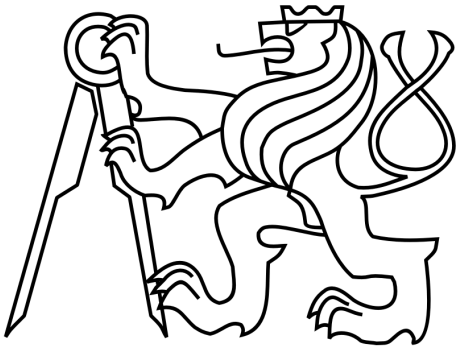
České vysoké učení technické v Praze

Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE



Bc. 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 2016

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Egermajer**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **Zabezpečení automaticky generovaných REST služeb**

Pokyny pro vypracování:

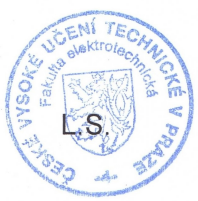
Analyzujte problémy týkající se autentizace a autorizace REST aplikací, včetně best-practice řešení pro platformu Java EE. Dále proveďte rešerši frameworku AspectFaces, který umožní automatické generování REST služeb pro práci s daty. Porovnejte možnosti současných řešení. Navrhněte, implementujte a otestujte vlastní řešení. Ověřte ve vzorové aplikaci, která implementované řešení integruje. Vyhodnoťte efektivitu, výhody a omezení daného řešení.

Seznam odborné literatury:

[1] Tomas Cerny, Karel Cemus, Michael J. Donahoo, and Eunjee Song. 2013. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. In Applied Computing Review, Vol. 13, Issue 4, ACM, New York, NY, USA, 53-65. ISSN 1559-6915 http://www.sigapp.org/acr/Issues/V13.4/ACR-13-4-2013.pdf
[2] Tomas Cerny and Eunjee Song. 2011. UML-based enhanced rich form generation. In Proceedings of the 2011 ACM Symposium on Research in Applied Computation (RACS '11). ACM, New York, NY, USA, 192-199. DOI=10.1145/2103380.2103420 http://doi.acm.org/10.1145/2103380.2103420
[3] Miroslav Macik, Tomas Cerny, Pavel Slavik. 2014. Context-sensitive, cross-platform user interface generation. Journal on Multimodal User Interfaces. Springer Berlin Heidelberg, DOI=10.1007/s12193-013-0141-0
http://link.springer.com/article/10.1007/s12193-013-0141-0

Vedoucí: Ing. Tomáš Černý, MSc.

Platnost zadání: do konce letního semestru 2015/2016



prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 31. 10. 2014

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

Seznam obrázků

Seznam tabulek

1 Úvod

V dnešní době je těžké nenarazit při svých každodenních činnostech na nějakou webovou službu, i když o tom vůbec nemusíme vědět. Stačí, když si v mobilním telefonu vytvoříme úkol na nadcházející den, a telefon zavolá právě takovou 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

Před samotným návrhem a implementací vlastního řešení se musíme nejdříve seznámit s nejdůležitějšími pojmy v oblasti webových služeb. V této kapitole tedy tyto pojmy zadefinujeme.

2.1 Web, Webová služba

Web je dle [10] definován jako prostor informací, které jsou globálně dostupné přes síť Internet. Informace v tomto prostoru jsou označené jako zdroje, jejichž identifikátory se označují jako URI (Uniform Resource Identifiers). V souvislosti s Web definuje [13] Webového agenta - člověk nebo software, který v tomto prostoru pracuje.

Webová služba je abstraktní definice, která musí být implementována konkrétním agentem. Agent je v tomto případě software (příp. i hardware), který přijímá a odesílá zprávy. Teoreticky tedy můžeme mít pro jednu službu (zdroj) několik různých agentů založených na různých technologiích (např. Java a .NET), které mají stejnou funkcionalitu (splňují abstraktní definici). Webové služby fungují na principu klient-server, kde klient vždy zašle požadavek (zprávu) serveru, server požadavek zpracuje, a odešle klientovi odpověď (zprávu).

2.2 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.

2.2.1 Omezení REST

Klient-server

REST architektura požaduje využití architektury klient-server. Je to jedna ze základních síťových architektur. Klient (může být například internetový prohlížeč) posílá požadavky přes síť na server, server je zpracuje, a vrátí klientovi odpověď. 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.

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 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** Klíčový prvek v REST architektuře je *zdroj*. Zdroj může být každá informace, která se dá pojmenovat (obrázek, dokument,

...). Fielding definuje zdroj R jako funkci $M_R(t)$, která pro čas t vrátí *reprezentaci* daného zdroje, nebo jeho *identifikátor*. Některé zdroje mohou být statické (hodnota funkce $M_R(t)$ se nemění v průběhu času), jiné se mohou měnit. Identifikátor zdroje slouží k jednoznačnému určení zdroje. Pojmenovávání není nijak dané a záleží pouze na správci služby, jak službu pojmenuje. Správce je pak také zodpovědný, že danému zdroji zůstane toto pojmenování a nebude se v čase měnit.

- **Manipulace se zdrojem skrze jeho reprezentace** K zobrazení současného stavu zdroje nebo k vyjádření námi požadovaného budoucího stavu zdroje slouží *reprezentace* zdroje. Reprezentace se skládá z posloupnosti bytů a metadat, která popisují bytovou informaci. Datový formát reprezentace je známý jako "Media type"(Multipurpose Internet Mail Extensions - MIME).

- **Samopopisující zprávy** Tento požadavek souvisí s omezením o bezestavovosti. Protože si server neuchovává žádné informace o předchozích požadavcích klienta, musí klient v požadavku uvést všechny potřebné informace pro zpracování na serveru.

- **Hypermedia** Jsou definována jako rozšíření hypertextu, tedy nelineární proud informací, které obsahují obrázky, zvuky, videa, text a odkazy.

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.

2.3 HTTP

Hypertext Transfer Protocol je definován jako bezestavový protokol, který je založený na zasílání požadavků a přijímání odpovědí [14]. Protokol HTTP se aktuálně používá ve verzi 1.1. Protože HTTP pracuje na principu klient-server, definujeme pojmy HTTP klient a HTTP server. HTTP klient je agent, který vytvoří připojení se serverem, kterému pošle jednu nebo více zpráv. HTTP server je agent, který toto připojení přijme, zprávy zpracuje, a odešle klientovi odpovědi. Pokud je v jedné komunikaci agent chová jako server (klient mu zašle požadavek), v jiné komunikaci se může chovat jako klient (pro zaslání odpovědi klientovi zašle dotaz jinému agentovi, jehož odpověď zpracuje, a vrátí odpověď klientovi).

2.3.1 HTTP zpráva

HTTP zpráva je v případě klienta HTTP požadavek, v případě serveru HTTP odpověď. Oba typy HTTP zpráv jsou podobné, liší se pouze v prvním řádku (start-line), a v algoritmu výpočtu velikosti těla zprávy[14].

Ukázka 2.1: Syntaxe HTTP zprávy[14]

```
HTTP-message    = start-line
                  *( header-field CRLF )
                  CRLF
                  [ message-body ]
start-line      = Request-Line | Status-Line
```

Ukázka 2.1 ukazuje syntaxi HTTP zprávy. HTTP zpráva tedy vždy začíná prvním řádkem (start-line). V případě HTTP dotazu obsahuje HTTP metodu, cíl požadavku, a verzi HTTP. HTTP odpověď zde má stav odpovědi. Dále následují jednotlivé řádky hlavičky. Po prázdném řádku případně následuje samotné tělo HTTP zprávy.

2.3.2 HTTP požadavek

HTTP požadavek zasílá klient na server. Protože je HTTP jako takové bezestavové, požadavek musí obsahovat vše, co server potřebuje k úspěšnému vyřízení požadavku. Požadavek vždy obsahuje hlavičku zprávy, v některých případech může obsahovat i tělo.

Ukázka 2.2: Ukázkový HTTP požadavek (bez těla)

```
GET /doodles HTTP/1.1
Host: www.google.com
```

První řádek HTTP požadavku

Ukázka 2.3: Syntaxe prvního řádku HTTP požadavku (SP - single space)[14]

```
request-line = method SP request-target SP HTTP-version
  CRLF
```

Z 2.3 je vidět syntaxe prvního řádku HTTP požadavku. Skládá se z metody, cíle požadavku, a http verze. Na 2.2 vidíme ukázkový HTTP požadavek. V tomto případě jsme zaslali požadavek metodou GET na url https://www.google.com/doodles.

HTTP metoda reprezentuje záměr klienta s vyžádaným zdrojem. HTTP 1.1 definuje 8 metod, které jsou standardizované a pro všechny zdroje stejné, nemusíme tedy definovat pro každý zdroj jinou sadu metod.

HTTP metoda může mít několik vlastností[14]:

- Metoda je **bezpečná**, jestliže je její sémantika definována jako "read-only". To znamená, že klient při použití této metody nevyžaduje a ani neočekává jakoukoliv manipulaci se zdrojem na serverové straně.
- Metoda je **idempotentní**, jestliže několik po sobě jdoucích identických HTTP dotazů mají na serveru stejný učinek, jako jediný z těchto dotazů. Z principu jsou do idempotentních metod zahrnuty všechny bezpečné metody (read-only).
- Metoda je **kešovatelná**, jestliže se odpovědi na tuto metodu mohou uchovávat pro budoucí použití.

Zde je seznam všech HTTP metod, které definuje HTTP 1.1:

- **GET** Metoda pro získání současné reprezentace daného zdroje. Tato metoda je bezpečná, idempotentní i kešovatelná.
- **HEAD** Tato metoda je stejná, jako metoda GET s rozdílem, že se ze serveru neposílá tělo odpovědi, ale pouze první řádek a hlavička. Tato metoda je bezpečná, idempotentní a kešovatelná.
- **POST** Tato metoda slouží k tomu, aby server zpracoval data přiložená k požadavku podle semantiky zdroje. Jedna ze semantik je vytvoření nového zdroje z reprezentace obsažené v požadavku. Tato metoda není bezpečná, není idempotentní, dle specifikace může být ale kešovatelná.
- **PUT** Metoda slouží k přepsání (aktualizaci) daného zdroje, jehož reprezentace je přítomna v požadavku. Tato metoda je idempotentní.
- **DELETE** Metoda pro mazání zdroje. Tato metoda je idempotentní.
- **CONNECT** Vytvoří tunel mezi klientem a serverem, který je identifikovaný zdrojem.
- **OPTIONS** Po zaslání požadavku s touto metodou by měl server vrátit možnosti, jakými je možné komunikovat s daným zdrojem. Klient tak může předem zjistit, jaké metody může ke komunikaci použít, s jakým serverem komunikuje apod. Tato metoda je bezpečná a idempotentní.
- **TRACE** Po zavolání této metody se klientovi ze serveru vrátí jako odpověď to, co přijal jako požadavek (až na pár informací v hlavičce). Klient tak má možnost nahlédnout do toho, jaké informace přijdou na server, a využít toho např. k ladícím účelům. Tato metoda je bezpečná a idempotentní.

Hlavička HTTP požadavku

Hlavička HTTP požadavku může obsahovat informace o kontextu, ve kterém vznikl HTTP požadavek (například User-Agent, tedy na jakém klientovi dotaz vznikl), preferované formáty odpovědi, autentikaci apod.

