VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2016 Martin Matějka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Matějka**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Modern Entrepreneur s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Peter Chovanec**

Konzultant bakalářské práce: Lukáš Vlček

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26. dubna 2016 ..... Matějka .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 26. dubna 2016

.....

Moderní Podnikatel
Modern Entrepreneur s.r.o.
Adresa:
Pohraniční 504/27
703 00 Ostrava
Kontakt:
Info@modpreneur.cz
www.modpreneur.cz
IČO: 026 250 41
DIČ: CZ 026 250 41

Rád bych poděkoval všem pracovníkům společnosti Modern Entrepreneur s.r.o. za vytvoření příjemného pracovního prostředí a odbornou pomoc při absolvování této odborné praxe.

Abstrakt

Tato bakalářská práce popisuje průběh odborné praxe, kterou jsem absolvoval ve firmě Modern Entrepreneur s.r.o. Je zde popsáno mé odborné zaměření na praxi a zadané úkoly společně s jejich řešením. Hlavním úkolem bylo vytvořit API rozhraní pro vyhledávání, na jehož základech byly poté postaveny další úkoly. V závěrečné části jsou uvedeny mnou získané znalosti, dosažené výsledky a jejich zhodnocení.

Klíčová slova: odborná praxe, PHP, Symfony, Doctrine, Codeception, vyhledávání, newslettery

Abstract

This barchelor thesis describes the process of professional practice which I practiced in Modern Entrepreneur s.r.o. company. There is described my specialization on practice and assigned tasks along with their solutions. Main task was to create API interface for search, which was the basis for other tasks. In the end of this work are described my gained knowledge, achieved results and their assessment.

Key Words: professional practice, PHP, Symfony, Doctrine, Codeception, search, newsletters

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
CRON	–	Command run on
DQL	–	Doctrine query language
HTML	–	Hypertext markup language
JSON	–	JavaScript object notation
MVC	–	Model-View-Controller
NQL	–	Necktie query language
OOP	–	Objektově orientované programování
ORM	–	Objektově-relační mapování
URL	–	Uniform Resource Locator

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato bakalářská práce popisuje mou odbornou praxi ve společnosti Modern Entrepreneur s.r.o.[5] Pro její absolvování jsem se rozhodl především proto, že jsou v současné době kladeny stále větší požadavky na odborné znalosti zaměstnanců. Kromě teoretických znalostí získaných v průběhu studia mají vysokou váhu hlavně praktické zkušenosti, mezi které patří zkušenosti s vývojem sofwaru, schopnost práce v týmu a komunikace nejen se spolupracovníky, a také schopnost pracovat pod tlakem daných termínů. Jednou z možností, jak se těmto dovednostem naučit a ověřit kvalitu budoucích pracovníků je účast studentů na odborných praxích ve firmách již v průběhu jejich vzdělávání. Své dovednosti a dosud nabyté teoretické znalosti jsem se tedy rozhodl prověřit absolvováním bakalářské praxe.

V následujících kapitolách vás tedy provedu náplní mé praxe, kde jsem pracoval na jednotlivých úkolech samostatně, za pomoci mého vedoucího.

2 Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta

2.1 Popis odborného zaměření firmy

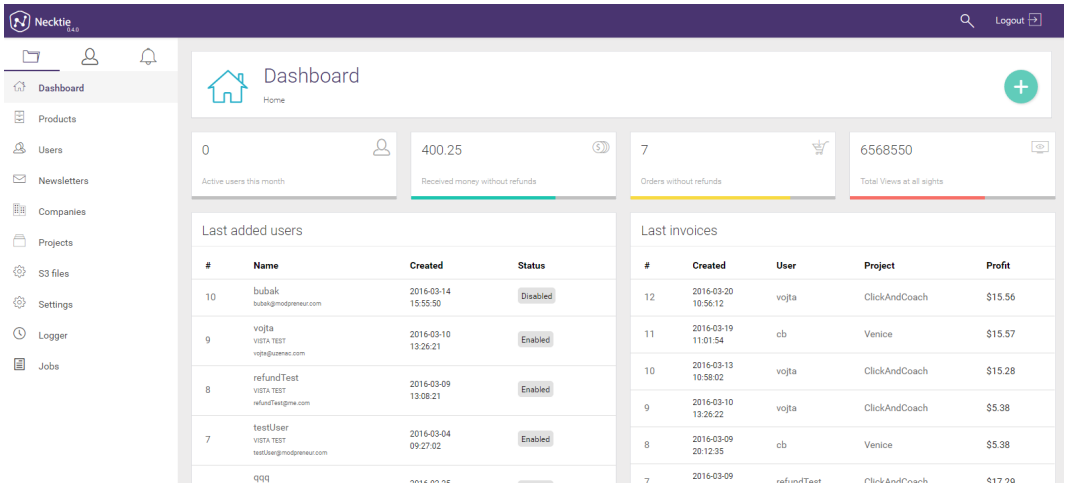
Firma Modern Entrepreneur s.r.o. byla založena v roce 2014 a poskytuje služby, mezi které patří vývoj webu, softwaru, mobilních aplikací včetně designu, user testing a split-testing. Cílí především na zákazníky z USA. Hlavní sídlo společnosti se nachází v Ostravě, pobočku má také v Brně.

Firma v současnosti vyvíjí 3 vlastní produkty:

- Ascot - nástroj pro tvorbu frontendových částí webů
- Venice - webový backend sloužící k distribuci uživatelského obsahu
- Necktie - CRM systém pro správu uživatelů, který zastřešuje předchozí produkty

2.2 Pracovní zařazení studenta

Ve firmě jsem pracoval jako PHP programátor. V rámci mé bakalářské práce mně byly přiděleny úkoly týkající se převážně rozvíjení firemního produktu Necktie, CRM systému sloužícího ke správě uživatelů.



