FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Datový sklad ČVUT - řízení přístupu k datům
Student:	Bc. Cyril Černý
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je analyzovat, navrhnout a implementovat funkční prototyp infrastruktury, která umožní přístup k dokumentaci datových tržišť datového skladu ČVUT (DW), a na základě rolí přístup k samotným datům. Návrh a implementace navazuje na prototyp datového skladu ČVUT, dokumentační portál EBIE a stávající infrastrukturu rolí v Identity Manager (IdM) ČVUT.

1. Vypracujte návrh struktury technických rolí pro přístup k datům v DW, který bude konsistentní se stávajícími rolemi IdM ČVUT.
2. Analyzujte a navrhněte řešení, které umožní na základě těchto rolí:
  - přistoupit k dokumentaci datových tržišť v portálu EBIE
  - získat informaci o REST API, které umožní data čerpat
  - získat data podle role
3. Navržené řešení implementujte jako funkční prototyp, řádně otestujte a zdokumentujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D. vedoucí katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. děkan

V Praze dne 12. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Datový sklad ČVUT - řízení přístupu k datům

Bc. Cyril Černý

Katedra softwarového inženýrství
Vedoucí práce: Michal Valenta

5. května 2018

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Michalu Valentovi, Ph.D. za jeho trpělivost a vždy pozitivní náladu při vedení této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2018 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Cyril Černý. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Černý, Cyril. *Datový sklad ČVUT - řízení přístupu k datům.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práce se zabývá řízením přístupu k REST API poskytující data z datového skladu. Je diskutována podoba nové technické role a řízení přístupu na jejím základě. Výsledkem práce je funkční autorizační vrstva pro existující aplikaci.

Klíčová slova EBIE, REST API, Autentizace, Autorizace, Ruby, Sinatra, Rack middleware, Technické role

Abstract

This thesis is about access control for REST API providing data from data warehouse. New role design and its use for access control is discussed. The result of this thesis is fully working authorization layer for existing aplication.

Keywords EBIE, REST API, Authentication, Authorization, Ruby, Sinatra, Rack middleware, Technical roles

Obsah

Úvod **1**

1 Cíle práce **3**

2 Aktuální stav EBIE **5**
2.1 EBIE WEB 5
2.2 EBIE API 6

3 Technologie **9**
3.1 Programovací jazyk Ruby 9
3.2 Rack 11
3.3 Sinatra 13
3.4 Pliny 14

4 Analýza a návrh technických rolí **17**
4.1 Získání dat 17
4.2 Struktura role 17
4.3 Podoba nové role 18

5 Rešerše možných řešení **21**
5.1 Požadavky 21
5.2 Pundit 23
5.3 CanCanCan 24
5.4 Consul 26
5.5 Six 27
5.6 Shrnutí 28

6 Návrh řešení **31**
6.1 Základní filosofie 31
6.2 Zvolení nástroje pro autorizaci 33

6.3 Autentizace . . . 34
6.4 Autorizace . . . 35
6.5 Omezení výsledku . . . 39
6.6 Pomocné metody . . . 41

7 Implementace **43**
7.1 Autorizační modul . . . 44
7.2 Middleware . . . 45
7.3 Reprezentace uživatele . . . 47
7.4 Identifikátory organizačních jednotek . . . 49
7.5 Vytváření SQL dotazů . . . 51

8 Testování **55**
8.1 Automatické testy . . . 55
8.2 Vyzkoušení EBIE API . . . 57

9 Zhodnocení a další vývoj **61**
9.1 Mezipaměť . . . 61
9.2 Tvorba SQL dotazů . . . 61
9.3 Další práce na autorizaci . . . 62

Závěr **63**

Reference **65**

A Seznam použitých zkratek **67**

B Tipy při tvorbě endpointů **69**
B.1 Analýza autorizačních požadavků . . . 69
B.2 Analýza datamartu . . . 69
B.3 Definice endpointu . . . 71
B.4 SQL dotaz . . . 71

C Obsah přiloženého CD **73**

Seznam obrázků

2.1 Ilustrace přístupu k datům v EBIE. 6
3.1 Ilustrace zpracování požadavku přes Rack middleware 12
6.1 Základní koncept zpracování požadavku 32
8.1 Pokrytí kódu po testování 56
B.1 Schéma pro výsledky předmětů 70

Seznam ukázek kódu

1 Výsledek kontroly tokenu z OAAS Zuul 7
2 Ukázky syntaxe Ruby 10
3 Rozdíl mezi symbolem a řetězcem v Ruby 11
4 Primitivní Rack aplikace 12
5 Příklad velmi jednoduchého middlewaru 13
6 Sinatra DSL pro definici cest 14
7 Pundit definice pravidel 23
8 Pundit autorizace 24
9 CanCanCan definice způsobilostí uživatele 25
10 CanCanCan volání autorizace 25
11 Consul definice pravidel 26
12 Consul použití 27
13 Six definice pravidel 28
14 Six použití 28
15 Příklad definic cest v EBIE API 36
16 Nejednoznačnost v definicích cest 37
17 SQL dotaz v Ruby 40
18 Upravený SQL dotaz 40
19 Přihlášení uživatele 45
20 Autorizace podle parametrů 46
21 Middleware metoda `call` 46
22 Získání parametrů z požadavku 47
23 Inicializace uživatele 48
24 Kontrola rolí 49
25 Obsah souboru pro mapování 51
26 Pomocná metoda pro omezení fakult 52
27 Test při selhání externí služby 57
28 Možná volání EPIE API přes cURL 58
29 Kostra SQL pro výsledky předmětů 71

Úvod

Dnes je pojem datového skladu populární. Větší organizace generují velké množství dat a je škoda taková data nevyužít Proto existují nástroje pro jejich analýzu a vizualizaci. ČVUT v tomto není výjimkou a tak na fakultě informačních technologií vzniká datový sklad, který sdružuje data z více zdrojů. Vedle vlastního datového skladu také vznikají podpůrné aplikace pro dokumentaci a získání dat ze skladu.

Ne všechna data jsou veřejná. Není přípustné poskytovat data komukoliv. Vzniká tak potřeba pro řízení přístupu k datům. Dat je tolik, že není možné pověřeným osobám umožnit číst všechna data. Uživatel by tak byl zbytečně zahlcen obrovským množstvím zbytečných a irelevantních dat. Proto by přístup k datům měl být podmíněn nějakou rolí, která uživatele bude opravňovat číst menší množinu dat, která bude pro danou roli relevantní.

Nejprve si v práci určím cíle a představím projekt EBIE. V popisu EBIE se zaměřím převážné na záležitosti kolem řízení přístupu a v jakém rozsahu (a jestli vůbec) je již problematika řešena. Po uvedení do projektu stručně popíši klíčové technologie a nástroje použité v projektu.

Po představení technologií následuje první část řešení. Proběhne analýza existujících ČVUT technických rolí a bude vysloven návrh na podobu nové role pro použití v řešení řízení přístupu. Následně vznikne obecný návrh řešení problému který bude respektovat již použité technologie a novou roli. Tento návrh bude vzápětí implementován a nasazen. Nakonec zbude implementaci ověřit a otestovat.

Kapitola 1

Cíle práce

Tato práce má za cíl navrhout a vytvořit systém pro přístup k datům z datového skladu. Část dat je dostupná v dokumentačním portálu EBIE [1] v grafické podobě (různé vizualizace dat, tabulky). Dále existuje REST API, která předává data z datového skladu do portálu. Přístup k datům by měl být zabezpečen tak, aby je směli číst pouze pověřené osoby. Jako příklad pověřených osob jsou děkani nebo vedoucí kateder.

Cíle je možné rozdělit do několika tématických částí. Jedním cílem je analyzovat již existující systém rolí použitých na ČVUT. Zaměřit se mám na role technické, které jsou podobné byznys rolím. Po této analýze by měl vzniknout návrh na nové role, které musí splňovat náležitosti pro technickou roli ČVUT. Jde tedy o rozšíření rolí v manažeru identit ČVUT. Tyto role budou použity v dalších částech práce.

Dále je třeba se seznámit strukturou projektu EBIE. Tuto část jsem částečně vypracoval ve své bakalářské práci [2].Ve své bakalářské práci jsem se zabýval integrací služeb do portálu EBIE. Část zkušeností a vědomostí tedy mohu využít i v této práci. Jde převážně o způsoby autentizace a analýza byznys rolí. V souvislosti analýzy struktury EBIE se nově seznámím s REST API, která je klíčovým zdrojem dat pro portál EBIE.

Úkolem je vytvořit systém využívající navržené role k autentizaci a autorizaci uživatelů. První část tohoto cíle je návrh takového systému. Tedy jak využít dostupné informace k autorizaci uživatelů. Základní otázky pro systém jsou:

- Je možně uživatele identifikovat? (autentizace)
- Má uživatel oprávnění číst data? (autorizace)
- Jaká data smí uživatel číst?

Druhou částí je implementace takového systému do již existující struktury EBIE a její REST API. Po realizaci autorizace je možné zpřístupnit REST API ostatním lidem tak, aby si mohli zpracovávat data podle vlastních představ.

Mým osobních cílem je neměnit radikálně zavedenou strukturu projektů a co nejméně zasahovat do funkcionality, která nesouvisí s autorizací.

Vedlejším cílem práce je vytvořit dokumentaci, aby budoucí vývojáři pracující na EBIE mohli rozšiřovat její funkcionalitu snadněji a nemuseli hledat informace ve zdrojových kódech nebo experimentálně. Měl by tak vzniknout manuál pro vývojáře a současně také pro přímé uživatele REST API, aby věděli jaké podmínky musí splnit pro možnost získat data z datového skladu přes REST API.

Kapitola 2

Aktuální stav EBIE

EBIE (Extended Business Inteligence Encyclopedia)[1] je portál vnikající na fakultě informačních technologií. Není to zcela nový projekt a stále probíhá vývoj. EBIE není jedna samostatná aplikace. Skládá se z více funkčních celků. Základy všech dílčích aplikací jsou již vytvořeny. Původní autor komponent již na projektu nepracuje a tak na vývoji pokračuje hrstka studentů. V této kapitole čtenáře seznámím se strukturou a stavem EBIE a zaměřím se na řešení autentizace a autorizace v komponentách.

2.1 EBIE WEB

Hlavní účel webového portálu je snadno zpřístupnit data pověřeným osobám. EBIE tak nabízí mimo jiného takzvané reporty, které zobrazují užší množinu dat (například informace o výsledcích předmětů) v grafické podobě. Report obsahuje interaktivní grafy a uživateli umožňuje v prohlížeči filtrovat data podle potřeb.

Webová aplikace EBIE je vytvořena ve frameworku Ruby on Rails [3]. Slouží k zobrazení reportů a dokumentace datových zdrojů. Data získává z přidružené REST API.

EBIE podporuje základní autentizaci uživatelům s ČVUT účtem. Protože ale nejsou data veřejně dostupná, bylo třeba omezit přistup. K přihlašování a ověření identity je použit OAuth2 autorizační server Zuul [4]. OAuth autentizací jsem se již zabýval ve své bakalářské práci a proto ji zde nebudu detailněji popisovat. Po ověření uživatele získá aplikace access token, který následně používá při komunikaci s API.

Autorizace je řešena pouze přes „whitelist“. Tato metoda explicitně definuje množinou uživatelů, kteří mohou používat portál. Ostatním lidem je přistup zamítnut. Uživatelé v seznamu mají plný přístup k datům bez omezeni. Portál nerozlišuje typ uživatele nebo role, pouze zda je nebo není uveden v seznamu autorizovaných osob.

2.1.1 Správa obsahu EBIE

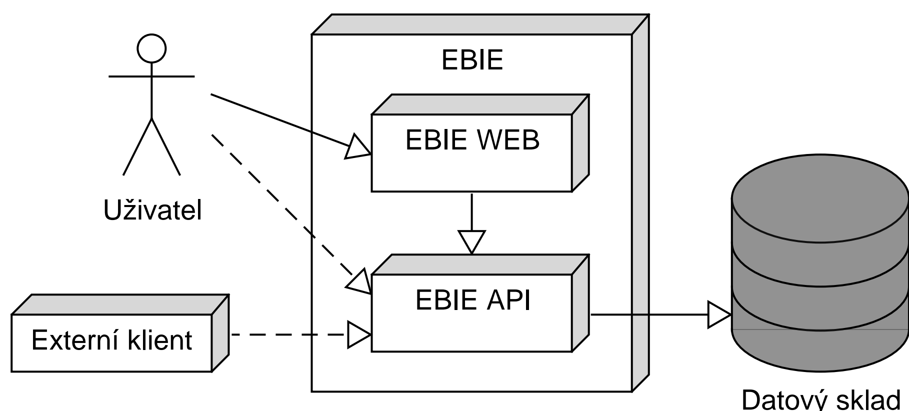
Součástí webového portálu EBIE je komponenta pro správu obsahu. Jedná se o oddělený repositář. Tato komponenta generuje ze souborů ve formátu Asciidoc HTML dokumenty. Je určen pro snadnější tvorbu dokumentace bez nutnosti znát webové technologie nebo Ruby. Za jeho implementací stojí David Hajčiar, který jej vytvořil v rámci své bakalářské práce [5].

V kontextu mé práce není nutné znát detailní fungování správy obsahu, protože „pouze“ generuje obsah a nemá na starosti autentizaci nebo autorizaci.

Dokumentace je přístupná všem bez nutnosti vlastnit specifickou roli. Komponenta obsahuje i definice reportů, které čtou data z REST API. To však není problém, protože report zobrazí pouze taková data, která vrátí API. Je nutné, aby REST API vracela správná data.

2.2 EBIE API

Zobrazovaná data portál získává z vedlejší aplikace. Pro její účel ji budu označovat jako EBIE (REST) API. Tato aplikace běžící vedle webového portálu poskytuje REST API pro získávaní dat z datového skladu. Jejím jediným konzumentem je v době psaní práce pouze portál EBIE. V budoucnu by měla být schopna obsloužit i jiné klienty, kteří se náležitě identifikují.



Obrázek 2.1: Ilustrace přístupu k datům v EBIE.

Na obrázku 2.1 je aktuální podoba přístupu k datům. Uživatel reálně může získávat data pouze omezeně v podobě EBIE reportů. Náročnější uživatelé by tak byli nuceni získat data přímo z datového skladu. To však vyžaduje mít

přistup k datovému skladu. Jako kompromis vznikla EBIE API, která by měla umožnit získat data používaná webovým portálem. Aktuálně je API zamčena pouze pro portál EBIE a neumožňuje jiným službám využívat dostupných dat. Přerušované čáry na obrázku 2.1 naznačují, jaký je záměr pro využívání EBIE API. Uživatelé po získání potřebných oprávnění (role) by mohli přistupovat přímo k RESP API a vytvářet si vlastní klienty využívající data z datového skladu.

EBIE API je také vytvořena v programovacím jazyce Ruby, ale s využitím frameworku Sinatra[6].

2.2.1 Autentizace a autorizace

Přístup k API je momentálně omezen striktně na uživatele přihlášené v portálu EBIE. Není tedy podporován jakýkoliv jiný přístup. K autentizaci vyžaduje validní access token. Stejně jako u webové aplikace, po úspěšné autentizaci jsou přístupná všechna data. Cílem této práce je vytvořit autorizační schéma, které omezí přistup k datům na základě rolí a tím otevře možnost otevřít API pro ostatní klientské aplikace.

Autentizace je již v EBIE API vyřešena s použitím OAAS Zuul. Vlastní proces přihlášení zde ale neprobíhá. V aplikaci je kontrolován již existující token, který musí být předán při každé interakci s REST API. Tento token aktuálně vzniká při přihlášení do portálu EBIE. EBIE při komunikaci s REST API předává uživatelův token v každém požadavku.

OAAS Zuul vystavuje endpoint `/oauth/check_token` pro kontrolu platnosti tokenu. Momentálně se porovnává identifikátor klienta pro kterého byl token vydán. Klient je klientská aplikace, která si vyžádala přihlášení uživatele. V tomto případě je to portál EBIE.

```
{
  "exp": 1522427296,
  "user_name": "cernycyr",
  "scope": ["cvut:umapi:read"],
  "authorities": ["ROLE_USER"],
  "aud": ["usermap-api-248763"],
  "client_id": "abcd1234-ab12-cd23-a444-abcde124567b"
}
```

Ukázka kódu 1: Výsledek kontroly tokenu z OAAS Zuul

Na ukázce 1 je odpověď OAAS Zuul na kontrolu tokenu. Momentálně se kontroluje pouze identifikátor klientské aplikace (`client_id`). Musí se shodovat s identifikátorem portálu EBIE. Tato kontrola by měla být nahrazena autorizačním řešením z této práce. Autorizační server by tak byl využit pouze pro kontrolu identity uživatele a platnosti jeho tokenu.

2.2.2 Datové konce (endpointy)

V EBIE API jsou implementovány endpointy pro přístup k datům z datového skladu. Portál EBIE aktuálně nabízí reporty pro výsledky předmětů a počty studií. Datové konce umožňují rozdělit data podle fakult nebo semestrů. Možnost zaměřit se na katedry není dostupná. Nejjemnější rozdělení dat je na předměty (zobrazení dat o konkrétním předmětu). Data neobsahují informace o konkrétních studentech nebo učitelích.

Protože EBIE API je primárně určena pro spolupráci s portálem, má pro tento účel navržené endpointy. Pro každý typ reportu je vytvořena skupina endpointů na míru. Každá skupina používá vlastní databázové schéma. Aktuálně jsou definovány endpointy pro následující okruhy dat:

- Počty studií
- Výsledky předmětů
- Vyhodnocení ankety dle učitele

Vyhodnocení ankety bylo vytvořeno poslední a nemá v portálu implementováno report.