Ukázka 2.4: Ukázka validního HTTP 1.0 požadavku

```
GET / HTTP/1.0
```

HTTP ve verzi 1.0 nevyžadovalo žádnou z hlaviček požadavku, 2.4 ukazuje nejjednodušší možný HTTP 1.0 požadavek. HTTP 1.1 už vyžaduje hlavičku Host, 2.2 tedy ukazuje nejjednodušší HTTP 1.1 dotaz. Zde jsou příklady některých HTTP hlaviček:

- **Accept** Hlavička, která upřesňuje akceptovatelné typy odpovědí, např. Accept: text/html
- **User-Agent** Obsahuje informace o klientovi (např. internetový prohlížeč, verzi systému apod.), např.
  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:43.0) Gecko/20100101 Firefox/43.0

Ukázka 2.5: Ukázka proaktivního content negotiation

```
Accept-Language: en, cs;q=0.7,en-US;q=0.3
```

Některé z hlaviček mohou obsahovat tzv. Content negotiation. Server totiž může umět vracet v odpovědi různé způsoby reprezentace zdrojů, ať už jde o formát dat (např. xml, json), o jazyk (cs, en), nebo o kódování. Rozlišujeme proaktivní a reaktivní content negotiation.

Při proaktivním pošle klient v hlavičce své preference (2.5), a server se pak na základě informací z hlavičky rozhodne, který obsah poskytne. Může se rozhodovat na základě preferencí, které klient zašle, na základě klientovy IP adresy, nebo hlavičky User-agent.

Při reaktivním content negotiation si klient vyžádá ze serveru prvotní data a informace o tom, jaké možnosti server nabízí. Následně se klient může rozhodnout, jestli se spokojí s odpovědí serveru, nebo si pomocí dalšího dotazu vybere jinou alternativu (na základě metadat z odpovědi).

2.3.3 HTTP odpověď

HTTP odpověď je zpráva, kterou zašle server klientovi jako reakci na zaslání požadavku klienta. Odpověď má stejnou strukturu jako požadavek, liší se v

prvním řádku a v obsahu HTTP hlavičky.

Ukázka 2.6: Ukázka HTTP odpovědi

```
HTTP/1.1 200 OK
Date: Tue, 29 Dec 2015 15:51:12 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
```

První řádek HTTP odpovědi

Ukázka 2.7: Syntaxe prvního řádku HTTP odpovědi (SP - single space)[14]

```
status-line =
        HTTP-version SP status-code SP reason-phrase CRLF
```

Z 2.7 je vidět syntaxe prvního řádku HTTP odpovědi. Skládá se z verze použitého HTTP protokolu, z návratového kódu a ze slovního popisu stavu.

Stav je trojciferné číslo, které dává klientovi informaci o tom, jak byl jeho požadavek zpracován. První cifra určuje zařazení do kategorií, další dvě cifry upřesňují. Dle [15] máme následující kategorie stavů:

- **1xx - informační** Požadavek byl přijat, ale je stále zpracováván. Stavy začínající 1 nejsou v HTTP 1.0 podporované, server je může používat jen při komunikaci přes HTTP 1.1.
- **2xx - úspěšné zpracování** Požadavek byl přijat, server požadavku rozuměl a požadavek vykonal. Např. stav 200 při použití metody GET znamená, že server vrací reprezentaci požadovaného zdroje.
- **3xx - přesměrování** Tyto kódy indikují, že ke splnění požadavku je třeba další činnost na straně klienta. Např. stav 307 znamená, že se zdroj dočasně přesunul na jinou URI, ale při dalším volání má použít opět aktuální, ne dočasnou.
- **4xx - chyba na straně klienta** Kódy začínající na 4 napovídají klientovi, že je v jeho požadavku chyba. Např. stav 403 indikuje že server porozuměl požadavku, ale požadavek odmítl autorizovat.
- **5xx - chyba na straně serveru** Server se dostal do neočekávané situace a požadavek preventivně odmítl. Kupříkladu stav 501 znamená, že server nemá implementovanou funkcionalitu pro daný požadavek, např. nepodporuje klientem zaslanou HTTP metodu.

2.4 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) je komunikační protokol založený na HTTP. HTTPS využívá ke komunikaci protokol HTTP, komunikace je ale navíc šifrovaná pomocí další vrstvy TLS (případně jeho předchůdcem SSL). Mezi klientem a serverem je nejdříve vytvořeno zabezpečené připojení, poté je zaslán samotný HTTP požadavek.

3 Rešerše

V předchozí kapitole jsme se věnovali definici základních pojmů, které se vztahují k webovým službám používajícím architekturu REST. Seznámili jsme se s pojmem web a webová služba, definovali jsme samotný pojem REST, ukázali jsme protokol HTTP a šifrovanou verzi HTTPS.

Pro návrh a implementaci vlastního řešení ochrany webových služeb je třeba prozkoumat současné metody zabezpečování webových služeb, rozdělení na autentizaci a autorizaci, analyzujeme některé druhy útoků a jejich prevenci. Ke správnému návrhu architektury mého řešení je také třeba prozkoumat architekturu Java EE aplikací obecně a porovnat již existující řešení.

3.1 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.

3.2 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.**

3.2.1 Další architektonická doporučení pro autentizaci

Dle serveru owasp[5] by se měl architektonický návrh aplikace držet následujících doporučení (uvádím 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.

3.2.2 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

Jde o útok, kdy útočník získá session id uživatele. 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.

Man-in-the-middle

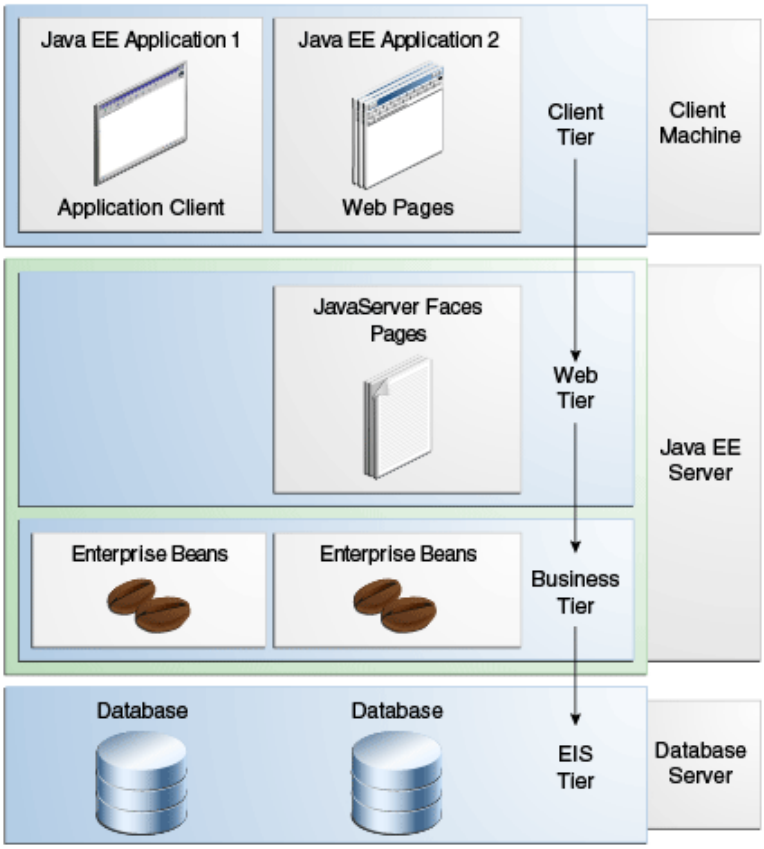
V tomto případě útok spočívá v tom, že klient nekomunikuje přímo s cílovým serverem, ale komunikuje přes útočníka. Útočník pak může číst komunikaci klienta s cílovým serverem a zároveň může tuto komunikaci měnit (klient pošle požadavek na zaplacení 20Kč, ale útočník požadavek změní a na server dorazí 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.

3.3 Autorizace

Ve chvíli, kdy klient požádá server o manipulaci se zdrojem, je cílem autorizace tento požadavek buď povolit, nebo zamítnout. Autorizační proces zpravidla následuje po procesu autentizačním. Podrobněji se budu autorizaci věnovat v následujících kapitolách.

3.4 Architektura JavaEE

JavaEE používá pro své aplikace model distribuovaného vícevrsvého aplikačního modelu [17].



Obrázek 3.1: Architektura JavaEE aplikace[17]

Vrstvy jsou následující:

- **Klientská vrstva** Tato vrstva se nachází na straně klienta. Může to být desktopová aplikace, která vzdáleně pracuje s Java EE aplikací, nebo například internetový prohlížeč.
- **Webová vrstva** Webová vrstva běží na straně JavaEE serveru. Obsahuje *Servlety* (Java třídy, které zpracovávají klientské požadavky a vytvářejí odpovědi) a *JSP* (textové dokumenty, které se chovají jako servlet, ale nabízí přirozenější cestu pro vytváření obsahu).
- **Business vrstva** Business vrstva běží na straně JavaEE serveru. Tato vrstva obsahuje kód, který obsahuje vlastní logiku pro zpracování klientských požadavků.
- **EIS vrstva** Tato vrstva zpravidla obsahuje databázový server.

3.4.1 Specifikace JAX-RS

JAX-RS, aktuálně ve verzi 2.0 (JSR 339), poskytuje API pro vývoj, vystavení a přístupování webových služeb, které splňují principy REST[11]. Toto api je

součást samotné Java EE, konkrétně v balíčku javax.ws.rs.

3.5 Existující řešení

V této části prozkoumáme již existující řešení zabezpečení REST služeb. Nejdříve prozkoumáme existující řešení pro programovací jazyk Java, poté i pro jiné programovací jazyky. Na existujících řešeních provedeme analýzu jejich integraci do webové aplikace a výhody a omezení jejich použití.

3.5.1 Jersey

Jersey je jedna z knihoven, která implementuje specifikaci JAX-RS (JSR 339 i starší JSR 311). Kromě implementace vlastní specifikace JAX-RS Jersey zároveň tuto specifikaci rozšiřuje i o vlastní API, například o tvorbu klientských požadavků (balíček javax.ws.rs.client).

Ukázka 3.1: Ukázka jednoduché třídy reprezentující zdroj v Jersey[12]

```
1  package org.glassfish.jersey.examples.helloworld;
2
3  import javax.ws.rs.GET;
4  import javax.ws.rs.Path;
5  import javax.ws.rs.Produces;
6
7  @Path("helloworld")
8  public class HelloWorldResource {
9
10     @GET
11     @Produces("text/plain")
12     public String getHello() {
13         return "Hello World!";
14     }
15
16 }
```

Třídy, které v Jersey reprezentují zdroj, jsou typu POJO, tedy třídy, která zpravidla neimplementuje žádné rozhraní a nerozšiřuje žádnou třídu. Tyto třídy jsou následně doplněny o upřesňující anotace - identifikace zdroje, určení jednotlivých HTTP metod apod. Příklad takovéto třídy můžeme vidět na 3.1. Jde zde vidět zdroj identifikovaný anotací @Path jako ”helloworld”, a jedna metoda getHello() zpracovávající požadavky přicházející metodou GET.

Jersey podporuje několik způsobů autentizace:

- **Basic** Tento způsob autentizace je založený na tom, že se klient musí pro nějakou oblast webu prokázat uživatelským id a heslem. Server úspěšně

zpracuje klientův požadavek pouze v případě, že má pro danou oblast správné údaje. Tyto údaje se pak přenáší jako hlavička HTTP dotazu ve tvaru *Authorization: Basic base64(u':'p)*, kde base64 je funkce převádějící plaintext do base64, 'u' je uživatelské id a 'p' je heslo. Konečná hlavička pak může vypadat takto:
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

- **Digest** Na rozdíl od Basic autentizace neposílá Digest heslo ve snadno dekódovatelné formě jako je base64, ale používá několikanásobné hashování.

- **OAuth 1** Jersey má plnou podporu pro OAuth 1.0 (jak pro klientskou část, tak pro serverovou). *OAuth* umožňuje autentizaci na server pomocí služby třetí strany (autentizační) bez toho, aby klient musel zasílat své autentizační údaje skrze náš server. Klient se autentizuje skrze třetí stranu, a náš server poté pracuje pouze s autentizačním tokenem, kterým si u třetí strany ověří platnost autentizace.

