MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

REST a webové služby v jazyce Java

Diplomová práce

Bc. Jiří Kadlec

Brno, jaro 2010

Prohlášení:

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Brně dne 20. května 2010

_____

Podpis

Poděkování:

Tímto bych rád poděkoval vedoucímu práce Ing. Petru Adámkovi za cenné rady a připomínky v průběhu našich konzultací. Dále pak rodině a přátelům, kteří mě podporovali v průběhu celého studia a umožnili mi tím dokončit i tuto diplomovou práci.

Vedoucí práce: Ing. Petr Adámek

Shrnutí

Práce seznamuje čtenáře s architektonickým stylem pro vývoj webových aplikací REST. Podává jeho vysvětlení na intuitivní úrovni pomocí příkladu WWW. Z této intuitivní definice přechází k definici formální a vysvětluje šest základních omezení pro architektury, které se dají označit jako RESTful (architektury splňující podmínku REST).

V dalších sekcích práce aplikuje myšlenky stylu REST na webové služby (vysvětleny v úvodní části). Výsledkem je vymezení pojmu RESTful webových služeb. RESTful služby jsou postaveny do kontrastu s dříve probranými službami postavenými na protokolu SOAP. Uvedené srovnání stanoví hlavní rozdíly v obou přístupech.

Velkou výzvou práce je také představení prvního standardního API pro vývoj RESTful webových služeb v Javě (JAX-RS). Standard JSR-311 přináší celou řadu dobrých myšlenek a nápadů, které jsou v práci probrány a vysvětleny. Některé myšlenky jsou také zpětně konfrontovány s omezeními stylu REST, což přináší hlubší pochopení problematiky.

V diplomové práci jsou představeni čtyři zástupci implementací webových služeb v Javě (dva reprezentanti REST a dva SOAP). Z uvedených zástupců jsou vybráni kandidáti pro závěrečné srovnání přístupů REST a SOAP na praktickém příkladu (jeden REST a jeden SOAP). Příklad demonstruje výhody a nevýhody jednotlivých přístupů od samotného návrhu aplikace, přes implementaci až k porovnání zpráv odesílaných mezi klientem a službou. Jsou hodnoceny všechny relevantní faktory jako například obtížnost implementace, objem přenášených dat, obtížnost konzumace služeb atd. V závěru jsou zhodnoceny výsledky srovnání obou přístupů a definovány typy aplikací, které je vhodnější implementovat pomocí REST či SOAP.

Klíčová slova

REST, JAX-RS, JAX-WS, JAX-RPC, JAXB, Java, JSR-311, HTTP, GET, POST, webové služby, jednotné rozhraní, SOAP, Jersey.

Obsah

Seznam tabulek:

Seznam příkladů:

Seznam obrázků:

Úvod

V poslední době je možné ve světě IT pozorovat rychlý nástup informačních systémů (IS) dostupných pomocí webových prohlížečů. Díky neustálému růstu počtu uživatelů dochází ke zvyšování nároků na propustnost těchto IS. Potřeba vyšší propustnosti je pak hlavním důvodem nasazování moderních IS na více strojů, které tvoří distribuovaný systém. Velmi často se též setkáváme s požadavkem na propojení nových IS s jinými různorodými systémy (tzv. interoperabilita systému).

Architektonický styl REST (*Representational State Transfer*) definuje požadavky na architekturu systému schopného vyhovět uvedeným požadavků. Poprvé byl představen v disertační práci Roye Fieldinga [1], kde byla formulována omezení na architekturu, která odpovídá stylu REST. Systém splňující tato omezení bude schopen řešit jak problémy interoperability, tak problémy velké zátěže a distribuovaného prostředí.

Současný trend vývoje software, který je dostupný pomocí webových prohlížečů a zároveň poskytuje výbornou interoperabilitu systému, je vývoj pomocí webových služeb. Tento způsob vývoje nabízí, kromě zmiňované interoperability, i velmi dobrou možnost znovupoužití jednotlivých služeb (službu je možné využít ve více aplikacích). Velmi často zmiňovanou výhodou webových služeb je také snížení závislosti na konkrétním dodavateli, neboť je možné odebírat software od různých (nezávislých) dodavatelů.

V mojí práci bych rád představil spojení těchto dvou myšlenek, tj. aplikování principu REST na webové služby, zvláště pak na webové služby v jazyce Java. V úvodních kapitolách jsou představeny technologie a pojmy, které jsou nezbytné pro lepší orientaci čtenáře v navazujících kapitolách. Následuje představení principu REST a jeho hlavní myšlenky. Tato myšlenka je aplikována na webové služby (RESTful webové služby), které jsou dále srovnány se službami postavenými na SOAP protokolu. Ze srovnání vyplynou rozdíly, ke kterým se budu v průběhu práce vracet, ať už při představení jednotlivých standardů, nebo v praktickém příkladu v závěru práce.

Hlavní část práce se zabývá standardem JSR-311 JAX-RS. Tento standard vznikl jako reakce na absenci standardu pro budování webových služeb založených na principu REST v jazyce Java. V minulosti byly v Javě pro tento účel používány nejrůznější rámce jako je např. Restlet, ale tento postup nebyl standardizován. JSR-311 tudíž představuje první standardní API pro webové služby REST v Javě.

Dále se práce věnuje konkrétním implementacím a standardům spojeným s vývojem webových služeb v Javě. Představeni jsou zástupci webových služeb postavených na SOAP protokolu (JAX-RPC, JAX-WS) i REST stylu (Servlet API, JAX-RS). Z představených zástupců je vybrán jeden reprezentant pro každý přístup (SOAP, REST) pro srovnání na praktickém příkladu.

Praktická část práce se věnuje porovnání přístupu SOAP a REST na příkladu zjednodušeného inzertního systému. Srovnání je vedeno již od návrhu aplikace, kde zdůrazním nejen odlišnosti, ale i důležité kroky a časté chyby, které jsou pro návrh REST služeb specifické. Následuje srovnání implementací stejných funkcí pomocí standardů pro REST a SOAP (JAX-RS a JAX-WS). Zde jsou sledovány takové faktory, jako je například složitost a efektivita jednotlivých operací z hlediska přenášených dat nebo pružnost definice odpovědí. Na konci této části jsou shrnuty výhody každého z přístupů a vymezeny typy aplikací, kde je výhodné použití SOAP a kde naopak můžeme profitovat z výhod stylu REST.

V závěru práce hodnotím přínosy stylu REST pro architektury splňující jeho omezení. Shrnuji výsledky srovnání přístupů REST a SOAP (teoretické i praktické) a hodnotím i přínos samotného JAX-RS API.

1. Základní pojmy

Z důvodu lepšího pochopení výkladu REST a jeho použití pro webové služby budou vysvětleny pojmy, na kterých budu stavět následující kapitoly. Jedná se o tyto základní celky:

- Protokol HTTP – bázový protokol pro přístup ke zdrojům na webu.
- Webové služby – základní princip, vysvětlení filosofie, SOAP, WSDL.

1.1 Protokol HTTP

Hypertext Transfer Protocol je internetový protokol pracující na aplikační vrstvě podle TCP/IP modelu. Pracuje na principu dotaz-odpověď (*request-response*) mezi klientem a serverem (*client-server* architektura – princip viz 2.2.1). Jedná se o bezstavový protokol – mezi odesláním jednotlivých požadavků server neudržuje žádný stav, více viz 2.2.2. Nejobvyklejším použitím je dotaz (ve formě čistého textu) odeslaný pomocí klienta v podobě internetového prohlížeče, který je následně zaslán na server. Ten zpracuje *klientský požadavek* a pošle *odpovědní zprávu* zpět klientovi. HTTP specifikace definuje pravidla, jak má vypadat formát klientských požadavků a odpovědí serveru. V dnešní době je nejpoužívanější verze HTTP 1.1, která je definována v RFC 2616 [3]. Z této specifikace byly také čerpány podklady pro další podsekce.

1.1.1 Klientský požadavek

Každý klientský požadavek je zahájen specifikováním HTTP metody (definice metod viz tabulka 1.1.1), ta je následována URL (*Uniform Resource Locator*). URL jednoznačně identifikuje zdroj, nad kterým má být metoda provedena. V poslední části prvního řádku je specifikována použitá verze protokolu HTTP. Další řádky obsahují hlavičky upřesňující požadavek (např. jazyk, ve kterém je zdroj požadován). Samotná uživatelská data jsou oddělena od hlaviček jedním prázdným řádkem. Přesný formát klientského požadavku podle HTTP shrnuje následující příklad:

```
#HTTP Metoda# #URL# HTTP/1.x
#obsah hlavičky#
#Prázdný řádek#
#samotná data#
```

Příklad 1.1.1: HTTP klientský požadavek

Definice HTTP metod:

Metody definované protokolem HTTP, které je možné použít v rámci klientských požadavků, shrnuje následující tabulka.[3]

Název metody	Popis
GET	Žádost o reprezentaci dokumentu (přesněji zdroje), definovaného pomocí identifikátoru URL. Tato metoda by neměla být používána na operace, které mohou způsobit vedlejší účinky. Často je používána webovými vyhledávači. Více viz bezpečné metody pod touto tabulkou.
HEAD	Stejný požadavek jako v případě metody GET s tím rozdílem, že jako výsledek je navrácena pouze hlavička odpovědi a nikoliv tělo obsahující zdroj.
POST	Používá se pro odeslání uživatelských dat na server. Typicky se jedná o data z HTML formuláře. Ta jsou uložena jako tělo požadavku, tj. za prázdným řádkem následujícím po hlavičce. Tato metoda má většinou za následek vytvoření nějakého nového objektu, případně úpravu stávajícího objektu.
PUT	Vloží reprezentaci nějakého zdroje na server.
DELETE	Opačná metoda k PUT, vymazání zdroje specifikovaného pomocí URL.
TRACE	Odešle zpět klientovi přijatý požadavek a tím umožní klientovi vidět, jak byl jeho původní požadavek změněn při průchodu přes jednotlivé servery až k cílovému serveru.
OPTIONS	Umožní ověřit podporované HTTP metody nad uvedeným zdrojem definovaným pomocí URL. Metodu je také možné použít na zjištění HTTP metod poskytovaných serverem a to použitím * v místě URL zdroje.
CONNECT	Vytvoří spojení mezi objektem přes uvedený port. Používá se pro ustanovení SSL, při průchodu přes proxy.

Tabulka 1.1.1: Metody protokolu HTTP

Kromě těchto metod definuje HTTP také dvě speciální skupiny metod a to metody bezpečné (*Safe methods*) a metody idempotentní (*Idempotent methods*)[3].

Bezpečné metody – (save methods)

Hlavním důvodem pro definici této skupiny bylo vymezení metod, které nemají vedlejší účinky, tj. mělo by jít o metody výhradně pro získávání dat a nikoliv o metody, které mohou nějakým způsobem změnit stav serveru a tím i chování serveru vůči jiným klientům v pozdějších dotazech. Po provedení těchto metod by nemělo dojít k ovlivnění chování serveru. Do této skupiny patří metody GET, HEAD, OPTIONS a TRACE.

Jako kontrast k bezpečným metodám je možné brát metody POST, PUT a DELETE. Tyto mohou způsobit svým vykonáním vedlejší účinky, jako například odeslání emailu, provedení finanční transakce a podobně. Z tohoto důvodu nejsou tyto metody používané webovými vyhledávači jako je např. Google.

Idempotentní metody – (idempotent methods)

Do této skupiny se řadí metody, u kterých platí, že několikanásobné provedení stejného požadavku má stejný výsledek jako provedení pouze jednoho požadavku. Do této skupiny patří metody PUT a DELETE a také všechny bezpečné metody.

Opět lze jako kontrast uvést metodu POST, která není idempotentní. Např. při vícenásobném odeslání formuláře dojde při prvním odeslání k uložení nového zdroje stejně jako v druhém případě, což způsobí duplicitní uložení zdroje a tím je porušena podmínka několikanásobného provedení metody při zachování stejného výsledku.

Závěrem lze říci, že toto rozdělení do skupin je více doporučení než stoprocentní garance, protože je klidně možné napsat aplikaci, která bude pomocí požadavku GET provádět například vložení řádku do databázové tabulky či jinou operaci, která porušuje definici bezpečné metody.

Základní typy hlaviček – požadavek

Hlavičky jsou velmi důležitou součástí HTTP požadavku, definují nejrůznější omezení a charakteristiky kladené na server, jako preferovaný jazyk, kódování atd. Od verze 1.1 se staly povinnou částí klientského požadavku a to z důvodu možnosti používání tzv. virtuálních serverů. Hlavička *Host* specifikuje doménové jméno serveru. To umožní jednomu fyzickému serveru obsluhovat více doménových serverů, protože je mezi nimi schopen rozlišit právě podle hodnoty hlavičky *Host*. Následující list shrnuje nejpoužívanější typy uživatelských hlaviček, jejichž kompletní výčet naleznete v [3]:

- Hlavičky začínající `Accept` definují, co klient očekává jako výsledek svého požadavku. Tyto hlavičky umožňují serveru přizpůsobit formát dat v odpovědi na základě požadavků klienta. Do této skupiny patří:
  - `Accept` – definice MIME, které je schopen klient přijmout (př. `Accept: text\plain` – přijímá obyčejný text) .
  - `Accept-chartset` – kódování.
  - `Accept-language` – jazyk, který je klient schopný přijmout.
- Hlavičky `Content` obsahují dodatečné informace o datech klientského požadavku.
  - `Content-length` - délka uživatelského požadavku.
  - `Content-type` – definice MIME typu.
- `Host` – doménové jméno serveru, použití pro virtuální severy (viz výše).
- `Refer` – udává stránku, ze které byla aktuálně požadovaná stránka odkázána.
- `If-Modified-Since` – používá se pro dotaz na změnu zdroje od specifikovaného data. Tato hlavička je úzce spojena s kódem odpovědi 303 Not modified.

1.1.2 Odpověď serveru (response)

Každá odpověď začíná specifikací verze protokolu HTTP 1.x. Dále následuje hodnota stavového kódu, který určuje, zda se odpověď na požadavek zdařila, případně zda došlo k nějaké chybě (viz tabulka 1.1.2). První řádek uzavírá stavové hlášení, které je krátkou zprávou odpovídající stavovému kódu (např. „OK“ v případě úspěchu). Na dalších řádcích následují hlavičky, které dělí od samotných dat odpovědi jeden prázdný řádek. Přesný formát odpovědi podle HTTP shrnuje následující příklad:

```
HTTP 1.x #stavový kód# #stavové_hlášení#
#hlavičky#
#prázdný řádek#
#data odpovědi#
```

Příklad 1.1.2: Formát odpovědi dle HTTP

Souhrn stavových kódů HTTP odpovědí

Hodnota prvního čísla stavového kódu identifikuje jednu z pěti tříd HTTP odpovědí. Význam jednotlivých tříd shrnuje následující seznam:

- 1XX – informační stavový kód. Jedná se o tzv. dočasné odpovědní zprávy, jako např. požadavek byl obdržen a akceptován a uživatel může zaslat uživatelská data požadavku. Není podporováno v HTTP 1.0.
- 2XX – úspěch. Značí, že požadavek klienta byl úspěšně přijat a zpracován.
- 3XX – přesměrování. Vyžaduje další interakci od uživatele pro splnění požadavku. Nemusí jít vždy o přímý zásah uživatele, může být zastoupen tzv. uživatelským-agentem v podobě např. internetového prohlížeče.
- 4XX – chyba na straně klienta. Pokud dojde k chybě při provádění kterékoliv z metod kromě HEAD, měl by server kromě chybového kódu poskytnout klientovi entitu podrobně popisující chybu a případnou informaci, zda jde o chybu dočasnou nebo permanentní.
- 5XX – chyba na straně serveru. Poskytnutí informací o chybě podobně jako u 4XX.

Následující tabulka shrnuje nejvíce používané typy odpovědí jednotlivých tříd[3]:

Kód	Název	Popis
200	OK	Standardní odpověď na úspěšné splnění požadavku. Obsah dat závisí na použité metodě. Pokud použijeme GET obsahem je entita popisující požadovaný zdroj. U metody POST je navrácena entita, popisující výsledek POST operace.
201	Created	Značí, že požadavek byl úspěšně vykonán a jeho výsledkem bylo vytvoření nového zdroje.
202	Accepted	Požadavek byl akceptován, ale ještě nebylo dokončeno jeho zpracování.
204	No Content	Požadavek byl úspěšně zpracován, ale výsledkem nejsou žádná data pro klienta.
301	Moved Permanently	Zdroj byl trvale přemístěn na jiné URL.
302	Found	Zdroj dočasně přesunut (v HTTP 1.0 dříve označováno jako *Moved Temporarily*).
304	Not modified	Server odpoví tímto kódem, pokud zdroj nebyl změněn od data uvedeného v hlavičce klientského požadavku `if-modified-since`.
400	Bad Request	Chyba v syntaxi požadavku.
401	Unauthorized	Správný požadavek, ale výsledek je dostupný pouze po autorizaci.
404	Not found	Požadovaný zdroj nebyl nalezen, ale je možné, že bude

		dostupný v budoucnu.
500	Internal server error	Generický kód pro interní chybu serveru. Typicky je použitý v případě, že žádné další informace nejsou k dispozici.
501	Not Implemented	Server nepodporuje metodu použitou v HTTP požadavku.
503	Service Unavailable	Služba je nedostupná. Jedná se většinou o chybu dočasnou, např. když je server přetížen velkým množstvím požadavků a nestíhá odpovídat na další.

Tabulka 1.1.2: Odpovědní kódy HTTP

Základní typy hlaviček – odpovědí

Podobně jako hlavičky požadavků, podávají hlavičky odpovědí dodatečné informace k odpovědní zprávě. Mohou to být informace jako např. kódování obsahu odpovědi, platnost požadovaného zdroje, informace zda je možné použít vyrovnávací paměti pro pozdější znovunačtení zdroje atd. Následující výpis zahrnuje alespoň příklady těch nejdůležitějších:

- Skupina `Content-` informace o datech obsažených v odpovědi serveru, obdoba `Content` hlaviček klientského požadavku.
- `Cache-Control` – udává, zda může být výsledný dokument uložen do vyrovnávací paměti.
- `Server` – informace o serveru (jméno popřípadě operační systém serveru).
- `Expires` – datum, kdy vyprší platnost dokumentu a bude jej nutné znovu načíst.

1.2 Webové služby

Webové služby (WS) jsou stavebním kamenem, který bude hojné používán v dalších kapitolách. Z pohledu laika se dají WS definovat jako webové API, které umožňuje komunikaci systémů bez ohlednu na to, v jakém jazyce jsou napsány, nebo na jaké platformě momentálně běží. Komunikace mezi nimi běžně probíhá pomocí posílání zpráv ve formátu XML a výše popsaného protokolu HTTP. Ovšem i WS, jako většina světově rozšířených a používaných API, má definován svůj standard. Ten je zastřešen konsorciem W3C[4]. Dříve, než si uvedeme samotnou definici WS podle W3C konsorcia, vysvětlím některé důležité pojmy a technologie použité v definici.

1.2.1 SOAP (Simple Object Access Protocol)

SOAP byl vytvořen jako odlehčený (*light-weight*) protokol pro výměnu strukturovaných informací v decentralizovaném a distribuovaném prostředí. Definuje strukturu jednotlivých zpráv, které si aplikace v distribuovaném prostředí vyměňují. Struktura zpráv je definována ve formátu XML. Jedná se o protokol aplikační vrstvy, který však k samotnému odesílání jednotlivých zpráv používá jiný protokol aplikační vrstvy, převážně pak HTTP.

Formát zprávy:

XML jazyk byl vybrán jako základ pro definici formátu SOAP zpráv a to hlavně z důvodu jednoduché transformace XML dat do formátu potřebného pro specifickou aplikaci komunikující pomoci SOAP protokolu. XML syntax sebou nese výhodu v čitelnosti zpráv uživatelem, v možnosti jednoduše validovat obsah zpráv a předejít tak chybám při samotném zpracování. Na druhou stranu XML formát způsobuje nemalou režii z hlediska objemu přenesených dat,

čímž degraduje rychlost komunikace v porovnání s jinými metodami distribuované komunikace, které zasílají data v binární formě.

SOAP zprávy se skládají ze tří částí:

- **Obálka (*envelope*)** – kořenový element SOAP zprávy. Obsahuje definice jmenných prostorů (*namespaces*), které umožňují rozlišit elementy popisující strukturu SOAP (`soap:headers`, `soap:body`) od elementů popisujících obsah samotné zprávy. Dále obaluje následující dva elementy - hlavičku a tělo.

- **Hlavička (*header*)** – je prvním prvkem uvedeným uvnitř obálky. Slouží k definici transakčních elementů. Tyto transakční elementy definují operace, které je nutné provést ještě před zpracováním samotného těla zprávy. Např. zahájení transakce, ověření práv atd. V okamžiku, kdy je u některého z transakčních elementů nataven atribut `mustUnderstand` na hodnotu `1`, musí server akceptující tuto zprávu provést okamžité zpracování tohoto transakčního elementu. Pokud server nerozumí sémantice dané tímto transakčním elementem, je zpráva odmítnuta. Zpětnou kompatibilitu pro starší klienty, kteří nepodporují operace definované pomocí transakčních elementů, je možné zajistit definováním atributu `mustUnderstand` s hodnotou `0`. Pokud je hodnota `mustUnderstand` nastavena na 1 a server tuto operaci nepodporuje, je vyvolána odpovědní zpráva, která definuje chybový prvek v těle zprávy. Ten poskytuje relevantní informaci klientovi o vzniklé chybě. V elementech hlavičky je také možné uvést atribut `actor`. Ten slouží k definování adresy uzlu, kterého se transakční element skutečně týká. SOAP zpráva může procházet při svém zpracování více uzly a pomocí atributu `actor` můžeme vymezit, pro které uzly jsou jednotlivé definice v hlavičce určeny.

- **Tělo (*body*)** – obsahuje informace, na jejichž základě je provedena některá z metod poskytovaných serverem. Definuje jméno operace a případné parametry, která jsou předány jako vstup pro operaci. V případě odpovědní zprávy obsahuje XML data reprezentující výsledek požadované operace.