Obrázek 1: CRM systém Necktie

3 Seznam úkolů zadaných studentovi v průběhu odborné praxe

3.1 Seznámení se s PHP frameworkem Symfony a testovacím frameworkem Codeception

Jelikož se ve firmě využívá hojně PHP[6] framework Symfony[8] a já s ním a jeho architekturou neměl dosud žádné zkušenosti, byl mi pro začátek přidělen úkol se s ním podrobněji seznámit. Následně jsem se také seznámil s frameworkem Codeception[1], který je nadstavbou testovacího frameworku PHP Unit[7] a s jehož pomocí jsem měl udělat následující úkol.

3.2 Testování webových API

S pomocí frameworku Codeception jsem měl za úkol vytvořit testy k webovým API, sloužícím primárně jako zdroj dat pro mobilní aplikace, které by tak ulehčily programátorovi čas strávený nad jejich manuálním testováním, a také snížily pravděpodobnost možného vzniku chyby.

3.3 Vytvoření vyhledávacího API

Úkolem bylo vytvořit API, které vrátí pole všech záznamů z databáze, odpovídající zadanému dotazu, zadaného dle předem definované syntaxe. URL adresa by měla být v následujícím tvaru:

```
/search/{název entity}/?q={dotaz}
```

3.3.1 Syntaxe vyhledávacího dotazu

Vyhledávací dotaz může být složen z následujících částí, a to v tomto pořadí:

- Výběr sloupců - požadované sloupce vepíšeme mezi kulaté závorky. Jednotlivé sloupce oddělíme čárkou. V případě, že chceme přistupovat za pomocí cizích klíčů ke sloupcům provázané tabulky, napíšeme název cizího klíče, dvojtečku a poté název sloupce tabulky, na kterou se cizí klíč odkazuje.

  ```
  (id,name,clients,billingPlan:price)
  ```

- Podmínka - vepíšeme mezi složené závorky. Syntaxe je obdobná jako u klasického SQL dotazu. Podmínky mohou být i vnořené.

  Mezi povolené operátory patří:

  ```
  <, >, =, <=, >=, !=, AND, OR
  ```

  ```
  {billingPlan:price > 500 AND billingPlan:price < 1000}
  ```

- Omezení počtu vrácených záznamů - použijeme klíčové slovo LIMIT a poté uvedeme maximální počet požadovaných záznamů.

  ```
  LIMIT=5
  ```

- Stránkování - v případě, že chceme získat záznamy až od určitého pořadí, použijeme klíčové slovo OFFSET

  ```
  OFFSET=10
  ```

- Řazení - výsledky mohou být seřazeny za pomocí klíčového slova ORDERBY, za kterým následují čárkou oddělené názvy sloupců a směr, kterým chceme záznamy seřadit.

  ```
  ORDERBY client ASC, billingPlan:price DESC
  ```

3.3.2 Příklad výsledné URL adresy

- ```
  /search/product/?q=
  (id,name,billingPlan:price) {billingPlan:initialPrice > "14"}
  LIMIT=10 OFFSET=0 ORDERBY clients:name ASC, billingPlan:price
  DESC
  ```

3.4 Využití vyhledávacího API jako zdroje dat pro tabulky

V rámci tohoto úkolu bylo požadováno využít již implementované API z předchozího úkolu. Cílem bylo vytvořit vrstvu nad tímto vyhledáváním, která bude umět modifikovat příchozí data dle předepsaných pravidel. Jako příklad využití lze uvést modifikaci ceny produktu v závislosti na přihlášeném uživateli nebo obalení atributů kusem HTML kódu, neboť mohla být takto data zobrazena přímo uživateli bez nutnosti větších zásahů do JavaScriptu.
  ```

3.5 Vytvoření filtrů pro správu newsletterů

Jednalo se o implementaci možnosti přiřadit ke každému dostupnému newsletteru v systému určitou podmínku, za které bude uživatel k newsletteru přihlášen nebo naopak odhlášen, a to za pomocí syntaxe přívětivé pro běžného uživatele. Následně by byla tato podmínka byla pravidelně vyhodnocována - systém by vyhledal všechny odpovídající uživatele a provedl příslušnou akci (přihlášení nebo odhlášení z newsletteru). Předpokladem bylo, aby bylo možné zapisovat složitější podmínky, tedy využívat operátory AND a OR.

3.5.1 Požadované funkce

- Vyhledat uživatele, kteří si (ne)zakoupili daný produkt

  ```
  USER HAS [NOT] PRODUCT "SomeProduct"
  ```

- Vyhledat uživatele, kteří (ne)vrátili daný produkt

  ```
  USER HAS [NOT] RETURNED PRODUCT "SomeOtherProduct"
  ```

4 Zvolený postup řešení zadaných úkolů

4.1 Seznámení se s PHP frameworkem Symfony a testovacím frameworkem Codeception

Symfony je open source PHP framework firmy Sensio Labs, vycházející z návrhového vzoru MVC[4] a sloužící pro rychlejší a pohodlnější vývoj webových aplikací. Programátor se může více soustředit na logiku aplikace, než na rutinní úlohy, které jsou zajišťovány knihovnami frameworku za využití best practices a standardů.

Codeception je moderní testovací framework pro PHP, inspirován BDD. Poskytuje možnost psaní akceptačních, funkčních i unit testů. Je založen na testovacím frameworku PHPUnit.

4.2 Testování webových API

Codeception podporuje dva formáty testů:

- Cept - formát založený na scénáři - jeden PHP soubor reprezentuje jeden konkrétní testovací scénář.
- Cest - formát založený na využití třídy - jeden PHP soubor obsahuje jednu třídu, která reprezentuje kolekci testů, týkajících se určitého problému. Každý test je reprezentován jednou třídní funkcí. Existují zde dvě speciální funkce *_before(ApiTester $I)* a *_after(ApiTester $I)*, které se volají před každým, resp. po každém testu.

Jelikož byly ostatní testy v projektu psány ve formátu Cest, bylo mi doporučeno v této konvenci pokračovat. Tento způsob se mi zdál také uživatelsky přívětivější.

V projektu se dosud žádné API testy nevyskytovaly, musel jsem proto vytvořit novou sadu testů. To se provede příkazem