- **OAuth 2** OAuth ve verzi 2.0 podporuje Jersey pouze na straně klienta.

Ukázka 3.2: Ukázka zabezpečení zdroje přes web.xml

```
<security-constraint>
    <web-resource-collection>
        <url-pattern>/rest/orders/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>customer</role-name>
    </auth-constraint>
</security-constraint>
```

Pokud je REST služba použita v servletovém kontejneru (tedy aplikace je nasazena na aplikačním serveru), může být autorizace řešena přes aplikační deskriptor (web.xml). Toto řešení je vidět na 3.2. Dokáže zabezpečit zdroj na základě uživatelských rolí, rozlišuje i různé nastavení pro různé HTTP metody jednoho zdroje. Výhoda tohoto řešení je v tom, že web.xml je nezávislý na Jersey i na JAX-RS. Nevýhoda je, že pokud se změní identifikace zdroje (@Path), je třeba toto změnit i ve web.xml. Dále se toto řešení omezuje pouze na autorizaci pomocí uživatelských rolí.

Další možností je použití třídy RolesAllowedDynamicFeature, kterou poskytuje Jersey.

Ukázka 3.3: Registrace třídy RolesAllowedDynamicFeature

```
public class MyApplication extends ResourceConfig {
    public MyApplication() {
        super(MyResource.class);
        register(RolesAllowedDynamicFeature.class);
    }
}
```

3.3 ukazuje registraci třídy RolesAllowedDynamicFeature. Po registraci je možné použít anotace z balíčku javax.annotation.security, tedy například @DenyAll, @PermitAll a @RolesAllowed. Tyto anotace je možné použít jak nad typem, tak nad metodou.

Ukázka 3.4: Rozšíření 3.1 o zabezpečeí podle rolí

```
1  @Path("helloworld")
2  @PermitAll
3  public class HelloWorldResource {
4      @GET
5      @RolesAllowed("admin")
6      @Produces("text/plain")
7      public String getHello() {
8          return "Hello World!";
9      }
10 }
```

Na příkladu 3.4 je vidět použití javax.annotation.security anotací. Anotace @PermitAll nad typem definuje, že všechny uživatelské role mají přístup k metodám třídy HelloWorldResource, pokud metoda tuto anotaci nepřepisuje. Anotace @RolesAllowed nad metodou getHello přepisuje anotaci uvedenou nad třídou, a povoluje pouze uživatelskou roli admin.

Interně třída RolesAllowedDynamicFeature funguje tak, že nejdříve provede inspekci daného zdroje a konkrétní vyžádané metody, kde zjistí přítomnost/nepřítomnost jednotlivých anotací javax.annotation.security. Poté na základě výsledků inspekce inicializuje filtr RolesAllowedRequestFilter a zařadí jeho instanci do řetězce ostatních filtrů. Tento filtr pak rozhoduje o vyvolání výjimky ForbiddenException v případě neautorizovaného požadavku, nebo v případě autorizovaného požadavku neudělá nic a nechá proces pokračovat.

Je možné místo třídy RolesAllowedDynamicFeature použít vlastní impmelentaci, vlastní filtr musí implementovat jedno z pěti rozhraních daných JAX-RS:

- **ContainerRequestFilter** Implementace tohoto filtru má možnost upravovat HTTP hlavičku klientova dotazu ještě před zpracováním.

- **ContainerResponseFilter** Funkcionalita je stejná jako u ContainerRequest-Filter, jen není upravována hlavička HTTP požadavku ale odpovědi (po zpracování).
- **ReaderInterceptor** Toto rozhraní umožňuje modifikovat tělo klientského požadavku.
- **WriterInterceptor** Toto rozhraní umožňuje modifikovat tělo odpovědi.
- **Feature** Implementace tohoto rozhraní umožňuje rekonfigurovat JAX-RS konfiguraci

3.5.2 RESTEasy

RESTEasy (nyní ve verzi 3.0.13)[20] je další z implementací JAX-RS. Jeho funkcionalita je ve velkém shodná s funkcionalitou Jersey, jsou zde ale rozdíly.

RESTEasy rozšiřuje JAX-RS o anotace, které definují kešovací hlavičky HTTP odpovědi. Pomocí anotací @Cache a @NoCache knihovna sama přidává potřebné hlavičky do HTTP odpovědi.

Ukázka 3.5: Ukázka vytvoření cache hlavičky v RESTEasy

```
1  @Path("helloworld")
2  @PermitAll
3  @Cache(maxAge=86400, isPrivate  = true)
4  public class HelloWorldResource {
5      @GET
6      @RolesAllowed("admin")
7      @Produces("text/plain")
8      public Response getHello() {
9          return Response.ok("Hello World!").build();
10     }
11 }
```

Na 3.5 je vidět anotace Cache v RESTEasy. Z následujícího kódu je patrný rozdíl mezi RESTEasy a Jersey. V Jersey je nutné kešování definovat kódem, v RESTEasy hlavičkou.

Ukázka 3.6: Ukázka vytvoření cache hlavičky v Jersey

```
@Path("helloworld")
@PermitAll
public class HelloWorldResource {
    @GET
    @RolesAllowed("admin")
    @Produces("text/plain")
    public Response getHello() {
        CacheControl cc = new CacheControl();
        cc.setMaxAge(86400);
        cc.setPrivate(true);

        ResponseBuilder builder = Response.ok("Hello
            World!");
        builder.cacheControl(cc);
        return builder.build();
    }
}
```

Narozdíl od Jersey také podporuje OAuth 2.0 na serverové straně. RESTEasy má také zabudovanou integraci se Spring a Spring MVC.

3.5.3 Restlet

Restlet je další framework, který implementuje JAX-RS. Restlet používá v základu vlastní anotace (org.restlet.resource.Get, org.restlet.resource.Post apod.), nepodporuje tedy anotace JAX-RS. To se dá ale snadno změnit stažením příslušného JAX-RS Restlet rozšíření (org.restlet.ext.jaxrs). Restlet v základu podporuje HTTP Basic a SMTP Plain (obdoba HTTP Basic) autentizaci, použitím rozšíření se dá využít i HTTP Digest a OAuth 2.0.

Ukázka 3.7: Ukázka autorizace v Restlet

```
1  @Override
2  public Restlet createInboundRoot() {
3  ...
4      RoleAuthorizer authorizer = createRoleAuthorizer();
5  ...
6      authorizer.setNext(Resource0.class);
7  ...
8      baseRouter.attach("/resourceTypePrivate",
           authorizer);
9  ...
10 }
11
12 private RoleAuthorizer createRoleAuthorizer() {
13     //Authorize owners and forbid users on roleAuth's
           children
14     RoleAuthorizer roleAuth = new RoleAuthorizer();
15     roleAuth.getAuthorizedRoles().add(Role.get(this,
           ROLE_OWNER));
16     roleAuth.getForbiddenRoles().add(Role.get(this,
           ROLE_USER));
17     return roleAuth;
18 }
```

Autorizace se v Restlet definuje v kódu3.7. Je možné využít již implementované mechanizmy na ochranu zdroje podle uživatelských rolí, nebo metod daného zdroje podle autorizovaného/neautorizovaného uživatele. Je také možné vytvořit si vlastní autorizační mechanizmus rozšířením třídy Authorizer a implementováním metody *protected abstract boolean authorize(Request request, Response response);*.

Toto řešení má výhodu v tom, že je možné ho generovat za běhu aplikace na základě například inspekce zdrojového kódu. Nevýhoda oproti jax-rs anotacím je zřejmá - mnoho řádků kódu navíc. Na celý příklad 3.7 by stačila jedna jax-rs anotace nad zdrojem *@RolesAllowed("owner")*.

Ukázka 3.8: Ukázka Restlet Java SE[21]

```
1  public class FirstServerResource extends
       ServerResource {
2
3      public static void main(String[] args) throws
           Exception {
4          // Create the HTTP server and listen on port
               8182
5          new Server(
6              Protocol.HTTP,
7              8182,
8              FirstServerResource.class)
9          .start();
10     }
11
12     @Get
13     public String toString() {
14         return "hello, world";
15     }
16
17 }
```

Jednou z hlavních výhod tohoto frameworku je jeho nezávislost na servletovém kontejneru, restlet může být tedy spuštěn i jako Java SE aplikace, viz 3.8. Další výhodou je rozsáhlost knihovny a jejich oficiálních rozšíření.

3.6 Shrnutí

V této kapitole jsme si ukázali rozdíl, mezi autentizací a autorizací, architekturu JavaEE aplikací, základní standard pro JavaEE REST služby JAX-RS i s několika knihovnami, které tento standard implementují.

JAX-RS definuje pouze jeden způsob autorizace uživatelů - autorizaci podle uživatelských rolí. Knihovna zjistí, jaké má autentizovaný uživatel role, a inspekcí zdrojového kódu buď operaci povolí, nebo zamítne. To nemusí být vždy dostatečné řešení. Mějme například rezervační systém cestovní kanceláře, a v něm dvě role - registrovaný uživatel a administrátor. Uživatel může vytvářet rezervace, administrátor je navíc může upravovat a mazat. JAX-RS už nám ale neumožní přidat podmínku, kdy by uživatel mohl navíc rezervaci upravovat 14 dní před odletem. Tato podmínka nám ukazuje dva problémy. Zaprvé je třeba použít vlastní implementaci, protože JAX-RS takovouto podmínku u autorizace nepodporuje, a zadruhé na tuto podmínku nestačí použít pouze inspekci zdrojového kódu, ale je nutné nejdříve načíst z úložiště (databáze) žádaný záznam rezervace a porovnat aktuální datum s datem odletu.

Další pozivitum je možnost implementace vlastního autorizačního filtru, která je probrána v podkapitole Jersey. Filtry (a Interceptory) slouží k modifikaci HTTP požadavku a HTTP odpovědi, tedy vzhledem k předchozímu odstavci není možné je použít. Bylo by nutné přesunout část logiky z třídy reprezentující zdroj do filtru, konkrétně načítání dat z databáze, a to není z architektonického hlediska přijatelné. Jako pozitivum si tedy odneseme možnost implementace knihovnou definovaného rozhraní, které do knihovny zaregistrujeme a ona zařídí jeho zpracování (i když ne ve formě filtrů). Jde nám tedy o snadnou rozšiřitelnost a přizpůsobivost knihovny.

V podkapitole Restlet je rozebráno porovnání mezi použitím anotací pro definici autorizace a definicí kódem. Výhoda v použití anotací je ta, že má programátor (uživatel knihovny) lepší přehled o tom, kde je jaké zabezpečení uplatněno (podívá se na zdroj, a přímo u něj a u jeho metod zabezpečení vidí). Pokud zabezpečení definuje v kódu, musí vynaložit větší mentální úsilí aby zjistil současné nastavení a provedl případné úpravy. Nevýhoda řešení pomocí anotací je taková, že anotace nemusí dostatečně dobře nebo přehledně vyjádřit programátorův úmysl nového typu zabezpečení. Kupříkladu pokud budeme mít seznam 20 000 uživatelů (generovaný jinou aplikací do souboru users.txt), kteří mají oprávnění přistupovat ke zdroji, není reálné všechny je vyjmenovat do anotace. Zde by bylo lepší u daného zdroje číst tento seznam z filesystemu (nebo z jiné webové služby).

