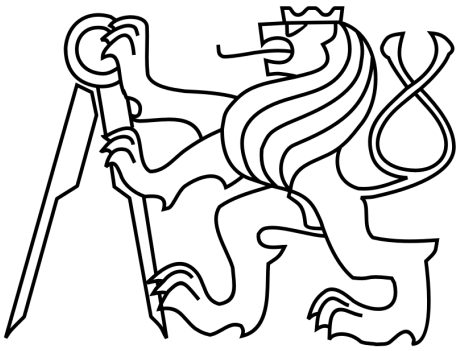
České vysoké učení technické v Praze

Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE



Martin Egermajer

Zabezpečení automaticky generovaných REST služeb

Katedra počítačové grafiky a interakce

Vedoucí diplomové práce: Ing. Tomáš Černý, MSc.

Studijní program: Otevřená informatika

Studijní obor: Softwarové inženýrství

Praha 2015

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že ČVUT v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Abstrakt:

Cílem této práce je návrh a implementace zabezpečovací knihovny pro automaticky generované REST služby. Vytvořil jsem procesor, který na základě modelu rozhodne, zda má přihlášený uživatel právo s daty pracovat (podle rolí a vlastnictví). Vytvořené řešení je snadno rozšiřitelné o další způsoby zabezpečení.

Klíčová slova: REST, zabezpečení, autentikace, autorizace

Abstract:

The goal of this thesis is to design and implement security library, that will provide authorization service for generated REST web services. I developed a main processor, which decides if the crud operation of the currently logged user is authorized or unauthorized (due to roles and ownership permissions). This solution can be easily extended by another user-created authorization mechanism.

Keywords: REST, Security, Authentication, Authorization

Obsah

1 Úvod **5**

2 Analýza **6**

2.1 REST 6

2.1.1 Omezení REST 6

2.2 Autentizace vs autorizace 7

2.3 Autentizace 8

2.3.1 Antentizační nástroje pro JavaEE 8

2.3.2 Další architektonická doporučení pro autentizaci 9

2.3.3 Ochrana před některými typy útoků 9

2.4 Autorizace 10

3 Framework AspectFaces **12**

3.1 Architektura 12

3.2 Konfigurace 13

3.3 Metadata 15

3.4 Standardní vs. distribuované uživatelské rozhraní 16

3.5 Dotazovací jazyk pro distribuované uživatelské rozhraní 17

4 Návrh a implementace vlastního řešení **18**

4.1 Architektura 18

4.1.1 Implementované procesory 18

4.2 Popis jazyka 22

4.3 Rozšiřitelnost 22

4.4 Limity 23

5 Vzorová aplikace **24**

5.1 Popis 24

5.2 Aplikace AspectFaces na vzorovou aplikaci 25

5.3 Aplikace zabezpečení na vzorovou aplikaci 27

5.3.1 Úprava controlleru 27

5.3.2 Úprava javascriptu 27

5.3.3 Úprava modelu 28

5.3.4 Příklady funkcionality 31

6 Testování **33**

6.1 Unit testy 33

6.2 Testy výkonnosti 34

6.2.1 Prostředí 34

6.2.2 Samotná vzorová aplikace 35

6.2.3 Po aplikaci aspectFaces 35

6.2.4 Po aplikaci zabezpečení 35

6.2.5 Porovnání výsledků 36

7 Závěr **38**

7.1 Shrnutí 38

7.2 Výhody řešení 38

7.3 Nevýhody řešení 38

7.4 Práce do budoucna 39

Literatura **40**

Seznam použitých zkratek **41**

Přílohy **42**

Seznam obrázků

3.1 Ukázka životního cyklu frameworku 13

4.1 Diagram jednotlivých procesorů 21
4.2 Struktura hlavního procesoru 21

5.1 Původní databázové schéma Seam Booking 25
5.2 Ukázka původního zobrazení hotelů v Seam Booking 25
5.3 Ukázka formuláře generovaného AspectFaces - Hotel 27
5.4 Databázové schéma po aplikaci zabezpečení Seam Booking 28
5.5 Rozhraní User, Role a SecuredEntity 28
5.6 Rozdíl mezi administrátorem a normálním uživatelem - zabezpečení třídy 31
5.7 Rozdíl mezi administrátorem a normálním uživatelem - zabezpečení atributu 32

6.1 Struktura unit testů 34
6.2 Ukázka stahování js a css souborů 37

Seznam tabulek

6.1 Časy zobrazení v ms 36

1 Úvod

V dnešní době je těžké nenarazit při svých každodenních činnostech na některou RESTful službu, i když o tom vůbec nemusíme vědět. Stačí, když si v mobilním telefonu vytvoříme úkol na nadcházející den, a telefon zavolá právě takovou RESTful službu proto, aby tento záznam uložil na server do databáze. To znamená, že někde na webu existuje controller, který obsluhuje požadavky na data zdroje úkol.

Pokud bychom ale chtěli konkrétní úkoly přiřadit ke konkrétnímu uživateli, musíme do databáze přidat tabulku uživatel, a vazbou M:N ji propojíme s tabulkou úkol. Pak musíme přidat další funkcionalitu, která bude obsluhovat požadavky na data zdroje uživatel. Cílem automaticky generovaných RESTful služeb je, aby se s rozšířením modelu nemusela rozšiřovat i funkcionalita ostatních částí webové aplikace, ale aby se změnil opravdu jen model. Navíc, pokud použijeme řešení, které umí nad takovouto službou automaticky generovat uživatelské rozhraní, usnadníme si tím i práci při vývoji frontendové části aplikace. Pokud se pak například stane, že jednu třídu modelu rozšíříme o nový atribut, změna se nám automaticky projeví i v uživatelském rozhraní, aniž bychom v něm museli cokoliv upravit.

Cílem mé práce je zaměřit se na tu část automaticky generované RESTful služby, která se stará o zabezpečení, hlavně o část autorizační. To znamená, že budu řešit otázku zabezpečení jednotlivých záznamů v databázi před uživateli, kteří nemají právo s těmito záznamy pracovat, a to i na úrovni jednotlivých sloupců.

2 Analýza

2.1 REST

REST (Representational state transfer) je architektonický styl, který představil Roy Thomas Fielding ve své dizertační práci[4]. Je to architektonický styl složený z několika síťově orientovaný architektonických stylů, s několika omezeními navíc, které definují jednotné rozhraní.

2.1.1 Omezení REST

Klient-server

Rozdělení na klienta a server. Výhodou tohoto oddělení je to, že se klient nemusí starat o čistě serverové činnosti a naopak. Např. klient se nemusí starat o ukládaní dat, server se zase nemusí zabývat uživatelským rozhraním. Klient i server navíc mohou být vyvíjeni nezávisle na sobě, protože se oba vyvíjí proti jednotnému rozhraní. Spojení funguje vždy tak, že klient kontaktuje server s požadavkem, a server na něj buď odpoví, nebo ho zamítne.

Bezestavovost

Na straně serveru se neuchovává žadná informace o stavu klienta. To znamená, že každý požadavek v sobě musí obsahovat všechny informace potřebné k úspěšnému zpracování, bez ohledu na jakékoliv předcházející požadavky. Na straně serveru tím dojde k ušetření zdrojů, protože po každém dotazu může server informace o klientovi zahodit. Na druhou stranu se posíláním všech potřebných informací s každým dotazem zvyšuje zátěž na síti.

Použití cache

Toto omezení požaduje, aby byla v každé odpovědi ze serveru informace o tom, zda klient může nebo nemůže daný dotaz cachovat. Klient pak nemusí pokaždé posílat dotaz na server, ale může si vzít data z vlastní cache. Tím se sníží zátěž, která je kladena na síť.

Vrstvený system

Komponenty, ze kterých se systém skládá, je na sebe možné komponenty navazovat. Komponenty v jedné vrstvě ale mohou komunikovat pouze s komponentami v další nejbližší vrstvě.

