VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

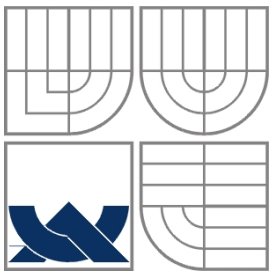
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYLEPŠENÍ PODPORY REST V JBOSSESB

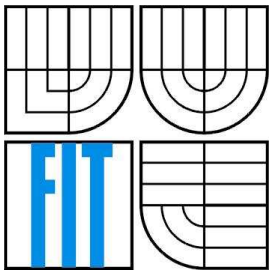
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Filip Eliáš
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYLEPŠENÍ PODPORY REST V JBOSSESB

IMPROVE SUPPORT FOR RESTFUL PROCESSING IN JBOSSESB

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR

Bc. FILIP ELIÁŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK LETKO

BRNO 2011

Abstrakt

Cílem této práce je umožnit JBoss ESB komunikovat se vzdálenými aplikacemi pomocí REST. JBoss ESB je platforma, která odděluje aplikační logiku od procesních funkcí a umožňuje komunikaci mezi aplikacemi s různým rozhraním pomocí zasílání zpráv. REST je architektura, která poskytuje obecné rozhraní pro komunikaci mezi počítačovými systémy v síti. Tato práce popisuje návrh a implementaci integrace REST do JBoss ESB. Integrace je rozdělená na dvě části. Vstupní část přijímá REST požadavky, propaguje jejich obsah do JBoss ESB a generuje odpovědi. Výstupní část čeká na zprávy z JBoss ESB, ze kterých vytvoří REST požadavky, zašle je adresátovi a přijme odpovědi. Při implementaci REST v JBoss ESB je kladen důraz na integraci s projektem RESTEasy. RESTEasy poskytuje framework, který ulehčuje vývoj aplikací podporujících REST komunikaci.

Abstract

The goal of this work is to allow JBoss ESB to communicate with remote applications using REST. JBoss ESB is a platform that separates application logic from the process functions and enables communication between applications with different interfaces by sending messages. REST is an architecture that provides universal interface for communication between computer systems in the network. This work describes integration of REST communication architecture with JBoss ESB platform. The integration is divided into two parts. The input part accepts REST requests, propagates their content to the JBoss ESB and generates a response. The output part waits for the messages from JBoss ESB from which it creates the REST requests, sends them to the recipients and receives responses. The implementation puts emphasis on integration with the RESTEasy project. RESTEasy provides a framework that facilitates development of applications that support REST communication.

Klíčová slova

Java, J2EE, REST, JBoss ESB, RESTEasy

Keywords

Java, J2EE, REST, JBoss ESB, RESTEasy

Citace

Bc. Filip Eliáš: Vylepšení podpory REST v JBossESB, diplomová práce, Brno, FIT VUT v Brně, 2011

Vylepšení podpory REST v JBossESB

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Zdeňka Letka
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Filip Eliáš
24.5.2011

Poděkování

Zde bych rád poděkoval Ing. Zdeňku Letkovi za pomoc s formálními aspekty řešení a Ing. Jiřímu Pechanci za pomoc s technickými problémy.

Obsah

1 Úvod

V dnešní době je internet velice rozšířen a jeho hlavní službou je World Wide Web (WWW), která používá protokol Hyper Text Transfer Protocol (HTTP), což je aplikační protokol pro přenos textových dat v síti. Díky tomu je protokol HTTP také velice rozšířen a podporován téměř všemi síťovými prvky. Jedním z autorů protokolu HTTP je Roy Fielding, který ve své disertační práci navrhl Representation State Transfer (REST). REST je architektura, která poskytuje obecné rozhraní pro vzdálené aplikace, které komunikují přes síť. Ačkoli REST není určen přímo pro HTTP, je téměř vždy spojován právě s protokolem HTTP.

Tato práce má za úkol vylepšit podporu REST v JBoss ESB. JBoss ESB je platforma, která poskytuje jednotné komunikační rozraní pro rozdílné aplikace a odděluje aplikační logiku od procesních funkcí. Pro definici rozhraní při komunikaci s různými aplikacemi bude využita právě architektura REST. Dalším cílem této práce je také integrovat JBoss ESB s projektem RESTEasy. Projekt RESTEasy obsahuje sadu frameworků, které ulehčují implementaci aplikací podporujících REST komunikaci.

Tato práce je rozdělena do 13 kapitol. V kapitolách 2 až 7 budou představeny použité technologie. V druhé kapitole bude popsána Java Enterprise Edition, zmíněny budou i její komponenty a vrstvy. Třetí kapitola pojednává o HTTP, protože se zabýváme architekturou REST přes HTTP, proto je potřeba pochopit, jak pracuje protokol HTTP a jaké má vlastnosti. Ve čtvrté kapitole bude popsána samotná architektura REST. Pátá kapitola obsahuje popis platformy JBoss ESB. V šesté kapitole bude popsána JAX-RS, což je specifikace programového rozraní pro vývoj serverových aplikací s architekturou REST v jazyce Java. Ve stejné kapitole bude také představen projekt RESTEasy, který implementuje JAX-RS specifikaci. V další kapitole bude popsána současná podpora architektury REST v JBoss ESB.

Implementační část této práce je popsána v kapitolách 8 až 12. V osmé kapitole budou uvedeny základní požadavky pro integraci REST do JBoss ESB. V deváté části bude vytvořen návrh integrace architektury REST pro JBoss ESB. Návrh je rozdělen na vstupní část, která přijímá REST požadavky do JBoss ESB, a výstupní část, která REST požadavky vysílá z JBoss ESB. V další části bude popsána implementace integrace REST do JBoss ESB, vycházející z návrhu v předešlé kapitole. Kapitola 11 obsahuje sadu příkladů, na kterých bylo testováno, zda implementace integrace REST do JBoss ESB odpovídá návrhu. Poslední kapitola před závěrem zhodnocuje výsledky této práce.

2 Java Enterprise Edition

Java je v současnosti jedním z nejpoužívanějších objektově orientovaných programovacích jazyků. Java byla představena firmou Sun Microsystems v roce 1995 a od té doby prošla výrazným vývojem a rozdělila se na různé platformy, které se zaměřují na vývoj aplikací pro různá prostředí. Kromě Java Enterprise Edition (Java EE), popsané v této kapitole, existují ještě následující Java edice:

- Standard Edition: Základní a nejstarší edice Javy. Tato edice obsahuje základní knihovny pro vývoj přenosných desktopových nebo serverových aplikací. Na této edici jsou založeny všechny ostatní edice Javy.
- Micro Edition: Tato edice slouží jako prostředí pro běh Java aplikací na zařízeních, které by nezvládly Java Standard Edition. Jsou to zařízení s omezenými prostředky, jako jsou mobilní telefony nebo PDA.
- Java Card: Poskytuje prostředí pro aplikace na čipových kartách a dalších zařízeních s velmi omezeným výkonem.

Java EE je ucelená kolekce technologií pro nasazení, vývoj a provoz podnikových aplikací napsaných v jazyce Java. Java EE obsahuje různé služby, rozhraní a protokoly, které umožňují vyvíjet distribuované, vícevrstvé a webové aplikace.[1]

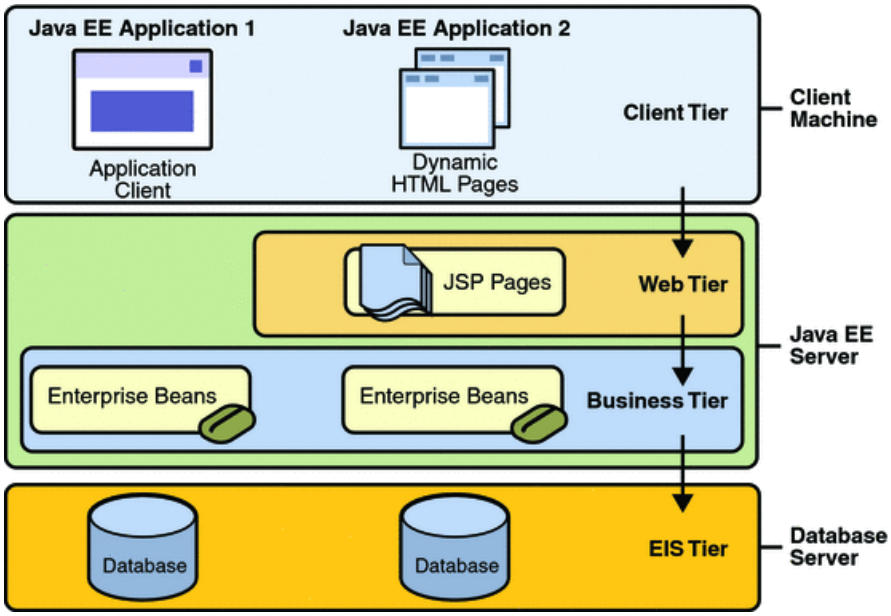
V následujících podkapitolách si vysvětlíme, co jsou to komponenty a vrstvy v Java EE a také si představíme vybrané technologie Java EE. Jelikož budeme v pozdějších kapitolách využívat Java anotací, tak si v poslední podkapitole vysvětlíme, co je to Java anotace a jak se používá.

2.1 Komponenty a vrstvy

Java EE aplikace je rozdělená do vrstev. Vrstva je oddělená logická část aplikace, která obsahuje komponenty podle její funkce. Na obrázku 2.1 je zobrazena typická vícevrstvá aplikace. Za klientskou vrstvu (client tier) považujeme tu, se kterou pracuje uživatel. Webová vrstva (web tier) generuje odpovědi na základě požadavků od klienta. Do webové vrstvy patří, např. Java Server Pages. Pro aplikační logiku a práci s daty slouží business vrstva (business tier), kde hlavním zástupcem jsou Enteprise JavaBeans. Enterprise Information System (EIS) vrstva slouží, např. k přístupu do databáze nebo k integraci s již existujícími systémy.

Komponenty jsou samostatné a znovupoužitelné funkční jednotky, které komunikují s ostatními komponentami. Java EE specifikace definuje tyto komponenty:

- Komponenty běžící u klienta: Slouží jako rozhraní pro uživatele. Může to být webový klient nebo aplikační klient.
- Webové komponenty: Vytváří obsah, který je poté předán klientovi. Jsou to, např. servlety či JSP stránky.
- Business komponenty: Zajišťují potřeby konkrétní aplikace. Přijímají a zpracovávají data od klientů. Ukládají informace do databáze.



Obrázek 2.1: Vícevrstvá aplikace. [31]

Běhovou podporu pro komponenty na serveru zajišťuje kontejner. Kontejner je rozhraní mezi komponentami a platformě závislým prostředím. Každá komponenta musí být vložena do kontejneru, aby mohla být spuštěna. Kontejner poskytuje komponentám také další služby, např. synchronizaci nebo transakce. [1, 2]

2.2 Vybrané technologie Java EE

V této kapitole krátce představíme vybrané technologie platformy Java EE.

2.2.1 Servlet

Servlet je komponenta běžící v rámci webového Java serveru. Servlety přijímají požadavky od webových klientů a generují odpovědi. Komunikace s klientem většinou probíhá pomocí HTTP. Požadavek mohou zpracovat samy nebo zavolat jinou komponentu, např. Enterprise JavaBean. Životní cyklus servletu je následující:

1. Servlet je načten webovým kontejnerem.
2. Kontejner vytvoří instanci servletu.
3. Kontejner inicializuje servlet.
4. V této fázi je servlet připraven přijímat a odpovídat na požadavky.
5. Pokud již servlet není potřeba, tak kontejner odstraní instanci servletu.

Nevýhodou servletu je, že není možné oddělit statickou část odpovědi od dynamické. Vše je uloženo v Java třídě, což je nevhodné z hlediska srozumitelnosti a udržovatelnosti aplikace. Další informace je možné najít ve specifikaci servletu. [3]

2.2.2 Java Server Pages

Java Server Pages (JSP) je technologie, která umožňuje snadný vývoj dynamického webového obsahu. Narozdíl od servletu, JSP odděluje uživatelské rozhraní od generování dynamického obsahu a tím umožňuje, např. měnit vzhled uživatelského rozhraní bez nutnosti měnit implementaci generování obsahu. Kód JSP stránky je také více srozumitelný a lépe udržovatelný než servlety.

JSP je složená z tagů a tzv. *scriptletů*. Tagy jsou nejčastěji značkovacími jazyky HyperText Markup Language (HTML) [4], nebo Extensible Markup Language (XML) [5], a scriptlety jsou oddělené části Java kódu, které jsou při požadavku vykonány a nahrazeny svým výstupem. Při prvním požadavku na zobrazení bude JSP stránka převedena na servlet. [6]

2.2.3 Enterprise JavaBeans

Enterprise JavaBeans (EJB) je komponenta, jenž implementuje aplikační logiku, která řeší běžné funkce, jakými jsou perzistence, bezpečnost nebo transakce. EJB jsou navrženy tak, aby byly znovupoužitelné. Programátor nemusí tedy řešit stále stejné problémy. EJB specifikace také definuje, jak má být EJB umístěno do EJB kontejneru. Existují tři typy EJB:

1. Bezstavové: Neuchovávají stav klienta mezi obsluhou jednotlivých požadavků.
2. Stavové: Uchovávají svůj stav mezi jednotlivými požadavky v rámci jednoho sezení.
3. Řízené zprávami: Jsou vyvolány zprávou místo metody.

Tyto a další informace je možné najít ve specifikaci EJB. [7]

2.3 Anotace v jazyce Java

Anotace jsou forma metadat, které je možné přidat do zdrojového Java kódu. Anotace umožňují připojit k třídě, metodě nebo proměnné nějakou informaci mimo běžný kód. V jazyce Java je možné používat specializované anotace, např. anotace `@deprecated` značí, že metoda by se neměla již používat. Od vydání JDK verze 1.5 je možné si definovat vlastní anotace a používat je. Anotace přímo neovlivňují sémantiku aplikace, ale mají vliv na to, jak je k aplikaci přistupováno jinými programy nebo knihovnami. Anotace se nejčastěji používají pro třídy, metody nebo atributy.

Díky anotacím není potřeba spolu s aplikací distribuovat také konfigurační soubory, které popisují, jak se aplikace bude chovat. Například u EJB verze 2.1 a nižší, je nutné spolu s EJB distribuovat také tzv. popisovač umístění (deployment descriptor), který popisuje, jak má být EJB umístěna na serveru. Od verze 3.0 již deployment descriptor není nutný, protože může být zastoupen anotacemi, které jsou součástí implementace samotné EJB. [8]

Pomocí anotace také můžeme injektovat parametry do metody. Při definici metody anotujeme její parametr nějakou anotací, a když pak budeme metodu volat pomocí balíku *java.lang.reflection*, tak podle typu anotace můžeme rozhodnout, jakou hodnotu vložíme za anotovaný parametr.

Anotace se definují podobně jako rozhraní. Na výpisu 2.1 je možné vidět definici anotace `@Kategorie`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
 public @interface Kategorie {
  int nazev();
  String popis() default "[zadny]"; }
```

Výpis 2.1: Definice anotace.

Anotace se definuje pomocí klíčového slova `@interface`. Poté je možné definovat metody, které představují parametry anotace. Metody nesmí mít žádné parametry a musí vracet primitivní datové typy, `String`, `Class`, výčtový typ nebo pole těchto typů. Anotace na výpisu 2.1 má tedy parametry `nazev` a `popis`. Parametr `popis` má navíc zadanou standardní hodnotu, která bude použita v případě, že tento parametr nebude uživatelem zadán. Anotace na výpisu 2.1 také obsahuje dvě metaanotace. Metaanotace jsou anotace, které anotují anotaci. Metaanotace `@Target` značí, jakému prvku je anotace určena. Druhá metaanotace `@Retention` určuje možnost přístupu k anotaci a má tři možné hodnoty:

- RetentionPolicy.SOURCE: Anotace je dostupná ve zdrojovém kódu. Překladačem je pak odstraněna. Používají se, např. pro potlačení varovných výpisu kompilátoru.
- RetentionPolicy.CLASS: Anotace je přístupná i po překladu. Tento typ anotací používají různé nástroje, které používají anotace pro generování kódu, např. XML souborů.
- RetentionPolicy.RUNTIME: K tomuto typu anotací je možné přistupovat i za běhu programu.

3 HyperText Transfer Protocol

HyperText Transfer Protocol (HTTP) je textový, aplikační protokol pro přenos dat na internetu. Určen je hlavně pro výměnu dokumentů ve formátu HTML. HTTP však umí přenášet také soubory, jakými jsou obrázky nebo zvuk. V současné době se běžně používá verze HTTP/1.1, která byla definována v RFC 2616. [9]

V následující podkapitole si uvedeme, jak probíhá komunikace pomocí HTTP protokolu, a popíšeme si HTTP požadavek a odpověď. V druhé podkapitole se seznámíme se základní autentizací klienta pomocí HTTP.

3.1 Komunikace

Protokol HTTP pracuje na principu Klient/Server. Uživatel (klient) většinou pomocí webového prohlížeče pošle požadavek na server. Server vytvoří odpověď a pošle ji zpět klientovi. Protokol HTTP je bezstavový, tedy pokud klient pošle dva dotazy na stejný server, je s nimi zacházeno jako se dvěma různými nezávislými požadavky. Server tedy neuchovává stav komunikace, což je nevýhodné pro složitější aplikace, kdy je potřeba uchovat stav komunikace s klientem na serveru, např. u přihlášení uživatele k určité webové aplikaci. [9]

Metoda	Popis
OPTIONS	Reprezentuje požadavek na informace o komunikačních možnostech serveru identifikovaném URL.
GET	Tato metoda slouží k příjmu objektu, který je identifikován URL, ze serveru, např. Obrázek, HTML text nebo XML.
POST	Tato metoda se používá k odeslání objektu v těle dotazu na server. Používá se nejčastěji pro odeslání dat z webových formulářů.
HEAD	Má stejnou funkci jako GET, ale server vrátí jen hlavičku odpovědi. Hlavička odpovědi by měla být identická, jako kdyby byla použita metoda GET.
PUT	Tato metoda slouží k tomu, aby objekt přiložený k požadavku byl uložen na serveru. Na rozdíl od metody POST, pokud pod daným URL již existuje nějaký objekt, je nahrazen objektem přijatým v požadavku.
DELETE	Metoda DELETE slouží ke smazání objektu identifikovaným URI v požadavku.
TRACE	Umožňuje sledovat cestu dotazu. Klient může zjistit co se na požadavku mění, když požadavek prochází přes různé servery.
CONNECT	Tuto metodu použije klient, aby nařídil proxy serveru, vytvořit spojení se vzdáleným serverem.

Tabulka 3.1: Metody HTTP/1.1.

HTTP dokáže přenášet různé soubory různých formátů díky Multipurpose Internet Mail Extensions (MIME), což je standard, který umožňuje v emailových zprávách zasílat text v jiném než standardním kódování. Hlavně dovoluje připojit ke zprávě soubory různých formátů. MIME je dnes využíváno nejen v emailové komunikaci, využívají jej i jiné protokoly, např. právě HTTP.

Pro identifikaci dokumentu na serveru slouží Uniform Resource Locator (URL). URL je řetězec znaků, který přesně identifikuje zdroj v síti. Na výpisu 4.1 je formát URL, kde první část identifikuje protokol, který bude pro přenos dat použit. *Host* je buď doménové jméno, nebo IP adresa serveru, na kterém se zdroj nachází. Dále následuje port, na kterém server přijímá požadavky. Za portem je cesta na serveru, na které se zdroj nachází. Na konci mohou být parametry, které jsou od zbytku URL odděleny otazníkem. Parametry slouží pro předání dat do webové aplikace.

```
Protokol://Host:Port/Cesta?Parametry
```

Výpis 3.1: Formát URL v HTTP.

3.1.1 HTTP Požadavek

Každý požadavek musí mít definovanou metodu (přehled metod je definován v tabulce 3.1). Metoda určuje, jak má server požadavek zpracovat a jakou odpověď má zaslat zpět klientovi. Za metodou následuje cesta k požadovanému zdroji, ta je následována verzí HTTP, která je v požadavku použita. Na dalších řádcích je pole hlaviček, tyto hlavičky dále upřesňují požadavek. Za tímto polem musí být prázdný řádek, za ním mohou být užitečná data od klienta. [9]

3.1.2 HTTP Odpověď

Na prvním řádku odpovědi je verze protokolu a stavového kódu, který určuje výsledek dotazu. Tabulka 3.2 zobrazuje některé stavové kódy. Kód začínající číslem 2 znamená úspěch požadavku. Pokud kód začíná trojkou, dává server klientu vědět, že klient musí provést další akce, aby získal požadovaný objekt. Když je na začátku kódu číslo 4, je chyba na straně klienta, pokud číslo 5 je problém na serveru. Na dalším řádku následuje pole hlaviček, které obsahují informace o obsahu odpovědi. Pole je jako u požadavku ukončeno prázdným řádkem. Na konci může být tělo odpovědi.[9]

Kód	Popis
200 OK	Dotaz byl obsloužen bez chyb. Pokud je to odpověď na požadavek GET, je v těle zprávy připojen požadovaný objekt.
201 Created	Výsledkem zpracování dotazu na serveru bylo vytvoření nového objektu, který je identifikován pomocí URL.
301 Moved Permanently	Oznámení že, objekt byl přesměrován na jiné URL. Klient se musí zeptat na URL, které je odesláno v hlavičce Location. Jedná se tedy o přesměrování.
400 Bad Request	Tímto kódem server oznamuje klientovi, že požadavek nemá správnou syntaxi.
401 Unauthorized	Server oznamuje, že obsloužení dotazu je podmíněno některým z identifikačních požadavků.
404 Not Found	Server oznamuje, že objekt, který dotaz požaduje, neexistuje.
405 Not Allowed	Oznámení, že metoda, která je uvedena v požadavku, není povolena pro dané URL.
500 Internal Server Error	Server tímto kódem dává vědět klientovi, že během zpracování požadavku došlo na serveru k chybě.

Tabulka 3.2: Stavové kódy odpovědi. [7]

3.2 Autentizace

HTTP obsahuje několik autentizačních mechanismů. Zde si uvedeme pouze nejjednodušší základní autentizaci pomocí hlavičky *WWW-Authenticate*. Pokud klient nezašle serveru požadavek s platnými přihlašovacími údaji, tak server jako odpověď na požadavek pošle kód 401 Unauthorized (Odpověd severu je na výpisu 3.2).

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Muj Server"
Content-Length: 0
```

Výpis 3.2: Příklad odpovědi serveru.