Výše uvedené knihovny buď klientovi zamítnou jeho požadavek jako neautorizovaný, nebo ho provedou celý. Žádná z těchto knihoven ale nenabízí možnost provést požadavek jen částečně. Řekněme, že si chce klient zobrazit všechny zájezdy, které cestovní kancelář nabízí. V databázi jsou ale i historické záznamy, které jsou určené pouze pro interní použití cestovní kanceláří. Programátor tedy nakofiguruje aplikaci tak, aby se registrovaný uživatel k zájezdům dostal, zároveň ale musí ručně naprogramovat logiku, která odfiltruje historické záznamy pro normálního uživatele, ale pro administrátora ne. Částečné provedení požadavku ale souvisí i s tím, že přihlášený uživatel nemusí vidět všechny informace k jednomu záznamu. Vezměme si internetový obchod. U každého jeho produktu je uvedena poruchovost (kolik reklamací bylo uplatněno na počet prodaných kusů). Takto informace slouží k interním účelům, klient by si ji ale zobrazit neměl (při výběru by preferoval pouze méně poruchové produkty). Pokud si tedy klient vyžádá reprezentaci daného produktu (zdroje), vrátí se mu jiná data než administrátorovi.

Z této kapitoly tedy vyplývají následující požadavky pro návrh zabezpečovací knihovny:

- **Integrace knihovny s JavaEE aplikací** Knihovna musí být v architek-

tuře umístěna na takové místo, aby měla přístup k reprezentaci zdroje, a zároveň aby měla informace o přihlášeném uživateli.

- **Rozšiřitelnost a přizpůsobitelnost** Protože nedokážeme pokrýt všechny možné způsoby autorizace, které budoucí uživatel knihovny bude chtít využít, musí být knihovna rozšiřitelná a přizpůsobitelná.
- **Umožnění různých způsobů definice autorizace** Knihovna musí být navržena tak, aby podporovala různé způsoby definice autorizace, ochrany zdroje (anotace, konfigurační soubor, volání vzdálené služby apod.)
- **Částečné zpracování požadavku** Knihovna by měla umět kromě autorizování požadavků automaticky filtrovat data tak, aby tuto funkcionalitu nemusel ručně programovat programátor.

4 Framework AspectFaces

Aspect faces je framework, který usnadňuje vývoj a údržbu webových aplikací, hlavně jejich frontendovou část - uživatelské rozhraní (UI).

Mějme formulář, který v UI reprezentuje entitnu na backendu (entita - třída označená anotací javax.persistence.Entity. Typicky entita reprezentuje tabulku v databázi, její instance jsou potom jednotlivé řádky v tabulce). Pokud se rozhodneme, že do databázové tabulky přidáme nějaký sloupec, a tedy rozšíříme i entitní třídu, je třeba provést další úpravy na frontendu. Je třeba ručně přidat (naprogramovat) do formuláře nové vstupní pole podle typu přidaného atributu (text, textarea, select, ...). Je třeba naprogramovat vzhled (přidat popisek, umístit vstupní pole do formuláře, upravit css třídy apod.), naprogramovat plnění tohoto pole hodnotou, naprogramovat klientské validace a další. Všechny tyto činnosti jsou potřeba vykonat znovu vždy, když se rozšíří entita. Tento přístup má mnoho nevýhod - velké množství kódu (pro relativně malou změnu entity), který obsahuje mnoho duplicit, v kódu jsou na jednom místě smíchané různé oblasti aplikace dohromady (vzhled, zabezpečení, validace, data).

AspectFaces poskytuje řešení v podobě automaticky generovaného UI na základě konfigurace a inspekce zdrojového kódu.

4.1 Architektura

AspectFaces využívá AOP (Aspect Oriented Programming) k dosažení co největší modularity kódu, tedy rozdělit kód do modulů s podobnou funkcionalitou tak, aby se moduly svou funkcionalitou co nejméně překrývaly. Při použití frameworku tedy programátor naprogramuje pro každou oblast znovupoužitelný kód, jejich spojení v uživatelském rozhraní zajistí framework.

Framework pracuje ve třech hlavních krocích[6]: Inspekce kódu, transformace, a integrace.

4.1.1 Inspekce

Během tohoto kroku jsou na základě znalosti konkrétní instance entitní třídy nejdříve zjištěny veškeré informace dostupné ze zdrojového kódu (atributy třídy, validace, anotace apod.). Z těchto informací je vytvořen metamodel reprezentující data. Dále je metamodel upraven na základě znalosti aplikačního kontextu (role přihlášeného uživatele, jeho nastavení apod.).

4.1.2 Transformace

Tranfsormace je druhý krok v procesu. Právě tento krok využívá AOP. Dělí se na tři podkroky: Aplikace zobrazovacích pravidel, aplikace šablony, a aplikace rozložení.

Zobrazovací pravidla

Tato pravidla určují, jakým způsobem se mají mapovat datové atributy (atributy entity) do UI, tedy jaká část kódu (widget) se použije pro zobrazení v UI.

Ukázka 4.1: Zobrazovací pravidla - String a Integer

```
<configuration>
   <mapping>
      <type>String</type>
      <default
         tag="inputTextTag"
         maxLength="255"
         required="false"/>
      <condition
         expression="${not empty email and email == true}"
         tag="emailTextTag"/>
   </mapping>
   <mapping>
      <type>Integer</type>
      <default
         tag="numberTag"
         required="false"/>
   </mapping>
</configuration>
```

Na 4.1 je vidět ukázka mapování. Je v něm definováno mapování pro datový typ String a Integer, kde jako výchozí widget pro String je *inputTextTag* s atributy *maxLength* a *required*, a pro Integer je to widget *numberTag*. Je zde také vidět možnost podmíněného výběru, kde pokud je v kontextu vlastnost email a je nastavena na hodnotu true, místo widgetu *inputTextTag* se použije widget *emailTextTag*.

Šablona

Cílem zobrazovacích pravidel je vybrat žádanou šablonu (widget). Tato šablona je naprogramována v konkrétním jazyce tak, jak chceme zobrazovat daný atribut v UI. AspectFaces navíc tento jazyk rozšiřují o možnost přistupovat k metamodelu na úrovni atributů, a ke kontextu.

Ukázka 4.2: Ukázková šablona pro text

```
1  <ui:define name="input">
2     <h:inputText
3           rendered="#{empty render$field.firstToUpper()$ ? 'true' :
            render$field.firstToUpper()$}"
4           required="#{empty required$field.firstToUpper()$ ? $required$ :
            required$field.firstToUpper()$}"
5           id="#{prefix}$field$"
6           maxlength="$maxLength$"
7           size="$size$"
8           title="#{text['$entityBean.shortClassName().firstToLower()$.$field$']}"
9           value="#{$entityBean.shortClassName()$.$field$}"/>
10 </ui:define>
11
12 <ui:define name="output">
13    <h:outputText value="#{$entityBean.shortClassName()$.$field$}"/>
14 </ui:define>
```

4.2 ukazuje widget napsaný pomocí JSF. Symbol *$* je ona nadstavba (ve formě EL), kterou poskytuje AspectFaces. Například vlastnost *maxlength="maxLength"* na řádku 6 je získána z 4.1, řádku 6.

Rozložení

Jak zobrazovací pravidla, tak šablony se týkaly pouze jednotlivých atributů. Pokud má tedy entita více atributů, pro každý z nich se vybírala šablona podle zobrazovacích pravidel. Úkolem rozložení je definovat ‘obal’ na jednotlivé widgety, tedy jak mají být jednotlivé widgety rozvrženy (v jednom sloupci pod sebou, ve dvou sloupcích, apod.).

Ukázka 4.3: Ukázka jednosloupcového rozložení (tabulkou)

```
1  <table>
2     <af:iteration-part maxOccurs="100">
3        <tr>
4           <td>
5              $af:next$
6           </td>
7           <!-- <td>
8              $af:next$
9           </td> -->
10       </tr>
11    </af:iteration-part>
12 </table>
```

4.3 ukazuje jednosloupcové rozložení (po odkomentování řádků 7 - 9 se z něj stane dvousloupcové).

4.1.3 Integrace

Tento krok vezme výsledky z předchozích kroků, a vytvoří reprezentaci UI, které rozumí interpreter daného jazyka (v ukázce Java a JSF).

4.2 Použití frameworku

Samotné použití frameworku je velmi jednoduché.

Ukázka 4.4: Ukázka použití frameworku

```
<div id="someWrapper">
   <af:ui
      instance="#{controller.personInstance}"
      layout="personDoubleLayout"/>
</div>
```

Vykreslení instance uložené v proměnné *instance* je vidět na 4.4. Pro vykreslení je vyžadováno rozložení (layout) someLayout.

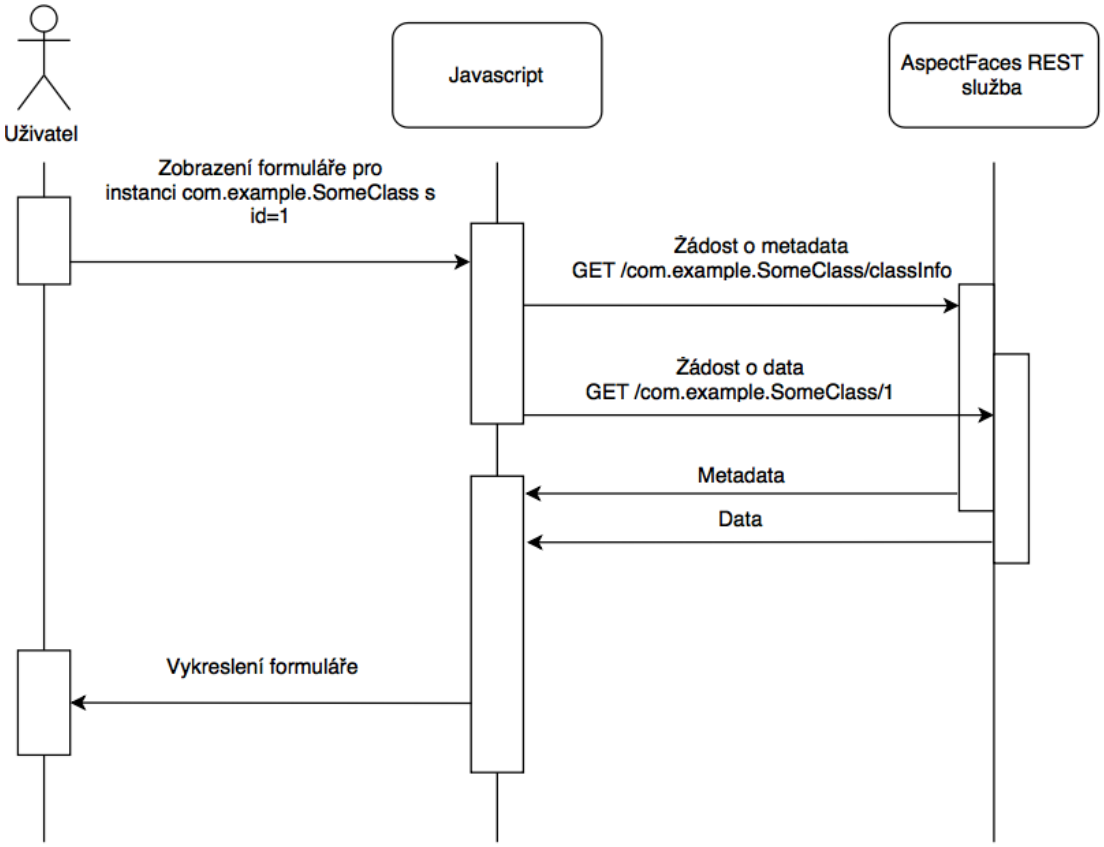
4.3 Distribuované uživatelské rozhraní

Framework AspectFaces umožňuje kromě použití tagů4.4 využít distribuované uživatelské rozhraní. To znamená, že vykreslení instance neprobíhá na straně serveru, ale na straně klienta. Tím jsou přesunuty i některé kroky frameworku AspectFaces ze serveru ke klientovi. Na serveru zůstává inspekce kódu a z transformace pouze část zobrazovacích pravidel. Zbytek logiky je přesunut ke klientovi.

