Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Automatizovaná rekonstrukce rozhraní webových služeb reverzním inženýrstvím

Plzeň 2018 Gabriela Hessová

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracovala samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. května 2018

Gabriela Hessová

Poděkování

Ráda bych poděkovala doc. Ing. Přemyslu Bradovi, MSc. Ph.D. za vstřícnost a cenné rady, které mi pomohly tuto diplomovou práci vypracovat.

Abstract

In the field of web services, REST services are currently very frequently used. The goal of this master thesis is to reconstruct REST service interface from an archive implemented on the Java platform in order to verify compatibility of components such as server compatibility with its clients. The theoretical part deals with the definition of what the REST service interface is and with the reasearch of existing formats for its description. Following is an analysis of the most used technologies for constructing the REST service interface in Java. In the practical part, an algorithm for its reconstruction is designed and implemented, which is integrated in a form of a plugin to the Component Repository supporting Compatibility Evaluation (CRCE) developed at the Department of Computer Science and Engineering at the University of West Bohemia. The plugin output is a REST service interface which is saved to a database in the form of metadata for the possibility of performing further analyses.

Abstrakt

V oblasti webových služeb patří v současnosti k nejpoužívanějším služby typu REST. Cílem této práce je rekonstrukce rozhraní REST služeb z archivu implementovaného pomocí platformy Java za účelem ověření kompatibility komponent, např. kompatibility serveru vzhledem k jeho klientům. Teoretická část se zabývá definicí toho, co to vlastně je rozhraní REST služby, a průzkumem existujících formátů pro jeho zachycení. Následuje analýza nejpoužívanějších technologií pro konstrukci REST služeb v Javě. V praktické části je navržen a implementován algoritmus pro rekonstrukci jejich rozhraní, který je integrován jako rozšíření do úložiště Component Repository supporting Compatibility Evaluation (CRCE) vyvíjeného na Katedře informatiky a výpočetní techniky na Západočeské univerzitě. Výstupem rozšíření je rozhraní REST služby, které je ve formě metadat uloženo do databáze pro možnost provádění dalších analýz.

Obsah

1 Úvod

Mezi webovými službami zaujímají významný podíl služby s REST architekturou. Tyto služby jsou postaveny na principech webu a měly by být samopopisné. REST pro své fungování nevyžaduje žádný veřejný dokument specifikující rozhraní, jako tomu například je u dříve zavedených služeb postavených nad protokolem SOAP s rozhraním definovaným veřejně dostupným WSDL dokumentem.

V praxi se však ukázalo, že strojově zpracovatelná reprezentace rozhraní je potřebná i v případě REST služeb, např. pro zjišťování kompatibility webové služby ve vztahu k jejím klientům. Z toho důvodu vznikla potřeba zpětné rekonstrukce rozhraní z existujícího programu. Hlavním cílem této práce je rekonstrukce rozhraní REST služby z Java bytecode za účelem provádění analýz určených k ověřování kompatibility komponent.

V první kapitole je představen koncept webových služeb typu REST, princip konstrukce jejich rozhraní a možné způsoby jeho dokumentace. Je zde představen Richardsonův model pro určování míry „*RESTovosti*“ webové služby a uveden přehled popisných jazyků pro REST. V poslední sekci jsou pak představeny formáty reprezentace strukturovaného těla zpráv.

Druhá kapitola pojednává o implementaci REST služeb v Javě. Jsou zde představeny frameworky implementující standard JAX-RS, dále pak Spring Web MVC a Restlet. V druhé části kapitoly jsou uvedeny technologie pro konverzi formátů pro strukturovaný text do Java objektů a zpátky.

Třetí kapitola čtenáře stručně seznámí s komponentovým úložištěm CRCE se zaměřením na jeho datový model a nástrojem JaCC určeným pro analýzu Java bytecode.

V dalších kapitolách je představen návrh úpravy datového modelu CRCE pro reprezentaci rozhraní REST služeb a algoritmu pro jeho rekonstrukci. Následuje popis implementace rozšíření CRCE, které za využití uvedeného algoritmu provede rekonstrukci rozhraní a uloží ho v podobě metadat do databáze.

V poslední kapitole jsou uvedeny výstupy programu po aplikaci na testovací archivy obsahující implementace REST služeb.

2 Webové služby a reprezentace REST rozhraní

V této kapitole jsou představeny webové služby jakožto nástroj integrace aplikací, s důrazem na služby s architekturou REST. Je zde definováno REST rozhraní a uvedeny nejčastější způsoby jeho dokumentace. V poslední části jsou uvedeny formáty zpráv vyměňovaných v rámci REST služeb.

2.1 Integrace aplikací

Tvorba větších programových celků zpravidla vyžaduje, aby aplikace netvořila monolitický celek, ale kombinovala možnosti několika částí – komponent. Tyto komponenty spolu komunikují přes programové rozhraní.

> Programové rozhraní, API (Application Programming Interface), je sada procedur, funkcí či tříd programové komponenty, které mohou být využívány programátorem jiné komponenty. [DIKR]

Z pohledu architektury aplikací můžeme mluvit o třech úrovních integrace:

- **Prezentační**. Integrace aplikací s využitím jednotného uživatelského rozhraní.
- **Datová**. Integrace dat pocházejících z různých zdrojů.
- **Aplikační**. Integrace aplikační logiky, jedním z jejích nástrojů jsou webové služby.

Jedním z přístupů k integraci aplikací na úrovni služeb je využití *servisně orientované architektury* (SOA).

> Servisně orientovaná architektura je distribuovaná architektura, ve které komponenty komunikují většinou vzdáleně prostřednictvím některého z protokolů pro vzdálený přístup a navzájem využívají části své funkcionality nazývané služby. [Ric16]

2.2 Webové služby

Neexistuje jedna všeobecně platná definice webové služby. V obecném kontextu se dá říci, že webová služba je služba poskytovaná jedním zařízením jinému zařízení, přičemž tato zařízení komunikují přes web. Zařízení, které službu poskytuje, se nazývá *server*, a zařízení, které ji využívá, se nazývá *klient*. Klientem webové služby zpravidla není přímo koncový uživatel, nýbrž aplikace, která získá od webové služby data a ta následně prezentuje uživateli.

Webové standardy jsou vytvářeny a spravovány mezinárodním konsorciem W3C (*World Wide Web Consortium*), jež definuje pojem webová služba následovně:

> Webová služba je softwarový systém určený pro podporu mezistrojové interakce v rámci internetu. Rozhraní webové služby je popsáno ve strojově zpracovatelném formátu (konkrétně WSDL[1]). Jiné systémy s webovou službou interagují s využitím zpráv specifikovaných protokolem SOAP[2], typicky postaveným na protokolu HTTP[3] a serializaci XML ve spojení s dalšími webovými standardy. [W3C04]

Uvedené definici vyhovuje např. koncept vzdáleného volání procedur RPC (*Remote Procedure Call*) a jeho objektový ekvivalent RMI (*Remote Method Invocation*).

V současné době se rozšířil nový koncept webových služeb, který definici od W3C neodpovídá. Jedná se o webové služby založené na architektonickém stylu REST (*REpresentational State Transfer*). Z definice se vymykají z následujících důvodů:

- Nepoužívají komunikační protokol SOAP.
- Nejsou svázány s WSDL, dokonce neexistuje obecný standardizovaný formát pro popis jejich rozhraní.
- Formát zpráv není omezen na XML.

2.3 Koncept REST

Koncept REST představil v roce 2000 Roy Fielding ve své disertační práci *Architectural Styles and the Design of Network-based Software Architectures* [FT00].

[1] Web Services Description Language - jazyk pro popis webových služeb, typicky svázaných s protokolem SOAP

[2] Simple Object Access Protocol - protokol pro výměnu zpráv mezi webovými službami

[3] Hypertext Transfer Protocol - protokol pro výměnu hypertextových dokumentů

[4] eXtensible Markup Language - značkovací jazyk

Motivací pro vznik REST byla snaha o vytvoření modelu správného chování webu. Jeho autor Roy Fielding je jedním z autorů specifikace protokolu HTTP, který představuje jeden ze základních pilířů webu.

Architektonický styl je množina architektonických omezení pojmenovaná za účelem snadného odkazování. Tato omezení určují role a vlastnosti architektonických prvků a vztahů mezi nimi.

V souvislosti s pojmem REST mluvíme o architektuře orientované na zdroje (ROA - *Resource Oriented Architecture*). Ačkoliv tedy REST řadíme mezi webové služby, které jsou nástrojem integrace na aplikační úrovni, můžeme v jeho případě mluvit zároveň o integraci na úrovni dat.

Zdroj je základní abstrakcí informace v RESTu. Každá informace, kterou můžeme pojmenovat, může představovat zdroj: dokument, obrázek, služba informující o dnešním počasí, množina jiných zdrojů atd. Jinými slovy každý koncept, jenž může být cílem hypertextového odkazu, musí odpovídat definici zdroje.