Klient (webový prohlížeč) následně zobrazí přihlašovací dialog, kde uživatel zadá své přihlašovací údaje. Klient pošle serveru nový požadavek s přihlašujícími údaji. Požadavek poté bude vypadat podle výpisu 3.3.

```
GET /tajneDokumenty/ HTTP/1.1
Host: www.httpwatch.com
Authorization: Basic aHR0cHdhdGNoOmY=
```

Výpis 3.3: Příklad požadavek klienta.

Klient posílá jméno a heslo nezašifrované v base64 kódování. Proto by vždy tato základní autentizace měla být použita ve spojení s HTTPS, což je rozšíření protokolu HTTP, které umožňuje zabezpečit spojení mezi klientem a serverem před odposloucháváním a podvržením dat. [10]

4 Přenos reprezentativního stavu

Přenos reprezentativního stavu, angl. Representational State Transfer (REST), je softwarová architektura pro distribuované systémy. REST byla definována Royem Fieldingem v jeho disertační práci v roce 2000. Roy Fileding je také spoluautor protokolu HTTP. REST není přímo specifikován pro jediný protokol, ale dnes se nejčastěji setkáme právě s REST přes protokol HTTP. V této diplomové práci se budeme zabývat právě jen REST ve spojení s HTTP. Použití REST je výhodné hlavně pro uniformní a snadný přístup ke zdrojům.

V následujících podkapitolách se seznámíme s podmínkami, které musí splňovat REST aplikace. Také si popíšeme, co je to zdroj v REST a jaké rozhraní je použito při komunikaci mezi klientem a serverem. Nakonec provedeme srovnání REST s konkurenčním protokolem SOAP.

4.1 Architektura

Architektura REST pracuje stejně jako HTTP systémem klient/server. Klient se pomocí URL dotazuje na nějaký zdroj a server odešle zpět klientu reprezentaci tohoto zdroje.

Tato architektura se nazývá Reprezentační (Representational), protože server vrací klientovi reprezentaci zdroje, který je identifikován pomocí URL. Klient se po získání zdroje dostane do jiného stavu, odtud název Stav (State). Klient se tedy může dostávat stále do jiných stavů podle toho, jaký server vrací zdroj (Přenos stavu).

Aby se aplikace mohla nazývat *RESTful*, musí být splněny určité podmínky. Při splnění těchto podmínek bude aplikace schopna využít všech služeb architektury REST. Architektura REST popisuje šest následujících podmínek:

- **Klient-Server:** Klienti jsou oddělení od serverů jednotným rozhraním. Klient nepotřebuje vědět o způsobu uložení dat na serveru, čímž se zvyšuje portabilita klienta. Zároveň server nezná podrobnosti o klientovi, např. uživatelské rozhraní nebo stav klienta.
- **Bezstavový:** Komunikace mezi klientem a serverem je omezena tím, že na serveru není ukládán žádný kontext klienta, a proto je potřeba, aby v každém požadavku klienta byly všechny informace potřebné k obsloužení požadavku.
- **Kešovatelný**: Odpověď pro klienta může být označena jako kešovaná, což informuje klienta o možné neaktuálnosti nebo neplatnosti odpovědi.
- **Vrstvený systém**: Zapouzdření každé komponenty tak, že každá komponenta ví pouze o té, se kterou spolupracuje. Díky tomu jsou sousední vrstvy mezi sebou relativně nezávislé.
- **Jednotné rozhraní**: Uniformní rozhraní zjednodušuje a vytváří nezávislé vazby mezi klientem a serverem. Díky tomu se mohou klient a server vyvíjet nezávisle na sobě.
- **Kód na požádání (Code-on-demand)**: Tato podmínka je volitelná.n Umožňuje klientovi stáhnout a vykonat kód, jako např. Java Applety. Kód na požádání není povinná podmínka. [11]

4.2 Zdroje

Každá informace, která může být pojmenována, může být v architektuře REST zdrojem. Za zdroj lze považovat obrázek, dokument, službu nebo kolekci těchto elementů. Tato obecná definice zahrnuje mnoho zdrojů informací, aniž by je nějak dělila podle typu nebo implementace.

Reprezentace zdroje je obvykle dokument, např. HTML, XML, který zachycuje současný stav zdroje. Server posílá reprezentaci zdroje podle vlastností požadavku. To je výhodné, protože se odkazuje na zdroj a neodkazuje přímo na reprezentaci. Když je potřeba změnit nebo přidat reprezentaci zdroje, nemusí se měnit odkaz na zdroj. Každý zdroj musí být jednoznačně identifikován. Protože se zabýváme pouze REST přes HTTP, je zdroj identifikován pomocí URL. [12]

4.3 Rozhraní

Jak již bylo řečeno v kapitole 4.1, všechny zdroje mají jednotné rozhraní. Rozhraní je tvořeno pomocí několika metod, které jsou převzaty z protokolu HTTP. Těmto metodám se také říká slovesa, protože říkají poskytovateli služby, co má se zdrojem dělat. Metody se nazývají GET, POST, PUT, DELETE a HEAD. V následujících odstavcích si popíšeme jednotlivé metody.

Metoda GET se používá pro získání určité reprezentace zdroje. Typ reprezentace se určí podle hlavičky atributu. Metoda GET podporuje efektivní kešing, a proto může požadavek s metodou GET obsloužen keš serverem. Metoda GET je idempotentní, což znamená, že můžeme požadavek bez problémů použít vícekrát za sebou. Například pokud bychom poslali požadavek s metodou GET a nedorazila nám odpověď, tak nemůžeme vědět, zda požadavek dorazil na server, nebo se pouze ztratila odpověď ze severu. Protože je metoda GET idempotentní, můžeme požadavek jednoduše vyslat znovu.

Metoda POST vytvoří nový zdroj na serveru. V těle požadavku by měla být reprezentace, ze které se vytvoří nový zdroj. Server uloží zdroj, např. do databáze, a přiřadí nové URL zdroji. Pokud byla operace metody POST úspěšná, tak server vrátí spolu se stavovým kódem také URL nově vytvořeného zdroje. Metoda POST není idempotentní. Pokud pošleme požadavek a nepřijde nám odpověď, tak již nemůžeme poslat stejný požadavek znovu, neboť pokud by dorazily na server oba požadavky, mohlo by dojít například k nekonzistenci, kdy by došlo k vytvoření dvou zdrojů místo jednoho.

Metoda PUT je velmi podobná metodě POST. Metoda PUT stejně jako POST vytváří nový zdroj v systému, ale pokud pod uvedeným URL již existuje nějaký zdroj, provede PUT aktualizaci tohoto zdroje. Metoda PUT je idempotentní, protože další požadavky se stejným URL, způsobí jen přepsání původního zdroje. PUT by měla mít v těle požadavku reprezentaci, pomocí které se aktualizuje původní zdroj.

Metoda DELETE smaže zdroj, který identifikuje URL v požadavku. Metoda DELETE je idempotentní, protože pokud zdroj jednou smažeme a pošleme znovu stejný požadavek, tak stav systému se nezmění, jelikož zdroj bude stále smazaný.

Poslední metodou je HEAD, která je podobná GET. Tato metoda nevrací reprezentaci zdroje, ale pouze HTTP hlavičku. Používá se, pokud potřebujeme vědět, např. byl-li modifikován zdroj nebo jeho velikost. Metoda HEAD má stejné vlastnosti jako metoda GET. [13]

4.4 Srovnání REST a SOAP

Simple Object Access Protocol (SOAP) je protokol pro volání vzdálených metod a aplikací na serveru. SOAP používá jako transportní protokol HTTP. Přenáší zprávy ve formátu XML, které popisují možnosti aplikací a metod a tím umožňuje ostatním aplikacím je využívat.

REST a SOAP se liší hlavně v definici rozhraní a uložení užitečných dat ve zprávě. REST může jednoduše využívat keš a proxy serverů, zatímco u SOAP je použití těchto serverů náročnější. SOAP podporuje řadu užitečných funkcí, např. zabezpečení, transakce, spolehlivou komunikaci, ale u REST je použití těchto funkcí závislé na klientovi a serveru. Výše uvedené rozdíly si blíže představíme v následujících odstavcích.

REST používá rozhraní, které je dobře známé a používané. Tím je URL a metody HTTP protokolu GET, POST, PUT, DELETE a HEAD. U architektury REST je definice metod pevně dána podle protokolu HTTP, zatímco u SOAP je možno si definovat vlastní SOAP metody. REST vhodně využívá vlastností protokolu HTTP naproti SOAP, který používá HTTP v zásadě jen jako transportní protokol. Dalším rozdílem je to, že SOAP je pevně definovaný protokol a REST je architektura.

U REST jsou data uložena buď přímo do hlavičky nebo do těla požadavku. SOAP zabalí požadavek do SOAP obálky, která má tvar XML dokumentu podle SOAP formátu, což více zatěžuje prostředky počítačové sítě. U SOAP nastávají potíže, pokud je potřeba přenést jiný formát, než kterým je XML. SOAP vytvoří XML dokument a teprve uvnitř bude jiný formát.

Díky tomu, že REST používá HTTP, může lépe využít proxy serverů, protože mohou analyzovat požadavek podle URL nebo metody. U SOAP je využití proxy složitější, protože proxy server standardně nerozumí SOAP zprávám. Je tedy potřeba, aby byl proxy server o podporu SOAP rozšířen.

SOAP také nemůže využít kešování, protože všechny požadavky jsou zasílány metodou POST a keš server nemůže zjistit, zda jsou nějaká data požadována nebo posílána. Ani nemůže zjistit, jaký zdroj je odkazován.

Na druhou stranu SOAP podporuje WS-Security, která například umožňuje identifikaci pomocí prostředníka nebo standardní implementaci integrity a důvěrnosti. SOAP také podporuje transakce, které jsou implementovány i v REST. Tyto transakce v REST plně nesplňují ACID (atomicity, consistency, isolation, durability) [14], zatím co SOAP plně ACID podporuje.

SOAP také podporuje spolehlivou komunikaci mezi klientem a serverem, protože má integrované opakované posílání nedoručených zpráv. Zatímco u architektury REST si klient musí sám poradit s chybami v komunikaci. [12, 13, 15, 30]

5 Java rozhraní pro RESTful webové služby

Java rozhraní pro RESTful webové služby angl. Java API for RESTful web services (JAX-RS) je serverové rozhraní, které velmi ulehčuje vývoj RESTful webových aplikací v jazyce Java. Vývoj JAX-RS byl zahájen v roce 2007 s cílem vytvořit flexibilní a lehce použitelné rozhraní, které by vývojáře webových aplikací podporovalo ve vývoji RESTful aplikací. JAX-RS verze 1.0 byla dokončena v říjnu 2008 a setkala se s velmi kladným ohlasem od vývojářů. Od verze 1.1 je JAX-RS součástí platformy Java EE. V současnosti již existuje několik implementací tohoto rozhraní, např. Jersey, RESTEasy nebo Apache CXF.

V roce 2011 byly zahájeny práce na JAX-RS 2.0, které bude obsahovat i klientské API. Většina dnešních implementací JAX-RS do určitého stupně klientské API podporuje, ale současná verze JAX-RS 1.1 klientské API nepopisuje. Další novinkou bude podpora pro architekturu Model-View-Conroller (MVC).[16]

JAX-RS používá anotace ke směrování klientských požadavků do odpovídajících metod Java tříd a deklarativně mapuje požadavky na parametry těchto metod. Anotace se také používají pro definici statických metadat pro vytváření odpovědí. JAX-RS poskytuje klasické třídy a rozhraní pro dynamický přístup k datům požadavků a k vytváření odpovědí. [17]. Podle specifikace JSR 311 je rozhraní JAX-RS:

- **HTTP centrické**: Protokolem pro přenos dat je HTTP. Rozhraní JAX-RS poskytuje mapování mezi HTTP, URI a korespondujícími API třídami a anotacemi. Rozhraní také poskytuje podporu pro běžné způsoby využití HTTP a je dostatečně flexibilní, aby podpořilo různé aplikace HTTP.
- **Nezávislé na formátu dat**: Rozhraní je aplikovatelné na širokou škálu různých formátu těla HTTP zprávy.
- **Nezávislé na kontejneru**: JAX-RS je možné umístit do mnoha různých kontejnerů.

V dalších podkapitolách si představíme jednotlivé prvky v JAX-RS. V podkapitole 5.1 si vysvětlíme pojem zdrojová třída a popíšeme si, jak musí vypadat její konstruktor. Podkapitola 5.2 obsahuje definici zdrojové metody a následující podkapitola vysvětluje, které typy může zdrojová metoda vracet. V podkapitole 5.4 je popsána anotace `@Path`, která může patřit buď zdrojové třídě, nebo zdrojové metodě. Podkapitola 5.5 obsahuje popis parametrů zdrojových metod a anotací, které mohou tyto parametry anotovat. V další podkapitole si představíme anotace pracující s MIME typem REST požadavku a odpovědi. V podkapitole 5.7 bude představena specifikace JAXB, kterou musí každá implementace JAX-RS podporovat. Na konec si popíšeme projekt RESTEasy, což je implementace JAX-RS. Informace uvedené v této kapitole je možné najít ve specifikaci JAX-RS [18].

5.1 Zdrojová třída a její konstruktory

Zdrojovou třídou v JAX-RS budeme nazývat takovou třídu, která používá JAX-RS anotace, aby implementovala určitý webový zdroj. Standardně se pro každý požadavek vytvoří nová instance zdrojové třídy. Nejdříve je zavolán konstruktor, a poté se provedou injekce parametrů metod. Následně je vyvolána samotná metoda obsluhující požadavek. Po zpracování požadavku je instance třídy odstraněna.

Kořenová zdrojová třída je taková zdrojová třída, která je anotována anotací `@Path` (viz. kapitola 6.2). Kořenová zdrojová třída poskytuje kořen pro strom zdrojových tříd a umožňuje přístup do svých podtříd.

Konstruktor pro kořenovou zdrojovou třídu je volán běhovým prostředím JAX-RS, a proto musí obsahovat jen takové parametry, které je běhové prostředí JAX-RS schopno naplnit (např. i konstruktor bez parametrů). Pokud je více konstruktorů vhodných pro volání, je vybrán ten, který má nejvíce různých parametrů. Pokud konstruktory mají stejný počet parametrů, tak JSR 311 nestanovuje přesně pořadí, ale mělo by být na serveru logováno varování. Konstruktory nekořenových tříd jsou volány aplikací, a proto pro ně neplatí žádné speciální požadavky.

5.2 Zdrojové metody

Zdrojové metody jsou metody zdrojových tříd, které jsou anotovány jednou z anotací mapující Java metody na HTTP metody. JAX-RS obsahuje pro tento účel 5 anotací: `@GET, @POST, @PUT, @DELETE a @HEAD.`

Každá metoda může být anotována pouze jednou z těchto anotací. Protokol HTTP se stále vyvíjí a existuje řada jeho rozšíření, jako např. WebDAV, které umožňuje vytvářet, mazat a přesouvat dokumenty mezi webovými servery. WebDAV k tomu využívá sadu vlastních HTTP metod, např. MOVE. Proto JAX-RS umožňuje si definovat vlastní anotace, které budou mapovat jiné HTTP metody, než ty výše uvedené. Na výpisu 5.1 můžeme vidět zdrojový kód námi vytvořené anotace MOVE, která může sloužit pro mapování HTTP požadavků obsahujících MOVE metodu.

```
...
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod(“MOVE“)
public @interface MOVE {
}
```

Výpis 5.1: Zdrojový kód anotace MOVE.

Na anotacích definujících HTTP metodu není důležité jejich jméno, ale jak můžeme vidět na výpisu 5.1, je to metaanotace `@HttpMethod`, jejíž hodnota určuje, kterou HTTP metodu bude anotace mapovat. Na výpisu 5.1 je také metaanotace `@Target`, jenž určuje, kterému elementu je anotace určena. Na výpisu 5.2 si můžeme všimnout, že anotaci z výpisu 5.1 můžeme použít stejně jako standardní anotace definující HTTP metodu. Každá zdrojová metoda by měla být typu public a aplikační server by měl vypsat varování, pokud je metoda typu *private* nebo *protected*.

```
@MOVE
public void MoveDoc() {
...//zdrojový kód metody
}
@GET
public void getDoc() {
...//zdrojový kód metody
}
```

Výpis 5.2: Příklad použití anotace MOVE.

5.3 Návratové hodnoty metod

Zdrojové metody mohou vracet jen některé návratové typy, protože JAX-RS implementace musí vědět, jak z návratové hodnoty zdrojové metody vytvořit HTTP odpověď. V této podkapitole si uvedeme, které datové typy mohou zdrojové metody vracet.

Zdrojové metody mohou vracet datový typ `void`, `StreamingOutput`, `Response` nebo také jiný datový typ. Pokud zdrojová metoda vrací typ `void`, tak odpověď bude mít prázdné tělo a bude vracet vždy stavový kód 204 „No Content“. Kód 204 říká klientovi, že požadavek proběhl v pořádku, ale v těle odpovědi nejsou žádná data.

Metoda také může vracet objekt typu `StreamingOutput`, který je součástí JAX-RS. Jelikož je StreamingOutput pouze rozhraní, je potřeba implementovat jeho jedinou metodu *`void`* `write`*`(OutputStream output)`*, např. podle výpisu 5.3. Jestliže během provádění metody nedojde k žádné výjimce, tak tělo odpovědi bude obsahovat data zapsaná do objektu `StreamingOutput` metodou `write` a stavový kód odpovědi bude 200 „OK“.

```
@GET
StreamingOutput getData() {
 return new StreamingOutput() {
    public void write(OutputStream output) {
         output.write("Telo odpovedi".getBytes());
    }};
}
```

Výpis 5.3: Příklad vytvoření těla odpovědi pomocí StreamingOutput.

Objekt typu `Response`, který je také součástí JAX-RS, může být zdrojovou metodou vrácen, pokud je potřeba mít větší kontrolu nad odpovědí. `Response` je abstraktní třída obsahující tři metody. Metoda `getEntity()` vrací objekt, který chceme konvertovat do těla odpovědi. Druhá metoda `getStatus()` slouží pro získání stavového kódu odpovědí. A metodu `getMetadata()` vrací hlavičky, které chceme mít v odpovědi. Instanci `Response` nemůžeme vytvořit přímo, ale je potřeba použít třídu `ResponseBuilder`, která vytvoří instanci `Response`. Objekt `Response` standardně vrací stavový kód 200 „OK“, pokud uživatel nestanoví jinak. Pro definici stavového kódu odpovědi je určen výčtový typ Status, jak je možné vidět na výpisu 5.4.

Pokud metoda vrátí typ NULL, tak bude vrácen stavový kód 204 „No Content“. Tělo odpovědi zůstane prázdné. Je možné také vracet další Java objekty, např. `String` nebo JAXB objekt (viz. 5.7). Odpověď je také možné vytvořit vyvoláním výjimky `WebApplicationException`, kterou můžeme použít, jestliže nastane nějaká neočekávaná událost při provádění metody. `WebApplicationException` je možné předat objekt `Response`, ze kterého chceme vytvořit

odpověď nebo výjimce můžeme předat jen stavový kód a tělo odpovědi zůstane v tomto případě prázdné. Implementace JAX-RS vyvolanou výjimku odchytí a zavolá metodu `getResponse()`, která vrátí objekt `Response`, ze kterého bude vytvořena odpověď. Pokud výjimce nepředáme objekt `Response` ani stavový kód, zůstane tělo odpovědi prázdné a do hlavičky bude vložen stavový kód 500 „Internal Server Error“.

Kromě `WebApplicationException` je možné ve zdrojových metodách vyvolávat i jiné výjimky, jenž jsou pak zpracovány kontejnerem, ve kterém webová aplikace běží. To není příliš vhodné z hlediska flexibility a zabalit všechny ostatní výjimky do `WebApplicationException` není také příliš vhodné. Proto je možné si definovat tzv. *mapovače výjimek*, které dokáží mapovat výjimky na `Response` objekty. Nejprve je potřeba implementovat rozhraní `ExceptionMapper` a jeho metodu `toResponse(E exception)`, např. podle výpisu 5.4. Námi vytvořený mapovač musí být anotován anotací `@Provider`, která řekne implementaci JAX-RS, že třída je komponentou JAX-RS. A tím tedy při vyvolání výjimky `MyException` z výpisu 5.4, bude implementací JAX-RS zavolán mapovač výjimky `MyExceptionMapper`, který vrátí `Response` objekt se stavovým kódem 204 „Created“.

```
@Provider
    public class MyExceptionMapper
                implements ExceptionMapper<MyException> {
      public Response toResponse(MyException e) {
         return Response.status(Response.Status.CREATED).build();
      }
    }
```

Výpis 5.4: Implementace rozhraní ExceptionMapper.