Jednotné rozhraní

Jedná se o nejdůležitější požadavek REST architektury. Tento požadavek Fielding[4] definuje pomocí čtyř omezení:

- **Identifikace zdroje** - jednotlivé zdroje musí být identifikovány, tento identifikátor pak musí být v požadavku klienta
- **Manipulace se zdrojem** - Pokud uživatel získal přístup ke zdroji, může na něm provádět CRUD operace (pokud na to má oprávnění)
- **Samopopisující zprávy** - Každý požadavek, který přijde, v sobě musí obsahovat všechny informace, které jsou potřeba pro úspěšné splnění požadavku
- **Hypermedia**

Code-On-Demand

Toto nepovinné omezení umožňuje klientům rozšířit si vlastní funkcionalitu tím, že si stáhnou kód ve formě apletu nebo skriptu a následně ho vykonají.

2.2 Autentizace vs autorizace

Zabezpečení REST služeb se skládá ze dvou hlavních částí, a to autentizace a autorizace. Autentizace je proces ověření identity uživatele. Autorizace obvykle

následuje po autentizaci. Je to proces, který ověří, že daný autentizovaný uživatel má oprávnění k akci, kterou chce uskutečnit, například smazat záznam v databázi.

2.3 Autentizace

Na začátku tohoto procesu je neznámý uživatel, který se potřebuje autentizovat. Na konci by měl být přihlášený uživatel, u kterého je ověřena jeho identita.

Cílem by mělo být[5]:

- **Provázat systémovou identitu s uživatelem pomocí přihlašovacích údajů.** To znamená, že pro každého uživatele musíme mít systémovou identitu. Propojení uživatele a systémové identity se pak provede podle přihlašovacích údajů.
- **Poskytnout rozumné autentizační zabezpečení s ohledem na důležitost aplikace.** Není nutné utrácet obrovské peníze za dokonalé zabezpečení nějaké chatovací služby, na druhou stranu je nežádoucí používat basic autentizaci pro přihlášení do internetového bankovnictví. Musí se zvolit zabezpečení, které odpovídá důležitosti aplikace.
- **Odepření přístupu útočníkům.**

2.3.1 Antentizační nástroje pro JavaEE

Díky tomu, že je Java jedním z nejpoužívanějších programovacích jazyků[7], existuje i mnoho již existujících řešení zabývajících se autentizací. Pokud nepotřebujeme použít nějaký neobvyklý typ autentizace, který není v žádném existujícím řešení implementovaný, je lepší použít již existující řešení, které je používané komunitou a je pravidelně aktualizované. Tam není taková pravděpodobnost existence chyby, než kdybychom si vyvíjeli vlastní autentizační nástroj.

Příkladem může být framework Spring security. Jedná se o velmi rozšířený open-source framework (pro zajímavost má okolo 900 "fork" na GitHub), který je zároveň aktivní (poslední commit do repozitáře byl pár dní zpátky). Dle dokumentace tento framework podporuje i velké množství typů autentizace, například: Basic, Digest, LDAP, SSO.

2.3.2 Další architektonická doporučení pro autentizaci

Dle serveru owasp[5] by se měl architektonický návrh aplikace držet následujících doporučení (uvádím pouze zde pouze výběr):

- Všechny chráněné funkce a zdroje by měly být chráněny jedním společným autentizačním mechanizmem.
- Přihlašovací údaje musí být přenášeny v zabezpečené formě. To znamená, že by se neměly přenášet v plaintext formě, ale buď by měly být nejdříve u klienta zašifrovány (např. javascriptem), nebo je třeba použít ke komunikaci zabezpečený protokol (např. HTTPS).
- Přihlašovací údaje se v databázi musí ochránit použitím hashovací funkce a soli.
- Při špatném přihlášení by aplikace neměla sdělovat, za je špatné přihlašovací jméno nebo heslo. Útočník této informace pak může využít k tomu, že už bude mít potvrzené přihlašovací jméno a bude zkoušet jen prolomit heslo.
- Pro důležitější operace je vhodné použít mechanizmus podepsání transakce. To znamená že k potvrzení akce se kromě přihlašovacích údajů přidá ještě ověření pomocí nějakého uživatelem vlastněného zařízení. Může to být například ověření pomocí zaslaného SMS kódu, nebo pro potřeby vysokého zabezpečení to může být použití podepisovacího zařízení. Je to kalkulačka, která generuje přihlašovací token. Uživatel si touto metodou může ověřit, že opravdu komunikuje s bankovním serverem, protože server pro kontrolu vygeneruje uživateli zpět druhý token, který si uživatel může ověřit ve své kalkulačce.

2.3.3 Ochrana před některými typy útoků

Predikce session

Ve chvíli, kdy se na serveru vytvoří session, zapíše se ke klientovi cookie s identifikátorem session (session id). S každým dotazem klienta pak server podle hodnoty této cookie pozná, která session patří k danému klientovi. Pokud je generování

session id předvídatelné a zároveň útočník ví, jak se jmenuje cookie se session id, může se pokusit uhádnout session id a dostat se přes autentizační mechanizmus. Např. v hlavičce dotazu pošle cookie JSESSIONID=uzivatel1 (JSESSIONID je cookie běžně používaná Java enterprise aplikacemi), a pokud je uživatel uzivatel1 přihlášený, může se útok podařit. Řešením je používat co nejméně predikovatelný algoritmus na generování session id.

Odchycení session

Útočník může například na veřejně přístupné síti odchytávat okolní síťovou komunikaci, a pokud není komunikace řádně šifrována, může útočník z dotazů ostatních uživatelů odchytit session id. Poté pod touto session může pracovat na serveru.

Main-in-the-middle

V tomto případě útok spočívá v tom, že klient nekomunikuje přímo s cílovým serverem, ale komunikuje přes útočníka. Útočník pak může číst komunikaci klienta s cílovým serverem a zároveň může tuto komunikaci měnit (klient pošle požadavek na zaplacení 20Kč, ale útočník požadavek změní a na server dorazí požadavek na zaplacení 20 000Kč na útočníkovo číslo účtu). Proti tomuto útoku nepomůže ani komunikace chráněná SSL, protože klient otevře SSL připojení k útočníkovi, a útočník otevře jiné SSL připojení k serveru. Možnosti ochrany jsou například použití další vrstvy šifrování pomocí tajných klíčů, ověřování SSL certifikátu u důvěryhodné autority, nebo kontrola pomocí veřejných klíčů.

2.4 Autorizace

Jedním z nejběžnějších způsobů autorizace je definice oprávnění u metod controllerů. To je v pořádku v případě, že existuje pro každou třídu modelu jeden controller, který obsahuje metody pro všechny CRUD operace na této konkrétní třídě. Tento přístup ale nemůžeme použít v našem případě, kdy budeme mít jeden univerzální controller, který bude zpracovávat CRUD operace pro více modelových tříd. Definování přístupů tedy bude muset být definováno jinde než v controlleru.

Listing 2.1: Ukázka autorizace podle rolí (spring security - přístup na stránku)

```
1 @Secured("ROLE_ADMIN")
2 @RequestMapping(value="/admin", method = RequestMethod.GET
      )
3 public String admin(ModelMap model){
4 return "admin_page";
5 }
```

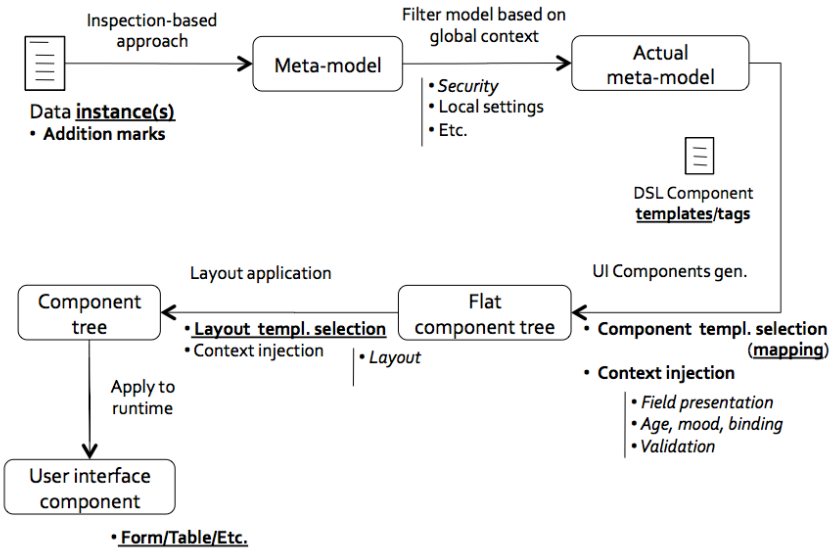
3 Framework AspectFaces

Framework aspect faces je framework, který usnadňuje vývoj frontendové části webových aplikací - uživatelského rozhraní. Poskytuje podporu pro automatické generování UI podle kontextu, automatické použití validací, umožňuje použití vlastních rozložení (layout) a mnoho dalšího.