Klient prostřednictvím automaticky generované REST služby získá data potřebná pro vykreslení, provede jejich integraci a zajistí jejich vykreslení.

Ukázka 4.5: Použití distribuovaného uživatelského rozhraní

```
<div class="wrapper">
   <span id="marker" />
</div>
<script type="text/javascript">
   loadForm("com.example.SomeClass",#{someId},"marker", false);
</script>
```



Obrázek 4.1: Diagram komunikace distribuovaného UI

Ukázka 4.5 a Obrázek 4.1 ukazují použití distribuovaného uživatelského rozhraní a komunikaci mezi klientem a serverem, v tomto případě je na klientské straně internetový prohlížeč a javascript. Stejně tak ale může být klientská aplikace např. mobilní aplikace, která je vykreslována na základě dat ze serveru, nebo desktop aplikace.

Ukázka 4.6: Syntaxe vytvoření url pro dotazovací jazyk

```
URL      = / class-name postfix
postfix  = | /id | @attribute-name postfix
```

Definice dotazů, které server přijímá je vidět na Ukázce 4.6 (ukazuje syntaxi url, na kterou jsou posílány klientské dotazy). Url se skládá ze jména třídy a postfixu, kde postfix je buď prázdný, nebo obsahuje /id (tedy identifikátor zdroje),

nebo atribut entitní třídy, na který se dotazujeme (plus rekurzivně opět postfix). Operaci nad zdrojem definuje klient pomocí HTTP metody. Zde je několik příkladů dotazů:

- **GET /com.example.SomeClass/13**
  Vrátí reprezentaci zdroje com.example.SomeClass s id 13
- **DELETE /com.example.SomeClass/13**
  Smaže zdroj com.example.SomeClass s id 13
- **GET /com.example.SomeClass@someAttribute/13**
  Vrátí reprezentaci zdroje pro atribut someAttribute zdroje com.example.SomeClass s id 13

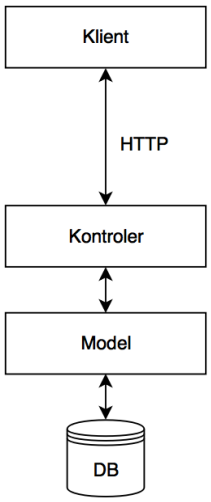
5 Návrh a implementace vlastního řešení

5.1 Architektura

Z kapitoly Rešerše vyplnynulo několik požadavků, které má zabezpečovací knihovna řešit. V této části požadavky projdeme, a na jejich základě navrhneme architekturu knihovny.

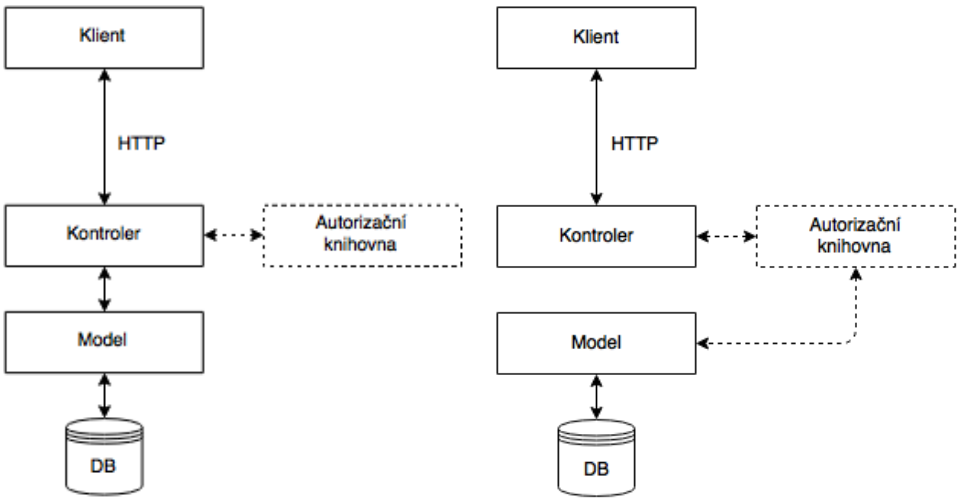
5.1.1 Integrace knihovny s JavaEE aplikací

Z tohoto požadavku vyplývá umístění knihovny v rámci Java EE aplikace.



Obrázek 5.1: Architektura pro zpracování HTTP dotazu

Na obrázku 5.1 je vidět jednoduchá architektura pro zpracování HTTP dotazu. Máme klienta, který pošle HTTP dotaz na server. Na serveru dotaz přijde do kontroleru, který zjistí, jakou operaci dotaz požaduje. Kontroler předá požadavek do modelu, který zajistí jeho provedení nad datovým úložištěm (databáze). Informace o provedení operace (příp. data) jsou přes model vrácena kontroleru, který vytvoří HTTP odpověď.



(a) Nezávislé na databázové vrstvě (b) Nad databázobou vrsvtou

Obrázek 5.2: Možnosti umístění knihovny

Máme několik možností, jak knihovnu do architektury integrovat.

První možnost je umístění knihovny nezávisle na modelu. Tedy kontroler nejdříve získá od modelu data, předá je autentizační knihovně a zjistí výsledek autentizace, a pak případně provede/neprovede klientem požadovanou akci. Výhodou tohoto řešení je naprostá nezávislost na implementaci získávání dat (databáze, textový soubor apod.). Nevýhodou je složitější implementace pro programátora REST služby - v kontroleru je třeba logika navíc, která bude zpracovávat jak výsledky z databáze, tak výsledky autorizační knihovny.

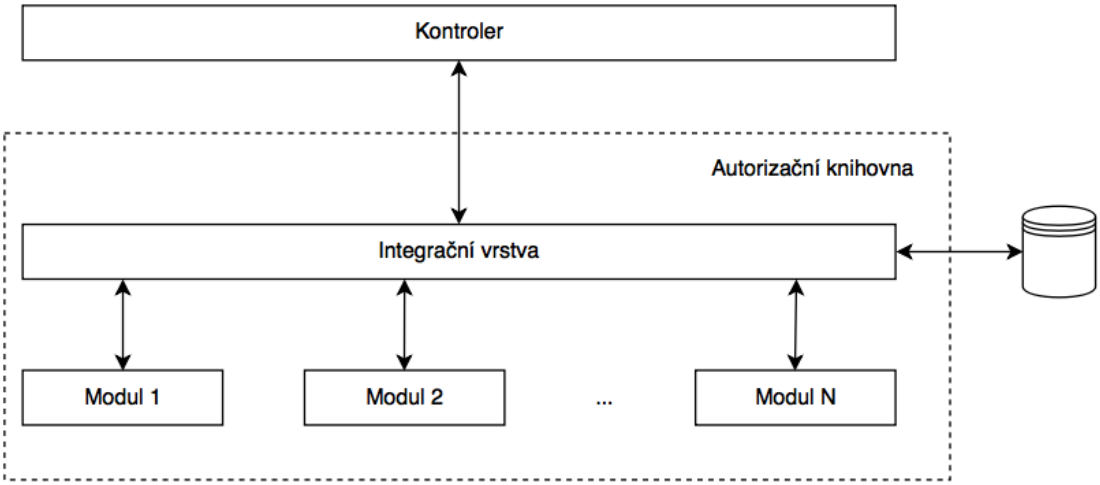
Další možnost je vložit knihovnu mezi kontroler a model. Kontroler předá všechny potřebné informace autorizační knihovně a ona sama zajistí operaci nad daty.Usnadní se tím práce programátorovi, protože bude pracovat pouze s autorizační knihovnou. Nevýhodou bude závislost na konkrétním datovém řešení. Tuto nevýhodu může odstranit správný návrh inicializace autorizační knihovny.

5.1.2 Rozšiřitelnost a přizpůsobitelnost

Protože nemůžeme pokrýt všechny potřeby, které budou mít budoucí uživatelé knihovny, musí být knihovna snadno rozšiřitelná a přizpůsobitelná. Proto se bude knihovna skládat ze dvou částí:

- **Konkrétní moduly řešící autorizaci** Tato vrstva se bude skládat z jednotlivých modulů řešících autorizaci. Každý z těchto modulů bude řešit autorizaci podle jeho kompetence (jeden modul podle rolí, druhý modul podle vlastníka záznamu apod.).

- **Integrační vrstva** Tato vrstva bude přijímat požadavky od kontroleru, bude zajišťovat běh jednotlivých modulů, a na základě jejich výsledků bude dávat dohromady celkový výsledek, který vrátí kontroleru.



Obrázek 5.3: Návrh vnitřní architektury knihovny

Je nutné také ošetřit případy, kdy dva různé moduly vrátí protichůdné výsledky (jeden z nich požadavek autorizuje, druhý ho zamítne). Tato situace bude běžně nastávat (například autorizace podle vlastnictví záznamu požadavek zamítne, protože je ale uživatel administrátor, tak mu autorizace podle rolí operaci autorizuje). Proto bude navíc každý modul vracet kromě informace o autorizovaní/zamítnutí požadavku i prioritu, s jakou dané rozhodnutí učinil. Pokud jeden modul zamítne autorizaci s prioritou 1, a druhý povolí s prioritou 2, vezme se výsledek s vyšší prioritou.

5.1.3 Umožnění různých způsobů definice autorizace

Tento požadavek je rozšíření předchozího bodu tak, aby byla knihovna nejen rozšiřitelná, ale také nezávislá na konkrétní definici autorizace. Tedy aby knihovna nenutila jednotlivé moduly využívat například pouze anotací (nebo pouze xml definice apod.). Toho docílíme správnou volbou rozhraní, které bude definovat jednotlivé modulové implementace.

5.1.4 Částečné zpracování požadavku

Kromě samotného povolení/zakázání operace uživateli je nutné zapracovat i částečné povolení/zamítnutí požadavku. Jednotlivé moduly tedy nebudou pouze rozhodovat o povolení/zamítnutí požadavku jako celku, ale zároveň budou toto roz-

hodnutí dělat i pro každý atribut objektu. Na integrační vrstvě pak bude zodpovědnost za správné vyhodnocení informací z jednotlivých procesorů a provedení celého/částečného požadavku.

5.2 Implementace

5.2.1 Hlavní integrační modul (procesor)

Hlavní modul je navržen tak, aby při své inicializaci přijímal buď instanci JPA třídy javax.persistence.EntityManager, nebo implementaci rozhraní cz.ctu.egermma1.security.db.DatabaseHandler, který vyžaduje implementaci CRUD operací. Programátor tedy může implementovat toto rozhraní, a pracovat s daty umístěnými například v textovém souboru.

Ukázka 5.1: Rozhraní pro práci s daty

```
public interface DatabaseHandler {
    <T> void create(T entity);
    <T> T read(Class<T> pClass, Long pId);
    <T> List<T> readAll(Class<T> pClass);
    <T> T update(T entity);
    <T> void delete(T entity);
}
```

Hlavní modul vystavuje čtyři metody pro každou CRUD operaci. Všechny metody jsou volány s definicí požadavku (query, definováno níže), a s daty o aktuálně přihlášeném uživateli (user). Metody pro create a update navíc potřebují data, se kterými mají zdroj vytvořit (upravit).

