Univerzita Komenského, Bratislava
Fakulta Matematiky Fyziky a Informatiky

Modernizácia portálu lekar.sk

Bakalárska práca

2014

Juraj Macháč

Univerzita Komenského, Bratislava
Fakulta Matematiky Fyziky a Informatiky

Modernizácia portálu lekar.sk

Bakalárska práca

Študijný program: Informatika
Odbor: 2508 Informatika
Katedra: Katedra informatiky
Vedúci: RNDr. Tomáš Kulich, PhD.

Bratislava, 2014
Juraj Macháč

93804668

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Juraj Macháč
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Modernizácia portálu lekar.sk

Cieľ: Modernizovať portál lekar.sk; refaktorovať kód, opraviť základné programátorské a UX chyby, spraviť základný profiling nad databázou a optimalizovať SQL queries.

Vedúci: RNDr. Tomáš Kulich, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 24.10.2013

Dátum schválenia: 24.10.2013 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Pod'akovanie

Touto cestou by som rád pod'akoval RNDr. Tomášovi Kulichovi, PhD. za odborné vedenie, cenné rady a pripomienky pri tvorbe tejto práce.

Abstrakt

Web portál lekar.sk slúži primárne na vyhľadávanie a hodnotenie slovenských doktorov. Táto práca sa zaoberá optimalizáciami čo do rýchlosti a do vylepšenia užívateľskej skúsenosti (UX) tohto portálu.

Kľúčové slová: optimalizácia, AJAX, vyhľadávacie nástroje

Abstract

Web portal lekar.sk serves primarily for searching and evaluating slovak doctors. This bachelor thesis deals with optimalizations in terms of speed and enhancing user experience of this web portal.

Keywords: optimalization, AJAX, search tools

Obsah

Zoznam obrázkov

Úvod

Určite sa každý už stretol s webovou stránkou, ktorá bola skôr pre trápenie, ako pre úžitok. Či už na vine bola neintuitívnosť, niekoľkosekundové čakanie pri zdanlivo jednoduchých operáciach (napríklad stránkovanie), chaotickosť, nekorektnosť dát, alebo čokoľvek iné, čo zhoršilo zážitok uživateľa. Je celkom ľahko možné, že mnoho užívateľov demotivuje surfovanie po takej stránke, hoci jej idea môže byť veľmi dobrá. Práve preto je mojou úlohou chyby podobného typu z portálu lekar.sk odstrániť. Odstránenie týchto chýb vyžaduje pochopenie internej štruktúry databázy, web application framework-u a taktiež spôsob myslenia programátorov, ktorí tú stránku naprogramovali.

Na docielenie časovej optimalizácie načítavania stránky budeme často využívať technológiu AJAX, vďaka ktorej sa obnovujú vždy len potrebné komponenty. Celá webová aplikácia je naprogramovaná vo frameworku Yii nad jazykom PHP.

Optimalizácia sa samozrejme týka aj databázových dotazov (queries), algoritmov prebiehajúcich na serveri a tiež spôsobu rozdelenia práce medzi serverom a klientom - prenesenie časti práce na klienta za účelom zrýchlenia servera. Tejto problematike sa budeme bližšie venovať v druhej kapitole.

K vylepšeniu intuitívnosti a UX (user experience) pomôžu okrem optimalizácií aj pridané vyhľadávacie nástroje, ktoré pomocou heuristických metód a AJAXu umožňujú užívateľom dostať sa k požadovaným dátam rýchlejšie a efektívnejšie. Tieto nástroje budeme rozoberať v tretej kapitole.

Kapitola 1

Základné pojmy a využité metódy

1.1 Dynamické webové stránky

Na počiatku internetovej éry boli stránky statické a teda zobrazovali stále tie isté dáta, až kým ich niekto ručne neprepísal. To je ale veľmi pracné a hlavne nepraktické - nie je možná prakticky žiadna automatizácia, práca s databázou, ani nič podobné. Táto éra ale netrvala dlho a začali sa postupne vyvíjať spôsoby, ako meniť stránku dynamicky - teda bez zásahov do jej kódu. Dynamické stránky umožňujú HTML kód generovať ön-the-fly", čiže keď sa stránka znovu načíta, tak sa môže vygenerovať nový HTML kód (napríklad s iným časom, alebo inými dátami), alebo sa HTML kód mení na stránke aj bez načítania (napríklad pomocou JavaScriptu). Podľa toho sa dynamické webové stránky dajú rozdeliť na **server side** a **client side**

- **Server side** - stránka sa generuje na serveri pomocou skriptu a následne sa odošle klientovi
- **Client side** - vzhľad HTML stránky určuje JavaScript, alebo iný client-side skriptovací jazyk. Rovnako môže upravovať jednotlivé DOM (Document Object Model) počas behu stránky.

Dynamické webové stránky teda umožňujú dáta čerpať z databázy a tým pádom automaticky zobraziť aktuálnu (s dátami z databázy) stránku pri každom novom načítaní (napríklad News Feed). Prípadne sa dajú pomocou dynamických stránok používať schémy (layouts), čo značne uľahčuje tvorbu stránok pre programátorov - napríklad pri jemnej zmene v navigačnom bare nie je potrebné ho meniť na každej strane, ktorá ho obsahuje, ale len v schéme pre ten navigačný bar.

Niekedy sa dá z pohľadu užívateľa dynamická stránka odlíšiť od statickej veľmi jednoducho. Ak prípona súboru na stránke je ".html", resp. ".htm", tak je stránka statická, a ak je

prípona ".php", ".asp", alebo ".jsp", tak je pravdepodobne dynamická. No nie každá stránka, ktorá je dynamická (jej súbor má príslušnú príponu) musí mať aj dynamický obsah - jej kód môže byť čistý a nemenný HTML kód. Každopádne, drvivá väčšina dynamických stránok majú aspoň nejaký obsah, ktorý sa dynamicky mení.

Samozrejme, niektoré malé stránky nemá zmysel robiť dynamicky (napríklad ak obsahujú len zopár stránok, ktoré sa prakticky nemenia), no pre väčšie stránky, ktoré majú byť často aktualizované je to prakticky nutnosť. Dynamické webové stránky sa od svojho počiatku rozšírili veľmi rýchlo a teraz už ťažko nájdete nejakú komplexnejšiu stránku, ktorá by nebola dynamická.

1.2 Software Framework

Software Framework je predprogramovaný, všeobecne použiteľný kód, ktorý poskytuje základnú funkcionalitu a efektívne rieši časté problémy v danej oblasti. Obsahuje knižnice, podporné programy a API (Application Programming Interface), ktoré tieto komponenty spájajú a umožňujú finálnu funkcionalitu. Vďaka frameworkom nemusia programátori pri každej novej aplikácii nanovo implementovať základnú funkcionalitu, čo ušetrí veľa času.

Existuje mnoho typov software framework-ov, no pre účel tejto práce sa budem zaujímať len o Web application framework

Web Application Framework

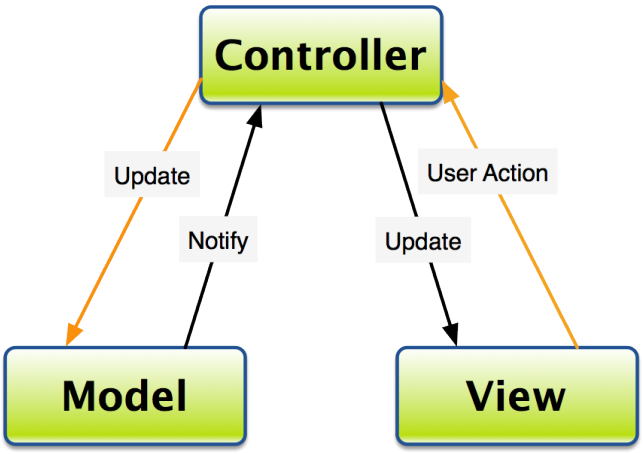
Tento typ framework-u je nadizajnovaný na vývoj dynamických webových stránok. Jeho účelom je pomôcť programátorom čeliť základným problémom, vyvarovať sa bežným chybám (napríklad bezpečnostným) a vyvíjať webové aplikácie pohodlnejšie a efektívnejšie. Často obsahuje knižnice napríklad pre prácu s databázou, autorizáciu návštevníkov alebo vzor stránok. Väčšina týchto framework-ov má architektúru založenu na princípe Model View Controller (MVC).

Model View Controller

MVC je vzor na implementáciu užívateľského rozhrania. Jeho úlohou je rozdeliť aplikáciu na 3 prepojené časti, aby sa rozdelila interná reprezentácia dát (Model) od tej, ktorá je akceptovaná (Controller) alebo prezentovaná (View) užívateľom. Vďaka tomu napríklad pri zmene vzhľadu stránky, ktorá ale pracuje s rovnakými dátami, sa zmena dotkne len komponentu View.

Okrem toho, že aplikáciu rozdeľuje, definuje aj vzťahy medzi nimi:

- **Model** - reprezentácia informácií, s ktorými aplikácia pracuje. Upozorní svoj Controller, ak v ňom niečo zmenilo, a to umožnuje Controlleru následne upraviť View. Model je na View nezávislý, nepotrebuje mať o ňom žiadne informácie.
- **View** - vidí ho užívateľ, svoje požiadavky posiela Controlleru, ktorý vykoná potrebnú administratívu a obnoví View. Pri obnovovaní môže View dostať od Controller-a aj odkaz na Model - môže dáta získavať priamo z neho.
- **Controller** - používaný užívateľom, a prostredníctvom neho mení stav Modelu, alebo mení vzhľad Modelu vo View (vizuálnu reprezentáciu dát Modelu)



Obr. 1.1: Schéma fungovania MVC architektúry

Príklad fungovania architektúry MVC pre jednoduchý formulár:

1. Užívateľ vyplní formulár (ku ktorému existuje príslušný model) a stlačí Submit.
2. Príslušný Controller je upozornený, že nastala takáto akcia a obdrží dáta. Týmito dátami sa snaží upraviť Model.
3. Model overí validitu dát a podľa toho upraví svoj stav a pošle správu o výsledku Controller-u.
4. Controller podľa správy od Modelu upozorní View a podľa potreby mu pošle odkaz na Model, z ktorého môže získavať dáta.