Definice pojmu REST je velmi abstraktní a nechává mnoho volného prostoru pro implementaci, proto Leonard Richardson shrnul principy RESTu ve své publikaci [RR07]. Zde je uvedena stručná, ale neúplná definice RESTu v Richardsonově pojetí, přičemž její jednotlivé části jsou vysvětleny v následující sekci:

REST je množina omezení vztahujících se na způsob manipulace se zdroji. Každý zdroj je identifikován názvem, popř. více názvy, přičemž každý z nich by měl mít smysl. Klient nemá ke zdroji přímý přístup, přistupuje pouze k jeho reprezentaci zprostředkované webovou službou. Přístup je definován uniformním rozhraním sestávajícím z konečné množiny metod.

2.3.1 Richardson Maturity Model

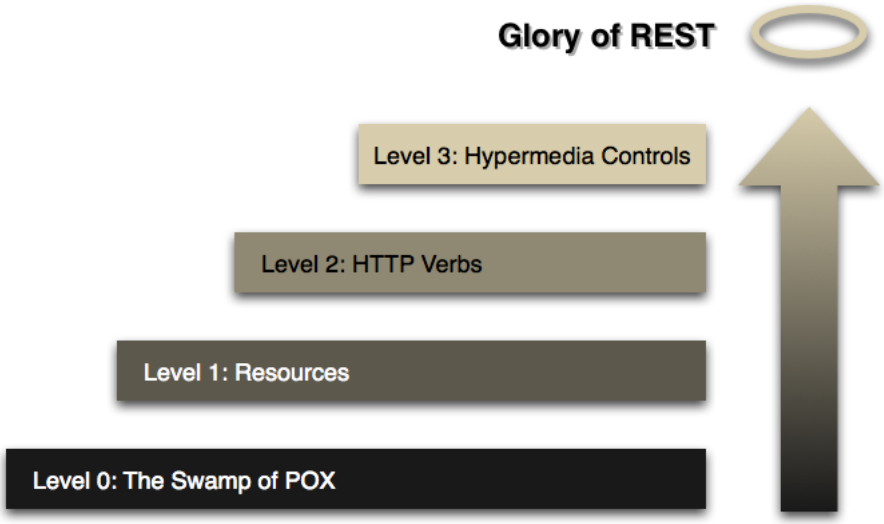
To, do jaké míry webová služba odpovídá konvencím pro REST služby, ilustruje model vytvořený Leonardem Richardsonem, tzv. *Richardson Maturity Model*.

Model je popsán v článku Martina Fowlera [Fow10], který je zdrojem informací pro tuto sekci. Pokud služba splňuje konvence odpovídající všem úrovním uvedeným níže, říkáme o ní, že je *plně RESTová* (angl. *RESTful*).

0. úroveň

Základem REST služby je využívání protokolu HTTP jako prostředku pro vzdálené interakce založené na vzdáleném volání procedur. REST jakožto architektonický styl technicky vzato není svázán s žádným protokolem, bavíme-li se ale o RESTových *webových*

službách, můžeme za komunikační protokol vždy označovat HTTP. Tělo požadavků a odpovědí má strukturu definovanou daným formátem, dříve nejčastěji používaným XML[4]. Tento princip je na obrázku označen jako 0. úroveň, protože je společný pro všechny webové služby.



Obrázek 2.1: Richardson Maturity Model

Požadavky jsou typicky směřovány na jeden vstupní bod webové služby (endpoint). Mohou vypadat následovně:

```
POST /webapi/articles HTTP/1.1

<article>
  <author>...</author>
  <date>2018-05-17</date>
  <title>True Story</title>
  <content>Once upon a time...</content>
</article>
```

Jedná se o požadavek protokolu HTTP verze 1.1 metody `POST` směřovaný na vstupní bod webové služby nacházející se na URL `/webapi/articles`. V těle požadavku je obsažena strukturovaná zpráva, zde reprezentovaná XML elementem `article`, jenž má definovanou strukturu, která je známa serveru i klientovi.

Jiný požadavek může být opět směrován na stejný endpoint, s využitím té samé metody, jen podoba elementu `article` se bude v nějakém bodě lišit.

[4]Popis 0. úrovně v obrázku obsahuje zkratku POX, která odkazuje právě na XML formát (*Plain Old XML*).

Dle specifikace HTTP může zpráva obsažená v těle požadavku mít různý formát označovaný jako typ internetového média, tzv. MIME typ (*Multipurpose Internet Mail Extensions*). Příkladem typu média je `text/plain` pro prostý text či `application/xml`, `application/json` nebo `application/x-yaml`[5] pro strukturovaný text.

1. úroveň - zdroje

První úroveň se týká identifikace zdrojů (*resources*). Místo směřování všech požadavků na jeden endpoint komunikují klienti přímo s konkrétními zdroji, přičemž každý z nich je identifikován jiným URI[6].

```
POST /webapi/authors HTTP/1.1

<author>
  <firstName>Ansiedad</firstName>
  <lastName>Cansada</lastName>
</author>
```

```
POST /webapi/articles/1234/comments HTTP/1.1

<comment>
  <author>...</author>
  <date>2018-05-18</date>
  <text>No comment.</text>
</comment>
```

2. úroveň - HTTP metody

V obou předchozích případech byla pro každý požadavek použita jen jedna metoda, a to `POST`. REST služby 2. úrovně využívají všechny HTTP metody způsobem, jakým opravdu mají být v rámci HTTP využívány.

1. Nepoužívají na všechny požadavky jen jednu metodu (typicky GET), ale využívají i ostatní. Protokol HTTP definuje celkem 9 metod: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, TRACE, CONNECT.
2. Metody jsou aplikovány na základě principu CRUD (*Create, Read, Update, Delete*), který definuje čtyři základní operace nad zdrojem:
   - vytvoření (create),
   - čtení (read),
   - editace (update),

[5] JSON - JavaScript Object Notation, YAML - YAML Ain't Markup Language

[6] Uniform Resource Identifier - jednotný idenfitikátor zdroje, podmnožinou URI je množina URL (*Uniform Resource Locator*), který říká, jak se ke zdroji dostat

- smazání (delete).

Výběr HTTP metod používaných v REST je zobrazen v tabulce 2.1. Často diskutovaným problémem je rozdíl mezi metodami POST a PUT. Obě jsou používány pro vkládání zdroje, PUT by ale měla být *idempotentní*, což znamená, že i po opakovaném provedení operace výsledek bude vždy stejný (jeden vytvořený zdroj). Naopak POST bude mít za následek vytváření stále nových zdrojů, sice se stejnou strukturou, ale jiným identifikátorem.

Tabulka 2.1: Význam HTTP metod v REST službách

metoda	**význam**	**idempotentnost**
POST	vytvoření	NE
GET	čtení	ANO
PUT	aktualizace/náhrada	ANO
PATCH	aktualizace/editace	ANO
DELETE	smazání	ANO

Následují typické příklady použití různých HTTP metod. Pomocí prvního HTTP požadavku zobrazíme seznam článků, pomocí druhého konkrétní článek smažeme.

```
GET /webapi/articles?offset=50&limit=10 HTTP/1.1
Host: www.example.com
```

```
DELETE /webapi/articles/42 HTTP/1.1
```

3. úroveň - hypermediální řízení

Poslední úroveň odkazuje na princip HATEOAS (*Hypermedia As The Engine Of Application State*), kdy klient objevuje rozhraní dynamicky na základě odpovědí od serveru.

```
GET /webapi/articles/444 HTTP/1.1
Host: www.example.com
```

Server zašle odpověď:

```
HTTP/1.1 200 OK

<article>
  ...
  <links>
     <link rel="comments" uri="/webapi/articles/444/comments"/>
  </links>
</article>
```

Element `article` v odpovědi obsahuje mj. položku links, v níž se nachází odkaz na komentáře týkající se daného článku.

Princip HATEOAS spočívá v tom, že se dozvíme URI souvisejících zdrojů a akce, které s nimi můžeme provádět. Klient tudíž nemusí explicitně vědět, kam směřovat další požadavky, díky HATEOAS se v odpovědi na první požadavek dozví, jak pokračovat.

2.3.2 Endpoint jako základní stavební jednotka rozhraní

Klient webové služby musí umět správně sestavit požadavek a vědět, jakou odpověď může očekávat, aby ji mohl správně zpracovat. V předchozích oddílech byl několikrát zmiňován pojem *endpoint*, jehož definice v souvislosti s pojmem rozhraní může vypadat následovně:

Endpoint je základní stavební jednotka REST rozhraní, které typicky odpovídá jedna obslužná procedura. Endpoint je definován podobou HTTP požadavku a množinou jemu příslušných HTTP odpovědí.

HTTP požadavek a HTTP odpověď jsou definovány těmito položkami:

- HTTP požadavek:
  - **URI**.
  - **HTTP metoda**. Protokol HTTP 1.1 definuje celkem 9 metod.
  - **Tělo požadavku**. Datový typ, popř. struktura těla požadavku (řetězec, číslo, objekt, mapa).
  - **MIME typ těla požadavku**. Formát těla požadavku, např. text/plain, application/xml, application/json. Definováno hlavičkou požadavku Content-Type.
  - **Parametry požadavku.** Parametry požadavku nesoucí sémantický význam.
    - Parametry dotazu - součástí URI.
    - Maticové parametry - součástí URI.
    - Proměnné cesty - součástí URI.