3.1 Architektura

Vše, co musíme udělat pro zobrazení dat na UI, je předat frameworku instanci, kterou chceme vykreslit[6]. Proces, kterým framework projde před vykreslením dat:

- Po předání instance si framework vytvoří pomocí inspektoru metamodel dané třídy, ten pak ještě upraví na základě globálního kontextu
- Následuje výběr šablon pro jednotlivé atributy podle aktuálního kontextu
- Poté se vybere rozložení (layout), podle kterého se bude UI vytvářet
- Následuje zobrazení dat uživateli na UI



Obrázek 3.1: Ukázka životního cyklu frameworku

3.2 Konfigurace

V hlavním konfiguračním souboru aspectfaces-config.xml se registrují všechny druhy anotací (pro použití framworku, validací, jpa apod.) a mapovací soubory. V mapovacím souboru se uvádí přiřazení jednotlivých datových typů šablonám, které ho budou vykreslovat na UI. Dá se zde použít i podmíněné mapování, kdy se šablona vybírá ještě podle zadaného výrazu. Zde je příklad části mapovacího souboru, včetně podmíněných mapování:

Listing 3.1: Ukázka mapovacího souboru

```
1 <mapping>
2 <type>String</type>
3 <default tag="inputTextTag" maxLength="255" size="30"
    required="false"/>
4 <condition expression="${not empty email and email == true
    }" tag="inputTextTag"/>
5 <condition expression="${not empty password and password
    == true}" tag="inputTextTag"/>
6 </mapping>
```

Příklad šablony pro číslo (number.xhtml):

Listing 3.2: šablona pro number (tag)

```
1  <ui:param name="id" value="\#{prefix}$field$"/>
2  <ui:param name="label"
3           value="\#{text['$entityBean.shortClassName().
               firstToLower()$.$field$']}"/>
4  <ui:param name="value"
5           value="\#{$entityBean.shortClassName()$.$field$}"/
               >
6  <ui:param name="rendered"
7           value="\#{empty render$field.firstToUpper()$ ? '
               true' : render$field.firstToUpper()$}"/>
8  <ui:param name="required" value="\#{empty required$field.
       firstToUpper()$ ? $required$ : required$field.
       firstToUpper()$}"/>
9  <ui:param name="minValue" value="$min$"/>
10 <ui:param name="maxValue" value="$max$"/>
11
12 <ui:define name="input">
13 <h:inputText
14 rendered="\#{rendered}"
15 required="\#{required}"
16 id="\#{id}"
17 title="\#{label}"
18 value="\#{value}">
19 <f:convertNumber/>
20 <f:validateLongRange minimum="\#{empty minValue ? '0' :
       minValue}" maximum="\#{empty maxValue ? '10000000000' :
        maxValue}"/>
21 </h:inputText>
22 </ui:define>
23
24 <ui:define name="output">
25 <h:outputText value="\#{value}"/>
26 </ui:define>
```

Framwork umožňuje mít nastaveno více mapovacích souborů pro různé kontexty. Konfigurace se pak vybírá podle jejího jména:

Listing 3.3: Ukázka různých konfiguraci pro různé kontexty

```
1 <configurations-registration>
2 <configuration name="default" path="/WEB-INF/af/html.
    config.xml" check-modification="true" lazy-load="true"/
    >
3 <configuration name="table" path="/WEB-INF/af/table.config
    .xml" check-modification="true" lazy-load="true"/>
4 </configurations-registration>
```

Listing 3.4: získání konfigurace

```
1 Configuration configuration = ConfigurationStorage.
    getInstance().getConfiguration("default");
```

3.3 Metadata

Pro každou třídu modelu se vytváří tzv. metamodel této třídy. Obsahuje informace o samotné třídě:

- její jméno
- seznam atributů, které se mají na frontendu zobrazit
- Každý atribut obsahuje tyto informace:
  - jméno
  - návratový typ
  - šablona, která se má použít při vykreslení item proměnné, které se uplatní při vytváření tagu z šablony (např. viditelnost, povinnost, maximální délku u čísla)

Metadata jsou potřebná pro to, aby mohl framework pracovat obecně se všemi třídami modelu aniž by věděl, jakou konkrétní třídu aktuálně zpracovává.

3.4 Standardní vs. distribuované uživatelské rozhraní

AspectFaces se může použít buď ve standardní verzi, kde se používají jsp tagy, nebo v distribuované verzi, kde se k modelu přistupuje javascriptem. Javascript ajaxem přes univerzální REST controller provádí CRUD operace.

V distribuované verzi nejsou šablony pro jednotlivé typy vstupních polí definované v xhtml souborech, ale v samotném javascriptu, zde je například vidět vstupní pole pro číslo:

Listing 3.5: javascriptová šablona pro čislo

```
1 Templates.numberTag = Templates.prefixTag
2 + "+'<input id=\"'+vars['prefix']+vars['name']+'\""
3 + " required=\"'+ifNull(vars['required'],false)+'\""
4 + " value=\"'+vars['value']+'\""
5 + " name=\"'+vars['prefix']+vars['name']+'\""
6 + " type=\"number\""
7 + " maxlength=\"'+vars['maxLength']+'\""
8 + " size=\"'+vars['size']+'\">' "
9 + Templates.suffixTag;
```

Javascriptová šablona je oproti jsp tagu celkově hůře čitelná a špatně se upravuje, hlavně díky mnoha uvozovkám, které se v šabloně nachází. Na druhou stranu se v reálu úprava těchto šablon nebude dít často, šablona se nadefinuje jednou, a pak už by se neměla výrazněji měnit.

Velká výhoda distribuovaného uživatelského rozhraní je ta, že je vystavena jedna obecná REST služba, na kterou se nemusí dotazovat jen jeden typ uživatelského rozhraní (webové), ale na tu samou službu se může dotazovat například mobilní aplikace, desktopová aplikace, jiná webová služba.

Ve své práci pracuji právě s verzí, kde se používá distribuované UI, protože právě na tuto REST službu budu aplikovat svou zabezpečovací knihovnu.

3.5 Dotazovací jazyk pro distribuované uživatelské rozhraní

Javascript se dotazuje Controlleru pomocí jednoduchého jazyka. V dotazu posílá jméno třídy, se kterou chce klient pracovat. Do jména třídy se může přidat pomocí znaku '@' i atribut. Dotaz na server pak může vypadat takto:

- GET /some.package.SomeClass@someAttribute/13

Tento dotaz načte z databáze instanci třídy SomeClass s id = 13, a vrátí hodnotu atributu someAttribute této instance. Takto klient získá data, která chce zobrazit na UI. K datům ale musí získat i metadata, aby věděl, jak má záznam zobrazit. Na metadata se klient zeptá takto:

- GET /some.package.SomeClass@someAttribute

Server v tomto případě nenačítá nic z databáze, ale načte si třídu SomeClass, získá Field someAttribute a vrátí jeho metadata. Poté, co klient získá jak data, tak metadata, může obojí využít k zobrazení dat.

4 Návrh a implementace vlastního řešení

4.1 Architektura

Protože způsobů, podle kterých omezovat/povolovat přístup k datům, je nepřeberné množství a všechny bych nedokázal pokrýt, musel jsem své řešení udělat jako snadno rozšiřitelné podle potřeb uživatele. Implementoval jsem tedy zastřešující třídu, která zajišťuje běh jednotlivých autorizačních procesorů (Procesor). Tato třída pro všechny CRUD operace zajišťuje kontrolu oprávnění uživatele a případné provedení/neprovedení operace, nebo její částečné omezení. Částečné omezení znamená, že má uživatel právo na operaci s danou entitou, ale nemá oprávnění tuto operaci provádět se všemi jejími atributy.