1.3 AJAX

jQuery - JavaScript knižnica, ktorá programátorom značne zjednodušuje programovanie v JavaScripte. Jej syntax je spravená tak, aby zjednodušovala prácu s dokumentom - výber DOM (Document Object Model) elementov, ošetrovanie udalostí, vytváranie animácií a samozrejme vývoj AJAXových aplikácií. AJAX requesty sa pomocou tejto knižnice naozaj píšu veľmi jednoducho a je v nich len to najpotrebnejšie a to v intuitívnej forme. Táto knižnica sa dnes používa už skoro na všetkých väčších stránkach.

JSON - JavaScript Object Notation - je formátový štandard na prenos dát tvaru key-pair (asociatívnych máp). Aj keď má v názve JavaScript, tento formát je nezávislý na programovacom jazyku. Jeho dáta sú v čitateľnom tvare a často sa používa práve vo webových aplikáciach na prenos dát medzi serverom a klientom.

História AJAXu: V minulosti (približne do roku 1998) sa väčšina stránok správala tak, že sa pri každej akcii musela celá stránka znovu načítať aj s komponentami, ktoré sa vôbec nezmenili. Tým pádom sa robilo zbytočne veľa práce naviac, čo malo za následok zbytočne vysokú záťaž na serveri a pomalé načítavanie stránok u klienta. Preto bola snaha preniesť aspoň nejakú prácu na klienta - napríklad pomocou LiveScript (predchodca JavaScriptu) sa dala kontrolovať validita formulára predtým, ako sa odoslal na server. Neskôr sa vyvinul Dynamic XML, ktorý fungoval tak, že stránka mala namiesto HTML kódu XML kód, ktorý sa následne renderoval pomocou JavaScriptu - to umožnovalo premiestniť napríklad rôzne zoraďovanie a filtrovanie položiek na klienta. No žiadna z týchto technológií neumožňovala získavať viac dát - to umožnil až XMLHttpRequest, ktorý bol prvý krát implementovaný v Internet Explorer 5. Neskôr si tento XMLHttpRequest postupne osvojili aj iné prehliadače až sa vyvinul do technológie, ktorú poznáme pod názvom AJAX.

AJAX - Asynchronous Javascript And XML - technológia, ktorá umožňuje tvorbu asynchrónnych webových aplikácií. Nie je to samostatný programovací jazyk, ale len spôsob využitia existujúcich štandardov. Pomocou AJAXu sa dajú teda posielať a prijímať dáta asynchrónne v pozadí stránky - teda bez toho, aby sa musela znovu celá načítavať. Tým pádom AJAX umožnuje sťahovať len tie skutočne potrebné dáta a zaplniť nimi komponenty, ktorých sa zmena dotkla (napr. pri stránkovaní sa vďaka AJAXu nemusí nanovo načitavať CSS, JavaScript ani reklamy okolo, ale len ďalšia strana dát). Skladá sa z viacerých technológií - CSS + HTML zobrazujú stránku, DOM (Document Object Model) sa mení pomocou JavaScriptu, a cez XMLHttpRequest (požadovaný text nemusí byť XML, môže to byť ľubovoľný text) je zabezpečená asynchrónna komunikácia. Cez AJAX sa často získavajú aj texty

vo formáte JSON a celú prácu s AJAXom a samotné requesty sú zjednodušené pomocou knižnice jQuery.

Hlavnou výhodou AJAXu teda je, že namiesto toho, aby sa musela na serveri znovu renderovať a posielať celá zobrazovaná stránka a k nej aj niekoľko ďalších súborov (napr. CSS alebo JS súbory), sa pošle len jeden jednoduchý textový súbor, ktorý obsahuje dáta v požadovanom formáte - väčšinou JSON, no môže obsahovať ľubovoľný text - a tieto sa následne cez JavaScript (resp. jQuery) spracujú a zobrazia na správnych miestach.

Keďže AJAX rádovo urýchľuje beh stránky, tak ho budem v mojich optimalizáciach často používať.

Funkcia jQuery.ajax() (alebo skrátene $.ajax()) poskytuje mnoho voliteľných možností, no na ukážku sily **jQuery** si ukážeme jednoduchý príklad AJAX requestu, ktorý cez Google Geocode získa z adresy v premennej location zemepisné súradnice:

```
$.ajax({
    type: 'GET',
    url: 'http://maps.googleapis.com/maps/api/geocode/json?address='
        +location+'&sensor=false';
    dataType: "json",
    success: function(data) {
        if (data.status == "OK") {
            console.log("Data recieved:" + data); // JSON form
            console.log("Latitude: " + data.results[0].geometry.location.lat +
            "\n Longitude: "+data.results[0].geometry.location.lng);
            // Parsed lat-long from the data
        } else {
            alert("Location was not found");
        }
    },
    error: function(jqXHR, textStatus, errorThrown) {
        alert('Error occured: '+jqXHR.responseText);
    },
});
```

type: typ requestu, ktorý sa má odoslať (POST / GET)

url: URL adresa, kam sa má odoslať request

dataType: očakávaný formát dát - môže byť aj xml, html, text..

success: funkcia s parametrom prijatých dát, ktorá sa vykoná, ak request prebehne úspešne

error: funkcia, ktorá sa vykoná, ak niekde v requeste nastal error - voliteľné parametre jqXHR, textStatus, ErrorThrown

Prijaté dáta sú typu JSON, takže sa dajú interpretovať ako vnorené mapy a polia, a tým pádom sa k dátam dá pristupovať ako ku skutočným JavaScript objektom s rovnakou štruktúrou.

1.4 Bezpečnosť

Modernizácia portálu nie je len otázkou nekonečných optimalizácií čo do rýchlosti, ale aj do bezpečnosti. Vrámci webových aplikácií je niekoľko oblastí, v ktorých existujú rôzne bezpečnostné štandardy, ktoré by mal každý slušný portál spĺňať. Keďže týchto oblastí je viac, priblížime si len tie, ktorým sa budeme neskôr venovať.

1.4.1 Ukladanie hesiel

Ukladanie hesiel v databáze nie je triviálna záležitosť. Hlavným pravidlom je, že ani pri ukradnutí databázy nesmie byť možné heslo efektívne zistiť. Mnoho užívateľov by ale možno chcelo, aby pri zabudnutí hesla dostali naspäť to svoje, ktoré zabudli, no ak sa chceme zamerať na bezpečnosť, tak to ani nie je realizovateľné - ukážeme si, prečo.

Aby sa totiž heslo mohlo vrátiť v pôvodnom tvare, musí si ho databáza pamätať (alebo si pamätať nejaký spôsob, ako ho vykonštruovať). Pamätanie hesla v databáze v otvorenom tvare sa priamo prieči s pravidlom pre bezpečnosť hesla - pri ukradnutí databázy sú všetky heslá prístupné - takže to nie je dobrý prístup.

Šifrovanie hesiel: Jeden prístup ako zabezpečiť spokojnosť užívateľov a zároveň bezpečnosť je, že sa heslá ukladajú do databázy zašifrované nejakým kľúčom, a tým pádom sa užívateľom dajú heslá vrátiť v pôvodnom tvare - no ak sa zameriame na bezpečnosť, tak to má svoje nedostatky. Keďže algoritmus šifrovania nemôže byť tajný, pretože sila dobrej šifry nesmie záležať na utajenosti algoritmu (je ťažké udržať algoritmus v tajnosti, preto sa treba sústrediť na utajovanie kľúčov), musia byť tajné kľúče. Tu vznikajú nasledovné problémy:

1. Ak sú všetky heslá šifrované jedným kľúčom - bezpečnosť hesiel v databáze záleží aj na bezpečnosti kľúča, teda ak sa odmaskuje kľúč, tak sú prístupné heslá - čím sa oslabila bezpečnosť uložených hesiel

2. Ak sa heslá šifrujú rôznymi kľúčmi - okrem problému podobnému prvému prípadu nás to stavia na rovnakú pozíciu, ako predtým - treba zabezpečiť bezpečnosť a reverzovateľnosť (v tomto prípade dokonca efektívnu kvôli prihlasovaniu) kľúčov

Existuje niekoľko modifikácií tohto prístupu, ktoré sa snažia zabezpečiť bezpečnosť šifrového kľúča napríklad tým, že sa nebude často používať, no aj tak je bezpečnosť hesiel užívateľov oslabená aj bezpečnosťou šifrového kľúča. Preto sa pre ukladanie hesiel na úkor reverzibilnosti, no s vyššou bezpečnosťou, často používa iný prístup - hashovanie.

Hashovanie: Do databázy sa neuložia heslá samotné, ani ich šifrované podoby, no ich odtlačky - hashe. Tieto hashe sú vypočítané hashovacou funkciou, ktorá musí byť: ireverzibilná, odolná voči kolíziam, závislá len na vstupnom reťazci (teda rovnaký reťazec vždy zobrazí na rovnaký hash). Tieto vlastnosti spĺňajú napríklad hashovacie funkcie z rodiny SHA-2 (SHA-224, 256, 384, 512). Databáza nepozná pôvodné heslo, pozná len jeho odtlačok a tieto odtlačky vždy porovnáva pri prihlasovaní. Dostávame sa k nasledovným dvom tvrdeniam:

1. Keďže hashovacia funkcia je odolná voči kolíziam, je vysoko nepravdepodobné, že sa niekto prihlási s nesprávnym heslom (menej ako 10^{-74}).