    - Pole formuláře - ve skutečnosti uloženy v těle požadavku.
    - Ostatní hlavičky - většinou uživatelsky definované hlavičky.
    - Cookie - uloženy v hlavičce požadavku Cookie.

- HTTP odpověď:
  - **Stavový kód odpovědi.**
  - **Tělo odpovědi.**
  - **MIME typ odpovědi**. Formát těla odpovědi, např. text/plain, application/xml, application/json. Definováno hlavičkou odpovědi Content-Type.
  - **Parametry odpovědi**. Parametry odpovědi nesoucí sémantický význam.
    - ostatní hlavičky - většinou uživatelsky definované hlavičky.
    - cookie - uloženy v hlavičce odpovědi Set-Cookie.

Vztah zdroje a endpointů

Množina logicky souvisejících endpointů zajišťuje obsluhu požadavků vztahujících se k jednomu zdroji, jak je znázorněno na příkladech.

Následující endpoint zpracovává požadavky typu GET směřované na URL `/webapi/articles/{articleId}`, kde articleId je parametr cesty. Klient může očekávat dvě odpovědi; v případě nalezení článku odpověď s návratovým kódem 200 OK a tělem obsahujícím článek v podobě strukturovaného textu a v případě nenalezení článku odpověď s kódem 404 Not Found.

```
GET /webapi/articles/444 HTTP/1.1
Host: www.example.com
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type:  application/json
{
   "author":{
      "id":2,
      "firstName":"Who",
      "lastName":"Knows"
   },
   "date":"2018-12-01",
   "title":"Glaucus atlanticus"
   "content":"...",
}
```

```
HTTP/1.1 404 Not Found
```

Jiný endpoint může obsahovat požadavky příslušné stejnému URL, ale může se lišit např. v parametrech nebo HTTP metodě.

```
DELETE /webapi/articles/444 HTTP/1.1
Host: www.example.com
```

Oba endpointy se vztahovaly ke zdroji představujícímu *články* (`articles`). Následující endpoint se vztahuje ke zdroji *uživatelé* (`users`).

```
GET /webapi/users?limit=100 HTTP/1.1
Host: www.example.com
```

2.4 Formáty reprezentace rozhraní (IDL)

Tato podkapitola se zabývá strojově zpracovatelnými formáty rozhraní reprezentace REST služeb. K popisu aplikačního rozhraní softwarové komponenty obecně slouží tzv. IDL jazyky (*Interface Description Language*). Jelikož IDL popisují rozhraní mezi komponentami, které mohou být implementované pomocí rozdílných technologií, znamená to, že nejsou závislé na programovacím jazyce.

O IDL se nejčastěji mluví v souvislosti s webovými službami. Jedním z nejpoužívanějších IDL jazyků je jazyk WSDL, s nímž jsou nejúžeji svázány webové služby postavené nad protokolem SOAP.

Bavíme-li se o webových službách typu REST, je potřeba položit si otázku, co vše by popis jejich rozhraní vlastně měl obsahovat. Autoři RSDL [Pas13] říkají následující:

Popis REST služby měl zachycovat sémantiku sahající za funkcionalitu známou obecnému REST klientu.

Sémantika známá obecnému REST klientu zahrnuje používání HTTP metod a dalších principů zmíněných v kapitole 2.3.

Následuje výběr nejpoužívanějších IDL pro popis rozhraní RESTových služeb. Pro názornost jsou uvedeny ukázky dokumentace jednoduchého zdroje v jednotlivých formátech. Jedná se o zdroj na relativní adrese /webapi/articles přístupný pomocí HTTP metody GET, který vrátí seznam článků v JSON reprezentaci.

2.4.1 WADL

Nejznámějším popisným jazykem pro REST je WADL (*Web Application Description Language*) [W3C09]. Tento jazyk vznikal v letech 2006 až 2009 pod záštitou W3C

jakožto analogie k jazyku WSDL.

WADL se zapisuje pomocí XML. V současné době je to asi nejpoužívanější IDL pro REST. Poskytuje detailní strukturovaný popis rozhraní a je podporován některými Java frameworky (viz kapitola 3.2).

WADL je vytýkáno, že v jistém smyslu jde proti základním principům REST, protože neklade důraz na hypermediálně řízený návrh. Zavádí těsné spojení mezi klientem a serverem, takže změny provedené na serveru mohou znemožnit fungování existujícím klientům[7]. Popisuje rozhraní prostřednictvím statických metadat místo podpory postupného objevování rozhraní za využití odkazů (`link`).

```
<resources base="www.example.com/webapi/">
   <resource path="articles">
     <method id="getArticles" name="GET">
       <request>
         <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
              name="offset" style="query" type="xs:string" default="0"/>
         <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
              name="limit" style="query" type="xs:string" default="10"/>
       </request>
       <response>
         <representation mediaType="application/json"/>
       </response>
     </method>
   </resource>
</resources>
```

W3C doporučuje používání WADL a jeho případné rozšiřování namísto vytváření nových popisných formátů, aby vznikl jakýsi společný základ pro webové aplikace. Webová komunita se tím příliš neřídí, a tak v současné době existuje poměrně velké množství IDL.

2.4.2 WSDL 2.0

WSDL (*Web Service Description Language*) je nejpoužívanějším IDL pro popis webových služeb obecně, především proto, že je nezbytný pro popis rozhraní služeb postavených nad protokolem SOAP.

WSDL 1.1 nebylo dostačující pro popis služeb s architekturou REST, proto v roce 2007 vznikla specifikace WSDL 2.0 [W3C07]. Ta umožňuje kompletně popsat rozhraní

[7] Záleží, z jakého úhlu se na tuto vlastnost díváme. Pro účely této diplomové práce se naopak jedná o značnou výhodu.

webové služby, ale je velmi složitě lidsky čitelná (viz ukázka), pro člověka neznalého specifikace WSDL téměř nesrozumitelná. Pro popis rozhraní RESTových služeb se dnes prakticky nepoužívá.

```
<wsdl:description>
  <wsdl:documentation>    ...  </wsdl:documentation>
  <wsdl:types>
    <xs:import namespace="http://www.example.com/articlelist/xsd"
       schemaLocation="articlelist.xsd"/>
  </wsdl:types>
  <wsdl:interface name="ArticlesInterface">
    <wsdl:operation name="getArticles"
       pattern="..."
       style="..."
       wsdlx:safe="true">
      <wsdl:input element="msg:getArticles"/>
      <wsdl:output element="msg:articleList"/>
    </wsdl:operation>
  </wsdl:interface>
  <wsdl:binding name="ArticlesHTTPBinding"
     type="http://www.w3.org/ns/wsdl/http"
     interface="tns:ArticleListInterface">
    <wsdl:operation ref="tns:getArticlesList" whttp:method="GET"/>
  </wsdl:binding>
  <wsdl:service name="ArticleList" interface="tns:ArticleListInterface">
    <wsdl:endpoint name="ArticleListHTTPEndpoint"
       binding="tns:ArticleListHTTPBinding"
       address="http://www.example.com/articles/">
    </wsdl:endpoint>
  </wsdl:service>
</wsdl:description>
```

2.4.3 RSDL

RESTful Service Description Language (RSDL) [Pas13] je jazyk psaný pomocí XML určený pro návrh RESTových služeb kladoucích důraz na hypermediální řízení a vzájemné provázání zdrojů odkazy.

Byl navržen v roce 2013 Michaelem Pasternakem na konferenci Balisage, ale příliš se nerozšířil a v současné době se téměř nepoužívá. Není dostatečně zdokumentovaný a stejně jako WADL a WSDL se řadí mezi hůře čitelné formáty.

```
<media-types>
        <media-type id="med-article"
         name="application/vnd.example.article+xml">
          <documentation> Something about articles</documentation>
          <description href="example.com/mediatypes/articles.rnc"
                       type="rnc"/>
        </media-type>
</media-types>
<resource id="res-articles" name="articles">
  <location uri="www.example.com/webapi/articles"/>
  <links>...</links>
  <methods>
        <method name="GET">
          <response>
                <representation media-type-ref="med-article"
                                entity="article/>
          </response>
        </method>
  </methods>
</resource>
```

2.4.4 OpenAPI

OpenAPI Specification [Swa] (dříve Swagger Specification) je otevřená specifikace pro popis rozhraní RESTových služeb, které vznikla v rámci Open API Initiative, projektu Linux Foundation.