4.1.1 Implementované procesory

Ve svém řešení jsem implementoval dva autorizační procesory, a to podle uživatelských rolí (RoleRestrictProcessor), a podle vlastnictví záznamu (OwnerRestrictProcessor). Oba tyto procesory jsou řízené anotacemi.

RoleRestrictProcessor

Tento procesor omezuje přístup na základě uživatelských rolí. Využívá anotaci RoleRestrict, která definuje allow/deny role pro všechny CRUD operace.

Listing 4.1: Anotace RoleRestrict

```
1  @Target(value = {ElementType.TYPE, ElementType.FIELD})
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface RoleRestrict {
4  String[] allowCreate() default {};
5  String[] denyCreate() default {};
6  String[] allowRead() default {};
7  String[] denyRead() default {};
8  String[] allowUpdate() default {};
9  String[] denyUpdate() default {};
10 String[] allowDelete() default {};
11 String[] denyDelete() default {};
12 int priority() default 1;
13 }
```

Pokud není u třídy nebo u atributu uvedena anotace @RoleRestrict, vrací procesor automaticky příznak ALLOW s prioritou Integer.MIN_VALUE, takže operaci povolí, ale jen s nejnižší možnou prioritou. Stejně se procesor chová, když sice anotace existuje, ale pro danou operaci je pole allow i deny prázdné. Zde je příklad:

Listing 4.2: Přiklad aplikace anotace RoleRestrict

```
1  @RoleRestrict(
2  allowRead = {"REGISTERED\_USER", "ADMIN"},
3  allowUpdate = {"ADMIN"},
4  allowDelete = {"ADMIN"}
5  )
6  public class SomeClass implements SecuredEntity
```

V tomto případě může:

- uživatel s rolí REGISTERED_USER nebo ADMIN číst záznamy (s prioritou 1 - defaultní priorita)
- kdokoliv vytvářet záznam (s prioritou Integer.MIN_VALUE)
- uživatel s rolí ADMIN upravovat záznamy (s prioritou 1)
- uživatel s rolí ADMIN mazat záznamy (s prioritou 1)

Stejně to platí i pro atributy.

OwnerRestrictProcessor

Tento procesor omezuje přístup na základě vlastnictví daného záznamu (každý záznam má uvedeno, jaký uživatel ho vlastní). Využívá anotaci OwnerRestrict, která definuje operace, které může provádět vlastník záznamu.

Listing 4.3: Anotace OwnerRestrict

```
1 @Target(value = {ElementType.TYPE, ElementType.FIELD})
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface OwnerRestrict {
4 OperationType[] allowed() default {};
5 int priority() default 1;
6 }
```

Pokud není u třídy nebo u atributu uvedena anotace @OwnerRestrict, vrací procesor automaticky příznak ALLOW s prioritou Integer.MIN_VALUE, stejně jako u RoleRestrict procesoru. Pokud je anotace přítomna, procesor vrátí příznak ALLOW jen v případě, že je uživatel vlastník daného záznamu a zároveň operace, kterou požaduje, je uvedena v anotaci. Zde je příklad:

Listing 4.4: Použiti anotace OwnerRestrict

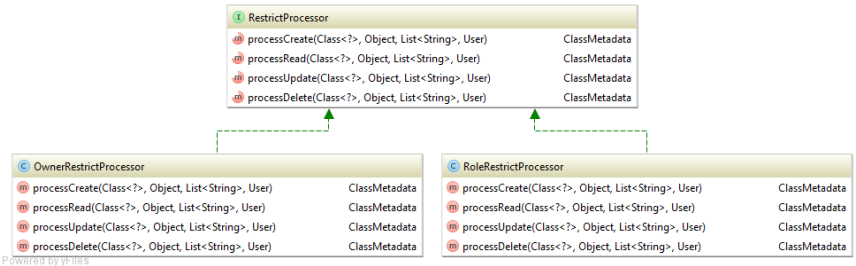
```
1 @OwnerRestrict(
2 allowed = {OperationType.CREATE, OperationType.READ,
      OperationType.UPDATE}
3 )
4 public class Booking implements SecuredEntity
```

Pokud je uživatel vlastník záznamu, může záznam vytvořit, číst a upravovat, nesmí ho jen smazat.

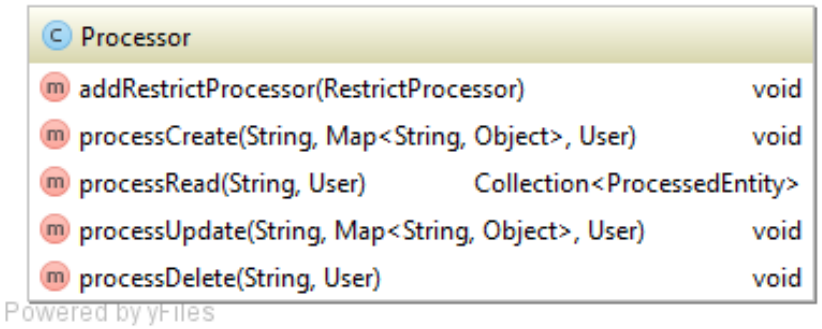
Práce hlavního procesoru

Pokud přijde hlavnímu procesoru požadavek na operaci se záznamy, zavolá hlavní procesor jednotlivé procesory, které mu vrátí příznak allow/deny. Pokud se sejdou různé příznaky z různých procesorů, rozhodne hlavní procesor buď podle vrácené priority, nebo upřednostní allow před deny v případě, že jsou priority stejné. Po zavolání všech zaregistrovaných procesorů hlavní procesor buď provede operaci, nebo vyhodí výjimku UnauthorizedException. Pokud se rozhodne provést operaci, tak:

- **čtení** procesor načte z databáze záznam, atributy, které uživatel nemá právo číst nastaví na null
- **úprava** procesor načte z databáze starý záznam, všechny atributy, které má uživatel právo upravovat, nastaví na novou hodnotu, ostatní ponechá podle starých hodnot. Pak záznam uloží
- **vytvoření** procesor vytvoří nový záznam, hodnoty, které uživatel může vytvářet, nastaví dle nových hodnot, ostatní nastaví na null
- **smazání** procesor smaže záznam z databáze



Obrázek 4.1: Diagram jednotlivých procesorů



Obrázek 4.2: Struktura hlavního procesoru

4.2 Popis jazyka

Jazyk, který přijímá hlavní procesor jsem původně převzal z jazyku AspectFaces, musel jsem ale provést některé úpravy. Vstupní queryString může obsahovat:

- **@** - oddělovač atributů
- # - id entity

Vstupní queryString tedy může vypadat například takto:

some.package.SomeClass#13@someAttribute@someAnotherAttribute

Pokud chceme s tímto queryStringem provést například operaci *update*, postupuje procesor takto:

- **operace *read*** - procesor rozdělí queryString podle @, a provádí operaci *read* od prvního do předposledního prvku. Pokud se někdy v této fázi stane, že uživatel nemá oprávnění k operaci *read*, tak celý proces končí zamítnutím operace. Uživatel tedy musí mít právo na čtení pro všechny prvky queryStringu
- **provedení samotné operace** - pokud má uživatel právo na čtení celého queryStringu, procesor podle autorizace zajistí nebo odmítne danou operaci

4.3 Rozšiřitelnost

Každý autorizační proces musí implementovat rozhraní RestrictProcessor, které definuje metodu pro každou CRUD operaci. Každá z těchto metod vrací informaci o tom, zda má uživatel oprávnění operaci provést, v jakém rozsahu, a s jakou prioritou. Hlavní procesor pak tyto informace podle priority sbírá a určí, zda se může operace provést, nebo na to uživatel nemá oprávnění.