Kapitola 3

Technologie

Před samotnou analýzou a návrhem je potřeba se seznámit s použitými technologiemi. Nejde pouze o znalost jmen použitých nástrojů. Architektura a některé detaily použitých frameworků budou podstatné pro návrh řešení. Protože aplikace (portál a API) jsou již vytvořeny, je potřeba se podřídit již použitým nástrojům a nedělat radikální změny (pokud nejsou opravdu nutné) v jejich struktuře. Tato kapitola je určena pro ty, kteří nutně neznají použité technologie a potřebují je lehce přiblížit. Pro podrobnější informace je u každé technologie uveden zdroj.

3.1 Programovací jazyk Ruby

Společným jazykem celé práce je Ruby [7]. Jedná se o dynamický objektově orientovaný skriptovací jazyk. Referenční interpret je vytvořen v jazyce C.

Ruby je inspirováno jazyky Perl, Smalltalk, Eifell, Ada a Lisp [8]. Naučit se úplné základy Ruby není obtížné pro lidi, kteří již znají základní konstrukce v programování. Přechod od jazyků jako C, C++, Java nebo Python je tak opravdu snadný. Pro plné pochopení a schopnost v jazyce efektivně pracovat je však potřeba praxe.

3.1.1 Gemy

V Ruby komunitě vnikají různá rozšíření a knihovny. Těmto knihovnám se v Ruby říká „gemy“. V jiných jazycích se takovým knihovnám říká například „pluginy“.

Gemy řeší obvyklé problémy a vývojářům tak šetří práci. V mnoha případech stačí nalézt již existující řešení například v podobě frameworku a dopsat specifickou funkcionalitu dané domény. Tento přístup není specifický pouze pro Ruby. Je však podstatnou součástí ekosystému a dovoluje se soustředit na řešení doménových detailů. Určité gemy (nebo skupina gemů) se staly víceméně

standardem pro řešení některých oblastí. Příkladem budiž vývoj webových aplikací, kde dominuje framework Ruby on Rails.

3.1.2 Syntax a konvence

Pro lepší pochopení ukázek kódu v práci zde budou vysvětleny některé používané Ruby konvence a syntax.

V Ruby je všechno objekt. A to včetně zdánlivě primitivních typů jako je číslo, řetězec nebo nil. Není běžné, aby se v Ruby kontroloval typ objektu (například, zda se jedná o instanci konkrétní třídy). Pokud chceme zkontrolovat, že objekt podporuje nějakou operaci, je možné se zeptat zda na ni umí odpovědět (každý objekt má metodu `respond_to?`). Pokud tedy provádíme operaci nad polem, nemusí to nutně být instance třídy `Array`, ale objekt, který implementuje stejné rozhraní.

```
def add(first, second)
  first + second
end

def say_hello
  puts "Hello World"
end

say_hello
# => "Hello World"
add 5, 3
# => 8
add "hello", " world"
# => "hello world"
```

Ukázka kódu 2: Ukázky syntaxe Ruby

Ruby nevyžaduje psát středník za příkazy na konci řádky. K oddělování různých příkazů se používá odřádkování. Pokud však z nějakého důvodu je nutné mít více příkazů na jedné řádce, je možné středník použít. Odsazování není vynucované (na rozdíl od Pythonu).

V mnoha případech není nutné psát kolem argumentů metod závorky. Pro řetězení volání to však nutné je. V případě vnořování metod je to doporučeno, protože interpret nemusí vyhodnotit výraz podle představ programátora. Závorky se také nemusí psát pokud metoda nemá žádný parametr.

Definice těla metody není ve složených závorkách. Tělo metody je započato deklarací signatury a ukončeno klíčovým slovem `end`.

Všechny Ruby metody vrací výsledek vyhodnocení posledního příkazu. Není tedy nutné zakončovat metodu klíčovým slovem `return`.

Je běžné, aby jméno metody obsahovalo vykřičník (`update_user!`) nebo otazník `user_exists?`). Metoda končící vykřičníkem značí, že je „potenciálně nebezpečná“. To může znamenat, že mění obsah předaných argumentů nebo mění obsah vlastního objektu. Metody s otazníkem naznačují otázku a zpravidla vrací `true` nebo `false`.

Symbol (`:symbol`) je specialita Ruby. Značí se dvojtečkou před jménem. Jedná o objekt nesoucí pouze svou identitu. Není to proměnná. Nenese žádnou jinou hodnotu. Symbol ale může být obsahem proměnné. Většinou nahrazuje funkci řetězce jako slovní identifikátor nebo příznak. Symbol má vždy stejnou identitu (každý objekt má identitu) v rámci běhu programu. Proto bývá používán místo řetězce jako slovní identifikátor. Řetězce, které byly jindy vytvořeny, nejsou identické z hlediska objektové identity i když mají stejný obsah. Na ukázce 3 je ilustrován rozdíl mezi symbolem a řetězcem.

```ruby
:foo.object_id == :foo.object_id
# => true
"foo".object_id == "foo".object_id
# => false
```

Ukázka kódu 3: Rozdíl mezi symbolem a řetězcem v Ruby

3.2 Rack

Jeden z velmi podstatných Ruby gemů pro vývoj webových aplikací je Rack [9]. Rack je malý a modulární interface mezi webovou aplikací (například Ruby on Rails) a serverem (například Puma). Umožňuje vývoj webových aplikaci, bez nutnosti řešit specifickou komunikaci s webovým serverem. Velké množství webových frameworků je postaveno nad Rackem. To znamená, že jsou nadstavbou Racku a využívají jeho interface pro komunikaci. Většina aplikačních serverů pro Ruby je schopna spouštět Rack aplikace.

Základní myšlenka je jednoduchá. Uživatel pošle HTTP požadavek, ten se postupně zpracuje a vrátí se odpověď. Požadavek je množina různých atributů, které se říká „prostředí“(enviroment). Odpověď obsahuje HTTP status kód, hlavičky a tělo. Rack aplikace je objekt, který má definovanou metodu `call` a přijímá hash (podobné asociativnímu poli v jiných jazycích) s prostředím v parametru. Metoda musí vracet pole `[status, header, body]`. Hash s prostředím obsahuje základní informace o požadavku.

- HTTP metodu (GET, POST apod.)
- URL k požadovanému zdroji
- informace o serveru
- přidané informace z middleware a Racku.

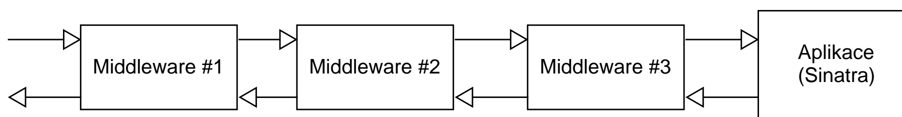
```ruby
class HelloWorld
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hello world!"]]
  end
end
```

Ukázka kódu 4: Primitivní Rack aplikace [10]

Běžně je Rack využíván jako prostředník mezi webovým serverem a aplikací, která dodržuje základní konvence Racku. Proč je tedy vhodné znát Rack, když ho v pozadí bez vědomí vývojáře používá velké množství frameworků jako je například Ruby on Rails? Pro jednoduchý vývoj skutečně není nutné znát Rack a jeho vlastnosti, ale v případě potřeby rozšíření funkcionality daných aplikací zde vstupuje další pojem: middleware.

3.2.1 Middleware

Middleware je implementací principu roury (pipeline pattern). V některých materiálech je možné najít označení pro middleware jako filtr. Rack middleware, jak již název napovídá, funguje jako roura nebo sada filtrů poskládaných za sebou. Přicházející požadavek je tak poslán do této roury a postupně prochází přes vrstvy až k samotné aplikaci. Aplikace požadavek zpracuje, vygeneruje odpověď a pošle zpátky rourou k webovému serveru. Middleware je možné využít například k logování, autentizaci, směrování požadavků, správu sezení nebo zpracování chybových stavů z aplikace. Na obrázku 3.1 je znázorněna cesta požadavku přes middleware až k aplikaci. Na pořadí middlewarů záleží. Poslední middleware, který zpracovává požadavek před aplikací je první, ke kterému se dostane odpověď od aplikace.



Obrázek 3.1: Ilustrace zpracování požadavku přes Rack middleware

Během průchodu může být požadavek upravován a do prostředí je možné přidávat nové informace. Middleware může dokonce požadavek předčasně zastavit a poslat zpátky odpověď (například cache nebo autentizační middleware). Tato struktura usnadňuje a zpřehledňuje vývoj aplikací. Vlastní aplikace se tak mohou soustředit na vlastní funkcionalitu.

```
class SimpleMiddleware
  def initialize(app)
    @app = app
  end

  def call(env)
    # práce s požadavkem před zavoláním dalšího middlewaru
    result = @app.call(env)
    # práce s výsledkem volaného middlewaru
  end
end
```

Ukázka kódu 5: Příklad velmi jednoduchého middlewaru

Rack middleware je opět jednoduchý objekt. Stejně jako předtím, musí mít definovanou metodu `call`. Navíc však vyžaduje schopnost inicializace s jedním argumentem. Rack aplikace tak může sloužit i jako middleware, pokud je možné ji vytvořit s jedním parametrem. V tomto parametru je odkaz na další middleware nebo aplikaci v pořadí. Pro předání požadavku dál v rouře tak stačí zavolat `call` nad předaným objektem. Při návratu odpovědi k uživateli je možno tuto odpověď odchytit, upravit a předat dál.

Při porovnání ukázek 4 a 5 můžeme vidět rozdíl mezi Rack aplikací a middleware. Jde tedy pouze o to, zda vytváříme funkcionalitu filtru. Pokud ne, nemusíme definovat konstruktor s jedním argumentem a nemusíme volat další middleware v pořadí (protože už žádný další není).

Díky teto velmi jednoduché architektuře Racku je možné přidávat obalující logiku kolem vlastní aplikace, která se může soustředit na vlastní funkcionalitu. To platí i při využití velkých a složitých frameworků, protože v pozadí jsou pouze aplikací, která se volá na konci roury.

3.3 Sinatra

Sinatra [6] je jednoduchý framework pro vývoj webových aplikací. Podstatná funkcionalita je definována v jediném souboru. Není to tedy příliš velký framework (na rozdíl od Ruby on Rails). Účel je poskytnout základní sadu nástrojů pro vývoj webové aplikace. Sinatra není příliš vhodná na rozsáhlé portály. Pro vývoj REST API je vhodná, protože neobsahuje potenciálně zbytečnou funkcionalitu a vývojář si doplní gemy se specifickou funkcionalitou podle potřeby.

3.3.1 Sinatra DSL

Sinatra definuje DSL (domain specific language), pomocí kterého lze vytvářet definice cest a jejich obsluhu. Na ukázce 6 je ilustrace Sinatra DSL. První klí-

čové slovo definuje HTTP metodu, kterou má cesta přijímat a poté následuje relativní cesta. Pokud by aplikace poslouchala na adrese `localhost`, tak druhá metoda v ukázce 6 by byla zavolána při požadavku na `localhost/foo`. Metody mohou vracet libovolné objekty, které je možné representovat řetězcem znaků (string). To znamená, že vracený objekt musí mít definovanou metodu `to_s`. V Ruby tuto metodu mají implicitně definované všechny podtřídy třídy `Object`.

```
get '/' do
  # app root
end

get '/foo' do
  # get foo
end

post '/foo' do
  # create foo
end
```

Ukázka kódu 6: Sinatra DSL pro definici cest

Samozřejmě Sinatra DSL umožňuje detailnější definici cest pomocí regulárních výrazů. V případě, že množství cest začne narůstat přestává definice na jednom místě stačit. Jako řešení slouží možnost rozdělit tyto definice do jmenných prostorů (namespaces). To dovoluje rozdělit kód na logické části.

Je běžné přijímat parametry. Například od cesty `/fakulta/10` bychom čekali, že dostaneme odpověď obsahující informace o fakultě s identifikátorem 10. V DSL je možné takovou cestu napsat jako `/fakulta/:id`. Dvojtečka v definici cesty značí, že zde má Sinatra očekávat parametr. V těle metody poté bude k dispozici obsah parametru `id` v `params["id"]`.

Další možnost předání parametru je poslání požadavku `/fakulta?id=10`. Text za otazníkem se nazývá „query string parametr“. Od takového dotazu bychom očekávali výsledek se všemi fakultami, které mají identifikátor rovný 10. Jde o sémantický detail, kde query parametr se spíše využívá k filtrování výsledků.

Sinatra mezi těmito druhy parametrů striktně nerozlišuje a oba zpřístupní v `params["id"]`. Pokud je parametr v cestě a query pojmenován stejně, má prioritu parametr v cestě.

3.4 Pliny

EBIE API byla vytvořena pomocí gemu Pliny [11]. Jedná se o šablonu, která vygeneruje základní strukturu REST API v Sinatře. Dále přidává předdefino-

vané výjimky pro HTTP status kódy a upravuje směrování pro podporu verzí API. Tento gem se snaží více zjednodušit vývoj Sinatra aplikací a přiblížit se strukturou například k Ruby on Rails. Důvod výběru toho gemu neznám. Předpokládám, že byl zvolen pro předdefinovanou strukturu aplikace a původní autor EBIE API tak mohl začít rychle pracovat na definici jednotlivých cest. Protože byl tento gem použit spíše jako šablona, nemá příliš vliv na další analýzu a návrh. Veškeré přidání nebo změna funkcionality tak musí dodržet principy Racku a Sinatry. Pliny pouze ovlivní, kam se v rámci struktury nové zdrojové kódy uloží nebo importují.

Kapitola **4**

Analýza a návrh technických rolí

V této kapitole popíši strukturu technických rolí používaných na ČVUT. Nebudu se příliš zabývat technickými detaily rolí. Tím se zabýval Jan Matys ve své diplomové práci [12]. Jeho analýzu využiji při potvrzování některých vlastností atributů role. Budu se tak hlavně zabývat podobou role tak, jak ji lze vidět například na ČVUT Usermap [13].

4.1 Získání dat

K analýze jsem využil vlastní seznam rolí z Usermap. Taková množina rolí je ale poměrně malá a tak jsem využil Usermap API [14]. Ta umožňuje vypsání všech rolí, které zná. Bohužel vrátila pouze některé role spojené s fakultou informačních technologií. Předpokládám, že se jedná o filtrování výsledků podle vlastněných rolí a takový výsledek je tedy v pořádku. I přes omezený výsledek mám již dostatečnou množinu rolí k tomu, abych mohl vyslovit některé předpoklady na podobu technické role.

4.2 Struktura role

Podívejme se výpis pár rolí pro ilustraci.

```
T-ZP-18000-VEDOUCI-PRACE
T-ZP-18000-OPONENT-PRACE
T-FITWEB-18104-SPRAVCE-OBSAHU
T-FITWEB-18924-SPRAVCE-OBSAHU
```

Role obsahuje následující informace:

Typ role Role začínající na písmeno „T“ jsou technického typu. V případě byznys role začínají písmenem „B“.

Přidružený systém Jméno, nebo zkratka systému, pro který je role určena. V ukázce tak máme role pro systém závěrečných prací (`ZP`) a správu webu fakulty (`FITWEB`).

Organizační jednotka Identifikátor organizační jednotky značí k jaké jednotce role patří. Většinou se shoduje s jednotkou, do které vlastník dané role patří. V příkladu role s `18000` znamená, že vlastník takové role je z fakulty informačních technologií. V případě role s `18104` je upřesněna náležitost ke katedře počítačových systémů.

Jméno role Jméno by mělo stručně vystihovat účel přidělené role.

Technická role má tedy tvar

```
T-{přidružený systém}-{organizační jednotka}-{jméno role}
```

Některé části role mohou mít podmínky na hodnoty nebo tvar. Identifikace přidruženého systému je unikátní a delší než 1 znak. V dostupných rolích délka toho identifikátoru nepřesáhla 6 znaků. Jméno role je obvykle česky. Jinak na jméno role není kladena žádná speciální podmínka. Dle diplomové práce Jana Matyse [12] má celá role splňovat regulární výraz `^[a-zA-Z0-9_\-()/]+\$`. Z toho usuzuji, že jméno role by mohlo být prázdné. Takové (ne)omezení je zvláštní. Je vhodné, aby jméno existovalo pro určení účelu role. Ze získaných rolí jsem dále vysledoval, že pokud má poslední část role víceslovné pojmenování (`VEDOUCI-PRACE`), je místo mezery použita pomlčka.

Identifikátor organizační jednotky je pětimístné číslo. Organizační jednotkou je jakákoliv součást ČVUT včetně samotného ČVUT. Tomu je přiřazen speciální identifikátor `00000`. První 2 cifry identifikátoru jsou pro určení fakult, ústavů a dalších jednotek přímo podřízených ČVUT. V této práci nás zajímají hlavně fakulty a proto uvidíme převážně označení od 11000 do 18000. Následující 3 cifry jsou pro identifikaci katedry nebo pracoviště. Formát pro identifikátor fakult a katedrami je tedy zjednodušeně `FFKKK` kde `F` značí fakultu a `K` katedru.

4.3 Podoba nové role

Při sestavovaní role je třeba odpovědět na následující otázky:

- Jak se jmenuje systém nebo aplikace, pro který role vzniká?
- Na jakou organizační jednotku se role bude vázat?
- Jaké typy uživatelů aplikace má?

Odpovědi na tyto otázky postupně urči podobu role.