Následující příklady ukazují jednu z možných komunikací mezi klientem a serverem pomocí SOAP protokolu. Jedná se o funkci pro převod datového typu String na Integer.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
  <soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   soap:encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<soap:Header>
  <m:Trans xmlns:m="http://www.soapexamples.com/transaction/"
```

```
  soap:mustUnderstand="1">234
  </m:Trans>
</soap:Header>

    <soap:Body>
        <ns1:stringAnInteger xmlns:ns1="urn:MySoapServices">
            <param1 xsi:type="xsd:String">123</param1>
        </ns1:stringAnInteger>
    </soap:Body>
  </soap:Envelope>
```

Příklad 1.2.1: SOAP zpráva reprezentující klientský požadavek

Na příkladu můžeme vidět, že zpráva je poměrně dobře čitelná a to nejen strojově. Jak je vidět na příkladu, element `envelope` nám definuje několik jmenných prostorů (např. výchozí soap nebo xsd pro definice elementů z XML schématu). Element reprezentující hlavičku je pro tento typ funkce zcela zbytečný, zde má pouze ilustrativní charakter. Můžeme vidět, že má definován element `trans`, který mohl například obsahovat identifikaci transakce, v rámci které bude operace provedena. V těle zprávy vidíme element s názvem operace (`stringAnInteger`), která převede parametr specifikovaný v elementu `param1` z typu `String` na typ `Integer`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

  <soap:Envelope …>

    <soap:Body>
        <ns1:stringAnInteger
         xmlns:ns1="urn:MySoapServices">
            <param1 xsi:type="xsd:int">123</param1>
        </ns1:stringAnInteger >
    </soap:Body>
  </soap:Envelope>
```

Příklad 1.2.2: SOAP zpráva reprezentující odpověď

V odpovědní zprávě již vidíme výsledek operace `stringAnInteger`. Můžeme si všimnout jiného návratového typu definovaného pomocí vestavěného typu XML Schématu `xsd:int`. Tři tečky v elementu `envelope` zastupují definici stejných jmenných prostorů, jako v případě požadavku.

Komunikace mezi klientem a serverem probíhá zasíláním obdobných SOAP zpráv pomocí některého z dalších protokolů aplikační vrstvy, nejčastěji HTTP. Existují samozřejmě složitější typy SOAP správ, některé z nich si uvedeme při srovnání konkrétních implementací webových služeb v kapitole 5. Pro samotné pochopení principu SOAP by měly být tyto příklady dostatečné, stejně jako pro pochopení problematiky této práce. Pro další informace o SOAP je vhodné použít tutoriály z W3Schools [7], nebo specifikaci SOAP protokolu [6].

1.2.2 WSDL (Web Service Description Language)

WSDL je dalším stavebním kamenem pro definici webových služeb. Jedná se o jazyk, který slouží pro popis rozhraní webových služeb (dále jen služeb). Podobně jako SOAP je postaven na

jazyku XML. Velmi často se používá právě ve spojení s XML schématem a SOAP protokolem pro práci se službami na internetu. WSDL konkrétní služby popisuje operace poskytované službou, data zprávy používané při volání jednotlivých operací a také porty a adresy, na kterých je možné se službou komunikovat.

V současné době existují dvě verze WSDL 1.1 a 2.0. Specifikace 2.0 je pouze přejmenovanou verzí 1.2, a to z důvodu výraznější odlišnosti od předchozí verze. Krom přejmenování některých objektů je asi nejvýraznějším rozdílem možnost mapování operací webových služeb na všechny typy HTTP metod (WSDL 1.1 umožňuje pouze mapování na GET a POST). Toto umožní lepší použití WSDL pro definici webových služeb REST, o kterých si povíme více v kapitole 2.5. Hlavní nevýhodou WSDL 2.0 je nedostatečná podpora vývojových nástrojů, které umožňují jednodušší práci s WSDL.

Popis rozhraní služby se dá rozdělit do dvou úrovní (abstraktní, konkrétní). Toto dělení umožní opakované použití některých části definice.

Abstraktní úroveň definuje rozhraní WS z pohledu odesílaných a přijímaných zpráv a také poskytovaných operací. Tyto operace následně seskupuje pomocí rozhraní. Hlavní myšlenkou je, že popis operací a zpráv je striktně oddělen od přesného formátu pro samotný fyzický přenos, nebo typu systému, na kterém WS běží.

Konkrétní úroveň definuje zbývající nutné systémové detaily pro samotný běh služby jako je např. přenosový formát zprávy, síťová adresa, kde je služba dostupná, a také konkrétní rozhraní z abstraktní sekce, které služba implementuje.

Objekty (komponenty) ve WSDL 1.1 (2.0)

XML schéma je k samotnému popisu formátu WSDL nedostatečné, proto WSDL 2.0 standard zavádí tzv. model komponent (*component model)*. Tento model definuje skupinu komponent, které společně popisují WS. Komponenty se dají chápat jako fragmenty XML dokumentu reprezentujícího WSDL, protože každá z komponent má definovanou svoji XML reprezentaci. Verze 1.1 explicitně nedefinuje žádný model, ale na místo toho definuje přímo XML elementy (objekty), které jsou do jisté míry srovnatelné s komponentami z modelu verze 2.0. Následující seznam shrnuje popis nejdůležitějších komponent definovaných ve verzích WSDL 1.1/2.0 (komponenta = objekt pro verzi 1.1). Příklady definic jednotlivých elementů jsou uváděny pro verzi 2.0, pokud není explicitně uvedeno jinak.

Definition/Description (*Popis*)

Komponenta nejvyšší úrovně (*high level*), která slouží pouze jako kontejner další komponenty.

Port/EndPoint (*výstupní bod*)

Definuje konkrétní bod připojení k WS. Typicky je reprezentován pomocí URL, na kterém je WS dostupná (atribut address). Kromě atributu obsahujícího adresu má každý výstup jedinečné jméno v rámci služby a je provázán s komponentou propojení pomocí atributu binding. Jedná

se o komponentu konkrétní úrovně, která je potomkem komponenty služba. Ukázka v rámci příkladu 1.2.3.

Service (*služba*)

Popisuje množinu výstupních bodů, na kterých je daná služba poskytována. Kromě skupiny výstupních bodů má každá služba definované jméno a vazby na komponentu rozhraní. Reprezentuje definice, kterou použije klientská aplikace pro přístup k webové službě.

```
<service name="SampleServiceName"
         interface="tns:InterfaceReference">
   <endpoint name=" EndpoinName"
             binding="tns:BindingName"
             address="http://www.example.com/serviceAddress/"/>
       ..
</service>
```

Příklad 1.2.3: Definice služby ve WSDL 2.0

Message/Not-defined (*zpráva/nedefinováno*)

Každá zpráva se skládá z jedné nebo více logických částí. Zpráva typicky odpovídá nějaké operaci a obsahuje informace, které jsou potřebné k provedení této operace. Každá část má přiřazené jedinečné jméno v rámci jedné zprávy. Typ části zprávy je potom definován pomocí tzv. *(message-typing atribute*), který není nic jiného než vazba na definici typu v XML schématu. Ve verzi 2.0 byl tento element vypuštěn a namísto něj se používá přímý odkaz na definici elementu z XML schématu.

```
<message name="messageUniqueName">
       <part name="nameOfPart"
             element="tns:schemaElementReference"/>
   ..
</message>
```

Příklad 1.2.4: Definice zprávy WSDL 1.1

Operation/Operation (*Operace*)

Operace může být přirovnána k funkčnímu volání metody v programovacím jazyce. Specifikuje jednu operaci, kterou je možné pomocí služby vykonat. Rozlišujeme operace rozhraní (*Interface operation)* a operace propojení (*Binding operation*) podle toho, zda je operace definována v rozhraní nebo v rámci propojení. Operace rozhraní má definované jedinečné jméno v rámci rozhraní a také vstupní a výstupní zprávy, pomocí kterých se operací komunikuje. Operace propojení definuje, jakým způsobem máji být vstupní a výstupní data operace rozhraní zakódována do přenosového formátu. Příklady viz 1.2.5 a 1.2.6.

PortType/Interface (*Rozhraní*)

Komponenta rozhraní popisuje sekvenci zpráv, pomocí kterých se dá se službou komunikovat. Tyto zprávy potom seskupuje v rámci jednotlivých operací v podobě vstupních a výstupních zpráv operace. Rozhraní může rozšiřovat jiné rozhraní. Takové rozšíření potom poskytuje všechny metody rozšiřovaného rozhraní a metody přímo definované v rozšíření.

```
<interface name="SampleInterface">
<fault name="ClientError" element="tns:response"/>
  <operation name="sayHelloTo"
           pattern="http://www.w3.org/ns/wsdl/in-out">
    <input messageLabel="inputMsgLabel"
           element="tns:requestSchemaRef"/>
    <output messageLabel="outMsgLabel"
            element="tns:responseSchemaRef"/>
     </operation>
</interface>
```

Příklad 1.2.5: Definice rozhraní WSDL 2.0

Binding/Binding (*Propojení*)

Komponenta propojení definuje konkrétní formát pro přenos zprávy včetně protokolu, který bude použit pro její přenos. Tento protokol je možné použít také pro definici výstupního bodu. Komponenta doplňuje implementační detaily potřebné pro fyzický přístup k definované službě.

```
<binding name="SoapBinding"
     interface="tns:SampleInterface"
     type="http://www.w3.org/ns/wsdl/soap"
  wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
  wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-
           response">
      <operation ref="tns:sayHelloTo" />
</binding>
```

Příklad 1.2.6: Definice propojení pro protokol SOAP WSDL 2.0

Types/Types (*typy*)

Typy se používají pro definici dat používaných v jednotlivých zprávách nebo operacích. K definici dat se používá syntaxe XML schématu. Elementy definované pomocí XML schématu jsou pak odkazovány v rámci ostatních komponent. Následující příklad ilustruje jednu z možných definicí typu. Tři tečky v elementu XML schématu reprezentují vynechanou definici jmenného prostoru `xs`, z důvodu lepší přehlednosti příkladu.

```
<types>
  <xs:schema …>
  <xs:element name="schemaElement">
     <xs:complexType name="sayHelloTo">
       <xs:sequence>
      <xs:element name="nameTo" type="xs:string" minOccurs="0"/>
    </xs:sequence>
      </xs:complexType>
  </xs:element>
 </xs:schema>
</types>
```

Příklad 1.2.7: Definice typu WSDL 2.0

Toto jsou základní komponenty definující rozhraní webové služby. Kompletní seznam všech komponent WSDL 2.0 můžete najít zde [8].

Každý WSDL dokument, který je validní vůči svému XML schématu a splňuje podmínky kladené na definici jednotlivých elementů (komponent), lze prohlásit za dokument WSDL odpovídající specifikaci WSDL 1.1/2.0.

Nyní máme definovány dva základní stavební kameny pro pochopení definice webových služeb dle W3C konsorcia.

1.2.3 Definice WS (W3C)

„Webová služba je softwarový systém podporující spolupráci mezi dvěma stroji prostřednictvím počítačové sítě. Rozhraní webové služby je popsáno pomocí strojově zpracovatelného formátu (přesněji formát WSDL). Ostatní systémy komunikují s webovou službou v souladu s předepsaným WSDL rozhraním s použitím SOAP zpráv typicky pomocí protokolu HTTP s XML serializací, která je v souladu s dalšími webovými standardy.“ (volný překlad z [4])

Kromě této definice konsorcium W3C rozlišuje dva hlavní typy (třídy) webových služeb. Prvním typem jsou služby splňující podmínku stylu REST tzv. *REST-compilant*. Tyto služby využívají pro zpracování dat sadu jednotných operací, které jsou pro všechny služby stejné. Styl REST a webové služby jsou hlavním tématem práce a bude jim věnována následující kapitola. Druhou hlavní třídou jsou potom libovolné služby tzv. *arbitrary web services*. Tyto služby mají na rozdíl od REST služeb možnost definice libovolné množiny operací. Tato množina je popsána pomocí WSDL. [4]

1.2.4 Příklad využití webových služeb a jejich význam vůči SOA

Nyní již víme, co to jsou webové služby, ale pro dokonalé seznámení s tímto tématem bych ještě velmi krátce shrnul, k čemu se webové služby používají a také zdůraznil místa, ve který se používají výše rozebírané technologie.

Webové služby byly vyvinuty jako implementace myšlenky SOA (*Service Oriented Architecture*). Základem SOA je rozdělení funkcí do malých nezávislých jednotek, které označuje pojmem služba (*service*). Tyto služby jsou volně svázané (*loosely-coupled*) s operačním systémem nebo programovacím jazykem, ve kterém jsou napsány. Jsou dostupné prostřednictvím sítě a komunikují pomocí dobře definovaných otevřených formátů (např. XML), což umožňuje vývojářům jejich spojování nebo znovupoužití ke splnění různých požadavků. Služby působí jako nezávislé, ale často je nutné definovat složitější operace, které se skládají z volání několika služeb po sobě v určitém pořadí a za určitých podmínek. Tomuto propojování volání jednotlivých služeb se v SOA říká orchestrace služeb.[20]

Když přirovnáme myšlenky SOA k právě definovanému standardu webových služeb, můžeme si všimnout nemalých podobností. Webové služby taktéž tvoří nezávislé celky, které jsou dostupné prostřednictvím sítě a které nejsou závislé na operačním systému a programovacím jazyce.

Hlavním doménou využití webových služeb jsou tedy systémy, kde je potřeba jednoduchým způsobem sdílet a nabízet funkce ostatním systémům, nezávisle na použitém jazyku pro implementaci. Konkrétním případem mohou být například různé pobočky nadnárodní firmy. Každá z nich může mít vyvinutý svůj vlastní systém upravený pro potřeby konkrétní pobočky. Pomocí webových služeb pak mohou jednotlivé pobočky poskytovat důležité údaje pro vedení, které potřebuje určitá data ze všech poboček.

2. REST jako architektonický styl

V této kapitole představím základní myšlenky a koncepty spojené s architekturou REST, bude vysvětlen význam zkratky REST a omezení, které je nutné splnit, aby systém mohl být označen jako RESTful systém (tj. systém splňující podmínky architektonického stylu REST). Dále bude uvedena ukázka aplikace myšlenky REST na praktických příkladech (HTTP, webové služby). V případě webových služeb bude REST porovnán s webovými službami, které pracují na protokolu SOAP. V závěru kapitoly budou sumarizovány klíčové poznatky ohledně tohoto stylu, budou zdůrazněny hlavní výhody, které REST přináší.

2.1 Pojem REST

Poprvé byl termín REST představen v disertační práci Roye Fieldinga v roce 2000 [1]. REST (Representational State Transfer) byl vyvinut jako architektonický styl distribuovaných hypermédií (jako např. World Wide Web), vznikal souběžně s protokolem HTTP 1.1 a dá se říci, že největší známou implementací systému, odpovídajícímu omezením stylu REST, je WWW. Některé články dokonce uvádějí, že REST definuje ve své podstatě vlastnosti, které činí WWW tak úspěšným (např. [2]).

Nyní se podíváme na to, co pojem REST znamená. Pro lepší představu budu pojem REST a jeho základní myšlenku demonstrovat na příkladu webu a na závěr uvedu definici REST. Web se skládá z jednotlivých zdrojů (dokumentů, webových stránek HTML a podobně). Například automobilka Škoda může definovat nějaký zdroj Škoda Octavia. Na ten mohou jednotliví uživatelé přistoupit pomocí URL: *http://www.skoda-auto/auto/octavia*. Pokud tak učiní, reprezentace (**R**epresentational) zdroje Octavia je navrácena například v podobě souboru octavia.html. Zobrazení tohoto zdroje v prohlížeči dostává klientskou aplikaci (klienta) do nějakého stavu (**S**tate). Pokud klient následně zvolí některý z odkazů na webové stránce octavia.html je přesměrován na jinou stránku (zdroj). Tím se mění reprezentace zdroje a také stav samotného klienta. Poslední slovo ze zkratky pak vzniklo díky tomu, že stav aplikace se mění (**T**ransfer) při každé změně stránky. Když spojíme následující poznatky dohromady, dostáváme Representational State Transfer.

2.1.1 Definice Representational State Transfer podle Fieldinga:

„REST by měl vyvolávat představu o tom, jak by se měla chovat dobře navržená webová aplikace: síť webových stránek (virtuální stavový automat), kde uživatel postupuje v aplikaci pomocí výběru odkazů (stavových přechodů), což má za následek zobrazení další stránky (reprezentující další stav aplikace), která je přenesena k uživateli a zobrazena pro jeho použití.“ (volný překlad z [1]).

REST je tedy architektonický styl, nejedná se o žádný standard, který naleznete na stránkách W3C, SUN apod. Ačkoli REST vznikl paralelně s protokolem HTTP/1.1 a vysvětluji ho na příkladu webu, samotná myšlenka REST není svázaná s žádným protokolem.

V této sekci jsme si vysvětlili pojem REST, avšak k definování architektonického stylu je nutné definovat seznam omezení, které umožní formálně říci, zda se jedná o styl splňující požadavky REST či nikoliv (zda lze označit systém za RESTful).

2.2 Omezení nutná pro REST

REST je také někdy označován jako hybridní styl, protože vznikl postupnou derivací jednotlivých omezení (pravidel) z již existujících architektur. Roy Fielding zkonstruoval tento styl z takzvaného výchozího „NULL STYLE“ [1] přidáváním jednotlivých omezení, které styl definují. Následujících sekce vysvětlují šest základních omezení na systémy splňující myšlenku REST. Ty jsou pak označovány jako RESTful. Jedná se o tato omezení (jejich popis vychází z [1]):

- klient-server (*client-server*);
- bezstavovost (*stateless*);
- možnost využití vyrovnávací paměti (*cacheable*);
- kód na vyžádání (*code on demand*);
- vrstvený systém (*layered system*);
- jednotné rozhraní (*uniform interface*).

2.2.1 Klient-server (client-server)

Omezení říká, že systém má komponentu server, která nabízí definovanou množinu služeb a naslouchá požadavkům na tyto služby. Klientská komponenta potom spouští jednotlivé služby pomocí požadavku odeslaného na server. Ten buď požadavek odmítne, nebo přijme a odešle zpět odpověď ke klientovi. Existuje několik variant klient-server systémů, prozatím budeme uvažovat klient-server v nejjednodušší podobě, k jeho upřesnění dojde v rámci dalších sekcí.

Hlavní motivací pro zavedení tohoto omezení bylo důsledné oddělení klientů od serverů pomocí jednotného rozhraní (viz sekce se stejným názvem níže). Díky tomu není klientská aplikace zatížena správou dat aplikace (role serveru), což umožňuje zvýšení přenositelnosti (*portability*) klienta. Na druhou stranu servery neřeší samotnou prezentaci dat a uživatelské rozhraní, což umožňuje celkové zjednodušení služeb serveru a tím i větší škálovatelnost (*scalability*).

2.2.2 Bezstavovost (stateless)

Nyní již víme, že systém podle REST architektury musí být typu klient-server. Dalším omezením je, že veškerá komunikace mezi klientem a serverem musí být bezstavová. To znamená, že klientská aplikace musí serveru odeslat veškeré informace o jejím aktuálním stavu v rámci svého požadavku. Veškeré informace o stavu aplikace jsou uloženy na klientovi. Spojením prvních dvou omezení získáváme tzv. *client-stateless-server* [1].

Toto omezení sebou přináší tři hlavní výhody: škálovatelnost, viditelnost a spolehlivost. Bezstavovost opět zjednoduší implementaci jednotlivých služeb a zároveň umožní rychleji obsluhovat jednotlivé požadavky a uvolňovat zdroje. Tato fakta umožňují další vylepšení škálovatelnosti celého systému. Stejně tak je velice jednoduché takovýto bezstavový systém umístit do distribuovaného prostředí a rozprostřít zátěž na více PC. V systému, který si nemusí ukládat stav jednotlivých aplikací, je také mnohem jednodušší zotavení z případné chyby a tím pádem se zlepšuje spolehlivost. Fakt, že není nutné prohledávat nic jiného, než samotný požadavek klienta pro zjištění kompletního stavu aplikace, umožňuje lepší viditelnost požadavků pro monitorovací systém a tím je dosaženo poslední výhody tzv. viditelnosti.

Bohužel toto omezení sebou nese i nevýhody. Tou hlavní a nejpodstatnější je celkové snížení výkonnosti a propustnosti systému díky opětovnému zasílání stejných dat, která by v případě vypuštění tohoto omezení mohla být uložena na serveru.

2.2.3 Možnost využití vyrovnávací paměti (cacheable)

Hlavní motivací pro toto omezení bylo celkové zvýšení propustnosti systému a efektivity využití sítě. Toto omezení požaduje, aby systém umožňoval explicitní označování dat (vyrovnávací paměť povolena/zakázána *cacheable/non-cacheable*) odesílaných jako odpověď na některý z požadavků serveru. Pokud jsou data označena příznakem cacheable, potom je možné data uložit do vyrovnávací paměti z důvodu jejich pozdějšího použití při vyvolání stejného požadavku.

Hlavní výhoda je zcela zřejmá, při zavedení vyrovnávací paměti může v některých případech dojít k eliminování interakcí se serverem a tím dochází k výraznému zvýšení efektivity, škálovatelnosti a reálného výkonu aplikace díky zmenšení průměrného zpoždění, které je způsobeno voláním služeb serveru. Na druhou stranu může mít vyrovnávací paměť za následek snížení spolehlivosti, pokud se budou data v ní obsažená lišit od dat, které by aplikace získala při vyvolání požadavku v momentě, kdy čerpá data z vyrovnávací paměti.

2.2.4 Kód na vyžádání (code on demand)

Toto omezení říká, že je možné při některých požadavcích odesílat nejen data, ale i celé části programu (kódu), který se vykoná na klientovi. Může jít např. o Javascript, Java aplety atd. To má za následek další vylepšení odezvy a zjednodušení rozhraní serveru, protože část operací, které by za normálních okolností vykonával server, může být přenesena na klienta.