2. Keďže je ireverzibilná, tak pri ukradnutí databázy je nemožné heslo rekonštruovať.

Je ale to naozaj tak? Posledné dve tvrdenia zneli celkom dôveryhodne, no jedno z nich je v reálnom svete nepravdivé. Tvrdenie 2 platí len ak majú všetky heslá entropiu rovnakú, ako reťazec náhodných znakov, čo ale pri užívateľských heslách nie je ani zďaleka pravda. Existuje databáza mnohých najčastejších hesiel a keďže užívatelia s rovnakým heslom majú rovnaké odtlačky, tak si odtlačky stačí zoradiť podľa počtu výskytov a skúšať najčastejšie heslá - tento útok je výpočtovo nenáročný a je veľmi vysoká pravdepodobnosť, že sa ním podarí zistiť netriviálny počet hesiel.

Treba zabezpečiť, aby užívatelia s rovnakým heslom mali v databáze iný odtlačok toho hesla. Z tohto dôvodu sa používa soľ - salt. Salt je reťazec znakov (dostatočne dlhý - napríklad 40 znakov), ktorý sa pri hashovaní pripojí pred, alebo za heslo - **nemusí byť tajný**. Je dôležité, aby sa pre každé heslo používal unikátny salt. Ak by sa totiž použil všade rovnaký salt, tak sa ním prakticky nič nevyriešilo, lebo ak majú užívatelia rovnaké heslo, tak s rovnakým saltom budú mať stále rovnaký odtlačok, a tým pádom by stále fungoval útok pomocou

skúšania najčastejších hesiel - čo je veľká bezpečnostná diera. Ak sú teda jednotlivé salty unikátne, tak týmto prístupom staviame útočníka pri ukradnutí databázy do pozície, kde má pred sebou odtlačky, z ktorých vysoko pravdepodobne (z dôvodu odolnosti hashovacej funkcie voči kolíziam) žiadne dva nie sú rovnaké. Takže pri snahe zistiť heslo je postavený pred rovnaký problém ako reverzovať danú hashovaciu funkciu, resp. nájsť kolíziu pre daný hash. Ak by salt nebol tajný, tak je útočník pri každom užívateľovi postavený do rovnakej situácie, ako keby chcel jeho heslo hádať (čo môže vždy - možnosť uhádnutia hesla sa nepokladá za bezpečnostnú dieru).

1.4.2 SQL Injection

SQL Injection je technika používaná na útok na aplikácie pracujúce s databázami. Tento útok prebieha tak, že do užívateľského vstupu sa vložia nebezpečne SQL príkazy, ktoré sa v nesprávne naprogramovaných aplikáciach aj vykonajú. Preto je potrebné v dotazy, kde vystupuje vstup od užívateľa, proti tomuto útoku zabezpečiť. Jednoduchá ilustrácia nebezpečného SQL dotazu pre prihlásenie užívateľa:

```
$query = "SELECT * FROM users WHERE name = '" + userName + "' AND password
    = '" + password + "';"
```

Tento dotaz je nebezpečný, lebo užívateľ nemusí byť čestný, a ak vie, ako vyzerá vykonávaný dotaz, tak tam môže zadať také výrazy, aby zmenil jeho podstatu. Napríklad vie prihlásiť ľubovoľného užívateľa (v tomto prípade admin): zadá *userName:* **admin' OR ('1' = '1** a do *password* zadá: **') AND '1' = '1**. Tým pádom bude celý dotaz vyzerať nasledovne:

```
SELECT * FROM User WHERE name = 'admin' OR ('1' = '1' AND password ='' )
    AND '1' = '1'
```

Po vykonaní tohto dotazu sa teda vráti z databázy riadok, ktorý obsahuje meno **admin** alebo má prázdne heslo (čo je veľmi nepravdepodobné) - takže aplikácia dostane práve jeden riadok, ktorý očakáva a užívateľa prihlási.

Na tento dotaz sa dá zaútočiť rôzne, okrem prihlásenia užívateľa sa dajú spraviť aj horšie veci - napríklad vymazanie celej databázy.

Ochrana proti SQL Injection

Je niekoľko spôsobov ochrany, najpriamočiarejší je ödvýznamniť "(escape) zo vstupu znaky, ktoré v SQL niečo znamenajú. Manuál pre SQL DBMS (Database Management System) špecifikuje, ktoré znaky majú špeciálny význam - tým pádom sa dá vyrobiť zoznam znakov,

ktoré potrebujú preklad. V **PHP** je na to vstavaná funkcia *mysqli_real_escape_string()*, ktorá sa o všetky takéto znaky postará.

Trochu odlišný prístup k riešeniu problému je pomocou pripravených príkazov (prepared statements). Tieto pripravené príkazy fungujú interne ako šablóny, do ktorých sa len podosádzajú hodnoty. DBMS si tento pripravený príkaz rozparsuje, skompiluje bez hodnôt a vykoná na ňom dotazové optimalizácie. Potom je pripravený na spustenie so zadanými hodnotami, ktoré už netreba ošetrovať, lebo SQL už na tomto mieste očakáva nejakú validnú hodnotu a teda SQL Injection by v tomto prípade bol považovaný za neplatný vstup. Okrem iného tieto pripravené príkazy umožňujú efektívne vykonávať viac štrukturálne rovnakých dotazov s rôznymi parametrami veľmi efektívne, keďže sa nemusí znovu kompilovať a optimalizovať.

My na ochranu budeme používať knižnice frameworku Yii, ktoré tento problém riešia pomocou pripravených príkazov.

```php
<?php
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);

// insert one row
$name = 'one';
$value = 1;
$stmt->execute();

// insert another row with different values
$name = 'two';
$value = 2;
$stmt->execute();
?>
```

Obr. 1.2: Pripravený príkaz v PHP (Zdroj: [Net14])

Kapitola 2

Štruktúra web portálu lekar.sk

Web portál lekar.sk je naprogramovaný vo web application frameworku Yii 1.1 jazyka PHP. Architektúra tohto frameworku je **MVC**. Keďže táto aplikácia obsahuje viac ako sto modelov, tak sa samozrejme nebudeme zaoberať všetkými, ale len si trochu priblížime celkovú štruktúru portálu a podrobnejšie sa budeme venovať len tej časti, ktorá nás bude finálne zaujímať.

2.1 Princíp fungovania portálu lekar.sk

Tento portál obsahuje veľkú databázu slovenských lekárov s ich špecializáciami a lokalitami. Registrovaní užívatelia majú možnosť týchto lekárov hodnotiť a pridať svoj komentár, a tak pomáhať iným návšetvníkom stránky vybrať si toho najlepšieho doktora. Pre doktorov je to tým pádom aj motiváciou sa zlepšiť. Lekári sa dajú hodnotiť v rôznych kategóriach, konkrétne: *Pomoc pacientovi, Komunikácia s pacientom, Ceny za služby, Zdravotná sestra*. Lekári majú okrem toho možnosť reagovať na otázky týkajúce sa zdravia od užívateľov na fórach, a tak zanechať trochu lepší dojem. Na tomto portáli je okrem toho spravených niekoľko vyhľadávaní, ktoré umožňujú vyhľadať doktorov podľa rôznych zadaných dát.

2.2 Framework Yii

Yii (skratka pre "Yes it is!") je framework nad jazykom PHP. Tento framework implementuje dizajn **MVC**. Okrem toho poskytuje funkcie pre zjednodušenie validácií v dotazníkoch (forms), ochranu proti SQL injection, podporu autorizácie a autentifikácie a samozrejme má svoj **ORM - Yii Active Record**. Toto ORM umožnuje s databázovými záznamami pracovať ako s objektami v PHP, uľahčuje CRUD operácie a zjednodušuje jednoduché operácie s databázou. Je považované (podľa tvorcov Yii) za veľmi efektívne, no väčších dotazoch nemôže byť efektívnejšie, ako surový dotaz (keďže si ho musí vždy celý vyrobiť). Okrem

toho pri vypýtaní modelu vráti vždy len daný model bez jeho ďalších atribútov, ktoré sú s ním v relačnom vzťahu v databáze, a teda ak je potrebné sa dostať k takýmto atribútom, tak je to vždy na 2 dotazy. ORM vo všeobecnosti slúži skôr na ušetrenie kódu a pohodlné programovanie, než efektívnosť. Ak ide o efektivitu, odporúča sa písať priamo SQL dotazy. Tým pádom v momentoch, kde je hlavným bodom efektívnosť, sa budeme tomuto ORM vyhýbať a prikloníme sa radšej k písaniu surových SQL dotazov.

2.3 Vzťahy medzi modelmi

Medzi niektorými modelmi existujú vzťahy, ktorým sú prispôsobené vzhľady tabuliek v databáze. Tieto vzťahy môžu byť nasledovné: *MANY:MANY, MANY:ONE, ONE:MANY, ONE:ONE* . Vzťah *MANY:MANY* potrebuje separátnu tabuľku, ostatné majú v oboch tabuľkách jeden stĺpec, ktorý ich prepája. Pre každý model sú definované všetky jeho relačné vzťahy s ostatnými modelmi. Vďaka tomu vie *Yii* sprostredkovať prístup k jednotlivým tabuľkám, ku ktorým má daný model nejaký relačný vzťah, ako k svojim premenným - no vždy pri tom musí vykonať dotaz na databázu. Aj keď sú tieto dotazy jednoduché, prinajhoršom je to *JOIN* cez tri tabuľky, tak vykonanie každého dotazu trvá nejaký minimálny čas, čiže sa neodporúča k takýmto dátam pristupovať tak často ako k premenným. Ukážeme si niekoľko príkladov, ako sú v tejto databáze prepojené dáta, ktorým sa budeme venovať:

- Názvy ulíc sú uložené v tabuľke Address a názvy miest v City. Keďže v City nie je Bratislava ako jedno mesto, ale je rozdelená podľa mestských častí, tak v takýchto prípadoch je potrebné sa pozrieť do tabuľky Region.
  $Doctor \xleftrightarrow{MANY:MANY} Office \xrightarrow{MANY:ONE} Address \xrightarrow{MANY:ONE} City \xrightarrow{MANY:ONE} Region$

- Čo sa hodnotení týka, všetky hodnotenia, ktoré boli kedy odoslané sú v tabuľke Evaluation. Keďže každý lekár môže byť ohodnotený v niekoľkých oblastiach, tak existuje ešte tabuľka EvaluationValue, kde je uložená váha ohodnotenia, ktorá sa viaže na EvaluationCategory, kde sú všetky kategórie. Celkové hodnotenie lekára v nejakej oblasti sa určuje ako priemer EvaluationValue.value pre daného lekára a oblasť.
  $Doctor \xleftrightarrow{MANY:MANY} Evaluation \xrightarrow{ONE:MANY} EvaluationValue \xrightarrow{MANY:ONE} EvaluationCategory$

- Špecializácie doktorov sú uložené v tabuľke DoctorSpecialization, a keďže jeden lekár môže mať viac špecializácií, a jednej špecializácie je viac lekárov, tak je tento vzťah definovaný ako:
  $Doctor \xleftrightarrow{MANY:MANY} DoctorSpecialization$

- Čo sa týka údajov o užívateľovi, sú rozdelené do tabuliek User a Email, medzi ktorými je vzťah:
  $User \xleftrightarrow{ONE:ONE} Email$

Zo všetkých vzťahov v databázovom modeli sú tieto tie, ktorým sa budeme najviac venovať.