4.3.1 Jméno aplikace

Jméno aplikace je víceméně jasné. Práce se zabývá vytvořením autorizace pro EBIE. EBIE má však 2 komponenty. Webový portál a API. Jsou tedy 2 varianty.

- Jednotná role `EBIE`
- Oddělené role `EBIEWEB` a `EBIEAPI`

Cílem této práce je řízení přístupu k datům z datového skladu. Tato data jsou dostupná pouze prostřednictvím EBIE API. Webový portál pouze zobrazuje získaná data. EBIE API by měla být schopna obsloužit i jiné klienty, než pouze webový portál. Je tedy zřejmé, že autorizace bude řešena v EBIE API. Webový portál si tak vystačí s autentizací uživatelů. Pokud by se naskytla potřeba specifických autorizací přímo v portálu, je možné delegovat veškerou autorizaci na API.

Z toho vyplývá, že vhodnou variantou na jméno aplikace v roli je `EBIE`. Vybrané jméno v roli vyjadřuje oprávnění k přístupu k datům v celém projektu EBIE. Nezáleží, zda uživatel využívá portál, nebo přistupuje přímo k REST API.

4.3.2 Organizační jednotka

Identifikátor organizační jednotky se bude měnit podle osoby, které bude role přidělena. Většinou by osoba měla dostat roli s vazbou na organizační jednotku, na které působí. Protože je EBIE určena spíše pro vedoucí pozice, nemá zatím smysl uvažovat o běžných zaměstnancích (například vyučující). Běžný uživatel EBIE portálu tak bude vedoucí katedry, děkan nebo jejich zástupci. Není to ale nutná podmínka. Roli smí získat kdokoliv podle úsudku přidělující osoby.

Z role máme k dispozici identifikaci až ke katedře. Jsme tak schopni rozlišit, zda role patří ke katedře softwarového inženýrství, nebo aplikované matematiky. Po konzultaci s vedoucím práce jsme došli k závěru, že využijeme plné schopnosti identifikátoru organizační jednotky v roli a bude se kontrolovat příslušnost až ke katedře.

V návrhu nové role budou povoleny identifikátory fakult a kateder. Přístup k datům bude omezen podle náležitosti k příslušné katedře (např. 18102). V případě, že uživatel bude mít roli s identifikátorem fakulty (např. 18000), bude moci přistupovat k datům všech podřízených kateder. Pro neomezený přístup do EBIE API je možné využit identifikátor 00000, který značí přímou příslušnost k ČVUT. Tento identifikátor je validní Usermap identifikátor organizační jednotky a dodržuje tedy pravidla technických rolí.

4.3.3 Jméno role

Poslední část technické role je jméno nebo označení role. Mělo by srozumitelně vyjadřovat pozici uživatele k systému. V uvedených příkladech to je například `VEDOUCI-PRACE` nebo `SPRAVCE-OBSAHU`. Je zřejmé, v jaké jsou pozici. Při návrhu si tedy musíme položit otázku: Z jakých pozic mohou uživatelé přistupovat k datům v EBIE?

EBIE API poskytuje data, která uživatel čte. Je tedy jisté, že nejzákladnější oprávnění je možnost data číst. Další potenciální variantou je, že by REST API povolila data přidávat, měnit nebo mazat (ekvivalenty HTTP POST, PUT a DELETE). Tato možnost je však zcela nepřípustná, protože dostupná data jsou poskytována z datového skladu, který data poskytuje pouze ke čtení.

Otázkou by mohlo být, jak lidé spravující dokumentaci EBIE přidávají obsah? Dostanou přístup do repositáře pro správu obsahu EBIE, který jsem zmínil v sekci 2.1.1. Přidávaní reportů je také záležitost získání přístupu k repositářům konkrétních komponent.

EBIE API je skutečně pouhým prostředníkem mezi databází a uživatelem. Jediná možná operace je čtení. Proto každý, kdo využije služby EBIE API je pouze jejím uživatelem. Pro účel ČVUT IdM je to tedy `UZIVATEL`

4.3.4 Výsledná role

Pokud poskládáme závěry z předchozích sekcí do jednoho uceleného výsledku, vyjde nám následující návrh na podobu nové role pro použití v EBIE.

```
T-EBIE-{organizační jednotka}-UZIVATEL
```

Identifikátor organizační jednotky je podstatným nositelem informace pro další zpracování v systému. Je na něj kladena ještě jedna dodatečná podmínka vedle již uvedených podmínek v sekci 4.2. Organizační jednotka musí být fakulta, katedra nebo 00000 (označení kořene organizačních jednotek). V budoucnu by mohl být systém rozšířen i pro další součásti ČVUT, ale v této práci se budu zabývat pouze detaily pro fakulty a katedry.

Role uživatele bude opravňovat číst data přidružené katedry nebo fakulty. Role `T-EBIE-18102-UZIVATEL` tedy umožní čtení dat, která jsou spjata s katedrou softwarového inženýrství na fakultě informačních technologií. V případě, že by uživatel měl roli vázanou na fakultu (`T-EBIE-18000-UZIVATEL`), mohl by číst data spjata s fakultou a tedy i všech kateder dané fakulty.

Kapitola 5

Rešerše možných řešení

Před tvorbou vlastního řešení je vhodné se podívat, zda již někdo nevytvořil autentizační a autorizační systém založený na rolích. Protože komunita jazyka Ruby je poměrně velká, existuje široká nabídka gemů. V ideálním případě nalezneme vhodný gem, který bude stačit nastavit a nasadit v aplikaci. Pokud by takový gem nebyl nalezen, stále mohou existující řešení sloužit jako inspirace při vzniku vlastního řešení

5.1 Požadavky

Nejprve je potřeba si zadefinovat požadavky na funkcionalitu gemu nebo skupiny gemů. Gem by měl být open-source a dostatečně udržován komunitou. Budu preferovat řešení, která mají dokumentaci a dostatečně velkou komunitu. V případě, že gem bude spíše inspirací se čtení zdrojového kódu nevyhnu. Je třeba, aby řešení bylo schopné pracovat s formátem technické role ČVUT. To znamená velké množství potenciálních rolí. Role by neměly být nijak ukládány v databázi, protože jich je velké množství a nesou svou podstatnou informaci ve jméně.

Při výběru je relevantní, jak je řešení rozsáhlé. Komplexní gemy je složité zkoumat a budou obsahovat velké množství funkcionalit, které nejsou potřeba. Je větší pravděpodobnost, že se složitěji nastavují. Dalším důležitým kritériem je poslední verze. Nebudu uvažovat gemy které jsou několik let opuštěné. Je žádoucí, aby byl nástroj kompatibilní s novými verzemi Ruby a ostatními použitými gemy. Jako rozhodující kritérium je možnost implementace ČVUT technických roli. Řešení může obsahovat jakkoliv chytré rozhodování, ale pokud nepůjde snadno implementovat požadovaný systém rolí, bude výhodnější vytvořit vlastní řešení.

Pojmy autorizace a autentizace bývají často zaměňovány a používány jako synonyma. Ještě před samotnou rešerši tedy upřesním, co se těmito pojmy chápe.

5.1.1 Autentizace

Autentizace je proces ověření identity. Při přihlašování do systému proběhne identifikace na základe zadaných údajů. Pokud komunikujete s REST API, která vyžaduje například OAuth2 token, jste identifikováni podle předaného tokenu. Systém může ověřit, že uživatel je skutečně osobou, za kterou se vydává.

V EBIE API je autentizace již vyřešena. Používá se vlastní řešení využívající OAuth2 autorizační server Zuul, který je nasazen na fakultě informačních technologii. Přestože je server autorizační, je použit pro autentizaci uživatele. Autorizace probíhá ze strany uživatele (o tom později). Není důvod přecházet na jiné řešení a proto nemá smysl hledat gemy, které řeší striktně autentizace. Proto nebudou dále rozebírány následující gemy:

- Devise
- Doorkeeper
- Omniauth
- AuthLogic
- Warden

Jejich společnou funkcionalitou je správa a přihlašování uživatelů. Některé podporují více metod přihlašování nebo spravují uživatelovo sezení (session).

5.1.2 Autorizace

Autorizace je proces ověření oprávnění přístupu k prostředkům. Systém během autorizace rozhodne, zda uživatel smí provést operaci (například čtení nebo modifikace) nad požadovaným zdrojem. V případě EBIE se bude autorizovat jediná operace: čtení.

Většinou autorizaci předchází autentizace, abychom mohli určit identitu a oprávnění uživatele. Je však možné autorizovat i anonymní uživatele bez nutnosti znát jejich identitu (přístup k veřejné části portálu nebo dat).

V předchozí kapitole byl zmíněn autorizační server Zuul. V případě EBIE je využíván pouze pro ověření identity uživatele. Uživatel je po přihlášení na fakultním serveru dotázán, zda povoluje aplikaci EBIE přistupovat jeho jménem k datům na Usermap API. Zde probíhá povolení přístupu uživatelem, nikoliv naší aplikací. Aplikace dostane povolení (autorizaci) od uživatele. V budoucnu by mohla být do seznamu API pro autorizaci přidána i EBIE API.

Hledáme tedy již existující řešení pro přistup k datům. Pojďme se tedy podívat, co nabízejí používaná open-source řešení. Jedná se o omezený výběr používaných a aktuálních řešení. Pro hledání dostupných řešení jsem využil

vyhledávač gemů The Ruby Toolbox [15]. Ve výsledku vyhledávaní jsem nevybral pouze několik nejvýše zobrazených. Vynechal jsem dlouho (roky) neaktualizované nebo na první pohled nevyhovující (špatně zařazené autentizační) gemy. Většina ukázek kódu pochází z dokumentací patřičných gemů.

5.2 Pundit

Pundit [16] poskytuje sadu pomocných metod pro autorizaci. V Ruby třídě poté stačí vložit pomocné metody pomocí `include Pundit`. Vytváření autorizačních pravidel je záležitostí definice tříd s definovanou strukturou.

```
class PostPolicy
  attr_reader :user, :post

  def initialize(user, post)
    @user = user
    @post = post
  end

  def update?
    user.admin? or not post.published?
  end
end
```

Ukázka kódu 7: Pundit definice pravidel

Pundit předpokládá několik skutečností [16]

- Jméno třídy s pravidly je stejné jako třída modelu s koncovkou „Policy“.
- Existuje metoda `current_user`, která vrací přihlášeného uživatele. Výsledek této metody je předán konstruktoru jako `user`.
- Druhý argument je jakýkoliv ruby objekt, nad kterým se provádí autorizace.
- Metody definované ve třídě odpovídají operacím nad objekty. Toto jméno se namapuje na volání v controlleru (třída obsluhující jednotlivé akce).

Z těchto předpokladů je možné vypozorovat, že Pundit pravděpodobně vznikl nad nějakým frameworkem. Z dokumentace je patrné, že autoři tento gem vytvářeli nad Ruby on Rails. V Ruby on Rails je controller třída, která řídí chování jednotlivých operací ve webové aplikaci. Metody v této třídě se mapují na klasické CRUD (create, read, update, delete) operace. Toto je potenciální komplikace, protože bude nutné se přizpůsobit chování gemu, nebo jej modifikovat.

Použití autorizace je následně poměrně jednoduché. Na ukázce 8 je vidět, že se pouze zavolá metoda `authorize` a předá se ji objekt nad kterým se má rozhodnout. V případě zamítnutí přístupu je vyhozena výjimka.

```
def update
  @post = Post.find(params[:id])
  authorize @post
  if @post.update(post_params)
    redirect_to @post
  else
    render :edit
  end
end
```

Ukázka kódu 8: Pundit autorizace

Na ukázce 8 je možné si všimnout, že se neupřesňuje pravidlo, které se má uplatnit (update). Pundit je schopný ze jména třídy a metody, ze které je `authorize` zavoláno, určit, kterou metodu ve třídě s pravidly zavolat. Je však možné použít `authorize(@post, :update?)`. V druhém parametru se předává jméno metody, která se má zavolat. Není tedy nutné striktně pojmenovávat metody.

Použiti Pundit tak vyžaduje reprezentovat přihlášeného uživatele objektem. Data, ke kterým se přistupuje je objekt modelu. Protože Pundit staví na Ruby on Rails, je takový model většinou uložený v DB a jedná se spíše o reprezentaci nějaké konkrétní věci (například konkrétní příspěvek diskusního fóra). Nejedná se tak zpravidla o množinu abstraktních dat.

5.3 CanCanCan

CanCanCan [17] je pokračováním vývoje gemu CanCan. Řeší autorizaci na základě „způsobilostí“ (abilities). Funguje velmi podobně jako Pundit. Také poskytuje sadu pomocných metod a předpokládá určitou strukturu aplikace. V popisu rovnou říká, že je určený pro Ruby on Rails. Neznamená to, že by nešel použít i jinde, ale je větší riziko komplikací při integraci do jiného frameworku (v našem připadě Sinatra). Nevyhneme se tak modifikacím samotného gemu, nebo již existující aplikace.

CanCanCan na povrchu funguje velmi podobně jako Pundit. Pravidla pro autorizaci jsou třídou. V konstruktoru se pak definuje celá autorizační politika pro uživatele. Tento způsob je méně přehledný, než u Pundit.

Metoda `can` se používá k definici oprávnění. První argument je prováděná operace a druhý je třída (nikoliv objekt), nad kterou se operace provádí. Je možné využít speciální příznaky, které reprezentují všechny operace (`:manage`)) a všechny třídy (`:all`). CanCanCan umožňuje přímo v definici

```ruby
class Ability
  include CanCan::Ability

  def initialize(user)
    can :read, :all # všichni smí číst
    if user.present?
      # pouze vlastník smí editovat
      can :manage, Post, user_id: user.id
      if user.admin?
        can :manage, :all # administrátor smí vše
      end
    end
  end
end
```

Ukázka kódu 9: CanCanCan definice způsobilostí uživatele

pravidla dodatečné podmínky. Tyto podmínky jsou třetím nepovinným parametrem. Například `can(:read, Project, user_id: user.id)` by povolilo číst objekty třídy `Project` pouze uživateli, který má stejný identifikátor, jako ten uložený v atributu objektu projektu. To se dá interpretovat jako povolení čteni pouze majiteli projektu.

CanCanCan poté umožňuje se ptát na oprávnění pomoci metody `can?`, která vrátí booleovskou hodnotu, zda je uživatel oprávněn provádět operaci. Tato pomocná metoda se hodí například při rozhodování, zda zobrazit část webového obsahu.

Stejně jako Pundit nabízí metodu `authorize!`. Ta při nepovoleném přístupu vyhodí výjimku.

```ruby
def show
  @article = Article.find(params[:id])
  authorize! :read, @article
end
```

Ukázka kódu 10: CanCanCan volání autorizace

CanCanCan a Pundit jsou podobné gemy. CanCanCan se pokouší být pohodlnější variantou Pundit pro běžné webové aplikace. Liší se v implementačních detailech, ale podstata je stejná. Vytvořit třídu pro pravidla a při kontrole předávat uživatele a požadovaný objekt. Proto je stejně jako u Pundit potřeba najít možnost, jak representovat uživatele a data objektem, který by se těmto nástrojům předal.

5.4 Consul

Dalším existujícím řešením je Consul [18] vytvořený společností Makandra. Jedná se o další open-source gem určený pro Ruby on Rails. Jako u předchozích řešení to nemusí nutně znamenat problém, je však potřeba dát si pozor na specifické Ruby on Rails vlastnosti, které jsou při fungování gemu předpokládány.

Definice pravidel je ve třídě `Power`. V této třídě se definují pravidla jako [18]

- rozsah záznamů, které se vrátí
- zda uživatel smí vidět konkrétní obrazovku,
- list hodnot, které smí uživatel přiřadit určitému atributu.

Třetí bod není pro nás relevantní, protože data v EBIE API nelze měnit. V ukázce 11 je možná definice pravidel.

```ruby
class Power
  include Consul::Power

  def initialize(user)
    @user = user
  end

  power :users do
    User if @user.admin?
  end

  power :notes do
    Note.by_author(@user)
  end

  power :dashboard do
    true
  end
end
```

Ukázka kódu 11: Consul definice pravidel

Rozdíl od předchozích řešení je ve způsobu řízení přístupu. Pundit a CanCanCan rozhodují na základě předaných objektů pro uživatele a zdroje. Neovlivňují data, pouze „schválí operaci“. Pokud bychom po schválení vykonali něco jiného, není to jejich problém. Consul k autorizaci přistupuje z druhé

strany. Pokud chceme data, požádáme o ně Consul. Consul vrátí pouze relevantní množinu dat podle oprávnění uživatele. Na ukázce 12 jsou různé možnosti volání jednoho pravidla. Každá z možností se chová podle běžných konvencí v Ruby. Otazník znamená otázku a vrací `true` pokud by metoda `notes` vrátila data. Varianta s vykřičníkem způsobí vyhození výjimky, pokud nastane problém (neautorizovaný přístup), nebo je vrácena prázdná množina.

```
power = Power.new(user)
power.notes # => množina dat nebo nil
power.notes? # => true / false
power.notes! # => výjimka, pokud nemá oprávnění
```

Ukázka kódu 12: Consul použití

Consul tak slouží jako vrstva nad ORM (Object-relational mapping). V případě Ruby on Rails je implicitně Active Record. Consul vrací relaci Active Record, kterou developer dále použije. Využití ActiveRecord je problémové, protože EBIE API nepoužívá žádné ORM. Databáze je vzdálená a aplikace nezná její schéma. Není tak možné jednoduše namapovat schéma databáze do tříd.