Konrétně musí každý procesor vracet instanci třídy ClassMetadata, která obsahuje informace:

- **restrictType** - atribut obsahující ALLOW/DENY pro operaci na daném záznamu
- **priority** - priorita vztahující se k restrictType

- **attributes** - seznam metadat o atributech, každý obsahuje:
  - **name** - jméno atributu
  - **restrictType** - ALLOW/DENY pro operaci na daném atributu
  - **priority** - priorita vztahující se k restrictType

Pokud by uživatel potřeboval rozšířit funkčnost hlavního procesoru o další typ ochrany, může si u procesoru zaregistrovat vlastní implementaci. Tato třída musí implementovat rozhraní RestrictProcessor. Poté zavoláním metody addRestrictProcessor může zaregistrovat svůj procesor. Tím se přidá mezi ostatní procesory a hlavní procesor ho začne používat.

Listing 4.5: Rozhraní RestrictProcessor

```
1 public interface RestrictProcessor {
2 ClassMetadata processCreate(Class<?> clazz, Object
     instance, List<String> attributes, User currentUser);
3 ClassMetadata processRead(Class<?> clazz, Object instance,
      List<String> attributes, User currentUser);
4 ClassMetadata processUpdate(Class<?> clazz, Object
     instance, List<String> attributes, User currentUser);
5 ClassMetadata processDelete(Class<?> clazz, Object
     instance, List<String> attributes, User currentUser);
6 }
```

4.4 Limity

Hlavní procesor zpracovává výsledky z jednotlivých procesorů relací OR, to znamená, že umí povolit operaci v případě, že je uživatel vlastník záznamu, nebo má roli ADMIN. Neumí ale spojit výsledky tak, aby se dalo zajistit, že záznam může upravovat jen vlastník, který má navíc roli ADMIN. Na to by bylo třeba postavit ještě spojovací vrstvu nad jednotlivými procesory, a definovat její vlastní jazyk, podle kterého by spojovala výsledky z jednotlivých procesorů.

Další omezení představuje datový typ, který je použitý pro ID. Současné řešení umí pracovat pouze s datovým typem Long. Pokud má být entita součástí automaticky generované REST služby, musí používat jako primární klíč Long. Toto by se dalo vyřešit parametrizací primárního klíče, nicméně by to byl celkem velký zásah jak do funkcionality zabezpečovací knihovny, tak do modelu. Zatím tedy nechávám primární klíč typu Long jako povinný.

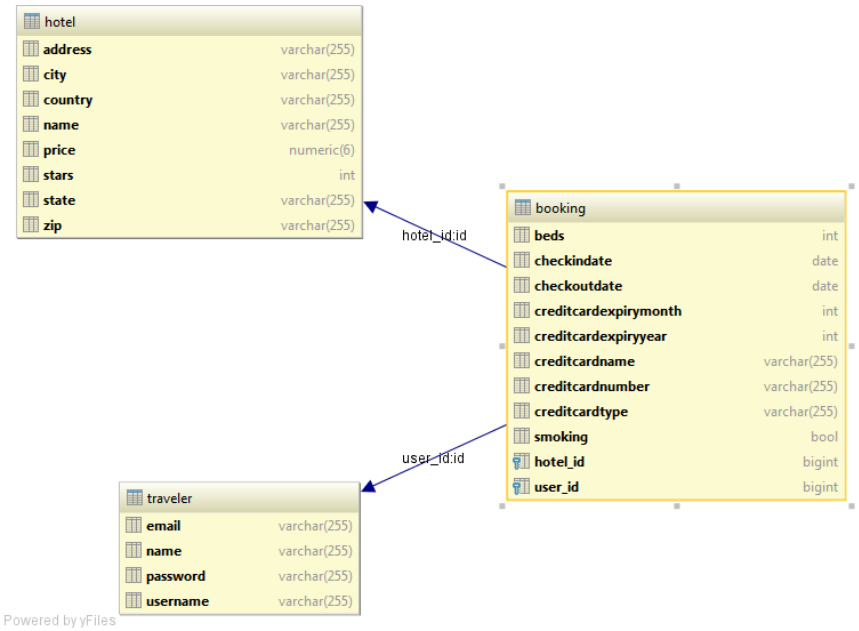
5 Vzorová aplikace

Jako vzorovou aplikaci jsem si chtěl vybrat již existující, relativně známou aplikaci. Vybral jsem si proto jeden ze vzorových projektů Seam, konkrétně Seam Booking[8]. Tento projekt jsem nejdříve upravil tak, aby fungoval s distribuovaným uživatelským rozhraním AspectFaces (tedy přes automaticky generovanou REST službu). Poté jsem přidal zabezpečovací knihovnu a nastavil zabezpečení modelu tak, aby dávalo co největší smysl.

5.1 Popis

Seam Booking je projekt založený na frameworku Seam 3. Je to jednoduchá webová aplikace na správu rezervací hotelů. Je možné se přihlásit, prohlížet hotely, spravovat vlastní vytvořené rezervace, upravovat je. Aplikace pracuje s jednoduchou databázovou strukturou. Má tři tabulky:

- **Hotel** - Tabulka pro jednotlivé hotely
- **Traveler** - Tabulka pro registrované uživatele
- **Booking** - Tabulka vytvořených rezervací (vazba mezi uživatelem a hotelem)



Obrázek 5.1: Původní databázové schéma Seam Booking

Hotel name	Address	Location	Zip	Action
Marriott Courtyard	Tower Place, Buckhead	Atlanta, GA, USA	30305	View
Doubletree Atlanta-Buckhead	3342 Peachtree Road NE	Atlanta, GA, USA	30326	View
Ritz-Carlton Atlanta	181 Peachtree St NE	Atlanta, GA, USA	30303	View

Obrázek 5.2: Ukázka původního zobrazení hotelů v Seam Booking

5.2 Aplikace AspectFaces na vzorovou aplikaci

Cílem aplikace AspectFaces na vzorovou aplikaci je vytvoření distribuovaného uživatelského rozhraní, tedy vytvoření jednotného Controlleru, který bude zpracovávat CRUD operace nad modelem. Dotazy na Controller bude posílat javascript u klienta, který se také bude starat o jejich správné zobrazení podle dat a metamodelu.

AspectFaces Controller vyžadoval, aby třídy modelu rozšiřovaly společnou třídu TimestampedEntityObject, to bylo potřeba v aktuálním modelu změnit. Také bylo třeba upravit mapování AspectFaces tak, aby zahrnovalo třídy z modelu SeamBooking:

Listing 5.1: Mapping pro typ User

```
<mapping>
        <type>User</type>
        <default tag="entitySelectTag" required="false"/>
</mapping>
```

Po aplikaci AspectFaces vypadal např. kód pro vykreslení aktuálních rezervací přihlášeného uživatele takto:

Listing 5.2: Vykreslení rezervací pomocí javascriptu

```
<div class="section">
        <h:panelGroup rendered="#{not identity.loggedIn}">
                You must be logged in to see the list of
                   your hotel bookings.
        </h:panelGroup>
        <span id="bookings" />
</div>
<c:if test="#{not empty currentUser}">
        <div class="id">#{currentUser.id}</div>
        <script type="text/javascript"> loadListForm('org.
            jboss.seam.examples.booking.model.User@bookings
            ', #{currentUser.id}, 'bookings', true);
        </script>
</c:if>
```