Jedná se o dobře lidsky čitelný formát, který má velké rozšíření v rámci Swagger[8] komunity. Specifikace mohou být psány v jazycích YAML a JSON. Je podporováno generování dokumentace API pomocí Java frameworků, ale je potřeba do zdrojového kódu doplnit další anotace. Poté lze API názorně zobrazit a testovat pomocí nástroje Swagger-UI. Také podporuje generování stub objektů pomocí nástroje Swagger Codegen.

```
{
  "/webapi/articles": {
    "get": {
      "description": "Returns all articles",
      "produces": [
        "application/json"
```

[8] https://swagger.io/docs/specification/about/

```
      ],
      "responses": {
        "200": {
          "description": "A list of articles.",
          "schema": {
            "type": "array",
            "items": {
              "ref": "#/definitions/article"
            }
          }
        }
      }
    }
  }
}
```

2.4.5 API Blueprint

API Blueprint [Api] je jazyk používaný společností Apiary. Značkovacím jazykem pro API Blueprint je Markdown. Je poměrně snadno lidsky čitelný. Apiary umožňuje návrh API v Apiary Editoru a generování stub objektů pro testování rozhraní.

```
# Group Web magazine
## Articles Collection [/articles]
### List All Articles [GET]
+ Parameters
   + limit (number)
   + offset (number)
+ Response 200 (application/json)
     //body structure in JSON schema
```

2.4.6 RAML

RESTful API Modeling Language [Ram] (RAML) vznikl v roce 2013 a je zaštiťován organizací MuleSoft. Podporuje i dokumentaci API, které není plně RESTové. Používaným formátem pro zápis je YAML. RAML je podporován mj. frameworkem Restlet (viz dále 3.2.3) a v poslední době nabývá na popularitě.

```
title: Web magazine
  baseUri: www.example.com/webapi
```

```
traits:
  - paged:
      queryParameters:
        offset:
          description: The number of first page to return (offset)
          type: number
        limit:
          description: The number of pages to return
          type: number
  - secured: !include http://raml-example.com/secured.yml
types:
  Article:
    type: object
    properties:
      author: object
      date: date
      title: string
      text: string
/articles:
  is: [ paged, secured ]
  get:
      response: body: Article[]
```

2.4.7 Shrnutí

V současné době je v rámci webové komunity používáno velké množství IDL pro REST služby. Většina IDL je schopná zachytit i API, které nesplňuje všechna kritéria pro to, aby mohlo být označeno za tzv. *plně RESTové* (angl. *RESTful*), což je velmi výhodné, protože v realitě takové *ne plně RESTové* služby naprosto převládají.

Ačkoliv neexistuje jeden standard, stále nejpoužívanější a nejpodporovanější formát je WADL, což se ale může v blízké době změnit a může ho nahradit jiný formát nabývající na popularitě (např. RAML).

Různé formáty reprezentace rozhraní dovedou zachytit tyto prvky:

- URL jednotlivých zdrojů (*resources*),
- HTTP metody aplikovatelné na zdroje,
- parametry a hlavičky požadavku,
- typ média s volitelným odkazem na popisné schéma pro jeho strukturu,
- návratové HTTP kódy,
- odkazy symbolizující vztahy mezi zdroji,

- autentizační mechanismy.

V tabulce 2.2 je přehled nejpoužívanějších IDL, existuje však ještě řada dalších.

Tabulka 2.2: IDL pro rozhraní RESTových služeb

IDL	**formát**	**výhody a nevýhody**
WADL W3C, 2006	XML	+ velmi rozšířený, podpora u Java frameworků
WSDL 2.0 W3C, 2007	XML	+ kompletní dokumentace rozhraní − velmi špatná čitelnost, pro REST téměř nepoužívaný
RSDL Michael Pasternak, 2013	XML	+ podpora odkazů a HATEOAS − málo rozšířený
OpenAPI Open API Initiative	YAML, JSON	+ snadno čitelný, rozšířený ve Swagger komunitě
API Blueprint Apiary	Mark-down	+ snadno čitelný, rozšířený v Apiary komunitě
RAML MuleSoft, 2013	YAML	+ snadno čitelný, nabývá na oblibě, používaný frameworkem Restlet

2.5 Reprezentace strukturovaných dat

Webové služby si mezi sebou často vyměňují strukturovaná data, jejichž formát musí být nezávislý na platformě. Těla složitějších HTTP požadavků a odpovědí pak tvoří strukturovaný text. V této sekci jsou představeny nejrozšířenější formáty strukturovaných dat XML, JSON a YAML spolu s jejich popisnými schématy.

2.5.1 XML

XML (*eXtensible Markup Language*) je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C. Kromě serializace dat umožňuje také snadné vytváření konkrétních značkovacích jazyků např. pro publikování dokumentů.

Syntax

Pomocí XML značek (*tagů*, uzavřených ve znacích <>) vyznačujeme v dokumentu význam jednotlivých částí textu, tzv. elementů. Značky mohou dále obsahovat atributy, dvojice klíč-hodnota. Správně strukturovaný XML element musí splňovat následující vlastnosti:

- Musí mít právě jeden kořenový element.
- Neprázdné elementy musí být ohraničeny startovací (<element>) a ukončovací značkou (</element>). Prázdné elementy mohou být označeny tagem „prázdný element“ (<element/>).
- Všechny hodnoty atributů musí být uzavřeny v uvozovkách – jednoduchých (') nebo dvojitých ("), ale jednoduchá uvozovka musí být uzavřena jednoduchou a dvojitá dvojitou. Opačný pár uvozovek může být použit uvnitř hodnot.
- Elementy mohou být vnořeny, ale nemohou se překrývat; to znamená, že každý (ne kořenový) element musí být celý obsažen v jiném elementu.

```
<author id="3">
   <firstName>Rebecca</firstName>
   <lastName>Watson</lastName>
</author>
```

Popisné schéma

Jestliže máme XML dokument, který zachycuje vlastnosti strukturovaného objektu, jsou tyto vlastnosti představovány hodnotami elementů a atributů. To, které elementy a atributy může tento XML dokument obsahovat, definuje jeho popisné schéma zvané XML Schema nebo také XSD (*XML Schema Definition*)[9].

XSD dokument definuje

- elementy a atributy, které se mohou objevit v dokumentu XML,
- které elementy jsou „podelementy“ jiných,
- pořadí a počet „podelementů“,
- „obsah“ elementu (zda je prázdný či obsahuje text),
- datový typ elementů a atributů,
- výchozí a fixní hodnoty elementů a atributů.

[9]Ve skutečnosti existuje popisných schémat pro XML více, ale XSD je z nich nejrozšířenější.

Následující XML schéma popisuje strukturu XML dokumentu uvedeného výše.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="author">
    <xs:complexType>
      <xs:attribute type="xs:long" name="id" use="required"/>
      <xs:sequence>
        <xs:element type="xs:string" name="firstName" use="required"/>
        <xs:element type="xs:string" name="lastName" use="required"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

2.5.2 JSON

JSON (*JavaScript Object Notation*) je platformně nezávislý způsob zápisu dat. Je schopen zachytit libovolnou datovou strukturu (číslo, řetězec, boolean, objekt nebo z nich složené pole) a vytvořit její reprezentaci v podobě textového řetězce. Objekt je představován množinou dvojic klíč:hodnota.

Syntax

JSON má následující syntax:

- Data představují dvojice klíč:hodnota, kde klíč je vždy psán v dvojitých uvozovkách.
- Data jsou oddělena čárkami.
- Objekty jsou ohraničeny složenými závorkami.
- Pole jsou ohraničena hranatými závorkami.
- Datové typy hodnot mohou být jen řetězce, čísla, JSON objekty, pole, boolean (true/false), null.

Níže je popis objektu představovaného XML dokumentem z předchozí sekce ve formátu JSON.

```
{
     "id": 3,
     "firstName": "Rebeca",
```

```
     "lastName": "Watson"
}
```

Popisné schéma

Ekvivalentem XSD pro JSON je JSON–schema. To předepisuje strukturu dokumentu, popis atributů (*properties*), jejich datový typ, povinnost výskytu, výchozí hodnoty, minimum a maximum u číselných datových typů a další.

```
{
    "title": "author",
    "type": "object",
    "properties": {
        "id": {
            "type": "long"
        },
        "firstName": {
            "type": "string"
        },
        "lastName": {
           "type": "string"
        }
        "age": {
            "description": "Age in years",
            "type": "integer",
            "minimum": 0
        },
    },
    "required": ["id", "firstName", "lastName"]
}
```

Ačkoliv to nebývá zvykem, pro popis struktury objektu popsaného v JSON notaci lze využít i XSD, jak je uvedeno výše.

2.5.3 YAML

Dalším formátem pro serializaci strukturovaných dat je YAML (*YAML Ain't Markup Language*), který je ze tří uvedených nejnovější. YAML dokumenty jsou snadno lidsky čitelné a používají se často pro konfigurační soubory.

Syntax

Syntax formátu YAML obsahuje poměrně velké množství pravidel, z nichž několik je uvedeno v následujícím seznamu.

- Struktura a hierarchie dat je řešena indentací (předsazením). Předsazení o jednu úroveň sestává ze 2 nebo 4 mezer; tabulátory nejsou povoleny.
- Komentáře začínají znakem # a pokračují do konce řádky.
- Seznamy (pole) mohou být
  - s prvky na samostatné řádce uvozenými pomlčkou,
  - uzavřeny v hranatých závorkách s prvky oddělenými vždy čárkou a mezerou.
- Mapy (objekty, asociativní pole, slovníky) obsahují prvky klíč:hodnota, které mohou být
  - na samostatné řádce,
  - odděleny vždy čárkou a mezerou, přičemž je celá mapa ohraničena složenými závorkami.
- Řetězce mohou být ve dvojitých i jednoduchých uvozovkách, nebo nemusí být uvozeny vůbec.