```
php codecept.phar generate:cest api User
```

V tomto příkazu musí být uveden mimo formátu také název první kolekce testů. Jako první jsem vytvořil kolekci testů týkajících se uživatele. Pro tyto účely musel být v systému vytvořen nový uživatel, aby bylo možné kontrolovat správnou činnost API.

První API, pro něhož jsem testy vytvářel, sloužilo k výpisu profilu uživatele. Ve vygenerované třídě *UserCest* jsem proto vytvořil funkci *testListUserProfile(ApiTester $I)*. Test provedu následujícím způsobem:

1. Zavoláním funkce *wantTo($text)* na instanci třídy *ApiTester* nastavím stručný popis prováděného testu. Tento popis se bude následně vypisovat v konzoli při provádění jednotlivých testů.
2. Zavoláním funkce *sendGET($url)* se provede GET požadavek na zadanou URL adresu.

3. Pomocí funkce *amOnPage($url)* ověřím, zda se nacházím na správné URL. V tomto případě by měla být shodná s URL, na kterou se zasílal GET požadavek.

4. Funkcí *canSeeResponseCodeIs($number)* zjistím, zda požadavek proběhl úspěšně, tedy zda server odpověděl kódem 200.

5. Ověřím, zda tělo odpovědi obsahuje data ve formátu JSON funkcí *seeResponseIsJson()*.

6. Porovnám, zda je vrácený JSON řetězec korektní, tedy odpovídá zadanému vzoru. V tomto případě musela odpověď obsahovat atribut status=ok. K tomuto slouží funkce *seeResponseContainsJson($value)*. Funkci můžeme předat asociativní pole nebo objekt a framework provede při testu automatickou serializaci do formátu JSON.

```
public function testListUserProfile(ApiTester $I) {
      $I->wantTo("Test API for list user profile");
      $I->sendGET("URL");
      $I->amOnPage("URL");
      $I->canSeeResponseCodeIs(200);
      $I->seeResponseIsJson();
      $I->seeResponseContainsJson(['status' => 'ok']);
}
```

Výpis 1: Testování API pro výpis profilu uživatele

Výsledek testu závisí na tom, zda všechny výše uvedené kroky budou vyhodnoceny kladně. Pokud je některý krok vyhodnocen záporně, je v konzoli vypsán název neúspěšně provedené akce a test již dále nepokračuje.

Stejný postup jsem zvolil pro vytvoření dalších API testů. Jelikož měly všechny scénáře většinou stejný průběh, který se lišil jen v závislosti na použité metodě HTTP požadavku, vytvořil jsem ve třídě *ApiTester* pro testování GET požadavku funkci *testGetRequest($wantTo, $url)*, která slouží jako substituce všech výše uvedených kroků. V dalších testech lze pak pro testování stejného scénáře s jinou URL tuto funkci využít.

Pro metodu POST jsem vytvořil funkci *testPostRequest($wantTo, $url, array $data)*, která otestuje téměř stejný scénář jako předcházející funkce, pouze s rozdílem, že nebude volat funkci *sendGET($url)*, ale zavolá *sendPOST($url, $data)*.

```php
public function testPostRequest($wantTo, $url, array $data) {
      $this->wantTo($wantTo);
      $this->sendPOST($url, $data);
      $this->canSeeResponseCodeIs(200);
      $this->seeResponseIsJson();
      $this->seeResponseContainsJson(['status' => 'ok']);
      $this->amOnPage($url);
}
```

Výpis 2: Testování POST požadavku

4.3 Vytvoření vyhledávacího API

Základním kamenem a možná zpočátku nejsložitějším úkolem byla nutnost rozparsovat zadaný dotaz, především jeho část s podmínkami. Touto částí jsem také práce započal. Vytvořil jsem třídu *Where*, reprezentující podmínkovou část dotazu a funkci *parseCondition*, ve které bude samotné parsování probíhat. Celá podmínka bude reprezentovaná stromovou strukturou, kde jednotlivé uzly bude reprezentovat třída *WherePart*. Funkce *parseCondition* má za úkol projít všechny znaky v zadaném řetězci (podmínce) a každý z nich vyhodnotit. Snažil jsem se najít nějaké efektivnější řešení, kde by stačilo využít například pouze regulární výraz, ale bohužel jsem na něj nepřišel. Počítá se tedy s těmito situacemi:

- Aktuálním znakem je levá kulatá závorka - v tomto případě se pravděpodobně jedná o začátek vnořené podmínky. Vytvořil jsem funkci *findPairBracketIndex*, která vrátí pozici, na které se nachází párová závorka, pokud existuje. Na základě této pozice zavolám znovu, rekurzivně, funkci *parseCondition*, které předám řetězec mezi oběma závorkami. Výsledek uložím do stromu a nakonec se v řetězci posunu na pozici párové závorky, jelikož část řetězce mezi závorkami je již zpracovaná.
- Aktuálním znakem je mezera - očekávám operátor AND nebo OR. Zjistím tedy, který z nich následuje, vložím je do stromu a posunu se o příslušný počet znaků dále.
- Pokud nenastane žádný z předchozích případů, je možné očekávat podřetězec typu *klíč=hodnota*. Pro parsování a validaci tohoto podřetězce jsem využil regulární výraz. Následně se posunu v řetězci dále o délku nalezeného podřetězce a pokračuju v parsování.

Ukázku zdrojového kódu parsování podmínkové části Where je možné nalézt v příloze B

Po zpracování podmínkové části jsem přešel ke zpracování selekce sloupců. K tomuto účelu jsem vytvořil třídu *Select*, která obsahuje rovněž funkci *parse*, vracející instanci třídy *Select*, obsahující pole sloupců. Jednotlivé sloupce bude reprezentovat třída *Column*. Pro parsování sloupců jsem využil regulární výraz