Naštěstí není nutné vracet Active Record relace. Consul dovoluje vracet jakýkoliv objekt. EBIE API tak může získat data z databáze a vrátit je jako pole nebo hash.

V ukázce 11 je možné si všimnout definice `power :dashboard`, která vrací pouhé `true`. Consul metody nemusí sloužit k získaní dat, ale k dotazování pouze na práva. Vracení relevantních dat je ale hlavní rozdíl od předchozích gemů.

5.5 Six

Six je velmi malé autorizační řešení. Jako jediný z užšího výběru není specificky vytvořen pro Ruby on Rails. Protože není určen pro žádný specifický framework, nemá ani příliš předpokladů na existující metody. Objekt s informacemi o uživateli se tak například předává manuálně. Nenabízí toho tolik, jako předchozí řešení. Zaměřuje se pouze na definici a ověřování pravidel. Nesnaží se vynucovat pravidla vyhazováním výjimek nebo neomezuje vracenou množinu dat. To je přenecháno na vývojáři aplikace.

Definice pravidel je velmi podobná předchozím gemům. Na jménu třídy nezáleží. Jediný požadavek je definice statické metody `allowed`. Je tak možné přidat definici i do již existující třídy. Tato metoda musí vracet pole povolených aktivit. Povolená aktivita je jakýkoliv symbol, který si vývojář definuje.

Takto definovaná třída se pak přidá do instance třídy `Six`. Tato instance slouží k vyhodnocování pravidel. K zavolání dotazu, zda předávaný uživatel

```
class BookRules
  def self.allowed(author, book)
    rules = []

    return rules unless book.instance_of?(Book)

    rules << :read_book if book.published?
    rules << :edit_book if book.author?(author)

    if book.author?(author) && book.is_approved?
      rules << :publish_book
    end

    rules
  end
end
```

Ukázka kódu 13: Six definice pravidel

smí vykonat činnost nad předaným objektem, je metoda `allowed?`. Žádné vyhazování výjimek. Vrátí se pouze ano, nebo ne.

```
abilities = Six.new
abilities << BookRules
abilities.allowed?(@user, :read_book, @book)
```

Ukázka kódu 14: Six použití

Tento gem se řídí pravidlem „v jednoduchosti je síla". Výhoda Six je skutečně v poměrně základní a jednoduché implementaci autorizací pro Ruby. Nepředpokládá ani nevyžaduje žádný framework.

V kontextu s EBIE API je opět potenciální problém předávat objekt, nad kterým se rozhoduje o oprávnění, protože nemáme žádný model. EBIE API vrací množinu předem neznámých dat, nikoliv namapované instance objektů z databáze.

5.6 Shrnutí

Mohl bych vypsat mnoho dalších gemů. Většina z nich je ale velmi podobná gemům Pundit a CanCanCan. Většinou se snaží napodobit jejich funkcionalitu vlastním způsobem. Gem Six je právě takovým případem. Dále velké množství gemů je vytvořeno pro Ruby on Rails. Již jsem zmiňoval, že to není nepřekonatelný problém. Je to ale práce navíc a potenciální riziko při integraci s aplikací založené na Sinatře. Six je vytvořen pro obyčejné Ruby a bez závislostí. Proto

není potřeba se přizpůsobovat jeho předpokladům. Je však potřeba doplnit logiku kolem něj.

Velmi zajímavý je Consul, který k autorizaci přistupuje jinak. Skutečně autorizuje přístup k datům a ne oprávnění pracovat s objektem modelu. Tato filosofie je blízká požadavkům na EBIE API. Ze zmíněných řešení se jedná o pravděpodobně nejvhodnějšího kandidáta k použití.

Všechna zmíněná řešení mají co nabídnout. Je možné se jim přizpůsobit za cenu práce navíc. K tomu autoři CanCanCan mají poznámku [19]: „Also consider, if you have very unique authorization requirements, the best choice may be to write your own solution instead of trying to shoe-horn an existing plugin.“ Nevím, za jak unikátní se považuje implementace technických rolí ČVUT. Pokud se ale autorizace nad rolemi jako je uživatel, administrátor a editor považují za běžné, tak ČVUT technické role jsou unikátní. Pokud by nebylo vybráno žádné zmíněné řešení a vydal bych se cestou vlastní implementace, jsou tato řešení stále dobrou inspirací.

Kapitola 6

Návrh řešení

V této kapitole vznikne návrh řešení autorizace na základě technických rolí ČVUT. Autorizace bude realizována v EBIE API. V portálu stačí dosavadní autentizace. Jak již bylo zmíněno v sekci 4.3.3, data se pouze čtou. Není potřeba řešit rozdíl mezi oprávněním pro čtení, přidávaní, mazání nebo modifikaci dat. V následujících sekcích budou postupně popsány části autorizačního procesu.

6.1 Základní filosofie

Cílem je zobrazit data pouze těm lidem, kteří získali patřičná oprávnění. V našem případě se jedná o získání technické role. Podoba role je diskutována v sekci 4.3.4. Tato role umožní uživateli používat EBIE API. Jakmile si uživatel vyžádá konkrétní zdroj, je zkontrolováno, zda smí ke zdroji přistupovat. Oprávnění k přístupu se kontroluje porovnáním organizační jednotky v roli a vyžádaným zdrojem.

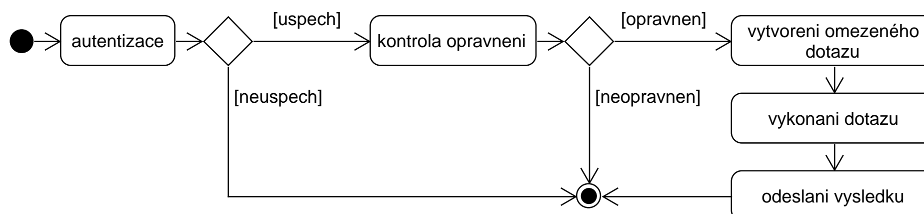
Jako příklad může být dotaz na data fakulty 10 (fakulta informačních technologií). Aplikace porovná roli a požadavek. Následně vrátí výsledek, nebo uživatele odmítne obsloužit. Co když ale bude bude požadavek na všechny předměty v semestru B142? Předpokládejme, že uživatel má roli vázanou na nějakou katedru. Jakou má uživatel dostat odpověď? Je oprávněným uživatelem. Zobrazíme opravdu všechny předměty? V tom případě jsou zobrazeny i předměty, které nepatří k dané katedře. Odmítneme požadavek, protože by odpověď obsahovala informace, ke kterým uživatel nemá mít přistup? Uživatel tak bude nucen upřesnit svůj dotaz pouze na předměty katedry v semestru. To je však nepohodlné a zbytečné. Rozumnější řešení by bylo jednoduše vrátit pouze ta data, ke kterým ho jeho role opravňuje.

Z těchto příkladů by měly být vidět 2 různé způsoby omezení přístupu k datům.

- Kontrola, zda uživatel má oprávnění přistupovat k datům

- Omezení výsledku podle role

Některým by se mohlo zdát, že druhá varianta je schopna pokrýt funkcionalitu té první. Čistě teoreticky je to tak. Pokud by uživatel žádal o data jiné katedry, než ke které má přístup, byla by mu vrácena prázdná množina. Nedostal se k datům, ke kterým neměl a tím byl účel splněn. Není to ale vhodné řešení problému. Uživatel neví, zda udělal chybu nebo nesmí data číst. Je žádoucí informovat uživatele, proč nedostal výsledek, jaký si (asi) představoval. Prázdná množina by měla znamenat, že dotaz byl oprávněný, ale neexistují data vyhovující požadavku. Pokud by dotaz oprávněný nebyl (chtěl výslovně data jiné katedry), uživatel by měl dostat jako odpověď zamítnutí požadavku.



Obrázek 6.1: Základní koncept zpracování požadavku

Na obrázku 6.1 je ilustrace zpracování požadavku. Nejprve je uživatele nutné „poznat“. Následně se kontrolují jeho role. V tomto kroku se nejprve zkontroluje vlastnictví jakékoliv EBIE role. Pokud má alespoň jednu EBIE roli, je jeho požadavek prozkoumán pro možná omezení. Vyhodnotí se, zda požadavek nevyžaduje příslušnost k fakultě nebo katedře. Pokud má uživatel správné role, vytvoří se dotaz na databázi, ve kterém se musí zahrnout všechna omezení vyplývající z rolí uživatele. Výsledná data se nakonec předají uživateli.

6.1.1 HTTP Status kódy

Součástí odpovědi při komunikaci s REST API nejsou pouze data. Odpověď obsahuje status kód, který vyjadřuje, jak operace dopadla. V případě úspěchu jsou data v těle odpovědi. Standard (RFC 2616 [20]) definuje jejich význam. Je tak možné vyjádřit nejen selhání operace, ale i důvod pomocí status kódu. V případě kódů 4xx se jedná o chyby ze strany uživatele. Pro chyby na straně serveru slouží kódy 5xx. V rámci autorizaci nás zajímá, zda byl uživatel vpuštěn, nebo nikoliv. Při úspěchu se běžné vrací kód 200. Při chybě během autentizace a autorizace není vhodné vrátit obecný status kód 400. Vhodnější je vrátit takový kód, který lépe popisuje důvod selhání a případné detaily mohou být dopsány v těle odpovědi. Následující status kódy jsou pro nás relevantní.

400 Bad Request Jedná se o obecnou chybu. Velké množství REST API vrací tento kód a následně upřesní charakter chyby v těle odpovědi. Většinou jde o chyby v předaných parametrech. Podle RFC 2616 je možné vrátit 400, pokud server nerozumí požadavku z důvodu špatné syntaxe.

401 Unathorized Značí problém při autentizaci. Jméno kódu je zavádějící, protože neznačí žádný problém s autorizací. Měl by se používat při chybějící, nebo odmítnuté autentizaci. Pro problém s autorizací je určen 403.

403 Forbidden Server rozumí požadavku, ale odmítá vyhovět. Toto se týká i neoprávněného přístupu, kde uživatel je známý, ale server mu aktivně odmítne data poskytnout, protože nemá oprávnění.

404 Not Found Vyjadřuje, že požadovaný zdroj nebyl nalezen.

422 Unprocessable Entity Tento kód je z rozšíření HTTP WebDAV (RFC 4918 [21]). Znamená, že server rozumí požadavku, ale jsou v něm sémantické chyby. Server ví, co má dělat, ale například kombinace předaných parametrů je nesmyslná a požadavek z toho důvodu nelze možné dokončit.

V autorizačním řešení by tak měli být vhodně použity správné status kódy. Pokud se nepovede identifikovat uživatele, vrátí se 401. Pokud ale uživatel nemá oprávnění pokračovat, mělo by se vrátit 403. Mělo by se omezit použití 400, pokud bude možné využít jiný kód, který chybu vyjadřuje přesněji.

6.2 Zvolení nástroje pro autorizaci

V kapitole 5 bylo popsáno několik již existujících řešení. Většina nabízí široké možnosti. Poskytují relativně srozumitelnou strukturu, která je připravena pojmout i náročnější požadavky velkých domén. Pro potřeby EBIE API stačí schopnost autorizovat jedinou akci (čtení). Na druhou stranu vyžaduje schopnost pracovat s neurčitým počtem roli, které mají jasně definovanou strukturu. Mnohá existující řešení předpokládají poměrně jednoduchý systém rolí, který bývá na mnoha webových portálech. Příklady takových rolí jsou administrátor, uživatel nebo editor. Předem známá malá množina. Definice pravidel pak spočívá v určení, které zdroje smí konkrétní role používat.

Existující nástroje vyžadují reprezentaci zdroje (dat) jako objekt. To je dáno předpokladem, že jsou nasazeny v aplikacích, které mají vlastní databáze a znají její přesnou strukturu. To EBIE API nemá. Při návrhu EBIE API se přistoupilo k manuálnímu generování SQL dotazů nad vzdálenou databází. Vývoj nových endpointů v EBIE API tak spočívá ve čtení dokumentace na portálu EBIE a manuálním vytváření SQL dotazů.

Z předchozích dvou důvodů nevypadá reálně použití nástrojů Pundit, CanCanCan a Six. Zbývající kandidát je Consul. Sice je navrhovaný pro použití s nějakým ORM (například ActiveRecords) nad lokální (nebo alespoň vlastní) databází, ale nevyžaduje tuto skutečnost. Consul je vytvořený pro framework Ruby on Rails a očekává tedy jeho strukturu. Je možné se přizpůsobit a integrovat Consul se Sinatrou. Mám však obavy z komplikací při integraci.

Požadavky EBIE API nejsou příliš komplexní, ale jsou specifické. Nakonec byl zvolen vývoj vlastního autorizačního řešení, které bude vytvořeno na míru potřebám EBIE. Neměl by vzniknout rigidní framework bez možnosti úprav. Cílem je, aby nový systém splňoval požadavky a současně byl flexibilní pro případné úpravy a rozšíření do budoucna. K tomu bude nutná znalost fungování Racku a Sinatry, aby bylo možné plně využít jejich potenciálu.

Autorizační systém bude rozdělen na 2 větší logické celky. První je middleware, který se bude starat o identifikaci uživatele a striktní autorizaci. Po úspěšné autorizaci předá middleware informace o omezeních druhé vrstvě. Ta je složena z pomocných metod přímo v aplikaci. Správné fungování obou vrstev závisí na dodržení určitých konvencí a postupů.

6.2.1 Middleware

Pojem middleware v kontextu Sinatry je popsán v sekci 3.2.1. Využití middleware bylo zvoleno pro schopnost analyzovat požadavek a odmítnout ho ještě předtím, něž se jej pokusí zpracovat samotná aplikace. Úloha této vrstvy je zastavit očividně neoprávněný přístup k API. Propustí pouze ty požadavky, které mají smysl být zpracované aplikací. Mělo by být zaručeno, že pokud požadavek projde touto vrstvou, uživatel smí (třeba i omezeně) přistupovat k vyžadované organizační jednotce. Middleware nebude upravovat výsledek požadavku. Pouze předá získané uživatelovi role dále aplikací k dalšímu zpracování.

6.3 Autentizace

Před vlastní autorizací je potřeba uživatele identifikovat. Autentizace je již vyřešena, ale bude se lehce upravovat využití získaných údajů z fakultního autorizačního serveru. Na ukázce 1 v sekci 2.2 je příklad získaných údajů z OAAS Zuul. Přestane se používat striktní kontrola identifikátoru klientské aplikace. Místo toho se začne využívat údaj o username uživatele pro získání dalších informací o uživateli.

Kontrolovaný token musí být získán procesem zvaným „authorization code grant“. Takto získaný token lze poznat podle hodnoty `ROLE_USER` v atributu `authorities` v odpovědi od OAAS Zuul při kontrole. Aplikace poté může zastupovat uživatele při komunikaci s autorizovanými službami používající OAAS Zuul. Jedna z těchto služeb je mimo jiného Usermap API, která se využije pro získání rolí.

Druhá metoda získání tokenu je „client credentials grant“. Ta však slouží pouze k identifikaci klientských aplikací ve fakultních službách. V tomto procesu se žádný uživatel nepřihlašuje a tak access token získaný tímto procesem neobsahuje informaci o uživateli. Použití této metody tak není vhodné pro účely EBIE API. Ve specifikaci OAuth2 je více procesů [22], ale OAAS Zuul aktuálně podporuje pouze ty zmíněné [23].

Během identifikace uživatele probíhají první autorizační kontroly. Na uživatele jsou během procesu kladeny následující požadavky:

- Uživatel (nebo za něj jednající klient) musí v každém požadavku předat access token.
- Token musí být platný a musí obsahovat informaci o uživateli.

Pokud není předán platný token, je požadavek zamítnut a odešle se odpověď `401 Unathorized`.

Po autentizaci se provede první předběžná kontrola rolí. Pokud uživatel nemá žádnou technickou EBIE roli (popsanou v sekci 4.3.4), je odmítnut s odpovědí `403 Forbidden`.

6.4 Autorizace

Nyní je uživatel identifikován a víme, že vlastní alespoň jednu EBIE roli. K rozhodnutí, zda může pokračovat dál potřebujeme mít základní informace o požadovaných datech. K tomu slouží definice datových konců (endpoint).

6.4.1 Endpoint

Endpoint je definice cesty, na které aplikace čeká požadavek. Každý z definovaných konců má určené chování a vrací data. Pokud uživatel odešle požadavek na `/vysledky-predmetu/semestr/B142`, vrátí EBIE API výsledky všech předmětů v semestru B142 (letní semestr v akademickém roce 2014/2015). Tyto definice vytváří vývojář nebo návrhář API. Struktura cest by se měla držet filosofie REST architektury.

Definice datového konce v Sinatra DSL pro výše uvedený příklad by vypadala následovně: `/semestr/:sem_id`. Předpokládejme, že jsme ve jmenném prostoru `/vysledky-predmetu`. Sinatra při zpracování požadavku předá do těla zpracování endpointu parametr se jménem `sem_id`.

Cílem je využít již existujících definic cest a postavit nad nimi autorizační vrstvu. Autorizační pravidla by tak byla uvedena přímo v definicích cest. Autorizační middleware by pak mohl právě s využitím těchto definic určit, jaké role má po uživateli vyžadovat. Je tak využitá již existující definice a od budoucího vývojáře bude vyžadovat pouze dodržení relativně jednoduchých konvencí.