Dokument z předchozích příkladů má v YAML poměrně jednoduchou strukturu:

```
id: 3
firstName: Rachel
lastName: Watson
```

Popisné schéma

Neexistuje standardizované popisné schéma pro YAML, nicméně pro tento formát můžeme využít XSD i JSON–schema. Je možné definovat schéma vlastní (viz násl. zápis), což by ale vyžadovalo implementaci vlastního parseru, a proto tento přístup není v praxi používaný.

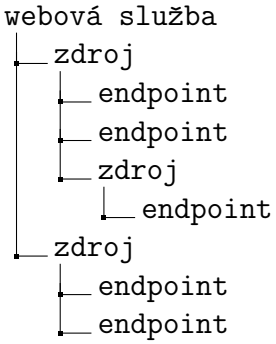
```
title: author
type: dictionary
properties:
  -- id:
       type: long
  -- firstName:
       type: string
  -- lastName :
       type: string
  -- age:
       description: Age in years
```

```
        type: integer
        minimum: 0
required: [id, firstName, lastName]
```

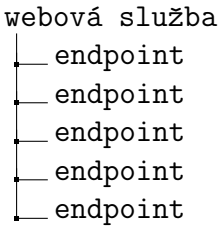
2.6 Shrnutí kapitoly

REST je architektonický styl definující způsob manipulace se zdroji. Webové služby typu REST jsou úzce spjaty s protokolem HTTP. Rozhraní REST služeb lze zachytit buď hierarchicky, nebo s plochou strukturou. Popisné jazyky pro reprezentaci rozhraní REST služeb využívají hierarchickou strukturu.

- **Zdroj**. Zdroj je logické seskupení endpointů. Obsahuje informace o některých vlastnostech společných jemu příslušným endpointům, např. část hierarchické cesty. Zdroj musí obsahovat minimálně jeden endpoint.
- **Endpoint**. Endpoint symbolizuje jeden případ užití webové služby. V případě ploché struktury rozhraní obsahuje veškeré informace pro sestavení HTTP požadavku tak, abychom na něj dostali očekávanou odpověď. V případě hierarchické struktury reprezentace rozhraní je kompletní informace získána agregací informací o endpointu a jemu nadřazených zdrojích.



Obrázek 2.2: Hierarchická struktura rozhraní



Obrázek 2.3: Plochá struktura rozhraní

Rozhraní REST služby jakožto komponenty tedy chápeme jako množinu endpointů, aneb jednotlivých dílčích služeb. Jeden endpoint v kontextu HTTP je definován podobou HTTP požadavku a jemu odpovídajících HTTP odpovědí.

3 Implementace služeb typu REST v platformě Java

Následující kapitola pojednává o způsobu implementace REST služeb v platformě Java. V úvodní části jsou popsány principy implementace obecné webové aplikace, následuje popis nejčastěji používaných frameworků pro konstrukci REST služeb a frameworků pro mapování obsahu strukturovaných zpráv. Než začneme, zopakujeme některé základní pojmy, týkající se dané problematiky.

> Server je aplikace, která poskytuje služby nebo data klientům na základě jejich požadavků. Jeden server může mít libovolné množství klientů. Klient iniciuje komunikaci zasláním požadavku na server, server jeho požadavek zpracuje a zašle mu odpověď.

Z technického hlediska chápeme pojem aplikační server pouze jako aplikaci, která zajišťuje zpracovávání příchozích HTTP požadavků a odesílání odpovědí. Součástí serveru jsou pak webové aplikace obsahující veškerou aplikační logiku.

3.1 Struktura webové aplikace

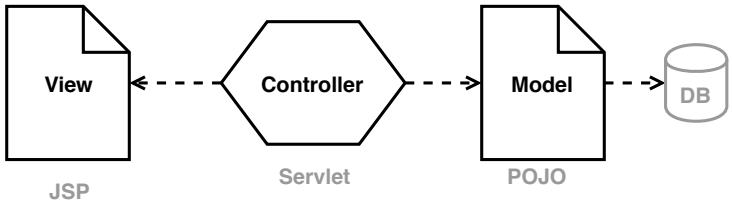
Webové aplikace v platformě Java jsou implementovány pomocí specifikace Java EE (*Java Enterprise Edition*). Java EE je jakási nadřazená specifikace, která v sobě zahrnuje jiné specifikace jako Servlet API, JSP API a další.

Architekturu typických webových aplikací lze popsat na základě návrhového vzoru **Model-View-Controller** (MVC) [BSB08]:

- **Model**. Třídy modelu zajišťují aplikační logiku a přístup k datům.
- **View**. View (česky pohled) je zodpovědný za prezentaci.
- **Controller**. Kontroler zpracovává vstup od uživatele, na jeho základě mění stav modelu a zpřístupňuje ho pohledu.

Následující obrázek znázorňuje architekturu aplikace tvořenou dle návrhového vzoru MVC. Na obrázku se nacházejí některé nové pojmy. POJO (z angl. *Plain Old Java*

Object) je označení pro třídy Javy, na něž nejsou kladena žádná omezení. JSP (*JavaServer Pages*) představují spojení Javy a HTML a tvoří v klasické webové aplikaci v Javě prezentační část. Servlet je speciální třída z kolekce Java EE a její funkce bude vysvětlena dále.



Obrázek 3.1: Model-View-Controller

Existují dvě možné podoby webových aplikací:

- **Aplikace poskytující webové služby.** Webovou aplikaci tvoří pouze webový server, který neposkytuje grafické uživatelské rozhraní. Klientem takové aplikace může být buď přímo webový prohlížeč, nebo jiná aplikace implementovaná v libovolné platformě.
- **Aplikace poskytující grafické uživatelské rozhraní.** Tento server generuje grafické uživatelské rozhraní, přičemž jsou v odpovědi zasílány HTML stránky. Klientem serveru v této architektuře je vždy webový prohlížeč.

Existují také kombinace uvedených architektur, kdy část požadavků generuje HTML stránky a část požadavků vrací odpovědi jiného typu.

Základní architektonickou jednotkou webové aplikace v jazyce Java je servlet. Servlet je zvláštní třída určená pro tvorbu dynamického obsahu. Jeho primární funkcí je zpracovávání HTTP požadavků a generování HTTP odpovědí. Aplikace typicky sestává z více servletů obsažených v tzv. webovém kontejneru, který spravuje jejich životní cyklus.

Při příchodu požadavku na server webový kontejner na základě URI určí, kterému servletu je určen, a předá mu řízení. Servlet definuje metody pro obsluhu HTTP požadavků. Pro každou metodu protokolu HTTP existuje odpovídající metoda jazyka Java. Vstupem těchto metod jsou objekty nesoucí informace o požadavku a kontextu servletu, výstupem je objekt představující odpověď. Poté, co servlet dokončí generování odpovědi, předá řízení opět webovému kontejneru a ten odpověď převede do HTTP a pošle klientovi.

Průběh zpracování požadavku může být ovlivněn dalšími komponentami, např. *filtry*. Obslužný kód filtru je vykonán před předáním řízení servletu. Typickým využitím filtru je úprava kódování požadavku nebo autentizace uživatele.

Webovou službu lze na platformě Java implementovat s využitím servletů a ostatních tříd z kolekce Java EE, ale pro zjednodušení práce programátorů byly vyvinuty její nadstavby v podobě aplikačních frameworků.

3.2 Frameworky pro REST služby

Následující definice slova framework je převzata z Wikipedie [Fra]:

> Framework (česky také aplikační rámec) je softwarová struktura, která slouží jako podpora při programování a vývoji a organizaci softwarových projektů. Může obsahovat podpůrné programy, knihovny, API, podporu pro návrhové vzory nebo doporučené postupy při vývoji. Cílem frameworku je převzetí typických problémů dané oblasti, čímž se usnadní vývoj tak, aby se návrháři a vývojáři mohli soustředit pouze na své zadání.

V Javě existuje několik frameworků a specifikací, které zajišťují mapování HTTP požadavků na příslušné obslužné metody, které vygenerují odpověď. Framework následně zajistí její zaslání na adresu, ze které byl zaslán požadavek. Framework tak usnadňuje práci vývojářům a pomáhá zajistit větší přehlednost a lepší udržitelnost zdrojového kódu. Z praktického hlediska si můžeme framework představit jako sadu knihoven.

Framework pro REST službu tak vynucuje při návrhu aplikace dodržování jistých architektonických zásad, což umožní následnou rekonstrukci rozhraní webové služby.

Na obrázku 3.2 je zobrazeno schéma REST serveru. Standardní průběh zpracování požadavku zajišťují především následující komponenty:

- **Dispečer požadavků**

  HTTP požadavky jsou analyzovány *dispečerem*, který určuje postup jejich zpracování. Dispečerem typicky bývá nějaký servlet.

  Více o dispečerech je uvedeno v sekci 3.2.

- **Třídy pro mapování obsahu** (*content mappers*)

  Pokud požadavky mají tělo, jsou předány na základě hlavičky Content-Type některé z tříd určených k mapování obsahu, který převede strukturovaný text do objektu. Při vytváření odpovědi naopak převede Java objekt do některého z formátů používaných v HTTP.