Obr. 2.1: Relačná schéma databázy (vyhotovené pomocou programu *MySQL Workbench*)

Kapitola 3

Bugfixy

3.1 Optimalizácia a vylepšenie UX

Stránky sa často načítavajú znovu celé pri každej drobnej požiadavke - vždy sa pošlú všetky obrázky, javascripty, css súbory, hoci vo výslednom zobrazení sa zmení len pár riadkov. Je to tým pádom veľká zátaž pre sever a zbytočne dlhé čakanie pre klienta. Tieto problémy sa dajú riešiť pomocou AJAXu - vypýtame si a zmeníme len tie dáta, pre ktoré je to naozaj potrebné.

3.1.1 Zoznam všetkých doktorov

Zdanlivo jednoduchý zoznam doktorov so špecializáciou a lokalitou zoradených podľa mena, kde je možnosť výberu začiatočného písmena doktora a zobrazenie 30 na stranu so stránkovaním. Tento zoznam bol pôvodne spravený tak, že pri klinutí na stranu alebo písmeno sa príslušná požiadavka poslala na server, kde sa spracovala nasledovne:
Vytvoril sa dotaz na databázu na vyhľadanie id všetkých doktorov ktorí vyhovujú požiadavkám pre stranu a počiatočné písmeno a potom sa pre každé nájdené id doktora vykonali nasledovné dotazy:
Našiel sa v tabuľke doktorov príslušný doktor, vyrobil sa ďalší dotaz, kde sa našli jeho špecializácie, a potom sa vyrobil ďalší dotaz, kde sa zistila jeho lokalita.
Keď už boli všetky dáta pozbierané, controller dal príkaz vyrenderovať celú stránku aj so všetkými príslušnými komponentami. Celý tento dotaz trvá zhruba **4s** (samozrejme zaležiac od internetu a momentálnej zataženosti stránky) a prenesie sa ≈ **550KB** dát, pričom jediné, čo sa na stránke zmení je 30 krátkych riadkov textu.

Keď si to rozoberieme na drobné, tak sa niet čomu čudovať. Odosielanie a vykonávanie mnohých malých dotazov je časovo oveľa náročnejšie, ako jeden veľký dotaz. Je tomu tak, lebo každý dotaz je potrebné odoslať na server (čo je minimálne čas pingu) a pre každý z

nich si MySQL musí vymyslieť, ako ho bude vykonávať tak, aby to bolo čo najefektívnejšie - táto operácia je síce rýchla a narastá s komplexnosťou dotazu, no je spojená aj s nejakým minimálnym konštantným časom potrebným pre každý dotaz. Pôvodne sa teda odosielalo pre každú stranu minimálne $2n$ dotazov (v skutočnosti ešte o niečo viac), kde n je počet doktorov zobrazených na danej strane. To je ale na takú jednoduchú vec neskutočne veľa, tieto dáta sa dajú pozbierať aj na oveľa menej dotazov.

Riešenie

Všetky potrebné dáta pre jednu stranu získame jediným väčším dotazom. Keďže ale chceme mať na spodu stránky pekný selector, je žial potrebné, aby si pri dotaze na nové začiatočné písmeno vypýtal najprv počet strán, čo je jeden krátky a jednoduchý dotaz - čiže občas budú celkovo 2, čo je ale stále menej, ako ≈ 70. Ten dotaz pre jednu stranu si v jednom poddotaze najprv zoradí doktorov podľa mena a vyberie z nich hľadanú množinu pre danú stranu:

```
SELECT d.id, d.surname, d.name, d.titleBack, d.titleFront
FROM Doctor d
WHERE d.surname LIKE CONCAT(:letter,'%') COLLATE utf8_general_ci
AND locked=0
ORDER BY d.surname ASC, d.name ASC, d.id ASC
LIMIT :limit,".$pageSize."
```

Túto množinu následne JOINe so špecializáciami:

```
SELECT doctors.*, ds.name AS spec
FROM (...) AS doctors
LEFT OUTER JOIN DoctorHasSpecialization dhs ON dhs.doctorId = doctors.id
LEFT OUTER JOIN DoctorSpecialization ds ON ds.id = dhs.specializationId
```

a nakoniec s adresami (kde nás zaujíma len názov mesta):

```
SELECT DISTINCT doctors.*, c.name AS city
FROM (...) AS doctors
LEFT OUTER JOIN DoctorHasOffice dho ON dho.doctorId = doctors.id
LEFT OUTER JOIN Office o ON dho.officeId = o.id
LEFT OUTER JOIN Address a ON a.id = o.addressId
LEFT OUTER JOIN City c ON c.id = a.cityId
ORDER BY doctors.surname ASC, doctors.name ASC, doctors.id ASC,
    doctors.spec ASC, c.name ASC
```

Keďže doktor môže mat viac špecializácií, či ordinácií, je potrebné tieto dáta pospájať. Vďaka spôsobu ich zoradenia to môžeme riešiť tak, že budeme pozerať ako sa menia stĺpce *id* a *spec* a podľa toho rozlišovať, či ide o toho istého doktora a či jeho mestá už máme zistené (keď sa zmení *spec* a nezmení *id*, tak už nemusíme ukladať mestá, lebo sa opakujú). Toto zvládneme spraviť jednoprechodovo v lineárnom čase od počtu riadkov.

Výsledok: Priemerný čas na request ≈ **120ms**, pričom objem prenesených dát je niečo **pod 2KB**.

3.1.2 Vyhľadávanie doktorov

Zo stránky sa dá niekoľkými spôsobmi dostať na stránku, kde je vyhľadávací bar, menu pre výber špecializácie a lokality, a nakoniec v strede zoznam doktorov, ktorí danú požiadavku spĺňajú. Keďže tento zoznam môže byť potenciálne veľmi dlhý, tak je na ňom nastavené stránkovanie s 10 doktormi na stranu. Problém ale je, že každá strana v stránkovaní je samostatný odkaz na stránku a teda všetko sa robí vždy nanovo a tým pádom sa znovu stretávame s veľkou záťažou na server. Užívateľ musí na každú stranu čakať 5s a databáza vykonáva rádovo viac dotazov, ako je potrebné

Analýza:

Ďalším problémom je, že toto vyhľadávanie bolo myslené ako také ïnteligentné"vyhľadávanie, čiže za nájdenou množinou a hlavne jej zoradením je mnoho logiky, ktorú veľmi ťažko zreprodukovať v čistom SQL - čiže v tomto prípade sa asi ORM nevyhneme, hoci je pomalé. Ak teda použijeme tie isté dotazy, ako predtým, treba aspoň zabezpečiť, aby sa nič neposielalo zbytočne - AJAXom si pošleme len tie zmenené dáta. Tým pádom bude "bottleneck"samotný dotaz na databázu cez ORM, ktorý sa môže ďalej vylepšovať.

Na to, aby sme mohli dotaz vhodne optimalizovať treba zistiť, čo z neho trvá najdlhšie. Krátky prieskum ukázal, že samotné nájdenie množiny modelov doktorov trvá len asi 15% času, pričom ten zvyšný čas zaberá zisťovanie všetkých špecializácií a lokalít pre každého doktora. To je ale neskutočne dlho na takú jednoduchú záležitosť. Na druhú stranu sa niet čomu čudovať, lebo kvôli databázovej štruktúre a ORM sa robí pre každého doktora veľmi veľa dotazov na databázu. Rozoberieme si počet dotazov na databázu pre zistenie adresy jediného doktora:

Listing 3.1: Zisťovanie lokalít pre jedného doktora

```
$cities = array();
$offices = $doctor->getOrdinations();
foreach ($offices as $office) {
	$city = "";
	if ($address = $office->getAddress()) {
		$city = trim($address->city->name);
		if (!in_array($city,$cities)) {
			$cities[] = $city;
		}
	}
}
```

Teda okrem toho, že sa pre každého doktora zisťujú dáta samostatne, čo už samé o sebe je veľa dotazov, tak všetky tieto metódy tried sú naprogramované tak, že vykonávaju dotazy na databázu a často nie jeden. Napríklad metóda *getOrdinations*() modelu *Doctor* je naprogramovaná tak, že sa nájdu všetky jeho kancelárie, následne pre každú z nich zavolá funkcia *getAddress*() ktorá na jeden dotaz zistí ulicu, na ktorej sa kancelária nachádza a na ďalší zistí mesto, v ktorej sa táto ulica nachádza (pričom vráti ulicu - nie mesto - iba ak mesto existuje). Navyše sa táto metóda používa znovu na zistenie ulice, pre ktorú (pomocou ďalšieho dotazu) sa zistí mesto, ktoré sme v konečnom dôsledku chceli zistiť. Takže napríklad sa pri každom doktorovi pre každú jeho adresu pýtame zbytočne na mesto 3x. Celkovo sa teda na zistenie lokality pre jedného doktora odošle na databázu minimálne 6 dotazov (a to za optimistického predpokladu, že má doktor len jednu kaceláriu). Taktiež je potrebný dotaz na zistenie špecializácií a ďalší dotaz na zistenie počtu hodnotení. To celkovo robí minimálne 8 dotazov na každého doktora, teda na vygenerovanie 10 doktorov na stranu je to 80 dotazov len na pozbieranie dát pre už nájdených a správne zoradených doktorov. Toto sa dá teda bezpochýb spraviť oveľa lepšie.