5.4 Anotace @Path

Anotace `@Path` může náležet buď třídě, nebo metodě. Pokud `@Path` náleží třídě, označuje tuto třídu jako kořenovou zdrojovou třídu a určuje, na jaké URL se bude aplikace nacházet. Anotace `@Path` má jeden parametr, který určuje URL relativní cestu k webové aplikaci. Tedy pokud webová aplikace s kořenovou zdrojovou třídou z výpisu 5.5 běží na serveru, který je na adrese *http://Muj.server.cz*, tak výsledná adresa webové aplikace bude *http://Muj.server.cz/zajmy*.

```
...
@Path(/zajmy)
public class myRootClass{
...
```

Výpis 5.5: Kořenová zdrojová třída.

V případě, že anotace `@Path` patří ke zdrojové metodě, označuje relativní URL adresu, na které může být metoda volána. Výsledné URL zdrojové metody je konkatenace adresy serveru, relativní adresy kořenové zdrojové třídy a hodnoty anotace `@Path` zdrojové metody. Pokud zdrojová metoda není anotována anotací `@Path`, bude se nacházet na URL kořenové zdrojové třídy.

Anotace `@Path` může také obsahovat regulární výraz, který označuje relativní URL zdrojové metody. Takový regulární výraz se označuje jako šablona. Zdrojová metoda je poté volána vždy, když je URL požadavku kompatibilní s šablonou v její anotaci `@Path`. Jestliže v kořenové zdrojové třídě

z výpisu 5.5 bude existovat zdrojová metoda s anotací `@Path("clanek/{name : .*}/")`, pak bude kompatibilní s relativním URL požadavku */zajmy/clanek/auta/*, ale již nebude zavolána při požadavku na */zajmy/auta/*.

V anotaci `@Path` je také možné si pomocí šablony pojmenovat část URI cesty, což můžeme vidět na výpisu 5.6. Šablona je uzavřená ve složených závorkách a skládá se z dvou částí oddělených dvojtečkou. První část je identifikátor šablony, druhá část je regulární výraz, který může reprezentovat jakoukoliv posloupnost znaků neobsahující znak lomítko. Pomocí identifikátoru šablony pak můžeme odkazovat na část URL cesty, která je reprezentována regulárním výrazem. Tedy v případě, že bude přijat požadavek s relativním URL *clanek/hobby/nazev*, který bude zpracován metodu na výpisu 5.6, bude šablona `kategorie` odkazovat na řetězec `hobby`. Pro injekci hodnoty šablony do zdrojové metody slouží anotace `@PathParam`, která bude popsána v následující kapitole.

```
@GET
@Path(clanek/{kategorie :  .+}/nazev)
public getClanek(@PathParam("kategorie") String type){
.....
```

Výpis 5.6: Pojmenovaná část URI cesty.

5.5 Parametry zdrojových metod

Implementace JAX-RS musí před voláním zdrojové metody inicializovat její parametry. Proto se parametry funkcí musí držet určitých pravidel, aby implementace JAX-RS věděla, jak je správně inicializovat.

Pokud chceme mít přístup k tělu požadavku, můžeme zdrojové metodě definovat parametr `java.io.InputStream` nebo `java.io.Reader`, implementace JAX-RS se již postará o vytvoření jejich instance a naplnění daty. `InputStream` a `Reader` přistupují k tělu požadavku na nízké úrovni a je tedy vhodné je použít pokud v těle požadavku budou binární data nebo obrázky. Většina HTTP požadavků obsahuje v těle pouze text a proto je možné pro přístup k tělu požadavku využít také datový typ `String` nebo `char[]`.

Pro přístup k informacím v hlavičce požadavku je potřeba použít jednu z anotací popsaných v tabulce 5.1.

Anotace	**Popis**
@MatrixParam	Vrátí hodnotu maticového parametru http požadavku.
@QueryParam	Vrátí hodnotu Query parametru http požadavku.
@PathParam	Vráti hodnotu pojmenované URI cesty http požadavku.
@CookieParam	Vrátí hodnotu cookie http požadavku.
@HeaderParam	Vrátí hodnotu parametru hlavičky http požadavku.

Tabulka 5.1: Anotace pro přístup k hlavičce požadavku.

Všechny anotace z tabulky 5.1 mohou anotovat parametry zdrojové metody. Pokud je volána zdrojová metoda, jejíž parametr je anotován některou z uvedených anotací, tak implementace JAX-RS nalezne příslušný prvek v požadavku a předá jej funkci jako parametr. Anotace z tabulky 5.1

mohou anotovat jen parametry určitých datových typů, protože implementace JAX-RS musí vědět jak datový typ inicializovat. Povolené datové typy jsou [18]:

1. Primitivní typy
2. Typy, které mají konstruktor s jedním parametrem typu `String`
3. Typy, které mají statickou metodu `valueOf` s jedním parametrem typu `String`
4. Typy `List<T>`, `Set<T>` a `SortedSet<T>`, kde T uspokojuje 2. nebo 3. výše

Na výpisu 5.7 je příklad použití anotace `@QueryParam`, která je použita pro předání Query parametru `id` do parametru zdrojové metody `number`. Při předávání hodnoty z požadavku do funkce je nutné, aby implementace JAX-RS hodnotu konvertovala podle typu parametru funkce. Tedy hodnotu Query parametru na výpisu 5.7 je nutné nejprve konvertovat na datový typ `Integer`. Pokud není možné konverzi provést (např. query parametr `id` nebude číslo), tak je požadavek klienta chybný a bude mu zaslána odpověď 404 „Not Found“.

```
@GET
public OutputStream getClanek(@QueryParam(“id”) int number){
...
```

Výpis 5.7: Příklad použití anotace @QueryParam.

Anotace `@PathParam` slouží k injekci hodnoty šablony v anotaci `@Path` do zdrojové metody. Anotaci `@PathParam` je potřeba předat jeden parametr, kterým je identifikátor šablony, jak je vidět na výpisu 5.6. Pokud tělo HTTP požadavku obsahuje data z HTML formuláře, můžeme je injektovat do zdrojové metody pomocí anotace `@FormParam`.

Některé parametry HTTP požadavku, které injektujeme do zdrojové metody, mohou být volitelné. Jestliže klient takovéto parametry neposkytne, tak implementace JAX-RS místo nich vloží standardní hodnoty. Pro primitivní datové typy to je nulová hodnota a pro objekty je to NULL. Pomocí anotace `@DefaultValue` je možné si definovat vlastní hodnotu, která bude funkci předána, v případě že hodnota parametru nebude dodána klientem. Příklad použití `@DefaultValue` je na výpisu 5.8, na kterém je parametr `number` anotovaný anotacemi `@DefaultValue` a `@MatrixParam`. Pokud by v HTTP požadavku nebyl maticový parametr `id` přítomen, bude pro naplnění parametru `number` použit parametr anotace `@DefaultValue`, tedy 1.

```
@POST
public void setClanek(@DefaultValue(“1”) @MatrixParam(“id”) int
number){
...
```

Výpis 5.8: Příklad použití anotace @DefaultValue.

5.6 Anotace pracující s hlavičkou Content-Type

Aby aplikace mohla správně zpracovat HTTP požadavek, musí znát typ dat, které jsou přenášeny v těle HTTP požadavku. Typ přenášených dat je v HTTP protokolu uložen pod hlavičkou *Content-Type* pomocí MIME (Multipurpose Internet Mail Extension).

Jestliže chceme, aby zdrojová metoda byla zavolána, jen pokud tělo HTTP požadavku obsahuje určitý typ dat, musíme metodu anotovat pomocí anotace `@Consumes`. Do anotace vložíme MIME typy, které bude zdrojová metoda podporovat. Implementace JAX-RS poté ověří, zda je typ uvedený v anotaci `@Consumes` kompatibilní s typem uvedeným v hlavičce HTTP požadavku. Použití `@Consumes` můžeme vidět na výpisu 5.9, kde metoda `setClanekXML()` bude zavolána, jen pokud bude HTTP požadavek obsahovat metodu POST a zároveň bude mít v hlavičce Content-Type MIME typ kompatibilní s typem `application/xml`. Pro nastavení hlavičky Content-Type v HTTP odpovědi slouží anotace `@Produces`. Této anotaci předáme typ MIME, který bude poté vložen do odpovědi.

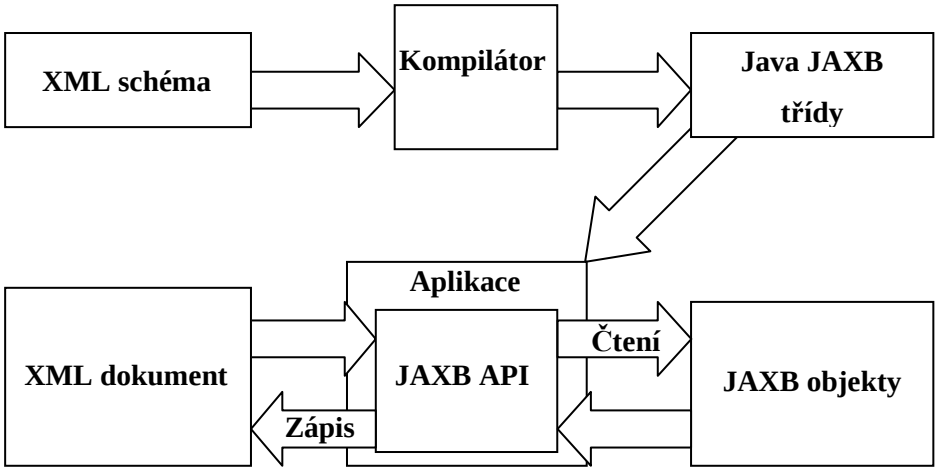
```
@POST
@Consumes(“application/xml”)
public void setClanekXML(InputStream body){
...
```

Výpis 5.9: Použití anotace @Consumes.

5.7 Java Architecture for XML Binding

Java Architecture for XML Binding (JAXB) je specifikace umožňující snadnou konverzi Java objektů do XML a naopak. Ačkoliv specifikace JAX-RS nedefinuje JAXB jako takové, každá implementace JAX-RS musí konverzi pomocí JAXB podporovat. JAXB má dvě hlavní komponenty, kompilátor a JAXB API. Kompilátor transformuje XML schéma na hierarchii Java JAXB tříd, jenž odpovídá struktuře tohoto XML schématu. XML schéma je specifikace, která určuje, jaké komponenty se mohou vyskytovat v XML dokumentu a vztahy mezi jednotlivými komponentami. Například může určit, které elementy se mohou vyskytovat v XML dokumentu a jaké budou mít atributy.

JAXB třídy jsou anotovány speciálními anotacemi, pomocí kterých pak může JAXB API zpracovat XML dokumenty. JAXB API poskytuje rozhraní pro čtení (unmarshalling) a vytváření (marshalling) XML dokumentů. JAXB API dokáže transformovat hierarchii JAXB objektů na XML dokument a naopak. Na obrázku 5.1 je zobrazena architektura JAXB. Nejprve pomocí kompilátoru vytvoříme z XML schématu JAXB třídy. JAXB třídy pak můžeme pomocí JAXB API a vstupního XML dokumentu převést na JAXB objekty, nebo naplněný JAXB objekt převést na XML dokument.



Obrázek 5.1: Architektura JAXB. [19]

JAXB třída je označena anotací `@XmlRootElement`, která má dva volitelné parametry `name` a `namespace`. Podle parametru `name` bude pojmenován kořenový element XML dokumentu a parametr `namespace` definuje jmenný prostor. Pokud parametr `name` není definován programátorem, bude mít kořenový element XML dokumentu název podle jména třídy.

Pomocí anotace `@XmlAttribute` označujeme atributy v JAXB třídě, které budou mapovány jako atribut v XML dokumentu. Stejně jako v předcházející anotaci, můžeme definovat jméno a jmenný prostor XML atributu. Anotace `@XmlAttribute` také obsahuje parametr `required`, kterým označuje, zda je tento atribut povinný nebo volitelný. Všechny parametry anotace `@XmlAttribute` jsou volitelné.

Pro mapování atributu v JAXB třídě na element v XML dokumentu použijeme anotaci `@XmlElement`. Tato anotace má stejné parametry jako `@XmlAttribute`, k tomu navíc je možné definovat standardní hodnotu elementu pomocí parametru `defaultValue`. Také je možné definovat, zda hodnota elementu může být NULL.

Na výpisu 5.10 můžeme vidět část JAXB třídy `Clanek`, která obsahuje dva atributy `nazev` a `textClanku`. Atribut `nazev` bude mapována jako atribut kořenového elementu `katalog`. Element katalog bude mít jeden podelement s názvem `text`, ve kterém bude obsah atributu `textClanku`.

```
@XmlRootElement(name="katalog")
public class Clanek{
   @XmlAttribute
   private int nazev;
   @XmlElement(name="text")
   private String textClanku;
   public int getNazev(){ return this.nazev; }
   public void setNazev(String nazev){ this.nazev = nazev; }
```

Výpis 5.10: Část JAXB třídy.

5.8 Projekt RESTEasy

RESTEasy je projekt společnosti Red Hat, který poskytuje různé frameworky pro vytváření RESTful webových aplikací v jazyce Java. Jedná se o implementaci specifikace JAX-RS. RESTEasy pro svůj běh potřebuje servletový kontejner. Nejvhodnější je použít kontejner v JBoss Application Serveru, protože je s RESTEasy nejlépe integrován, a proto je konfigurace mnohem jednodušší než v ostatních kontejnerech. V kontejnerech jako Apache Tomcat nebo GlassFish bude RESTEasy fungovat také. Kontejner musí běžet minimálně na JDK 5, aby RESTEasy správně fungovalo.

RESTEasy kromě serverové části poskytuje framework pro klientské aplikace. Tento framework využívá JAX-RS anotace pro vytváření HTTP požadavků, které jsou pak použity pro invokaci vzdálené RESTful webové služby. Vzdálená webová služba nemusí nutně využívat specifikace JAX-RS, ale muže to být kterákoliv služba přijímající HTTP požadavky. Na výpisu 5.11 je možné vidět příklad použití klientského frameworku. Na tomto příkladu můžeme vidět rozhraní RESTClient definující metodu `getClanek()`, která představuje HTTP požadavek s metodou GET, s relativní cestou */basic* a MIME typem `text/Plain`.

RESTEasy také obsahuje kontejner Tiny Java Web Server and Servlet Container (TJWS) [20]. TJWS je malý kontejner s velmi rychlým startem, který je začleněn do RESTEasy pro testování, aby

vývojář mohl testovat volání svých JAX-RS zdrojových metod bez složité instalace, konfigurace a spouštění plnohodnotného kontejneru.

Z dalších funkcí, které RESTEasy podporuje, je vhodné ještě zmínit GZIP kompresi, kdy je tělo požadavků a odpovědí komprimováno, což může velmi zvýšit propustnost systému. RESTEasy také podporuje kešování HTTP odpovědí, ale také kešování požadavků, kdy se vnitřní reprezentace často volaných JAX-RS zdrojových metod kešuje, což také může významně zvýšit výkon systému. RESTEasy také musí dle specifikace JAX-RS implementovat automatickou konverzi JAXB objektů na dokument XML a naopak. RESTEasy dokáže JAXB objekty konvertovat kromě XML také do formátu JavaScript Object Notation (JSON) [22]. [21]

```
public interface RESTClient {
  @GET
  @Path("basic")
  @Produces("text/plain")
  String getClanek(String jmeno);
    ...//dalsi definice metod
```

Výpis 5.11: Definice HTTP požadavku v klientském frameworku.

6 JBoss Enterprise Service Bus

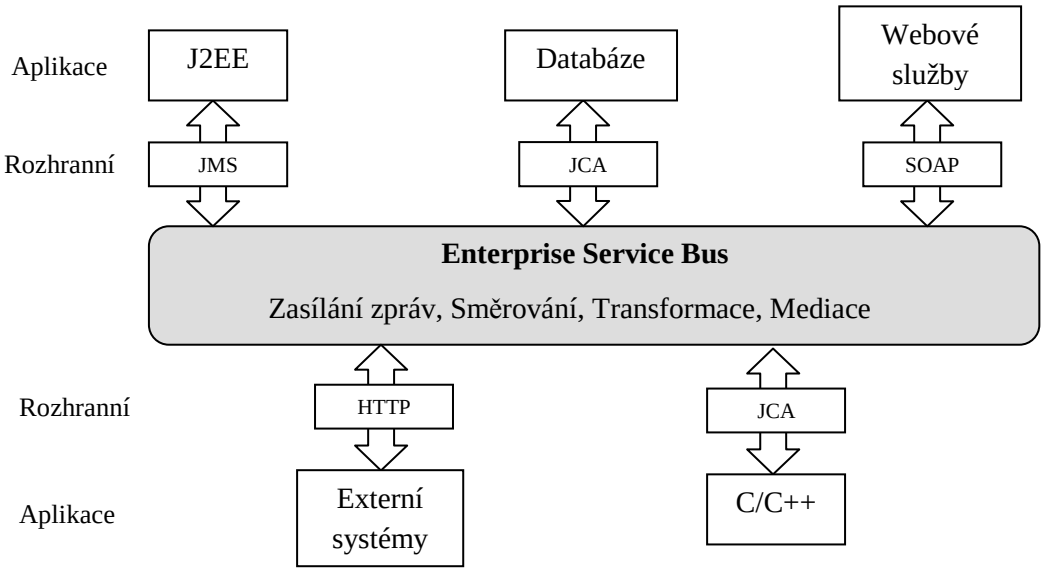
JBoss Enterprise Service Bus (JBoss ESB) vyvíjí společnost Red Hat, Inc. od roku 2006. Zatím poslední verze JBoss ESB 4.9 vyšla v srpnu roku 2010. JBoss ESB poskytuje nástroje a metody, které ulehčují izolaci aplikační logiky od přenosu a spouštěcích mechanizmů, logování aplikačních a procesních událostí, které jím prošly. JBoss ESB také umožňuje vkládat flexibilní pluginy aplikační logiky a transformaci dat. Ty se poté vkládají do JBoss ESB serveru jako ESB soubory a v této práci je budeme nazývat akce.

V JBoss ESB můžeme na vše nahlížet, jako na službu nebo zprávu. Každá služba zapouzdřuje aplikační logiku nebo integruje již existující systémy. Pomocí zpráv pak mohou služby komunikovat s klienty a ostatními službami.

Enterprise Service Bus (ESB) je softwarová architektura, která je součástí SOA. ESB dokáže integrovat nekompatibilní aplikace, což znamená, že různé systémy budou moci komunikovat přes jednotné komunikační rozhraní. Jednotlivé aplikace mohou být připojeny do ESB jako služby, které je možné snadno měnit, připojovat, odpojovat a řídit.

ESB implementuje softwarovou sběrnici, jejímž cílem je, aby služby nekomunikovaly přímo mezi sebou, ale pouze přes sběrnici. Na obrázku 6.1 můžeme vidět příklad připojení aplikací do ESB. Aplikace musí být opatřeny rozhraním, aby mohly komunikovat s ESB a fungovat jako služby. Na připojení aplikací implementovaných v J2EE můžeme použít JMS (Java Message Service). JMS je standard, který umožňuje komponentám založeným na J2EE vytvářet, posílat a přijímat zprávy mezi sebou. Více o JMS lze nalézt v jeho specifikaci [24]. Pro připojení již existujících systémů (napsaných, např. v C/C++) je možno použít Java EE Connector Architecture (JCA) [25]. JCA je vhodné také použít pro připojení databází k ESB. Webové služby můžeme připojit k ESB pomocí protokolů HTTP nebo SOAP.

Aplikace jednoduše pošlou zprávu do ESB, která se již postará, aby zpráva došla správnému adresátovi. ESB umožňuje komunikovat aplikacím, které mají různá rozhraní pomocí služeb, které transformují zprávu do požadovaného formátu. [26]



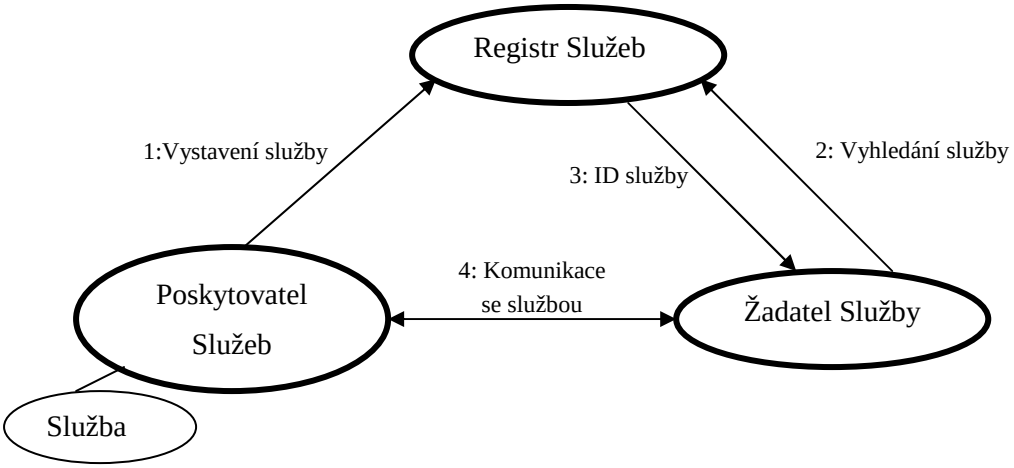
Obrázek 6.1: Propojení aplikací s ESB. (Upraveno z [27])

JBoss ESB poskytuje základní infrastrukturu pro vývoj aplikací na Architektuře Orientované na Služby, angl. Service Oriented Architecture (SOA). Proto si v následující podkapitole SOA představíme. V druhé podkapitole si popíšeme služby a zprávy v JBoss ESB.