6.4.2 Definice cest

V rámci této práce se autorizují pouze 2 parametry: fakulta a katedra. Aby mohla autorizační vrstva identifikovat, že musí kontrolovat příslušnost k organizační jednotce, musí být některý z parametrů v cestě pojmenován buď `:faculty` nebo `:department`. Při použití těchto parametrů se dá autorizační vrstvě najevo, že musí zkontrolovat příslušnost uživatele k daným jednotkám.

Podívejme se na již existující endpoint `/predmety/fakulta/:fakulta`. V tomto případě by autorizační vrstva ignorovala parametr `fakulta`, protože není správně pojmenován. Pro správnou definici je potřeba změnit definici na `/predmety/fakulta/:faculty`. Tím se stane tento datový konec chráněný. Pro přistup k tomuto konci bude vyžadována správná role (tedy uživatel musí mít roli z dané fakulty).

Na předchozím příkladu byla porušena konvence EBIE API, kde všechny definice endpointů jsou česky. Pro usnadnění práce vývojářům a možnost vytvářet české cesty bude vytvořen systém aliasů, pomocí kterých budou moci vývojáři definovat jiná jména pro klíčová slova `:fakulty` a `:department`. K definici fakulty tak bude moci být použita například česká varianta `:fakulta`. Jedná se tak pouze o kosmetický detail, který nijak nemění funkcionalitu.

6.4.3 Query string

V definicích cest se uvažují pouze parametry, které jsou součástí cesty. Je ale možné přijímat parametr jako takzvaný „query string“. Jak již bylo zmíněno v sekci 3.3.1, Sinatra nerozlišuje URL a query parametry v těle zpracování datového konce. Všechny předané parametry bez rozdílu jsou předány v proměnné `params`. EBIE API není reálně přizpůsobena pro práci s těmito parametry. Na ukázce 15 jsou vidět 2 různé definice. Obě ale volají stejnou metodu

```
get "/semestr/:semestr" do
  encode fetch_data_by(params)
end

get "/semestr/:semestr/fakulta/:fakulta" do
  encode fetch_data_by(params)
end
```

Ukázka kódu 15: Příklad definic cest v EBIE API

s tím, že jí předávají neupravený seznam parametrů od Sinatry. Oba konce ale mohou vrátit stejný výsledek. Požadavky `/semestr/B142?fakulta=10` a `/semestr/B142/fakulta/10` vrátí zcela identické výsledky. Toto chování nemusí být zcela chybné. Je však na pováženou, zda bylo skutečně zamýšleno a zda je vhodné mít 2 různé cesty, jak se dostat ke stejnému výsledku.

V rámci autorizačního systému by zde mohl nastat velmi závažný problém. V druhé definici cesty je explicitně zmíněný parametr `fakulta`. To upozorní autorizační middleware, aby zkontroloval oprávnění. Při využití query stringu by ale žádná kontrola neproběhla. Proto autorizační middleware vedle pojmenovaných URL parametrů bude kontrolovat také všechny query parametry. Jakmile v query parametru objeví jedno z výše uvedených jmen (nebo jejich alias), provede patřičné kontroly. To umožní v budoucnu smysluplně využívat query parametry pro filtrování výsledků bez obav ohledně autorizace těchto parametrů.

Je nutné se vypořádat s kolizí jmen URL a query parametru. Požadavek `/semestr/B142/fakulta/10?fakulta=104` obsahuje 2 parametry se jménem `fakulta`. V tomto případě autorizační systém bude ctít chování Sinatry a upřednostní URL parametr. Query parametr je v tomto případě zcela ignorován a není předán v proměnné `params`.

Záleží na autorech endpointů, jaké typy parametrů zvolí při návrhu. Autorizační middleware kontroluje obě varianty.

6.4.4 Nejednoznačné cesty

Při kontrole parametrů middleware porovná cestu v požadavku s definicemi cest. Je teoreticky možné, že některé definice budou nejednoznačné. Na ukázce 16 je definice nejednoznačných cest. Při požadavku na `/semestr/42`, jaká

```
get '/semestr/:katedra' do
  'Here?'
end

get '/semestr/:fakulta' do
  'Or here?'
end
```

Ukázka kódu 16: Nejednoznačnost v definicích cest

metoda jej má zpracovat? Cesta odpovídá oběma definicím. Sinatra tento problém řeší pořadím definic. První shoda má přednost. V tomto případě bude odpověď `Here?`.

Jedná se o zcela chybný návrh cest a vývojář by měl cesty změnit. I přesto s tím autorizační vrstva počítá a kontroluje parametry všech cest, které odpovídají cestě v požadavku. Autorizační systém tak zkontroluje příslušnost jak k fakultě, tak ke katedře. Doporučuje se však tomuto případu zcela vyhnout a navrhovat jednoznačné cesty.

6.4.5 Kontrola rolí

K vyhodnocení, zda uživatel smí pokračovat potřebujeme znát k jakým datům chce uživatel přistupovat a jaké má role. Role byly získány v procesu autentizace. Potřebné informace o datech jsou v definici cesty popsané v předchozích sekcích. Například při požadavku na `/predmety/fakulta/10` nás ani tolik nezajímá, že uživatel chce seznam předmětů, ale že chce výslovně předměty na konkrétní fakultě. Middleware nezajímá přesně typ dat, která se vracejí. Vývojář datových konců je zodpovědný za smysluplnost vrácených dat. K vyhodnocení oprávnění tedy stačí znát pouze, zda datový konec dovoluje upřesnit katedru nebo fakultu. Připomínám, že se zde bavíme pouze o základní kontrole, zda má smysl předat požadavek dál aplikaci. Omezení výsledné množiny dat podle rolí bude řešeno později.

Podstatou kontroly je tedy porovnat identifikátor organizační jednotky (fakulty nebo katedry) s identifikátorem ve všech rolích uživatele. Protože ani nemůžeme znát přesný charakter vracených dat (krom vazby na organizační jednotku), musí být vyhodnocení oprávnění k přístupu také obecné. Není tedy přípustné vyhodnocovat exaktní shodu a je nutné vzít v potaz hierarchickou strukturu organizačních jednotek. Vztahy mezi katedrou a fakultou je možné popsat následovně

- katedra je součástí fakulty,
- fakulta je nadřízena svým katedrám,
- ČVUT je nadřízeno všem fakultám,
- fakulty jsou součástí ČVUT

Ačkoliv pravidla zní banálně, je potřeba si tyto vztahy uvědomit. V middleware se kontroluje příslušnost k organizační jednotce. Nekontroluje se nutně, zda má uživatel roli konkrétní organizační jednotky. Pro umožnění univerzálního přístupu k API je možné vlastnit roli vázanou přímo na kořen hierarchie organizačních jednotek (ČVUT). Taková role umožní přistupovat k datům všech fakult (a tedy i kateder).

Při kontrole kateder a fakult mohou nastat 4 různé situace.

1. Požadavek neobsahuje identifikátor fakulty ani katedry. Není co kontrolovat a požadavek je předán aplikaci ke zpracování.
2. V cestě je upřesnění fakulty. Dál je puštěn uživatel, který patří k dané fakultě. To mimo jiné znamená, že osoby s rolí podřízené katedry jsou součástí fakulty a jsou puštěni dál.
3. V cestě je pouze parametr katedry. Vyžaduje se tedy příslušnost ke konkrétní katedře. Navíc jsou puštěny ty osoby, které mají přímou vazbu na nadřazenou fakultu.

4. Pokud je požadována kontrola obou parametrů, provede se kontrola smysluplnosti. Jsou zastaveny ty požadavky, ve kterých není ve vztahu katedra a fakulta. Není možné žádat například o data z katedry teoretické informatiky (na fakultě informačních technologií) a současně předat identifikátor pro fakultu strojní. Takový požadavek nedává smysl a je možné jej rovnou zastavit. Pokud požadavek má smysl, provede se kontrola z bodu 3, protože v tomto případě je informace o fakultě irelevantní.

Pravidlo 4 je přítomno z možného důsledného dodržování REST architektury při definici cest. Protože je katedra „pod“ fakultou, bude mít vývojář tendenci vytvořit plnou cestu v podobě `/fakulta/:fakulta/katedra/:katedra`. Ačkoliv je identifikátor fakulty pro kontrolu rolí v tomto případě irelevantní, tak při tvorbě nových datových konců takové cesty budou vznikat. Je tedy vhodné kontrolovat i náležitost katedry k fakultě.

Může se vyskytnout situace, kdy je potřeba kontrolovat striktní vazbu na fakultu. Tedy aby byla vpuštěna pouze osoba s rolí příslušné fakulty a nikoliv z podřízených kateder. Při dodržení zmíněných pravidel to ale možné není, protože pokud korektně označíme kontrolovaný parametr jako `:fakulty`, budou vpuštěni i lidé z podřízených kateder. Jako řešení je možné využit vlastností kontroly katedry. Podle pravidla 3 se katedra kontroluje striktní kontrolou role. Pokud tedy označíme identifikátor fakulty jako katedru, zkontroluje se přímá příslušnost k organizační jednotce. Takové chování ale bude vývojáře mást. Proto bude zaveden nový alias `:fakulty_strict` pro parametr `:department`. Vývojáři tak mohou kontrolovat přímou vazbu k fakultě přehledněji.

Pokud uživatelův požadavek úspěšně projde přes všechny kontroly, je předán aplikaci k vyřízení. Současně s tím se předají informace o uživatelových rolích, aby mohl být vytvořen korektní SQL dotaz na databázi.

6.5 Omezení výsledku

Omezení výsledků probíhá již v samotné EBIE API. Protože se vytváří SQL dotazy nad databází ručně, není možné jednoduše automatizovat tuto fázi autorizace. Pro budoucí vývojáře ale budou existovat pomocné metody a příklady, jak postupovat při vývoji nových endpointů. SQL dotazy jsou tvořeny na míru každému endpointu. Díky tomu je možné dotazy optimalizovat, ale přidává to práci navíc. Při práci vývojář musí mít při ruce dokumentaci datových zdrojů. Tato dokumentace je dostupná na portálu EBIE.

6.5.1 Tvorba SQL dotazu

Při tvorbě dotazů je potřeba, aby vývojář uvažoval, jaký charakter data mají a jaké role k nim smí přistupovat. Základní autorizaci už za něj provedl mi-

ddleware (pokud správně zadefinoval cesty). Je však nutné vytvořit takový dotaz, aby byla vrácena pouze omezená množina dat.

V případě požadavku na konkrétní organizační jednotku není o čem uvažovat. Dotaz bude obsahovat takové podmínky, aby vrátil vyžádaná data. Pokud ale bude v požadavku jakýkoliv stupeň volností, musí se data omezit. Volnost může být například u požadavku na fakulty. Jedná se o jednoduchý požadavek, který portál EBIE používá jako před-filtr pro další výběr dat. Nemá smysl vracet seznam všech fakult, protože osoba nebude mít přístup ke všem fakultám. Je tedy na místě omezit fakulty pouze na takové, ke kterým člověk patři.

```
SELECT DISTINCT fakulta_id, nazev_fakulty AS fakulta_nazev
FROM d_studium
ORDER BY fakulta_id;
```

Ukázka kódu 17: SQL dotaz v Ruby

Na ukázce 17 je dotaz pro `/vysledky-predmetu/fakulty` bez dodatečného omezení výsledku. Výsledek takového dotazu vrátí všechny fakulty. To je víceméně správné, pokud bychom nechtěli omezovat data na základě rolí. Je tedy vhodné přizpůsobit dotaz tak, aby mohl vrátit data pouze relevantní fakulty.

Autorizační middleware pro tento účel předá datovou strukturu se seznamem všech organizačních jednotek, ke kterým má uživatel přímou vazbu. V ukázce 17 stačí přidat do nějaké části dotazu klauzuli `WHERE IN`. Místo přidání záleží na struktuře a charakteru dat. V tomto případě je jediné možné místo před klíčovým slovem `ORDER BY`. Na ukázce 18 je již upravený dotaz. Je přidána podmínka. Tato podmínka by měla být generována pomocnou metodou. Vývojář by neměl strávit výrazně více času při vytváření dotazů, něž to bylo doposud.

```
SELECT DISTINCT fakulta_id, nazev_fakulty AS fakulta_nazev
FROM d_studium
WHERE fakulta_id IN ('10','74')
ORDER BY fakulta_id;
```

Ukázka kódu 18: Upravený SQL dotaz

Pro každý datový konec se bude místo a jméno filtrovaného slupce lišit. V některých případech nebude tabulka obsahovat sloupec `fakulta_id`, ale obecný `orgj_id`. V některých schématech dokonce není sloupec `katedra_id`. Je to z toho důvodu, že v dané datové kostce není informace o jednotlivých katedrách relevantní. Ve sloupci `orgj_id` bývají identifikátory jak kateder, tak fakult. Po vývojáři je tak vyžadována analýza charakteru vracených dat k tomu aby správně filtroval výsledky. Se samotnou tvorbou dotazů se mu po-

kusí pomoci existence pomocných metod pro generování SQL filtrů se správným seznamem organizačních jednotek.

6.6 Pomocné metody

Mnoho operací nad seznamem povolených organizačních jednotek půjde předpřipravit pro snadné použití. Proto vznikne soubor pomocných metod, které mají za cíl vývojářům usnadnit práci s autorizačními podmínkami. Budou se tak moci soustředit na vznik samotných SQL dotazů. Pomocné metody budou poskytovat

- Seznam přímých vazeb na fakulty. Tento seznam bude obsahovat pouze přímou příslušnost k fakultě.
- Seznam povolených fakult. Osoba z katedry bude mít v tomto seznamu svou nadřízenou fakultu.
- Seznam povolených kateder. V tomto seznamu by měli být všechny katedry, ke kterým uživatel smí přistoupit. Role s fakultou automaticky opravňuje přistupovat ke všem podřazeným katedrám.
- Seznam všech povolených organizačních jednotek. Jedná se o sjednocení předchozích seznamů.
- Konverze Ruby pole do SQL seznamu.
- Vytvoření řetězce s klauzulí `WHERE IN` pro použití v SQL dotazu.

Kapitola 7

Implementace

Protože vzniká rozšíření již existující aplikace. Je mnoho faktorů daných. Proto je třeba se přizpůsobit již použitým technologiím a architektuře projektu. Pracujeme převážně v jazyce Ruby. EBIE API je postavena na webovém frameworku Sinatra s pomocí gemu Pliny, který vygeneroval adresářovou a logickou strukturu aplikace. Základní struktura aplikace je následující:

- `bin` .... binární soubory, užitečné pro vývoj
- `config` .... konfigurace aplikace
  - `initializers` .... inicializátory spuštěné při startu aplikace
  - `config.rb` .... nastavení pravidel pro parametry aplikace
- `db` .... soubory databáze, nevyužité
- `lib` .... zdrojový kód aplikace
  - `endpoints` .... definice endpointů
  - `application.rb` .... vstupní bod aplikace
  - `initializer.rb` .... hlavní inicializátor, importuje jednotlivé části
  - `routes.rb` .... definice middleware a připojení endpointů
- `spec` .... RSpec testy
- `.env` .... definice proměnných prostředí
- `config.ru` .... konfikurační soubor pro spouštěč `rackup`
- `Gemfile` .... definice závislostí aplikace

Autorizace je nadstavba funkcionality a proto by měla být logicky oddělena od zbytku aplikace. Většina funkcionality nového autorizačního systému bude tedy ve složce `/lib/authorization`. Jednotlivé komponenty se následně pouze importují na patřičných místech. Je tak zajištěna přehledná struktura a při dalším vývoji nebude překážet.

Základní komponenty řešení jsou:

- Middleware
- Autorizační modul

- Pomocná třída pro konverzi identifikátorů
- Struktura pro reprezentaci uživatele a jeho rolí
- Sada pomocných metod pro vytváření SQL dotazů

V následujících sekcích popíši jejich úlohy a jak fungují.

7.1 Autorizační modul

Autorizační modul funguje jako knihovna všech autorizačních metod. Používá ji převážně middleware ke své práci. Obsahuje metody pro autentizaci, autorizaci a získávání informací o uživateli. Jako jediná komponenta pracuje přímo s reprezentací uživatele. Pokud modul zjistí, že uživatel má nedostatečné role, vyhodí příslušnou výjimku.

Výjimky jsou logicky spárované s HTTP status kódy a nástroj Pliny se stará o jejich korektní zpracování. Proto je možné na různých místech aplikace vyvolat například výjimku `Pliny::Errors::Forbidden`, která je odchycena. Uživateli přijde korektní HTTP odpověď s kódem 403.

Při pojmenování metod jsem se inspiroval naučenými konvencemi Ruby a metody obsahující vykřičník jsou potenciálně „nebezpečné“. To znamená, že mohou vyhodit výjimku nebo upravují předávané parametry.

7.1.1 Autentizace

Autentizace je přímočará. Na ukázce 19 je celý proces. Metoda přijímá access token předaný v požadavku. EBIE API očekává token v hlavičce požadavku ve tvaru `Authorization: Bearer {access token}`.

Metoda vrací objekt reprezentující uživatele a jeho role. Pokud vrátí `nil`, stala se chyba a uživatel není identifikován. V tento moment nevadí, pokud uživatel nemá žádnou vyhovující technickou roli. Účel metody je identifikovat uživatele. Proto nevyhazuje žádné výjimky.

7.1.2 Autorizace

První autorizační kontrola je na existenci uživatele. Pokud selhala identifikace (bylo vráceno `nil`), zastaví se požadavek a vrátí se `401 Unathorized`. Tato kontrola je v metodě `check_user!`. Je volána pokaždé, když je potřeba zkontrolovat, že uživatele známe.