Riešenie:

Urobíme to teda tak, že všetky dáta pre všetkých doktorov pozbierame jedným dotazom. Keďže ale týchto doktorov nemáme v tabuľke, ale v nejakej dátovej štruktúre, je potrebné si takúto tabuľku vytvoriť - teda bude obsahovať id a poradie doktora, pričom pre rýchlosť nastavíme indexovanie na id. Toto sú 2 dotazy, pričom jeden z nich maže tabuľku, čo trvá zanedbateľne krátko. Skonštruujeme teda jeden väčší dotaz, v ktorom si efektívne zistíme všetky potrebné dáta. Urobíme to podobným spôsobom, ako v predošlom prípade. Najprv si nájdeme v databáze tých doktorov, ktorých to vyhľadávanie uložilo do pamäte:

```
SELECT d.id, d.surname, d.name, d.titleBack, d.titleFront,
   d.evaluationAverage AS evalavg, t.ord
FROM Doctor d
INNER JOIN tmpdoctortable as t ON t.id = d.id
```

Následne pre nich spočítame počet hodnotení:

```
SELECT d.*, COUNT(dhe.evaluationId) AS numevals
FROM (...) AS d
LEFT OUTER JOIN DoctorHasEvaluation dhe ON d.id = dhe.doctorId
```

Zistíme všetky špecializácie a mestá pôsobenia rovnako, ako pri predošlom probléme až na to, že sa teraz nebudú zoraďovať podľa priezviska, ale podľa stĺpca *ord*, ktorý v tabuľke *tmpdoctortable* označoval poradie doktora v o vyhľadávaní. Tým pádom získame výsledky podobného typu, ako predtým a môžeme ich teda podobným spôsobom rozparsovať - jednoprechodovo.

Týmto vylepšením sme znížili čas čakania na jednu stranu pre užívateľa z 5s na 0.7s pričom sa mu zmenia len tie dáta, ktoré sa zmeniť majú a objem prenesených dát je zanedbateľný.

Možné vylepšenia

1. Optimalizovať databázový dotaz na vyhľadanie správnej usporiadanej množiny doktorov.

2. Inteligentný caching u klienta: Získanie viacerých stránok zo servera je pomerne lacné, lebo nie je potrebné vykonávať znovu operáciu hľadania, no len získať informácie pre viac doktorov, čo je už teraz pomerne rýchle. Tieto dáta si môžeme u klienta pamätať v premennej a vždy pri vypýtaní novej strany sa najskôr pozrieť, či sa tam pre ňu už tie dáta nenachádzajú. Ak nie, tak si vypýta *vhodný* počet strán okolo. Aký počet je vhodný? Čím viac strán si vypýtame, tým dlhšie budeme čakať pri kliknutí na nezakešovanú stránku, no v priemernom čase budeme na jednu stranu čakať kratšie. Ako najvhodnejšia vzdialenosť sa ukázala 2 na každú stranu, pričom potom čaká uživateľ po kliknutí na nezakešovanú stránku približne 1.5s, no stiahne sa mu ich hneď 5, na ktorých sa môže okamžite preklikávať (taktiež je menší počet dotazov na server) a môže sa rýchlo a pohodlne vrátiť na strany, na ktorých už bol. Možno sa zdá, že takéto riešenie musí zaberať veľa pamäte, no keďže jedna strana (10) doktorov má objem približne 2KB, tak v najhoršom prípade pri počte 30 000 doktorov zaberie táto cache 6MB, čo je v dnešnej dobe priam zanedbateľne málo.

3.1.3 Autocomplete

Na stránke sú na niekoľkých miestach vyhľadávacie bary, ktoré slúžia na vyhľadanie doktorov podľa mena alebo špecializácie a užívateľa potom s nájdenými výsledkami odkážu na podstránku "vyhladavanie", ktorú sme v predošlej sekcii optimalizovali. Zadávanie do týchto vyhľadávacích polí je momentálne ale celkom mätúce, lebo ako postupne zadávate text, tak stránka vyzerá, že nereaguje, nič neponúka a teda malý preklep znamená, že požadovaný výsledok nedostanete. Na väčšine stránok je preto pri takomto vyhľadávacom bare aj funkcia autocomplete, ktorá pri každom zadanom písmenku pohotove reaguje s návrhmi na validné dokončenie. Takýto autocomplete kedysi fungoval aj na týchto vyhľadávacích baroch, no mal problémy s race conditions (pri nekvalitnom pripojení občas ponúkal návrhy pre starý text), a je možné, že ho preto odstránili.

Aby sme uviedli veci na správnu mieru, tento autocomplete nebol odstránený, len začal posielať nevalidný JSON. A to preto, lebo bola preprogramovaná funkcia *Yii::app()->end()* tak, aby vždy vypísala errory, prípadne "No errors"(to je okrem iného dôvod, prečo sa na náhodných miestach na stránke táto hláška občas zobrazuje). No štandardom asynchrónneho programovania je po odoslaní validného JSON response uzavrieť spojenie, a tým pádom by sa takto nedal poslať žiaden JSON asynchrónne. Po zrušení tohto výpisu sa síce zobrazujú návrhy na dokončenie, no stále sa nevyriešil problém s race conditions. Okrem toho pokus o zvýraznenie časti, ktorá sa zhoduje so zadaným textom, nefunguje s diakritikou.

Riešenie:

Na riešenie tohto problému sa potrebujeme pozrieť bližšie na $SiteController$, ktorý posiela návrhy a plugin $jquery.autocompletePacient.js$, ktorý tieto návrhy spracováva - zobrazuje, posiela a zvýrazňuje. Jednoduchým riešením je pridanie časovej pečiatky do každého listu návrhov poslaných zo servera. Na klientovi sa jednoducho zobrazí len vtedy, ak nezobrazuje práve aktuálnejšiu verziu.

Čo sa týka zvýrazňovania, tak si definujeme metódu na normalizáciu stringu - teda odstránenie diakritiky. Aplikujeme túto normalizáciu na kópiu stringov zadaného textu a navrhnutého doplnenia, potom môžeme vyhľadať spoločné podreťazce, a pre tie si nájsť koncové body jednotlivých zvýraznených častí. Nájdeme ich tak, že vždy musí predošlá zvýraznená časť skončiť, aby mohla začať ďalšia. Tým pádom vieme jednoducho a efektívne postupne zľava nahrádzať tieto koncové body výrazmi $< b >$ resp. $< \backslash b >$.

Výsledok:

Autocomplete v každom momente zobrazuje len najaktuálnejšie návrhy, pričom zvýrazňuje všetky podreťazce v navrhovanom dokončení pre každé slovo zo zadaného textu, ignorujúc rozdiely v diakritike.

3.2 Bezpečnosť ukladania hesiel

Pri modernizácii tohto webportálu prišlo aj k otázkam bezpečnosti. Pri prezeraní konfiguračných súborov a zdrojových kódov som prišiel na skutočnosť, že všetky heslá sa ukladajú s rovnakou soľou (salt). Ako sme opisovali v úvode, keď sa heslá ukladajú s rovnakým saltom, tak je to prakticky rovnaká bezpečnosť, ako keby sa ukladali bez neho. Hoci bol tento salt skrytý v kóde, tak už len znalosť, že salt nie je unikátny vedie k rovnakým odtlačkom pre rovnaké heslá. Tým pádom, ak má útočník prístup k databázovým dátam, vie bez problémov zistiť prihlasovacie údaje hneď niekoľkých užívateľov - stačí si hashe zoradiť podľa počtu výskytov a skúšať najčastejšie heslá. A skutočne, do 5 minút som vedel zistiť približne 300 hesiel. Túto chybu treba bezpodmienečne napraviť.

Riešenie:

Ako sme spomínali v úvode, salt musí byť pre každého užívateľa unikátny, no nie nutne tajný. Na tomto webovom portáli je nutnosť unikátnosti emailov, čo môže teda tvoriť celkom dobrý salt pre každého užívateľa. Keďže ale nevieme, aké majú užívatelia heslá v skutočnosti, nemôžeme každé jednoducho chytiť a prehashovať inak. Jedno jednoduché riešenie je zmeniť celkovo spôsob hashovania tak, že namiesto toho, aby hashe v databáze boli tvaru: *hash(password+salt)* budú tvaru *hash(hash(password+salt)+email)*. Keďže *hash(password+salt)* už máme v databáze pre každého užívateľa, tak len stačí každý z týchto záznamov znovu prehashovať s pripojeným emailom. Okrem toho treba na všetkých miestach v kóde, kde prebieha autentifikácia zmeniť tento spôsob uloženia hesla na nový.

Výsledok:

Skutočne, po takomto prehashovaní databázy neboli žiadne dva hashe rovnaké - čo vyplýva z odolnosti hashovacej funkcie voči kolíziam. Tým pádom je už toto ukladanie hesiel bezpečné a pre útočníka neexistuje lepší spôsob zistenia hesla ako útokom úplným preberaním alebo útokom na reverzovateľnosť hashovacej funkcie. Keďže táto hashovacia funkcia je z rodiny *SHA-2* a neboli pre nu zistené žiadne kolízie, môžeme sa spoľahnúť na jej ireverzibilnosť.