6.1 Architektura orientovaná na služby

Cílem architektury orientované na služby je integrace rozdílných systémů do vzájemně spolupracujících, znovupoužitelných a na sobě nezávislých služeb. SOA je v zásadě kolekce služeb, které mezi sebou komunikují pomocí standardního rozhraní. JBoss ESB poskytuje základní infrastrukturu, na které mohou být vyvíjeny SOA aplikace.

SOA vznikla proto, že požadavky na aplikace se stále mění, již existující software tak často nesplňuje měnící se požadavky, které jsou na něj kladeny a výměna tohoto softwaru za zcela nový by byla velmi finančně náročná. Naproti tomu s pomocí SOA je možné vytvořit aplikace, které budou snadno udržovatelné, flexibilní a budou schopny spolupracovat se stávajícím softwarem.



Obrázek 6.2: Vytvoření a volání služby v SOA.

V SOA můžeme nalézt následující tři role:

- **Poskytovatel služeb**: Umožňuje přístup ke službám a vytváří popis služeb. Také je zodpovědný za vystavení služby v registru služeb
- **Žadatel služby**: Vyhledá službu v registru služeb pomocí popisů služeb a je zodpovědný za připojení ke službám, které obsahuje poskytovatel služeb.
- **Registr služeb**: Obsahuje registr popisů služeb. Jeho hlavním úkolem je spojit žadatele služby s poskytovatelem služby.

Na obrázku č. 6.1 můžeme vidět vytvoření a volání služby. Nejprve je vytvořena služba. Poskytovatel služeb vytvoří popis služby a tento popis vystaví do registru služeb (1). Žadatel služby, který chce službu využít, musí službu nejprve vyhledat v registru služeb (2). Registr služeb poté vrátí nalezenou službu (3). Pak již může probíhat komunikace mezi žadatelem služby a poskytovatelem služeb (4). [23]

6.2 Služby a zprávy v JBoss ESB

V této kapitole si definujeme služby v JBoss ESB a popíšeme si, z čeho se skládají. Také se seznámíme se zprávami v JBoss ESB, pomocí kterých služby komunikují.

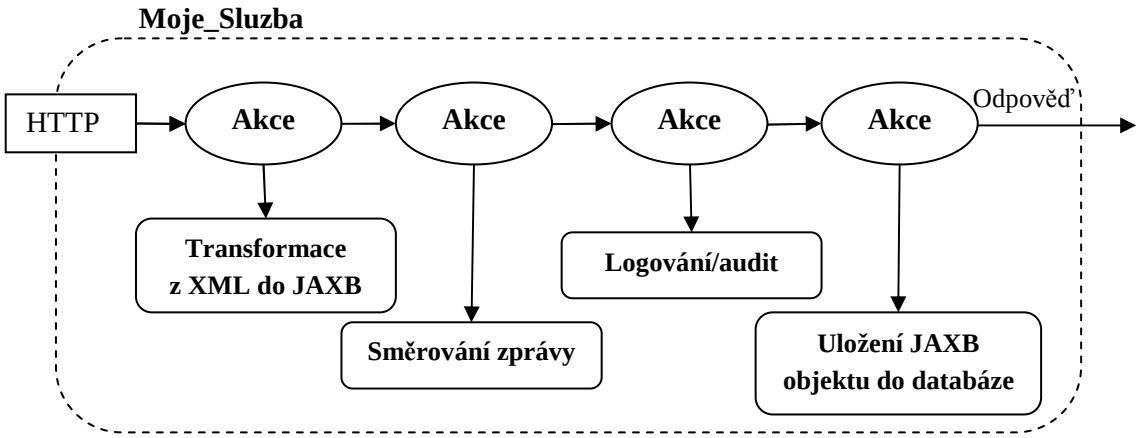
6.2.1 Služby

Službu můžeme v JBoss ESB definovat jako sled akcí, které zpracovávají zprávu v sekvenčním pořadí. Sled akcí ve službě se nazývá *action pipeline.* Implementační detaily, jako například úprava databáze nebo transformace, jsou ukryty právě v akcích. JBoss ESB používá zprávou řízený model pro definici služeb. Při vytváření služby je potřeba nejprve definovat rozhraní, které zpřístupní službu klientům. Rozhraní na serveru se definuje zprávou. Klient pak může používat službu tak dlouho, dokud se nezmění definice zprávy. Jak je poté zpráva v akcích zpracována, je již ukryto za rozhraním jednotlivých akcí.

Na obrázku 6.3 můžeme vidět příklad služby. Služba může definovat sadu tzv. *naslouchačů*, angl. *listener*, které směrují zprávy do sledu akcí. Služba na obrázku 6.3 čeká na přijetí HTTP zprávy, kterou pak dále směruje do první akce, kde se tělo HTTP zprávy transformuje na JAXB objekt a je přeposlán do další akce. V další akci dojde ke směrování zprávy podle jejího obsahu. V předposlední akci se zpráva zaloguje a v poslední akci je již business logika, kdy se JAXB objekt uloží do databáze. Výsledek poslední akce se poté odešle zpět klientovi.

Každá služba má dva hlavní atributy, kterými jsou jméno a kategorie. Pokud chceme vložit službu do JBoss ESB, tak právě pomocí těchto dvou atributů se vystaví naslouchače služby v registru služeb. Klienti pak mohou vyvolat službu pomocí kteréhokoliv naslouchače, např. pomocí HTTP, JMS, ale i vytvořením souboru v adresáři. V JBoss ESB existují dva typy naslouchačů:

- **Vstupní brána (Gateway listener)**: Slouží jako vstupní brána pro zprávy do ESB sběrnice a má na starosti normalizování těla zprávy tak, že tělo zabalí do ESB zprávy (viz. kapitola 6.3.2.) .
- **ESB-znalý (ESB-aware) naslouchač**: Je určen k výměně zpráv na ESB sběrnici mezi komponentami, které rozumí ESB zprávám.



Obrázek 6.3: Příklad služby.

Výše uvedené informace a podrobnější popis je možné najít v programátorské dokumentaci k JBoss ESB [26].

6.2.2 Zprávy

Jak jsme již uvedli v předchozích kapitolách, klienti se službami komunikují pomocí zasílání zpráv. Zprávy můžeme rozdělit na 2 hlavní typy. Jsou to ESB zprávy a ESB-neznalé (angl. ESB-unaware) zprávy. Pomocí ESB zpráv se realizuje vnitřní komunikace na ESB sběrnici, např. mezi službami připojenými ke sběrnici. Jednotlivé akce ve službách rozumí pouze ESB zprávám, ale klienti většinou posílají ESB-neznalé zprávy, tedy např. HTTP či SOAP. Proto je potřeba službě definovat naslouchač typu gateway, který dokáže zkonvertovat ESB- neznalou zprávu na ESB zprávu, kterou již jednotlivé akce ve službě umí zpracovat.

ESB zpráva se skládá ze čtyř částí, které jsou uvedeny na obrázku 6.4. Hlavička zprávy obsahuje směrovací a adresovací informace. V hlavičce se také nachází identifikace zprávy. V tabulce 6.1 jsou popsány jednotlivé části hlavičky.

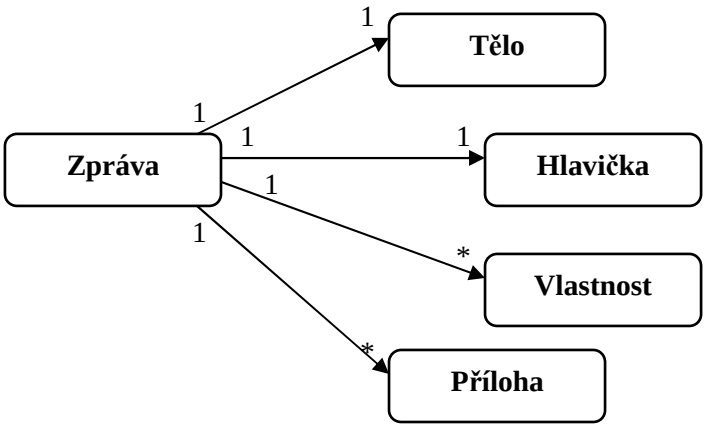
Pole Hlavičky	**Popis**
To	Identifikuje adresáta zprávy.
From	Obsahuje adresu odesílatele.
ReplyTo	Na adresu v tomto poli bude zaslána odpověď, pokud je odpověď požadována. Jestliže `ReplyTo` není definováno, použije se adresa z pole `From`.
FaultTo	Obsahuje adresu, na kterou se budou zasílat chyby spojené se zprávou. Jestliže FaultTo neni definováno, hlášení o chybě se pošle na adresu v poli `ReplyTo`. Pokud i `ReplyTo` je prázdné, bude použito pole `From`.
Action	Zde je identifikátor, který určuje význam zprávy.
Message ID	Toto pole jednoznačně identifikuje zprávu v rámci JBoss ESB

Tabulka 6.1: Pole Hlavičky zprávy

V těle zprávy se nachází užitečná data libovolného datového typu. Pokud je v těle zprávy objekt, je potřeba ho při přenosu zprávy *serializovat*. Serializace je proces, při kterém je objekt převeden do sekvenční podoby, aby jej bylo možné uložit do souboru nebo přenést přes síť. Při vkládání dat do těla zprávy je potřeba si dávat pozor na to, aby si jednotlivé akce svá užitečná data nepřepisovaly. Do přílohy zprávy se vkládají data, které uživatel nechce mít v těle zprávy, např. obrázky nebo binární data. Vkládat přílohu do ESB zpráv není v současnosti vývojáři JBoss ESB doporučováno, protože implementace přílohy zprávy se ještě stále vyvíjí, a proto není vyloučeno, že v dalších verzích JBoss ESB může dojít k podstatným změnám ve funkcionalitě příloh. Poslední částí zprávy jsou vlastnosti, ve kterých je možno specifikovat další vlastnosti zprávy, které nejsou obsaženy v hlavičce.

Zaslání odpovědi, je možné definovat několika způsoby. Pokud je sled akcí definován jako Požadavek/Odpověď, tak se odpověď zašle na adresu v poli `ReplyTo`. Pokud není `ReplyTo` definováno, použije se adresa z pole `From`. Jestli pole `From` je také prázdné, odpověď nebude odeslána.

Když je sled akcí nastaven jako jednosměrný, odpověď nebude nikdy zasílána. V případě, že není zacházení s odpovědí definováno, použije se podobný model jako Požadavek/Odpověď jen s tím rozdílem, že pokud jsou pole `ReplyTo` i `From` prázdné. Tedy odpověď nemůže být zaslána a bude vytvořeno chybové hlášení v logu serveru. Jestliže kterákoliv z akcí v sledu akcí vrátí NULL jako zprávu, tak se provádění sledu akcí zastaví a žádná odpověď nebude zaslána. [26]



Obrázek 6.4: UML diagram struktury zprávy. [26]

7 Současná podpora REST v JBoss ESB

REST v JBoss ESB není přímo podporován, ale existují zde akce pro zasílání a příjem HTTP požadavků. Jelikož REST používá HTTP, můžeme vytvářet pomocí těchto akcí REST požadavky a přijímat REST dotazy klientů. [26]

7.1 Výstupní část

Pro vytváření HTTP požadavků v současnosti existují v JBoss ESB dvě akce, *JBoss Remoting HttpRouter* a *Apache Commons HttpRouter*.

JBoss Remoting HttpRouter používá JBoss Remoting pro vyvolání HTTP požadavku. Nachází se ve třídě `org.jboss.soa.esb.actions.routing.HttpRouter`. Tato akce je již zastaralá a je doporučeno použít raději *Apache Commons HttpRouter*.

Tato akce zašle na určité URL HTTP požadavek, ve kterém bude tělo zprávy, jenž akce přijala přes ESB. URL se definuje v konfiguračním souboru pod atributem `routeURI`. Pokud není specifikováno žádné URI, je požadavek zaslán na adresu *localhost* na port 5400.

JBoss Apache Commons HttpRouter Tato akce pro vyvolání požadavku používá *Apache Commons HttpClient*. Můžeme ji najít ve třídě `org.jboss.soa.esb.actions.routing.http.HttpRouter`. V této akci již můžeme nastavit mnohem více parametrů HTTP požadavku. V tabulce 7.1 jsou vypsány všechny možné parametry, které je možné zadat do konfiguračního souboru.

Atribut	Popis	Povinné
endpointUrl	Koncové URL, na které bude zpráva zaslána.	ANO
http-client-property	Tato akce používá třídu *HttpClientFactory*, aby nastavila a vytvořila instanci třídy *HttpClient*. Tímto atributem může uživatel specifikovat soubor, ve kterém je nastavení instance HttpClienta.	NE
method	Tímto atributem se nastaví metoda požadavku. V této akci jsou podporovány metody GET a POST.	ANO
responseType	Zde lze specifikovat, v jakém formátu by měla být zpět zaslána odpověď. Podporován je BYTE a STRING.	NE
headers	Uživatel také může specifikovat hlavičky, které budou zaslány v HTTP požadavku.	NE

Tabulka 7.1: Atributy v HttpRouteru.

Příklad nastavení REST požadavku, pokud chceme získat XML soubor ze serveru *www.twitter.cz* je na výpisu 7.1, kde je definována akce s názvem `httprouter`. Akci musíme atributem `class` přiřadit třídu, která danou akci implementuje. Pak pomocí elementů `property` můžeme nastavit parametry akce.

```
<action name="httprouter"
class="org.jboss.soa.esb.actions.routing.http.HttpRouter">
<property name="endpointUrl"
   value=http://www.twitter.cz:80/polozka/14741/>
<property name="method" value="GET"/>
    <property name="headers">
        <header name="Content-Type" value="text/xml" ></header>
    </property>
</action>
```

Výpis 7.1: Nastavení požadavku.

7.2 Vstupní část

Pro příjem HTTP požadavků do JBoss ESB slouží HTTP Gateway. Tato brána používá HTTP kontejner pro vystavení koncových bodů. Každý koncový bod má své URL. HTTP Gateway umožňuje definovat vzor URL, což je maska, která označuje určitou množinu URL. HTTP Gateway příjme všechny požadavky na URL, které jsou kompatibilní s daným vzorem. URL se definuje pomocí atributu `urlPattern`. HTTP Gateway vytváří URL, na kterém bude naslouchat podle toho, jestli je definován URL vzor. Pokud je URL vzor definován, tak bude HTTP Gateway naslouchat na URL podle schématu na výpisu 7.2, a jestliže URL vzor není definován, tak bude použito schéma na výpisu 7.3.

```
http://<host>:<port>/<jméno .esb souboru>/http/<URL vzor>
```

Výpis 7.2: Tvorba URL s URL vzorem.

```
http://<host>:<port>/<jméno .esb souboru>/http/<kategorie služby>/
                        <jméno služby>
```

Výpis 7.3: Tvorba URL bez URL vzoru.

Prvek `host` z výpisu 7.2 a 7.3 je URL adresa serveru, na kterém běží kontejner s HTTP Gateway. Porty, na kterých HTTP Gateway bude naslouchat, je možné nastavit atributem `allowedPorts`. Za portem následuje jméno souboru se službou, který jsme umístili na server a pevný řetězec `http`. Nakonec následuje buď URL vzor, nebo kategorie, a název služby podle toho, zda je URL vzor definován či ne.

Po příjmu HTTP požadavku je jeho tělo konvertováno podle MIME typu do `String` nebo `byte [ ]`, a poté uloženo do zprávy, kde k němu budou mít přístup jednotlivé akce, které jsme si definovali v kapitole 5. Informace z hlavičky dokumentu jsou také uloženy do zprávy jako objekt třídy `HttpRequest`.

```
<service category="logovani" name="clanek" description="sluzba">
<listeners><http-gateway name="Http" urlPattern="clanky/*" />
</listeners>
    <actions mep="RequestResponse">
    </actions>
</service>
```

Výpis 7.4: Nastavení HTTP Gateway.

Na výpisu 7.4 je příklad nastavení služby používající HTTP Gateway. Službě je nejdříve potřeba definovat naslouchač typu *http-gateway*, čímž budou všechny HTTP požadavky, které vyhovují URL vzoru *clanky/** konvertovány do ESB zprávy. ESB zpráva bude poté směrována do sledu akcí, kde jednotlivé akce HTTP požadavek zpracují. Sled akcí by měl být typu Požadavek/odpověď, protože HTTP protokol je postaven na principu dotaz-odpověď. Klient tedy bude od serveru očekávat odpověď.

Následující akce ve sledu akcí poté uloží HTTP odpověď do objektu `HttpResponse` a vloží jej do zprávy. HTTP Gateway z objektu `HttpResponse` vytvoří HTTP odpověď a zašle ji klientovi. Odpověď ze sledu akcí musí být dostupná standardně do 30 vteřin, ale je možné tuto dobu změnit pomocí atributu `synchronousTimeout`.

Pro vytvoření odpovědi je také možné si v HTTP Gateway definovat tzv. mapovač výjimek, ve kterém výjimkám přiřadíme určitý HTTP stavový kód. Mapování můžeme definovat buď přímo v konfiguraci HTTP Gateway, nebo ve speciálním souboru.

Výše byly uvedeny jen ty nejdůležitější parametry HTTP Gateway. Mnoho dalších možností je možné najít v JBoss ESB programátorské dokumentaci. [26]

Podpora REST v JBoss ESB je velmi malá. HttpRouter dokáže sice zasílat HTTP požadavky, ale uživatel musí tyto požadavky definovat staticky v konfiguračním souboru, což je nevhodné, pokud je potřeba zasílat rozdílné požadavky na různé URL. HttpRouter dokáže zasílat HTTP požadavky pouze metodami GET a POST. Všechny tyto nedostatky bude potřeba odstranit. HTTP Gateway pouze přijímá HTTP požadavky. Bude tedy potřeba vytvořit rozhraní nad HTTP Gateway, které bude moci uživatel použít pro příjem REST požadavků.

8 Analýza požadavků integrace REST do JBoss ESB

Úkolem této práce je vylepšit podporu REST v JBoss ESB. Také je potřeba integrovat JBoss ESB s projektem RESTEasy.V této kapitole si uvedeme základní požadavky pro podporu REST v JBoss ESB. Podpora REST pro JBoss ESB bude rozdělena na vstupní a výstupní část. Vstupní část bude přijímat požadavky od vzdálených klientů a po zpracování požadavku, vytvoří odpověď, kterou zašle zpět klientovi. Výstupní část bude přijímat ESB zprávy, ze kterých poté vytvoří REST požadavek. REST požadavek odešle a bude čekat na odpověď, kterou zpracuje. Také zde budou uvedeny požadavky na integraci REST vstupní a výstupní části s RESTEasy.

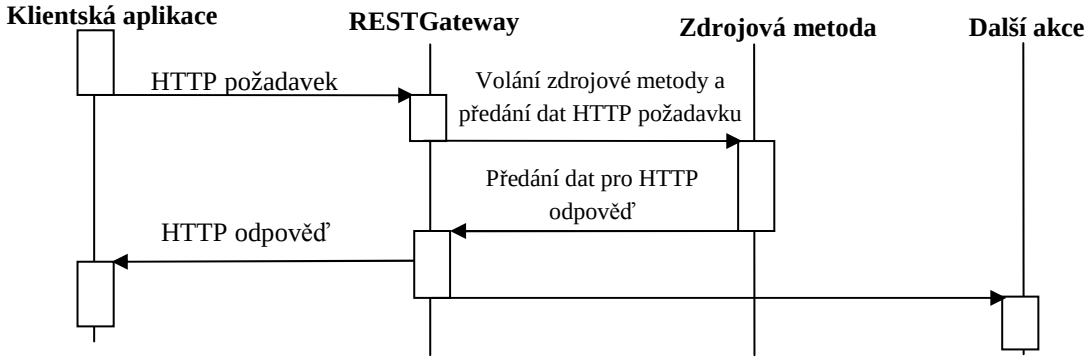
8.1 Vstupní část

Cílem vstupní části bude umožnit JBoss ESB přijímat REST požadavky a vytvářet odpovědi, které poté budou zaslány odesílateli. Pro příjem REST požadavků bude sloužit akce, která bude nazvána *RESTGateway*. RESTGateway bude využívat specifikace JAX-RS, protože JAX-RS definuje způsob příjmu REST požadavků, a proto bude vhodné ji využít. RESTGateway částečně implementuje specifikaci JAX-RS. Implementace bude jen částečná, protože kompletní implementace JAX-RS by byla velmi časově náročná. Budou tedy implementovány jen některé části JAX-RS, které jsou pro příjem REST požadavků nejdůležitější.