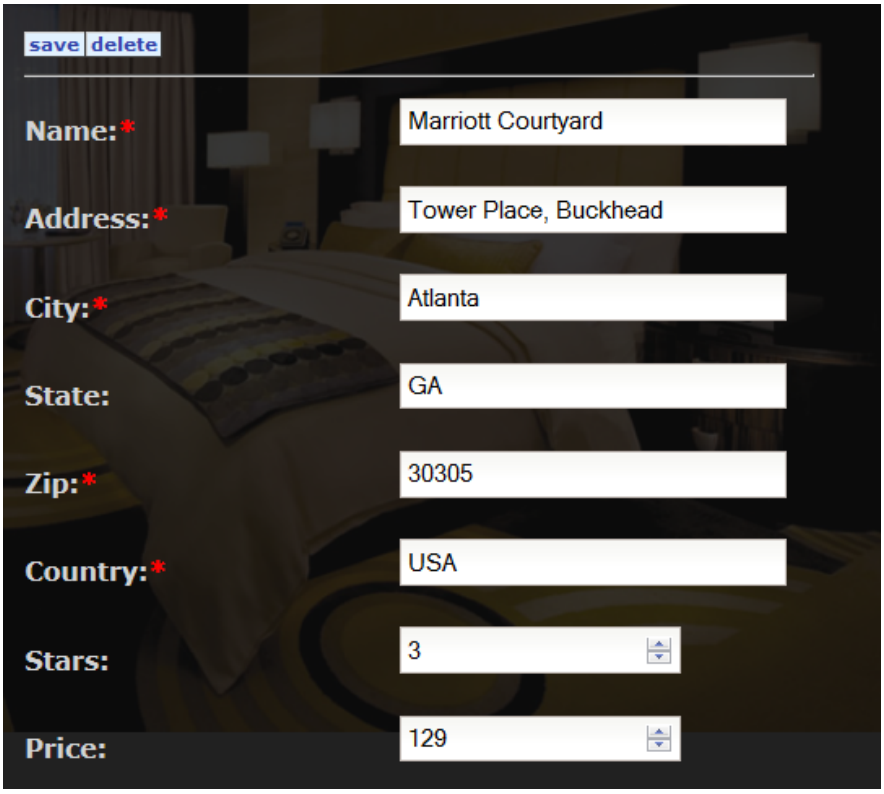
Po zavolání funkce loadListForm javascript nejdříve na controller poslal dotaz na strukturu:

- GET ./af/org.jboss.seam.examples.booking.model.User@bookings

Controller našel třídu podle jména, získal z ní atribut bookings, a vrátil javascriptu metadata třídy org.jboss.seam.examples.booking.model.Booking, což je typ atributu User.bookings. Poté javascript poslal dotaz na samotná data:

- GET ./af/org.jboss.seam.examples.booking.model.User@bookings/1

Controller našel třídu User, načetl si z databáze záznam s id 1, vzal si hodnotu atributu bookings, a ten vrátil javascriptu. Javascript pak na základě metadat o třídě Booking a dat z databáze sestavil formulář s rezervacemi.



Obrázek 5.3: Ukázka formuláře generovaného AspectFaces - Hotel

5.3 Aplikace zabezpečení na vzorovou aplikaci

5.3.1 Úprava controlleru

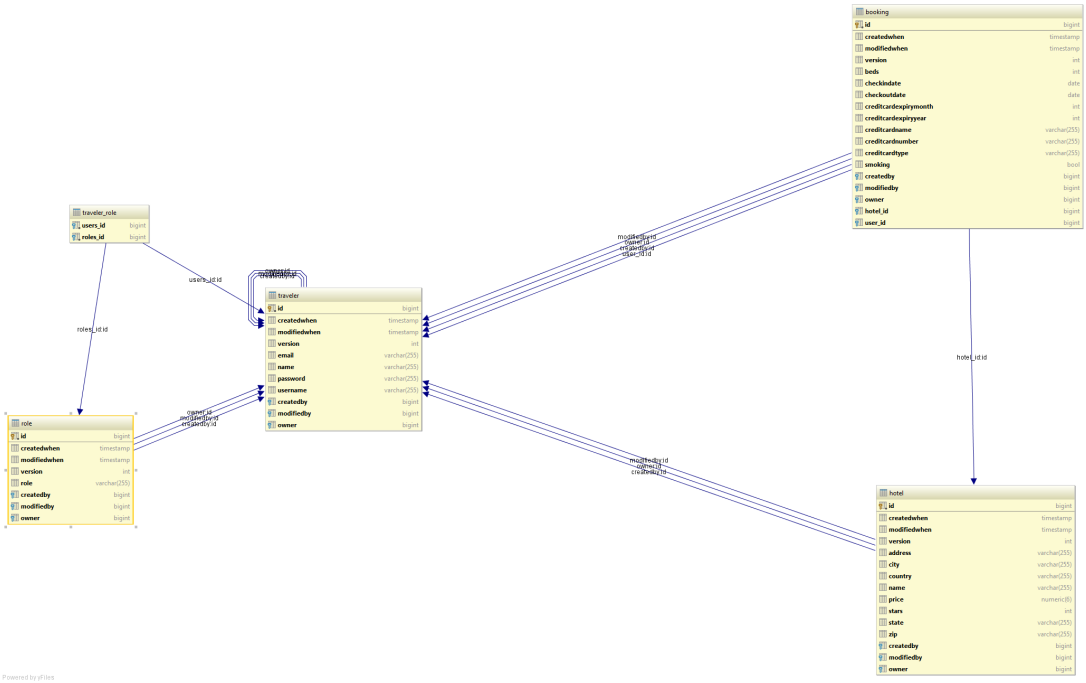
Pro aplikaci zabezpečení bylo třeba upravit i samotný controller. Tím, že zabezpečovací knihovna řeší sama i práci s databází, tak většina práce s databází ubyla z controlleru. Controller také upravuje původní jazyk AspectFaces pro použití v zabezpečovacím frameworku (umisťuje id na správné místo do queryStringu).

5.3.2 Úprava javascriptu

Vzhledem k tomu, že se po aplikaci zabezpečení mohou lišit metadata různých instancí jedné entity, je potřeba, aby se javascript dotazoval ne na metadata pro třídu, ale na metadata pro konkrétní záznam podle id.

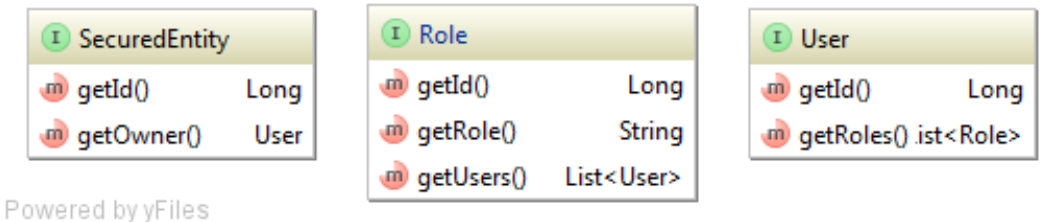
5.3.3 Úprava modelu

Do databáze přibyla entita Role, která je vazbou M:N provázána s entitou User. To je požadavek RoleRestrictProcessoru. Také ke každé entitě přibyl atribut owner, což je vazba na tabulku User. To vyžaduje OwnerRestrictProcessor. Nové schema pak vypadá takto:



Obrázek 5.4: Databázové schéma po aplikaci zabezpečení Seam Booking

Třídy modelu musí implementovat rozhraní SecuredEntity, což je zajištěno dědičností skrz TimestampedEntityObject. Rozhraní SecuredEntity spolu s rozhraním User a Role definuje základní databázovou strukturu. Zajišťuje, že každá entita bude mít atributy id a owner, také zajišťuje to, že tabulka s uživateli bude navázána na tabulku s rolemi.To jsou podmínky, které musí být splněny pro použití RoleRestrict a OwnerRestrict procesorů.:



Obrázek 5.5: Rozhraní User, Role a SecuredEntity

Do modelu byly přidány anotace @OwnerRestrict a @RoleRestrict tak, aby to co nejvíce odpovídalo realitě.

Hotel

Zabezpečení třídy Hotel jsem nastavil tak, že číst záznamy může kdokoliv, ale ostatní CRUD operace jsou povolené jen administrátorům:

Listing 5.3: Zabezpečení třídy Hotel

```
1  @RoleRestrict(
2      allowCreate = {"ADMIN"},
3      allowUpdate = {"ADMIN"},
4      allowDelete = {"ADMIN"}
5  )
6  public class Hotel extends TimestampedEntityObject
       implements Serializable
```

User

Administrátor má na entitě User plná práva, vlastník má právo číst a částečně i upravovat.