Kapitola 4

Rýchle vyhľadávanie

4.1 Motivácia

Keď je niekto nespokojný so svojim lekárom a potrebuje si nájsť nejakého lepšieho, darmo bude hľadať lekára podľa mena. A ak by chcel podľa špecializácie nájsť najlepšieho v okolí vzhľadom na hodnotenia, tak je potrebné uvážiť, že niektoré hodnotenia si mohol pre lepšie meno napísať aj ten doktor sám. Okrem toho, ak pacient býva niekde na kraji väčšieho mesta, tak by asi nechcel cestovať za obvodným lekárom cez celé mesto. Ďalší problém je, že každý pacient má iné preferencie - každá oblasť hodnotenia ho môže zaujímať rôzne. Napríklad niekomu nezáleží na cene, a zdravotnej sestre, hlavne nech je vyšetrenie kvalitné a čakacie doby krátke.

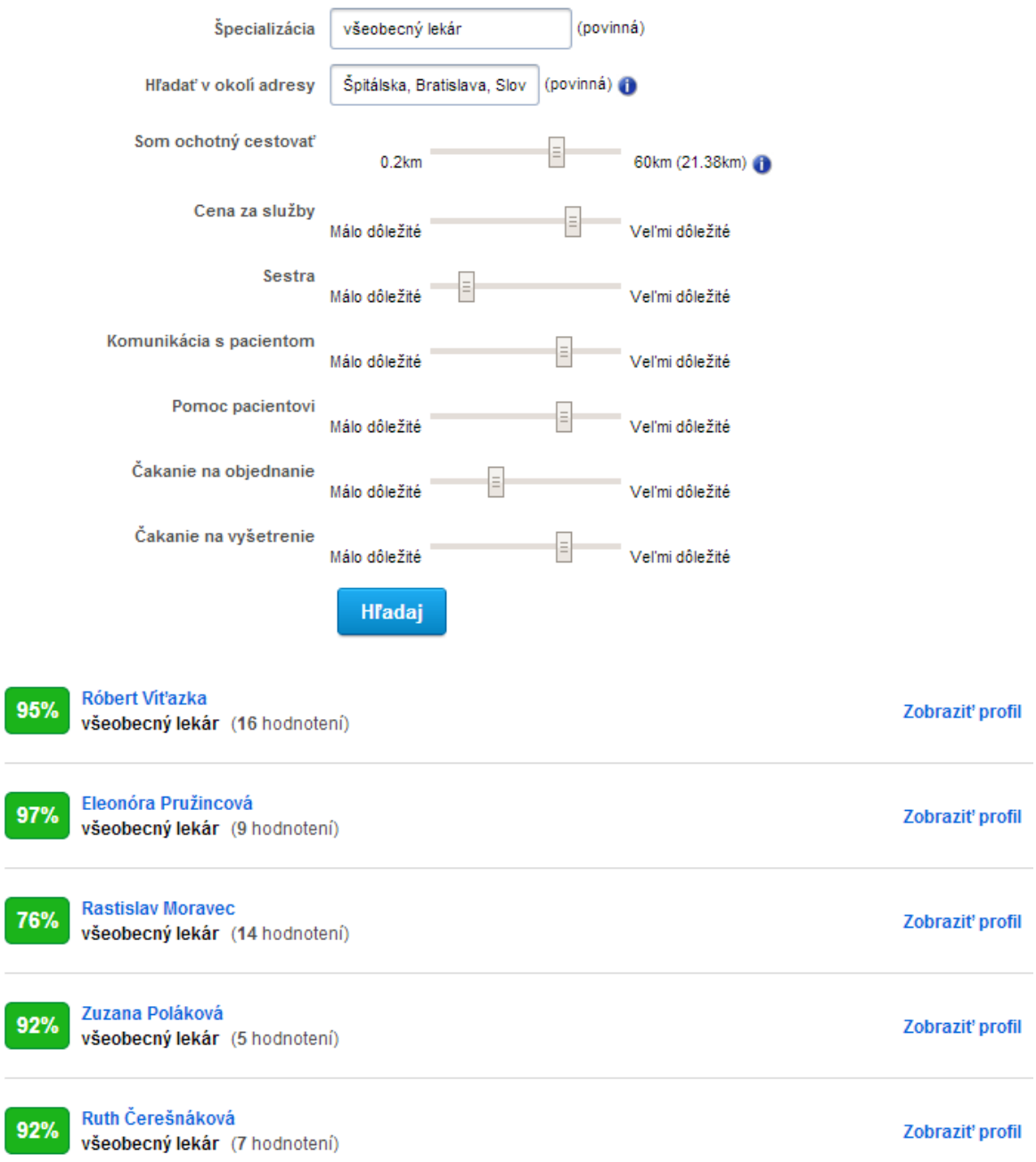
Úlohou tohto vyhľadávania je jednoducho, pohodlne a rýchlo nájsť pre daného pacienta najvhodnejšieho doktora. Toto rýchle vyhľadávanie rozsiahlo využíva *Google Place Autocomplete* a *Google Geocode API*

4.2 Špecifikácia

Pre zachovanie jednoduchosti je nutné, aby užívateľ jednoznačne vedel, čo sa od neho žiada, a aby toho nebolo veľa. Preto bude toto vyhľadávanie obsahovať len to najnutnejšie:

1. Pole pre špecializáciu
2. Pole pre polohu pacienta so sliderom pre vzdialenosť, ktorú je ochotný cestovať
3. Slidery pre jednotlivé kategórie, v ktorých sú doktori hodnotení, s prednastavenými hodnotami

Pre pohodlnosť budú všetky polia s automatickým doplňovaním. Pole pre špecializáciu je povinné, pretože ak niekto hľadá doktora, tak nehľadá najlepšieho ľubovoľného, ale nejakého danej špecializácie. Okrem toho to podstatne zrýchli requesty a odľahčí server. Pole pre polohu pacienta je voliteľné, prípadne sa stane povinným, ak bude server veľmi namáhaný. Automatické doplňovanie polohy bude fungovať pomocou *Google Place Autocomplete* . Vzdialenosti budú len približné, vzdušnou čiarou, pretože počítanie presných vzdialeností by bolo časovo veľmi vyčerpávajúce. Ak niekto zadá len mesto, bolo by vhodné, aby sa tam zobrazili lekári z toho mesta a zadaný počet kilometrov od mesta.



Obr. 4.1: Vzhľad rýchleho vyhľadávania

Keďže každý lekár je hodnotený v niekoľkých oblastiach, môžeme užívateľom poskytnúť možnosť vybrať si, čo ich najviac zaujíma. Na toto sú najlepšie slidery, ktoré budú reprezentovať váhu danej oblasti pre užívateľa. Okrem toho, nie všetkým hodnoteniam sa dá veriť - môžu si ich napríklad lekári napísať sami. Tým pádom, ak chceme vybrať pre každého užívateľa toho najlepšieho doktora, musíme nejakým spôsobom vziať do úvahy aj počet hodnotení.

4.3 Implementácia

Našim cieľom je skonštruovať jeden veľký dotaz na databázu, pomocou ktorého získame všetky potrebné dáta. Vzhľadom k špecifikácii môže byť tento dotaz dosť komplexný a keďže chceme mať tieto dáta efektívne(chceme, aby sa užívateľovi do sekundy zobrazil výsledok a zároveň, aby nezahltil server). Tým pádom je dôležité sa pri konštruovaní tohto komplexného dotazu zamerať na efektivitu jednodlitvých poddotazov.

4.3.1 Automatické doplňovanie

Pole pre špecializáciu si bude pri každej zmene asynchrónne cez AJAX pýtať aktuálne návrhy z databázy. Tak ako ostatné automatické doplňovania na tejto stránke, aj toto bude využívat plugin jquery.autocompletePacient, pričom requesty na množinu výsledkov sa budú posielať na site/autocompleteSpec. Návrhy sa budú vyberať nasledovne: z tabuľky špecializácií sa vyberú tie, pre ktoré je hľadaný reťazec ich podreťazcom. Následne dostanú vyššiu prioritu tie, ktoré týmto podreťazcom začínajú. Z týchto výsledkov sa vyberie rozumný počet (10), pridá sa časová značka, aby sa zabránilo race conditions, a odošle sa do jquery.autocompletePacient, kde sa overí, či je výsledok aktuálny (či je novší, ako najnovší) a podľa toho sa aktualizuje zoznam návrhov na stránke.

Automatické doplňovanie adresy sa pomocou Google Places API robí veľmi jednoducho, stačí pridať túto vlastnosť žiadanému elementu a ostatné za nás spraví Google. Aby boli výsledky čo najrelevantnejšie, upravíme ho tak, aby vracal len miesta na Slovensku.

4.3.2 Výber množiny lekárov

Po stlačení tlačidla Hľadaj (alebo Enter), sa najprv pošle AJAX request na *Google Geocode API*, z ktorého získame zemepisné súradnice adresy zadanej pacientom. Následne sa tieto súradnice, číslo požadovanej strany, aj všetky zadané dáta pošlú na server, kde sa odošle jeden komplexný dotaz na databázu. Rozoberieme si tento dotaz na niekoľko dôležitých častí.

Výber podľa špecializácie

Najprv sa vyberú lekári, ktorí majú aspoň jednu špecializáciu takú, ktorá začína, ako zadaná špecializácia (napr. pri zadaní "he"sa vyberú hematológovia aj hepatológovia)

Výber správnej oblasti

Je potrebné nejakým spôsobom určiť hranice kruhovej oblasti v zemepisných súradniciach dostatočne presne tak, aby tomu databázový dotaz rozumel. Zároveň je dôležité, aby tento výber bol dostatočne rýchly - je výhodné za daň 5% chyby ušetriť 20% času.