Ze specifikace JAX-RS bude potřeba implementovat anotace určující metody přijímaných REST požadavků (`@GET`, `@POST`). Také budou implementovány anotace pracující s HTTP hlavičkou Content-Type, aby si uživatel mohl definovat MIME typy, které budou vstupní částí přijímány. Jelikož uživatel bude potřebovat ve zdrojových metodách přistupovat k informacím, které jsou uloženy v REST požadavku, tak budou implementovány anotace vkládající hodnoty hlavičky požadavku do zdrojové metody. Také bude nutné umožnit uživateli přístup k tělu REST požadavku.

Uživateli také musí být umožněno vytvořit odpověď na REST požadavek. To bude možné stejně jako v JAX-RS pomocí speciálních objektů, ve kterých si uživatel bude moci definovat hlavičku i tělo odpovědi.

V RESTGateway bude možné vytvořit třídu se zdrojovými metodami podle specifikace JAX-RS, které budou volány podle typu přijatého HTTP požadavku. Na obrázku 8.1 je schéma komunikace akce RESTGateway.



Obrázek 8.1: Schéma komunikace vstupní části.

Klientská aplikace zašle akci RESTGateway HTTP požadavek. RESTGateway se na základě parametrů HTTP požadavku (URL, metoda, hlavičky) rozhodne, zda požadavek příjme. Při přijmu požadavku RESTGateway vyvolá příslušnou zdrojovou metodu a pomocí JAX-RS anotací do ní injektuje parametry z HTTP požadavku. Ve zdrojové metodě se provede nějaký užitečný kód, např. vytvoření záznamu v databázi, a poté zdrojová metoda vrátí data potřebná k sestavení HTTP odpovědi (hlavně stavový kód a tělo odpovědi). RESTGateway poté zašle odpověď klientovi a předá řízení další akci, pokud to bude potřeba, např. pro logování HTTP požadavků.

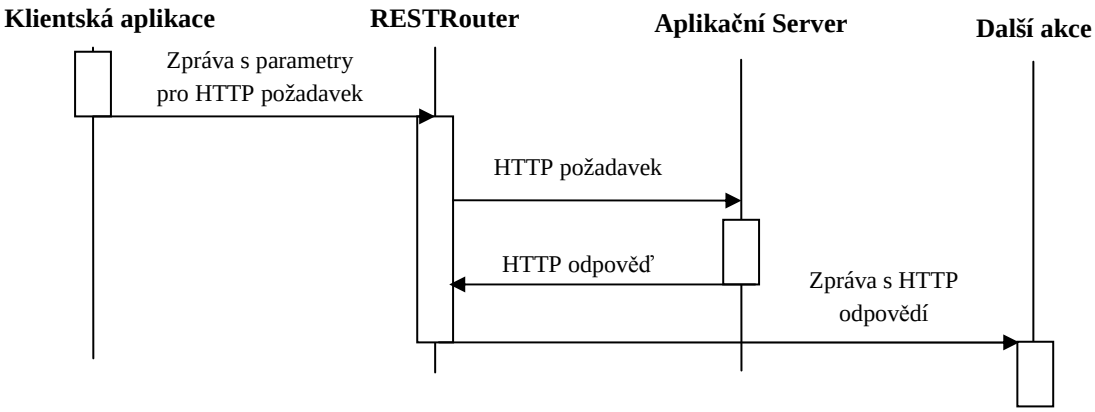
8.2 Výstupní část

Jak bylo uvedeno v kapitole 7.1, JBoss ESB umožňuje zasílat HTTP požadavky pomocí *JBoss Apache Commons HttpRouter* (dále jen HttpRouter). HttpRouter dovoluje nastavit parametry požadavku pouze staticky v konfiguračním souboru. Proto HttpRouter můžeme použít jen tam, kde akce zasílá požadavky na stále stejnou URL adresu a parametry požadavku se nemění. To je nevhodné pro aplikace, které zasílají různé požadavky na různé URL adresy, protože pro každý požadavek by bylo potřeba vytvořit novou akci a umístit ji na server.

Prvním požadavkem na akci, která bude pojmenována *RESTRouter*, bude poskytnout uživateli možnost definovat si parametry požadavku dynamicky pomocí zprávy. RESTRouter bude čekat na příchozí ESB zprávu, ve které budou uloženy informace potřebné k odeslání požadavku. Bude tedy možné odesílat různé požadavky na různé URL adresy pomocí jediné akce.

Na obrázku 8.2 je schéma komunikace. Klientská aplikace nejdříve sestaví zprávu s parametry pro HTTP požadavek a zašle ji akci RESTRouter. RESTRouter poté ze zprávy vytvoří HTTP požadavek a zašle jej na požadované URL, a poté bude čekat na odpověď. Když RESTRouter přijme HTTP odpověď, tak z ní vytvoří ESB zprávu, kterou pak může poslat další akci. Existence další akce za použitou akcí RESTRouter bude nutné, protože RESTRouter HTTP odpověď vloží do ESB zprávy a další akce provede užitečný kód podle typu odpovědi.

HttpRouter podporuje pouze HTTP metody GET a POST. Dalším požadavkem na RESTRouter bude rozšířit podporu HTTP metod o metody PUT a DELETE. Dále bude potřeba, aby RESTRouter umožnil autentizaci klienta pomocí jednoduchého ověření přístupu. Dalším požadavkem je zachovaní zpětné kompatibility s HttpRouterem, což znamená, že stále musí být možné definovat HTTP požadavek staticky v konfiguračním souboru.



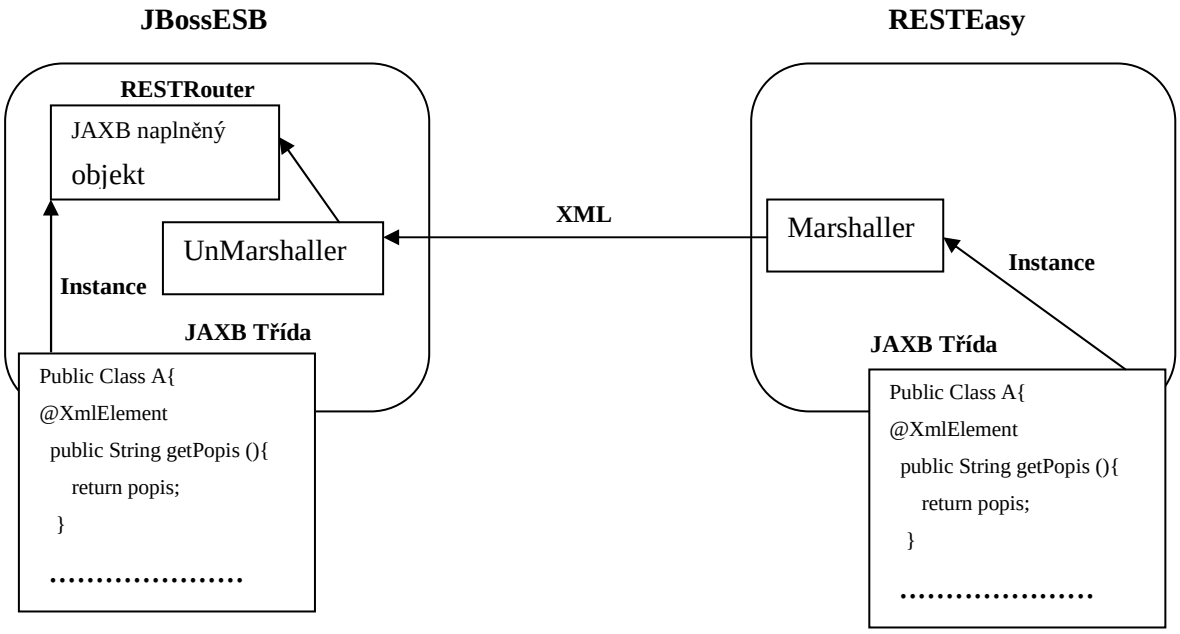
Obrázek 8.2: Schéma komunikace výstupní části.

8.3 Integrace JBoss ESB a RESTEasy

V kapitole 6.7 bylo uvedeno, že každá implementace specifikace JAX-RS musí podporovat automatickou konverzi Java objektů, dle specifikace JAXB. RESTEasy podporuje konverzi JAXB objektů do formátu XML a JSON. Taková automatická konverze vývojářům velmi ulehčuje práci, protože se již nemusí zabývat, jak zpracovat například XML dokument, a mohou se soustředit na vývoj aplikace. Proto je vhodné, aby akce RESTRouter podporovala automatickou konverzi také. Poté tedy bude možné do zprávy pro RESTRouter vložit JAXB objekt, který bude zkonvertován do požadovaného formátu, pokud to bude potřeba.

Požadavkem je integrovat JAXB do JBoss ESB tak, aby bylo možné použít stejnou anotovanou JAXB třídu v RESTEasy, ale také i v JBoss ESB. Na obrázku 8.3 je schéma komunikace mezi RESTEasy a JBoss ESB.

Jelikož RESTEasy již konverzi JAXB objektů do XML a JSON podporuje, můžeme pro konverzi JAXB objektů v RESTRouteru využít knihoven z RESTEasy. Nebude příliš těžké knihovny z RESTEasy do RESTRouteru integrovat a ušetříme tím mnoho práce a vyvarujeme se zbytečných chyb.



Obrázek 8.3: Integrace JBoss ESB a RESTEasy.

RESTEasy posílá objekt JBoss ESB, který pomocí JAXB převede na XML. JBoss ESB zprávu přijme pomocí akce RESTRouter a přes JAXB zprávu zpět převede na naplněný Java objekt. JBoss ESB i RESTEasy musí oba obsahovat stejnou JAXB anotovanou třídu.

V RESTGateway můžeme využít také knihoven z RESTEasy, jelikož RESTEasy je implementace JAX-RS a RESTGateway implementuje specifikaci JAX-RS pro použití v JBoss ESB. Z RESTEasy můžeme použít nejen knihovny, ale také některé zdrojové kódy. Integrace s RESTEasy bude na velmi vysoké úrovni, protože bude možné použít existující JAX-RS zdrojové třídy z RESTEasy také v RESTGateway.

9 Návrh integrace REST do JBoss ESB

V této kapitole se budeme nejprve zabývat návrhem RESTRouteru, kde si popíšeme formát zprávy s parametry pro HTTP požadavek. Také si zde uvedeme, jak bude fungovat automatická konverze JAXB objektů do XML a JSON a naopak. Popíšeme si strukturu akce pomocí stavového diagramu a diagramu tříd. V další části si uvedeme návrh RESTGateway, kde si řekneme, které časti specifikace JAX-RS budou implementovány a popíšeme si strukturu akce pomocí stavového diagramu a diagramu tříd.

9.1 Návrh RESTRouteru

Při návrhu RESTRouteru je potřeba dbát na to, že musí být zpětně kompatibilní s HttpRouterem, aby všechny stávající aplikace používající HttpRouter mohly fungovat i s RESTRouterem. Proto bude RESTRouter dědit třídu `org.jboss.soa.esb.actions.routing.http.HttpRouter` a pak podle potřeby používat její metody a atributy. Díky tomu nebude potřeba znovu implementovat to, co již HttpRouter podporuje a zároveň tím snížíme riziko nových chyb. RESTRouter bude primárně načítat parametry z konfiguračního souboru. Pokud je v konfiguračním souboru nenajde, bude je hledat v přijaté ESB zprávě.

Dále se budeme zabývat formátem ESB zprávy, který bude RESTRouter přijímat. Ve zprávě bude potřeba vytvořit mapu, např. `java.util.HashMap`, která bude obsahovat parametry HTTP požadavku. Mapa v jazyce Java mapuje klíče na hodnoty. Klíčem bude název parametru nebo hlavičky, ke klíči přiřadíme hodnotu parametru nebo hlavičky. V tabulce 9.1 je seznam parametrů, které bude možné vložit do mapy. Adresa URL a metoda HTTP požadavku jsou povinné, pouze pokud nebudou uvedeny v konfiguračním souboru. Samotná mapa pak musí být v ESB zprávě uložena pod jménem *params*.

Popis HTTP parametru	**Klíč, pod kterým bude uložen**	**Povinný**
URL adresa, na kterou bude požadavek zaslán.	url	ANO
Metoda HTTP požadavku.	method	ANO
Jméno pro jednoduché ověření přístupu.	username	NE
Heslo pro jednoduché ověření přístupu.	password	NE
Hlavička HTTP požadavku.	Název hlavičky	NE

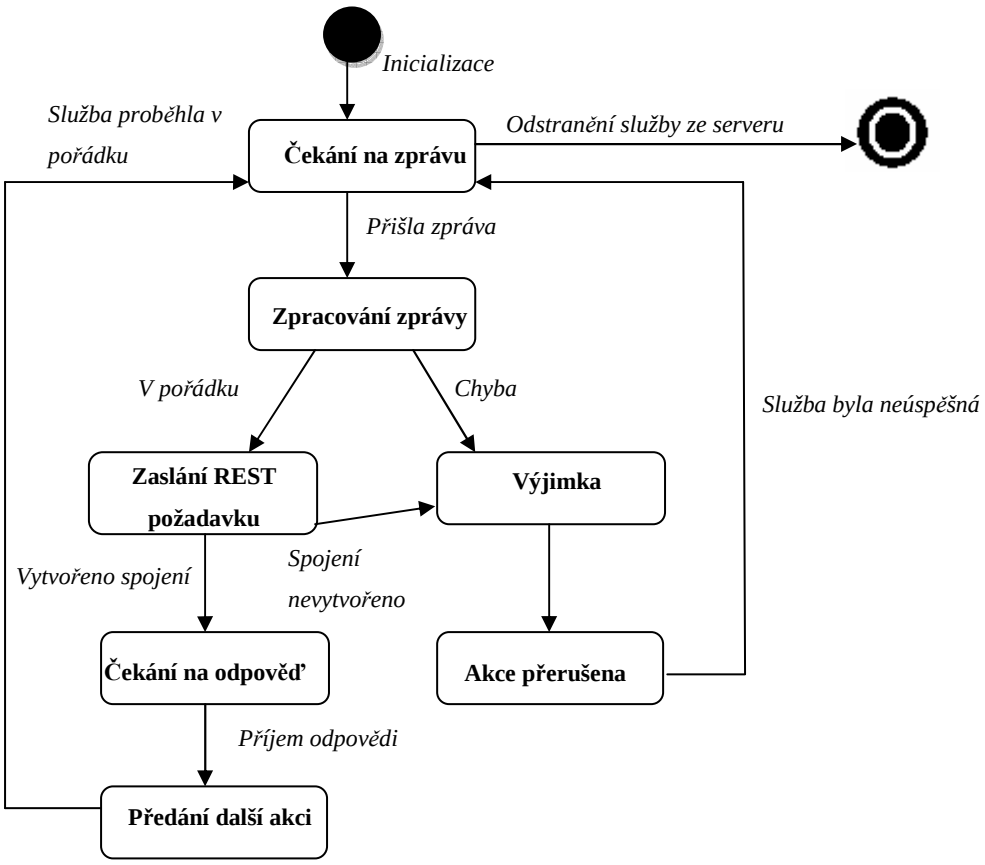
Tabulka 9.1: Parametry mapy zprávy

ESB zpráva také může obsahovat tělo HTTP požadavku pro metody POST a PUT. Tělo HTTP požadavku bude uloženo ve standardním umístění ESB zprávy.

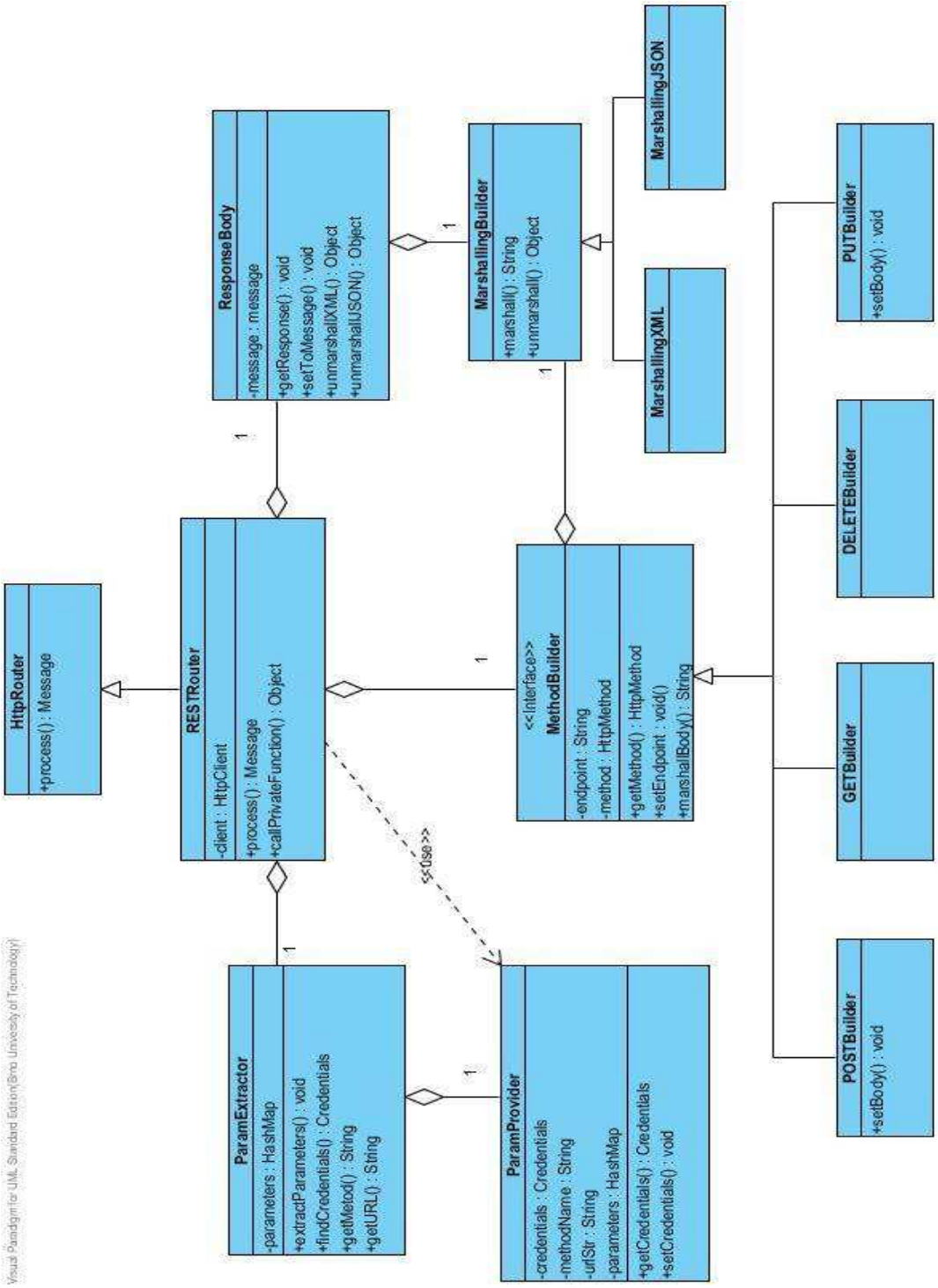
Do zprávy bude také možné vložit JAXB objekt, který se poté konvertuje do XML nebo JSON, podle nastavení hlavičky Content-Type v mapě s parametry pro HTTP požadavek. JAXB objekt bude ve zprávě pod jménem *marshall*. Pod jménem *unmarshall* bude JAXB třída, pomocí které pak bude možné konvertovat XML nebo JSON dokument v HTTP odpovědi na JAXB objekt.

Chování RESTRouteru je zobrazeno na obrázku 9.1, kde je stavový diagram, který obsahuje základní kroky RESTRouteru, od příjmu zprávy až do předání odpovědi další akci. Služba RESTRouter při příjmu zprávy vykoná následující kroky:

- Analyzuje zprávu, zda má požadovaný formát a zda obsahuje povinné informace. Povinnými údaji bude URL a HTTP metoda.
- Zkontroluje formát URL, a jestliže formát není správný, vyvolá výjimku a ukončí ihned akci.
- Poté si ověří, zda je zadaná metoda podporována. Kromě metod GET a POST budou podporovány také metody DELETE a PUT.
- Vytvoří HTTP požadavek podle parametrů v ESB zprávě nebo parametrů v konfiguračním souboru.
- Po vytvoření požadavku bude výsledek zaslán na požadované URL. Pokud se nepodaří s tímto URL navázat spojení, bude vyvolána výjimka.
- Po příjmu odpovědi na požadavek se tato odpověď zpracuje a předá další akci, pokud to bude potřeba.



Obrázek 9.1: Stavový diagram RESTRouteru.



Obrázek 9.2: Schematický diagram tříd RESTRouteru.