Další kontrola je ověření vlastnictví alespoň jedné EBIE role. Pokud uživatel nemá žádnou EBIE roli, je odmítnut s kódem `403 Forbidden`. Zde se již nevrací `401 Unathorized`, protože známe identitu uživatele a autentizace proběhla úspěšně. Chybějící role znamená, že není autorizován pro přistup k EBIE API a proto dostane odpověď `403 Forbidden`. Rozdíly mezi HTTP status kódy byly diskutovány v sekci 6.1.1.

```
def authenticate_and_create_user bearer_token
  return nil if bearer_token.nil? || bearer_token.empty?
  begin
    # Authenticate user via Zuul
    zuul_response = zuul_client.get do |req|
      req.url "/oauth/check_token?token=#{bearer_token}"
    end

    # Response must be positive = Zuul recognizes him
    if zuul_response.status == 200
      username = zuul_response.body["user_name"]
      return nil if username.nil? || username.empty?

      # Get users roles
      usermap_response = usermap_client(username,
      ↪  bearer_token).get
      roles = usermap_response.body["technicalRoles"]
      return User.new(roles)
    end
  rescue Faraday::Error::ConnectionFailed
    return nil
  end
  nil # user didn't authenticate correctly
end
```

Ukázka kódu 19: Přihlášení uživatele

Klíčová metoda volaná z middleware je `authorize_params!`(ukázka 20), která kontroluje, zda uživatel má dostatečná oprávnění pro použití endpointu. Samotná kontrola je delegována na reprezentaci uživatele. Zde se pouze vyhodnotí výsledek volání. Tento krok byl učiněn protože objekt uživatele obsahuje potřebné informace k provedení kontroly, stačí pouze předat identifikátory organizačních jednotek ke kontrole.

Metoda `process_params` sjednocuje zadefinované aliasy do jednoho parametru, aby bylo možné s nimi dále pracovat.

7.2 Middleware

Autorizační middleware je první linií při autorizaci. Slouží k autentizaci a základní autorizaci požadavků. Na základě informace v definicích cest a rolích uživatele rozhodne, zda má smysl se zabývat požadavkem.

Middleware je volán metodou `call`, která zpracuje a předá požadavek dál v rouře. V této metodě postupně proběhne autentizace, autorizace a při-

```
def authorize_params! params
  check_user!
  processed_params = process_params(params)
  requested_faculty = processed_params["faculty"]
  requested_department = processed_params["department"]
  return true if
  ↪  current_user.has_required_role?(requested_faculty,
  ↪  requested_department)
  fail Pliny::Errors::Forbidden, "Unathorized to access this
  ↪  organization unit."
end
```

Ukázka kódu 20: Autorizace podle parametrů

pravení seznamu autorizovaných organizačních jednotek. Seznam povolených jednotek je přidán do požadavku (proměnná `env` obsahuje informace o požadavku), který se předá aplikaci ke zpracování. Pomocná metoda `bearer_token`

```
def call(env)
  authenticate! bearer_token(env)
  authorize_user!

  params = get_params(env)
  authorize_params! params
  restrictions = create_restriction_params
  env["authorization.allowed_units"] = restrictions
  response = @app.call(env)
  return response
end
```

Ukázka kódu 21: Middleware metoda `call`

extrahuje token z hlavičky požadavku. Pokud jej nenalezne, vrátí `nil`. Metoda `create_restriction_params` je z modulu pro autorizace, která deleguje úlohu na objekt reprezentující uživatele.

Zajímavá metoda je `get_params`, která vrátí seznam předaných parametrů z požadavku. Tato úloha není zcela triviální, protože je nutné znát definice endpointů, aby bylo možné rozpoznat a extrahovat URL parametry. Základní myšlenka a část kódu pochází přímo ze Sinatry([24] metody `process_route` a `route!`). Objekt `pattern` je instance třídy `Mustermann`. Mustermann je speciální parser definic cest v Sinatře. Díky jeho existenci je možné bez velké práce získat všechny potřebné parametry a hlavně je zajištěno stejné chování jako v Sinatře při směrování požadavku. Poslední řádek získává query parametry a přidává je k získaným URL parametrům. Pokud je parametr přítomný v URL

parametrech, má přednost a není přepsán query parametrem.

```
def get_params(env)
  route = get_current_route(env)
  patterns = @settings.routes[env["REQUEST_METHOD"]]
  params = {}
  patterns.each do |pattern, condition, block|
    new_params = pattern.params(route)
    params.merge!(new_params) do |key, old_val, new_val|
      Array === old_val ? old_val << new_val : [old_val,
      ↪  new_val]
    end if new_params
  end
  return Sinatra::Request.new(env).params.merge params
end
```

Ukázka kódu 22: Získání parametrů z požadavku

Proměnná `@settings` obsahuje nastavení (včetně definic cest) následující aplikace v rouře. Proto je nutné, aby middleware byl zasazen přímo před aplikaci a nebyl za ním žádný další middleware. Je možné, že se v budoucnu podaří najít způsob, jak z této podmínky polevit.

Protože je technicky možné, aby cesty nebyly jednoznačné, prochází se všechny shody. Pokud cesty mají stejně pojmenované parametry, vytvoří se pole hodnot. Obecně se jedná o chybu a takovému případu by se mělo vyhnout definicemi jednoznačných cest (nebo alespoň jinak pojmenovat přijímané parametry). Middleware předá seznam dále ke zpracovaní bez vynucení tohoto pravidla. V budoucnu je možné případ povolit a upravit zpracovaní těchto parametrů.

7.3 Reprezentace uživatele

Třída `User` se stará o reprezentaci uživatelových rolí a práce s nimi. Slouží k porovnávání rolí a generování seznamu povolených organizačních jednotek.

Při vytváření instance třídy se přijímá pole všech rolí (nemusí být nutně technické). Z tohoto seznamu jsou vybrány pouze technické EBIE role popsané v 4.3.4. Ukládají se pouze přidružené identifikátory organizačních jednotek. Je možné, že se vytvoří uživatel bez jediné role. Takový uživatel byl korektně autentizován, ale neprojde autorizací.

Na ukázce 23 je vytvoření uživatele z dodaných rolí. Je jedno, jaké řetězce jsou v poli, inicializátor si „vytáhne“ pouze relevantní role. V proměnných `ROLE_PREFIX` a `ROLE_SUFFIX` jsou pouze definovány povinné části role (definované v sekci 4.3.4) `"T-EBIE-"` a `"-UZIVATEL"`.

```
def initialize(roles)
  if roles.respond_to? :to_ary
    extracted_ids = roles.map do |role|
      matched = role.match(
      ↪  /\A#{ROLE_PREFIX}(?<id>\d{5})#{ROLE_SUFFIX}\Z/ )
      matched ? matched["id"] : nil
    end
    @role_set = Set.new(extracted_ids.compact)
  else
    @role_set = Set.new()
  end
end
```

Ukázka kódu 23: Inicializace uživatele

Třída mimo jiné obsahuje metody pro určení, zda nějaký identifikátor označuje fakultu, nebo zda patří pod určitou fakultu. Tyto pomocné metody se používají hlavně při kontrole oprávnění v metodě `has_required_role?`.

7.3.1 Kontrola rolí

Metoda `has_required_role?` je jedna z nejpodstatnějších metod celého systému. Zde se vyhodnocuje, zda uživatel má dostatečná oprávnění. Metoda implementuje pravidla ze sekce 6.4.5. Jako parametr přijímá identifikátor fakulty a katedry. Oba parametry jsou povinné, ale v oba dovolují předání hodnoty `nil`.

V ukázce 24 není uvedena kontrola předaných parametrů. Kontroluje se zda se předané parametry „chovají" jako `String`. Metoda předpokládá, že je v každém parametru předán jediný identifikátor ve formě řetězce. Nepřijímá tedy žádná čísla nebo struktury. Pokud ale předaný objekt má implementovanou metodu `to_str`, dává tím najevo, že s ním je možno manipulovat jako s řetězcem.

Pokud jsou oba předané parametry `nil`, je rovnou vráceno `true`, protože není vyžadován přistup k žádné organizační jednotce. Naopak pokud jsou předány oba parametry, provede se test smysluplnosti. Ten spočívá v kontrole příslušnosti katedry k fakultě.

Zbývající 2 případy jsou kontrola příslušnosti k fakultě a katedře. V případě katedry se ptáme, zda má uživatel roli fakulty, nebo podřízené katedry v metodě `belongs_to_faculty?`. U náležitosti ke katedře nás zajímá, zda uživatel nemá nadřazenou roli (fakultní nebo master roli) a až poté se zkontroluje přímá shoda role.

Nikomu nebrání se pokusit tuto metodu oklamat a předat ji zavádějící informace. Například v parametru pro fakultu předat identifikátor katedry.

```ruby
def has_required_role? requested_faculty, requested_department
  ...
  faculty_id = requested_faculty ?
  ↪ IdMapper.dwh_to_usermap(requested_faculty) : nil
  department_id = requested_department ?
  ↪ IdMapper.dwh_to_usermap(requested_department) : nil
  unless department_id
    return faculty_id ? belongs_to_faculty?(faculty_id) : true
  else
    departments_faculty_id = extract_faculty_id department_id
    if faculty_id
      return false unless
      ↪ faculty_id.eql?(departments_faculty_id)
    end
    return true if @role_set.include?(departments_faculty_id) ||
    ↪ master_role?
    return @role_set.include? department_id
  end
end
```

Ukázka kódu 24: Kontrola rolí

V tom případě ho metoda `belongs_to_faculty?` zastaví, protože zjistí, že předaný identifikátor neodpovídá fakultě.

Předání fakultního identifikátoru v parametru pro katedry je povoleno. Způsobí to, že se k takovému identifikátoru bude systém chovat jako ke katedře a ověří tak striktní shodu role.

7.4 Identifikátory organizačních jednotek

EBIE API pracuje s různými systémy. Mezi nimi ale nejsou jednotné identifikátory organizačních jednotek. Proto je nutné je vzájemně převádět, aby bylo možné vyhodnotit pravidla pro přistup. V aplikaci se pracuje se dvěma typy identifikátorů. Jejich pojmenování je založeno na původu a nemusí odpovídat jejich oficiálnímu označení.

`DWH_id` Identifikátor z datového skladu. Hodnoty nemají žádnou strukturu. Tyto identifikátory přijímá a vrací EBIE API.

`Usermap_id` Identifikátor používaný na Usermap. Jsou součástí byznys i technických roli v ČVUT IdM. Má srozumitelnou hierarchickou strukturu. Na první pohled lze usoudit, že jednotka 18102 je součástí jednotky 18000. Nemusíme znát, k jakým konkrétním jednotkám patří k tomu, abychom určili vztah mezi nimi.

Je zřejmé, že kontroly autorizačních pravidel se budou velmi jednoduše provádět nad Usermap identifikátory. EBIE API ale přijímá v parametrech identifikátory z datového skladu. Současně takové identifikátory vrací. Proč API nekomunikuje pomoci Usermap identifikátorů? Protože v době vzniku komunikovala výhradně s databází. Bylo tak logické používat pouze `DWH_id`. EBIE API existovala pouze pro potřeby portálu EBIE a neobsahovala autorizací. Použití `Usermap_id` by tak pro tehdejšího vývojáře byla práce navíc. Bohužel v aktuálním stavu je využití `DWH_id` neintuitivní a bylo by vhodné API upravit. To však není tématem této práce.

Pro kontrolu rolí se využívá hierarchická struktura `Usermap_id`. V parametrech je ale předán identifikátor z datového skladu. Při vytváření SQL dotazu je na druhou stranu nutné používat `DWH_id`. Povolené organizační jednotky jsou ale získávány z rolí, které obsahují `Usermap_id`. Vznikla tak potřeba převádět `DWH_id` a `Usermap_id` mezi sebou.

Při vytváření nových endpointů vývojář pracuje pouze s identifikátory z datového skladu. Proto veškeré konverze probíhají uvnitř autorizačního systému, tak aby o nich vývojář nemusel vědět. V případě množiny povolených organizačních jednotek, které middleware předává jsou všechny identifikátory převedeny z `Usermap_id` na `DWH_id`.

7.4.1 Převody identifikátorů

Ke konverzi jednotlivých identifikátorů slouží třída `IdMapper`. Jeho použití je vidět na ukázce 24, kde se převádí předané parametry z `DWH_id` na `Usermap_id` pro použití při kontrolách. Třída je využita i pro tvorbu SQL dotazů. Nabízí metodu pro převod fakulty na seznam všech jejích kateder v podobě DWH identifikátorů.

Při načtení třídy jsou načtena data ze souboru obsahující informace o identifikátorech. Vytvoří se tak neměnná struktura, která je následně používána k převodům. Teoretická nevýhoda tohoto řešení je potřeba přidat identifikátory ručně, pokud vznikne nová fakulta nebo katedra. To je však tak neobvyklý jev, že to není problém. Pokud by skutečně vznikla nová organizační jednotka, přidání jednoho řádku do mapovací struktury není příliš obtížné.

7.4.1.1 Zdroj dat

Data potřebná pro převod identifikátorů organizačních jednotek jsou uložena v souboru `org_units.yaml`. Byl zvolen formát YAML pro jeho čitelnost a jednoduchost. Je tak možné bez velkých problémů upravovat obsah souboru. Na ukázce 25 je část definice organizačních jednotek. Klíče jsou `DWH_id` a hodnoty jsou `Usermap_id`. Přestože jsou dané identifikátory numerické, pracuje se s nimi jako s řetězci. Proto všechny identifikátory musí být napsány jako řetězec. Třída `IdMapper` soubor zpracuje a vytvoří si všechny potřebné struktury pro převod identifikátorů oběma směry.

```
"10":
  uid: "18000"  # Fakulta informačních technologií
  departments:
    "1004106": "18102"  # katedra softwarového inženýrství
    "347473675005": "18105"  # katedra aplikované matematiky
    "1006006": "18101"  # katedra teoretické informatiky
    "1006106": "18103"  # katedra číslicového návrhu
    "1006206": "18104"  # katedra počítačových systémů
"473":
  uid: "17000"  # Fakulta biomedicínského inženýrství
  departments:
    "574": "17111" # k. zdrav. oborů a ochrany obyv.
    "474": "17110" # katedra biomedicínské techniky
    "573": "17101" # katedra přírodovědných oborů
    "575": "17112" # katedra biomedicínské informatiky
    "593": "17220" # společné pracoviště BMI ČVUT a UK
    "718107954505": "17120" # Katedra ICTM
```

Ukázka kódu 25: Obsah souboru pro mapování

Data obsažena v souboru `org_units.yaml` byla získána z datového skladu. Při převodu dat do YAML nastal jeden problém. Některé katedry neměly `Usermap_id`. Identifikátory chybí u organizačních jednotek, které vznikly později. U fakulty informačních technologií toto nastalo u katedry aplikované matematiky, která vznikla jako poslední z fakultních kateder. Naštěstí bylo možné identifikátory doplnit ze školního Usermap portálu, kde identifikátory byly. Jde však o problém, který upozorňuje na možné rozdíly oproti realitě v datovém skladu.

7.5 Vytváření SQL dotazů

Tvorba SQL dotazů bude na budoucích vývojářích EBIE API. Během tvorby dotazů budou muset myslet na to, jaký rozsah dat smí vrátit a jak dotaz správně formulovat. Pro malé usnadnění práce mohou využít pomocných metod dodaných s autorizací. Tyto pomocné metody pracují nad seznamem povolených organizačních jednotek, který byl vytvořen při autorizaci.

Všechny metody jsou definovány v `authorization_helpers.rb`. Modul je vytvořen a importován jako sada doplňků do Sinatry. Hlavní 2 metody jsou `where_faculty_statement` a `where_department_statement`. Jejich základní úlohou je vytvořit část SQL dotazu obsahující omezení na fakulty nebo katedry. Tyto metody používají další, obecnější metody, které v případě nutnosti mohou vývojáři také použít. Jejich definice a chování si mohou přečíst ve zdrojovém kódu nebo v manuálu. Blíže bude popsána pouze jedna z hlavních

metod, protože druhá je velmi podobná.

```
def where_faculty_statement(attr_name: 'fakulta_id', faculty_id:
↪  nil, direct: true, skip_keyword: false)
  if faculty_id
    faculties = [faculty_id]
  else
    faculties = direct ? get_allowed_faculties :
    ↪  get_accesable_faculties
    return "" if faculties == :all
  end
  where_statement faculties, attr_name, skip_keyword
end
```

Ukázka kódu 26: Pomocná metoda pro omezení fakult

Na ukázce 26 je implementace jedné z metod pro vytváření SQL konstrukce `WHERE IN`. Předané parametry mění podobu výsledného řetězce

Metoda zavolána bez parametrů vytvoří `WHERE 'fakulta_id' IN`, kde v závorce bude seznam fakult, ke kterým uživatel přímo patří. To je vhodné pro striktní kontrolu fakulty nebo k zobrazení pouze fakultních informací například děkanům.

Parametr `attr_name` změní jméno sloupce v dotazu na předané jméno. Při předání jména `"orgj_id"` bude výsledný dotaz `WHERE 'orgj_id' IN`. Toto je vhodné, pokud tabulka neobsahuje sloupec `fakulta_id` a potřebujeme filtrovat podle identifikátoru organizační jednotky.

V případě, že uživatel žádá specifickou fakultu se požadovaný identifikátor předá v parametru `faculty_id`, Víme, že uživatel smí přistupovat k datům dané fakulty, protože prošel autorizační vrstvou. Ostatní omezení jsou irelevantní, protože uživatel chce jednu konkrétní fakultu. Vrátí se tedy fragment ve tvaru `WHERE 'fakulta_id' = '123'`.