Haversine - Na počítanie presných vzdialeností na guli sa dá použiť haversin funkcia. Pre každé dva body A, B na povrchu gule s polomerom r platí, že ak $\phi_{A,B}$ sú ich zemepisné šírky, $\theta_{A,B}$ sú ich zemepisné dĺžky a d ich sférická vzdialenosť, tak:

$$haversin\left(\frac{d}{r}\right) = haversin(\phi_B - \phi_A) + cos(\phi_B)cos(\phi_A)haversin(\lambda_B - \lambda_A)$$

$$\wedge$$

$$haversin(\theta) = \frac{1 - cos(\theta)}{2} = sin^2\left(\frac{\theta}{2}\right)$$

$$\Downarrow$$

$$haversin\left(\frac{d}{r}\right) = sin^2\left(\frac{d}{2r}\right) = sin^2\left(\frac{\phi_B - \phi_A}{2}\right) + cos(\phi_B)cos(\phi_A)sin^2\left(\frac{\lambda_B - \lambda_A}{2}\right)$$

$$\Downarrow$$

$$d = 2arcsin\left(\sqrt{sin^2\left(\frac{\phi_B - \phi_A}{2}\right) + cos(\phi_B)cos(\phi_A)sin^2\left(\frac{\lambda_B - \lambda_A}{2}\right)}\right)$$

Pomocou tohto vzorca sa teda dá vypočítať presná sférická vzdialenosť dvoch bodov a teda by sa dalo pre každého lekára zistiť jeho presnú vzdialenosť od zadaného bodu a podľa toho ho pridať do vybranej množiny. Nevýhodou tohto vzorca je ale to, že používa veľa zložitých matematických operácií na výpočet vzdialenosti každých dvoch bodov. Skúšať to pri každom dotaze pre každého lekára (30 000) je výpočtovo veľmi náročné a trvá neúnosne dlho (30s). Existuje spôosob, ako tento dotaz rádovo zrýchliť $\Rightarrow$ zúžiť výber, na ktorých sa bude počítať haversin takzvanou obálkou. Obálka je obdĺžnik, ktorého 4 vrcholy sú určené ako priesečníky priamok, ktoré ohraničujú útvar zo 4 strán. To, či sa adresa nachádza v obdĺžniku sa dá zistiť veľmi rýchlo, ak sú správne postavené indexy (20ms) a na tejto užšej množine sa už tento haversin dá počítať celkom rýchlo, samozrejme v závislosti od hustoty adries v okolí (200ms pre 30km okruh od stredu Bratislavy). 200ms nie je veľa, no ani málo, ak sa má s tými dátami robiť ešte niečo ďalšie. Okrem iného, pre dosiahnutie takejto rýchlosti je nutné, aby boli postavené nad tabuľkou spatial indexy, ktoré ale InnoDB nepodporuje, takže engine tejto tabuľky by musel byť MyISAM, čo je zas spojené s problémom, že má slabé cachovanie dotazov. Ako uvidíme, existuje aj lepší, hoci jemne menej presný, ale rýchlejší a implementačne oveľa jednoduchší spôsob.

Pythagorov kruh - Na území Slovenska platí, že jeden stupeň zemepisnej šírky je približne $u = 111.309km$ a jeden stupeň zemepisnej dĺžky je približne $v = 74.289km$. Tým pádom, ak chceme pracovať so zemepisnými súradnicami ako v euklidovskom priestore, je potrebné, aby sme si mierku jemne upravili. Nech d je maximálna vzdialenosť v kilometroch, ktorú je pacient ochotný cestovať, *lat* a *long* sú zemepisné súradnice zadanej adresy $\Rightarrow$ d kilometrov je teda $\frac{d}{u}$ stupňov zemepisnej šírky. Ak by sme pomocou tejto konverzie vytvorili kruh na zemepisných súradniciach $lat, long$ s predpisom: $(x - long)^2 + (y - lat)^2 < \left(\frac{d}{u}\right)^2$ tak dostaneme elipsu, ktorej hlavná os je rovnobežná s osou y. Tým pádom táto elipsa má ale hlavnú os dlhú d kilometrov a vedľajšiu os $d\frac{v}{u}$ kilometrov. Potrebujeme teda tento predpis kruhu jemne upraviť. Aby sme sa v rovine s daným pomerom mierok mohli viac priblížiť ku kruhu, je potrebné ten predpis na mierky prispôsobiť. Teda pri predpise: $\left(x\frac{u}{v} - long\right)^2 + (y - lat)^2 < \left(\frac{d}{u}\right)^2$ dostaneme niečo veľmi blízke kruhu pri malých ($< 100km$) vzdialenostiach d.

Tento spôsob je dostatočne presný pre naše účely - chyba je veľmi malá, no vzdialenosti sú vzdušné, čiže pre pacienta aj tak približné. Čo sa týka rýchlosti, tak v dotaze sú len 2 operácie násobenia a jedno sčítanie (keďže $\left(\frac{d}{u}\right)^2$ je počas dotazu konštanta), čiže žiadne časobo náročné operácie. Adresy v 30km okolí Bratislavy to vyfiltruje do 120ms. Keďže tento spôsob je rýchlejší, implementačne menej náročný a presnosťou ďaleko postačújúci, tak použijem tento.

4.3.3 Výpočet skutočného hodnotenia

Priemer všetkých hodnotení nevypovedá o skutočnom hodnotení lekára. Niektoré hodnotenia si mohol napísať aj on sám, aby mal lepšiu reklamu. Tiež sa mohlo stať, že ho ohodnotil len jeden pacient, ktorý sa zdal ukrivdený, no to nehovorí o tom, že by ten lekár bol celkovo zlý. Preto je potrebné nejakým spôsobom uvážiť aj počet hodnotení. Použijeme princíp viery v hodnotenie.

Princíp viery v hodnotenie

Čo môžeme povedať o lekárovi bez hodnotenia? Nevieme, aký je, takže najlepší predpoklad je, že je priemerný - priemer hodnotení všetkých lekárov. Ak má ale niekto veľa (> 25) hodnotení tak sa tomuto hodnoteniu dá veriť. Ak má tak 5 hodnotení, tak by očakávané hodnotenie mohlo byť v strede medzi celkovým priemerom a jeho priemerom. Princíp viery v hodnotenie je teda interpolácia medzi priemerným hodnotením všetkých lekárov a priemerným hodnotením daného lekára. Pokúsime sa teda nájsť funkciu f, ktorá z premenných p (priemerné hodnotenie lekára), P (priemerné hodnotenie všetkých lekárov), x (počet hodno-

tení) vypočíta očakávané hodnotenie, ktorému sa dá veriť. Zo zadaných predpokladov musí táto funkcia spĺňať nasledovné podmienky:

$f(0, p, P) = P$ - ak má 0 hodnotení, tak je priemerný

f je monotónna - s počtom hodnotení viera rastie

$\lim_{x\to+\infty} f(x, p, P) = p$ - s počtom hodnotení konverguje k hodnoteniu daného lekára

$f(5, p, P) = \frac{P+p}{2}$ - Empiricky určené: lekár s 5 hodnoteniami má očakávané hodnotenie v strede medzi p a P

Je mnoho takých funkcií, ktoré tieto podmienky spĺňajú, jedna z najjednoduchších je táto: $f(x, p, P) = p + g(x)(P - p)$, kde $g(x)$ je ľubovoľná monotónna funkcia spĺňajúca:

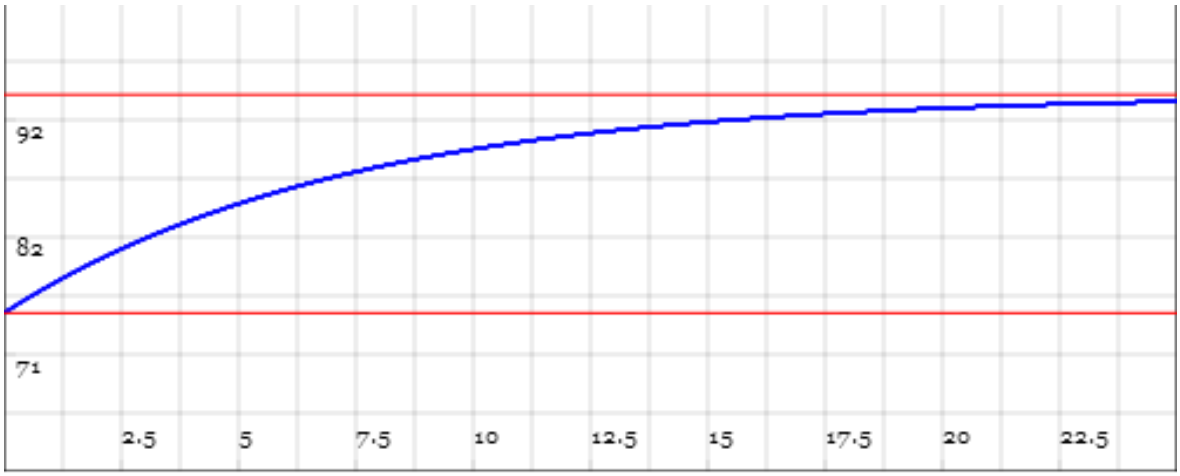
- $g(0) = 1$
- $\lim_{x\to+\infty} g(x) = 0$
- $g(5) = \frac{1}{2}$

$g(x)$ je teda zodpovedná za interpolovanie očakávaného hodnotenia medzi p a P. Môžeme ju chápať tak, že hovorí, aký je pomer $\left|\frac{f(x,p,P)-p}{P-p}\right|$, teda vzdialenosti očakávaného hodnotenia od priemeru lekára a vzdialenosti celkového priemeru od priemeru lekára. Poďme teda nájsť $g(x)$ spĺňajúcu dané vzťahy. Takýchto funkcií je tiež veľa, no aby sa s ňou dobre počítalo, tak nech je to nejaká pekná exponenciálna funkcia v tvare $g(x) = ae^{bx} + c, b \neq 0$. Postupne využijeme podmienky:

- $\lim_{x\to+\infty} g(x) = 0 \Rightarrow b < 0 \wedge c = 0$
- $g(0) = 1 \Rightarrow a + c = 1 \Rightarrow a = 1$
- $g(5) = \frac{1}{2} \Rightarrow ae^{5b} + c = \frac{1}{2} \Rightarrow e^{5b} = \frac{1}{2} \Rightarrow b = \frac{\ln\left(\frac{1}{2}\right)}{5}$

Tým pádom $g(x) = e^{-\frac{x\ln\left(\frac{1}{2}\right)}{5}} = 2^{-\frac{x}{5}} \Rightarrow f(x, p, P) = p + \left(2^{-\frac{x}{5}}\right)(P - p)$.