Z druhé strany tím, že se neposílají pouze data, dochází k výraznému snížení viditelnosti. Toto omezení bylo hlavním důvodem proč je požadavek code on demand jako jediný z této šestice označen jako volitelný. Ve své podstatě REST říká, že by měl být použit pouze v místech, kde je to výhodné nebo opravdu zapotřebí.

2.2.5 Vrstvený systém (layered system)

Hlavní myšlenkou vrstvených systémů je lepší zvládnutí velké složitosti systému jeho rozdělením do hierarchických vrstev. Každá vrstva může používat služby pouze nižší vrstvy a naopak je přímo využívána vyšší vrstvou. Jakékoliv jiné komunikace mezi vrstvami jsou zakázány. Tím je snížena složitost implementace některé z vrstev na nutnost znalosti pouze nižší vrstvy.

Toto omezení přidá klient-serveru možnost vkládat další uzly (prostředníky) mezi klienta a server. Takže klient nikdy neví, zda komunikuje přímo se serverem, či nějakým prostředníkem, který následně jeho požadavky přeposílá. Umístění prostředníků můžeme využít například pro vylepšení propustnosti (*load-balancing*), prosazování bezpečnostních politik nebo zjednodušení komponent přesunutím málo používaných funkcí na jednoho z prostředníků.

Omezení sebou nese i jednu velkou nevýhodu, která je obecná pro všechny systémy pracující na principu hierarchických vrstev a to snížení propustnosti díky latenci a vysoké režii komunikace mezi vrstvami. Tato nevýhoda je z části řešitelná pomocí sdílené vyrovnávací paměti na některé z vrstev, což může mít v některých případech naopak velmi pozitivní dopad na výkonnost.

2.2.6 Jednotné rozhraní (uniform interface)

Vůbec nejdůležitějším požadavkem RESTu je jednotné rozhraní. Mnoho knih používá pro vysvětlení tohoto pojmu příklad aplikace REST na HTTP protokol a uvádí, že jednotné rozhraní tvoří soupis definovaných sloves určujících operace, které je možné se zdrojem vykonávat (GET, POST, PUT, DELETE, apod.). Z pohledu Fieldinga[1] však není tento styl vázán na konkrétní protokol a pro definici jednotného rozhraní používá tato omezení:

- **Indetifikace zdroje** – jednotlivé zdroje jsou identifikovány v požadavcích klienta například použitím tzv. URI (*Uniform Resouce Identifier*). URI jednoznačně určuje, o který zdroj se jedná. Je rozdíl mezi zdrojem samotným a jeho reprezentací, která je navrácena klientovi. Server neodesílá klientovi jako odpověď „výpis z databáze“ tj. výpis konkrétního zdroje, ale namísto toho odesílá data, která jsou zabalená například v HTML, XML nebo ve formátu JSON [1](*Javascript Object Notation*) a jsou v jazyce a v kódování, které požaduje klientská stanice.

- **Manipulace se zdrojem a jeho reprezentacemi** – toto omezení znamená, že ve chvíli, kdy klient přistoupil na nějaký zdroj a vlastní jeho reprezentaci včetně připojených metadat, má dostatek informací proto, aby byl schopen zdroj modifikovat, popřípadě úplně smazat ze serveru, pokud má k této operaci potřebná práva.

[1] *JSON (Javascript Object Notation) – univerzální textový formát pro výměnu dat mezi systémy. Hojně používán v aplikacích, které využívají asynchronní volaní a Javascript (ten je schopen používat přímo struktur formátu JSON bez nutnosti jakékoliv konverze). JSON však není závislý na jazyku Javascript a je možné ho použít v širokém spektru programovacích jazyků.*

- **Samopopisující zprávy** – každá zpráva by měla obsahovat veškeré informace, které jsou nutné pro její úspěšné zpracování. Toto omezení implikuje definici standardizovaných metod nebo funkcí a také definici standardizovaných typů medií (typů obsahu zprávy). V případě aplikace na HTTP, můžeme jako standardizované metody brát přímo HTTP metody (GET, POST, …) a jako standardizované typy medií potom media dle tzv. MIME type (např. text\plain, text\xml apod.).

- **Hypermédia jako zdroj stavu aplikace** – vychází z myšlenky, že klient bude pravděpodobně chtít přistupovat také ke zdrojům, které mají nějakou souvislost s právě žádaným zdrojem. Např. položka objednávky a objednávka. Tyto související zdroje by měly být identifikovány v odpovědní zprávě, například právě pomocí odkazů ve formě URI souvisejících zdrojů.

Aplikováním podmínky jednotného rozhraní dochází k celkovému zjednodušení architektury a také k vylepšení viditelnosti jednotlivých akcí. Díky jasně definovanému a jednotnému rozhraní pro komunikaci jsou implementace jednotlivých aplikací odstíněny od řešení problému vzájemné komunikace. Jak již bývá zvykem i toto omezení sebou přináší jistý kompromis. Pro přenos a komunikaci se používá pouze jedno standardní rozhraní, přes které se přenáší data ve standardizované formě (namísto formy specifické pro konkrétní aplikaci), což může vést ke snížení efektivity komunikace jednotlivých komponent.

2.3 Klíčové cíle REST

V předchozích sekcích jsou definovány hlavní omezení pro systém, který je možné označit jako RESTful. Každý z nich vnáší do architektury jisté výhody a nevýhody. Následující seznam shrnuje klíčové výhody, které přináší dodržení stylu REST:

- škálovatelnost komunikace mezi komponentami
- jednotné rozhraní pro komunikaci mezi komponentami
- možnost použití prostředníků v komunikaci (proxy)
- robustnost a efektivita – možnost použití vyrovnávací paměti
- možnost nezávislého vývoje jednotlivých komponent (volně vázané systémy)
- jednodušší implementace komponent – není nutné řešit problém komunikace, všichni mají stejnou sadu metod a stejné rozhraní.

2.4 REST a HTTP

Když porovnáme omezení kladené na REST jako architektonický styl a specifikaci HTTP protokolu, můžeme si všimnout jistých podobností. Např. protokol HTTP byl navržen pro komunikaci mezi klientem a serverem a je bezstavový. Stejně tak není problém pomocí hlaviček a případně dalších prostředků využít vyrovnávací paměť. Podobně je tomu i u dalších omezení. V mnoha výkladech je HTTP prostředkem pro vysvětlení některých pojmů REST, například vysvětlení podmínky jednotného rozhraní pomocí HTTP metod. Celkově by se dal protokol HTTP označit jako jedna z povedených a úspěšných aplikací stylu REST v praxi. Je to pravděpodobně způso-

beno tím, že REST samotný vznikal souběžně se specifikací HTTP 1.1 a také tím, že Roy Fielding byl silně angažován při vývoji této specifikace.

2.5 REST a webové služby (RESTful web services)

Z úvodních kapitol již víme co je to webová služba a také víme, že RESTful webové služby jsou speciální podmnožinou webových služeb, které mají sadu předem definovaných operací. Tyto předem definované operace jsou přímo operace převzaté z protokolu HTTP, jedná se tedy o operace GET, POST, HEAD, atd. (více viz tabulka 1.1.1). Každá z operací má vymezené svoje postavení vůči zdroji, který je identifikován pomocí URL. Následující tabulka shrnuje použití jednotlivých HTTP funkcí z pohledu REST služeb.

Metody	Použití v REST	Odesílaná data
GET	Získání konkrétní reprezentace zdroje.	Veškerá data jsou umístěna v URI konkrétního zdroje.
POST	Vytvoření nového objektu v systému.	Data jsou odesílána nejčastěji ve formě XML v těle požadavku.
PUT	Úprava existujícího objektu.	Stejně jako v případě POST.
DELETE	Vymazání objektu ze systému.	Objekt pro vymazání je jednoznačně určen svým URI.
HEAD	Získání dat, které popisují formát zdroje.	Stejně jako v případě GET.

Tabulka 2.5.1: Význam HTTP metod v rámci webových služeb REST

Využití metod ve správném významu je zcela klíčové pro čisté REST služby. Existuje mnoho reálných implementací webových služeb, kde je právě tento princip porušen. Takovéto služby označuje [2] jako takzvané REST-RPC hybrid.

2.5.1 REST služby vs. REST-RPC hybrid

Jak tedy vypadá opravdu čistá REST služba? Pro REST a jeho správnou implementaci jsou velmi důležité informace, nad kterým zdrojem se pracuje a jaká operace se s tímto zdrojem provádí. Informace o tom, nad kterým zdrojem se pracuje, je v případě REST služeb uvedena přímo v URI (služby založené na SOAP mají tuto informaci v těle HTTP zprávy). Operace, která se má provést je určena HTTP metodami.

Existují však případy, kdy se služby tváří jako REST, avšak popírají jednu z výše uvedených podmínek. Mějme například službu, která identifikuje jednotlivé zdroje pomocí URI, avšak pro veškerou komunikaci používá pouze metody GET a to, jaká operace se má provést určuje pomocí dotazového parametru za otazníkem. URI takové služby by mohlo vypadat takto:

```
http://www.example.com/octavia_new?operation=delete
```

Příklad 2.5.1: Porušení REST operace nad zdrojem v URI

Zmiňovaná služba porušuje podmínky architektonického stylu REST, zvláště pak podmínku jednotného rozhraní, kdy používá operaci GET pro naprosto jiné účely než je získávání zdrojů. Tento typ služeb je označován jako REST-RPC hybrid[2].

2.5.2 Webové služby REST vs. SOAP

Nyní již máme dostatečné znalosti přístupů SOAP s REST k tomu, abychom mohli provést jejich srovnání na teoretické bázi. Hlavním a zcela evidentním rozdílem obou přístupů je flexibilita definice rozhraní. Definice metod RESTful webových služeb je omezena HTTP metodami, naproti tomu SOAP umožňuje definice vlastních metod. Na druhou stranu SOAP zdaleka neumožňuje tak dobré využití vlastností HTTP protokolu. Veškerá komunikace je zúžena pouze na použití POST metody z HTTP pro všechny operace, ať už jde o operaci získání nebo vytvoření objektu. Stejně tak využití proxy je komplikovanější, protože jednotlivé proxy členy musí rozumět formátu SOAP zprávy. V případě REST služeb jsou proxy výrazně jednodušší, protože používají čistě HTTP protokol.

Další velkou nevýhodou SOAP je přidávání obálky na data ve formě SOAP zprávy. Tato obálka s sebou nese hned několik omezení. Tím hlavním může být například přenos jiných formátů než XML. Pokud nějaká aplikace požaduje data v jiném formátu než XML např. ve formátu JSON, REST služba dokáže pomocí hlavičky `Accept` toto identifikovat a přímo poskytnout aplikací požadovaný formát. Naproti tomu služby založené na SOAP musí vrátit zprávu ve formě XML a teprve uvnitř této zprávy samotný JSON.

Dalším omezením, spojeným se zprávou SOAP, je množství přenesených dat pro získání nějakého zdroje. Zatímco REST služby pro získání nějakého zdroje identifikovaného pomocí ID použijí pouze URL obsahující toto ID, SOAP služba musí zaslat XML zprávu dle formátu SOAP v těle HTTP požadavku. Tato zpráva pak obsahuje ID zdroje a metodu, pomocí níž lze zdroj získat.

Na druhou stranu služby využívající SOAP protokol jsou velmi dobře zaběhnuté a používané v řadě aplikací. Jejich podpora v jednotlivých programovacích jazycích a vývojových prostředích je velmi propracovaná. Vytvoření takové služby může uživatel vykonat pomocí průvodců a většina kódu jako WSDL je vygenerována automaticky. Podobné je to i u klienta webové služby, který jde vygenerovat pomocí pokročilých nástrojů přímo z WSDL. Tudíž programátor pouze napíše službu dle zvyklostí konkrétní implementace a obrovské množství složitého kódu je vytvořeno automaticky.

Další výhodou je samotný přístup k vývoji služeb. Ten je více intuitivní pro programátory zvyklé na objektově orientované jazyky. Služba vystavuje vývojáři definované metody a oni nepřemýšlí nad tím, jak správně navázat jednotlivé metody v objektovém světě na jednotlivé URI a metody protokolu HTTP.

Existuje jistě celá řada dalších výhod/nevýhod popřípadě odlišností, avšak výše zmíněné považuji za zásadní a budu se k nim ještě vracet v následujících kapitolách, při srovnání jednotlivých API v Javě a v praktickém příkladu v závěru práce. Tato srovnání by měla ještě upřesnit celkovou představu o rozdílu uvedených přístupů.

3. JAX-RS API (JSR-311)

Specifikace JSR-311 byla vyvinuta z důvodu absence jakéhokoli standardu pro vývoj REST služeb v Javě. Před jejím vydáním byly k dispozici pouze rámce pro vývoj REST služeb (např. Restlet), žádná z nich však nebyla definována jako standard JSR.

Hlavní ideou JAX-RS je poskytnutí programového rozhraní, které ulehčí vývoj webových služeb REST. Tuto ideu se snaží naplnit pomocí množiny anotací, které umožňují vývojářům definici jednotlivých zdrojů a operací (ve smyslu REST), které mohou být s jednotlivými zdroji prováděny. Anotace jsou aplikovatelné přímo na POJO (*Plain Old Java Object*) a právě jejich užitím se stávají služby z těchto obyčejných POJO objektů.

3.1 Hlavní cíle specifikace JSR-311

- **Založeno na POJO objektech (*POJO-based*)** – viz hlavní idea.
- **Zaměření na HTTP (*HTTP-centric)*** – úzce svázané s protokolem HTTP, přímá podpora metod a dalších elementů odpovídajících http.
- **Nezávislé na formátu (*Format independence)*** – Nezávislost na formátu použitém v těle http zprávy. Možnost jednoduchého rozšiřování aplikací o nové formáty.
- **Nezávislost na kontejneru (*Container independece*)** – vyhovující implementace by měla být schopna pracovat na velkém množství webových kontejnerů. Specifikace definuje, jak může být nasazena v prostředí servlet kontejneru, nebo využita pomocí poskytovatele JAX-WS.
- **Zahrnutí do kontextu Java EE (*Inclusion in Java EE*)** – Specifikace definuje, jak použít funkce Java EE v rámci zdrojů definovaných pomocí anotací JAX-RS.

3.2 Náhled na popisovanou problematiku primitivním příkladem.

Před samotným vysvětlením jednotlivých částí, které tvoří standard JAX-RS, bych rád ukázal příklad HelloWorld služby. Tento příklad poslouží jako motivace k dalším sekcím a hlavně jako ukázka hlavní myšlenky o reprezentaci služby pomocí anotovaného POJO objektu.

```
package cz.muni.fi.jaxrsexamples.services;
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

@Path("HelloWorld")
public class HelloWorldResource {
    @GET
    @Produces("text/plain")
    public String sayHello() {
        return "Hello World";
    }}
```

Příklad 3.2.1: HelloWord v JAX-RS

Jak můžeme vidět, kód je velmi jednoduchý a čitelný a skutečně jde o pouhý POJO objekt s anotacemi. Jedná se o zdrojovou třídu `HelloWorldResource`, která reprezentuje službu dostupnou na URI `HelloWorld`. Tento fakt definuje anotace `@Path`. Dále si můžeme všimnout anotace `@GET`. Ta určuje, že metoda `sayHello()` bude vyvolána jako reakce na HTTP metodu GET. Poslední z použitých anotací `@Produces` definuje MIME type. Ten určuje formát výsledku navráceného klientovi. Standard poskytuje celou řadu dalších funkcí, které budou probrány v následujících sekcích.

3.3 Zdrojové třídy

Zdrojovou třídu si můžeme přestavit jako třídu v jazyce Java, která implementuje konkrétní webovou službu s užitím anotací JAX-RS. Jedná se o třídy, které mají alespoň jednu metodu označenou anotací @Path (viz sekce 3.4) nebo tzv. „indikátorem metody požadavku“ (*Request method designator*). Ten je podrobněji vysvětlen v sekci 3.5 (např. dříve zmiňovaný @GET). Zdrojové třídy jsou základním stavebním kamenem pro vytváření webových služeb REST pomocí JAX-RS.

3.3.1 Životní cyklus

Výchozí chování dle JSR-311 je takové, že instance zdrojové třídy je vytvářena znovu pro každý požadavek. Zdrojová třída prochází při obsloužení požadavku těmito kroky:

- Konstrukce
- Vložení závislostí
- Vyvolání odpovídající metody a obsloužení požadavku - viz kapitola 3.5
- Objekt je dostupný pro garbage collection

Konstrukce – volání konstruktoru

Každá kořenová zdrojová třída musí mít veřejný konstruktor, který může být prázdný nebo parametrizovaný. V konstruktoru jsou použitelné parametry označené následujícími anotacemi JAX-RS: `@Context`, `@HeaderParam`, `@CookieParam`, `@MatrixParam`, `@QueryParam` nebo `@PathParam`. Pokud má zdrojová třída více konstruktorů, pak je vždy použit vyhovující konstruktor s nejvíce parametry.

Uvedené podmínky neplatí pro zdrojové třídy, které nejsou kořenové tj. takové třídy, které jsou vytvářeny přímo aplikací z kódu služby samotné, nikoliv však běhovým prostředím JAX-RS.

Vložení závislostí

Po vytvoření instance zdrojové třídy jsou nastaveny její vlastnosti a to opět pomocí anotací, které shrnuje následující tabulka.

Anotace	Význam
`@MatrixParam`	Vybere hodnotu Matrix parametru z URI.
`@QueryParam`	Vybere hodnotu Query parametru z URI.

`@PathParam`	Vybere hodnotu Path parametru z URI.
`@CookieParam`	Vybere hodnotu cookies.
`@HeaderParam`	Vybere hodnotu hlavičky.
`@Context`	Vloží instanci jednoho z podporovaných objektů. Tím může být např. UriInfo (objekt poskytující informace o URI, kterým byla služba vyvolána). Více informací o dalších podporovaných typech viz [11] kapitola 5 a 6.

Tabulka 3.3.1: Anotace pro vkládání parametrů z URI

3.4 URI šablony a anotace @Path

Jak jsme již viděli na prvním demonstračním příkladu 3.2.1, anotace `@Path` definuje relativní URI, na kterém bude dostupná služba reprezentovaná takto anotovanou Java třídou. Specifikace jde však v tomto směru ještě dále a umožňuje v rámci `@Path` definovat takzvané URI šablony. Ty nám potom umožňují ještě efektivnější práci s URI dané služby.

3.4.1 URI šablona ve smyslu JAX-RS

URI šablony rozšiřují definici URI v rámci anotace `@Path` o možnost definice vlastních proměnných. Tyto proměnné jsou nahrazeny za běhu aplikace aktuálními hodnotami. Jednotlivé proměnné definované v URI šabloně jsou uzavřené ve složených závorkách. Následující příklad reprezentuje URI šablonu pro službu `PersonService` s proměnnou `personId`.

```
@Path("/PersonService/{personId}")
```

Příklad 3.4.1: URI šablona

Při zavolání služby pomocí `/PersonService/1` dojde k přiřazení hodnoty 1 do proměnné `personId`. Pro další práci s tímto parametrem, například v metodě zdrojové třídy, lze použít anotaci `@PathParam`, která umožní načíst proměnnou `personId` jako vstupní parametr metody. Tomuto se věnuje podrobněji následující kapitola.

Další velkou výhodou URI šablon je možnost omezení hodnot jednotlivých atributů pomocí regulárních výrazů:

```
@Path("/PersonService/{personId: [0-9]*}")
```

Příklad 3.4.2: URI šablona omezení parametru pomocí regulárního výrazu

Výše uvedený příklad pak bude akceptovat pouze URI, které budou mít za `PersonService` pouze číslo. Pokud tomu tak nebude, tj. uživatel zavolá službu pomocí `/PersonService/spatneID`, tak bude oznámena chyba pomocí návratového kódu 404 (žádaný zdroj je nedostupný).

3.5 Identifikátor metody požadavku a metody zdrojové třídy

Identifikátor metody požadavku je běhová anotace, která svazuje metody zdrojové třídy s jednotlivými metodami HTTP protokolu. Tyto anotace také vycházejí z jedné z myšlenek JAX-RS (zaměření na HTTP protokol). Standard definuje anotace `@GET`, `@POST`, `@HEAD`, `@PUT`, `@DELETE`, které reflektují jim odpovídající HTTP metody. Např. označíme-li libovolnou metodu zdrojové třídy pomocí `@GET`, potom v případě odeslání HTTP požadavku na URI, které odpovídá zdrojové třídě s takto označenou metodou (obsah `@Path`), bude požadavek obsloužen právě pomocí metody označené anotací `@GET`.

3.5.1 Definice vstupních parametrů

V dosavadních příkladech jsme viděli pouze metody, které neobsahovaly žádné parametry. JAX-RS však nabízí velmi jednoduchý přístup k parametrům, ať už se jedná o parametry z URI služby, nebo o parametry z hlavičky, či dotazové parametry umístěné za otazníkem. Následující příklad demonstruje metodu, která má na vstupu parametr specifikovaný v URI šabloně (šablona obsahuje `{personId}`).

```
@GET
@Produces("text/plain")
public String getText(@PathParam(value="personId")
                      Long personId) {
    return "Hello World " + personId;
}
```

Příklad 3.5.1: Zpracování parametru personId v URI

Pokud by třída obsahující tuto metodu měla URI šablonu `/personService/{personId}`, pak by při přístupu na službu pomocí adresy `/personService/1` byl nastaven parametr `personId` na číslo 1. Podobným způsobem je možné pracovat s dalšími parametry obsaženými v HTTP požadavku. Lze použít dříve definované anotace v tabulce 3.3.1. nebo anotaci `@FormParam`. Ta umožňuje vložit do proměnné parametr z HTML formuláře a tak výrazně zjednodušit zpracování těchto parametrů (pro jejich získání není nutné pracovat s komplikovanými objekty, které reprezentují HTTP požadavek).