Ukázka 5.2: Metody vystavené integračním modulem

```
1  public void addRestrictProcessor(
2          RestrictProcessor customRestrictProcessor);
3  public void processCreate(
4          String query,
5          Map<String, Object> attributes,
6          User user)
7                  throws UnauthorizedOperationException;
8  public Collection<ProcessedEntity> processRead(
9          String query,
10         User user)
11                 throws UnauthorizedOperationException;
12 public void processUpdate(
13         String query,
14         Map<String, Object> attributes,
15         User user)
16                 throws UnauthorizedOperationException;
17 public void processDelete(
18         String query,
19         User user)
20                 throws UnauthorizedOperationException;
```

Pokud si potřebuje programátor definovat vlastní autorizační modul (probráno dále v textu), může ho do autorizační knihovny zaregistrovat pomocí volání metody na řádku 1 v ukázce 5.2, která jako parametr přijímá vlastní implementaci rozhraní *cz.ctu.egermma1.security.processor.RestrictProcessor*.

Popis dotazovacího jazyka

Jazyk, který přijímá autorizační knihovna, jsem částečně převzal z frameworku AspectFaces, jeho syntaxi jsem ale rozšířil.

Ukázka 5.3: Syntaxe dotazu pro autorizační knihovnu

```
Query       = class-name postfix
postfix     = | #id postfix | @attribute-name postfix
```

Hlavní rozdíl je v práci s id. Autorizační knihovna umožňuje mít v rámci jednoho dotazu více id. Dotaz může vypadat například takto:

Ukázka 5.4: Ukázka validního dotazu (query)

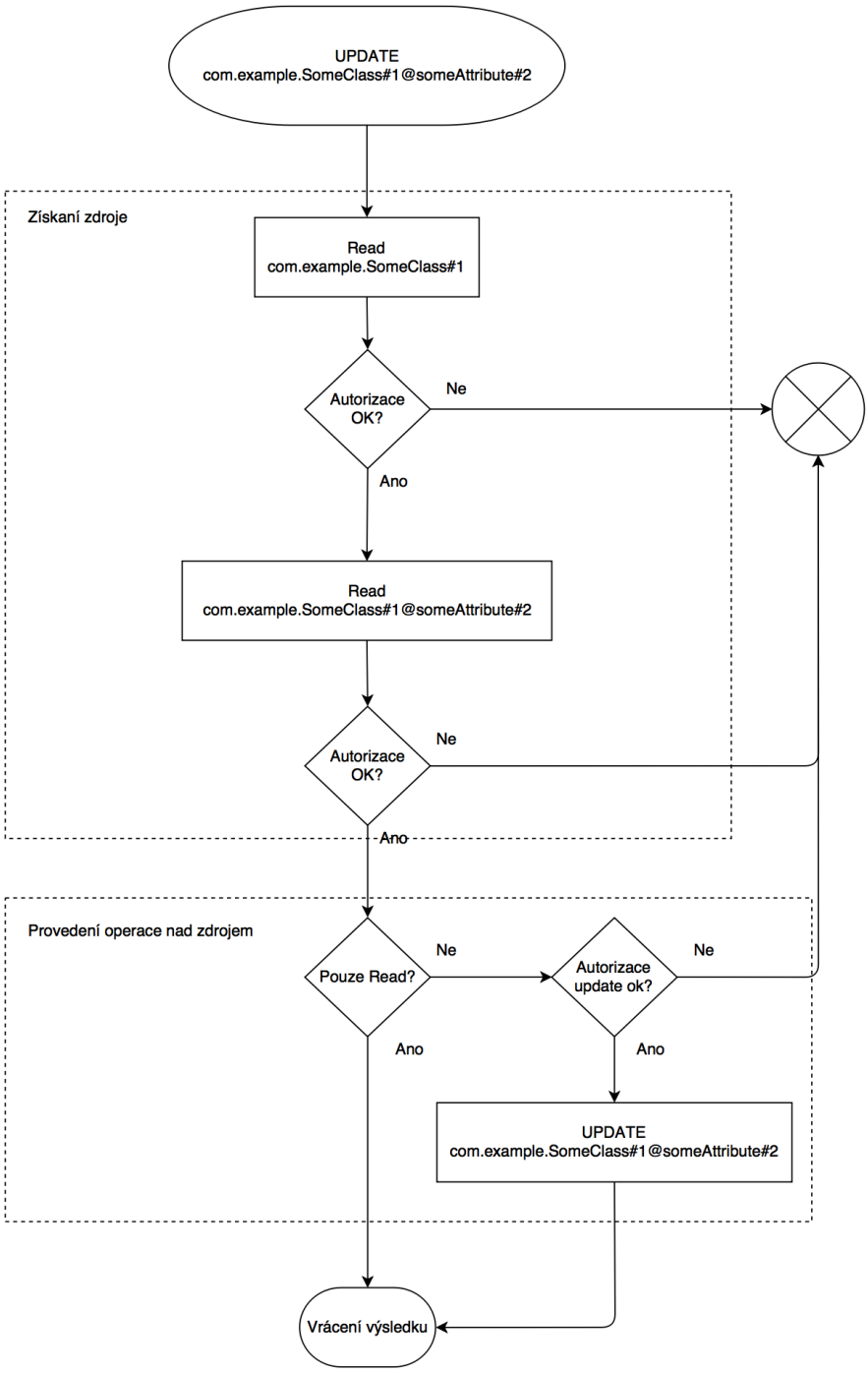
```
com.example.SomeClass#12@someAttribute#12
```

Práce integračního modulu

Integrační modul pracuje ve dvou fázích. V první fázi se dotaz rozdělí dle znaku @, z čehož dostaneme počet mezikroků, které musíme udělat před nalezením cílového

zdroje. Modul iteruje postupně podle mezikroků, a čte zdroj reprezentovaný danou částí dotazu. Jakmile narazí na část, na kterou nemá uživatel oprávnění ke čtení, ukončí práci a vrátí informaci o neautorizování požadavku.

V druhé fázi (tedy po nalezení zdroje podle dotazu) se v případě požadavku na operaci READ výsledek vrátí, v opačném případě se provede kontrola na autorizaci požadované operace. Pokud je vše v pořádku, knihovna operaci provede a vrátí výsledek, v opačném případě vrátí informaci o neautorizovaném požadavku.



Obrázek 5.4: Flow diagram pro požadavek UPDATE pro dotaz z ukázky 5.4

5.2.2 Autorizační podmoduly

Autorizační podmodul je kód, který zajišťuje konkrétní způsob autorizace. Tyto podmoduly pak postupně volá integrační modul, a z jejich výsledků dává dohromady jeden závěr. Každý podmodul musí implementovat rozhraní *cz.ctu.egermma1.security.processor.RestrictProcessor*:

Ukázka 5.5: Rozhraní RestrictProcessor

```
public interface RestrictProcessor {
    ClassMetadata processCreate(Class<?> clazz,
        Object instance,
        List<String> attributes,
        User currentUser);

    ClassMetadata processRead(Class<?> clazz,
        Object instance,
        List<String> attributes,
        User currentUser);

    ClassMetadata processUpdate(Class<?> clazz,
        Object instance,
        List<String> attributes,
        User currentUser);

    ClassMetadata processDelete(Class<?> clazz,
        Object instance,
        List<String> attributes,
        User currentUser);
}
```

Na ukázce 5.5 je vidět rozhraní *RestrictProcessor*. Každá z jeho metod vrací metadata - informaci o tom, jaký je výsledek konkrétního podmodulu o autorizaci požadavku. Metadata jsou ukázána na Ukázce 5.6. Samotná třída *ClassMetadata* reprezentuje celkový výsledek autorizace, tedy zda je operace alespoň částečně autorizovaná, nebo celá neautorizovaná. Pokud je autorizovaná, potom seznam metadat atributů *List<Attribute> attributes* poskytuje informaci o autorizaci operace nad jednotlivými atributy.

Ukázka 5.6: Obsah třídy ClassMetadata a Attribute

```
public class ClassMetadata {
    // ALLOW,DENY - informace o tom, zda je operace
       autorizovana
    private RestrictType restrictType;
    // Priorita, s jakou byl rozhodnut restrictType
    private int priority;
    // Metadata o atributech
    private List<Attribute> attributes;
}

public class Attribute {
    // Jmeno atributu
    private String name;
    // ALLOW,DENY - informace o tom, zda je castecna
       operace autorizovana na tomto atributu
    private RestrictType restrictType;
    // Priorita, s jakou byl rozhodnut restrictType
    private int priority;
}
```

V rámci knihovny jsem implementoval dva základní autorizační procesory, a to podle uživatelských rolí (RoleRestrictProcessor), a podle vlastnictví záznamu (OwnerRestrictProcessor). Oba tyto procesory jsou řízené anotacemi.

RoleRestrictProcessor

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

Ukázka 5.7: Anotace RoleRestrict

```
@Target(value = {ElementType.TYPE, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface RoleRestrict {
    String[] allowCreate() default {};
    String[] denyCreate() default {};
    String[] allowRead() default {};
    String[] denyRead() default {};
    String[] allowUpdate() default {};
    String[] denyUpdate() default {};
    String[] allowDelete() default {};
    String[] denyDelete() default {};
    int priority() default 1;
}
```

Pokud není u třídy nebo u atributu uvedena anotace @RoleRestrict, vrací podmodul automaticky příznak ALLOW s prioritou Integer.MIN_VALUE, takže operaci povolí, ale jen s nejnižší možnou prioritou. Stejně se podmodul chová, když

sice anotace existuje, ale pro danou operaci je pole allow i deny prázdné. Zde je příklad:

Ukázka 5.8: Přiklad aplikace anotace RoleRestrict

```
@RoleRestrict(
    allowRead = {"REGISTERED\_USER", "ADMIN"},
    allowUpdate = {"ADMIN"},
    allowDelete = {"ADMIN"}
)
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 podmodul 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.

Ukázka 5.9: Anotace OwnerRestrict

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

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 použití anotace @OwnerRestrict:

Ukázka 5.10: Použiti anotace OwnerRestrict

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

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

5.3 Použití knihovny

Knihovna je spravována pomocí nástroje Apache Maven[22], knihovna vyžaduje použití JRE alespoň ve verzi 1.7.

5.3.1 Instalace

Pro build aplikace stačí zavolat maven: *mvn clean package*, případně *mvn clean install*. Problém s buildem může způsobovat pouze knihovna Lombok[23], která je v autorizační knihovně použita. Tato knihovna umožňuje modifikaci zdrojového kódu během kompilace (modifikuje AST před samotnou kompilací Java kódu do bytekódu), bohužel není podporována některými Java kompilátory.

Po buildu lze knihovnu přidat do maven projektu pomocí tagu *dependency*:

Ukázka 5.11: pom.xml

```
<dependency>
    <groupId>cz.ctu.egermma1</groupId>
    <artifactId>security</artifactId>
    <version>${current_version}</version>
</dependency>
```

5.3.2 Použití programátorem

Po přidání knihovny do projektu musí programátor vytvořit třídu reprezentující REST zdroj. Tato třída může buď reprezentovat jeden zdroj, nebo může být (jako v případě frameworku AspectFaces) pro více zdrojů najednou. Programátor musí knihovně zajistit (případně sestavit) dotaz (viz podkapitola Popis dotazovacího jazyka), aktuálně autentizovaného uživatele, a pro operace CREATE a UPDATE i data, se kterými se má nový zdroj vytvořit (dle ukázky 5.2).

Na ukázce 5.12 je vidět volání autorizační knihovny pro opreaci READ. Knihovně je předán vytvoření dotaz *queryString* a přihlášený uživatel *user*, knihovna vrátí množinu instancí třídy *ProcessedEntity*. Tato třída neobsahuje nic jiného, než instanci reprezentující data zdroje, která prošla úplnou/částečnou autorizací (ostatní mají hodnotu null), a seznam atributů, které prošly částečnou/úplnou autorizací (aby nedošlo k záměně opravdové hodnoty null s neautorizovaným atributem), viz. Ukázka 5.13. Na základě těchto informací může kontroler vrátit klientovi reprezentaci požadovaného zdroje.

Ukázka 5.12: Volání autorizační knihovny programátorem

```
Collection<ProcessedEntity> processedEntities =
    getProcessor().processRead(queryString, user);
```

Ukázka 5.13: Třída ProcessedEntity

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ProcessedEntity {
    private Object instance;
    private Collection<String> attributes;
}
```

5.3.3 Rozšiřitelnost

Pokud programátorovi nestačí vestavěné autorizační podmoduly, může si implementovat vlastní. Podmínkou je implementace rozhraní *cz.ctu.egermma1.security.processor.RestrictProcessor* a registrace této implementace po inicializaci knihovny (viz. ukázka 5.14).