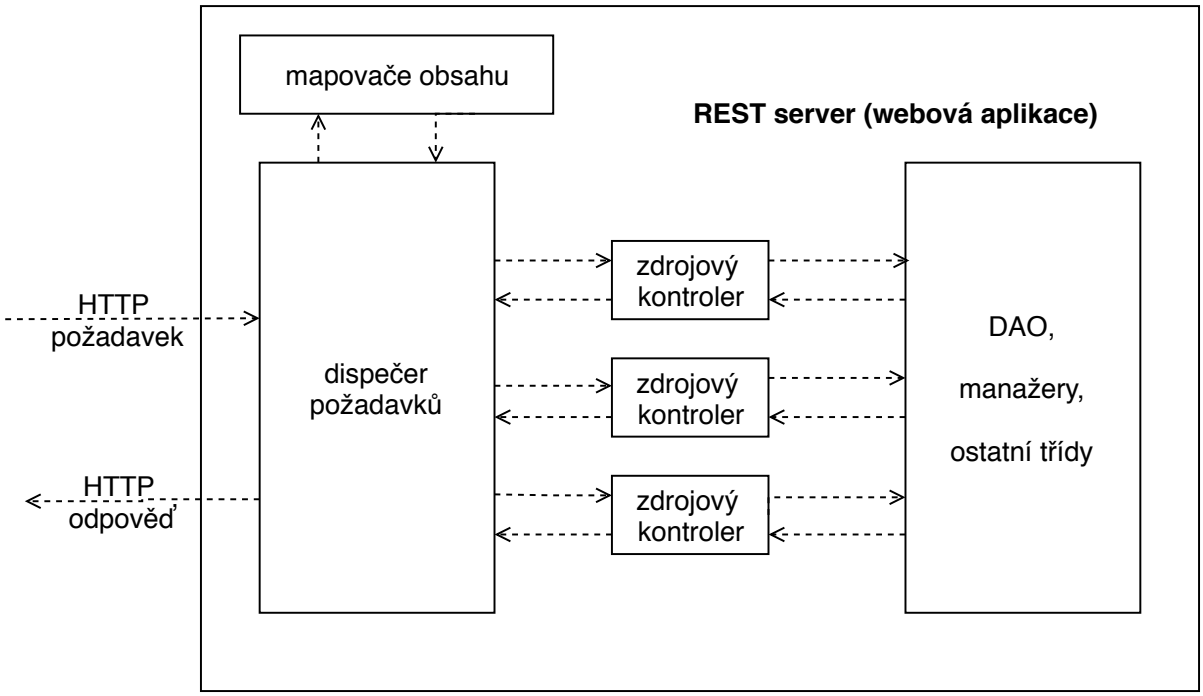
  O třídách pro mapování obsahu více v sekci 3.3.

- **Zdrojové kontrolery** (také *resource handlers*)

  Poté je řízení předáno příslušnému *zdrojovému kontroleru*. Ten má přístup k dalším třídám pro zpracování požadavku, jako jsou například databazové manažery, které

nejsou pro rekonstrukci rozhraní relevantní. Po zpracování požadavku *kontroler* vygeneruje odpověď.

Kontrolery jsou podrobně popsány v následující sekci 3.2.



Obrázek 3.2: REST server

Následující sekce obsahují rozbor nejpoužívanějších frameworků implementaci REST služeb v Javě.

3.2.1 JAX RS

JAX-RS (*Java API for RESTful Web Services*) je nejstarší rozšířená specifikace rozhraní REST služeb [Jaxb]. Jde o množinu anotací a rozhraní, kterou pak používají a implementují jednotlivé frameworky či knihovny, které tuto specifikaci splňují. Třídy, rozhraní a anotace JAX-RS jsou obsaženy v balíku `javax.ws.rs`, jehož kopie je součástí každé implementující knihovny.

> JAX-RS je specifikace implementovaná konkrétními frameworky, jako jsou Jersey, RESTEasy či Apache CXF. Všechny JAX-RS frameworky používají pro implementaci rozhraní stejnou sadu tříd a anotací.

Anotace

JAX-RS je založen na anotacích. Přehled základních JAX-RS anotací je k dispozici v dokumentaci od Oracle [Jaxa]. Retenční politika všech anotací je `RUNTIME`, což

znamená, že nejsou během překladu ani načítání třídy odstraněny a jsou k dispozici po celou dobu běhu programu.

Implementace zdrojů

Zdroj (*resource*) typicky odpovídá jedné třídě jazyka Java. Jeden zdroj zpracovává požadavky směřované na danou hierarchickou část cesty, např. začínající prefixem /articles.

Třída představující zdroj nemusí dědit žádnou zvláštní třídu ani implementovat rozhraní, její rozpoznání jakožto zdroje zajistí samotný framework. V JAX-RS musí být zdroj nejvyšší úrovně vždy označen anotací @Path. Každá třída obsahuje metody namapované na metody protokolu HTTP.

HTTP požadavky přicházejí na servlet, jehož implementace je závislá na konkrétním frameworku a který slouží jako tzv. *dispečer*, který požadavky směřuje na příslušný zdroj. Tento servlet musí mít někde uloženou informaci o zdrojích, které jsou k dispozici. Způsob registrace zdroje u dispečera zajišťuje framework.

Koncová metoda pro zpracování požadavku je vybrána podle následujících kritérií:

1. hierarchie anotací @Path odpovídá hierarchické části cesty požadavku
2. anotace metody odpovídá HTTP metodě požadavku
3. anotace @Consumes metody odpovídá hlavičce Content-Type požadavku

JAX-RS podporuje vytváření hierarchických zdrojů a oddělení aplikační logiky v rámci příslušných tříd. Na následujícím příkladu je zdroj Comments podřízený zdroji Articles. Výsledná cesta u metody getAllComments je získána konkatenací řetězců anotací @Path u nadřízené třídy, metody v nadřízené třídě, podřízené třídy a metody v podřízené třídě. Podřízený zdroj nemusí být označen anotací @Path. Lomítko na začátku řetězce u anotace @Path může být vynecháno.

```
@Path("/articles")
public class ArticlesResource {
   @Path("/{articleId}/comments")
      public CommentsResource getComments() {
         return new CommentsResource();
      }
}

public class CommentsResource {
    @GET
    public List<Comment> getAllComments(@PathParam("articleId")
                                        long articleId) {
```

```
        return commentsDAO.getAllComments(articleId);
    }
}
```

Získání informací z požadavku

Většina požadavků v sobě nese nějakou informaci, kterou z nich chceme získat. Jedná se o položky ze seznamu uvedeného v předchozí kapitole, v oddílu 2.3.2.

Proměnné a parametry jsou metodě typicky předány v podobě formálních parametrů označených anotací s hodnotou, která odpovídá jejich názvu.

```
@Path("/params")
@GET
public Response testParams(@MatrixParam("matrixParam") String matrixParam,
                           @HeaderParam("customHeader") String customHeader,
                           @CookieParam("cookie") String cookie) {
   return Response.ok().build();
}
```

Informace o URI lze získat z objektu typu javax.ws.rs.core.UriInfo označeného anotací @Context. Tato anotace je aplikovatelná také na javax.ws.rs.core.HttpHeaders, který obsahuje hlavičky požadavku.

```
@Path("/context")
@GET
public Response context(@Context UriInfo uriInfo,
                        @Context HttpHeaders httpHeaders) {
    String path = uriInfo.getAbsolutePath().toString();
    return Response.ok(path).build();
}
```

JAX-RS 2.0 obsahuje anotaci @BeanParam, která umožňuje sdružení parametrů do objektů, tzv. *JavaBeans*[1]. Anotace @DefaultValue nastaví výchozí hodnotu parametru, pokud není zaslán.

[1]JavaBean je jednoduchá třída jazyka Java, která splňuje několik základních konvencí. Všechny atributy třídy jsou soukromé a jsou zpřístupněny pouze přes speciální metody zvané *gettery* a *settery*.

```
public class PaginationBean {
    @QueryParam("offset") @DefaultValue("0") int offset;
    @QueryParam("limit") @DefaultValue("10") int limit;
    // getters and setters
}
@Path("test")
public class TestResource {
    @Path("/beanParam")
    @GET
    public Response context(@BeanParam PaginationBean bean) {
        return Response.ok("limit = " + bean.getLimit()).build();
    }
}
```

Pokud formální parametr metody není označen žádnou anotací, je považován za tělo požadavku a hledá se mapper pro jeho konverzi MIME typu do Java typu (viz dále oddíl 3.3). Očekávaný typ lze specifikovat anotací @Consumes. Pokud tělo představuje obsah HTML formuláře (soubor dvojic klíč – hodnota), jsou tyto dvojice předány metodě jako parametry s anotací @FormParam.

```
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response insertArticle(Article article, @Context UriInfo uriInfo) {
    URI location = uriInfo.getAbsolutePathBuilder()
                          .path(String.valueOf(article.getId()))
                          .build();
    articlesDAO.insertArticle(article);
    return Response.created(location).build();
}
```

Odeslání odpovědi

U HTTP odpovědi nás zajímá stavový kód odpovědi, její tělo a hlavičky (podrobně popsáno v předchozí kapitole v oddílu 2.3.2).