Počítanie exponenciálnej funkcie je v dotaze náročné, takže využijeme fakty, že počet hodnotení je nezáporná diskrétna premenná, a tiež, že $g(30) = 0.0.15625 \approx 0 \Rightarrow$ stačí si predpočítať hodnoty funkcie $g(x)$ pre $x \in \{0, .., 30\}$ a pre $x > 30$ uvažovať $g(x) = 0$.



Obr. 4.2: Graf $f(x, p, P)$ s parametrami $P = 70, p = 94$

4.3.4 Samotná implementácia

Klient nad ničím nerozmýšľa - len zozbiera dáta, pošle ich na server a čaká na odpoveď. Všetky dôležité výpočty sa dejú na serveri. Ak by sme len surovo pospájali všetky potrebné tabuľky a urobili nad nimi potrebné výpočty, trvalo by to neúnosne dlho (niekoľko minút). Preto je potrebné tento veľký dotaz rozdeliť na menšie poddotazy tak, aby sa každá drahá operácia robila nad čo najmenšou množinou - nepočítať nič zbytočne.

Rozdelíme ich na nasledovné poddotazy:

1. Predtým, ako začneme rátať skóre pre jednotlivých lekárov je potrebné vybrať množinu, ktorá spĺňa požadované podmienky pre polohu a špecializáciu. Najlacnejšia operácia je výber doktorov, ktorí majú danú špecializáciu:

```
SELECT DISTINCT d.id AS id, d.name, d.evaluationAverage,
    d.surname AS surname, d.titleFront, d.titleBack
FROM Doctor d
INNER JOIN DoctorHasSpecialization dhs ON dhs.doctorId = d.id
INNER JOIN DoctorSpecialization ds ON ds.id = dhs.specializationId
WHERE ds.name LIKE :spec COLLATE utf8_general_ci
    AND d.locked=0
```

2. Následne z týchto doktorov vyberieme tých, ktorí spĺňajú podmienky pre polohu. Tu sa musíme prispôsobiť flexibilite tohto poľa u klienta - teda ak je zadané len mesto, tak stačí, že ten lekár sídli v meste alebo požadovanú vzdialenosť od stredu mesta. Tieto mestá sú v tabuľke City, resp. niektoré väčšie sú v tabuľke Region. Na výpočet

vzdialenosti použijeme spomínanú metódu Pythagorovho kruhu.
(Pozn. tabuľka 'doctors' je výsledok predošlého poddotazu)

```
SELECT DISTINCT doctors.*
FROM (...) AS doctors
INNER JOIN DoctorHasOffice dho ON dho.doctorId = doctors.id
INNER JOIN Office o ON o.id = dho.officeId
INNER JOIN Address a ON o.addressId = a.id
INNER JOIN City c ON c.id = a.cityId
INNER JOIN Region r ON r.id = c.regionId
WHERE
    (POWER(a.lat-".$lat.",2)+POWER((a.long-".$long.")*74.289/111.309,2)
    < ".$distance.")
OR (:city LIKE CONCAT(c.name,'%') COLLATE utf8_general_ci)
OR (:city LIKE CONCAT(r.name,'%') COLLATE utf8_general_ci)
```

3. Keď už máme vybranú správnu množinu doktorov, už je "len"potrebné zoradiť ich podľa užívateľových preferencií od najlepšieho po najhoršieho a zobraziť požadovaný počet. Na toto potrebujeme zistiť priemer a aj počet hodnotení pre každú oblasť každého z vybraných lekárov (keďže si užívatelia môžu vybrať neohodnotiť lekára v nejakej oblasti). Ak je lekár nehodnotený v nejakej oblasti, tak neexistuje v tabuľke záznam s požadovaným id oblasti a tým pádom by sa pri jednoduchom spočítaní nezobrazil ani len záznam, že ich má 0. Tento problém vyriešime tak, že tabulku JOINeme s EvaluationCategory (ktorá obsahuje všetky hodnotené kategórie) a záznam, kde je riadok NULL - teda chýba hodnotenie danej oblasti započítame ako 0.

```
SELECT doctors.*, ec.id AS category,
    ROUND(AVG(EvaluationValue.value)) AS average,
    SUM(CASE WHEN EvaluationValue.evaluationCategoryId IS NULL
        THEN 0 ELSE 1 END) AS numevals
FROM (...) AS doctors
JOIN EvaluationCategory ec
LEFT OUTER JOIN DoctorHasEvaluation ON DoctorHasEvaluation.doctorId =
    doctors.id
LEFT OUTER JOIN Evaluation ON DoctorHasEvaluation.evaluationId =
    Evaluation.id
LEFT OUTER JOIN EvaluationValue ON (EvaluationValue.evaluationId =
    Evaluation.id
    AND ec.id = EvaluationValue.evaluationCategoryId)
WHERE (ec.isDoctor = 1 AND ec.id NOT IN (3,4))
```

```
GROUP BY d.id, ec.id
```

4. Zatiaľ máme len priemerné hodnotenie a počet hodnotení pre každú oblasť. Aby sme dostali skutočné hodnotenie, je potrebné aplikovať spomínanú exponenciálnu funkciu "viery"v hodnotenie - ExpApprox je tabuľka obsahujúca $g(x) = 2^{-\frac{x}{5}}\ \forall x \in N_0, x \leq 30$. AverageCategoryEvaluation obsahuje P pre každú kategóriu, t.average je p. Premenná $trust hodnotami medzi 0 a 1 určuje, ako veľmi zvažovať počet hodnotení (teda je to interpolácia medzi skutočným hodnotením a priemerným lekára). Vlastník portálu tam takýto slider nakoniec nechcel, takže jeho hodnota je prednastavená na 1.

```
SELECT doctors.*,
SUM(IF (t.numevals > 30, t.average*m.mul,
    IF (t.numevals = 0 OR t.average IS NULL, ace.average*m.mul,
    (".$trust."*(ace.average-t.average)*ea.approx+t.average)*m.mul)))
        AS score
FROM (...) AS doctors
LEFT OUTER JOIN ExpApprox ea ON ea.i = doctors.numevals
LEFT OUTER JOIN m ON t.category = m.cat
LEFT OUTER JOIN AverageCategoryEvaluation ace
    ON doctors.category = ace.category
GROUP BY doctors.id
ORDER BY score DESC
LIMIT ".$page.",".$pagesize."
```

5. Teraz už máme všetko vypočítané a v tabuľke máme tých doktorov, ktorých chceme zobraziť na danú stránku. Už sú len potrebné nejaké kozmetické úpravy, aby pri zobrazení so sebou niesli všetky potrebné informácie, teda celkový počet hodnotení a všetky špecializácie. Keďže je týchto záznamov v tabuľke už len veľmi málo (veľkosť strany 15), nemusíme sa zamýšľať nad efektivitou operácií.
   Počet hodnotení:

```
SELECT doctors.*, COUNT(dhe.evaluationId) AS numevals
FROM (...) AS doctors
LEFT OUTER JOIN DoctorHasEvaluation dhe ON doctors.id = dhe.doctorId
GROUP BY doctors.id
```

6. A nakoniec všetky špecializácie:

```
SELECT doctors.*, dss.name AS spec
```

```
FROM (...) AS doctors
INNER JOIN DoctorHasSpecialization dhs ON doctors.id = dhs.doctorId
INNER JOIN DoctorSpecialization ds ON ds.id = dhss.specializationId
ORDER BY doctors.score DESC, doctors.id ASC";
```

Tieto jednotlivé poddotazy spojíme do jedného veľkého dotazu a už nám len stačí urobiť jeden after-processing, kde zoskupíme týchto doktorov podľa ich id a popridávame im jednotlivé špecializácie. Následne môžeme tieto dáta odoslať klientovi, ktorý ich len pekne zobrazí.

Záver

Podarilo sa nám vytvoriť stránku, ktorá zobrazuje korektné dáta a dá sa po nej rýchlo a pohodlne prechádzať. Záťaž na server a databázu sa rádovo znížila, pričom sa zvýšila rýchlosť načítavania stránok. Okrem toho sme zabezpečili bezpečnostný štandard ukladania hesiel, čiže aj pri úniku záznamov z databázy sú užívateľské kontá v bezpečí. Vylepšili sme teda UX, a to tiež pomocou autocomplete, ktoré poskytuje správne návrhy bez ohľadu na kvalitu pripojenia.

Veľkým prínosom bolo, že sme pridali rýchle vyhľadávanie, ktoré zohľadňuje polohu užívateľa, užívateľské preferencie u doktora a využíva spôsob viery v hodnotenie pre dosiahnutie najdôveryhodnejšieho percentuálneho výsledku kvality doktora. Na základe týchto údajov dokáže efektívne nájsť pre daného užívateľa toho najlepšieho doktora.

Na tejto stránke by sa dalo ďalej pracovať na rôznych vylepšeniach. Určite by pomohla celková zmena relačnej schémy databázy, aby sa nevytvárali často JOIN-y cez mnoho tabuliek na získanie triviálnych dát.

Literatúra

[bsp14] http://learningfinest.blogspot.sk/, Január 2014.

[Dic14] The Tech Terms Computer Dictionary. http://www.techterms.com/, Január 2014.

[dRMSP04] doc. RNDr. Martin Stanek PhD. *Základy kryptológie* , December 2004.

[Enc14] Wikipedia The Free Encyclopedia. http://en.wikipedia.org/, Január 2014.

[Fra14] Yii Framework. http://www.yiiframework.com/, Február 2014.

[Goo14] Google. https://developers.google.com/maps/documentation/geocoding/, Február 2014.

[Mic14] Microsoft. http://msdn.microsoft.com/, Január 2014.

[MyS14] MySQL. http://www.mysql.com/, Február 2014.

[Net14] PHP Net. http://www.php.net/, Apríl 2014.

[Rub14] Alexander Rubin. http://www.mysqlperformanceblog.com/2013/10/21/, Február 2014.

[Swa14] Aaron Swartz. http://www.aaronsw.com/weblog, Január 2014.