3.5.2 Mapování parametrů na typy v Javě

Jak je vidět na příkladu 3.5.1, jednotlivé parametry, které lze pomocí anotací vložit do vstupních parametrů metod, lze mapovat na primitivní typy v Javě. Takto lze velmi jednoduše získat číselnou reprezentaci parametru, i když je samotné URI reprezentováno jako `String`. Mapování parametru na Java typy/třídy je možné, pokud typ/třída splňuje jednu z následujících podmínek [11]:

- Jedná se o primitivní typ
- Existuje konstruktor s jedním parametrem typu `String`.
- Třída má statickou metodu `valueOf`, která je aplikovatelná na parametr typu `String`. (např. `Double.valueOf(String)`)

- Jedná se o `List<T>`, `Set<T>` nebo `SortedSet<T>`, kde třída `T` splňuje jednu z dvou předchozích podmínek. Výsledná kolekce je potom pouze pro čtení.

3.6 Omezení typu vstupu a výstupu (HTTP content negotiation)

Další silnou zbraní, kterou podporují jak služby REST, tak samotný HTTP protokol, je omezení formátu vstupu a výstupu. V JAX-RS k tomu slouží anotace `@Consumes` a `@Produces`. Pomocí nich lze jednoduše vymezit, zda bude metoda vracet XML, JSON nebo jiný formát definovaný v hlavičce klientského požadavku.

Podporované formáty jsou v anotacích definovány pomocí MIME media typu, který daná metoda či třída akceptuje (`@Consumes`), případně poskytuje (`@Produces`). V rámci anotací je možné uvést i více typů pomocí pole obsahujícího jednotlivé typy (např. `@Produces ({"text/plain, text/xml})`. V případě definice více typů je pro obsloužení požadavku zvolen typ, který nejlépe odpovídá specifikaci v hlavičce klientského požadavku.

Pokud klientský požadavek odesílá data ve formátu, který není uveden v @Consumes (tj. není podporován), je vyvolána chyba HTTP *„415 Unsupported Media type error“*. Podobně v případě nepodporovaného výstupu je vyvolána chyba HTTP *“406 Not Acceptable error“*.

Anotace jsou aplikovatelné v rámci metody nebo třídy. Anotace na úrovni třídy se vztahuje na všechny metody takto anotované třídy, ale může být překryta anotací uvedenou přímo u metody. V případě absence výše probíraných anotací jsou akceptovány všechny MIME media typy a to platí jak pro výstupní, tak i vstupní data.

Tento způsob omezení vstupu a výstupu nám umožňuje definovat metody, které poskytují stejné informace, avšak v jiných formátech. To je výhodné např. pro aplikaci běžící na mobilním telefonu. Ten bude pravděpodobně požadovat nějaký velmi jednoduchý formát (např. obyčejný text). Na druhou stranu pokud službu na stejném URI použije klasický prohlížeč, bude pravděpodobně očekávat XML nebo jiný pokročilejší formát. Takovou situaci ilustruje následující příklad:

```
@Path("customers/{id :[0-9]*}")
public class HelloWorldResource {
    @GET
    @Produces("text/plain")
    public String getCustomerAsText(@PathParam(value="id") String id){
        //build text response
    }
    @GET
    @Produces("text/xml")
    public String getCustomerAsXml(@PathParam(value="id") String id){
        //build XML response
    }
}
```

Příklad 3.6.1: Omezení vstupu a výstupu

První metoda v příkladu `getCustomerAsText` vrátí uživatele reprezentovaného pomocí obyčejného textu. Naproti tomu metoda `getCustomerAsXml` vrátí reprezentaci uživatele v XML formátu. Obě budou dostupné na stejném URI a o tom, zda se provede první nebo druhá rozhodne hlavička klientského požadavku.

Velmi podobným způsobem funguje také anotace `@Consumes`, která nám umožní podobný výběr metody na základě typu klientem poslaných dat.

3.7 Mapování entit a sestavení odpovědi na požadavek

V předchozích sekcích jsme si ukázali, jakým způsobem je možné získat parametry metody z hlavičky požadavku, URI, nebo jiných parametrů HTTP požadavku. Dále jsme si ukázali, jak je možné definovat formát výstupu pomocí anotace. Jakým způsobem však dosáhnout toho, že výstupem bude skutečně definované XML, či jak pracovat s daty, které jsou uložena v těle HTTP požadavku? Přesně z tohoto důvodu jsou ve standardu definovány tzv. poskytovatelé entit (*Entity providers*). V JAX-RS existují základní dva typy poskytovatelů:

- **MessageBodyReader** – používá se pro čtení dat v těle HTTP požadavku a jejich následné mapování na Java objekty.
- **MessageBodyWriter** – je inverzí k předchozímu poskytovateli a používá se pro převedení Java objektu do odpovídajícího formátu v těle HTTP odpovědi.

Každý poskytovatel je vázaný s určitou třídou v Javě. Standard poskytuje sadu výchozích poskytovatelů, mezi které patří poskytovatelé pro typy byte[], String, File, ale také pro pokročilejší typy jako JAXBElement (typ pro mapování mezi XML a objekty v Javě). Kompletní seznam poskytovatelů naleznete v [11] sekce 4.2.4.

V drtivé většině aplikací si vystačíme s těmito standardními poskytovateli, ale je také možné definovat vlastního poskytovatele. Stačí pouze rozšířit jeden ze dvou základních typů poskytovatelů a uvést nad definicí rozšiřující třídy anotaci `@Provider`. Vlastní poskytovatel třídy Person by mohl začínat takto:

```
@Provider
public class CustomProvider implements MessageBodyReader<Person> {
//implement abstract methods
}
```

Příklad 3.7.1: Kostra poskytovatele pro třídy Person

3.7.1 Sestavení odpovědi na požadavek

Díky výše zmíněnému mapování mezi Java typy a tělem požadavku či odpovědi lze jako návratovou hodnotu použít přímo podporované Java typy, případně typy, pro které existuje poskytovatel. Pokud však potřebujeme vytvořit komplikovanější odpověď (např. přidat chybový návratový kód nebo nějaké parametry do hlavičky), pak můžeme využít objekt `ResponseBuilder`.

```
@GET
@Produces("text/xml")
public Response getXml(@PathParam(value="id") Long id) {
    Person person = null;
    try {
         person = personProvider.getPersonById(id);
        return Response.ok(person).build();
    } catch (Exception ex) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
```

Příklad 3.7.2: Využití třídy ResponseBuilder

Příklad znázorňuje jednoduché použití `ResponseBuilder`. V případě, že osoba s daným ID neexistuje, je navrácen chybový kód 404 (*not found*). Naopak při úspěchu je vrácen kód 200 a detail osoby. Tento způsob je velice pohodlný, avšak pro Java programátora je většinou chyba reprezentována pomocí nějakého typu výjimky. Následující sekce demonstruje, jak správně přistoupit ke zpracování výjimek.

3.7.2 Zpracování výjimek

S výjimkami je možné pracovat dvěma způsoby. Buď jsou všechny výjimky definované jako potomci třídy `WebApplicationException`, nebo jsou mapovány pomocí třídy `ExceptionMapper`. První způsob vynutí použití objektu Response v konstruktoru rozšiřující výjimky a tím zajistí vybudování správné odpovědní zprávy klientovi. Druhý způsob se řídí podobnou filozofií jako mapování Java typů na požadavek či odpověď. Jediný rozdíl je v použití jiné třídy pro rozšíření `ExceptionMapper` místo `MessageBodyWriter/Reader`. Více viz [11] sekce 4.4.

3.8 Závislé zdroje (zdrojové třídy)

Pokud označíme metodu pomocí anotace `@Path`, dojde k vytvoření metody závislého zdroje (*sub resource method*), případně identifikátoru závislého zdroje (*sub resource locator*).

První uvedený zpracovává HTTP požadavek přímo, tj. metoda označená `@Path` vykonává potřebné operace pro vytvoření odpovědi přímo. Specifikace označuje tento typ jako přítomný závislý zdroj (*present sub resource*). Z pohledu zpracování požadavku se postupuje stejně jako u kořenových metod, pouze s rozdílným URI. To je vytvořeno spojením obsahů v anotacích `@Path` u metody a třídy.

```
@Path("/rootservice")
public class SubResourceExample {
    @GET
    @Path("subres")
    @Produces("text/plain")
    public String presentSubResourceMethod() {
        return "I'm present sub-resource method";
    }
}
```

Příklad 3.8.1: Přítomný závislý zdroj (present sub resource)

Pro zavolání této metody by URI muselo vypadat následovně `/rootservice/subres`. Můžeme si povšimnout, že v anotaci `@Path` u metody chybí lomítko. To je automaticky doplněno při spojování s URI, definovaného v rámci třídy a metody.

Naproti tomu identifikátor závislého zdroje se vyznačuje tím, že nezpracovává HTTP požadavek přímo a na místo toho vrací objekt, který zodpovídá za zpracování požadavku. Tento typ je označován jako nepřítomný závislý zdroj (*absent*). Objekt, který zpracovává požadavek, může být dynamicky přidělen dle parametrů specifikovaných v URI, hlavičce nebo v jiných parametrech uvedených v HTTP požadavku. Zpracování se pak řídí anotacemi obsaženými v tomto objektu.

```
// root resource class
@Path("/admin")
public class Administration {
    @Path("{id}")     // Sub-resource locator method for {id}
    public User getUser(@PathParam("id") String id) {
        return User.getInstance(id);
} }
// sub resource handling request
public class User {
    private String id;
    … //constructor should be placed here
    @GET
    @Produces("application/xml")
    public String getUserDetails() {//code returning user details}
    @GET
    @Path ("orders")
    public String getUserOrders() {//code returning user orders}
}
```

Příklad 3.8.2: Nepřítomný závislý zdroj (absent sub resource)

Pokud je služba adresována pomocí URI `/admin/1` dochází k vyvolání metody `getUser()`. V rámci této metody je vytvořen objekt `User`, který je zodpovědný za obsloužení požadavku, což učiní pomocí své metody `getUserDetails()`. Pokud by však uživatel specifikoval URI končící `/admin/1/orders`, pak by byl požadavek obsloužen metodou `getUserOrders()` objektu `User`.

Využití:

Tento princip je vhodné použít v případě hierarchické struktury uživatelů, zákazníků či jiných objektů zájmu. Každý druh může obsahovat jiné informace a může mít implementované svoje vlastní metody pro získání detailů nebo jiných dat. Případně může poskytovat i jiné rozšiřující metody.

3.9 Dědičnost jednotlivých anotací

Specifikace umožňuje použití JAX-RS anotací u metod definovaných v rozhraní nebo v nadtřídách. Anotace se zdědí do příslušné implementace rozhraní nebo do potomka nadtřídy v případě, že potomek nemá definované žádné vlastní JAX-RS anotace. Naopak pokud je již

např. nad metodou použita nějaká z JAX-RS anotací, jsou ostatní (byť nepřekrývající se) anotace z rozhraní vynechány. Vše ilustrují následující dva příklady:

```
public interface ReadOnlyPersonXML {
   @GET
   @Produces("application/*+xml")
   String getPerson();
}

@Path("personFeed")
public class PersonService implements ReadOnlyPersonXML {
   public String getPerson() {...}
}
```

Příklad 3.9.1: Dědičnost anotací 1

Třída `PersonService` z výše uvedeného příkladu podědí definici anotací `@GET` a `@Produce` z rozhraní `ReadOnlyPersonXML`. Naproti tomu v následujícím příkladu třída `PersonService` nepodědí anotaci `@GET`, protože sama definuje nějakou anotaci ze skupiny JAX-RS anotací nad metodou `getPerson()`.

```
@Path("personFeed")
public class PersonService implements ReadOnlyPersonXML {
   @Produces("application/*+xml")
   public String getPerson() {...}
}
```

Příklad 3.9.2: Dědičnost anotací 2

Tento postup se velmi často využívá ve dříve zmiňovaných závislých zdrojích.

3.10 Co dále?

V předcházejících sekcích jsem se pokusil shrnout hlavní body, které činí JAX-RS, dle mého názoru, velmi perspektivním standardem. I přes mou snahu zachytit z něj co nejvíce, jsou části, které nebyly důkladně pokryty a vyžadují další studium. K některým částem jsou uvedeny přímo odkazy v sekcích, ostatní je možné dohledat buď přímo ve specifikaci[11] nebo v materiálech k referenční implementaci poskytovaných firmou Oracle (např. [13]).

4. Konkrétní implementace webových služeb v Javě

Tato kapitola představuje čtyři možnosti implementace webových služeb v Javě. Dvě z nich se věnují webovým službám v duchu REST a ostatní službám založeným na SOAP protokolu. V závěru jsou srovnány výhody a nevýhody jednotlivých implementací. Tato kapitola je přípravnou půdou pro srovnání dvou vybraných zástupců (jeden reprezentant SOAP a jeden REST) na praktickém příkladu.

4.1 Implementace REST služby pomocí Servletu

Jako první si představíme triviální způsob implementace REST webové služby pomocí *Java Servlet API* a třídy `Servlet`. Jedná se o základní výbavu každého webového kontejneru v Javě. Typickým reprezentantem může být například server Tomcat. Servlet API poskytuje základní funkce pro práci s HTTP požadavky v jazyce Java. Umožňuje definici vlastních metod pro adekvátní reakci na jednotlivé HTTP požadavky (definované v tabulce 1.1.1) pomocí metod `doGet`, `doPost` a podobně. Každému servletu je v rámci webové aplikace přiděleno URI, případně vzor pro URI určující, zda uvedené URI ukazuje na tento servlet. Jedná se o jedno ze základních API používaných pro vývoj webových aplikací v Javě, proto předpokládám jeho znalost čtenářem. Při výkladu si vystačíme s výše uvedenými informacemi. Více informací o Servlet API je pak možné získat v [9], [10].

Jak by tedy fungovala taková webová služba REST naimplementovaná pomocí servletu? Stačí vytvořit jednoduchou webovou aplikaci obsahující jeden servlet. Každá webová aplikace má svoje kořenové URL . Od tohoto URL je možné definovat URL servletu pomocí souboru web.xml (deskriptor pro webové aplikace obsahující nastavení, jako podporované jazyky, mapování servletu na URL a mnoho dalších). Mapováním servletu na URL zajistíme, aby takto mapovaný servlet odpovídal na požadavky odeslané na definované URL.

```
<servlet>
  <servlet-name>PersonService</servlet-name>
    <servlet-class>
        cz.muni.fi.simpliestrestimplementation.servlet.PersonService
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>PersonService</servlet-name>
    <url-pattern>/PersonService/*</url-pattern>
</servlet-mapping>
```

Příklad 4.1.1: Definice mapování servletu v web.xml

Takto definovaný servlet bude odpovídat na všechny URL, které mají formát kořenové URL aplikace + `/PersonService/` + cokoliv (např. `kontextaplikace/PersonService/1`. Samotná validace výsledného URL může být již provedena v obsluhujících metodách ser-

vletu. Posledním krokem pro vytvoření služby je naprogramování obslužných metod pro jednotlivé požadavky, tj. pro vytvoření nejjednodušší služby `PersonService` stačí pouze naimplementovat metodu `doGet` a zpracovat URL a hlavičky použité v `GET` metodě volající tento servlet. Velmi jednoduchá implementace pro výše uvedený příklad URL by mohla vypadat takto:

```
@Override
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {

  try {
     String pathInfo = request.getPathInfo();
     pathInfo = pathInfo.replace("/", "");
     Long id = Long.parseLong(pathInfo);
     Person resultPerson = providerBean.getPersonById(id);

     PrintWriter writer = response.getWriter();
     append("User id: " + resultPerson.getId()).append("\n");
      writer.append("User name: " + resultPerson.getName()).
             append("\n");
     writer.append("User surname: "+ resultPerson.getSurname()).
             append("\n");
  } catch (Exception ex) {
     response.sendRedirect("/SimpliestRestImplementation" +
                           "/error.jsp");
  }
}
```

Příklad 4.1.2: Elementární REST služba pomocí servletu

Výše uvedený příklad je opravdu elementární implementací. Servlet si pomocí `getPathInfo` zpracuje URL a získá z něj ID osoby, které potom použije pro načtení osoby z `providerBean`. Ta slouží jako poskytovatel osob. V poslední fázi zpracování požadavku zapíše servlet do odpovědi textovou reprezentaci nalezené osoby. V případě chyby přesměruje uživatele na stránku s chybovým hlášením.

Takto by mohla fungovat úplně nejjednodušší REST služba (používá jednotné rozhraní, běží na serveru, zdroj je jednoznačně identifikován atd.). Avšak při detailnějším pohledu si můžeme všimnout, že služba vůbec neřeší problematiku HTTP hlaviček. Např. vyžadovaný návratový formát (hlavička `Accept`). Místo toho vrací pokaždé stejný textový výsledek. Stejně tak zpracovaní URL je velmi triviální a neošetřuje žádné chybové stavy, na které by správně měla služba odpovídat chybovými kódy HTTP. Všechny tyto náležitosti je možné řešit pomocí metod Servlet API, avšak náročnost např. zpracovaní URL nebo práce s hlavičkami a samotným budováním odpovědi je poměrně vysoká. Pokud porovnáme tento způsob práce s HTTP požadavky s dříve popsaným standardem JAX-RS, můžeme si všimnout rozdílu ve složitosti práce s parametry, výstupními formáty atd. Samotné srovnání provedu v rámci poslední sekce této kapitoly.

4.2 Jersey – referenční implementace JAX-RS

Druhým zástupcem konkrétní implementace webových služeb REST je Jersey, která je také referenční implementací JAX-RS (JSR-311). Referenční implementace je typický způsob, jak podat ukázkovou implementaci, která splňuje všechny podmínky požadované standardem a tudíž slouží jako vzor pro další implementace tohoto API. Díky tomuto se na Jersey vztahují veškerá omezení, ale i výhody popisované v kapitole 3. V této kapitole si ukážeme pouze některá drobná rozšíření standardu, která Jersey poskytuje a standard jako takový o nich nemluví (např. klient pro webovou službu REST, omezení životního cyklu).

4.2.1 Nasazení aplikace

Jersey implementuje JAX-RS pomocí servlet kontejneru. Ten je registrován definicí v souboru web.xml (Java EE 5), nebo pomocí potomka třídy `javax.ws.rs.core.Application` a anotace `@ApplicationPath` (Java EE 6, Jersey 1.1.4 a lepší), v níž je specifikováno bázové URI, od kterého se odvozují konkrétní URI uvedené v `@Path` jednotlivých zdrojových tříd. Následující příklady demonstrují tyto přístupy:

```
@javax.ws.rs.ApplicationPath("resources")
public class ApplicationConfig extends
                                javax.ws.rs.core.Application {
}
```

Příklad 4.2.1: Registrace pomocí anotace (Java EE6)

```
<servlet>
  <servlet-name>ServletAdaptor</servlet-name>
    <servlet-class>
        com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>ServletAdaptor</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

Příklad 4.2.2: Registrace pomocí definice ve web.xml (Java EE5)

Z příkladů je zřejmé, že Java EE 6 společně s novou verzí Jersey, přinášejí značné zjednodušení celé definice. URI, na kterém budou jednotlivé služby vystaveny, je v prvním případě určeno pomocí anotace `@ApplicationPath("resources")`, v druhém případě potom pomocí obsahu elementu `url-pattern`. Pokud použijeme jednu z výše uvedených inicializací, pak bude služba anotovaná `@Path("testPath")` dostupná na adrese `kontextAplikace/resources/testPath`. V takto nakonfigurované aplikaci již můžeme využít všechny funkce popsané v rámci kapitoly 3.

4.2.2 Rozšíření životního cyklu

Jersey v tomto ohledu rozšiřuje standard z pohledu vytvářených instancí zdrojových tříd. Připomeňme si, že výchozí chování dle JAX-RS je vytváření nové instance zdrojové třídy pro každý

požadavek, což může mít za následek zpomalení a velkou paměťovou náročnost. Z tohoto důvodu přichází referenční implementace s rozšířením životního cyklu pomocí anotací `@Singleton` a `@PerSession`. Zdrojová třída, označená pomocí `@Singleton`, má pouze jednu instanci v rámci celé aplikace. U `@PerSession` je vytvořena separátní instance pro každou webovou relaci (*web session*) [13].

4.2.3 Jersey klient

Ačkoliv specifikace samotná toto nevyžaduje, referenční implementace poskytuje způsob, jak jednoduše konzumovat webové služby REST v Javě. K tomuto slouží tzv. *Jersey Client API*. Toto API umožňuje jednoduchým způsobem pracovat s jakoukoliv implementací REST webové služby, tj. nemusí se jednat konkrétně o služby definované pomocí JAX-RS. Klient je současně velmi dobře využitelný pro automatizované testy REST služeb.

4.2.4 Princip práce s klientem

Každý zdroj je obalen pomocí objektu `WebResource`, který v sobě ukrývá URI zdroje, umožňuje budování těla klientského požadavku a poskytuje metody, které odpovídají HTTP metodám pro odeslání takto vytvořených požadavků. Tímto způsobem lze pomocí jednoduchých metod nastavit HTTP hlavičky, případně tělo zprávy a přitom je možné využít principů použitých v JAX-RS (např. mapování mezi Java objekty a tělem HTTP zprávy viz sekce 3.7).

Pro vytvoření objektu reprezentujícího zdroj je nutné klienta inicializovat vytvořením objektu `Client`. Ten slouží jako základní přístupový bod do Jersey Client API. Jedná se o *heavy-weight* objekt (náročný na inicializaci), proto je dobré se vyvarovat jeho opětovnému vytváření a nejlepší je používat jednu instanci pro vytváření více zdrojů.

