenty sú perzistentné objekty resp. polia inštancií perzistentnej triedy \textit{Place}.

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

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

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

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

\begin{algorithm}

\lstset{

language=PHP,

basicstyle=\small\sffamily,

frame=none,

numbers=left,

xleftmargin=5.0ex,

numberstyle=\tiny,

stepnumber=1,

showstringspaces=false,

keywordstyle=\color{blue}\bfseries

}

\lstset{emph={% Adjust any special keywords

printf%

},emphstyle={\color[rgb]{1,0,0}\bfseries}%

}%

\begin{lstlisting}

MATCH (from:Node {name: "Source node" })

MATCH path = (from)-[:CONNECTED\_TO\*6]->()

WHERE ALL(n in nodes(path) WHERE 1 = length(filter(m in nodes(path) WHERE m = n)))

AND length(nodes(path)) = 7

RETURN path,

reduce(distance = 0, edge in relationships(path) | distance + edge.distance)

AS totalDistance

ORDER BY totalDistance ASC

LIMIT 1

\end{lstlisting}

\caption{Ukážka riešenia TSP pomocou dopytovacieho jazyka Cypher}

\label{cypherTSP}

\end{algorithm}

\subsection{Mapa}

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

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

%TODO ref tech

%TODO screenshoot markercluster ref

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

\subsection{Vykresľovanie tabuliek}

\noindent Aby sme zlepšili UX stránky rozhodli sme sa na vykresľovanie dynamického obsahu využiť namiesto HTML obsahu renderovaného na serveri javascriptovú knižnicu na prácu so šablónami a, dynamický obsah načítavame z API našej aplikácie vo formáte JSON. Na vykresľovanie dynamického obsahu sme zvolili knižnicu Handlebars.js. Na príklade \ref{dynamictable} môžeme ukázať príklad našej implementácie dynamickej tabuľky pre hlavnú tabuľku zobrazujúcu destinácie a ich detaily.

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

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

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

\section{Výsledky práce)}