Listing 5.4: Zabezpečení třídy User

```
1  @RoleRestrict(
2      allowRead = {"ADMIN"},
3      allowCreate = {"ADMIN"},
4      allowUpdate = {"ADMIN"},
5      allowDelete = {"ADMIN"}
6  )
7  @OwnerRestrict(
8      allowed = {OperationType.READ, OperationType.UPDATE}
9  )
10 public class User extends TimestampedEntityObject
       implements Serializable, cz.ctu.egermma1.security.model
       .User
```

Booking

Rezervace je zabezpečena takto:

- **administrátor** může na rezervaci provádět všechny CRUD operace
- **vlastník** může rezervaci vytvořit, číst a částečně i upravovat. Vlastník může z atributů upravovat jen možnost kuřáckého/nekuřáckého pokoje, žádný jiný atribut změnit nemůže.

Listing 5.5: Zabezpečení třídy Booking

```
@RoleRestrict(
    allowRead = {"ADMIN"},
    allowCreate = {"ADMIN"},
    allowUpdate = {"ADMIN"},
    allowDelete = {"ADMIN"}
)
@OwnerRestrict(
    allowed = {OperationType.CREATE, OperationType.READ,
        OperationType.UPDATE}
)
public class Booking extends TimestampedEntityObject
    implements Serializable, SecuredEntity
```

Listing 5.6: Zabezpečení atributů smoking a beds

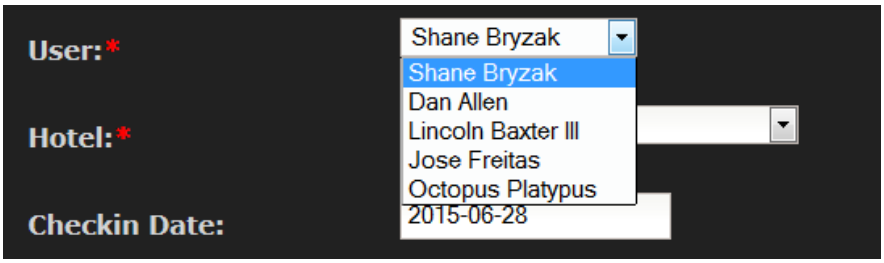
```
@RoleRestrict(
        allowRead = {"ADMIN"},
        allowCreate = {"ADMIN"},
        allowUpdate = {"ADMIN"}
)
@OwnerRestrict(
        allowed = {OperationType.CREATE, OperationType.READ
            , OperationType.UPDATE}
)
private Boolean smoking;

@RoleRestrict(
        allowRead = {"ADMIN"},
        allowCreate = {"ADMIN"},
        allowUpdate = {"ADMIN"}
)
@OwnerRestrict(
        allowed = {OperationType.CREATE, OperationType.READ
            }
)
private Integer beds;
```

5.3.4 Příklady funkcionality

Výběr entity

Práva uživatelů se aplikují i na výběr entity, na kterou se daný záznam má vázat. Z následujících obrázků je vidět rozdíl v dotazu na získání všech záznamů z tabulky User. Administrátor má právo vidět všechny uživatele, na rozdíl od standardního uživatele, který může číst jen sám sebe. V tomto případě se uplatňuje anotace umístěná nad typem.

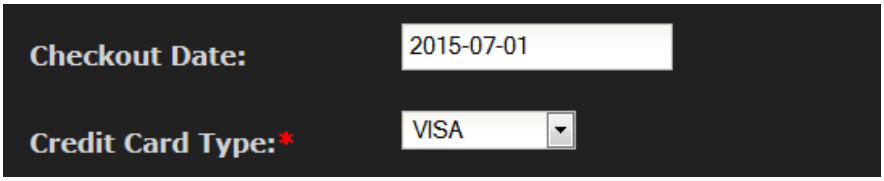


(a) Administrátor

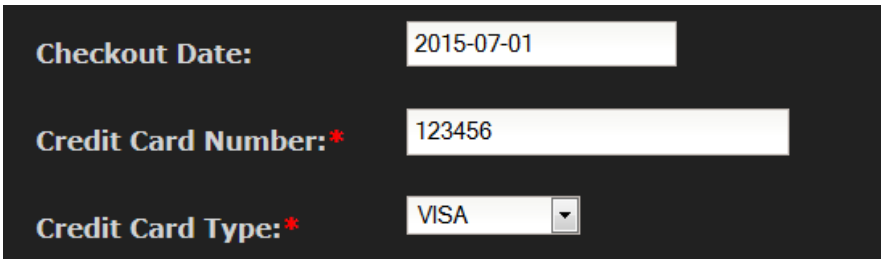
(b) Normální uživatel

Obrázek 5.6: Rozdíl mezi administrátorem a normálním uživatelem - zabezpečení třídy

V dalším příkladě je vidět použití anotace nad atributem. Pokud bychom chtěli zajistit například to, aby administrátor nemohl vidět číslo platební karty, kterou zadává uživatel při založení objednávky, můžeme z pole allowRead udělat denyRead a přidat roli “ADMIN”.



(a) Administrátor



(b) Normální uživatel

Obrázek 5.7: Rozdíl mezi administrátorem a normálním uživatelem - zabezpečení atributu

6 Testování

V rámci práce jsem provedl dva druhy testů. V aplikaci jsem implementoval Unit testy pro kontrolu správnosti procesoru, který zpracovává zabezpečení, a provedl jsem testy výkonnostní, abych mohl porovnat efektivitu zabezpečení oproti původní vzorové aplikaci.

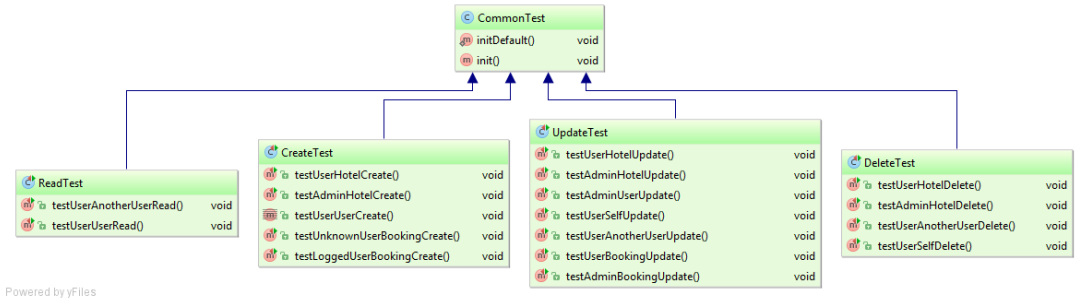
6.1 Unit testy

Protože jsem chtěl pokrýt co největší záběr funkcí zabezpečovací knihovny, vytvořil jsem sadu unit testů. K tomu jsem si vytvořil testovací model, který odpovídá modelu vzorové aplikace Seam booking. Implementoval jsem simulátor EntityManageru, který si místo ukládání dat do databáze drží data v operační paměti. Před spuštěním každého testu jsou data opět inicializována do výchozího stavu, aby se jednotlivé testy neovlivňovaly.

Testy jsou rozděleny do 4 tříd podle jednotlivých CRUD operací:

- **CreateTest** pro testování operace create
- **ReadTest** pro testování čtení záznamů
- **UpdateTest** pro testování operace update
- **DeleteTest** pro testování mazání záznamů

Celkově jsem implementoval 18 testovacích metod:



Obrázek 6.1: Struktura unit testů

6.2 Testy výkonnosti

K testům výkonnosti jsem si vybral stránku, na které je nejvíce změn oproti ostatním. Je to stránka vyhledávání hotelů (/search). Všechny testy byly měřeny po obnovení stránky CTRL+F5, takže při načítání stránky nebylo nic načtené v cache. Neměřil jsem čas do načtení html stránky a zavolání onload, ale až do stavu, kdy byla stránka kompletně načtena.

6.2.1 Prostředí

Celou dobu jsem vyvíjel i testoval na stejném prostředí. Jako aplikační server jsem si zvolil server Jboss AS 7.1.1. Vybral jsem tento server na doporučení vedoucího práce, ale také z důvodu, že ho znám nejlépe z Java aplikačních serverů. Server fungoval s parametry -Xmx4096m -XX:MaxPermSize=512m. Databázi jsem zvolil PostgreSQL. Hlavně proto, že s PotgreSQL nemám takové zkušenosti jako s jinými databázovými stroji, a chtěl jsem se s ním naučit pracovat.