```
/(?J)(^((?P<function>[^\s]+)\(((?P<alias>[^\s\.]+)\.)?
((?P<joinWith>[^\s\.]+)\.)?(?P<column>[^\s]+)\))$)|(^(
(?P<alias>[^\s\.]+)\.)?((?P<joinWith>[^\s\.]+):)?(?P<column>
[^\s]+)$)/
```

V regulárním výrazu je počítáno s možností volání funkcí s parametrem názvu sloupce, nicméně to nebyla tolik důležitá funkcionalita a v době mé praxe tak nebyla implementována.

Stejný postup jsem následně zpětně využil v podmínkové části, jelikož se tam sloupce také vyskytují.

Jelikož se funkcionalita začala pomalu rozrůstat, vytvořil jsem třídu *NQLQuery*, jejíž název je odvozen od interního pojmenování vytvořeného dotazovacího jazyka NQL. Ta bude zaobalovat parsování celého dotazu vlastní funkcí *parse*, která se bude starat o rozparsování jednotlivých částí dotazu.

Následně jsem pomocí regulárních výrazů zanalyzoval i části dotazu *Limit, Offset a OrderBy*.

Po kompletní analýze dotazu jsem se dostal ke druhé části úkolu, kterou je překonvertování mnou vytvořené stromové struktury do jednoho z již existujících dotazovacích jazyků. Jelikož se v projektu využívá pouze ORM framework Doctrine[2], rozhodl jsem se jej využít také, ačkoliv takto dojde ke dvojímu parsování. Tento framework využívá vlastní dotazovací jazyk DQL, který je podobný standardnímu SQL a snaží se být nezávislý na konkrétním databázovém systému. Odpadá zde tedy případná nutnost úpravy či duplikování části zdrojového kódu při využití jiného databázového systému. Pomocí existující třídy z Doctrine - *QueryBuilder* jej tedy do jazyka DQL převedu. K tomuto účelu bude sloužit nová třída *DQLConverter* s funkcí *convert*, přijímající parametr *NQLQuery*, kterým je zanalyzovaný dotaz.

V prvním kroku je potřeba načíst seznam dostupných entit (tabulek) a porovnat jej s entitou, kterou chceme prohledávat. Pokud není entita nalezena, vyvolám výjimku. Při této příležitosti bylo možné vytvořit také seznam ignorovaných entit z důvodu prevence před možným zneužitím a prohledáním tabulek, které jsou nepřístupné běžným uživatelům.

Poté vytvořím instanci třídy *QueryBuilder* a postupně procházím sloupce selekce, které do QueryBuilderu přidávám funkcí *addSelect*, funkcí *from* předám název prohledávané entity a poté do QueryBuilderu zapracovávám podmínkovou část dotazu.

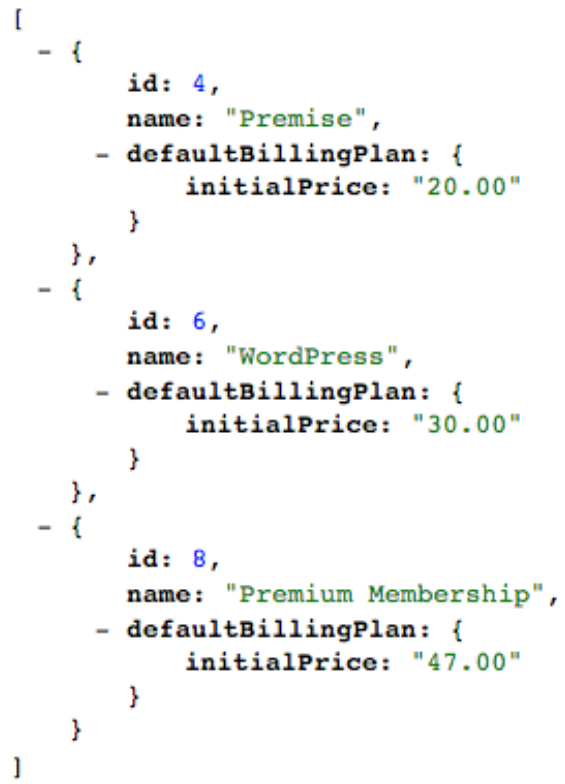
Zde to bylo již o něco složitější, jelikož jsem musel vytvořit rekurzivní funkci, která projde celý strom podmínek a zpětně strom převede na řetězec, nahradí hodnoty prvků typu "klíč=hodnota"zástupným symbolem "?"a zároveň tyto hodnoty uloží do samostatného pole, aby je bylo možné k takto parametrizovanému dotazu později přiřadit. Celý takto zkonstruovaný řetězec předáme QueryBuilderu funkcí *where* a parametry iterativně funkcí *setParameter*.

Jelikož je možné v původním dotazu přistupovat pomocí cizího klíče i k atributům jiné entity a framework Doctrine to sám od sebe neumí, bylo nutné vytvořit jakýsi "HashSet"všech sloupců odkazujících se na jiné tabulky. S těmito tabulkami je poté potřeba iterativně provést spojení, funkcí "join".

V poslední řadě se pak nastaví parametry řazení výsledků funkcí *addOrderBy*, dále limit výsledků a offset funkcí *setMaxResults*, respektive *setFirstResult*.

Jelikož využíváme selekci konkrétních atributů, výsledkem je po zavolání *getQuery()->getResult()* na instanci QueryBuilderu pole asociativních polí, které obsahují hodnoty požadovaných atributů a jejich názvy. V opačném případě by bylo výsledkem pole konkrétních objektů (entit) dle ORM.

Toto pole následně předám konstruktoru třídy *JsonResponse*, jejíž instance je výsledkem vykonávané akce kontroléru.



Obrázek 2: Ukázka výstupu vyhledávacího API

Pro porovnání, jak parsování ovlivnilo dotazování, jsem provedl měření času vykonání výsledného dotazu v jazyce DQL a celkového času včetně parsování původního dotazu.

Dotaz v jazyce NQL:

```
(id, name, defaultBillingPlan.initialPrice){defaultBillingPlan:initialPrice > 10
AND defaultBillingPlan:initialPrice < 50} LIMIT=10 OFFSET=0 ORDERBY
defaultBillingPlan:initialPrice ASC
```

Stejný dotaz v jazyce DQL:

```
SELECT product.id, product.name, product.defaultBillingPlan FROM
AppBundle:product product INNER JOIN product.defaultBillingPlan
defaultBillingPlan WHERE defaultBillingPlan.initialPrice>10 AND
defaultBillingPlan.initialPrice<50 ORDER BY defaultBillingPlan.initialPrice ASC
```

Výsledek měření:

- Čas vykonání DQL dotazu: 4,37 ms
- Celkový čas včetně parsování původního dotazu: 6,05 ms

Z měření vyplývá, že samotné parsování trvalo v tomto případě 1,68 ms, což je zanedbatelné.

4.4 Využití vyhledávacího API jako zdroje dat pro tabulky

K tomuto úkolu jsem dostal zadaný i orientační postup, s pomocí kterého bych měl úkol vyřešit. Řešení by mělo spočívat ve vytvoření TWIG[9] šablony, kde by byl pro každý atribut objektu z výsledku vyhledávání, jehož hodnota se bude upravovat, definovaný samostatný blok. Při iteraci ve výsledku vyhledávání by se pro každý atribut objektu zavolala funkce *renderBlock*, která by daný blok dle šablony vygenerovala. Takto upravený výsledek vyhledávání by byl poté vrácen jako JSON response.

Každou tabulku bude reprezentovat třída rozšiřující abstraktní třídu *BaseGrid*, ve které je definováno, jaká šablona se má použít pro zpracování výsledku vyhledávání. Následně jsem vytvořil třídu *GridManager* reprezentující manažera těchto tabulek. Jelikož by se v systému měla vyskytovat jen jedna instance této třídy, zaregistroval jsem ji jako službu. Každá tabulka musí být do GridManageru přidána funkcí *addGrid($alias, $grid)*.

Prvním krokem v kontroleru je analýza zadané vyhledávací podmínky pomocí služby *Search* z vyhledávacího API. Zde muselo být vyhledávání kvůli potřebě pracovat se sloupci ze selektivní části drobně upraveno, aby nevracelo ihned výsledek, ale napřed instanci třídy *NQLQuery* obsahující rozparsovaný dotaz. Do této třídy byla doplněna funkce *getQueryBuidler*, která vrátí Doctrine *QueryBuilder*, pomocí něhož je poté možné funkcí *getResult()* získat požadovaná data.

Poté na instanci služby *GridManager* zavolám funkci *convertEntitiesToArray($result, $columns)*, které předám výsledek vyhledávání a pole všech sloupců obsažených v části selekce. Návratovou hodnotou této funkce je pole s již modifikovanými hodnotami atributů, které následně vrátím jako JSON response.

Ve funkci *convertEntitesToArray()* nejprve provedu kontrolu vstupních parametrů. Následně zavolám funkci *getGridNameFromEntities($entities)*, která na základě prvního prvku v předaném poli vrátí název třídy, jejíž instancí tento prvek je. Ten předám funkci *getGrid($name)*, která vrátí instanci třídy rozšiřující *BaseGrid*. Pokud neexistuje, vyvolám výjimku.

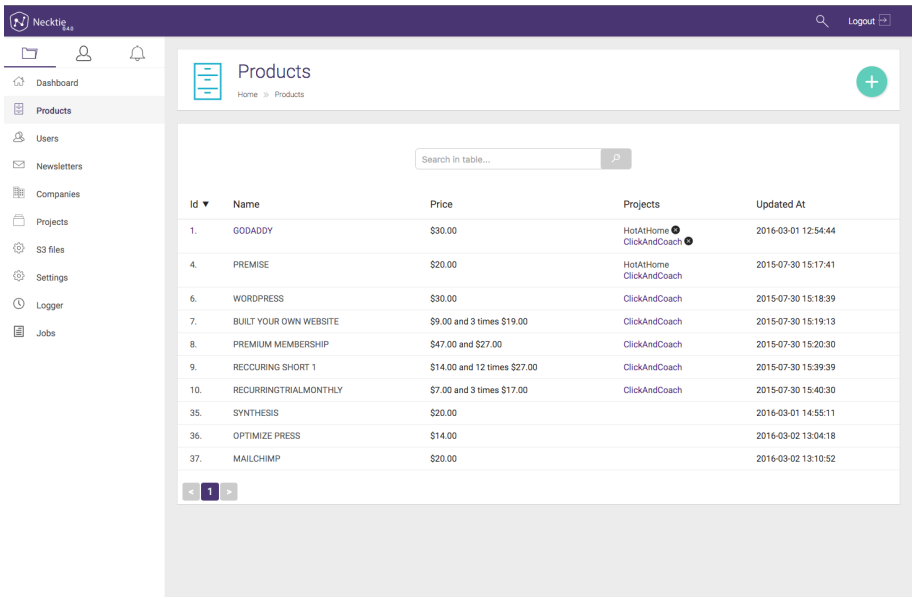
```
private static function getObject($entity, $values, $curValueIndex = 0)
{
      if($curValueIndex == 0) {
            $values = explode(":", $values);
      }