```
Client jersyClient = new Client();
WebResource personService =
                    jersyClient.resource("http://localhost" +
                          "8080/REST/resources/PersonService/");

personService.queryParam("overwrite", true").
                type(MediaType.TEXT_XML).
               post(new PersonXML("Jan", "Novák"));
```

Příklad 4.2.3: Klient REST služby

Uvedený příklad znázorňuje vytvoření zdroje a následné odeslání požadavku POST na tento zdroj. Můžeme si všimnout, že před samotným odesláním je nastaven parametr `overwrite` na `true` a také MIME typ odesílaných dat ve zprávě na `TEXT_XML`. Podobným způsobem je možné také nastavit *cookies*, HTTP hlavičky a další parametry požadavku. Velmi pěknou aplikací myšlenky pocházející z JAX-RS je ukázka odeslání objektu `PersonXML`. Ten jen automaticky převeden na XML reprezentaci právě pomocí mapování mezi Java objekty a tělem HTTP zprávy.

4.2.5 Zhodnocení

Jersey je velmi dobře propracovanou implementací JAX-RS, poskytuje dobrou dokumentaci včetně příkladů, jak toto API použít. Rozšiřuje standardem definované API o velmi užitečné funkce. Za nejpodstatnější rozšíření bych označil klienta pro Java aplikace. Velice dobrým zdrojem dalších informací je [14], kde jsou velmi detailně popsané postupy, jak používat další funkce poskytované touto implementací.

4.3 JAX-RPC

Další konkrétní implementace/standard, který si představíme je JAX-RPC (JAX-RPC RI – jako referenční implementace). Na rozdíl od dvou předešlých se jedná o implementaci služeb pracujících na protokolu SOAP (viz sekce 1.2.1). Podle typů služeb W3C, pak JAX-RPC realizuje libovolné webové služby (*arbitrary web services*). Jak již název napovídá, jedná se o implementaci vzdáleného volání procedur (RPC - *Remote Procedure Call*) pomocí protokolu SOAP. Později si ukážeme, že podobnost postupu implementace webových služeb JAX-RPC s implementací vzdáleného volání procedur pomocí Java RMI je více než zřejmá.

Samotný formát SOAP zpráv je velmi komplikovaný a pokud by se měl vývojář zabývat jeho konstrukcí, byl by celý vývoj aplikace složitý a výrazně pomalejší. Hlavní ideou standardu JAX-RPC je nabídnout takové rozhraní pro vývoj, které vývojáře oprostí od složitosti SOAP a WSDL a nabídne mu pohodlný způsob, jak definovat rozhraní a implementaci služby pomocí Java rozhraní a tříd. Samotné WSDL služby a jednotlivé SOAP zprávy jsou pak generovány právě pomocí JAX-RPC.

4.3.1 Postup při implementaci služby

Samotná implementace služby v JAX-RPC staví na podobných principech jako implementace komunikace pomocí RMI v Javě. Jako první je nutné naimplementovat tzv. koncový bod služby (*Service Endpoint*). Každý koncový bod je definován rozhraním a jeho implementací. Ty definují funkce poskytované službou. Rozhraní koncového bodu musí splňovat následující podmínky [13]:

- Musí rozšiřovat `java.rmi.Remote`
- Nesmí obsahovat definice konstant `public final static`
- Každá metoda musí vyvolávat výjimku `java.rmi.Remote` nebo jejího potomka
- Parametry a návratové hodnoty metody musí patřit do podporovaných typů, viz [14]

Velice dobrou ukázkovou implementaci a podrobný popis lze najít v [14]. Inspirace pro tento příklad byla čerpána z [14].

```
import java.rmi.RemoteException;

public interface RPCServiceSEI extends java.rmi.Remote {
  public String sayHelloTo(String name) throws RemoteException;
}
```

Příklad 4.3.1: HelloWorld služba a rozhraní v JAX-RPC

```
import java.rmi.RemoteException;

public class RPCServiceImpl implements RPCServiceSEI {
  private String hello = "hello ";

  public String sayHelloTo(String name) throws RemoteException{
        return hello + name;
  }
}
```

Příklad 4.3.2: HelloWorld služba a implementace v JAX-RPC

Posledním krokem před nasazením služby je definice mapovaní koncového bodu na WSDL, což demonstruje následující příklad:

```
<?xml version='1.0' encoding='UTF-8' ?>
<configuration xmlns='http://java.sun.com/xml/ns/jax-
                      rpc/ri/config'>
  <service name='RPCService'
           targetNamespace='urn:RPCService/wsdl'
           typeNamespace='urn:RPCService/types'
           packageName='cz.muni.fi.jaxrpc.services'>
  <interface name='cz.muni.fi.jaxrpc.services.RPCServiceSEI'
        servantName='cz.muni.fi.jaxrpc.services.RPCServiceImpl'>
        </interface>
    </service>
</configuration>
```

Příklad 4.3.3: Mapování koncového bodu na WSDL JAX-RPC

Toto mapování udává, že služba se jmenuje `RPCService`, WSDL používá výchozí jmenný prostor `urn:RPCService/wsdl`, atd. Nejdůležitější je však část definice interface, která určuje rozhraní koncového bodu a jeho implementaci.

Takto vytvořenou službu lze zkompilovat pomocí příkazu `wscompile`, který automaticky vygeneruje dokument WSDL, popisující její rozhraní.

Problematiku klienta této služby nebudu pro tuto implementaci rozebírat, ale ve své podstatě se JAX-RPC drží své hlavní myšlenky (jednoduchost implementace) a nabízí možnost vygenerovat základní strukturu, kterou lze využít pro volání služby. Více informací jak pracovat s JAX-RPC viz.[14].

4.3.2 Zhodnocení

Toto API bylo vyvinuto v rámci specifikace Java EE 1.4 a je kompatibilní s běhovým prostředím Java 1.4. Z toho plynou i jistá omezení v použití pro Javu jinak běžných věcí, jako jsou generické typy nebo anotace. Uvádím jej hlavně z historických důvodů a jako ukázku, kam směřuje vývoj webových služeb v Javě. Podobnost s implementací vzdáleného volání pomocí Java RMI může být výhodou pro některé vývojáře, protože postupy a filozofie pro ně budou velmi podobné. Další výhody a nevýhody budou probrány na konci této kapitoly. Jako zdroj dalších informací doporučuji např. [14].

4.4 JAX-WS

Druhý zástupce SOAP služeb v této práci je JAX-WS. Ten byl vyvinut jako následník dříve zmiňovaného standardu JAX-RPC. Ve své podstatě JAX-RPC 2.0 bylo přejmenováno na JAX-WS 2.0. Hlavní motivace zůstává stejná jako u předchůdce, tj. poskytnutí rozhraní, které značně ulehčí vývoj webových služeb a odstíní Java programátora od WSDL a SOAP zpráv.

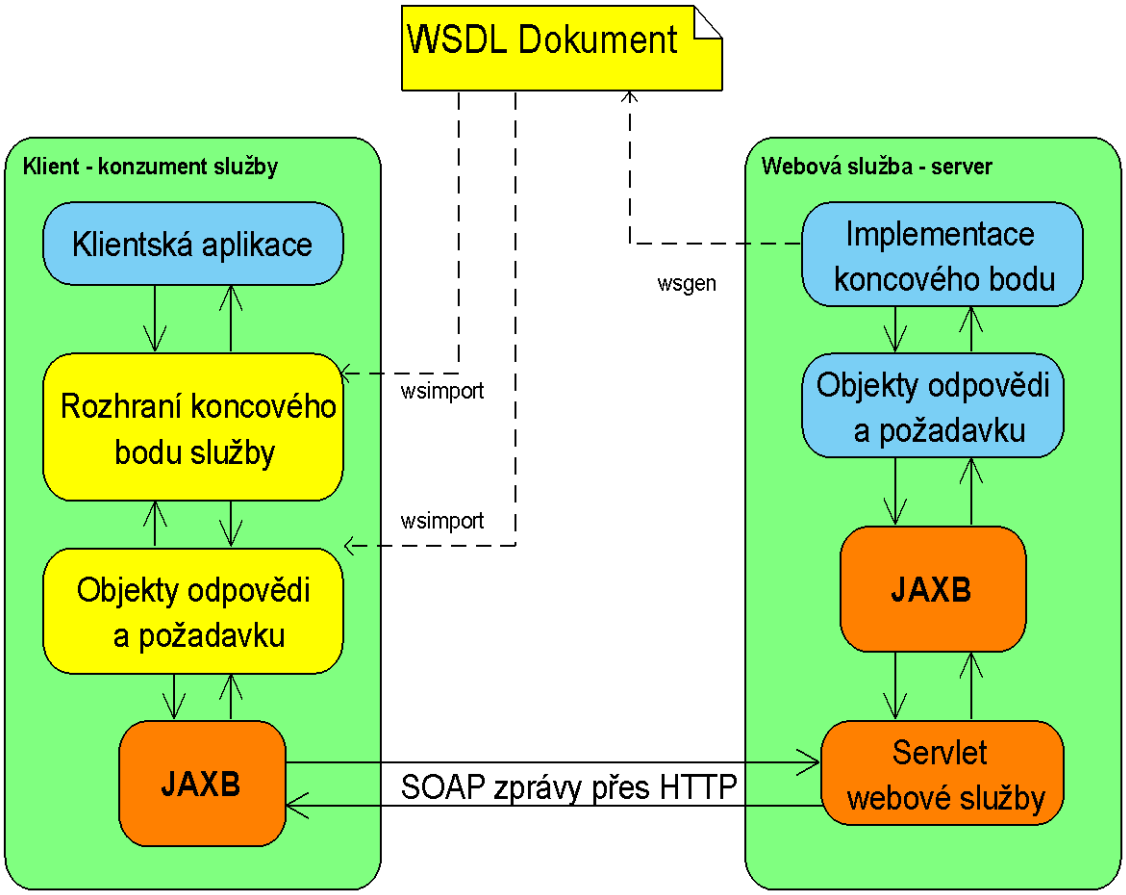
Jako správný nástupce s sebou JAX-WS přináší mnoho vylepšení. Tím nejvýraznějším je pak přechod na Javu 5 a s ní související příchod anotací a dalších výhod. Hlavním rozdílem je zjednodušení práce s koncovým bodem, kde byla přepracována filozofie převzatá z RMI a aktuální podoba (anotovaný POJO objekt) byla inspirací pro JAX-RS. Toto API není zpětně kompatibilní s předchozími verzemi Javy právě z důvodu použití anotací, což je i jeden z důvodů přejmenování. Druhým důvodem je odstup od RPC stylu při vytváření koncových bodů. Následující sekce shrnuje hlavní rozdíly JAX-WS oproti předchozímu JAX-RPC.

4.4.1 Hlavní novinky a přínosy oproti JAX-RPC

- Zavádí masivní používání anotací a celkově zjednodušuje kód pro vývoj služby.
    - Odpadají další konfigurační soubory např. mapování koncového bodu.
    - Celkově menší množství generovaných tříd – menší a kompaktnější aplikace (jak poskytovatelé, tak klienti).
- Změna konverze dat do SOAP zpráv – delegováno na JAXB (Java API for XML Binding), které značně vylepšuje možnosti mapování Java objektů na XML elementy. JAXB zaručuje 100% podporu XML schématu, na rozdíl od předchozí JAX-RPC, kde je definována určitá podmnožina typů z XML schématu definicí vlastních pravidel. Více o JAXB naleznete v [16].
- Podporuje novější protokoly (např. SOAP 1.2).
- Podporuje oba přístupy k mapování jak RPC styl mapování, tak i styl orientovaný na zprávy/dokumenty (*message/document oriented*). Podrobnosti a srovnání těchto mapovaní naleznete v [17].

4.4.2 Princip práce JAX-WS

V této sekci bych rád demonstroval, jakým způsobem probíhá komunikace mezi klientem a poskytovatelem služeb v JAX-WS. Následující obrázek znázorňuje tuto komunikaci (inspirováno [18]):



Obrázek 4.4.1: Jak pracuje JAX-WS, hlavní komponenty

Obrázek demonstruje, jak probíhá komunikace klienta a webové služby. Barevné rozlišení má následující význam:

- žluté komponenty – generovaný kód;
- modré komponenty – píše vývojář;
- oranžové komponenty – poskytuje JAX-WS API a další související knihovny, které JAX-WS využívá jako např. JAXB.

Vývojář je nucen implementovat koncový bod a případně specifikovat objekty odpovědi a požadavku (viz následující sekce). Z těchto dvou je schopen pomocí `wsgen` příkazu, který JAX-WS poskytuje, vytvořit WSDL dokument reprezentující rozhraní služby. Vývojář klientské aplikace může potom použít inverzní příkaz `wsimport`, který poskytne rozhraní koncového bodu a objekty pro komunikaci se službou. Komunikace probíhá následujících způsobem:

- Klient pomocí vygenerovaného rozhraní koncového bodu volá metody poskytované webovou službou. Jako parametry uvádí objekty požadavku.

- Objekty požadavku jsou pomocí JAXB převedeny do formátu XML, který reprezentuje tělo SOAP zprávy. Tělo zprávy je následně obaleno SOAP obálkou a odesláno pomocí HTTP protokolu na URL webové služby.
- Servlet webové služby rozhodne, o kterou ze služeb jde (vybere správnou třídu, která má požadavek obsloužit). Rozbalí obálku a poskytne její tělo JAXB.
- JAXB převede obsah těla zpět z XML do formátu Java objektů a ty jsou předány jako vstupní parametry implementaci koncového bodu, která vykoná požadované operace a odpověď pošle podobným způsobem jako klient svůj požadavek.

Uvedená ukázka se zaměřuje čistě na komunikaci klienta a webové služby, které jsou implementovány pomocí JAX-WS. Je však možné kteroukoliv ze stran zaměnit, tj. klienta nebo webovou službu nahradit jinou implementací v jiném programovacím jazyce a to právě díky web service interoperability standardu, který JAX-WS splňuje.

4.4.3 Webová služba v JAX-WS

Nyní již máme představu, jak probíhá komunikace mezi klientem a webovou službou. Jako poslední část krátkého představení JAX-WS bych rád ukázal, jak vytvořit jednoduchou reprezentaci služby. Všimněte si patrného posunu oproti předchozímu JAX-RPC.

Jako obvykle je nutné začít implementací koncového bodu. Ta je značně zjednodušena a stačí pouze jedna třída anotovaná pomocí `@WebService`. Pak již stačí pouze definovat její metody. Ty musí být opatřeny anotacemi `@WebMethod`, které určují název operace webové služby. Definice parametrů metody pak probíhá pomocí anotace `@WebParam`, která určuje jméno parametru operace služby. Stejný příklad jako u JAX-RPC by v případě JAX-WS vypadal následovně:

```
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService()
public class TestWebService {

    private String hello = "hello ";