Ukázka 5.14: Registrace vlastního podprocesoru

```
Processor processor = new Processor(dataHandler);
processor.addRestrictProcessor(customRestrictProcessor)
```

6 Vzorová aplikace

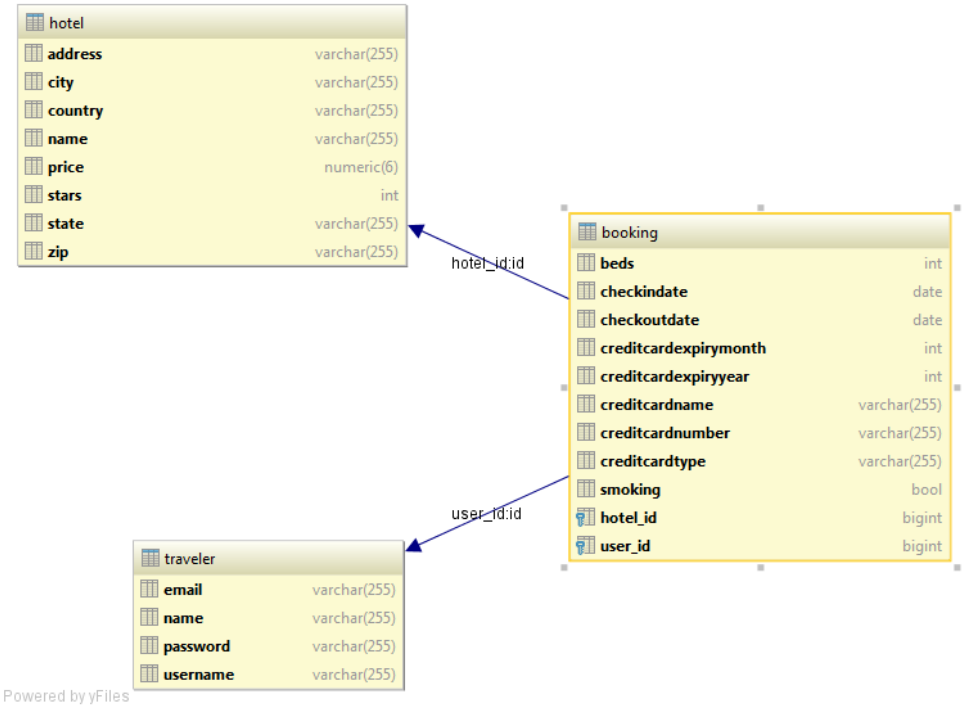
Abychom demonstrovali funkce autorizační knihovny, vezmeme nějaký veřejně známý projekt, a aplikujeme na něj naši autorizační knihovnu. Uživatelské rozhraní upravíme tak, aby bylo generováno frameworkem AspectFaces, konkrétně jeho verzí s distribuovaným UI. Na automaticky generovanou REST službu poté aplikujeme zabezpečovací knihovnu.

Za vzorovou aplikaci byl vybrán jeden z ukázkových projektů frameworku Seam, konkrétně Seam Booking[8].

6.1 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 6.1: 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 6.2: Ukázka původního zobrazení hotelů v Seam Booking

6.2 AspectFaces ve vzorové aplikaci

Prvním krokem je použití frameworku AspectFaces na vzorovou aplikaci. Do aplikace je přidána automaticky generovaná REST služba (tedy jedna třída představující více zdrojů), na kterou UI (javascript) posílá své požadavky. Následují kroky na straně serveru, které vyžaduje framework AspectFaces. Jsou upraveny entitní třídy (musí rozšiřovat třídu TimestampedEntityObject), jsou rozšířena zobrazovací pravidla (hlavně o nové entitní třídy Seamu), jsou upraveny javascriptové šablony (widgety). Nakonec je upraveno UI tak, aby bylo vykreslování pomocí javascriptu na základě odpovědí z REST služby.

Ukázka 6.1: Ukázka části zobrazovacích pravidel pro typ User

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

Ukázka 6.2: Ukázka kódu pro vykreslení rezervací

```
<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>
```

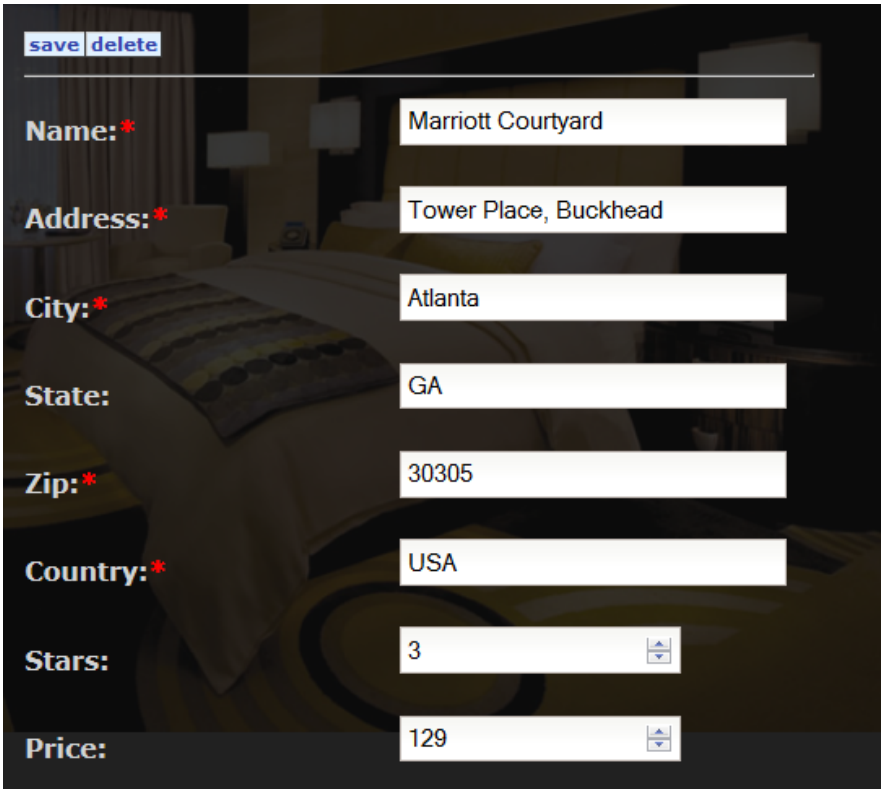
Po zavolání funkce loadListForm se javascript nejdříve dotáže na metadata atributu User.bookings:

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

Controller inspekcí zdrojového kódu najde třídu podle jména, získá z ní atribut bookings, a vrátí javascriptu metadata třídy org.jboss.seam.examples.booking.model.Booking, což je typ atributu User.bookings. Poté javascript pošle dotaz na samotná data:

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

Controller najde třídu User, načte si z databáze záznam s id 1, vezme si hodnotu atributu bookings, a ten vrátí javascriptu. Javascript pak na základě dat a metadat sestaví formulář s rezervacemi.



Obrázek 6.3: Ukázka formuláře generovaného AspectFaces - Hotel

6.3 Aplikace zabezpečení na vzorovou aplikaci

6.3.1 Úprava kontroleru

Konrtoler musí být pro aplikaci autorizační knihovny upraven dle podkapitoly 5.3. Z kontroleru ubyla většina práce s databází (byla přesunuta do zabezpečovací knihovny), v kontoleru naopak přibyla logika, která transformuje dotazovací jazyk frameworku AspectFaces na jazyk zabezpečovací knihovny.

6.3.2 Úprava javascriptu

Protože zabezpečovací knihovna může pro jednu entitní třídu vracet různě strukturované reprezentace zdroje (například pro jeden záznam vrátí 4 atributy, pro druhý 6), je nutné, aby si javascript vyžádal metadata ne pro třídu, ale pro každý záznam.

6.3.3 Úprava databáze a modelu

Do databáze přibyla tabulka Role, která je vazbou M:N provázána s tabulka User. To je požadavek RoleRestrictProcessoru. V každé tabulce přibyl sloupec owner,

což je vazba na tabulku User. To vyžaduje OwnerRestrictProcessor.

Entitní třídy 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.

Entitní třídy byly následně označeny anotacemi @OwnerRestrict a @RoleRestrict tak, aby konfigurace co nejvíce odpovídala reálnému použití.

Hotel

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

Ukázka 6.3: Zabezpečení třídy Hotel

```
@RoleRestrict(
    allowCreate = {"ADMIN"},
    allowUpdate = {"ADMIN"},
    allowDelete = {"ADMIN"}
)
public class Hotel extends TimestampedEntityObject
   implements Serializable
```

User

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

Ukázka 6.4: Zabezpečení třídy User

```
@RoleRestrict(
    allowRead = {"ADMIN"},
    allowCreate = {"ADMIN"},
    allowUpdate = {"ADMIN"},
    allowDelete = {"ADMIN"}
)
@OwnerRestrict(
    allowed = {OperationType.READ, OperationType.UPDATE}
)
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.

Ukázka 6.5: 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
```

Ukázka 6.6: 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;
```

6.3.4 Rozšíření knihovny

Abychom ukázali možnost rozšíření knihovny o vlastní autorizační podmodul, implementovali jsme vlastní podmodul *AttributeRestrictProcessor*, který není řízený anotacemi, ale konfigurací v xml. Jedná se o velmi jednoduchý podmodul, který na základě hodnoty atributu entitní třídy buď operaci nad jejími daty autorizuje, nebo zamítne. Z xsd byl pomocí JDK utility xjc vygenerován Java kód, který podmodul načítá pomocí JAXB[24].

Ukázka 6.7: Ukázka konfigurace implementovaného podmodulu