Tělo odpovědi může být určeno přímo návratovým typem metody, jako je vidět na následujícím příkladu. Mapování Java typu na MIME typ probíhá obdobně jako při zpracování těla požadavku. Typ média může být explicitně určen anotací @Produces. Na základě výsledku zpracování metody je použit jeden ze základních stavových kódů.

```
@GET
@Produces({ MediaType.APPLICATION_JSON})
    public List<Article> getArticlesJSON( /* parameters */) {
        return articlesDAO.getArticles(offset, limit);
    }
```

Častěji používanou variantou je použití návratového typu javax.ws.rs.core.Response. Prostřednictvím této třídy lze mj. explicitně určit stavový kód nebo nastavit hlavičky odpovědi.

```
@GET
@Path("/{articleId}")
@Produces(MediaType.APPLICATION_JSON)
public Response getArticle(@PathParam("articleId") int id) {
    if (id < 1) {
        return Response.status(Response.Status.BAD_REQUEST).build();
    }
    else {
        Article article = articlesDAO.getArticle(id);
        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
        else {
            return Response.ok(article).build();
        }
    }
}
```

Filtry a třídy pro mapování výjimek

JAX-RS poskytuje dvě speciální rozhraní pro filtr, na základě toho, zda se vztahují k požadavku či k odpovědi.

- ContainerRequestFilter. Obslužný kód filtru je vyvolán před zpracováním požadavku zdrojovou třídou. Filtry požadavků se dále dělí na dva druhy:
  - *Prematching*. Kód filtrů je vykonán před tím, než dispečer provede identifikaci endpointu pro zpracování požadavku. Používají se často právě k modifikaci atributů požadavku. Označují se anotací @Prematching.
  - *Postmatching*. Kód filtrů je vykonán až po identifikaci endpointu.

- ContainerResponseFilter. Kód filtru je vykonán po zpracování požadavku endpointem před odesláním odpovědi klientovi.

```
@Provider
@PreMatching
public class HttpMethodOverride implements ContainerRequestFilter {
   public void filter(ContainerRequestContext ctx) throws IOException {
      String methodOverride = ctx.getHeaderString("X-Http-Method-Override");
      if (methodOverride != null) ctx.setMethod(methodOverride);
   }
}
```

Dalším druhem poskytovatelů jsou obslužné třídy pro výjimky. Pokud je v průběhu vykonávání obslužné metody endpointu vyhozena výjimka, příslušná obslužná třída pro výjimku na základě jejího druhu vygeneruje odpověď a tu pošle klientovi.

```
@Provider
public class DataNotFoundExceptionMapper
       implements ExceptionMapper<DataNotFoundException> {
    @Override
    public Response toResponse(DataNotFoundException e) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}
```

Registrace zdrojů

Obecný postup pro registraci zdrojů je oddědění třídy javax.ws.rs.core.Application a překrytí její metody getClasses(), která vrací množinu zdrojů. Tuto třídu je pak ještě potřeba zaregistrovat u servletu buď pomocí anotace @ApplicationPath, nebo v konfiguračním souboru.

```
public class MyApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        return new HashSet<>(Arrays.asList(ArticlesResource.class,
                                           MessageResource.class,
                                           TestResource.class));
    }
}
```

Jersey

Referenční implementací JAX-RS je framework Jersey, vytvořený samotnými autory specifikace. Původní projekt Sun Jersey byl koupen organizací Glassfish, což vedlo ke změně balíku z com.sun.jersey na org.glassfish.jersey.

Funkci dispečera u frameworku Jersey zastává servlet org.glassfish.jersey.servlet.ServletContainer, popř. starší verze com.sun.jersey.spi.container.servlet.ServletContainer. Tento servlet společně se svým mapováním na URL je definován v souboru web.xml (*deployment descriptor*). Inicializační parametr jersey.config.server.provider.packages[2] představuje balíky s třídami obsahující zdroje a poskytovatele (*providery*)[3] pro REST službu.

```
<web-app ...>
    <display-name>Restful Web Application</display-name>
    <servlet>
        <servlet-name>jersey-servlet</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>resources, exception</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Jersey automaticky generuje popis rozhraní RESTové služby ve formátu WADL na relativní cestě application.wadl. Popis je generován dynamicky při přístupu na tuto adresu.

RESTEasy

RESTEasy [Resa] je framework od společnosti JBoss (dnes WildFly).

Dispečerem RESTEasy je org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher.-RESTEasy. RESTEasy podporuje automatické vyhledání resourců bez nutnosti jejich explicitní registrace.

[2] popř. com.sun.jersey.config.property.packages pro com.sun.jersey.spi.container.servlet.ServletContainer
[3] pomocné třídy např. pro mapování entit nebo zpracování výjimek

```
<context-param>
      <param-name>resteasy.scan</param-name>
      <param-value>true</param-value>
</context-param>
```

Apache CXF

Dalším frameworkem implementujícím JAX-RS je Apache CXF, vzniklý sloučením projektů Celtics a XFire. Apache CXF je používán nejčastěji v kombinaci s dalšími webovými frameworky, jako je např. Spring nebo Struts.

3.2.2 Spring Web MVC

Druhým velkým frameworkem pro tvorbu REST služeb v Javě je Spring Web MVC (známý spíše jako Spring MVC).

Spring Web MVC [Sprb] je postaven na Servlet API a je od samého začátku klíčovou součástí frameworku Spring. Je založen na třídě DispatcherServlet, která odesílá požadavky třídám zajišťujícím jejich zpracování (*kontrolerům*). Tyto třídy se typicky označují anotacemi @RestController a @RequestMapping. Podpora pro tvorbu REST služeb je přítomna od verze 3.0.

Implementace zdrojů

Tvorba rozhraní RESTových služeb se Spring MVC je stejně jako JAX-RS založena na anotacích. Třídám pro zpracování požadavků se říká *kontrolery* a jsou označeny anotací @RestController. Další důležitou anotací je @RequestMapping, která zajišťuje mapování požadavku na Java metodu a zastává více funkcí na základě hodnot jejích atributů (viz tabulka 3.1). Jestliže není specifikován atribut method, jsou na příslušnou metodu jazyka Java mapovány všechny HTTP metody, které framework obsluhuje. Existují i anotace pro konkrétní HTTP metody - @GetMapping, @PostMapping apod.

```
@RestController
@RequestMapping("/articles")
public class ArticlesController {
    @RequestMapping(method = RequestMethod.GET,
                    produces = MediaType.APPLICATION_JSON_VALUE)
    public List<Article> getArticlesJSON
                   (@RequestParam(value="offset") int offset,
                    @RequestParam(value="limit") int limit) {
        return articlesDAO.getArticles(offset, limit);
    }
```

Tabulka 3.1: Atributy @RequestMapping

atribut	JAX-RS ekvivalent	význam
value/path	@Path	hierarchická část cesty
method	@GET, @POST apod.	HTTP metoda
params	není	udává povinnou dotazovací část URL
headers	@HeaderParam, @Context HttpHeaders	hlavičky požadavku
consumes	@Consumes	MIME typ těla požadavku
produces	@Produces	MIME typ těla odpovědi

Spring MVC nepodporuje tvorbu podřízených kontrolerů jako JAX-RS, ale umožňuje rozdělení aplikační logiky do více kontrolerů, jak spolu s předchozí ukázkou znázorňuje následující kód.

```
@RestController
@RequestMapping("/articles")
public class CommentsController {
    @RequestMapping(value = "/{articleId}/comments",
                    method = RequestMethod.GET)
    public List<Comment> getComments(@PathVariable long articleId) {
        return commentsDAO.getAllComments(articleId);
    }
}
```

Získání informací z požadavků

Parametry a proměnné HTTP požadavku mohou být metodě předány různými způsoby.

Prvním z nich je jejich předání v podobě formálnách parametrů, kdy jsou doprovázeny příslušnou anotací. Pokud název formálního parametru odpovídá názvu parametru v požadavku, není třeba specifikovat hodnotu anotace.

```
@RequestMapping(path = "params", method = RequestMethod.GET)
public String testParams(@RequestParam(defaultValue="0") int offset,
                         @RequestParam(defaultValue="0") int limit,
                         @MatrixVariable("xxx") String matrixParam
                         @RequestHeader  String customHeader,
                         @CookieValue("susenka") String cookie) {
```

```
     return "Matrix param = " + matrixParam + ", header = " + customHeader
          + ", cookie = " + cookie;
}
```

Parametry lze sdružovat do objektů (analogie k JAX-RS @BeanParam), v případě Springu však parametr typu JavaBean nemusí být označen žádnou anotací.

```
public class PaginationBean {
    private int offset;
    private int limit;
    // getters and setters
}
@RestController
@RequestMapping("test")
public class TestController {
    @RequestMapping(path = "beanParam", method = RequestMethod.GET)
    public ResponseEntity context(PaginationBean bean) {
        return ResponseEntity.ok("limit = " + bean.getLimit() +
                                 + ", offset = " + bean.getOffset());
    }
 }
```

V případě HTTP požadavku s neprázdným tělem je metodě předán parametr označený anotací @RequestBody. MIME typ těla požadavku je definován atributem consumes anotace @RequestMapping.