    @WebMethod(operationName = "sayHelloTo")
    public String sayHelloTo(@WebParam(name = "name")
                                String name) {
        return hello + name;
    }
}
```

Příklad 4.4.1: Implementace koncového bodu v JAX-WS

Toto je vše, není již potřeba další soubor mapování nebo rozhraní. K takto definované službě stačí pomocí `wsgen` vytvořit WSDL dokument a službu vystavit společně s WSDL na aplikačním serveru.

4.5 Srovnání přístupů, hlavní výhody a nevýhody

V předcházejících čtyřech sekcích jsme si představili různé způsoby implementace webových služeb v Javě. Konkrétně šlo o dva reprezentanty pro vývoj REST služeb a dva pro vývoj služeb, komunikujících protokolem SOAP. V této sekci bych rád srovnal jednotlivé implementace nejdříve mezi sebou dle stylu (SOAP a REST) a vybral dva kandidáty, jejichž srovnání bude obsahem závěrečného příkladu.

4.5.1 Jersey vs. Servlet API (REST)

Servlet API bylo demonstrováno jako primitivní implementace REST služby. Dá se říci, že tento účel splnilo, ale jeho využití z pohledu složitějších aplikací je přinejmenším složité a tudíž nemůže konkurovat Jersey. Na druhou stranu poskytuje velmi dobrou představu, jak složité by bylo například samotné zpracování URI, bez anotací poskytnutých právě v Jersey. Jistě lepším protivníkem ve srovnání s Jersey by byly rámce jako je například Restlet. Ty však nejsou předmětem této práce. Dále stojí za zmínku, že Servlet API je bázovým API pro všechny standardy JAX-*, tj. každý z něho používá větší či menší část. Na závěr bych rád shrnul hlavní přínosy Jersey (resp. JAX-RS).

Výhody Jersey

- Reprezentace služby pomocí POJO – vytvoření služby pomocí anotací.
- Jednoduchá práce s URI a možnost URI šablon – jednoduchá práce s parametry v URI.
- Podpora MIME formátu a možnost jednoduchého rozšíření.
- Mapování objektu na MIME typy – možnost vazby mezi tělem HTTP požadavku a Java objekty.
- Podpora pro budování odpovědí včetně chybových kódů HTTP.
- Vývoj čistých REST služeb.

4.5.2 JAX-RPC vs. JAX-WS (SOAP)

Standard JAX-WS můžeme označit za právoplatného následníka JAX-RPC. Ukázalo se, že jeho výhoda není pouze v anotacích a přechodu na Javu 5. JAX-WS přináší celou řadu nových myšlenek a hlavně mnohem lepší pružnost v definici vazby Java typů na XML díky JAXB. Další výhody oproti JAX-RPC jsem již uvedl v rámci sekce 4.4.1, kde jsou shrnuty všechny novinky a většina z nich by se dala považovat za výhody.

Na druhou stranu, pokud potřebujeme zpřístupnit funkce pomocí webových služeb v rámci Javy 1.4, nemáme jinou volbu než JAX-RPC. Pro programátory znalé postupů implementace vzdáleného volání pomocí Java RMI může být výhodou použití stejné filozofie.

Vítězem je však jednoznačně JAX-WS, který posouvá jednoduchost vývoje webových služeb SOAP zas o krůček dále a stává se po Jersey druhým kandidátem pro srovnání v rámci závěrečného příkladu.

5. REST vs. SOAP v praxi

Z předcházející sekce byli vybráni reprezentanti dvou různých přístupů k budování webových služeb, konkrétně JAX-WS (SOAP) a JAX-RS (REST). Přístupy REST a SOAP byly již porovnány na teoretické úrovni v sekci 2.5.2. V této kapitole se zaměřím nejen na body vytyčené v této teoretické sekci, ale také se pokusím zhodnotit složitost samotného vývoje pomocí vybraných API. Výše uvedení reprezentanti jsou pouze standardy, nikoliv implementacemi. V tomto srovnávacím příkladu budou použity referenční implementace tj. Jersey (JAX-RS) a Glassfish JAX-RS RI (JAX-WS).

5.1 Zadání příkladu

Jako demonstrační příklad jsem zvolil zjednodušenou verzi systému pro inzerci na internetu. Systém bude poskytovat následující sadu funkcí:

- vložení nového inzerátu;
- úprava existujícího inzerátu;
- zrušení/smazání inzerátu – zrušení zachová informace o inzerátu a smazání ho odstraní z databáze inzerátů;
- vyhledávání v inzerátech podle data zveřejnění, názvu a kategorie inzerátu.

Tyto funkce by měly postačit na srovnání základních principů SOAP a REST. Účelem příkladu není poskytnout komplexní systém, proto nebudou implementovány funkce běžně očekávané od takového systému, jako jsou např.:

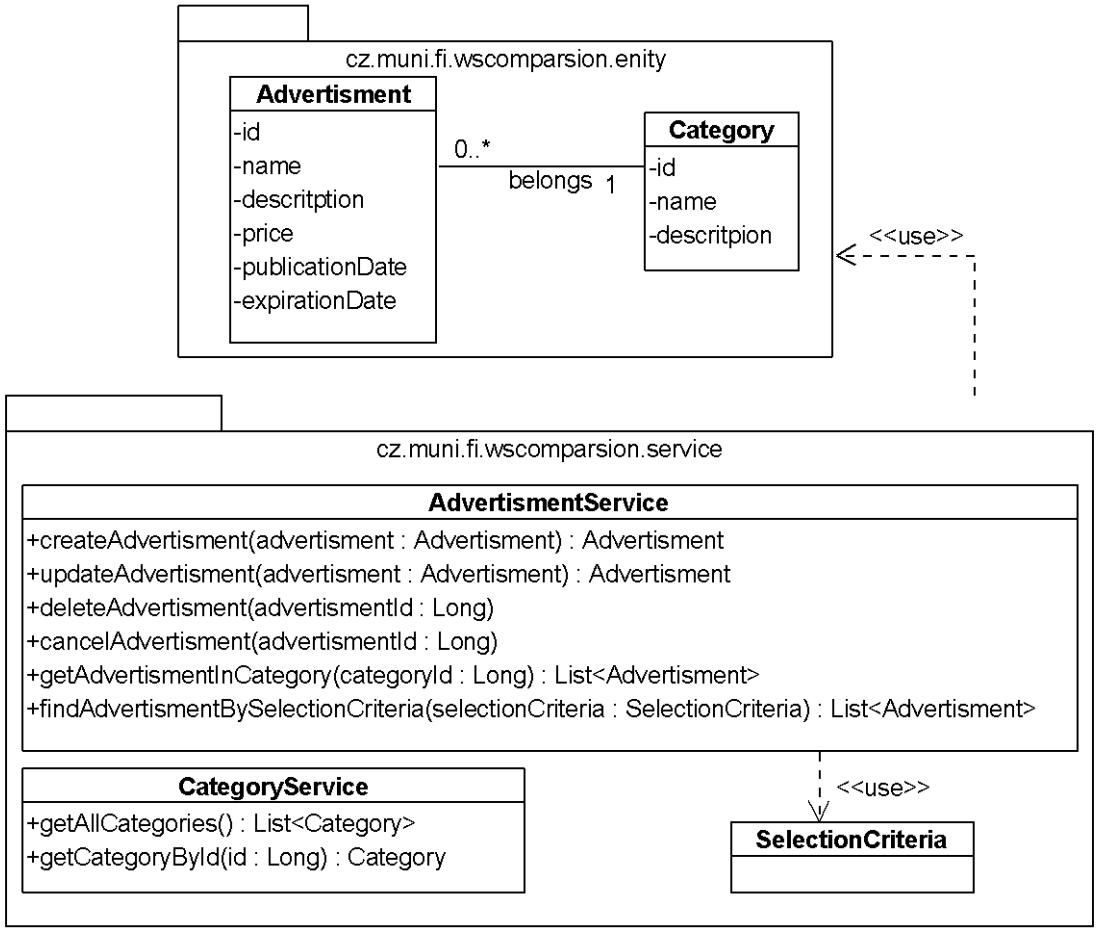
- registrace uživatelů a řešení uživatelských práv a přístupů (např. omezení, že pouze administrátor může smazat záznam);
- možnost platby online;
- více úrovní kategorií;
- editace kategorií.

5.1.1 Návrh řešení

V této sekci bych rád prezentoval návrh řešení demonstračního příkladu. Pro lepší názornost celkového srovnání jsem se rozhodl obě řešení postavit na stejné aplikační logice. Toto je také velmi častý způsob využití webových služeb. Funkce jsou již naprogramované a chceme je pouze zpřístupnit na Webu. Návrh aplikační logiky je uveden v první sekci. V dalších potom rozebírám návrh řešení pro SOAP a REST.

5.1.2 Návrh aplikační logiky

Z předchozí sekce víme, že potřebujeme pracovat s inzeráty a jejich kategoriemi. Následující diagram tříd ilustruje, jak by mohla vypadat aplikační logika, poskytující potřebné operace.

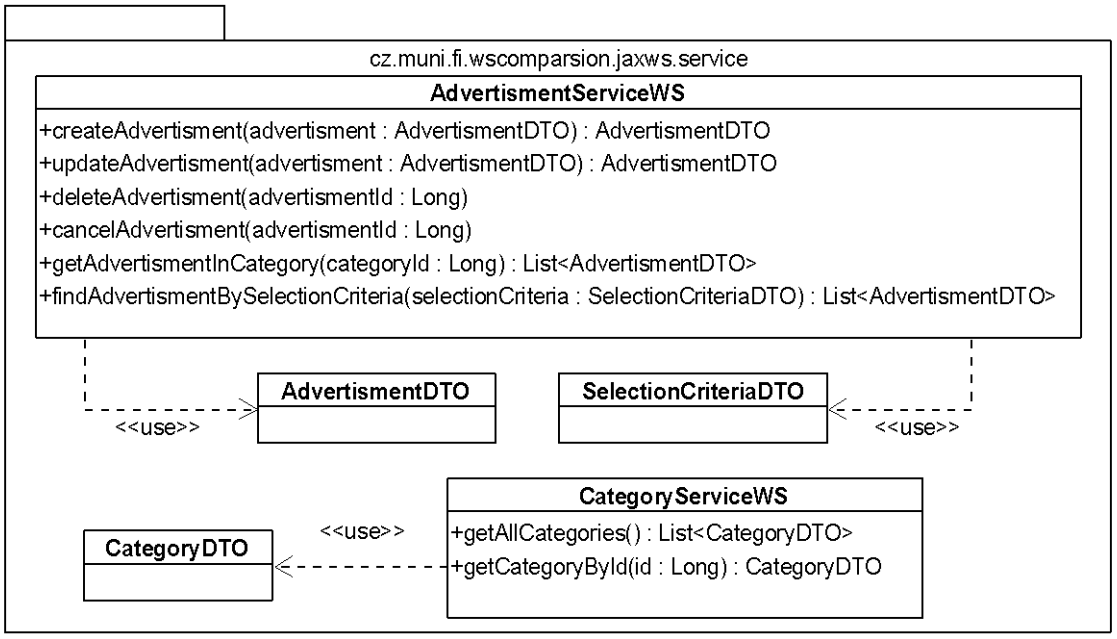


Obrázek 5.1.1: Aplikační logika inzertního systému

Z pohledu našeho příkladu není důležité podrobné pochopení diagramu, ale jeho hlavní myšlenky. Namodelovaná aplikační logika pomocí služby `AdvertismentService` poskytuje požadované operace nad inzerátem (vložení, vymazání, atd.). Služba `CategoryService` poskytuje operace pro získání informací o kategoriích. Tyto operace budou poskytovány klientům pomocí webových služeb. Kompletní popis jednotlivých tříd diagramu najdeme v příloze A.

5.1.3 Postup při návrhu SOAP služeb

Nyní bych rád zodpověděl otázku, jak vystavit výše představené funkce pomocí SOAP a JAX-WS? Postup je velmi přímočarý, stačí obalit existující služby aplikační logiky pomocí implementace koncových bodů (POJO označené `@WebService`). Návrh diagramu tříd tohoto řešení by byl v podstatě kopií služeb aplikační logiky. Je však velmi dobrou zvyklostí definovat na rozhraní přenosové objekty (DTO), které budou skrývat některé z atributů inzerátu nebo kategorie. Diagram tříd, který vznikl aplikací výše uvedených postupů, je znázorněn na obrázku 5.1.2.



Obrázek 5.1.2: Návrh služeb pro JAX-WS (SOAP)

Z obrázku vidíme, že jde skutečně o obalení stávajících služeb a jejich vstupních a návratových hodnot. Třídy s koncovkou DTO slouží jako obálka pro vstupní a výstupní hodnoty aplikační logiky. Koncovkou WS jsou potom označeny jednotlivé koncové body, které budou zprostředkovávat funkce poskytované aplikační logikou prostřednictvím JAX-WS. Vše ostatní nutné pro komunikaci s takovouto službou (převod do SOAP zpráv, vygenerování WSDL atd.) již zajistí JAX-WS.

5.1.4 Postup při návrhu pro REST – definování URI zdrojů

Jak ale stejné funkce vystavit pomocí REST? Zde je situace o trochu složitější. Z teorie REST víme, že pracujeme se zdroji a operacemi nad nimi, nikoliv přímo s metodami nějakých objektů. Z toho důvodu je nutné začít právě od zdrojů a jejich adres. Následující tabulka shrnuje identifikované zdroje, jejich URI a metody, které nad nimi lze vykonat:

URI	HTTP metody	Význam
`/advertisments/{id}`	GET	Detail konkrétního inzerátu.
	PUT	Úprava hodnot konkrétního inzerátu.
	DELETE	Vymazání konkrétního inzerátu.
`/advertisments`	GET	Seznam všech inzerátů.
	POST	Vložení nového inzerátu.
`/advertisments/search`	GET	Vyhledávání v inzerátech. Toto URI bude dále obohaceno o dotazovací parametry omezující vyhledávání. Např. `advertisments/search? categoryId=1`
`/category`	GET	Všechny poskytované kategorie.
`/category/{id}`	GET	Detail konkrétní kategorie.

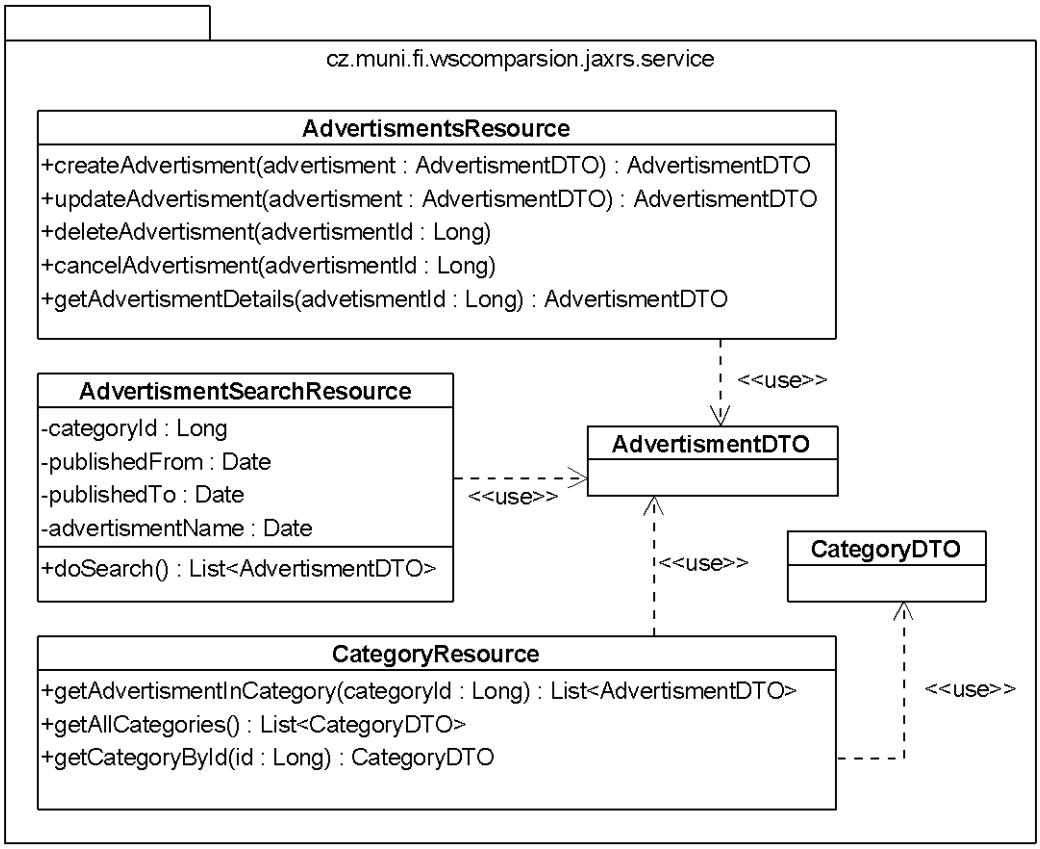
`/category/{id}/advertisments`	GET	Seznam inzerátů v dané kategorii

Tabulka 5.1.1: Identifikované URI zdrojů a aplikovatelné metody

Toto je nejdůležitější část návrhu REST. Je nutné si uvědomit smysl jednotlivých metod a vyvarovat se špatného použití filozofie REST, čímž může dojít k odklonění od čistého REST stylu a příklonu k REST-RPC hybrid stylu (probráno v sekci 2.5.1). Příkladem takového špatného návrhu by mohlo být implementování zrušení inzerátu pomocí DELETE HTTP metody a dotazovacího parametru (`/advertisments/{id}?cancel=true`)[2]. Takto definovaná operace neprovádí smazání zdroje (DELETE), ale spíš modifikaci dat zdroje (PUT).

Poté co si takto nadefinujeme zdroje, můžeme vytvořit diagram zdrojových tříd ve smyslu JAX-RS, které budou poskytovat operace nad výše identifikovanými zdroji.

Poslední krok je již velmi podobný implementaci SOAP a je jím definice objektů, pomocí kterých bude služba komunikovat s okolím. Opět by se dalo použít objektů aplikační logiky pro komunikaci se službou, avšak zde je vytvoření přenosových objektů téměř nutné, a to z důvodu jednodušší implementace zpracování požadavků a vytváření odpovědí. Tento princip si podrobněji vysvětlíme v sekci 5.2 při porovnávání implementací. Výsledný diagram tříd znázorňuje obrázek 5.1.3.



Obrázek 5.1.3: Návrh služeb pro JAX-RS (REST)

Můžeme si všimnout hlavního rozdílu v definici výběrových kritérií, kde je v případě SOAP použitý objekt SearchCriteriaDTO, na rozdíl od REST, kde předpokládáme zaslání výběrových kritérií pomocí dotazových parametrů v GET požadavku. Tyto parametry jsou v případě REST zpracovány pomocí vyhledávací služby `AdvertismentSerarchResource`. Ostatní rozdíly jsou z pohledu diagramu tříd zanedbatelné.

5.2 Porovnání implementace vybraných funkcí

V této sekci bych se rád zaměřil na srovnání z hlediska náročnosti na implementaci stejných funkcí implementovaných pomocí JAX-WS a JAX-RS. Konkrétně půjde o následující funkce:

- vložení inzerátu;
- vyhledávání inzerátů podle kritérií.

5.2.1 Vložení inzerátu

Z pohledu diagramu tříd je řešení této operace naprosto stejné (tj. implementace jedné metody a přenosového objektu), avšak diagram nezachycuje použité anotace jednoho či druhého API, které značně ovlivňují celé chování. Tuto situaci nám demonstrují příklady 5.2.1 a 5.2.2.

```
@WebService()
public class AdvertismentServiceWS {
...
@WebMethod(operationName = "createAdvertisment")
public AdvertismentDTO createAdvertisment(
          @WebParam(name = "advertisment")AdvertismentDTO advertisment)
           throws FinderException {
    Category category =
        categoryEjb.getCategoryById(advertisment.getCategoryId());
    Advertisment result =
            advertismentEjb.createAdvertisment(
    DTOConvertor.
       convertAdvertismentDtoToEntity(advertisment, category));
    return DTOConvertor.convertAdvertismentEntityToDto(result);
}…
```

Příklad 5.2.1: Vložení inzerátu JAX-WS (SOAP)

```
@Path("advertisments")
@Produces({MediaType.TEXT_XML, MediaType.APPLICATION_JSON})
public class AdvertismentsResource {
…
@POST
@Consumes({MediaType.TEXT_XML, MediaType.APPLICATION_JSON})
public AdvertismentDTO createAdvertisment(
        AdvertismentDTO advertisment) throws FinderException {
    try {
        Category category =
            categoryEjb.getCategoryById(advertisment.getCategoryId());
        Advertisment result =
                advertismentEjb.createAdvertisment(
        DTOConvertor.
           convertAdvertismentDtoToEntity(advertisment, category));
        return DTOConvertor.convertAdvertismentEntityToDto(result);
```

```
    } catch (FinderException finderException) {
        throw new NotFoundException(finderException);
    }
}…
```

Příklad 5.2.2: Vložení inzerátu JAX-RS (REST)

Složitost kódu a implementace

Na první pohled vypadají implementace velmi podobně, obě využívají pro implementaci služeb pouze anotované POJO objekty. Pokud opomineme složitější návrh v případě JAX-RS (návrh URI a HTTP metod), může se zdát, že třídy jsou, až na různé anotace, stejné.

Při detailnějším pohledu si můžeme všimnout `try-catch` bloku v případě JAX-RS a také rozdílných výjimek v deklaraci metody (`NotFoundException` vs. `FinderException`). V případě vyhledávání kategorie pro uložení inzerátu totiž může dojít k chybě při volání aplikační logiky. Tuto chybu signalizuje výjimka `FinderException`, která říká, že kategorie neexistuje.

V případě JAX-WS můžeme použít přímo výjimku z aplikační logiky (`FinderException`). Ta je obalena automaticky generovaným objektem (zaručuje JAX-WS). Tento objekt umožní převod výjimky do XML formátu a její odeslání v těle SOAP zprávy, kde je výjimka reprezentována chybovým elementem (*fault*).

Podobný princip by fungoval i v případě JAX-RS služby, pomocí výchozího reprezentanta `ExceptionMapper` (viz 3.7.2). Tj. byla by možná stejná deklarace výjimky jako v případě SOAP. Tento postup však způsobí navrácení chyby s HTTP kódem 500 *Internal server error*. V těle této zprávy bude sice popis výjimky, ale navrácení kódu 500 v případě neexistence zdroje znamená porušení principů REST, protože nepoužijeme správný návratový kód (nedodržíme kontrakt jednotného rozhraní). Správnou reakcí na tento typ chyby je navrácení zprávy s kódem 404 *Not Found*. Toto můžeme zajistit několika způsoby (viz 3.7.2). V příkladu je využitý způsob vytvoření nové výjimky `NotFoundException`, která rozšiřuje výchozí `WebApplicationException` a reprezentuje chybu nenalezení zdroje. V případě, že dojde k vyvolání `FinderException`, služba tuto výjimku zachytí a místo ní vyvolá výjimku `NotFoundException`.

```
public class NotFoundException extends WebApplicationException {
    public NotFoundException(Throwable cause) {
       super(Response.status(Response.Status.NOT_FOUND).
               entity(cause.getMessage()).build());
    }
}
```

Příklad 5.2.3: Výjimka reprezentující informaci o nenalezeném zdroji JAX-RS (REST)

Vidíme, že implementace je velmi triviální, tudíž ovlivní samotnou složitost velmi málo.

Implementace přenosových objektů je v obou řešeních naprosto stejná. Jedná se o objekty označené anotacemi JAXB. Tyto anotace určují, jakým způsobem bude objekt reprezentován

pomocí XML v SOAP zprávě, či těle HTTP požadavku. V případě JAX-WS není nutné objekty označovat anotacemi JAXB, protože JAX-WS podporuje jejich automatické vygenerování. To však způsobí vytváření redundantních obalujících objektů. V JAX-RS je nutné tyto objekty vytvořit ručně. Demonstrace takto označeného přenosového objektu je provedena v rámci následující sekce na příkladu 5.2.5.

Komunikace s aplikační logikou je ekvivalentní pro oba přístupy. Služby získají objekt `Category` podle `id` v objektu `AdvertismentDTO`. Následně pomocí konverzní funkce převedou `AdvertismentDTO` na objekt, se kterým umí aplikační logika pracovat. Zkonvertovaný objekt je následně zpracován aplikační logikou.

Další drobností, na kterou bych rád upozornil, je definice výstupu pomocí anotace `@Produces` v případě JAX-RS služby. Ta umožní odeslat klientovi kromě zmiňovaného XML také formát JSON. JAX-RS disponuje podporou automatické konverze objektů anotovaných dle zvyklostí JAXB do JSON formátu. Více o tomto bude probráno v sekci 5.3.

V případě JAX-WS ještě stojí za zmínku podpora ve vývojových nástrojích, která je podstatně lépe propracovaná než podpora pro JAX-RS. Kostru webové služby, která volá jednotlivé metody aplikační logiky lze přímo vygenerovat ze služby aplikační logiky. Toto může přispět k dalšímu urychlení a zjednodušení celého vývoje.

5.2.2 Vyhledávání inzerátů podle kritérií

Nyní se podíváme na implementaci jednoduchého vyhledávaní v inzerátech. Zde je odlišnost již patrná od základu, tj. od diagramu tříd, kde REST používá na tuto funkci speciální zdroj `AdvertismentSearchResource`. Zdroj by mohl být součástí třídy `AdvertismetnsResource` (viz obr. 5.1.3). Jeho oddělení jsem provedl záměrně z důvodu demonstrace principu vkládání závislostí v JAX-RS (viz sekce 3.3.1). Následující dva příklady (5.2.4 a 5.2.5) ukazují možnou implementaci vyhledávání pomocí JAX-WS. Řešení stejného problému v JAX-RS znázorňuje příklad 5.2.6.

```
@WebMethod(operationName = "findAdvertismentBySelectionCriteria")
public List<AdvertismentDTO>
     findAdvertismentBySelectionCriteria(@WebParam(name = "criteria")
                                  SearchCriteriaDto criteriaDto) {

    SearchCriteria criteria =
              DTOConvertor.
               convertSearchCriteriaDtoToSearchCriteria(criteriaDto);
    //call app logic and convert results same for both
…
}
```

Příklad 5.2.4: Implementace vyhledávání v inzerátech JAX-WS (SOAP)

```
@XmlRootElement(name = "category")
@XmlType(propOrder = {"publishedFrom", "publishedTo",
                      "advertismentName", "categoryId"})
public class SearchCriteriaDto {
    private Date publishedFrom;
    private Date publishedTo;
    private String advertismentName;
    private Long categoryId;
…//getters and setters
}
```

Příklad 5.2.5: Přenosový objekt (JAXB anotovaný) pro vyhledávání JAX-WS (SOAP)

Implementace vyhledávání pomocí JAX-WS je velice podobná postupu, jaký byl použit pro ukládání inzerátu v příkladu 5.2.1. Opět je zaslán objekt označený JAXB anotacemi (příklad 5.2.5 objekt `SearchCriteriaDto`), který obsahuje výběrová kritéria pro vyhledávání. Zaslaný objekt je dále transformován na objekt kompatibilní s vyhledávací metodou aplikační logiky (`SearchCriteria`) a odeslán na zpracování. Výsledek vyhledávání poskytnutý aplikační logikou je konvertován zpět do pole objektů označených JAXB anotacemi (konverzní funkce). Tento list objektů je automaticky transformován do výsledné SOAP zprávy. Jak ke stejnému problému přistoupit v JAX-RS? Na tuto otázku odpoví příklad 5.2.6.

```
@Produces({MediaType.TEXT_XML, MediaType.APPLICATION_JSON})
@Path("/advertisments/search")
public class AdvertismentSearchResource {

    @QueryParam(value = "categoryId")
    Long categoryId;

    @QueryParam(value = "dateFrom")
    Date publishedFrom;

    @QueryParam(value = "dateTo")
    Date publishedTo;

    @QueryParam(value = "advertismetnTitle")
    String advertismentName;

    @Inject
    AdvertismentService advertismentServiceEjb;

    @GET
    public List<AdvertismentDTO> serachForResults() {
        System.out.println("Test categoryId " + categoryId);
        SearchCriteria criteria =
               new SearchCriteria(publishedFrom, publishedTo,
                                  advertismentName, categoryId);
    //call app logic and convert results same for both
    }
}
```

Příklad 5.2.6: Implementace vyhledávání v inzerátech JAX-RS (REST)

Přístup JAX-RS je tentokrát poněkud odlišný. Namísto transportního objektu jsou použity dotazovací parametry (*QueryParams*), které jsou právě přiřazeny do jednotlivých parametrů

ve zdrojové třídě `AdvertismentSearchResource` pomocí principu vložení závislostí (viz sekce 3.3.1). To znamená, že data jsou službě odesílána v URL požadavku pomocí těchto parametrů nikoliv v těle požadavku, jak je tomu u JAX-WS. Výhody tohoto přístupu objasním v sekci 5.3. Samotné vyhledávání pak obstarává funkce `doSearch()`, která na rozdíl od konverze DTO (JAX-WS přístup) naplní objekt akceptovaný aplikační logikou přímo parametry získanými z dotazovacích parametrů v URL.

Úsilí vývojáře na vytvoření implementace pomocí jednoho či druhého přístupu se mi jeví jako velmi podobné. Rozdíl, zda parametry zaznamenám v objektu třídy, nebo v přenosovém objektu je téměř nulový. Jako menší úsilí navíc se může jevit vytvoření konverzní funkce pro DTO objekt, ale opět se nejedná o nic, co by výrazně zvýšilo náročnost implementace.

5.3 Data přenášená mezi klientem a webovou službou

Již víme, jaké úskalí s sebou nese implementace pomocí jednoho nebo druhého řešení. V této sekci se podíváme, jak je to s daty, která se přenáší při komunikaci s REST či SOAP službami. Zaměříme se na objem přenášených dat a také na pružnost návratového formátu při volání následujících funkcí:

- vložení inzerátu;
- detail konkrétního inzerátu.

5.3.1 Vložení inzerátu

V sekci 5.2.1 jsme si ukázali, jak by vypadaly implementace služeb poskytující funkce pro uložení v REST či v SOAP. Jak by vypadaly zprávy odesílané takovéto službě? Na to nám odpoví příklady 5.3.1 a 5.3.2.