      try {
            $obj = ObjectMixin::get($entity, $values[$curValueIndex]);
            if ($curValueIndex == count($values) - 1) {
                  return $obj;
            } else if ($obj instanceof PersistentCollection) {
                  $items = [];
                  foreach ($obj as $item) {
                        $items[] = array($values[$curValueIndex + 1] => self::
                            getObject($item, $values, $curValueIndex + 1));
                  }
                  return $items;
            } else if (is_object($obj)) {
                  return self::getObject($obj, $values, $curValueIndex + 1);
            } else {
                  return $obj;
            }
      } catch (\Exception $ex) {
            return "";
      }
}
```

Výpis 3: Získání hodnoty proměnné z entity případně vnořené entity pomocí dvojtečkové notace

Dále postupně projdu předané pole objektů, resp. výsledku vyhledávání a pro každý sloupec ze selekce zavolám funkci *getValue($entity, $column*, která má za úkol z daného objektu získat požadovaný atribut. Jelikož zde může být v rámci selekce požadován atribut, který je atributem jiného objektu, jehož referenci tento objekt uchovává, je nutné tuto funkci volat rekurzivně. Následně je nutné načíst definovanou šablonu tabulky skrze službu twig funkcí *loadTemplate($template)* a pokud šablona obsahuje blok se stejným názvem jako je název atributu objektu, zavolá se nad šablonou funkce *renderBlock($block, $parameters)*, která vygeneruje obsah daného bloku a tuto hodnotu použiju jako novou hodnotu atributu původního objektu.

```
[
  - {
      id: "<a title="Go to Premise" href="/app_dev.php/admin/product/4">4.</a>",
      name: "<a title="Go to Premise" href="/app_dev.php/admin/product/4"
      class="uppercase">Premise</a>",
      defaultBillingPlan:initialPrice: "$20.00"
    },
  - {
      id: "<a title="Go to WordPress" href="/app_dev.php/admin/product/6">6.
      </a>",
      name: "<a title="Go to WordPress" href="/app_dev.php/admin/product/6"
      class="uppercase">WordPress</a>",
      defaultBillingPlan:initialPrice: "$30.00"
    },
  - {
      id: "<a title="Go to Premium Membership"
      href="/app_dev.php/admin/product/8">8.</a>",
      name: "<a title="Go to Premium Membership"
      href="/app_dev.php/admin/product/8" class="uppercase">Premium
      Membership</a>",
      defaultBillingPlan:initialPrice: "$47.00 and $27.00"
    },
]
```


Obrázek 3: Ukázka výstupu API pro tabulky



Obrázek 4: Ukázka tabulky využívající toto API

4.5 Vytvoření filtrů pro správu newsletterů

Zpočátku jsem s kolegy vedl dlouhé diskuze ohledně algoritmu, který by dokázal nejlépe zastat tuto funkcionalitu a zároveň byl co nejméně náročný na výpočetní výkon a objem přenášených dat, jelikož je do blízké budoucnosti počítáno s vysokým počtem uživatelů, pohybujícím se okolo jednoho milionu a zpracování takto vysokého počtu záznamů je časově náročné. Dle zběžné analýzy se odhad pohyboval okolo 5 hodin. Proto bylo nutné, aby se zadaný řetězec, tedy vstup uživatele nejprve uložil do databáze a následně se zpracoval Postupně jsem tedy

dospěl k řešení, které vám budu prezentovat.

Předtím, než přišel na řadu samotný výběr uživatelů z databáze, bylo nutné analyzovat zadaný řetězec. Jelikož měla validace probíhat interaktivně ze strany JavaScriptu, nebylo nutné nějak zvlášť uživatelský vstup kontrolovat na straně serveru.

Když jsem se zaměřil na možné uživatelské vstupy, a také na funkce uvažované v budoucnu (např. výběr uživatelů starších než zadaný počet let), zjistil jsem, že každou funkci, kterou může uživatel zadat, lze zapsat z určitého pohledu jako podmínku a využít takto pro parsování celého vstupu již existující část vyhledávacího API. Zde bylo jediným nutným úkonem vymyslet způsob, jak takovouto funkci jako podmínku přepsat.

Nejjednodušším řešením bylo z mého pohledu využít regulárních výrazů. Vytvořil jsem proto abstraktní třídu *NewsletterAbstractFunction*, která v sobě nese proměnnou *pattern* představující příslušný regulární výraz a abstraktní funkci *getUsers(QueryBuilder $qb, $value, $operator)*, která bude poté v každé podporované funkci implementovaná - nastaví příchozí instanci *QueryBuilder* tak, aby byl poté schopen vrátit odpovídající uživatele. Dále třída obsahuje funkci *getHash*, která vrátí jedinečný identifikátor dané funkce. Za dostačující považuji ořezání regulárního výrazu o speciální znaky tak, aby výsledný hash bylo možné využít jako platný název sloupce při parsování podmínky.

Nyní jsem vytvořil třídu *NewsletterFilter*, která bude plnit hlavní funkci při filtrování uživatelů. Třída přijímá v konstruktoru *EntityManager*, ID newsletteru, jehož filtr bude zpracovávat a také název zamýšlené akce, kterou má nad uživateli provádět (subscribe, unsubscribe).

Hlavní myšlenkou tohoto způsobu řešení bylo omezit počáteční seznam uživatelů, nad kterými se bude filtr provádět tak, že pokud budeme provádět akci "subscribe", jako základ vybereme všechny uživatele, kteří ještě k danému newsletteru přihlášení nejsou, tedy všechny "unsubscribed"a naopak. Toto řešení bude efektivní zřejmě až počínaje v pořadí druhým filtrováním, kde takto dostaneme už jen nezbytný počet uživatelů. Jako předpis, jak takovéto uživatele získat, jsem vytvořil pro obě možné akce *"baseQueryBuilder"*, ve kterém je tato základní selekce nastavena. Ten poté budu předávat jako parametr každé z předem definovaných funkcí, které rozšiřují třídu *AbstractNewsletterFunction*.