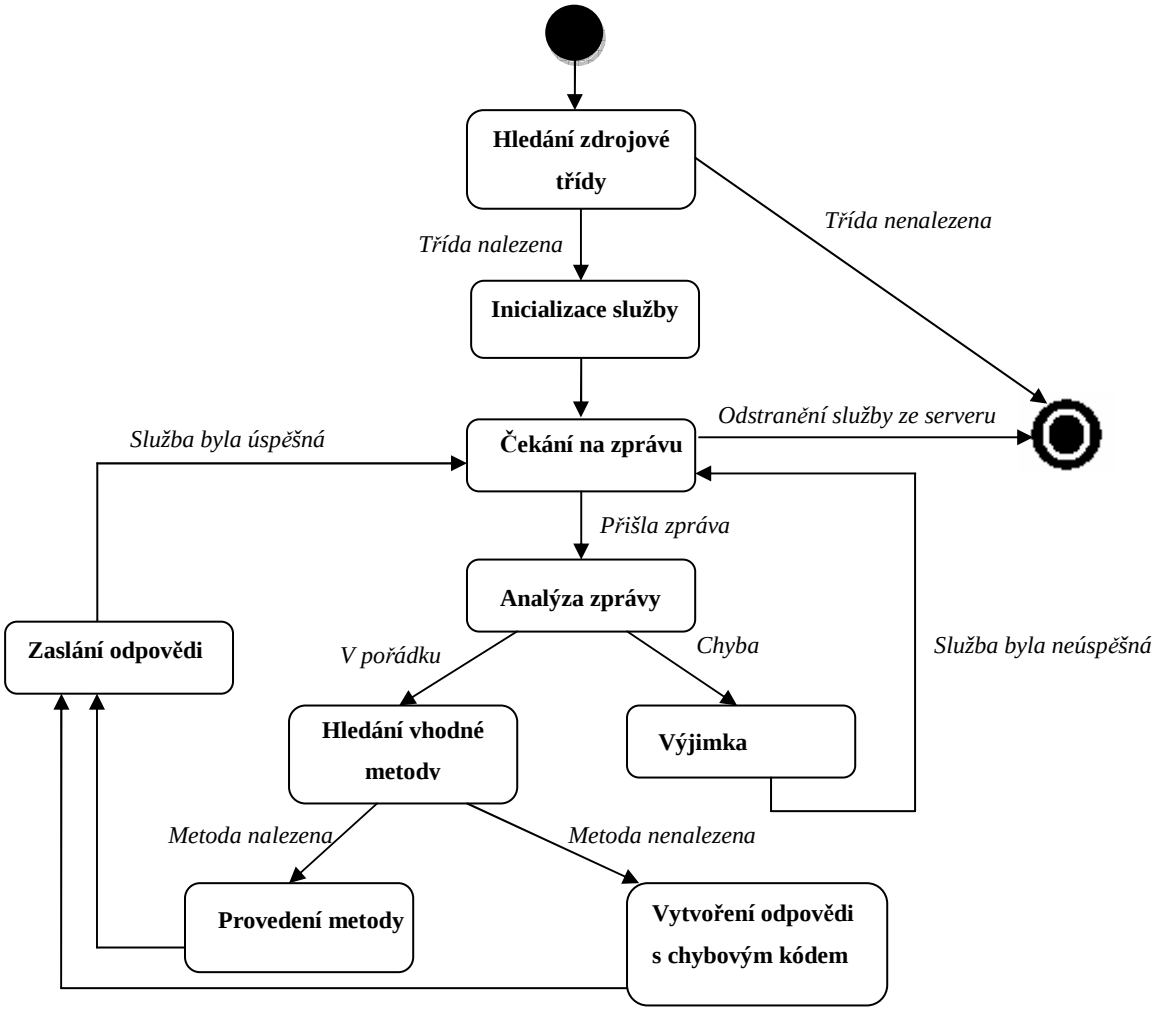
Na obrázku 9.2 je diagram tříd akce RESTRouter. Po příjmu ESB zprávy bude zavolána metoda `process()` třídy `RESTRouter`, ve které bude vytvořena instance třídy `ParamExtractor`. Třída `ParamExtractor` bude sloužit pro získání parametrů HTTP požadavku z ESB zprávy. Pro vytvoření HTTP metody bude sloužit abstraktní třída `MethodBuilder`. Třídu `MethodBuilder` budou rozšiřovat třídy `POSTBuilder`, `GetBuilder`, `DELETEBuilder` a `PUTBuilder`, jenž budou sloužit pro vytvoření instance konkrétní HTTP metody. Pro konverzi JAXB objektů do XML, JSON, a naopak je definováno

rozhraní `MarshallingBuilder`. Třídy `MarshallingXML`, resp. `MarshallingJSON`, implementují rozhraní `MarshallingBuilder` a provádějí samotnou konverzi do XML, resp. do JSON. Pro zpracování HTTP odpovědi bude vytvořena třída `ResponseBody`, která bude také pomocí rozhraní `MarshallingBuilder` konvertovat XML a JSON dokumenty v těle HTTP odpovědi na JAXB objekty.

9.2 Návrh vstupní části

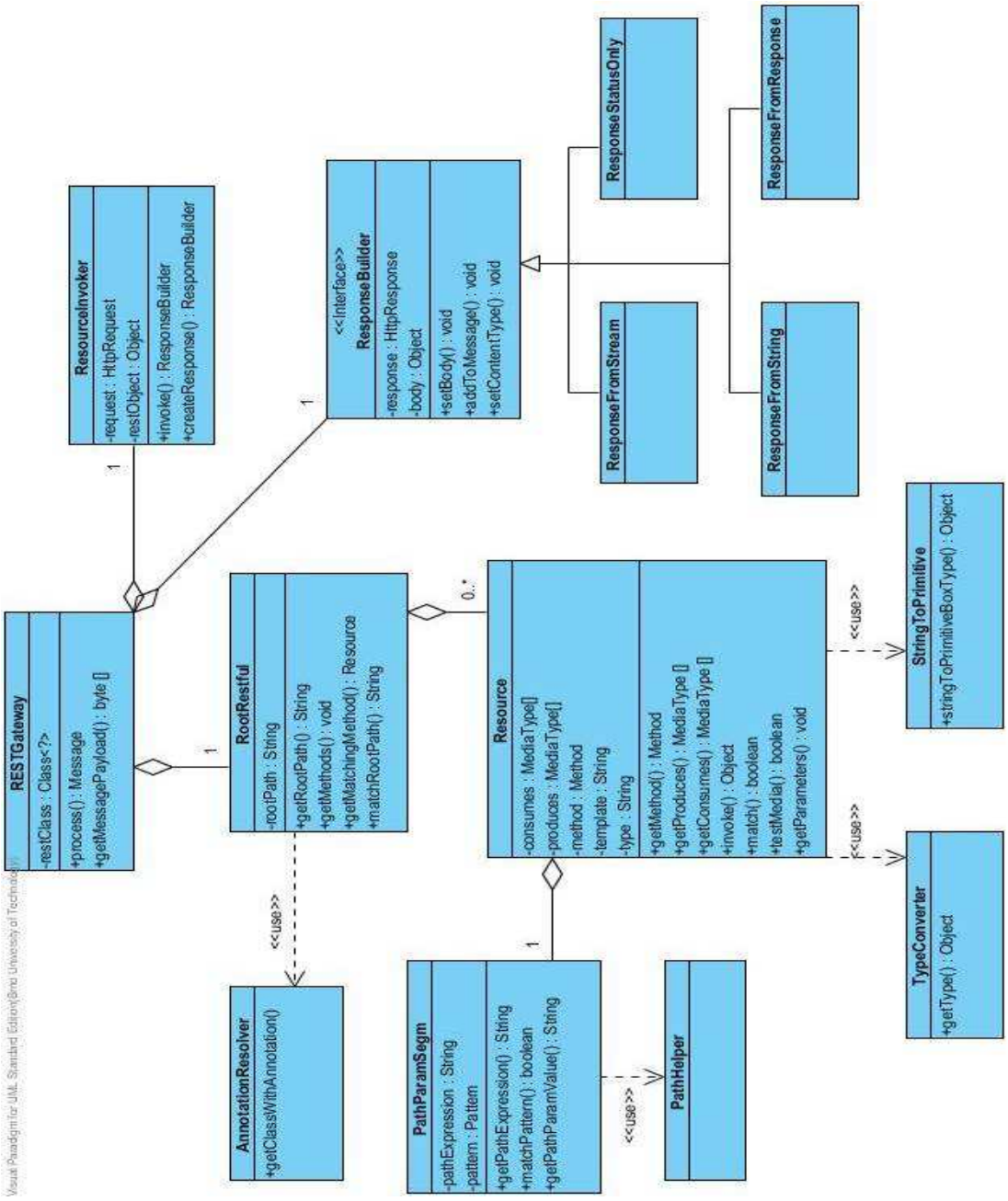
Pro příjem REST požadavků bude použita HTTP Gateway, která byla popsána v 7. kapitole. V konfiguračním souboru bude moci uživatel nastavit vzor URL, které bude HTTP Gateway přijímat. HTTP Gateway vytvoří z požadavku ESB zprávu, kterou zašle do akce RESTGateway.

Uživatel vytvoří zdrojovou třídu, ve které se budou vyskytovat metody, které budou anotovány anotacemi podle specifikace JAX- RS. Do konfiguračního souboru RESTGateway uživatel zadá cestu k jím vytvořené zdrojové třídě. Na obrázku 9.3 se nachází stavový diagram RESTGateway, kde můžeme vidět hlavní kroky při zpracování požadavku. Při inicializaci bude RESTGateway hledat zdrojovou třídu, buď na uživatelem zadané cestě, nebo ve standardním umístění. Pokud zdrojová třída nebude nalezena, tak služba vůbec nebude umístěna na server.



Obrázek 9.3: Stavový diagram akce RESTGateway

Když proběhne inicializace úspěšně, bude služba umístěna na server a bude čekat na příchozí ESB zprávu. Při příchodu zprávy, bude její obsah zpracován a podle parametrů zprávy, bude vyhledána metoda, která by mohla daný požadavek obsloužit. Jestliže žádná vhodná metoda nebude nalezena, tak bude zpět klientovi zaslána HTTP odpověď s chybovým kódem. Pokud vhodná metoda bude nalezena, tak bude zavolána a potřebné parametry HTTP požadavku budou do ní vloženy pomocí parametrů. Návratová hodnota metody bude poté zpracována do HTTP požadavku a odeslána zpět klientovi.



Obrázek 9.4: Schematický diagram tříd RESTGateway

Na obrázku 9.4 se nachází diagram tříd služby RESTGateway. Hlavní třídou je metoda RESTGateway, jejíž metoda `process()`, bude volána při příchodu zprávy. Metoda `process()` bude řídit celý proces od příchodu zprávy až do odeslání odpovědi. Třída RESTGateway vytvoří instanci třídy `RootRestful`, která analyzuje uživatelem vytvořenou zdrojovou třídu a podle toho vytvoří pro každou nalezenou zdrojovou metodu instanci třídy `Resource`. Třída `Resource` tedy

bude představovat jednu zdrojovou metodu a budou v ní uloženy všechny parametry potřebné k nalezení a zavolání metody. Pro zavolání konkrétní zdrojové metody bude sloužit třída `ResourceInvoker`. Pomocí třídy `PathParamSegm` budeme hledat, které URL šablony zdrojových metod jsou kompatibilní s URL adresou v HTTP požadavku. Třída `PathParamSegm` bude částečně převzata z RESTEasy. Ve zdrojovém kódu budou přesně vyznačeny převzaté části. Při implementaci výše uvedených tříd budeme využívat RESTEasy knihovnu *jaxrs-api-2.0.1.GA.jar*, která definuje rozhraní pro příjem REST požadavků a knihovnu *resteasy-jaxrs-2.0.1.GA.jar*, jenž toto rozhraní implementuje.

HTTP odpovědi budou tvořeny pomocí rozhraní `ResponseBuilder`, které bude implementováno čtyřmi třídami podle toho, jaký návratový typ bude mít zdrojová metoda, která bude volána. V tabulce 9.3 jsou uvedeny všechny návratové typy zdrojových metod, které bude RESTGateway podporovat.

Třídy AnnotationResolver, PathHelper, TypeConverter, StringToPrimitive jsou pouze pomocné a jsou zcela převzaty z RESTEasy. Třída `AnnotationResolver` nalezne kořenovou zdrojovou třídu v hierarchii tříd. Třída `PathHelper` pomáhá při porovnávání URL šablony s URL. Pomocí třídy `TypeConverter` je možné konvertovat objekt `String` na požadovaný primitivní datový typ a třída `StringToPrimitive` nalezne standardní hodnotu pro primitivní datový typ. Ze specifikace JAX-RS budou podporovány anotace popsané v tabulce 9.2:

Anotace	**Popis a podpora**
@GET	Přiřadí HTTP požadavek s metodou GET zdrojové metodě.
@POST	Přiřadí HTTP požadavek s metodou POST zdrojové metodě.
@PUT	Přiřadí HTTP požadavek s metodou PUT zdrojové metodě.
@DELETE	Přiřadí HTTP požadavek s metodou DELETE zdrojové metodě.
@Produces	Její parametr bude vložen do hlavičky Content-Type v HTTP odpovědi, ale jen pokud bude metoda standardně ukončena. Jestliže bude v metodě vyvolána nezachycená výjimka, tak hlavička Content-Type aktualizována nebude. Do parametru anotace bude možné vložit více MIME typů.
@Consumes	Tato anotace bude omezovat zdrojovou metodu na příjem požadavku pouze s určitým MIME typem v hlavičce Content-Type. Do parametru anotace bude možné vložit více MIME typů a bude stačit, aby v Content-Type požadavku byl alespoň jeden kompatibilní MIME typ.
@Path	Tuto anotaci bude možné použít stejně jako ve JAX-RS specifikaci pro třídu a pro metodu a bude určovat URL cestu k metodě. Jestliže bude anotace přiřazena k metodě, tak v ní bude možné stejně jako v JAX-RS možné definovat URL šablonu.
@PathParam	Anotace bude fungovat tak jako v JAX-RS. Podle šablony v `@Path` injektuje do metody URI cestu. Může anotovat primitivní datové typy a `String`.
@QueryParam	`@QueryParam` bude fungovat podle JAX-RS, jen bude možné anotovat pouze primitivní datové typy a `String`.
@HeaderParam	`@HeaderParam` bude také implementována podle JAX-RS a stejně jako v anotaci `@QueryParam` bude podporovat String a primitivní datové typy.
@DefaultValue	Také podle specifikace JAX-RS. Uživatel bude muset dávat pozor na to, aby její hodnotu, bylo možné přetypovat na datový typ parametru, který anotuje.

Tabulka 9.2: Podporované anotace v RESTGateway.

Anotace `@CookieParam` a `@MatrixParam` podporovány nebudou, protože nejsou příliš používané, a proto bude lepší se soustředit na ostatní anotace.

Pro vložení těla HTTP požadavku do zdrojové metody bude sloužit datový typ `InputStream`. Uživatel pouze definuje metodě parametr typu `InputStream` a ten při volání metody bude naplněn daty z těla HTTP požadavku. Parametry zdrojové metody musí být buď typu `InputStream`, nebo musí být anotovány `@PathParam`, `@QueryParam`, `@HeaderParam` nebo `@DefaultParam`. Jinak volání metody skončí chybou.

Návratový typ	Třída, která vytvoří odpověď
Response	ResponseFromResponse
String	ResponseFromString
OutputStream	ResponseFromStream
void	ResponseStatusOnly
NULL	ResponseStatusOnly

Tabulka 9.3: Povolené návratové typy zdrojových metod

10 Implementace

V této kapitole si podrobně popíšeme vlastní implementaci služeb RESTRouter a RESTGateway. Implementace vychází z návrhu uvedeném v kapitole 9, proto na něj budeme často odkazovat. Budou zde vysvětleny použité postupy při vytváření obou služeb.

Obě akce jsou implementovány v jazyce Java. Akce jsou vytvořené ve vývojovém prostředí Eclipse verze Helios pod operačním systémem Windows 7. Pří vývoji akcí byly využity knihovny z projektu RESTEasy.

10.1 Implementace RESTRouteru

Hlavní třídou této akce je třída `RESTRouter`. Její metoda `process()` bude volána vždy, když přijde akci RESTRouter zpráva. Metoda `process()` má jeden parametr a tím je právě ESB zpráva, která metodu `process()` vyvolala. ESB zpráva je metodě `process()` předána ve formě objektu `Message`. V metodě `process()` se vytváří instance třídy `ParamExtractor`, kde jsou z ESB zprávy vytaženy všechny parametry HTTP požadavku a jsou uloženy do objektu `ParamProvide`r, kde k nim lze snadno přistupovat. Na výpisu 10.1 je zobrazena metoda `getParams(),` ve které je vidět, jak je možné získat data z ESB zprávy.

```
private HashMap<String, String> getParams(Message message) {
 HashMap params;
 params=(HashMap<String, String>)message.getBody().get("params");
        return params;
}
```

Výpis 10.1: Metoda getParams().

Nejprve je potřeba získat tělo ESB zprávy pomocí metody `getBody()`, čímž dostaneme objekt `Body`. Konkrétní položku v těle zprávy vrací metoda `get(String arg0),` které jako parametr předáme název, pod kterým je položka umístěna. Pokud použijeme metodu `getBody()` bez parametrů, získáme položku ve standardním umístění. Na výpisu 10.1 tedy získáme mapu s parametry pro HTTP požadavek, která je v ESB zprávě umístěna pod jménem `params`, jak bylo uvedeno v kapitole 9.1.

V návrhu bylo uvedeno, že třída `RESTRouter` dědí třídu `HttpRouter`, aby mohla poskytovat zpětnou kompatibilitu pro služby, které třídu `HttpRouter` používají. Při dědění třídy HttpRouter jsme narazili na několik problémů, které bylo potřeba vyřešit.

První problém nastal u konstruktoru. Akce v JBoss ESB definují konstruktor s jedním parametrem typu `ConfigTree`, který představuje nastavení akce. Pomocí instance třídy `ConfigTree` můžeme například přistupovat k atributům akce, které jsou definovány v konfiguračním souboru. Pro naše potřeby stačí znát dvě metody definované ve třídě `ConfigTree`. Metodu `getAttribute(String name),` která vrací hodnotu atributu v konfiguračním souboru a druhou metodou je `getRequiredAttribute(String name)`, jenž vrací také hodnotu atributu, ale pokud jej v konfiguračním souboru nenajde, tak vyvolá výjimku `ConfigurationException`. Konstruktor třídy `HttpRouter` kontroluje, zda se

v konfiguračním souboru nachází povinné parametry pro HTTP požadavek, tedy URL a metoda. Pokud některý z těchto parametrů nenajde, tak vyvolá výjimku `ConfigurationException` a inicializaci akce ukončí, jak můžeme vidět na výpisu 10.2, kde je pomocí metody `getRequiredAttribute` získáno URL HTTP požadavku. Třída `RESTRouter` dědí `HttpRouter`, a proto musí ve svém konstruktoru nejdříve ze všeho volat konstruktor `HttpRouteru`.

```
public HttpRouter(ConfigTree config) throws ConfigurationException{
 try {
   endpointUrl=new URL(config.getRequiredAttribute("endpointUrl"));
 } catch (MalformedURLException e) {
     throw new ConfigurationException("Invalid endpoint URL ...) ;
           }}
```

Výpis 10.2: Konstruktor HttpRouteru.

Zde ovšem nastává problém, jelikož RESTRouter umožňuje URL i metodu definovat pomocí zprávy, a proto se v konfiguračním souboru metoda ani URL nacházet nemusí. Pokud v tomto případě zavoláme konstruktor `HttpRouteru`, tak bude inicializace akce přerušena, protože `HttpRouter` metodu ani URL samozřejmě nenajde. Odchytit výjimku `ConfigurationException` nelze, protože konstruktor nadřazené třídy musí být na prvním místě v konstruktoru podtřídy. Musíme tedy před voláním konstruktoru upravit instanci třídy `ConfigTree` tak, aby atributy URL a metodu vždy obsahovala. Proto jsem vytvořil metodu `preProcess(ConfigTree config)` ve třídě `RESTRouter`, která vkládá povinné atributy do objektu `ConfigTree`, pokud se zde nenachází. Metodu `preProcess(ConfigTree config)` ale nesmíme volat před příkazem zavolání konstruktoru HttpRouteru, proto ji vložíme do těla příkazu volání konstruktoru takto: `super(preProcess(config))`. Tím docílíme toho, že konstruktor `HttpRouteru` bude v prvním příkazu, ale před jeho zavoláním se ještě provede metoda `preProcess`. Metoda `preProcess` musí vracet instanci třídy `ConfigTree` a samozřejmě musí být statická, protože v době jejího volání ještě neexistuje instance třídy `RESTRouter`.

Druhý problém zdědění `HttpRouter` je ten, že všechny jeho metody a atributy jsou privátní. Tedy k nim nemůžeme přistupovat ani v podřazené třídě. Jednou z možností je změnit privátní metody a atributy `HttpRouteru` na veřejné, tím by ale vznikla nová verze `HttpRouteru`, která by nemusela být kompatibilní se stávajícími aplikacemi. Ani znovu implementovat privátní metody `HttpRouteru` není vhodné z hlediska udržovatelnosti kódu. Po konzultaci s technickým vedoucím jsme se proto rozhodli využít balíku *java.lang.reflect*, který zahrnuje třídy umožňující volat metody a přistupovat k atributům `HttpRouteru`, přestože jsou privátní. Na výpisu 10.3 je volána privátní metoda `setRequestHeaders` (`HttpMethodBase method, Message message`) třídy `HttpRouter`, která z konfiguračního souboru načte hlavičky HTTP požadavku.