```
POST kontext/AvertismentJAX-WS/AdvertismentServiceWSService HTTP/1.1
Content-type: text/xml

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://service.jaxws.wscomparsion.fi.muni.cz/">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:createAdvertisment>
         <advertisment>
            <name>Kolo Trek</name>
            <description>Krabonové kolo Trek Madone 5.2</description>
            <price>54000</price>
            <publicationDate>2010-06-10</publicationDate>
            <expirationDate>2010-06-15</expirationDate>
            <categoryId>1</categoryId>
         </advertisment>
      </ser:createAdvertisment>
   </soapenv:Body>
</soapenv:Envelope>
```

Příklad 5.3.1: Klientský požadavek na vložení inzerátu JAX-WS(SOAP)

```
POST kontext/AvertismentJAX-RS/resources/advertisments/ HTTP/1.1
Content-type: text/xml
Accept: text/xml

<advertisment>
      <name>Kolo Trek</name>
      <description>Krabonové kolo Trek Madone 5.2</description>
      <price>54000</price>
      <publicationDate>2010-06-10</publicationDate>
      <expirationDate>2010-06-15</expirationDate>
      <categoryId>1</categoryId>
</advertisment>
```

Příklad 5.3.2: Klientský požadavek na vložení inzerátu JAX-RS(REST)

Při vytváření objektu obě služby používají HTTP metodu POST, která odesílá data na předem definované URL (reprezentující službu/zdroj). Tento postup odpovídá i smyslu metody POST v rámci HTTP. Co je však velmi odlišné a zřejmé na první pohled, je tělo samotné zprávy. V případě REST odesíláme pouze objekt reprezentovaný pomocí XML a nic navíc. V případě SOAP zpráva obsahuje hlavičky, definice schémat a také definici metody, která se má nad daty provést. V případě REST probíhá výběr metody v Javě na základě použité HTTP metody (POST = vytvoř nový záznam) a URI služby. To nám ušetří mnoho jinak zbytečných dat přenášených ve zprávě. Extra data v SOAP jsou přenášena v rámci každé komunikace a způsobují nemalou režii. Tato režie je pak ještě zjevnější na následujících příkladech (sekce 5.3.2). Odpovědní zprávy by vypadaly velmi podobně, proto je v případě tohoto srovnání neuvádím, bude jim však věnován prostor v dalším příkladu.

5.3.2 Detail konkrétního inzerátu

Druhou funkcí, u které bych se rád věnoval přenášeným datům, je získání detailu konkrétního inzerátu. Zde uvidíme, že režie SOAP je v některých případech opravdu vysoká. Více již prozradí příklady 5.3.3 a 5.3.4.

```
POST kontext/AvertismentJAX-WS/AdvertismentServiceWSService HTTP/1.1
Content-type: text/xml

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://service.jaxws.wscomparsion.fi.muni.cz/">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getAdvertismentDetails>
         <id>1</id>
      </ser:getAdvertismentDetails>
   </soapenv:Body>
</soapenv:Envelope>
```

Příklad 5.3.3: Klientský požadavek získání inzerátu JAX-WS(SOAP)

```
GET kontext/AvertismentJAX-RS/resources/advertisments/1 HTTP/1.1
Accept: text/xml
```

Příklad 5.3.4: Klientský požadavek získání inzerátu JAX-RS(REST)

Zde vidíme, že rozdíl je opravdu markantní. REST zde používá pouze jednoduchého GET požadavku. Všimněme si, že `id` je uvedené v URL požadavku a tím splňuje i podmínku REST (jednoznačně identifikuje zdroj inzerátu s `id=1`). Podobně je tomu při použití GET pro získání reprezentace zdroje (dodržení významu GET). SOAP zde opět používá metody POST, přestože tentokrát žádá server o data. POST metodou posílá SOAP zprávu definující operaci získání detailu inzerátu (`getAdvertismentDetails`) a její parametr `id`. Takto vytváří zbytečná data, která zatěžují sít. Co je v dnešní době pár KB dat? Nesmíme však zapomenout na mobilní klienty (PDA, mobilní telefon), které mají stále omezenou přenosovou rychlost hlavně v případě menších měst, či slabého signálu. V takovémto případě je REST jednoznačný vítěz.

Nyní se pojďme podívat na odpovědi a možnosti jednotlivých implementací v tomto případě.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getAdvertismentDetailsResponse
xmlns:ns2="http://service.jaxws.wscomparsion.fi.muni.cz/">
         <return>
            <id>1</id>
            <name>Nazev 1</name>
            <description>nabidka supr kola</description>
            <price>10500.5</price>
        <publicationDate>2010-10-10T10:55:55+02:00</publicationDate>
         <expirationDate>2010-10-10T10:55:55+02:00</expirationDate>
            <categoryId>1</categoryId>
         </return>
      </ns2:getAdvertismentDetailsResponse>
   </S:Body>
</S:Envelope>
```

Příklad 5.3.5: Odpovědní zpráva získání inzerátu JAX-WS(SOAP)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<advertisment>
    <id>1</id>
    <name>Nazev 1</name>
    <description>nabidka supr kola</description>
    <price>10500.5</price>
    <publicationDate>2010-10-10T10:55:55+02:00</publicationDate>
    <expirationDate>2010-10-10T10:55:55+02:00</expirationDate>
    <categoryId>1</categoryId>
</advertisment>
```

Příklad 5.3.6: Odpovědní zpráva získání inzerátu JAX-RS(REST)

Zde již není rozdíl tak markantní, jako při samotném požadavku. Co je však zjevné, že v obou případech přenášíme malé množství dat, ale díky XML formátu se velikost celé zprávy více než zdvojnásobí. Podobně je tomu v případě odesílání požadavku na vytvoření inzerátu. XML má velkou spoustu výhod (validace, čitelnost), ale toto je jeden z jeho hlavních záporů. Jak tento problém řešit?

V případě SOAP nemáme příliš na výběr, protože výsledná zpráva musí být XML. Je sice možné použít JSON či jiný formát, ale neustále bude nutné výsledek uzavírat do obálek, které vyžaduje SOAP (minimálně: obálka, tělo, specifikace metody, parametr).

Co na to REST a JAX-RS? Již v sekci 5.2.1 jsem upozorňoval na anotaci `@Produces`. Ta nám umožní specifikovat výstupní formát zprávy. Další velmi důležitou funkcí je vestavěná podpora přímé konverze objektů označených anotacemi JAXB do formátu JSON. Když dáme tyto dvě funkce dohromady, dostáváme perfektní způsob řešení. Využijeme stejného objektu jako pro přenos XML a díky výše zmíněným funkcím JAX-RS bude možné získat odpověď ve formátu JSON. Požadavek na tento zjednodušený formát by vypadal takto:

```
GET kontext/AvertismentJAX-RS/resources/advertisments/1 HTTP/1.1
Accept: application/json
```

Příklad 5.3.7: Klientský požadavek získání inzerátu ve formátu JSON JAX-RS(REST)

Odpověď na tento požadavek by vypadala následovně.

```
{"id":"1", "name":"Nazev 1", "description":"nabidka supr ko-la",
"price":"10500.5","publicationDate":"2010-10-10T10:55:55+02:00",
"expirationDate":"2010-10-10T10:55:55+02:00","categoryId":"1"}
```

Příklad 5.3.8: Odpověď serveru ve formátu JSON JAX-RS(REST)

Když porovnáme tento výsledek s předchozím XML, dosahujeme opět lepších výsledků v oblasti objemu přenesených dat. Dalším nezanedbatelným přínosem je možnost zpracování tohoto formátu. Velká většina bohatých (*rich*) aplikací používá právě asynchronních volání služeb pomocí skriptovacího jazyka Javascript, který tento formát nativně podporuje a jeho zpracování je násobně rychlejší než v případě XML.

5.4 Klientské aplikace

Poslední oblastí, ve které srovnáme přístupy REST a SOAP je konzumace těchto služeb klientskými aplikacemi. Pro demonstraci jsem zvolil velmi primitivního klienta v Javě, který bude vypisovat detail inzerátu na standardní výstup. Následující příklady demonstrují komunikaci se službami JAX-WS a JAX-RS.

```
Long id = 1l;
try {//call service
    AdvertismentServiceWSService service =
                new AdvertismentServiceWSService();
    AdvertismentServiceWS port =
                service.getAdvertismentServiceWSPort();
    AdvertismentDTO dto = port.getAdvertismentDetails(id);
    //print result
    System.out.println("Inzerát: " + dto.getName() + "\n " +
    "Popisek: " + dto.getDescription() + " Cena: " + dto.getPrice());
} catch (FinderException_Exception ex) {
         log.fatal("Record with id: " + id + " wasn't found: "
        + ex.getMessage());
}
```

Příklad 5.4.1: Klientská aplikace vypisující detail inzerátu JAX-WS(SOAP)