```
class HasBoughtFunction extends NewsletterAbstractFunction
{
    protected $pattern = '/HAS\s(?<not>NOT\s)?BOUGHT\sPRODUCT\s(?<value
        >\"[^\."]+\"| [^\s]+)/';

    public function getUsers(QueryBuilder $qb, $value, $operator = "=") :
        QueryBuilder
    {
        switch($operator) {
            case "=":
                return $this->getUsersWhichBoughtProduct($qb, $value);
            case "!=":
                return $this->getUsersWhichNotBoughtProduct($qb, $value);
            default:
                return $qb;
        }
    }

    private function getUsersWhichNotBoughtProduct(QueryBuilder $qb, $value)
    {
        $qb2 = $qb->getEntityManager()->createQueryBuilder();
        return $qb
            ->andWhere($qb->expr()->notIn('user.username',
                $qb2->select('user2.username')
                    ->from('NecktieAppBundle:User', 'user2')
                    ->join('NecktieAppBundle:UserAction', 'userAction', 'WITH', '
                        userAction.user = user AND userAction.action = \'bill\'
                        AND userAction.entity = \'product\'')
                    ->join('NecktieAppBundle:Product', 'product', 'WITH', '
                        userAction.entityId = product.id AND product.name = :
                        productName')
                    ->getDQL()
            ))
            ->setParameter('productName', $value);
    }
```

Výpis 4: Ukázka funkce pro newsletter filtr

Pokud na již takto vytvořenou instanci třídy textitNewsletterFilter zavolám funkci filter($query) a předám ji vstup od uživatele, provedou se následující kroky:

- Zavolá se funkce *translateQueryToNQL($query)*, která projde postupně všechny dostupné funkce a regulárním výrazem nahradí všechny jejich výskyty za podmínky. Z výsledného dotazu se poté provede pokus o rozparsování pomocí třídy *NQLQuery*.
- Zavolá se funkce *evaluateConditions($conditions)*, které předáme zanalyzovaný strom podmínek. V jejím těle proběhne průchod tímto stromem a pro každou podmínku je zavolána funkce *getUsers* příslušné *NewsletterFunction* a provede se dotaz na databázi, jehož výsledek se k podmínce přiřadí.
- V tuto chvíli mám k dispozici výsledky jednotlivých funkcí oddělené operátory AND, případně OR a je nutné tyto výsledky na základě operátorů sjednotit. Bude se tedy volat funkce *getResult($conditions)*, která pro každý ze dvou operátorů provedu samostatný průchod stromem podmínek a vyhodnotím vždy dva s operátorem sousedící seznamy uživatelů. V případě operátoru AND volám funkci *findSameUsers($listA,$listB)*, pro operátor OR funkci *splitUsers($listA,$listB)*. Obě funkce vrátí vždy výsledný seznam uživatelů. Operátor AND má větší prioritu nad operátorem OR, a proto musí být zpracován jako první.

V tuto chvíli je vyhodnocování dokončeno a mám k dispozici výsledný seznam uživatelů, které je potřeba přihlásit či odhlásit od newsletterů.

Pro tuto akci existuje v systému služba *NewsletterService*, která obsahuje funkce *subscribeUser($user)* a *unsubscribeUser($user)*. Nyní tedy stačí jen iterativně projít výsledný seznam uživatelů a na základě definované akce provést subscribe či unsubscribe.

Jelikož bude tato akce prováděna jako CRON úloha, je zapotřebí vytvořit příkaz, který bude spustitelný v rámci Symfony konzole. V rámci tohoto příkazu se z databáze načte seznam všech dostupných newsletter filtrů z tabulky *NewsletterFilter* a pro každý z nich a pro každou z akcí se vykoná výše uvedený algoritmus.

5 Získané a využité zkušenosti a znalosti

Během vykonávání praxe jsem využil znalosti ze všech programovacích předmětů, především koncepci a styl programování. Zdokonalil jsem se v práci s programovacím jazykem PHP, naučil se základům hojně využívaného frameworku Symfony, verzovacího systému GIT[3], naučil se pracovat v týmu a nabyl schopností vést aktivně týmovou komunikaci. Přínosem byl pro mě také předmět Správa operačních systémů, jelikož jsem tak v rámci praxe mohl aktivně pracovat v prostředí operačního systému Debian.

6 Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení

Během bakalářské praxe jsem značně prohloubil své znalosti programování v jazyce PHP a obecně programování webových aplikací, se kterými jsem měl zpočátku jen okrajové zkušenosti. Velmi pozitivně hodnotím nabyté zkušenosti práce ve skutečné firmě a na skutečných problémech, díky kterým jsem mohl ve firmě po odpracování dnů v rámci bakalářské praxe zůstat, zařadit se mezi běžné zaměstnance a podílet se na dalším rozvoji projektů.

Literatura

[1] Codeception - BDD-style PHP testing [online]. [cit. 2016-01-31]. Dostupné z: http://codeception.com/

[2] Doctrine Project [online]. [cit. 2016-03-16]. Dostupné z: http://www.doctrine-project.org/

[3] Git [online]. [cit. 2016-04-05]. Dostupné z: https://git-scm.com/

[4] Model-View-Controller [online]. [cit. 2016-04-05]. Dostupné z: https://msdn.microsoft.com/en-us/library/ff649643.aspx

[5] Moderní podnikatel [online]. [cit. 2016-01-31]. Dostupné z: http://modernipodnikatel.cz/

[6] PHP: Hypertext Preprocessor [online]. [cit. 2016-03-16]. Dostupné z: http://php.net/

[7] PHPUnit - The PHP Testing Framework [online]. [cit. 2016-04-05]. Dostupné z: https://phpunit.de/

[8] Symfony, High Performance PHP Framework for Web Development [online]. [cit. 2016-01-31]. Dostupné z: https://symfony.com/

[9] Twig - The flexible, fast and secure PHP template engine [online]. [cit. 2016-04-05]. Dostupné z: http://twig.sensiolabs.org/

A Ukázka konzolového výstupu testu v Codeception

```
Codeception PHP Testing Framework v2.1.4
Powered by PHPUnit 4.8.21 by Sebastian Bergmann and contributors.