Nejdříve musíme získat referenci na metodu `setRequestHeaders`, což provedeme pomocí metody `getDeclaredMethod()`, které předáme název volané funkce a seznam jejich parametrů. Tím získáme instanci třídy `Method`, která představuje volanou metodu. Jelikož je metoda `setRequestHeaders` privátní, tak pomocí metody `setAccessible()` nastavíme, aby při jejím volání byl ignorován modifikátor `private`. Nakonec metodou `invoke()`, které předáme instanci třídy a její parametry, zavoláme metodu `setRequestHeaders`.

```
Class<?> arg[] = new Class[2];
arg[0] = HttpMethodBase.class;
arg[1] = Message.class;
Object arglist[] = new Object[2];
arglist[0] = method;
arglist[1] = message;
Method privSetRequestHeaders = null;
privSetRequestHeaders =
   HttpRouter.class.getDeclaredMethod(„setRequestHeaders“,arg);
privSetRequestHeaders.setAccessible(true);
privSetRequestHeaders.invoke(this,arglist);
```

Výpis 10.3: Volání privátní metody setRequestHeaders.

Pro vytvoření a zaslání HTTP požadavku jsem použil třídy z balíku *org.apache.common.httpclient*. Nejdříve je potřeba vytvořit instanci HTTP metody. K tomu slouží abstraktní třída `MethodBuilder`, kterou implementují třídy z balíku *RESTservice.method* podle toho, kterou HTTP metodu je potřeba vytvořit. Jelikož tvorba konkrétní HTTP metody je vždy ve zvláštní třídě, tak nebude problém později doplnit RESTRouter o podporu dalších HTTP metod. HTTP metodě je pak potřeba přiřadit URL a nastavit hlavičky. Pro odeslání požadavku jsem použil třídu `HttpClient`, jejíž metoda `execute(HttpMethod method)` odešle HTTP požadavek.

RESTRouter poté čeká na odpověď, která je následně zpracována v metodě `getResponse()` ve třídě `ResponseBody`. HTTP odpověď musí být zkonvertována do ESB zprávy, která pak bude zaslána další akci. Z hlavičky HTTP odpovědi bude vytvořen objekt `HttpResponse`, který bude vložen do ESB zprávy. Tělo HTTP odpovědi bude zkonvertováno podle hlavičky Content-Type na objekt `String`, pokud v těle bude text, nebo na `byte[]`, pokud budou v těle HTTP odpovědi binární data. Podle hlavičky Content-Type a podle toho, jestli je v ESB zprávě uložena JAXB třída, se rozhodne, zda bude tělo HTTP odpovědi konvertováno na JAXB objekt. Nakonec tělo HTTP požadavku také vložím do ESB zprávy.

Pro automatickou konverzi JAXB objektů jsou použity knihovny z RESTEasy *jaxb-api.jar*, která představuje rozhraní pro konverzi a *jaxb-impl.jar*, která toto rozhraní implementuje. Tyto knihovny jsou použity třídami `MarshallingJSON` a `MarshallingXML`. Funkce `marshall` ze třídy `MarshallingXML` je zobrazena na výpisu 10.4.

```
public String marshall(Object jaxb) {
  StringWriter writer = new StringWriter();
  try {
    JAXBContext context =JAXBContext.newInstance(jaxb.getClass());
    Marshaller marshaller = context.createMarshaller();
    marshaller.marshal(jaxb, writer);
 } catch (JAXBException e) {
    System.out.println("Problem occured when marshalling");
    e.printStackTrace();
    return null; }
return writer.toString();}
```

Výpis 10.4: Metoda marshall třídy MarshallingXML.

Nejdříve vytvoříme instanci třídy `JAXBContext`, která poskytuje vstupní rozhraní do JAXB API. Při vytváření instance je potřeba specifikovat cestu k JAXB třídě nebo vložit JAXB třídu přímo. Potom vytvoříme objekt třídy `Marshaller`, který obsahuje metody potřebné ke konverzi JAXB objektu na XML dokument. Samotnou konverzi pak provedeme metodou `marshal()` třídy `Marshaller`, které vložíme instanci JAXB třídy a objekt, kam se výsledek konverze zapíše.

10.2 Implementace RESTGateway

Hlavní třídou akce RESTGateway je `RESTGateway`. V konstruktoru této třídy proběhne nalezení zdrojové JAX-RS třídy vytvořené uživatelem. Zdrojová třída je nejprve hledána na cestě, kterou uživatel zadá do konfiguračního souboru pod atributem `RESTClass`. Metodou `getAttribute()` třídy `ConfigTree` získáme hodnotu atributu `RESTClass`. Jestliže volání metody `config.getAttribute(„RESTClass“)` vrátí NULL, tak uživatel cestu nezadal, a proto se třída bude hledat v balíku `restGateway`. Pokud ani zde nebude zdrojová třída nalezena, tak konstruktor vyvolá výjimku `ConfigurationException`, čímž bude akce ukončena. Po nalezení zdrojové třídy bude tato třída instanciována metodou `newInstance()`. V případě, že zdrojovou třídu nepůjde instanciovat, například protože nebude obsahovat konstruktor bez parametrů, bude také vyvolána výjimka `ConfigurationException`. Potom bude zdrojová třída zpracována pomocí objektu `RootRestful`.

V `RootRestful` nejdříve nalezneme kořenovou zdrojovou třídu, která je anotována anotací `@Path`, což může být samotná zdrojová třída nebo některá z jejích nadtříd. K nalezení kořenové zdrojové třídy použijeme třídu `AnnotationResolver`, která je převzata z RESTEasy, jak bylo uvedeno v návrhu. Anotace `@Path` obsahuje kořenové URL akce, které si uložíme do objektu `RootRestful`. Objekt `RootRestful` také načte všechny zdrojové metody, tedy takové metody, které jsou anotovány některou s anotací `@GET`,`@POST`, `@PUT` nebo `@DELETE`. Samotné načtení provádí metody `getMethods()` a `getRESTMethod()`, jenž jsou zobrazeny na výpisu 10.5.

```
public void getMethods(Class<?> restClass) {
   for (Method m : restClass.getMethods()) {
     getRESTMethod(m);
   }
}
public boolean getRESTMethod(Method m) {
   for (Annotation a : m.getAnnotations()) {
    HttpMethod met =
        a.annotationType().getAnnotation(HttpMethod.class);
    if (met != null) {
       resources.add(new Resource(m, met.value()));
       return true;
     }
   } return false;
}
```

Výpis 10.5: Metody getMethods() a getRESTMethod().

Metodě `getMethods()` předáme třídu se zdrojovými metodami. V této třídě a v jejich nadtřídách budou zdrojové metody hledány. Pro hledání metod využijeme metodu `getMethod()` třídy `Class`, která vrátí všechny metody dané třídy v podobě instancí třídy `Method` balíku *java.lang.reflection*. Přes všechny tyto metody budeme iterovat, v metodě `getRESTMethod()` budeme zjišťovat, zda je daná metoda zdrojová. Třída `Method` obsahuje metodu `getAnnotations()`, kterou použijeme, abychom získali všechny anotace patřící dané metodě. V získaných anotací pak bude hledat tu anotaci, která obsahuje metaanotaci `@HttpMethod`, protože právě tuto metaanotaci musí obsahovat všechny anotace určující metodu HTTP požadavku. Když metoda `getRESTMethod()` nalezne zdrojovou metodu, tak vytvoří instanci třídy Resource, která představuje jednu zdrojovou metodu.

Při vytváření instance třídy `Resource` se nejprve získají všechny anotace metody, kterou reprezentuje, kromě HTTP metody se ještě budou hledat anotace `@Path`, `@Consumes` a `@Produces`. Jestliže bude nalezena anotace `@Path`, tak její URL šablona bude připojena za kořenové URL akce. Výsledkem bude adresa, na které bude metoda přijímat požadavky. Hodnoty anotací `@Consumes` a `@Produces` budou uloženy jako objekty `MediaType`. `MediaType` je třída JAX-RS API, která představuje jeden MIME typ. Třída `MediaType` obsahuje metodu `isCompatible()`, díky které můžeme snadno zjistit, zda jsou dva MIME typy kompatibilní. Třída `Resource` také kontroluje, zda má zdrojová metoda podporovaný návratový typ a pokud ne, tak vypíše varování. Parametry zdrojových metod získáme metodou `getParameterTypes()` třídy Method a anotace, které parametry anotují, vrátí metoda `getParameterAnnotations()`.

Pokud vše výše uvedené proběhne v pořádku, tak RESTGateway bude umístěna na server a připravena přijímat zprávy s HTTP požadavky. Http Gateway vytvoří z hlavičky HTTP požadavku objekt `HttpRequest`, který vloží do ESB zprávy a zašle akci RESTGateway.Tělo HTTP požadavku je umístěno do standardního umístění v ESB zprávě. Z ESB zprávy pak `HttpRequest` získáme metodou `HttpRequest.getRequest(Message message)`. Z objektu `HttpRequest` pak můžeme získat URL, metodu i hlavičky HTTP požadavku. Dále musíme URL požadavku srovnat s kořenovým URL akce, což provádí metoda `matchRootPath()` třídy `RootRestful`. Pokud si dané URL neodpovídají, tak nemá smysl hledat zdrojovou metodu a klientovi hned odpovíme chybovým kódem 404 „Not Found“. V případě, že si URL budou odpovídat, tak můžeme začít hledat zdrojovou metodu, která bude moci HTTP požadavek obsloužit. Hledání provádí metoda `getMatchingMethod()` třídy `RootRestful`, která prochází všechny instance třídy `Resource` a pomocí její metody `match()` zjistí, zda zdrojová metoda může HTTP požadavek obsloužit.

Když je zdrojová metoda nalezena, můžeme ji zavolat. Pro volání zdrojových metod slouží třída `ResourceInvoker`, která zavolá metodu `invoke()` konkrétní zdrojové metody. Na výpisu 10.6 je část metody `invoke()`.Před zavoláním zdrojové metody musíme naplnit její parametry. Naplněné parametry volané metody jsou ukládány do pole `args`. Budeme tedy iterovat přes všechny její parametry, a pokud narazíme na typ `InputStream`, tak jej naplníme tělem HTTP požadavku a vložíme do pole `args`. Jestliže parametr není `inputStream`, tak musí být anotován některou z JAX-RS anotací, jinak bude vyvolána výjimka `InternalServerErrorException`, která značí, že klientovi se má poslat odpověď 500 „Internal Server Error“. Na výpisu 10.6 je ještě ukázáno, jak se zpracuje anotace `@PathParam`. Nejprve získáme hodnotu parametru anotace `@PathParam` pomocí metody `value()`, a pak zavoláme funkci `setParameterPath()`, která z šablony získá část URL cesty a vloží ji do pole `args`. Podobně se zpracují parametry, které jsou

anotovány `@HeaderParam` a `@QueryParam`. Když jsou všechny parametry metody naplněny, tak již můžeme zavolat metodu `invoke` třídy `Method`, která zdrojovou metodu vykoná a vrátí návratovou hodnotu metody.

```
public Object invoke(Object instance, HttpRequest request, byte[] body)
throws ....
{             ....
 Object[] args = new Object[this.paramTypes.length];
 String pathParamName;
 Object ret;
   for (Class<?> param : paramTypes) {
      if (param.equals(InputStream.class)) {
        args[0] = new ByteArrayInputStream(body);
      } else {
          if (paramAnotations[i] == null) {
            logger.error("Only parameters with annotations and
            InputStream are supported. ");
            throw new InternalServerErrorException("Only parameters..");
          } else {
             for (Annotation anot : paramAnotations[i]) {
               if (anot instanceof PathParam) {
                 pathParamName = ((PathParam) anot).value();
              setParameterPath(i,args,pathParamName,paramAnotations[i]);
               k++;
           } else if (anot instanceof QueryParam) {
                       .......
           } else if (anot instanceof HeaderParam) {
                       .......
     }
return method.invoke(instance, args);}
```

Výpis 10.6: Volání zdrojové metody.

.

Třída `ResourceInvoker` také vytvoří odpověď pro klienta. Pro tvorbu odpovědi využívá třídy z balíku *restgateway.response* podle toho, jaký je návratový typ odpovědi. Na výpisu 10.7 je část metody `setBody()` třídy `ResponseFromResponse`, kde získáme tělo HTTP odpovědi. Podrobněji si popíšeme, jak se vytváří odpověď z objektu `Response`. Třída `Response` je abstraktní, a proto ji musíme nejdřív zdědit třídou `ResponseFactory` a implementovat abstraktní metody `getStatus()` pro získání stavového kódu HTTP odpovědi, dále metodu `getEntity()` pro získání těla HTTP odpovědí a metodu `getMetadata()`, která vrací hlavičky HTTP odpovědi. Instanci `ResponseFactory` pak získáme voláním `ResponseFactory respEnt = new ResponseFactory((Response)body)`, kde `body` je `Response` objekt vrácený zdrojovou metodou. Voláním implementovaných metod pak získáme stavový kód, hlavičky i tělo HTTP požadavku.

```
public void setBody(Object body) {
  ResponseFactory respEnt = new ResponseFactory((Response)body);
  .....
  if(respEnt.getEntity() ==null){
    this.body="";
   } else if (respEnt.getEntity() instanceof OutputStream) {
   this.body=((ByteArrayOutputStream)respEnt.getEntity()).toByteArray();}
   else if (respEnt.getEntity() instanceof String) {
       this.body=(respEnt.getEntity());
   } else {
       logger.warn("Unsupported Entity object. Only String and
       OutputStream are supported");
       this.body="";
   } return ;}
```

Výpis 10.7: Získaní těla HTTP odpovědi z objektu Response.

Jak jsme popsali v návrhu, uživatel může do objektu `Response` jako tělo odpovědi vložit buď `OutputStream`, nebo `String`. Tělo odpovědi získáme metodou `getEntity()` a nejprve musíme otestovat, zda není NULL, protože v tom případě tělo odpovědi zůstane prázdné. Pak pomocí operátoru `instanceof` zjistíme, zda metoda `getEntity()` vrátí `OutputStream`. Pokud vrací, tak tělo požadavku uložíme do objektu `ByteArrayOutputStream`, protože `OutputStream` je abstraktní třída. Stejně vyzkoušíme, jestli uživatel do `Response` neuložil `String`. V případě že metoda `getEntity()` vrátí ještě jiný datový typ, tak vypíšeme varování a tělo HTTP odpovědi necháme prázdné. Tělo HTTP odpovědi potom uložíme do standardního umístění v ESB zprávě.

V návrhu je také uvedeno, že uživatel může ve zdrojové metodě vyvolat výjimku `WebApplicationException`. Tuto výjimku musíme tedy při volání zdrojové metody odchytávat, což je zobrazeno na výpisu 10.8.

```
try {
      responseBody = resource.invoke(restObject, request,body);
} catch (InvocationTargetException e) {
 if (e.getCause() instanceof WebApplicationException) {
    WebApplicationException ee =((WebApplicationException)
    e.getCause());
    ResponseFromResponse resp=new
    ResponseFromResponse(ee.getResponse());
    resp.setBody(ee.getResponse());
    return resp;
 }else{
 .....}}
```

Výpis 10.8: Zachytávání výjimky WebApplicationException.

Jelikož zdrojovou metodu voláme dynamicky pomocí balíku *java.lang.reflection*, nemůžeme jednoduše odchytávat výjimky vyvolané zdrojovou metodou přímo. Musíme nejprve odchytnout obecnou výjimku `InvocationTargetException`, a v ní pak pomocí metody `getCause()` zjistit, co stojí za vyvoláním této výjimky. Pokud metoda `getCause()` vrátí objekt typu WebApplicationException, tak víme, že uživatel vyvolal výjimku `WebApplicationException` a můžeme ji zpracovat. Pro hlavičku HTTP odpovědi slouží objekt `HttpResponse`, který pak bude vložen do ESB zprávy a zpráva zaslána Http Gateway.

11 Testování a vyhodnocení

Jedním z bodů zadání této práce je otestovat RESTRouter a RESTGateway na několika příkladech využívajících REST komunikaci. Proto provedeme několik funkčních testů, na kterých obě akce otestujeme. Na RESTRouteru otestujeme vytváření REST požadavků a příjem odpovědi. Také otestujeme automatickou konverzi JAXB objektů na dokument XML. V RESTGateway otestujeme anotace určující metody přijímaných REST požadavků. Také otestujeme anotace pro injekci dat z hlavičky REST požadavku a přístup k tělu požadavku. Na závěr otestujeme REST komunikaci mezi RESTGateway a RESTRouterem. Všechny příklady se nachází na přiloženém CD ve složce *testy* i s návodem, jak je spustit.

11.1 Testování RESTRouteru

V prvním příkladě otestujeme, zda se všechny parametry, které uživatel zadal, opravdu vyskytují v HTTP požadavku a zda bude správně zpracována odpověď. Vytvoříme si klientskou aplikaci, která RESTRouteru zašle zprávu s parametry pro HTTP požadavek. Na výpisu 11.1 je vidět tvorba mapy s HTTP parametry v klientské aplikaci. Do mapy vložíme 4 hlavičky, HTTP metodu a URL, na které se má HTTP požadavek zaslat.

```
Map<String, String> map=new java.util.HashMap<String, String>();
map.put("Accept", "text/html,image/png");
map.put("Accept-Charset","utf-8");
map.put(“Moje_Hlavicka“ ,"Moje_hodnota");
map.put(“Dalsi_Hlavicka“ ,"Dalsi_hodnota");
map.put("method","GET");
map.put("url", "http://www.fit.vutbr.cz");
```

Výpis 11.1: Tvorba mapy.

Také vytvoříme pomocnou akci, která bude vypisovat HTTP odpověď. Tuto odpověď RestRouter uloží do ESB zprávy. Hlavičku HTTP požadavku zachytíme pomocí programu Wireshark[1]. Wireshark je síťový analyzátor paketů, který zachycuje příchozí pakety a pomocí filtrů přehledně zobrazuje uživateli jejich obsah. Na výpisu 11.2 je zachycená hlavička HTTP požadavku.

```
GET / HTTP/1.1
Content-Type: text/xml;charset=UTF-8
Accept-Charset: utf-8
Moje_Hlavicka: Moje_hodnota
Accept: text/html,Dalsi_hodnota
User-Agent: Jakarta Commons-HttpClient/3.0.1
Host: www.fit.vutbr.cz
```

Výpis 11.2: Hlavička HTTP požadavku.

[1] http://www.wireshark.org/

Na výpisu 11.2, jsou všechny hlavičky, které jsme vložili do zprávy v klientské aplikaci. HTTP metoda je GET a požadavek je zaslána na adresu *www.fit.vutbr.cz*. Hlavička HTTP požadavku je tedy správná. Pomocná akce vypsala HTML kód *www.fit.vutbr.cz*, což je také správně.

V druhém testu vyzkoušíme automatickou konverzi JAXB objektu do XML. V klientské aplikaci vložíme do mapy parametry na výpisu 11.3.

```
map.put("Content-Type" ,"application/xml");
map.put("method","POST");
map.put("url", "http://localhost:8080/");
```

Výpis 11.3: Parametry HTTP požadavku pro druhý test.

Také vytvoříme pomocnou akci, ve které do ESB zprávy vložíme objekt JAXB třídy `Book`, která je zobrazena v příloze A. Třída `Book` je převzata z [17]. Jelikož jsme do hlavičky Content-Type vložili hodnotu `application/xml`, tak se JAXB objekt bude konvertovat na XML dokument. JAXB objekt ESB zprávy vložíme podle výpisu 11.4.

```
esbMessage.getBody().add("marshall",new Book("Mika Waltari",
"333-333", "Tajemny Etrusk"));
```

Výpis 11.4: Vložení JAXB objektu do ESB zprávy.

Výsledek opět zachytíme pomocí programu Wireshark na výpisu 11.5.

```
POST / HTTP/1.1
Accept-Charset: utf-8
Content-Type: application/xml
User-Agent: Jakarta Commons-HttpClient/3.0.1
Host: localhost:8080
Content-Length: 132

<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book
title="Tajemny Etrusk "> <author> Mika Waltari
</author><ISBN>333-333</ISBN></book>
```

Výpis 11.5: Výsledný HTTP požadavek ve druhém testu.

Tělo HTTP požadavku obsahuje XML dokument, který je vytvořen z JAXB objektu, který jsme vložili do ESB zprávy.

11.2 Testování RESTGateway

Pro testování RESTGateway je vytvořena zdrojová třída s JAX-RS anotacemi, na které budeme testovat chování akce. Zdrojová třída je v příloze B. Testovat budeme na *localhostu*. V prvním testu otestujeme volání metody `test1()`. Metoda `test1()` přijímá na HTTP požadavky na URL *http://localhost:8080//RESTGateway/http/clanek/typ/1*. Toto URL vložíme do webového prohlížeče. Po zaslání požadavku do RESTGateway webový prohlížeč vypíše: „Tento text je v tele HTTP odpovedi a je predan pomoci `OutputStream`.“, což je text, který metoda `test1()` vrací v Objektu `OutputStream`.