```
Long id = 1l;
try {//call service
    Client restClient = new Client();
    WebResource resource =
            restClient.resource("http://localhost:8080/" +
            "AvertismentJAX-RS/resources/advertisments/");
    AdvertismentDTO dto =resource.path(id.toString())
            .accept(MediaType.APPLICATION_JSON)
            .get(AdvertismentDTO.class);
    //print result
    System.out.println("Inzerát: " + dto.getName() + "\n " +
            "Popisek: " + dto.getDescription() + "Cena: " +
dto.getPrice());
} catch (UniformInterfaceException uie) {
        log.fatal("Record with id: " + id + " wasn't found: "
                +uie.getResponse().getEntity(String.class));
}
```

Příklad 5.4.2: Klientská aplikace vypisující detail inzerátu JAX-RS(REST)

Při pohledu na klienta SOAP vidíme, že pracujeme přímo s Java objekty, které reprezentují rozhraní služby a poskytují její operace. Tyto objekty jsou vygenerovány pomocí příkazu `wsimport` z WSDL dokumentu služby. Díky nim je vývojář kompletně odstíněn od problému s definováním adresy služby a návratového formátu, na rozdíl o Jersey klienta, který je nucen pracovat přímo s URL služby.

Postup volání metod služby JAX-WS je naprosto intuitivní a jednoduchý. Z objektu služby (`AdvertismentServiceWSService`) získáme její port (`AdvertismentServiceWS`), který již poskytuje operace služby, podobně jako třídy z běžné Java knihovny.

V případě Jersey je nutné pracovat s adresou webové služby, specifikovat jaký druh dat klient požaduje a vybrat správnou HTTP metodu. I když Jersey klient poskytuje velmi pěkně zpracované API pro tyto úkony, práce se službou SOAP je jednoznačně intuitivnější, protože je v podstatě ekvivalentní volání metod nějaké knihovny. Podobně je tomu v případě zpracování výjimek, kde je SOAP přístup opět intuitivnější a pohodlnější pro Java vývojáře.

Výhodou REST klienta je velikost a jednoduchost. Díky tomu, že Jersey klient nevytváří (negeneruje) spoustu tříd, které reprezentují službu a používá při komunikaci pouze objekty z Jersey klienta a přenosové objekty z rozhraní služby, je klient celkově podstatně menší a jednodušší.

Další nezanedbatelnou výhodou SOAP je výborná podpora ve velkém množství vývojových nástrojů. Ty jsou schopny, kromě vygenerování objektů pro komunikaci se službou, poskytnout také vytvoření kostry volání zvolené operace. Toto opět ušetří a značně sníží celkový čas potřebný na vývoj takového klienta.

Poslední devizou SOAP je samotné WSDL, které umožní generování kódu na straně klienta, ale hlavně poskytne velmi dobrý a detailní přehled o tom, jaké operace služba poskytuje. REST je v tomto ohledu pořád o něco pozadu. Existuje sice formát nazývaný WADL (*Web Application Description Language*), který by měl být pro REST služby tím, co je WSDL pro služby SOAP.

Avšak zatím se tento formát příliš neprosadil a není spojen ani se specifikací JAX-RS a v tuto chvíli neposkytuje žádné výhody typu vytvoření kostry klienta jako je tomu v případě WSDL, proto jsem se jím ani více nezabýval v práci jako takové. V budoucnu by však mohl zaplnit mezeru, která je v současné době právě v popisu rozhraní REST služeb pomocí nějakého formálního dokumentu.

Celkově můžeme říci, že z pohledu úsilí na implementaci klienta, který pracuje se službou v Javě, SOAP jednoznačně vítězí. Jak je tomu ale v jiných jazycích? Co když potřebuji službu konzumovat například v jazyce Javascript?

Zde je situace zcela jiná. U JAX-WS služby musíme složitě vytvořit SOAP zprávu a pracovat s XML, což bývá v jazyce Javascript časově náročná operace. V případě REST si vystačíme s jednoduchou Javascript knihovnou, která poskytuje základní podporu asynchronního volání HTTP metod. Navíc jsme schopni pomocí hlavičky `Accept` požádat o JSON, který je pro práci s daty v Javascriptu mnohem efektivnější. V tomto případě nehodlám jít do většího detailu, z důvodu nutnosti vysvětlení principů jazyka Javascript, který je mimo rozsah této práce.

5.5 Kdo je tedy vítěz

V předcházejících sekcích jsem se snažil demonstrovat jednotlivé odlišnosti v postupech při vývoji a použití služeb postavených na protokolu SOAP a implementovaných pomocí JAX-WS se službami postavenými na REST stylu a implementovanými pomocí JAX-RS. Nyní bych rád zrekapituloval tyto výsledky a pokusil se vytýčit hlavní výhody a nevýhody prvního či druhého přístupu.

Návrh aplikace

Při návrhu aplikace byla zjevná převaha JAX-WS (SOAP), kde stačilo pouhé obalení současných služeb objekty s pomocnými anotacemi. Přístup byl naprosto intuitivní a při znalosti JAX-WS by se dal označit za triviální. V JAX-RS (REST) se bylo nutné zamyslet nad zdroji jejich URI a nad mapováním metod z objektového světa na HTTP metody. Pro vývojáře, znalého objektově orientovaného přístupu, by vytvoření tohoto mapování znamenalo podstatně vyšší časovou náročnost, než pouhé obalení objektu objekty jinými, jak je tomu v případě JAX-WS (SOAP).

Implementace služeb

Zde se díky velmi dobře navrženému standardu JAX-RS podařilo rozdíl téměř smazat. V některých případech bylo sice nutné přidat více kódu (implementace obalující výjimky příklad 5.2.3), ale celkově by se dala složitost implementace samotné označit jako srovnatelná. Zde bych rád vyzdvihl práci vývojových týmů jednoho či druhého standardu, protože standardy jsou z pohledu jednoduchosti použití na vynikající úrovni.

Odesílaná data

Z pohledu odesílaných dat je situace naprosto opačná než v případě návrhu. REST přístup je podstatně flexibilnější, není svázaný jedním XML formátem zprávy, jak je tomu v případě SOAP. Poskytuje výrazně větší pružnost v definici formátu přenášených dat a tím je schopen snížit objem přenášených dat na minimum. Podpora HTTP hlaviček umožňuje velmi jednoduchou

specifikaci požadovaného formátu. Díky JAX-RS a jeho principu mapování Java objektů na různé formáty odpovědi (viz sekce 3.7), je implementace podpory pro další formát zjednodušena na maximum.

Klientské aplikace

Zde je z pohledu Javy vítězem JAX-WS, který nabízí intuitivnější přístup bližší objektově orientovaným vývojářům než JAX-RS. Webové služby však slouží na propojení aplikací napříč programovacími jazyky a například v Javascriptu je situace zcela opačná. Bohužel na kvalifikované porovnání více druhů klientů nezbyl prostor.

Co se do srovnání nevešlo

Bohužel díky rozsahu diplomové práce nezbyl prostor na další body, které by ještě zvýšily vypovídající hodnotu celého srovnání. Mezi ně patří např.

- zabezpečení;
- konzumace služeb větším množství klientů (.NET, C++, Rubby, Java-FX);
- demonstrace využití vyrovnávací paměti a proxy.

Závěr

Co tedy říci závěrem? Když sečteme výhry v jednotlivých oblastech, tak ve dvou případech vítězí SOAP, v jednom REST a v jednom případě je vítěz nejednoznačný. Z tohoto pohledu by se stal vítězem srovnání SOAP.

Dle mého názoru však není vítězství SOAP tak jednoznačné. V případě klientské aplikace totiž nebyl prostor na porovnání více typů klientů, kde by mohla být situace odlišná (viz příklad Javascript). Ne všechny jazyky poskytnou tak dobrou podporu SOAP a bude nutné pracovat přímo s XML reprezentující SOAP zprávu, což vyžaduje znalost tohoto formátu. Znalost HTTP požadavků je v tomto případě podstatně pravděpodobnější. Převaha REST v případě přenášených dat a flexibility formátu odpovědi byla naprosto jednoznačná. V některých případech se jedná o jeden z nejdůležitějších faktorů, obzvlášť při velkém množství požadavků. Z tohoto důvodu se domnívám, že by nebylo správné prohlásit jednoznačným vítězem SOAP či REST. Místo toho bych rád zmínil, kde je vhodné každý z přístupů použít.

JAX-WS (SOAP)

Tento přístup bych využil ve velkých podnikových aplikacích, kde půjde s výhodou využívat XML formátu zpráv, který bude validován a kde bude nutná vysoká míra zabezpečení. Toho lze totiž velmi dobře dosáhnout využitím transakčních elementů v hlavičce SOAP zprávy a bude zamezeno posílání některých dat jako ID a podobně v parametrech URI. Dalším důvodem, proč bych doporučil SOAP pro toto použití, je jeho podpora jiných protokolů než HTTP (např. přenos SOAP zpráv pomocí JMS – *Java Message Service*). Posledním důvodem je samotné stáří a zaběhlost standardu. Jde o odzkoušenou věc a je pro ni velká podpora nástrojů vývojových prostředí i komerčních API.

JAX-RS (REST)

Osobně bych REST a JAX-RS doporučil na drtivou většinu aplikací, kde je důležité nabídnout jednoduché a flexibilní rozhraní pro přístup ke službám. Tuto flexibilitu ocení hlavně jednodušší klienti např. mobilních zařízení, kde je kompaktní formát naprostým základem. Stejně tak bych REST použil pro aplikace, které mohou být pod obrovskou zátěží (musí odpovídat na velké množství požadavků). Požadavky jsou totiž v případě REST jednodušší než v SOAP a díky HTTP hlavičkám je možné využívat vyrovnávací paměti, která ještě zvýší rychlost aplikace. REST API jsou velmi mladé a neustále se vyvíjí. Jejich podpora v komerčních nástrojích a běhových prostředích stoupá a domnívám se, že jde o velmi perspektivní způsob jak vyvíjet webové služby.

Závěr

Cílem této diplomové práce bylo seznámení čtenáře s architektonickým stylem REST a možností jeho aplikace na webové služby. Úvodní sekce práce čtenáře seznámily se základními technologiemi a postupy spojenými s webovými službami, čímž mu umožnily proniknout do této problematiky. Následně byl představen REST jako architektonický styl pro webové aplikace na příkladu WWW, který intuitivně nastínil hlavní myšlenky stylu REST. Dále byla vysvětlena formální omezení vymezující styl REST. Zde byl kladen důraz na klady a zápory, které s sebou tato omezení přináší.

Styl REST byl následně propojen se znalostmi o webových službách z úvodních sekcí, což vedlo k definování pojmu RESTful webových služeb (webové služby odpovídající stylu REST). Další velkou výzvou bylo srovnání této skupiny služeb se službami založenými na protokolu SOAP. Ukázalo se, že RESTful služby přináší oproti SOAP službám velké výhody ve flexibilitě formátu přenášených dat (není omezen XML formátem, jak je tomu v případě SOAP zpráv) a lepším využitím možností HTTP protokolu (využití kompletní sady HTTP metod a hlaviček). Tyto výhody jasně naznačily, že efektivita komunikace se službou by měla být jednou z hlavních domén služeb založených na REST. Velká podpora vývojových prostředí a výborný formát definice rozhraní služby (WSDL), to byly naopak výhody identifikované u služeb založených na SOAP.

Výsledky tohoto srovnání jsme chtěli ověřit v praxi. Z tohoto důvodu bylo nutné čtenáře uvést do problematiky implementace obou skupin webových služeb (SOAP a REST) v Javě. Tímto se nám odkryl poslední velký úkol, který si práce vytyčila, a tím bylo seznámení čtenáře se standardem JAX-RS (JSR-311), který představuje první standardní Java API (JSR standard) pro skupinu služeb REST. Jedná se o mladý standard (specifikace schválena v roce 2008) vycházející v kontextu Java EE 6 a při jeho zpracování se potvrdilo, že jde o standard plný velmi dobrých myšlenek a postupů. Používání anotovaných POJO objektů pro implementaci služby (již převzatá idea z dřívějších API), možnost mapování Java objektů na tělo HTTP požadavku, či práce s URI služby pomocí šablon jsou jen některé z těchto výborných myšlenek. Troufám si tvrdit, že tento standard přináší skutečně zlom v oblasti implementace služeb REST.

Avšak samotné představení jednoho standardu není dostatečné pro kompletní praktické srovnání, proto byl čtenář před samotným závěrečným příkladem seznámen s vybranými Java API pro vývoj webových služeb. Byli vybráni dva nejperspektivnější zástupci, jeden pro styl REST (Jersey referenční implementace JAX-RS) a druhý pro služby založené na SOAP (JAX-WS RI) protokolu.

Závěrečná část práce demonstrovala rozdíly v přístupech SOAP a REST právě pomocí výše vybraných reprezentantů na příkladu inzertního systému. Ukázalo se, že princip SOAP a vzdálené volání procedur je jednodušší a intuitivnější z hlediska návrhu. Druhým bodem srovnání byla obtížnost implementace, kde vyšlo najevo, že návrháři standardů odvedli vynikající práci a v obou případech byla implementace přímočará, ovšem díky lepší podpoře nástrojů byl SOAP blíže k vítězství. Další oblastí srovnání byl přenos zpráv mezi klientem a službou. Jednoznačně

se potvrdily poznatky z teorie, které již dříve favorizovaly REST (lepší flexibilita formátu a menší objem přenášených dat). V poslední sekci srovnání byl zhodnocen přístup k obtížnosti konzumace služby (komunikace se službou z pohledu klienta). SOAP opět dominoval svojí podporou ve vývojových nástrojích, což ještě podtrhlo jednoduchost jeho použití v Javě.

Podařilo se ukázat silné a slabé stránky REST a SOAP přístupů. Nedá se jednoznačně říci, který z nich je lepší. Existují aplikace, které využijí předností REST (např. mobilní aplikace, aplikace obsluhující velký počet požadavků), stejně tak existují případy, kdy je vhodnější zvolit SOAP (např. podnikové systémy, kde je zanedbatelná režie XML).

Protože rozsah diplomové práce je omezen, nezbyl prostor pro srovnání další zajímavých aspektů, jako je řešení bezpečnosti, či komunikace s různými typy klientů. Srovnání komplexnějších systémů implementovaných pomocí SOAP či REST vidím jako velmi zajímavé pokračování této práce. Posledním tipem na rozšíření práce by bylo porovnání JAX-RS s jinými přístupy k vývoji REST služeb např. rámec Restlet.

Literatura

[1] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[2] RICHARDSON, Leonard; RUBY, Sam. *RESTful Web Services*. First Edition. Sebastopol : O'Reilly Media, Inc., 2007. 409 s. ISBN 0-596-52926-0.

[3] FIELDING, Roy Thomas, et al. *W3C* [online]. 1999 [cit. 2010-05-20]. Hypertext Transfer Protocol -- HTTP/1.1. Dostupné z WWW: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.

[4] BOOTH, David, et al. *W3C* [online]. 11. 2. 2004 [cit. 2010-05-20]. Web Services Architecture. Dostupné z WWW: <http://www.w3.org/TR/ws-arch/>.

[5] GUDGIN, Martin, et al. *W3C* [online]. 27. 4. 2007 [cit. 2010-05-20]. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Dostupné z WWW: <http://www.w3.org/TR/soap12-part1/#intro>.

[6] CLEMENTS, Tom. *Oracle Sun Developer Network (SDN)* [online]. leden 2002 [cit. 2010-05-20]. Overview of SOAP. Dostupné z WWW: <http://java.sun.com/developer/technicalArticles/xml/webservices/>.

[7] *W3schools.com* [online]. 2005 [cit. 2010-05-20]. SOAP Tutorial. Dostupné z WWW: <http://www.w3schools.com/soap/default.asp>.

[8] CHINNICI, Roberto, et al. *W3C* [online]. 26. 6. 2007 [cit. 2010-05-20]. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Dostupné z WWW: <http://www.w3.org/TR/wsdl20/>.

[9] Sun microsystems. *Java™ Servlet Specification : Version 3.0* [online]. Final. California : Sun Microsystems, Inc., 10. 12. 2009 [cit. 2010-05-20]. Dostupné z WWW: <http://cds-esd.sun.com/ESD5/JSCDL/servlet/3.0-final/servlet-3_0-final-spec.pdf?AuthParam=1274379743_c6875b738a2d2faf8d1f13f7fcc40bbb&TicketId=CJ0ihZ7AljSBxJDsVooIBAE%3D&GroupName=CDS&FilePath=/ESD5/JSCDL/servlet/3.0-final/servlet-3_0-final-spec.pdf&File=servlet-3_0-final-spec.pdf>.

[10] Sun microsystems. *Java Servlet Specification: Version 2.5* [online]. Maintenance Release 2. California : Sun Microsystems, Inc., 8. 8. 2007 [cit. 2010-05-20]. Dostupné z WWW: <http://cds-esd.sun.com/ESD34/JSCDL/servlet/2.5-mrel2/servlet-2_5-mrel2-spec.pdf?AuthParam=1274379568_9bb4a405bce449210d2ed885e1460933&TicketId=nodzB10UQHd7luUpnEafUZibcw%3D%3D&GroupName=CDS&FilePath=/ESD34/JSCDL/servlet/2.5-mrel2/servlet-2_5-mrel2-spec.pdf&File=servlet-2_5-mrel2-spec.pdf>.

[11] Sun microsystems. *JAX-RS: Java™ API for RESTful Web Services* [online]. Version 1.0. Montbonnot Saint Martin : Sun Microsystems, Inc., 8. 9. 2008 [cit. 2010-05-20]. Dostupné z WWW: <http://cds-esd.sun.com/ESD5/JSCDL/jaxrs/1.0-fr/jaxrs-1.0-final-spec.pdf?AuthParam=1274379406_cda7649ad6fb34896327d8a049cd331f&TicketId=B%2Fw4lhiBRFpPShBCPF5dkgDn&GroupName=CDS&FilePath=/ESD5/JSCDL/jaxrs/1.0-fr/jaxrs-1.0-final-spec.pdf&File=jaxrs-1.0-final-spec.pdf>.

[12] Sun microsystems. *The Java EE 6Tutorial,Volume I* [online]. Santa Clara : SunMicrosystems, Inc., prosinec 2009 [cit. 2010-05-20]. Dostupné z WWW: <http://java.sun.com/javaee/6/docs/tutorial/doc/JavaEETutorial.pdf>.

[13] Oracle. *Oracle Wikis Home* [online]. V. 65. 21. 4. 2009 [cit. 2010-05-20]. Overview of JAX-RS 1.0 Features. Dostupné z WWW: <http://wikis.sun.com/display/Jersey/Overview+of+JAX-RS+1.0+Features>.

[14] Sun microsystems. *Java.net* [online]. 2009 [cit. 2010-05-24]. Jersey 1.2 User Guide. Dostupné z WWW: <http://svn-mirror.glassfish.org/jersey/trunk/www/documentation/latest/user-guide.html>.

[15] ARMSTRONG, Eric, et al. *The J2EE™ 1.4 Tutorial* [online]. California : Sun Microsystems, Inc., 7. 12. 2005 [cit. 2010-05-20]. Building Web Services with JAX-RPC, s. Dostupné z WWW: <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/J2EETutorial.pdf>.

[16] ORT, Ed; MEHTA, Bhakti . *Oracle Sun Developer Network (SDN)* [online]. květen 2003 [cit. 2010-05-20]. Java Architecture for XML Binding (JAXB). Dostupné z WWW: <http://java.sun.com/developer/technicalArticles/WebServices/jaxb/>.

[17] BUTEK, Russell. *IBM developerWorks* [online]. 24. 5. 2005 [cit. 2010-05-20]. Which style of WSDL should I use?. Dostupné z WWW: <http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/#>.

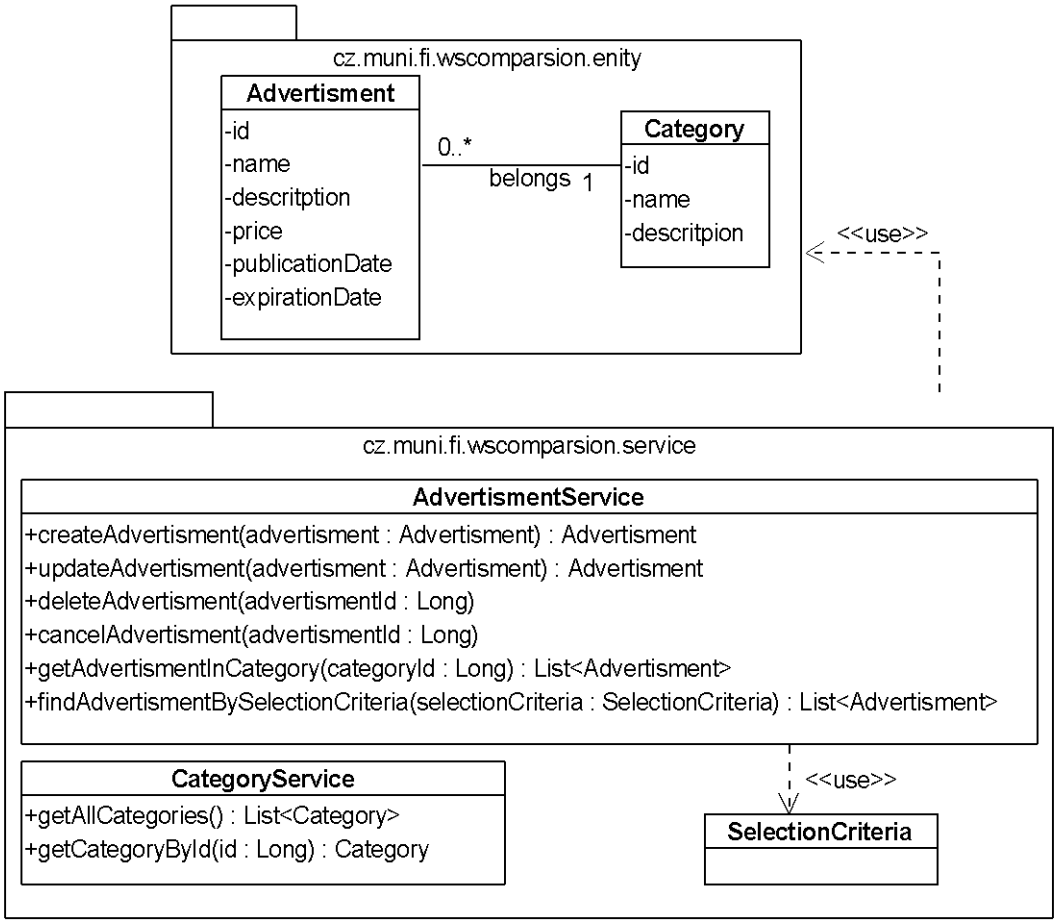
[18] YATES, John. *DevX.com* [online]. 1. 2. 2006 [cit. 2010-05-20]. JAX-RPC Evolves into Simpler, More Powerful JAX-WS 2.0. Dostupné z WWW: <http://www.devx.com/Java/Article/30459>.

[19] PODLEŠÁK, Jakub. *ENTERPRISE TECH TIPS* [online]. 26. 2. 2009 [cit. 2010-05-20]. Consuming RESTful Web Services With the Jersey Client API. Dostupné z WWW: <http://blogs.sun.com/enterprisetechtips/entry/consuming_restful_web_services_with>.

[20] HEWITT, Eben. *Java SOA Cookbook*. First Edition. Sebastopol : O'Reilly Media, Inc., 2009. 661 s. ISBN 978-0-596-52072-4.

Příloha A – diagram tříd

Pro samotné pochopení srovnání přístupů SOAP a REST nebylo důležité detailní pochopení diagramu tříd, avšak pro úplnost této práce a navržené aplikace bych rád detailně popsal význam jednotlivých tříd. Jako první si popíšeme diagram tříd aplikační logiky. Pro názornost uvádím opět jeho grafické ztvárnění následované popisem.



Obrázek A.1: Aplikační logika inzertního systému

Balík `cz.muni.fi.wscomparsion.entity`

Obsahuje třídy, které jsou využívané pro ukládání dat do databáze pomocí objektově relačního mapování. Tyto třídy jsou objektovou reprezentací tabulek databáze. Při jejich popisu bude kladen důraz na jejich význam z pohledu databáze.

Advertisment

Objektem typu (#Advertisment) je každá reprezentace inzerátu evidovaná v rámci systému. Např. Prodám kolo Trek, Popisek: Karbonové kolo Trek Madone 5.2. Cena: 54500Kč.

Category

Objektem typu (#Category) je každé označení skupiny předmětů prodávaných prostřednictvím inzertního systému. Mohou to být například: Zimní sporty, Nábytek, Letní sporty.

Vazba belongs

Inzerát (#Advertsiment), který patří do určité kategorie (#Category) 0,M :1,1.

Balík `cz.muni.fi.wscomparsion.service`

Tento balík obsahuje objekty reprezentující služby poskytované aplikační logikou. Jedná se o následující objekty:

Třída `AdvertismentService`

Reprezentuje službu poskytující operace nad inzerátem (vytvoření, vymazání, zrušení, vyhledávání, …) vyšším vrstvám (webovým službám). Přímo využívá knihovny pro ukládání dat do databáze pomocí objektů v balíku entity.

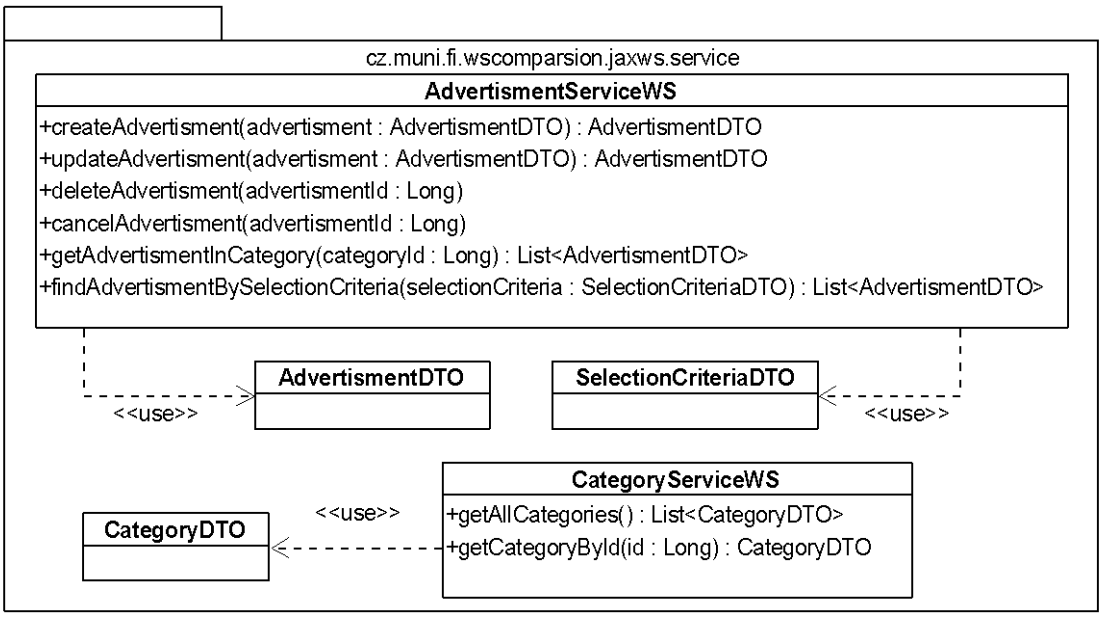
Třída `CategoryService`

Reprezentuje službu poskytující operace pro získávání dat o kategoriích (seznam kategorií a detail konkrétní kategorie). Třída je opět poskytovatelem těchto funkcí pro vyšší vrstvy.

Třída `SelectionCriteria`

Třída reprezentuje výběrová kritéria, podle kterých je aplikační logika schopna vyhledávat v inzerátech obsažených v databázi.

Další představený diagram reprezentoval třídy spojené s implementací webových služeb postavených na SOAP protokolu.



Obrázek A.2: Návrh služeb pro JAX-WS (SOAP)

Balík `cz.muni.fi.wscomparsion.jaxws.service`

Tento balík obsahuje třídy, které umožňují zprostředkování funkcí aplikační logiky pomocí webových služeb postavených na protokolu SOAP (JAX-WS).

Třída `AdvertismentServiceWS`

Tato třída reprezentuje webovou službu, která vystavuje funkce aplikační logiky spojené s operacemi nad inzerátem jako je: vložení, smazání, zrušení, úprava, získání detailu a vyhledávání. K tomuto účelu využívá třídy aplikační logiky `AdvertismentService` a třídu přenosového objektu `AdvertismentDTO`.

Třída `CategoryServiceWS`

Tato třída je webovou službou vystavující operace pro získání kategorií, k čemuž využívá třídy aplikační logiky `CategoryService` a třídy přenosového objektu `CategoryDTO`.

Třída `AdvertismentDTO`

Přenosový objekt reprezentující inzerát. Obsahuje všechny atributy inzerátu kromě vazby na kategorii. Místo této vazby je v objektu uloženo `Id` reprezentující kategorii. Tento objekt je použitý pro odesílání dat klientovi služeb.

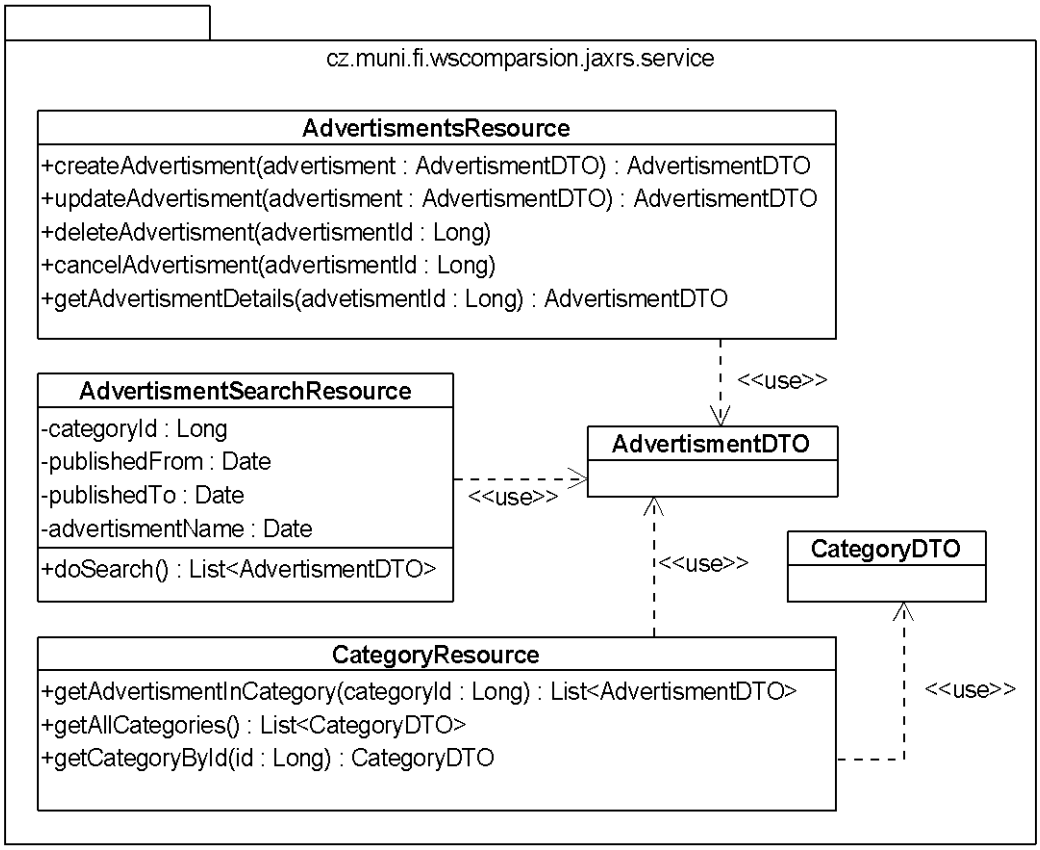
Třída `SelectionCriteriaDTO`

Třída slouží jako kontejner pro informace o zvolených vyhledávacích kriteriích. Obsahuje všechny atributy jako objekt aplikační logiky `SelectionCriteria`. Klient používá třídu pro odeslání výběrových kritérií službě.

Třída `CategoryDTO`

Jedná se o kontejner pro přenášení dat o kategorii ke klientovi. Nemá žádnou vazbu na inzeráty, které do této kategorie patří.

Posledním představeným diagramem byl diagram reprezentující návrh webových služeb pracujících na principu REST.



Obrázek A.3: Návrh služeb pro JAX-RS (REST)

Balík `cz.muni.fi.wscomparsion.jaxrs.service`

Tento balík obsahuje třídy, které umožňují zprostředkování funkcí aplikační logiky pomocí webových služeb REST (JAX-WS).

Třída AdvertismentResource

Třída reprezentující zdroj poskytující operace nad inzerátem (uložení, zrušení, vymazání, úprava, zobrazení detailu, nikoliv vyhledání!). Využívá třídy aplikační logiky `AdvertismentService` a třídy přenosového objektu `AdvertismentDTO`.

Třída AdvertsimentSearchResource

Třída poskytující vyhledávání inzerátů pomocí dotazovacích parametrů použitých v URI zdroje. Využívá třídu `AdvertismentService` a třídu přenosového objektu `AdvertismentDTO`.

Třída CategoryResource

Třída reprezentující zdroj poskytující informace o kategoriích inzerátů (seznam kategorií, detail kategorie, inzeráty patřící do dané kategorie). Využívá třídu `CategoryService` a třídu přenosového objektu `CategoryDTO`.

Třídy AdvertismentDTO, CategoryDTO

Naprosto stejné jako v případě balíku `cz.muni.fi.wscomparsion.jaxws.service`.

Příloha B – obsah CD

Následující seznam popisuje obsah jednotlivých složek na přiloženém CD:

- `obrazky` – obsahuje diagramy a jiné obrázky použité v práci (v plném rozlišení);
- `zdrojove_kody` – obsahuje zdrojové kódy srovnávacího příkladu z kapitoly 5. Všechny zdrojové kódy byly vytvořeny v nástroji Netbeans 6.8, mají formu projektů specifických pro tento vývojový nástroj a jsou dále rozděleny do následujících složek:
  - `AdvertismentAPPLogic` – projekt obsahující aplikační logiku použitou v obou poskytovatelích služeb (JAX-WS SOAP, JAX-RS REST). Postaveno na EJB 3.1, tudíž vyžaduje běhové prostředí Java EE6.
  - `AvertismentJAX-RS` – implementace poskytovatele funkcí nad inzeráty pomocí služeb REST (JAX-RS).
  - `AvertismentJAX-WS` – implementace poskytovatele funkcí nad inzeráty pomocí služeb SOAP (JAX-WS).
  - `JAX-Client` – ukázka volání služeb REST, SOAP z Javy (Java SE).
- `143417_kadlec_jiri_diplomova_prace.pdf` – text práce v elektronické podobě.