  Group 'failed' is empty, no tests are loaded

Api Tests (9) ------------------------------------------------------------------
Modules: \Helper\Api, REST, PhpBrowser, ConfigHelper
--------------------------------------------------------------------------------
Test list user profile (UserCest::testListUserProfile)
Scenario:
* I send get "/api/user/profile"
  [Request headers] []
  [Request] GET /api/user/profile
  [Response] {"status":"ok","data":{"id":76,"firstName":"George","lastName":"Cook",
"avatarPhoto":"","email":"apitester@webvalley.cz","username":"apitester","preferred
Units":"imperial","dateOfBirth":"1979-01-22","birthDateStyle":3,"facebookId":null,"
twitterId":null,"facebookAccessToken":null,"twitterAccessToken":null,"lastPasswordC
hange":null,"location":"London, UK","youtubeLink":"","snapchatNickname":"","privacy
Settings":{"publicProfile":true,"displayFullName":true,"displayEmail":true,"birthDa
teStyle":3,"displayLocation":false,"displaySocialMedia":true,"displayForumActivity"
:true,"displayProgressGraph":true},"profilePhotoOriginal":"https://flofit-prod.s3.a
mazonaws.com/profile_photos_original/56fce42316325.jpeg","profilePhotoCropped":"htt
ps://cdn.flofit.com/profile_picture_cropped/rc/jgkraLTk/56fce42316325.jpeg","cropSt
artX":0,"cropStartY":0,"cropSize":450}}
  [Headers] {"Cache-Control":["no-cache"],"Content-Type":["application/json"],"Date
":["Thu, 31 Mar 2016 11:06:04 GMT"],"Server":["nginx/1.6.2"],"Vary":["Authorization
"],"X-Powered-By":["PHP/5.6.19"],"Content-Length":["866"],"Connection":["keep-alive
"]}
  [Status] 200
* I am on page "/api/user/profile"
  [Page] /api/user/profile
  [Response] 200
  [Request Cookies] []
  [Response Headers] {"Cache-Control":["no-cache"],"Content-Type":["application/jso
n"],"Date":["Thu, 31 Mar 2016 11:06:04 GMT"],"Server":["nginx/1.6.2"],"Vary":["Auth
orization"],"X-Powered-By":["PHP/5.6.19"],"Content-Length":["866"],"Connection":["k
eep-alive"]}
* I can see response code is 200
* I see response is json
* I see response contains json {"status":"ok"}
PASSED

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
--------------------------------------------------------------------------------


Time: 2.62 seconds, Memory: 10.00Mb

OK (1 test, 6 assertions)
Martin-Mac-mini:FlofitVenice martinmatejka$ php bin/codecept run api UserCest:testL
istUserProfile -v
Codeception PHP Testing Framework v2.1.4
```

B Ukázka parsování podmínkové části dotazu

```
class Where
{
   ...

   private static function parseCondition($str) {
         $parts = array();

         // REMOVE TRAILING SPACES
         $str = trim($str);

         // LOOP THROUGH ALL CHARACTERS
         for ($i = 0; $i < strlen($str); $i++) {

               // IF CHARACTER IS LEFT BRACKET, FIND PAIR BRACKET AND
                  RECURSIVELY FIND CONDITIONS WITHIN THESE BRACKETS
               if ($str[$i] === "(") {

                     $pairBracketIndex = self::findPairBracketIndex($str, $i + 1)
                        ;

                     if ($pairBracketIndex > 0) {
                           $part = new WherePart();
                           $part->type = WherePartType::SUBCONDITION;

                           $subCondition = substr($str, $i + 1, $pairBracketIndex
                                - $i - 1);
                           $part->baseExpr = trim($subCondition);
                           $part->subTree = self::parseCondition($subCondition);

                           $parts[] = $part;

                           $i = $pairBracketIndex;
                           continue;
                     } else {
                           throw new SyntaxErrorException("Missing pair bracket")
                              ;
                     }
```

```
} // IF THERE IS SPACE - IT IS SIGN THAT THERE WILL BE "AND" OR "
    OR" CONDITION
else {
      if ($str[$i] === " ") {
            if (trim(substr($str, $i, 4)) === Operator:: AND) {
                  $part = new WherePart();
                  $part->type = WherePartType::OPERATOR;
                  $part->value = Operator:: AND;

                  $parts[] = $part;
                  $i = $i + 3;
            } else {
                  if (trim(substr($str, $i, 3)) === Operator:: OR)
                      {
                        $part = new WherePart();
                        $part->type = WherePartType::OPERATOR;
                        $part->value = Operator:: OR;

                        $parts[] = $part;
                        $i = $i + 2;
                  }
            }
      } // OTHERWISE WE EXPECTING KEY=VALUE
      else {
            $match = array();
            $wasFound = preg_match(self::$regKeyOpValue, substr(
                $str, $i), $match);

            if ($wasFound) {
                  $part = new WherePart();
                  $part->type = WherePartType::CONDITION;
                  $part->key = Column::parse($match['key']);
                  $value = $match['value'];
                  if(StringUtils::startsWith($value,'"') &&
                      StringUtils::endsWith($value, '"'))
                        $value = StringUtils::substring($value, 1,
                            StringUtils::length($value) - 1);
                  $part->value = $value;
```

```
                        $part->operator = $match['operator'];

                        $parts[] = $part;
                        $i = $i + strlen($match[0]) - 1;
                    } else {
                        $part = new WherePartType();
                        $part->type = "UNKNOWN";
                        $part->value = $str[$i];

                        $parts[] = $part;

                        $context = self::getErrorContext($str, $i);

                        throw new SyntaxErrorException(
                            "Unrecognized char sequence at '".$context['
                                subString']."' starting from index ".
                                $context['errorAt']
                        );
                    }

                }
            }
        }
        return $parts;
    }
}
```

Výpis 5: Ukázka parsování podmínkové části dotazu