6.2.2 Samotná vzorová aplikace

Vzorovou aplikaci jsem testoval ve stavu, ve kterém jsem ji stáhl z oficiálního repozitáře Seam.

V této verzi stránky není žádné dotahování dat javascriptem, vše se renderuje na serveru.

Z měření vychází, že se průměrná doba zobrazení pohybuje okolo 1714ms. Je to poměrně vysoké číslo, může to být dáno tím, že testy probíhaly na lokálním stroji, kde kromě serveru běželo navíc mnoho dalších aplikací - několik databázových strojů (mysql, postgresql, oracle), vývojové prostředí, internetový prohlížeč apod.

6.2.3 Po aplikaci aspectFaces

Po nasazení aplikace s AspectFaces se průměrný čas zvýšil na 2376ms, což je o cca 650ms více, než s čistou Seam Booking aplikací. V konzoli prohlížeče bylo vidět, že oněch cca 650ms byly dotazy javascriptu na data a metadata. Všechny testy ovšem probíhaly jako první načtení stránky, lze tedy předpokládat, že po uložení části dat do cache by následující zobrazení probíhaly rychleji.

6.2.4 Po aplikaci zabezpečení

Po přidání zabezpečovací knihovny opět stoupla průměrná doba načtení stránky, tentokrát na 3180ms, což je o cca 800ms více, než po uplatnění AspectFaces. V tomto případě jsem u každého dotazu také měřil, kolik času stráví dotaz na backendu (tedy od zavolání controlleru do vrácení odpovědi). Tento čas vyšel průměrně na 530ms. Zbývajících 270ms se dá přisoudit tomu, že se po uplatnění zabezpečení nemohou cachovat metadata pro jednotlivé třídy, ale pro každý záznam v datech se musí stáhnout vlastní metadata. Po prvním stažení už se ale cachují v lokálním úložišti prohlížeče, takže další dotazy už by měly být rychlejší.

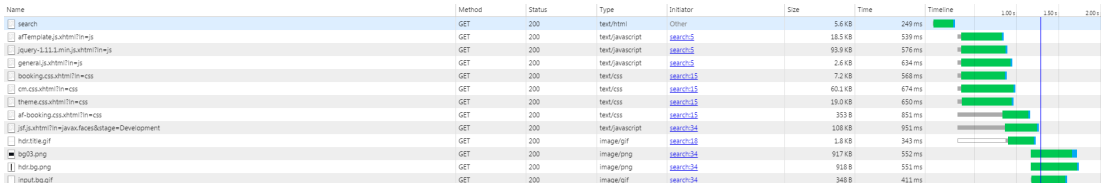
6.2.5 Porovnání výsledků

Zde jsou vidět kompletní data z testů. U každé varianty jsem udělal 15 pokusů. K pokusům jsem používal prohlížeč Google Chrome.

Test č.	Původní aplikace	AspectFaces	Zabezpečení - celkově	Zabezpečení - backend
1	1750	2250	3190	454
2	1810	2170	3500	474
3	1700	2780	3470	554
4	1650	2430	2950	378
5	1690	2250	3420	419
6	1620	2100	3040	579
7	2160	2340	2830	491
8	1920	2280	3520	332
9	1710	2480	3760	1035
10	1860	3010	2870	528
11	1660	2220	2870	587
12	1590	2250	3140	432
13	1550	2320	3310	438
14	1530	2610	2710	509
15	1510	2160	3120	746
průměr	1714	2376.67	3180	530.4

Tabulka 6.1: Časy zobrazení v ms

Z výsledků je jasně vidět, že přidání zabezpečení a automaticky generovaného REST rozhraní zpomalí načítání stránky, v tomto případě na téměř dvojnásobek. Rád bych ale v budoucnu provedl ještě další test na čistém "produkčním" prostředí vystaveném na webu, abych mohl vyzkoušet časy načtení stránky a prodlevu při dotahování dat javascriptem ze vzdáleného stroje, na kterém neběží tolik procesů jako na vývojovém stroji. Také bych chtěl zkusit provést minifikaci javascriptů a css, protože některé css nebo javascriptové soubory mají velikost přes 100KB.



Obrázek 6.2: Ukázka stahování js a css souborů

7 Závěr

7.1 Shrnutí

Cílem této diplomové práce bylo navrhnout a implementovat řešení, které bude řešit zabezpečení automaticky generovaných RESTful služeb. Tohoto cíle jsem dosáhl, řešení jsem navrhl, implementoval, a funkčnost ověřil na vzorové aplikaci Seam Booking. Na vytvoření automaticky generované RESTful služby jsem použil dle zadaní framework AspectFaces, který jsem též integroval do aplikace Seam Booking. Řešení jsem navrhl tak, aby bylo snadno rozšiřitelné i bez úprav v zabezpečovací knihovně tím, že si uživatel knihovny může sám přidat vlastní implementaci zabezpečovaciho procesoru, případně i nahradit obě implementace obsažené v knihovně svými vlastními.

7.2 Výhody řešení

Řešení je navržené tak, aby bylo snadno rozšiřitelné ve chvíli, kdy je potřeba přidat zcela nový typ zabezpečení. Uživatelem implementované nové zabezpečení nemusí brát nutně informace jen z anotací, může si například brát vstupní data z konfiguračního souboru, z jiné databáze, nebo se rozhodovat jen podle náhodných čísel. Hlavní procesor pak výsledky z jednotlivých procesorů spojí dohromady.

7.3 Nevýhody řešení

Současné řešení je navržené tak, že potřebuje mít v každé entitě atribut id typu Long, který reprezentuje primární klíč. To nutí uživatele k tomu, že ačkoliv chce použít primární klíč typu String, musí místo toho použít Long.

Další nevýhoda (omezení) je ta, že jednotlivé zabezpečovací procesory mezi sebou mají vazbu OR, takže není možné spojit výsledky dvou různých procesorů do podoby AND.

7.4 Práce do budoucna

Nejdříve bych rád alespoň zčásti odstranil nevýhody mého řešení. Rád bych provedl zobecnění primárního klíče tak, aby byl generický. Pak by bylo možné používat jakýkoliv objekt jako primární klíč, například i složený klíč. Pokud bude požadavek na jinou vazbu mezi procesory než OR, bude třeba zapracovat jazyk, který podle kontextu propojí jednotlivé procesory jinou vazbou.

Další věc, kterou je třeba udělat, je zajistit si "produkční" prostředí, a znovu vyzkoušet výkon vzorové aplikace. Do tohoto testu zahrnout i minifikaci javascriptů a css souborů.

Chtěl bych také práci přesunout z privátního do veřejného repozitáře, a pokud by byl zájem komunity, dále bych ji rozvíjel.

Literatura

[1] Tomas Cerny, Karel Cemus, Michael J. Donahoo, and Eunjee Song *Aspect-driven, Data-reflective and Context-aware User Interfaces Design.*. ACM, New York, NY, USA, 2013

[2] Tomas Cerny and Eunjee Song *UML-based enhanced rich form generation.*. ACM, New York, NY, USA, 2011

[3] Miroslav Macik, Tomas Cerny, Pavel Slavik *Context-sensitive, cross-platform user interface generation.* Springer Berlin Heidelberg, 2014

[4] Roy Thomas Fielding *Architectural Styles and the Design of Network-based Software Architectures.* UNIVERSITY OF CALIFORNIA, IRVINE, 2000

[5] Informace ze serveru owasp.org *www.owasp.org*

[6] Informace ze serveru aspectfaces.com *www.aspectfaces.com*

[7] PYPL *http://pypl.github.io*

[8] SEAM examples *https://github.com/seam/examples*

Seznam použitých zkratek

CRUD Create, Read, Update, Delete

HTTPS Hypertext Transfer Protocol Secure

LDAP Lightweight Directory Access Protocol

UI User Interface

REST Representational state transfer

SSL Secure Sockets Layer

SSO Single sign-on

Přílohy

- **1. Kompaktní disk**
  - Zdrojové kódy vzorové aplikace: source/SeamExamples
  - Zdrojové kódy zabezpečovací knihovny: source/security
  - Text diplomové práce