```
<rules
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="AttributeRestrictProcessor.xsd">
    <classes>
        <class
        className="org.jboss.seam.examples.booking.model.Hotel">
            <attributes>
                <attribute>
                    <name>deleted</name>
                    <comp>eq</comp>
                    <value>false</value>
                </attribute>
            </attributes>
        </class>
    </classes>
    <attributes>
    </attributes>
</rules>
```

Na ukázce 6.7 je vidět jednoduchá konfigurace podmodulu. V tomto případě je na instanci entitní třídy Hotel autorizován požadavek ve chvíli, kdy je její atribut deleted nastavený na false, tedy není označena jako smazaná. Registrace podmodulu vypadá následovně:

Ukázka 6.8: Registrace podmodulu po inicializaci knihovny

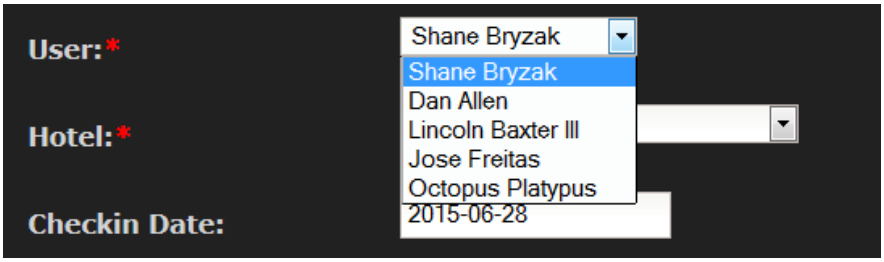
```
this.processor = new Processor(entityManager);
this.processor.addRestrictProcessor(
        new AttributeRestrictProcessor());
```

6.3.5 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.

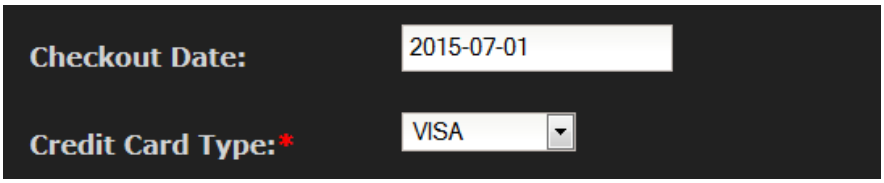


(a) Administrátor

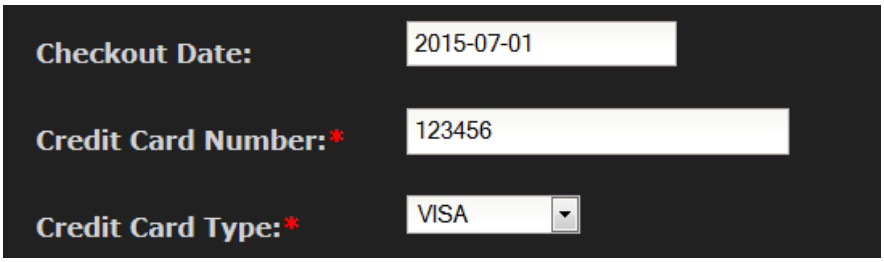
(b) Normální uživatel

Obrázek 6.4: 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 6.5: Rozdíl mezi administrátorem a normálním uživatelem - zabezpečení atributu

6.3.6 Použití knihovny mimo framework AspectFaces

Protože není autorizační knihovna nijak vázána na framework AspectFaces ani na projekt Seam, je možné ji využít v kterékoliv Java aplikaci. Proto byl vytvořen jednoduchý kontroler *SimpleController*, který toto demonstruje. Kontroler autentizuje uživatele pomocí Basic autentizace, neposkytuje podporu pro získání metadat, vrací pouze reprezentaci zdroje v reprezentaci JSON. Následující dvě ukázky ukazují dotaz na výpis všech uživatelů. První dotaz provedl normální uživatel, druhý dotaz provedl administrátor. Je vidět, že normální uživatel má oprávnění vidět pouze sám sebe, oproti tomu administrátor může vidět všechny uživatele.

Ukázka 6.9: Požadavek na vrácení všech uživatelů (User)

```
GET /seam-booking/ws/autoRest/org.jboss.seam.examples.
    booking.model.User HTTP/1.1
Host: localhost:8080
Authorization: Basic ZGFuOmxhdXJlbA==

HTTP/1.1 200 OK
Content-Type: application/json
[
    {..., "name": "Dan Allen", ...}
]
```

Ukázka 6.10: Požadavek na vrácení všech uživatelů (Admin)

```
GET /seam-booking/ws/autoRest/org.jboss.seam.examples.
    booking.model.User HTTP/1.1
Host: localhost:8080
Authorization: Basic c2hhbmU6YnJpc2JhbmU=

HTTP/1.1 200 OK
Content-Type: application/json
[
    {..., "name": "Shane Bryzak", ...},
    {..., "name": "Dan Allen", ...},
    {..., "name": "Lincoln Baxter III", ...},
    {..., "name": "Jose Freitas", ...},
    {..., "name": "Octopus Platypus", ...}
]
```

7 Testování

V rámci práce byly prováděny dva druhy testů. Byly implementovány jednotkové (unit) testy, které průběžně ověřovaly funkčnost knihovny, a na konci vývoje byly na vzorové aplikaci provedeny výkonnostní testy, tedy byla ověřena rychlost aplikace při použití zabezpečovací knihovny.

7.1 Unit testy

Unit testy jsou navrženy tak, aby ověřovaly jak činnost jednotlivých podmodulů, tak celkovou integraci hlavním modulem. Byl vytvořen testovací model, který odpovídá modelu vzorové aplikace Seam booking. Byla implementována testovací implementace rozhraní *DataHandler* (viz. Ukázka 5.1, která 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 bylo implementováno 18 testovacích metod.

7.2 Testy výkonnosti

Cílem těchto testů je ověřit, že aplikací zabezpečovací knihovny nedojde k výraznému zpomalení celé aplikace, což by prakticky znemožnilo její použití. Nejdříve je třeba prověřit dobu načítání vzorové aplikace bez jakýchkoliv úprav. Poté otestujeme dobu načítání po připojení frameworku AspectFaces a použití jeho distribuovaného rozhraní. Nakonec podrobíme testu aplikaci po připojení zabezpečovací knihovny.

Testy budou automatizovány pomocí javascriptu, který bude provádět předem daný počet obnovení HTML stránky vzorové aplikace, a časy načtení bude ukládat do cookies. Javascript bude muset vzít v úvahu i to, že nemůže brát načtení stránky jako *window.onload*, ale musí počítat i na dokončení případných javascriptových dotazů na server (distribuované rozhraní AspectFaces). Každý test se skládá ze tří iterací, každá z těchto iterací se provádí na jiném internetovém prohlížeči. Testovací prohlížeče jsou Google chrome v47.0, Mozilla Firefox 43.0.4, a Microsoft Internet Explorer 11.0. Každá z těchto iterací obsahuje 100x obnovení stránky.

Aplikace bude nasazena na aplikačním serveru JBoss AS 7.1.1[25], jako databázová vrstva byl zvolen PostgreSQL 9.3[26].

7.2.1 Samotná vzorová aplikace

První test se týká čisté vzorové aplikace. Jde tedy o test aplikace z oficiálního repozitáře Seam[8].

Prohlížeč	Počet pokusů	Minimum	Maximum	Průměr	Modus	Medián
Chrome	100	1032	1315	1074.65	1075	1068
Firefox	100	1170	1725	1264.14	1206	1231
IE	100	1039	1551	1190.56	1167	1180.5

Tabulka 7.1: Údaje získané v testu - vzorová aplikace (v ms)

7.2.2 Po aplikaci aspectFaces

Tento test proběhl po aplikaci frameworku AspectFaces a jeho distribuovaného rozhraní.

Prohlížeč	Počet pokusů	Minimum	Maximum	Průměr	Modus	Medián
Chrome	100	277	1241	403.2121	331	382
Firefox	100	214	1118	377.63	320	345
IE	100	349	835	501.15	443	477.5

Tabulka 7.2: Údaje získané v testu - AspectFaces (v ms)

7.2.3 Po aplikaci zabezpečení

Po připojení zabezpečovací knihovny vypadaly výsledky takto:

Prohlížeč	Počet pokusů	Minimum	Maximum	Průměr	Modus	Medián
Chrome	100	379	1544	514.13	501	475
Firefox	100	530	953	638.87	575	626
IE	100	417	912	574.25	579	567

Tabulka 7.3: Údaje získané v testu - Autorizační knihovna (v ms)

7.2.4 Porovnání výsledků

Pokud vezmeme mediány naměřených výsledků a porovnáme je, vznikne nám následující tabulka:

Prohlížeč	Vzorová aplikace	AspectFaces	Autorizační knihovna
Chrome	1068	382 [36 %]	475 [44 %]
Firefox	1231	345 [28 %]	626 [51 %]
IE	1180.5	477.5 [40 %]	567 [48 %]

Tabulka 7.4: Porovnání údajů[ms,%]

V tabulce 7.4 jsou shrnuté výsledky testování. Je zde vidět medián pro jednotlivé testy, u AspectFaces a autorizační knihovny je vidět informace o tom, kolik času trvalo vykreslení stránky oproti vzorové aplikaci. Po aplikaci AspectFaces je vidět značné zkrácení doby vykreslování (úspora času 60% - 70%). Je to dáno rozložením zátěže mezi klienta a server (server zpracovává data, klient generuje UI), klientská strana má také velkou výhodu v paralelním vykreslování dat do UI.

Po integraci autorizační knihovny je vidět zhoršení výkonu aplikace, úspora času oproti původní vzorové aplikaci je však stále okolo 50%.

8 Závěr

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 autorizačního podmodulu, případně i nahradit obě implementace obsažené v knihovně svými vlastními.

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 modul pak výsledky z jednotlivých procesorů spojí dohromady.

Nevýhoda současného řešení je, že potřebuje mít v každé entitní třídě identifikátor id typu Long, který reprezentuje primární klíč. 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. Toho se v současné chvíli dá dosáhnout pouze implementací jednoho podmodulu, který bude rozhodovat na základě více údajů.

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*

[9] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard *Web Services Architecture.* W3C Working Group Note 11, Únor 2004

[10] Tim Berners-Lee *The World Wide Web: Past, Present and Future.* Massachusetts Institute of Technology, Srpen 1996

[11] Oracle Corporation *Java API for RESTful Services (JAX-RS).* https://jax-rs-spec.java.net/

[12] Oracle Corporation *Jersey - RESTful Web Services in Java.* https://jersey.java.net/

[13] Ian Jacobs, Norman Walsh *Architecture of the World Wide Web, Volume One*. W3C Recommendation 15, Prosinec 2004

[14] Roy T. Fielding, Julian F. Reschke *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. http://tools.ietf.org/html/rfc7230, Červen 2014

[15] Roy T. Fielding, Julian F. Reschke *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. http://tools.ietf.org/html/rfc7231, Červen 2014

[16] Erik Wilde *Wilde's WWW: Technical Foundations of the World Wide Web*. International Computer Science Institute , Berkeley, 1999

[17] Oracle Corporation *Java Platform, Enterprise Edition*. https://docs.oracle.com

[18] Franks, et al. *RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication*. https://www.ietf.org/rfc/rfc2617.txt, Červen 1999

[19] Barry Leiba *OAuth Web Authorization Protocol*. Huawei Technologies, 2012

[20] RedHat *RESTEasy*. http://resteasy.jboss.org

[21] Restlet, Inc *Restlet Framework*. http://restlet.com/projects/restlet-framework/

[22] The Apache Software Foundation *Apache Maven*. https://maven.apache.org/

[23] The Project Lombok Authors *Project Lombok*. https://projectlombok.org/

[24] *Project JAXB*. https://jaxb.java.net/

[25] RedHat *JBoss Application Server*. http://jbossas.jboss.org/

[26] The PostgreSQL Global Development Group *PostgreSQL*. http://postgresql.org/

Slovník

AOP Aspect Oriented Programming. 27, 28

API Application Programming Interface. 18, 19

AST Abstract syntax tree. 42

CRUD Create, Read, Update, Delete. 36, 40, 54

EL Expression Language. 29

HTTP Hypertext Transfer Protocol. 1, 9

HTTPS Hypertext Transfer Protocol Secure. 1, 14

IP Internet Protocol. 12

Java EE Java Platform, Enterprise Edition. 15, 19

Java SE Java Platform, Standard Edition. 24

JAX-RS Java API for RESTful Services. 1, 18, 19, 22–24

JAXB Java Architecture for XML Binding. 50

JDK Java Development Kit. 50

JPA Java Persistence API. 36

JRE Java Runtime Environment. 41

JSF Aspect Oriented Programming. 29, 30

JSON JavaScript Object Notation. 52

JSP JavaServer Pages. 18

JSR Java Specification Requests. 18, 19

MIME Multipurpose Internet Mail Extensions. 9

POJO Plain Old Java Object. 19

REST Representational state transfer. 7, 18

SSL Secure Sockets Layer. 14

TLS Transport Layer Security. 14

UI User interface. 27–30, 44, 45, 56

URI Uniform Resource Identifiers. 7, 13

Web World wide web. 7

Přílohy

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