Parametrem `direct` se upravuje, zda nás zajímá přímá vazba k fakultě. Pokud ne, je vrácen seznam všech fakult, ke kterým uživatel nepřímo patří. Člen katedry nepřímo patří k nadřízené fakultě. Nastavení parametru na `false` je vhodné pro zobrazení nabídky fakult, ze které si uživatel vybere z jaké fakulty chce data.

Někdy je potřeba přidávat seznam fakult do složených podmínek ve tvaru `WHERE (podminka) AND/OR IN`. V tom případě nemusí být žádoucí, aby výsledek obsahoval klíčové slovo `WHERE`. Proto je umožněno předat jako další argument `skip_keyword: true` a při generování se vynechá `WHERE`. Výsledek má tvar `'fakulta_id' IN`. Je tak možné předat seznam i do složených podmínek.

V případě, že uživatel vlastní univerzální roli, je vrácen prázdný řetězec. Prázdný řetězec nijak nenaruší strukturu SQL dotazu a způsobní, že se nefil-

truje podle fakulty. Samozřejmě toto není aplikováno pokud si uživatel žádá specifickou fakultu.

Podobně funguje i metoda pro vytvoření fragmentů pro katedry. Obě metody by měly být schopné pokrýt velké množství případů a při vhodném použití umožní vývojářům vkládat části dotazů se správným obsahem do jejich SQL dotazů.

Kapitola 8

Testování

8.1 Automatické testy

První část ověření jsou automatické unit testy, které testují základní funkcionalitu jednotlivých modulů a jejich metod. Bohužel EBIE API neměla žádné rozumné testy. Jediné testy, které byly přítomné byly automaticky generované nástrojem Pliny. Bohužel byly zastaralé (testovaly neexistující endpointy) a hlavně nefunkční. Předpokládaly existenci vlastní DB a to se již dávno změnilo. Z tohoto jsem usoudil, že nikdo předtím nevytvářel pro EBIE API testy a nevyužil dostupné nástroje. I přesto jsem se pokusil neměnit vygenerovanou šablonu a postavil sadu testů pro autorizační řešení pomocí již dostupných nástrojů. Použité nástroje jsou následující:

RSpec Testovací framework poskytující DSL pro rychlý vývoj testovacích scénářů. Současně poskytuje DSL pro mockování. [25]

Webmock Nástroj pro mockování HTTP požadavků. [26]

SimpleCov Umožňuje analyzovat úroveň pokrytí kódu při testování. [27]

Rack::MockRequest Umožňuje mockovat požadavky na Rack aplikaci.

Jediný dodatečný nástroj, který jsem doplnil, byl Webmock, abych mohl simulovat chování autorizačního serveru Zuul a Usermap API. Současně dovoluje vypnout veškerou HTTP komunikaci mimo aplikaci. Díky tomu aplikace nekomunikuje s okolím při testování.

Každý modul byl testován samostatně. Pokud to bylo možné, byly jejich závislosti odříznuty a nasimulovány. K tomu slouží takzvaný „mock“ objekt. Takový objekt zastupuje ten pravý a nahrazuje jeho chování. Je pak možné vytvořit takový mock, který vždy vrátí konkrétní hodnotu při zavoláni jeho metody. Také může kontrolovat, zda byla metoda zavolána a s jakými parametry. Takové kontroly se hodí, pokud se chceme soustředit na chování modulu, který závisí na jiných částech. Speciální úlohu zastávala třída `IdMapper`. Její

převádění bylo použito během psaní testů. Usermap identifikátory jsou čitelnější a lépe se s nimi pracuje.

Testy nikdy nezaručí, že v aplikaci není žádná chyba, protože kontrolují pouze definované případy. Při každém spuštění testů je pořadí testů náhodné. Díky tomu se dají nalézt například problémy s neúmyslným ovlivňování stavu aplikace jako takové. Při rozumné definici testů je možné nalézt základní nedostatky. Během testování jsem narazil na několik chyb, které následně byly ošetřeny a byl přidán specifický test, aby se konkrétní chyba nevrátila. Pokud se upraví nebo přidá nová funkcionalita, je vhodné přidat nové testy pro validaci. Při každé úpravě je nutné spouštět testy aby bylo jisté, že nová, nebo upravená funkcionalita nenabourala zbytek.

Authorization (98.78% covered at 12.81 hits/line)

5 files in total. **246** relevant lines. **243** lines covered and **3** lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
lib/authorization /authorization.rb	**95.95 %**	143	74	71	3	4.9
lib/authorization /authorization_helpers.rb	**100.0 %**	113	54	54	0	7.7
lib/authorization /id_mapper.rb	**100.0 %**	53	29	29	0	18.1
lib/authorization /role_auth_middleware.rb	**100.0 %**	80	39	39	0	21.2
lib/authorization/user.rb	**100.0 %**	113	50	50	0	20.4

Showing 1 to 5 of 5 entries

Obrázek 8.1: Pokrytí kódu po testování

Pro otestování autorizace vzniklo přes 150 testů, které se pokouší pokrýt co nejvíce případů. Na obrázku 8.1 je výstup nástroje SimpleCov, který vytváří HTML reporty ohledně pokrytí kódu. Téměř kompletní pokrytí kódu je dobré znamení. Nezaručuje však otestování všech případů. Většina testů testuje obvyklé a očekávané případy. Očekávaná nejsou pouze správná použití metod a parametrů, ale i možné chyby nebo problémy při používání aplikace. Například je simulován případ, kdy autorizační server Zuul neodpoví na požadavek autentizovat uživatele. V tom případě aplikace musí prohlásit, že uživatele není možné identifikovat a ukončit jeho požadavek.

Na ukázce 27 je ilustrován test chování aplikace, pokud autorizační server Zuul aktivně odmítne uživatelův access token. Na ukázce jsou dokonce testy 3. Jsou ale tak podobné, že bylo možné je definovat v cyklu. Každý jednotlivý test testuje jeden aspekt. Proto jsou testy velmi krátké a mnohdy se skupiny testů liší pouze v detailech. Při testování kontroly oprávnění tak vzniklo například 16

```
[500, 400, 404].each do |code|
  it "returns #{code}, throw exception" do
    rs = stub_request(:any, /.*auth.fit.cvut.cz.*/)
      .to_return(status: code)
    expect{dummy.authenticate!("123")}
      .to raise_error Pliny::Errors::Unauthorized
    remove_request_stub(rs)
  end
end
```

Ukázka kódu 27: Test při selhání externí služby

různých kombinací argumentů a očekávaných výsledků. Následně se postupně testovaly všechny definované kombinace volání konkrétní metody.

Všechny testy aplikace (kde většina testů jsou pro autorizaci) se provedou pod pětinu sekundy. Pokud započítáme kompletní čas na načtení a provedení, jsme na 3 sekundách. Samozřejmě záleží na stroji, na kterém je to spuštěno, ale průměrný kancelářský počítač nebude pracovat o moc déle.

Při testování samotného middleware byla vytvořena velmi malá falešná Sinatra aplikace, která sloužila pro testování jednotlivých typů endpointů. Bylo tak pod kontrolou k čemu se middleware připojuje a jak ovlivňuje danou aplikaci.

Nejsou otestovány jednotlivé endpointy a jejich chování. Někdy v budoucnu by mohli vzniknout další automatizované testy, které ověří chování celé aplikace jako takové. Toto ale není tématem této práce.

8.2 Vyzkoušení EBIE API

Další fází ověření implementace byla schopnost EBIE API komunikovat s nasazenou autorizační vrstvou. Pro vyzkoušení jsem převedl endpointy týkající se výsledků předmětů. Aplikace je během těchto testů nasazena lokálně a je připojena ke všem externím službám. Protože samozřejmě v ČVUT IdM neexistují technické EBIE role, využil jsem whitelist, pomocí kterého si upravuji role podle potřeb. Zajímalo mě, zda aplikace zastaví požadavek při neoprávněném přístupu a zda omezí data podle přidělené role.

Tato sekce slouží také k demonstraci vzniklého řešení. V případě zájmu je možné si aplikaci nasadit lokálně a vyzkoušet. Práce bude nasazena na vývojový server EBIE, kde se ověří, zda skutečně plní svou úlohu za provozu.

8.2.1 Požadavky pro zkoušení

Na přiloženém médiu je zdrojový kód EBIE API. Aktuálnější zdrojový kód bude možné najít na fakultním GitLabu. Nebudu zde popisovat detailní postup

nasazení aplikace. V souboru README u projektu je rámcový postup.

Před lokálním nasazením je nutné si zařídit přistup do databáze (adresu a heslo), nebo si zajistit jiný datový zdroj, který má stejnou strukturu.

Pro přístup k EBIE API je nutné mít ČVUT účet a být autorizován. To spočívá v získání technické role nebo přidáním do whitelistu. EBIE API vyžaduje k přístupu access token získaný pomocí „authorization code grant" přes fakultní OAAS Zuul. Pokud nemáte žádný způsob, jak si token vygenerovat, je možné jej získat na portálu EBIE. Po přihlášení do portálu je na `/apitoken` dostupný platný access token pro komunikaci s EBIE API. Pro přístup k EBIE API přes samotný portál není nutné získávat token, protože portál ho předává sám.

8.2.2 Přímý přístup

Nejprve jsem požadavky posílal přímo na EBIE API. V předávaných parametrech jsem posílal různé kombinace parametrů. Aplikace korektně reaguje na chybějící, prošlý nebo nevhodný token. Na ukázce 28 jsou možná volání EBIE API pomocí nástroje cURL. Doporučuji si uložit access token (nebo celou autorizační hlavičku) do proměnné, pokud máte v úmyslu volat API vícekrát. Token je platný hodinu a je nutné jej po hodině obnovit (například v EBIE portálu dát obnovení stránky).

```
token=ba3deec6-7ee3-46ax-92a4-996cc05424s70
ebie=http://localhost:5000/api/v1

curl ${ebie}/vysledky-predmetu/fakulty

curl -H "Authorization: Bearer ${token}"
↪  ${ebie}/vysledky-predmetu/semestr/B151/fakulta/10

curl -H "Authorization: Bearer ${token}"
↪  ${ebie}/vysledky-predmetu/fakulta/10/predmet/1124406
```

Ukázka kódu 28: Možná volání EPIE API přes cURL

Pro zkoušení EBIE API na vývojovém serveru stačí upravit proměnnou `ebie` na `https://ebie-vyvoj.is.cvut.cz/api/v1`.

První uvedené volání neprojde, protože neobsahuje autorizační hlavičku. Druhé vrací výsledky předmětů v semestru B151 z fakulty informačních technologií. Počet vrácených předmětů záleží na roli, kterou uživatel bude mít.

Pouze fakultní role (`T-EBIE-18000-UZIVATEL`) uvidí všechny předměty. Navíc je této roli dovoleno poslat další parametr za URL. Pokud je za URL přidáno `?nasi_studenti`, vrátí se výsledky všech předmětů které si zapsali studenti fakulty. To zahrnuje i předměty jiných fakult, pokud si je některý

student „naší“ fakulty zapsal. V předmětech jiných fakult se samozřejmě zobrazují pouze statistiky výsledků „našich“ studentů. Toto je ukázka, že vývojář může dále upravovat chování endpointů pomocí query parametrů a využít k tomu informace z autorizační vrstvy.

Pokud by bude mít uživatel pouze roli katedry z fakulty informačních technologií, vrátí se mu pouze výsledky předmětů, které jsou vyučovány na dané katedře. V případě, že by uživatel měl nějakou EBIE roli, ale z jiné fakulty bude jeho požadavek zastaven autorizační vrstvou, protože nemá přístup k datům fakulty informačních technologií.

Ve třetí ukázce je dotaz na konkrétní předmět ve všech semestrech. Zde není moc co filtrovat a pouze se kontroluje uživatelova fakulta. Autorizace předmětu není v middleware zatím možná. Pokud by ale uživatel chtěl předmět z jiné katedry, než ke které má přístup, je SQL dotaz postaven tak, že mu vrátí prázdnou množinu.

8.2.3 Přístup přes portál EBIE

Velmi důležité je, aby nová verze EBIE API bylo schopna komunikovat s portálem EBIE. Portál je stále primárním klientem EBIE API a proto musí být zajištěna jejich kompatibilita. Data z API jsou dostupná v reportech. Reporty o výsledcích předmětů jsou dostupné v předpřipravených reportech v portálu.

Pro porovnání předmětů je nejprve nutné vybrat fakultu. Nabízený seznam obsahuje pouze fakulty, ke kterým uživatel patří. Dále report nabídne k výběru z relevantních semestrů. Následně již načte všechny dostupné předměty podle výběru a uživatelovi role. Fakultním rolím se zobrazí všechny předměty vyučované na fakultě. Možnost zobrazit i výsledky vlastních studentů v předmětech z jiných fakult portál nepodporuje. Je to však námět pro další vývoj.

Report pro vývoj předmětu v čase funguje podobně. Nejprve nabídne fakulty. Následně vytvoří seznam relevantních předmětů. Po výběru předmětu už zobrazí data.

Veškeré generování relevantních výběrů (seznamy fakult, předmětů nebo semestrů) probíhá na straně EBIE API. Portál pošle požadavek na API, aby vrátila seznam například předmětů. EBIE API podle rolí uživatele pak vytvoří správný seznam.

Při použití portálu by nikdy nemělo dojít k autorizační chybě (status 403), protože report v portálu postupně zjišťuje detaily podoby konečného dotazu, který je založen na informacích již z API vrácených. Tato metoda před-filtrů je praktická jak pro uživatele, tak z pohledu API, protože se již od začátku skládá validní dotaz z povolených dat.

Kapitola 9

Zhodnocení a další vývoj

Podařilo se vytvořit funkční systém, který je schopný automaticky autorizovat požadavky a následně pomáhat při dalším vytváření omezené množiny dat. Volba implementace vlastního řešení nebyla zcela zcestná a umožnila se soustředit na doménové požadavky. Stále si myslím, že forma middleware k automatické autorizaci je smysluplná, protože middleware zastaví neautorizovaný přistup rovnou a k aplikaci si tak nedostane. Další důvod je ten, že autorizace se provádí pokaždé a je tedy vhodné ji oddělit od aplikační logiky.

Pokud by bylo nutné povolit přistup k některým cestám i neautentizovaným lidem (tedy bez nutnosti se identifikovat), bylo by nutné middleware upravit tak, aby takové požadavky na dané endpointy pustil. V aktuálním stavu pokaždé vyžaduje autentizaci (a následně autorizuje).

9.1 Mezipaměť

Pro snížení množství požadavků na autorizační server Zuul a Usermap API je možné využít mezipaměť (cache), ve které vy se na krátkou dobu uchovávaly výsledky volání daných služeb. Jako klíč by mohl sloužit uživatelův access token. Délka života záznamu by byla maximálně délka životnosti access tokenu, která je omezena na jednu hodinu. Pro potřeby dat z Usermap API je to zcela v pořádku, protože data z Usermap se nemění tak často. V případě OAAS Zuul bych ale uchovával výsledky volání na kratší dobu, protože je možné ručně zneplatnit token. Takové výsledky by mohli být ukládány na řádově minuty. I tak by to mohlo snížit množství dotazů v krátkém intervalu.

9.2 Tvorba SQL dotazů

V sekci 6.2 jsem zavrhl gem Consul. Rozhodnutí nelituji, protože forma middleware mi je bližší pro autorizaci. Pro omezování dat v aplikaci je Consul ale stále velmi atraktivní. V budoucnu by stálo za to se podívat na jeho de-

tailní implementaci a rozmyslet si, zda by nebylo vhodné upravit strukturu vytváření endpointů. Nemuselo by se jednat nutně o využití gemu Consul, ale o lepši realizaci tvorby SQL dotazů. Jedna z možných cest by bylo vytvořit vlastní ORM, které by využívalo informace z autorizace. Consul je víceméně nadstavbou pro existující ORM jako je ActiveRecord.

V aktuální podobě musí vývojář sám vymýšlet SQL dotazy. Autorizace pomáhá při vytváření filtrů dodáním správné množiny organizačních jednotek v korektním formátu. Více by ale práci usnadnil nástroj pro chytré generování celých SQL dotazů. Takový nástroj ale může být velmi obtížné implementovat, protože se využívá více schémat (pro každý typ reportu jiné) a tato schémata nejsou v aplikaci definována. Muselo by se tedy jednat o nějaký obecný generátor.

I pokud by byl dostupný nástroj pro snadné generování SQL dotazu, není možné nahradit návrh endpointů. Vždy bude nutné se posadit, vzít si tužku a papír a rozmyslet si:

- Jaká data bude endpoint poskytovat?
- Jak vrácená data ovlivní různé role a jejich kombinace?

Dalším námětem je přizpůsobit EBIE API, aby lépe podporovala query parametry. Pokud má být API využívána i mimo portál, bylo by vhodné, aby podporovala filtrování výsledků pomocí parametrů za URL. Autorizace tuto variantu již podporuje a stačí tedy upravit fungování API jako takové. Bohužel je pravděpodobné, že se tímto dále zkomplikuje tvorba SQL dotazů. Tím spíš je vhodné vytvořit nástroj pro zjednodušení tvorby SQL dotazů.

Momentálně si EBIE API nejvíce zaslouží jasné určení konvencí a případné přepsání (refactor) některých částí.

9.3 Další práce na autorizaci

Byla snaha vytvořit flexibilní autorizační systém, který bude schopný reagovat na různé požadavky, včetně těch, které momentálně nejsou. Je však možné, že nějaký use-case unikl, nebo se objeví nový požadavek. V tom případě bude potřeba upravit přímo zdrojový kód.