Druhý test bude volat metodu `test2(@PathParam("id") String id, @HeaderParam("connection") String connection, @QueryParam("kod") int kod).` Na metodě `test2()` bude testovat anotace parametrů zdrojové metody. Metoda `test2()` má anotaci `@Path("/typ/2/{id}")`, proto bude na adrese *http://localhost:8080//RESTGateway/http/clanek/typ/2/**. Místo „*“ je možné zadat jakýkoli řetězec, který je platný v URL a není to „/“. Pro náš test zvolíme řetězec „auta“. Do požadavku ještě vložíme query parametr *kod=1234*. Do webového prohlížeče tedy můžeme zadat toto: *http://localhost:8080//RESTGateway/http/clanek/typ/2/auta?kod=1234.*

Druhý test také testuje návratový typ `Response`, který vrací stavový kód 200 a 2 hlavičky, *Moje_hlavicka* a *Druha_hlavicka*, a tělo požadavku ve formě objektu `String`. Po zavolání metody výše uvedeným URL prohlížeč vypíše (výpis 11.6):

```
Anotace @PathParam vlozila do metody cestu: auta
Anotace @HeaderParam vlozila do metody: keep-alive
Anotace @QueryParam vlozila do metody: 1234
```

Výpis 11.6: Výsledek testu 2.

Anotace tedy pracují správně. Webové prohlížeče standardně nezobrazují hlavičky HTTP odpovědi, proto jestli chceme zjistit stavový kód a hlavičky, musíme použít, např. LiveHttpHeaders[2]. LifeHttpHeaders je plugin pro webový prohlížeč Firefox[3], který uživateli zobrazí detailní popis hlaviček HTTP požadavků a odpovědí. Při zobrazení hlavičky odpovědi vidíme, že stavový kód je 200 „OK“ a hlavičky *Moje_hlavicka* a *Druha_hlavicka* se zde také nacházejí.

Ve třetím testu budeme testovat vyvolání výjimky `WebApplicationException`. Metoda test3() přijme HTTP požadavek s URL *http://localhost:8080//RESTGateway/http/clanek/typ/3*. V těle metody `test3()` bude vyvolána výjimka, do které vložíme `Response` objekt se stavovým kódem 409 „Conflict“. Při zaslání požadavku webovým prohlížečem je v HTTP odpovědi stavový kód 409.

Ve čtvrtém testu budeme testovat, jak funguje URL šablona u metody `test4()`. Metoda `test4()` je anotována anotací *`@Path("/ahoj/4/{id: [0-9]*}")`*, tedy přijímá pouze takové HTTP požadavky, které za URL cestou */typ/4/* obsahují sekvenci číslic. První vyzkoušíme metodu test4() zavolat pomocí URL: *http://localhost:8080//RESTGateway/http/clanek/typ/4/nazev*. Metoda v tomto případě není zavolána a webovému prohlížeči je odeslán stavový kód 404 „Not Found“. Poté zkusíme zavolat metodu pomocí URL: *http://localhost:8080//RESTGateway /http/clanek/typ/4/1234*. Nyní je metoda zavolána a potvrzení odesláno webovému prohlížeči.

11.3 Testování komunikace mezi RESTRouterem a RESTGateway

V těchto testech budeme testovat, jak bude fungovat volání zdrojových metod z RESTRouteru a jak RESTRouter zpracuje odpovědi, které mu zašle RESTGateway. Také si vyzkoušíme volání jinými HTTP metodami než GET.

[2] https://addons.mozilla.org/en-US/firefox/addon/live-http-headers/

[3] http://firefox.mozilla.cz/

V prvním testu budeme volat metodu `test5()`, která přijímá HTTP požadavky, jenž mají metodu POST a v těle je dokument JSON. Dokument JSON, který metoda `test5()` přijala, vypíše na standardní výstup a zašle odpověď zpět ve formě dokumentu XML, který HTTPRouter zkonvertuje na JAXB objekt. V pomocné akci obsah JAXB objektu vypíše na standardní výstup. Metoda `test5()` definuje parametr typu `InputStream`, pomocí kterého do metody injektujeme tělo HTTP požadavku. Na dokument JSON budeme konvertovat stejný JAXB objekt, jako ve druhém testu RESTRouteru. Na výpisu 11.7 jsou parametry požadavku, který bude přes RESTRouter zaslán RESTGateway.

```
map.put(“Content-Type“ ,"application/json");
map.put("method","POST");
map.put("url","http://localhost:8080//RESTGateway/http/clanek/
typ /5");
```

Výpis 11.7: Parametry HTTP požadavku.

Aby byl JAXB objekt zkonvertován na JSON dokument, musíme do hlavičky Content-Type vložit `application/json`. Na straně RESTRouteru také vytvoříme pomocnou akci, která vypíše HTTP odpověď zaslanou RESTGateway.

Po odeslání požadavku vypsala metoda `test5()` na standardní výstup:

```
Telo HTTP pozadavku je:  {"book":{"@title":" Tajemny Etrusk ",
 "author":{"$":"Mika   Waltari"}, "ISBN": {"$":"222-222"}}}
```

Výpis 11.8: Standardní výstup metody test5().

Na výpisu 11.8 vidíme, že RESTRouter správně zkonvertoval JAXB objekt na JSON dokument. Také injekce těla HTTP požadavku do zdrojové metody pomocí parametru typu `InputStream` proběhla také úspěšně. Na výpisu 11.9 je standardní výstup pomocné akce RESTRouteru, kde je vypsán obsah JAXB objektu, který byl vytvořen z XML dokumentu v odpovědi. Na výpisu 11.9 můžeme vidět, že XML dokument v těle odpovědi byl správně převeden na JAXB objekt.

```
Vypis JAXB objektu:
Autor: Jules Verne
ISBN: 222-222
Nazev: Dva roky prazdnin
```

Výpis 11.9: Standardní výstup pomocné akce RESTRouteru.

V posledním testu budeme pomocí RESTRouteru volat zdrojovou metodu `test6()`, která přijímá pouze HTTP požadavky metodou PUT a tělem `text/plain`. Nejdříve zašleme požadavek pomocí mapy na výpisu 11.10.

```
map.put(“Content-Type“ ,"application/xml");
map.put("method","PUT");
map.put("url","http://localhost:8080//RESTGateway/http/clanek/
typ/6");
```

Výpis 11.10: Parametry HTTP požadavku pro metodu test6().

Když odešleme požadavek na výpisu 11.10, tak RESTRouter vypíše varování „The Response code isn't successuful.(404)“, což je správné chování, protože metoda `test6()` přijímá pouze požadavky s tělem typu `text/plain`. Do hlavičky Content-Type vložíme `text/plain` a odešleme požadavek. Metoda `test6()` se v tomto případě zavolala a vypsala: „Byla zavolana metoda put“.

12 Vyhodnocení

V RESTRouteru byla implementována možnost dynamicky měnit parametry REST požadavků podle aktuální potřeby a tím je RESTRouter mnohem pružnější než již existující akce HttpRouter, kde je vše staticky vloženo v konfiguračním souboru. RESTRouter zároveň umožňuje definovat parametry požadavku stejně jako HttpRouter v konfiguračním souboru, a proto může v podstatě HttpRouter nahradit. RESTRouter podporuje kromě HTTP metod POST a GET, také metody PUT a DELETE.

V RESTRouteru byla taktéž implementována možnost automatické konverze JAXB objektů do formátu JSON, XML a naopak, což je pro uživatele velmi užitečné, protože se již nemusí zabývat tím, jak zpracovat XML nebo JSON dokumenty. JBoss ESB tedy může pomocí JAXB objektu vytvořit XML dokument a ten jednoduše zaslat po síti do RESTEasy, které pak z XML dokumentu pomocí stejné JAXB třídy vytvoří znovu objekt, který může jednoduše zpracovat. V RESTRouteru chybí podpora pro HTTP metodu HEAD, která není příliš používaná, a proto jsme se raději soustředili na implementaci ostatních HTTP metod.

Vhodným rozšířením RESTRouteru by bylo navržení konverze Java objektů ve formě OGNL (Object Graph Navigation Language) do XML a JSON. OGNL je jazyk pro nastavení a čtení atributů Java objektů. Uživatel by pak nemusel pro vygenerování XML nebo JSON dokumentu vytvářet JAXB třídy a vkládat je do ESB zprávy, ale jen by požadovaný objekt specifikoval pomocí OGNL notace. [28]

Akce RESTGateway poskytuje příjemné prostředí pro příjem REST požadavků v JBoss ESB, protože implementuje ty nejpoužívanější prvky z JAX-RS specifikace. Implementovány jsou anotace pro definici metod přijímaných REST požadavků. Zde také chybí podpora pro metodu HEAD ze stejných důvodů jako v RESTRouteru. Také byly implementovány anotace pro definici MIME typů požadavků a odpovědí. Přistupovat k informacím v hlavičce požadavku je možné pomocí anotací, které anotují parametry zdrojových metod. Pomocí speciálních datových typů je možné přistupovat k tělu REST požadavku a vytvářet odpovědi na tyto požadavky.

Uživatel se sice bude muset naučit, jak psát JAX-RS zdrojové třídy, ale odměnou mu budou lehce srozumitelné a spravovatelné aplikace používající RESTGateway. Díky RESTGateway můžeme v JBoss ESB pro příjem REST požadavků využívat stejné JAX-RS zdrojové třídy jako v RESTEasy, což dále vylepšuje integraci JBoss ESB a RESTEasy.

RESTGateway je možné vylepšit tak, že budeme dále rozšiřovat podporu specifikace JAX-RS. Například doplněním podpory pro anotace `@FormParam` a `@CookieParam` atd. Také by bylo vhodné implementovat automatickou konverzi JAXB objektů na dokumenty XML, JSON a další. Zdrojová metoda by pak jako návratovou hodnotu vracela JAXB objekt, který by byl automaticky převeden do dokumentu požadovaného formátu.

Všechny testy RESTRouteru i RESTGateway proběhly správně. REST byl úspěšně integrován do JBoss ESB. Pomocí akcí RESTRouter a RESTGateway je možné snadno vytvářet služby v JBoss ESB, které komunikují pomocí REST. Díky integraci JBoss ESB a RESTEasy je komunikace mezi nimi snadnější a v JBoss ESB je také možné použít zdrojové a JAXB třídy z RESTEasy.

13 Závěr

Cílem této práce bylo navrhnout architekturu podpory REST v JBoss ESB a tuto architekturu v JBoss ESB integrovat. Při integraci REST do JBoss ESB bylo potřeba se zaměřit na integraci s projektem RESTEasy.

V teoretické části jsme si popsali Java EE a protokol HTTP. Také jsme si představili architekturu REST a specifikaci JAX-RS, která velmi ulehčuje vývoj RESTful aplikací implementovaných v jazyce Java. Dále jsme si popsali server JBoss ESB. Na konec teoretické části jsme analyzovali stávající podporu REST v JBoss ESB.

V praktické části jsme nejprve provedli analýzu požadavků na REST komunikaci v JBoss ESB a analyzovali jsme možnosti integrace JBoss ESB a RESTEasy. Po té jsme navrhli dvě akce, které poskytují JBoss ESB možnost komunikace pomocí REST. První akce RESTRouter umožňuje dynamicky pomocí zpráv vytvářet a zasílat REST požadavky. RESTRouter také dokáže konvertovat JAXB objekty na dokumenty XML nebo JSON a naopak, a proto je při komunikaci mezi JBoss ESB a RESTEasy možné použít stejné JAXB třídy.

Druhá akce RESTGateway vytváří rozhraní pro příjem REST požadavků. RESTGateway částečně implementuje specifikaci JAX-RS, čímž umožňuje snadnou implementaci obsloužení REST požadavků a definici odpovědí na požadavky. Implementace specifikace JAX-RS sice není úplná, ale obsahuje ty nejpoužívanější prvky specifikace, a proto bude možné ve většině případů, použít stejné zdrojové třídy v RESTGateway i RESTEasy.

Obě navržené akce byly podle návrhu implementovány v jazyce Java a poté důkladně otestovány na sadě příkladů, které předvádějí možnosti obou vytvořených akcí a zároveň testují správnost implementace. Tím byly všechny body zadání této práce splněny.

Obě implementované akce poskytují uživateli silný nástroj pro vývoj RESTful aplikací pod JBoss ESB. Na obou akcích je však stále co vylepšovat, jak bylo uvedeno v předchozí kapitole. RESTRouter by mohl získat podporu dalších HTTP metod. RESTGateway může dále implementovat specifikaci JAX-RS, hlavně další anotace pro příjem REST požadavků a podporu automatické konverze JAXB objektů.

Seznam obrázků

Seznam tabulek

Literatura

[1] *Oracle* [online]. c2010 [cit. 2011-11-24]. The Java EE 5 Tutorial. Dostupné z WWW: http://download.oracle.com/javaee/5/tutorial/doc/fwbyj.html

[2] JOHNSON, Rod. *Expert One-on-One J2EE Design and Development* . Indianapolis: Wiley Publishing, 2002. 768 s.

[3] JSR 315. *Java Servlet 3.0 Specification*. [s.l.] : Java Community Process, 2009. 207 s. Dostupné z WWW: http://www.jcp.org/en/jsr/detail?id=315

[4] *HTML 4.01 Specification*. [s.l.] : W3C, 1999. 389 s. Dostupné z WWW: <http://www.w3.org/TR/html4/>

[5] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. [s.l.] : W3C, 2008. Dostupné z WWW: <http://www.w3.org/TR/xml/>

[6] JSR 152. *JavaServer Pages 2.0 Specification*. [s.l.] : Java Community Process, 2003. 478 s. Dostupné z WWW: http://www.jcp.org/en/jsr/detail?id=152

[7] JSR 220. *Enterprise JavaBeans 3.0* . [s.l.] : Java Community Process, 2006. 562 s. Dostupné z WWW: http://www.jcp.org/en/jsr/detail?id=220

[8] ZAKHOUR, Sharon, et al. *Java 6*. Praha : Computer Press, 2007. 536 s.

[9] RFC 2616. *Hypertext Transfer Protocol - HTTP/1*.1. [s.l.] : The Internet Society, 1999. 176 s. Dostupné z WWW: http://tools.ietf.org/html/rfc2616

[10] RFC 2617. *HTTP Authentication: Basic and Digest Access Authentication*. [s.l.] : The Internet Society, 1999. 34 s. Dostupné z WWW: http://tools.ietf.org/html/rfc2617

[11] *Architectural Principles of the World Wide Web : REST constraints*. [s.l.] : W3C, 2002. Dostupné z WWW: <http://www.w3.org/TR/2002/WD-webarch-20020830/#rest-constraints>.

[12] FIELDING, Roy. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000. 162 s. Diplomová práce. University of California, Irvine. Dostupné z WWW: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

[13] *Packetizer* [online]. c2010 [cit. 2010-12-16]. Representational State Transfer (REST). Dostupné z WWW: <http://www.packetizer.com/ws/rest.html>

[14] ELIAS, Jose. *Searchsqlserver* [online]. 1999 [cit. 2011-05-10]. ACID (atomicity, consistency, isolation, and durability). Dostupné z WWW: http://searchsqlserver.techtarget.com/definition/ACID

[15] SINGH, Tarandeep. *Geeknizer* [online]. 2009 [cit. 2010-12-18]. REST vs. SOAP – The Right WebService. Dostupné z WWW: http://geeknizer.com/rest-vs-soap-using-http-choosing-the-right-webservice-protocol/

[16] PERICAS-GEERTSEN, Santiago; POTOCIAR, Marek. *Java Community Process* [online]. 2011 [cit. 2011-05-11]. JAX-RS 2.0: The Java API for RESTful Web Services. Dostupné z WWW: <http://www.jcp.org/en/jsr/detail?id=339>

[17] BURKE, Bill. *RESTful Java with JAX-RS*. Sebastopol : O'Reilly Media, 2010. 288 s.

[18] JSR 311. JAX-*RS: The Java API for RESTful Web Services*. [s.l.] : Java Community Process, 2008. 39 s. Dostupné z WWW: http://jcp.org/en/jsr/detail?id=311

[19] ORT, Ed; MEHTA, Bhakti. Oracle [online]. 2003 [cit. 2011-04-04]. Java Architecture for XML Binding (JAXB). Dostupné z WWW: http://www.oracle.com/technetwork/articles/javase/index-140168.html#binsch

[20] *Tiny Java Web Server aka Miniature JW*S [online]. 2009 [cit. 2011-04-15]. Tiny Java Web Server and Servlet Container. Dostupné z WWW: http://tjws.sourceforge.net/

[21] RESTFul Web Services for Java : RESTEasy JAX-RS. [s.l.] : [s.n.], 2008. 208 s. Dostupné z WWW: <http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/pdf/RESTEasy_Reference_Guide.pdf>.

[22] *A JSON Media Type for Describing the Structure and Meaning of JSON Documents*. [s.l.] : IETF, 2010. 28 s. Dostupné z WWW: <http://tools.ietf.org/html/draft-zyp-json-schema-03>

[23] ERL, Thomas. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Boston : Prentice Hall, 2005. 792 s.

[24] JSR 914. *Java Message Service (JMS) API*. [s.l.] : Java Community Process, 2002. 138 s. Dostupné z WWW: http://www.jcp.org/en/jsr/detail?id=914

[25] SHARMA, Rahul ; STEARNS, Beth; NG, Tony. *J2EE Connector Architecture and Enterprise Application Integration*. New Jersey : Prentice Hall, 2001. 416 s.

[26] *Your guide to developing for the JBoss ESB : Programmers Guide*. [s.l.] : [s.n.], 2010. 226 s. Dostupné z WWW: http://www.jboss.org/jbossesb/docs/4.9/manuals/pdf/Programmers_Guide.pdf

[27] *Evget* [online]. 2010 [cit. 2011-12-11]. Fiorano Enterprise Service Bus. Dostupné z WWW: http://www.evget.com/zh-CN/product/2055/feature_en.aspx

[28] *OpenSymphony* [online]. c2004 [cit. 2011-04-20]. OGNL Language Guide. Dostupné z WWW: http://www.opensymphony.com/ognl/html/LanguageGuide/index.html

[29] JSR 222. *Java Architecture for XML Binding*. [s.l.] : Java Community Process, 2006. 376 s. Dostupné z WWW: http://www.jcp.org/en/jsr/detail?id=222

[30] *SOAP Version 1.2 Part 0:* Primer. [s.l.] : W3C, 2007. 100 s. Dostupné z WWW: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

[31] *Oracle* [online]. c2010 [cit. 2011-11-24]. Java EE Containers. Dostupné z WWW: http://download.oracle.com/javaee/5/tutorial/doc/bnabo.html

Seznam příloh

Příloha A: JAXB třída Book

```
package RESTService;

import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlAttribute;

@XmlRootElement(name = "book")
public class Book
{
   private String author;
   private String ISBN;
   private String title;

   public Book()
   {
   }

   public Book(String author, String ISBN, String title)
   {
      this.author = author;
      this.ISBN = ISBN;
      this.title = title;
   }
   @XmlElement
   public String getAuthor()
   {
      return author;
   }
   public void setAuthor(String author)
   {
      this.author = author;
   }

   @XmlElement
   public String getISBN()
   {
      return ISBN;
   }
   public void setISBN(String ISBN)
   {
      this.ISBN = ISBN;
   }
   @XmlAttribute
   public String getTitle()
   {
      return title;
   }

   public void setTitle(String title)
   {
      this.title = title;
   }
}
```

Příloha B: JAX-RS zdrojová třída

```
@Path("clanek")
public class RESTJava {

@GET
@Produces("text/plain")
@Path("/typ/1")
public OutputStream test1()throws WebApplicationException {
    ByteArrayOutputStream response=new ByteArrayOutputStream();
    try {
      response.write("Tento text je v tele HTTP odpovědi a je predan pomoci
      OutputStream.".getBytes());
    } catch (IOException e) {
      e.printStackTrace();
      }
    return response;
     }

@GET
@Produces("text/plain")
@Path("/typ/2/{id}")
public Response test2(InputStream b,@PathParam("id") String id,
@HeaderParam("connection") String connection, @QueryParam("kod") int kod){
     String response="Anotace @PathParam vlozila do metody cestu:"+id+"\n";
     response+="Anotace @HeaderParam vlozila do metody: "+ connection+"\n";
     response+="Anotace @QueryParam vlozila do metody: "+ kod+"\n";
     return Response.status(200).header("Moje_hlavicka",
"jeTady").header("Druha_hlavicka", "jeTadyTaky").entity(response).build();
}

@GET
@Produces("text/plain")
@Path("/typ/3")
public OutputStream test3()throws WebApplicationException {
     OutputStream out=null;
     System.out.println("Jsem v metode test3");
     if(1==1){
       throw new WebApplicationException(Response.
       status(Response.Status.CONFLICT).entity("409 Conflict").build());
     }
return out;
}
@GET
@Produces("text/plain")
@Path("/typ/4/{id: [0-9]*}")
public String test4() {
      String response="Byla zavolana metoda test4.";
return response;
}
```

```
@POST
@Produces("application/xml")
@Consumes("application/json")
@Path("/typ/5")
public String test5(InputStream body){
     byte[] by=null;
      try {
            by = new byte[body.available()];
      } catch (IOException e) {
            e.printStackTrace();
      }
      try {
            body.read(by, 0, by.length);
      } catch (IOException e) {
            e.printStackTrace();
      }
      System.out.println("Jsme v metode test5.");
      String bstr=new String(by);
      System.out.println("Telo HTTP pozadavku je: ");
      System.out.println(bstr);
      return "<?xml version=\"1.0\" encoding=\"UTF-8\"
      standalone=\"yes\"?><book title=\"Dva roky prazdnin \"> <author>Jules
      Verne</author><ISBN>222-222</ISBN></book>";
}

@PUT
@Consumes("text/plain")
@Path("/typ/6")
public void test6() {
      System.out.println("Byla zavolana metoda test6.");
}
}
```

Příloha C: Obsah přiloženého CD