V případě potřeby je možné rozšiřovat seznam organizačních jednotek i na neakademické součásti ČVUT. Pokud daná jednotka existuje na Usermap, neměl by být problém. Dokonce je možné vytvořit nové aliasy pro lepši čitelnost definic endpointů.

Závěr

Cílem této práce bylo vytvořit systém pro řízení přístupu k datům z datového skladu. Z tohoto zadání vyplynula potřeba vytvořit novou autorizační vrstvu pro aplikaci EBIE API, která poskytuje data z datového skladu do portálu EBIE. Díky vzniku této vrstvy je současně možné otevřít EBIE API více uživatelům nebo vývojářům klientských aplikací, kteří by chtěli přistupovat k datům přímo z EBIE API.

Nebylo možné zcela automatizovat definice autorizací a stále je přenechána část práce na vývojáři aplikace. Základ autorizační vrstvy je middleware, který kontroluje požadavky před jejich zpracováním samotnou aplikací. Tyto kontroly provádí na základě definic od vývojáře, který tak musí dodržovat pár základních konvencí.

Pro omezování výsledné množiny dat poskytuje práce doporučení a sadu pomocných metod, které je možno využít při vytváření nových SQL dotazů a implementaci jednotlivých endpointů.

Práce vznikla nad již existující implementaci EBIE API ve vlastním klonu repositáře. Jedná se tak o již připravenou implementaci k nasazení. Nebyly však převedeny všechny endpointy. Pro demonstraci práce s autorizační vrstvou byl převeden endpoint pro výsledky předmětů. Tento endpoint tak vrací pouze relevantní data podle vlastněných rolí.

Pro kontrolu funkcionalit vrstvy bylo vytvořeno množství automatických testů. Tyto testy je vhodné pouštět pokaždé, když se upraví nebo přidá funkcionalita aby bylo zajištěno, že jiné části nebyly ovlivněny.

Díky této práci jsem se opět o něco více seznámil s Ruby a hlavně s frameworkem Sinatra. I přes mnohé frustrující momenty při implementaci mě práce poměrně bavila a každý správně fungující kus kódu potěšil. S trochou štěstí bude i celá aplikace fungovat podle představ.

Reference

1. ČVUT. *DataWarehouse EBIE* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://ebie.is.cvut.cz.
2. ČERNÝ, Cyril. *Návrh a implementace integrace služeb do projektu EBIE*. 2016. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
3. HANSSON, David Heinemeier. *Ruby on Rails* [online]. 2018 [cit. 2018-03-14]. Dostupné z: http://rubyonrails.org/.
4. ČVUT. *OAuth 2.0 autorizační server Zuul* [online]. Ed. JIRŮTKA, Jakub. 2016 [cit. 2018-03-14]. Dostupné z: https://github.com/cvut/zuul-oaas.
5. HAJČIAR, David. *Návrh a implementace backend projektu EBIE*. 2016. Bakalářská práce. České vysoké učení technické v Praze Fakulta informačních technologií.
6. MIZERANY, Blake. *Sinatra* [online]. 2007 [cit. 2018-03-14]. Dostupné z: http://sinatrarb.com/.
7. MATSUMOTO, Yukihiro. *Ruby Programming Language* [online]. 1995 [cit. 2018-03-14]. Dostupné z: https://www.ruby-lang.org/en/.
8. MATSUMOTO, Yukihiro. *About Ruby* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://www.ruby-lang.org/en/about/.
9. NEUKIRCHEN, Christian. *Rack: a Ruby Webserver Interface* [online]. 2007 [cit. 2018-03-14]. Dostupné z: http://rack.github.io/.
10. NEUKIRCHEN, Christian. *Introducing Rack* [online]. 2007 [cit. 2018-03-14]. Dostupné z: http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html.
11. LEACH, Brandur; BELO, Pedro. *Pliny* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://github.com/interagent/pliny.

12. MATYS, Jan. *Analýza a vizualizace závislostí uživatelských rolí v IDM ČVUT*. 2015. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
13. ČVUT. *Usermap* [online]. 2018 [cit. 2018-03-17]. Dostupné z: https://usermap.cvut.cz/search.
14. ČVUT. *Usermap API* [online]. 2018 [cit. 2018-03-17]. Dostupné z: https://rozvoj.fit.cvut.cz/Main/usermap-api.
15. OLSZOWKA, Christoph. *TheRubyToolbox* [online]. 2009 [cit. 2018-03-14]. Dostupné z: https://www.ruby-toolbox.com.
16. VARVET. *Pundit* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://github.com/varvet/pundit.
17. CANCANCOMMUNITY. *CanCanCan* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://github.com/CanCanCommunity/cancancan.
18. KOCH, Henning. *Consul* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://github.com/makandra/consul.
19. CANCANCOMMUNITY. *Other Authorization Solutions* [online]. 2018 [cit. 2018-03-14]. Dostupné z: https://github.com/CanCanCommunity/cancancan/wiki/Other-Authorization-Solutions.
20. FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T. *Hypertext Transfer Protocol – HTTP/1.1* [online]. 1999 [cit. 2018-03-14]. Dostupné z: https://tools.ietf.org/html/rfc2616.
21. DUSSEAULT, L. *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)* [online]. 2007 [cit. 2018-03-14]. Dostupné z: https://tools.ietf.org/html/rfc4918.
22. HARDT, Dick. *The OAuth 2.0 Authorization Framework* [online]. 2012 [cit. 2018-03-14]. Dostupné z: https://tools.ietf.org/html/rfc6749.
23. ČVUT. *zuul-oaas wiki* [online]. Ed. JIRŮTKA, Jakub. 2016 [cit. 2018-03-14]. Dostupné z: https://github.com/cvut/zuul-oaas/wiki.
24. SINATRA. *Sinatra base class* [online]. 2007 [cit. 2018-03-14]. Dostupné z: https://github.com/sinatra/sinatra/blob/master/lib/sinatra/base.rb.
25. RSPEC. *Rspec* [online]. 2005 [cit. 2018-03-14]. Dostupné z: http://rspec.info/.
26. BLIMKE, Bartosz. *Webmock* [online]. 2009 [cit. 2018-03-14]. Dostupné z: https://github.com/bblimke/webmock.
27. OLSZOWKA, Christoph. *SimpleCov* [online]. 2010 [cit. 2018-03-14]. Dostupné z: https://github.com/colszowka/simplecov.

Příloha A

Seznam použitých zkratek

EBIE Extended business intelligence encyclopedia

REST Representional state tranfer

API Application programming interface

OAAS OAuth authorization server

URL Uniform resource locator

URI Uniform resource indetifier

DWH Data warehouse

ORM Object-relational mapping

DSL Domain specific language

Příloha B

Tipy při tvorbě endpointů

V této příloze bude stručný popis postupu při tvorbě endpointu. Postup při úpravě již existujících endpointů je velmi podobný. Je vhodné provést novou analýzu požadavků na endpoint a následně upravit existující SQL dotazy.

B.1 Analýza autorizačních požadavků

Během návrhu nového endpointu je nutné analyzovat oprávnění. **Nepodceňujte tento krok**. Analýza autorizačních požadavků by měla odpovědět na dotaz: **Kdo smí data číst a jak role ovlivní výsledek?**.

Je nutné se pro každý endpoint zamyslet, k čemu slouží a podle toho vyslovit omezení na data. Jiné požadavky budou mít endpointy pro získání seznamu fakult do před-filtru a pro vracení výsledků všech předmětů.

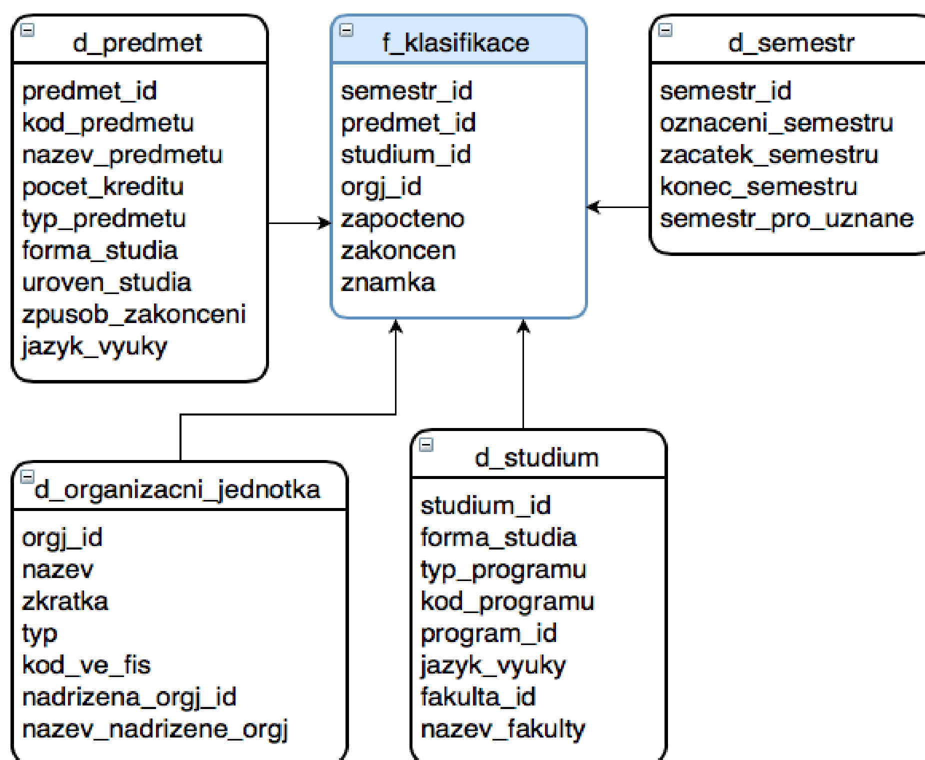
Řekněme, že chceme vytvořit endpoint pro získání výsledků předmětů vyučovaných na konkrétní fakultě a v konkrétním semestru. Autorizační požadavky mohou být následující:

1. Role katedry (vedoucí) smí číst pouze ty předměty, které jsou vyučovány na jeho katedře.
2. Roli fakulty (děkanovi) se vrátí seznam všech předmětů na fakultě.
3. Osoba, která není z požadované fakulty (ani z podřízené katedry), nemá nárok na žádná data.

Tím byla definována základní pravidla pro přístup.

B.2 Analýza datamartu

Následuje analýza schématu datamartu. Na obrázku B.1 je ukázka schéma datamartu pro výsledky předmětů. Zajímají nás sloupce `orgj_id`, `fakulta_id` a `katedra_id`. Je nutné určit **co vyjadřují**. Pro každý datamart se význam



Obrázek B.1: Schéma pro výsledky předmětů

může lišit, proto je vhodné si jasně určit, **jaký význam daný sloupec má** v konkrétním datamartu.

Pro výsledky předmětů taková analýza může vypadat následovně:

orgj_id je v tabulce `f_klasifikace`. Tato tabulka zaznamenává hodnocení studenta v předmětu. Víme, že zde budou identifikátory fakult a kateder. Sloupec nebude obsahovat pouze katedry, protože některé předměty mohou být organizovány přímo fakultou. Význam tedy je: **která jednotka zapsala klasifikaci**. Což se dá také vyložit jako: **která jednotka vyučuje daný předmět**

fakulta_id je v tabulce `d_studium`. Tato tabulka reprezentuje studium studenta na fakultě. Sloupec `fakulta_id` tedy znamená: **na jaké fakultě je student zapsán do studia**.

Sloupec `katedra_id` v ukázce není.

B.3 Definice endpointu

Při definici endpointu se snažíme delegovat co nejvíce práce na automatickou autorizaci. Pokud v parametrech endpointu je organizační jednotka, využijte toho a pojmenujte parametr správně. Autorizační vrstva reaguje na parametry **:faculty, :fakulta, :department, :katedra, :faculty_strict, :fakulta_strict**. Pokud se tedy jedná o endpoint, který přijímá v URL identifikátor fakulty, použijte jedno z uvedených jmen pro fakultu. Striktní kontrola znamená, že uživatel musí mít přímo roli dané fakulty a člen podřízené katedry není puštěn.

V našem příkladu pro získání výsledků předmětů určité fakultě bude definice endpointu `/semestr/:semestr/fakulta/:fakulta`. Automatická autorizace tak bude kontrolovat náležitost k fakultě. Tím je splněn třetí požadavek z analýzy požadavků. Další požadavky musíme splnit ručně.

B.4 SQL dotaz

Doporučuji nevytvářet extra univerzální dotazy, které jsou schopné vracet více druhů výsledků. Vždy vytvářejte pro každý endpoint vlastní SQL dotaz. Pokud uvidíte, že jsou téměř identické, můžete je spojit a parametrizovat. Při pokusech vytvořit hned univerzální dotaz máte velkou pravděpodobnost, že se do toho zamotáte a uděláte chybu.

Při vytváření dotazu se dívejte, kde používáte tabulky obsahující důležité sloupce (fakulta, katedra a organizační jednotka). V těchto místech bude potřeba přidat filtr tak, aby se vrátila správná data podle požadavků z analýzy.

Obecné pravidlo je: **V každém SQL dotazu by měl být filtr na katedru a/nebo fakultu**. Pokud tam není, musí to být vědomé rozhodnutí, které vyplývá z charakteru vracených dat.

Struktura SQL dotazu pro náš vzorový endpoint je na ukázce 29. Všimněte

```sql
SELECT ...
FROM (
  SELECT ...
  FROM (SELECT ... FROM d_studium
        WHERE fakulta_id = 'xxx') studium
  JOIN (SELECT ... FROM f_klasifikace
        WHERE orgj_id = 'xxx') klasifikace
  ON klasifikace.studium_id = studium.studium_id
  GROUP BY predmet_id, semestr_id, fakulta_id) pocty
JOIN d_predmet predmet
ON pocty.predmet_id = predmet.predmet_id
```

Ukázka kódu 29: Kostra SQL pro výsledky předmětů

si `WHERE` u obou důležitých sloupců. Nyní si říkáte: „Super, v URL parametru je fakulta, takže ji rovnou použiji k filtrování v tabulce d_studium“. Nikoliv. Chceme výsledky předmětů **vyučovaných na fakultě**. Sloupec fakulta ale znamená, **z jaké fakulty je student**. Předmět může studovat i student jiné fakulty a **proto v našem případě sloupec s fakultou není vhodný** pro filtrování. Z výsledného dotazu tento filtr zmizí. **Velmi pozorně se dívejte na význam dat, se kterými pracujete!**.

Pro získání předmětů fakulty je tedy nutné použít `orgj_id` z klasifikace. Protože může obsahovat identifikátory kateder i fakult, není možná podmínka tvaru `WHERE orgj_id = 'xxx'`. Podívejme se na význam sloupce `orgj_id`. **Jednotka, která zapsala známku v předmětu**. Fakulta si zapisuje pouze „své“ předměty. Většina předmětů je organizována (a zapisována) katedrami. Musí zde proto být **seznam všech organizačních jednotek pod fakultou včetně fakulty samotné**. Filtr tak bude mít tvar `WHERE orgj_id IN (...)`.

Protože není triviální z identifikátoru fakulty získat seznam všech jejích kateder, jsou dostupné pomocné metody pro generování filtrů. Máte k dispozici několik metod, které jsou popsány v dokumentaci a v této práci. V tomto případě je vhodné použít `where_department_statement`, která vrátí správný filtr. Zavoláním metody s parametry `attr_name: "orgj_id"`, `direct: false` a `faculty_id: url_params["fakulta"]`, je vytvořen `WHERE` filtr na organizační jednotky, který bude obsahovat požadované identifikátory jednotek. Více o úpravě chování metod pomocí parametrů je v této práci a dokumentaci projektu.

Na závěr pár tipů:

- Význam dat je důležitý. Je nutné se zamyslet, zda filtrování přes daný parametr dává smysl a opravdu vrací to, co chceme.
- Mějte jasně určeno, jaká data jsou od endpointu očekávána. Při mírně odlišné specifikaci v našem příkladu bychom najednou mohli/museli filtrovat přes fakultu. Například vracení výsledků pouze studentů konkrétní fakulty. Nebo výsledky studentů jedné fakulty v předmětech jiných fakult. Proto mějte už od začátku jasno, co chcete
- Pokud máte z URL parametru organizační jednotku, většinou je vhodné ji předat pomocné metodě. Výsledek se náležitě upraví tak, aby obsahoval pouze danou jednotku, nebo její součásti.
- Pro získání všech souvisejících jednotek použijte `direct: false`. V případě seznamu fakult se vrátí všechny fakulty, kterými je uživatel členem (tedy i pokud je pouze členem podřízené katedry). U seznamu kateder se „rozbalí“ povolené fakulty na všechny jejich katedry.
- Data z autorizační vrstvy nemusí být nutně použita k autorizačním účelům. Jsou to informace, které je možné použít i jinak.

Příloha C

Obsah přiloženého CD

```
├── readme.txt ............................ stručný popis obsahu CD
├── src
│   ├── ebie-api .......................... zdrojové kódy celé aplikace
│   │   ├── coverage ...................... reporty pokryti kódu
│   │   ├── lib
│   │   │   └── authorization ............. vlastní implementace autorizace
│   │   └── spec
│   │       └── authorization ............. testy autorizace
│   └── thesis ............................ zdrojová forma práce ve formátu LaTeX
└── text .................................. text práce
    └── thesis.pdf ........................ text práce ve formátu PDF
```