```
 public ResponseEntity insertArticle(@RequestBody Article article,
                                     HttpServletRequest request) {
        URI location = getUriFromRequest(request) + article.getId());
        articlesDAO.insertArticle(article);
        return ResponseEntity.created(location).build();
}
```

Metodě lze také předat celý objekt typu javax.servlet.http.HttpServletRequest, z čehož lze získat informace o URI požadavku, parametrech, hlavičkách a další.

```
@RequestMapping("request")
public ResponseEntity context(HttpServletRequest request) {
    //request.getQueryString();
```

```
    //request.getCookies();
    return ResponseEntity.ok("User-Agent=" + request.getHeader("User-Agent"));
}
```

Odeslání odpovědi

Pokud je návratovou hodnotou metody přímo objekt, který má být mapován na tělo odpovědi, provede se tak za využití příslušeného mapperu.

```
@RequestMapping(method = RequestMethod.GET, produces = "application/json")
    public List<Article> getArticlesJSON(/* parameters */) {
        return articlesDAO.getArticles(offset, limit);
    }
```

Druhý přístup zahrnuje použití objektu typu org.springframework.http.ResponseEntity, kterému lze programově nastavit stavový kód i entitu představující tělo odpovědi.

```
@RequestMapping(value = "{articleId}", method = RequestMethod.GET)
public ResponseEntity getArticle(@PathVariable("articleId") int id) {
    if (id < 1) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
    }
    else {
        Article article = articlesDAO.getArticle(id);
        if (article == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(article);
     }
}
```

Třetí možností je přistupovat přímo k objektu javax.servlet.http.HttpServletResponse.

```
@RequestMapping("cookie")
public String testCookie(HttpServletResponse response) {
    response.addCookie(new Cookie("susenka", "cokoladova"));
    return "Check cookies for 'susenka'.";
}
```

Filtry

Na rozdíl od JAX-RS Spring MVC nedefinuje speciální třídy pro filtrování požadavků, ale používá klasické filtry ze Servlet API.

3.2.3 Restlet

Framework Restlet [Resb] cílí na různá exekuční prostředí a je k dispozici v edicích pro Javu SE, Javu EE, Android, OSGi a další. K problematice REST služeb přistupuje odlišně od předchozích dvou specifikací.

REST služba může být ve frameworku Restlet implementována více způsoby. Třídy pro obsluhu požadavků (zdroje, kontrolery) dědí od třídy org.restlet.resource.ServerResource a jsou registrovány programově ve třídě dědící od org.restlet.Application.

```
public class MyRestletApplication extends Application {
    @Override
    public Restlet createInboundRoot() {
        Router router = new Router(getContext());
        router.attach("/articles/{articleId}", ArticlesResource.class);
        router.attach("/articles", ArticlesResource.class);
        return router;
    }
}
```

Třídy mohou buď překrývat metody get(), post() apod., nebo definovat nové metody a označit je příslušnými anotacemi. Díky atributům předka má třída k dispozici objekty typu org.restlet.Request a org.restlet.Response.

```
public class ArticlesResource extends ServerResource {

    private ArticlesDAO articlesDAO = ArticlesDAOMock.getInstance();

    @Get("json")
    public Representation getArticlesJson() {
        String articleIdPathVariable = getAttribute("articleId");
        if (articleIdPathVariable != null) {
            long articleId;
            try {
                articleId = Long.valueOf(articleIdPathVariable);
                Article article = articlesDAO.getArticle(articleId);
```

```
                if (article == null) {
                    getResponse().setStatus(Status.CLIENT_ERROR_NOT_FOUND);
                }
                return new JacksonRepresentation<>(article);
            } catch (NumberFormatException e) {
                getResponse().setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
                return null;
            }
        }
        else {
            Form query = this.getQuery();
            int offset, limit;
            String offsetParam = query.getValues("offset");
            String limitParam = query.getValues("limit");
            offset = offsetParam == null ?  0 : Integer.valueOf(offsetParam);
            limit = limitParam == null ?  10 : Integer.valueOf(limitParam);
            List<Article> articles = articlesDAO.getArticles(offset, limit);
            return new JacksonRepresentation<>(articles);
        }
    }
```

V souboru web.xml je definován dispečer požadavků org.restlet.ext.servlet.ServerServlet.

3.3 Mapování reprezentací zdrojů

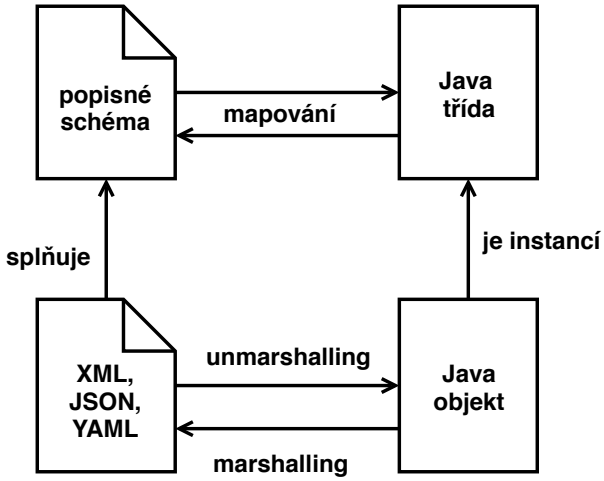
Těla HTTP požadavků a odpovědí velmi často mají formu strukturovaného textu. Pokud se jedná o typ média application/xml nebo application/json, představuje tento strukturovaný text objekt, pro který lze nalézt ekvivalent v objektovém programovacím jazyce.

Jedna třída jazyka Java případně doplněná o další informace představuje popisné schéma pro strukturovaný dokument (viz 2.5), definuje názvy jeho prvků a jejich datové typu. Konkrétnímu dokumentu pak odpovídá jedna instance této třídy.

V Javě existují frameworky pro mapování objektů na strukturovaný text a jejich vzájemnou konverzi. Tyto frameworky zpravidla obsahují anotace, kterými lze do třídy jakožto popisného schématu pro objekt doplnit další informace jako např. povinnost výskytu prvků, jejich pořadí, výchozí hodnoty a další.

3.3.1 Mapování XML

V Javě existuje standard definující API pro konverzi Java objektů do XML a zpátky (tzv. *marshalling* a *unmarshalling*[4]) s názvem *Java Architecture for XML Binding* (JAXB). JAXB představuje specifikaci, pro kterou je možné použít více implementací. JAXB je spolu s referenční implementací od Sun součástí Java SE i Java EE.



Obrázek 3.3: Konverze Java objektů do XML (*marshalling*) a zpátky (*unmarshalling*)

JAXB definuje tzv. *service provider*, který umožňuje výběr JAXB implementace.

Třída v Javě a její atributy jsou typicky označeny anotacemi z balíku javax.xml.bind.annotation, poskytujícími doplňující informace. Některé frameworky používají JAXB anotace i pro mapování objektů na JSON. V tabulce 3.2 je přehled základních JAXB anotací.

Následuje příklad použití anotací v Javě. Této třídě odpovídají ukázky popisných schémat uvedených dříve v sekci 2.5.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class User {
    @XmlAttribute(required = true)
    private Long id;
    @XmlElement(required = true)
    private String firstName;
    @XmlElement(required = true)
    private String lastName;
    @XmlTransient
    private String uselessField;
```

[4]Tyto pojmy bývají volně zaměňovány za pojmy *serializace* a *deserializace*.

```
    /* getters and setters */
}
```

Tabulka 3.2: Základní JAXB anotace

anotace	**význam**
@XmlRootElement	kořenový element, umožňuje definovat název a namespace
@XmlElement	element, umožňuje definovat název, namespace, povinnost výskytu, nulovost, výchozí hodnotu
@XmlAttribute	atribut, umožňuje definovat název, namespace a povinnost výskytu
@XmlType	aplikovatelný na třídu, interface nebo enum, umožňuje definovat pořadí prvků v něm obsažených
@XmlAccessorType	definuje modifikátor přístupu
@XmlWrapperElement	definuje obalovací element pro jiný, vhodné pro kolekce
@XmlTransient	příznak, že atribut nemá být mapován

Tabulka na Wikipedii [Jaxc] znázorňuje vzájemné mapování datových typů mezi Javou a XML.

Dalšími implementacemi JAXB jsou např. EclipseLink MOXy [Mox] nebo Apache Camel [Cam]. EclipseLink MOXy kromě JAXB poskytuje ještě vlastní API pro mapování XML.

Nadstavbou nad JAXB je také Spring OXM [Spra].

3.3.2 Mapování JSON a YAML

Další z knihoven implementujících JAXB je Jackson. Ta kromě mapování XML obsahuje vlastní API po mapování JSON a modul pro mapování YAML.

Pro standardní POJO splňující jmenné konvence pro gettery a settery lze Jackson pro mapování do JSON a YAML použít bez jakýchkoli anotací v modelové třídě.

Přesto Jackson obsahuje sadu doplňujících anotací, z nichž některé jsou uvedeny v tabulce 3.3.

Tabulka 3.3: Základní Jackson anotace pro JSON

anotace	**význam**
@JsonRootName	umožňuje definovat název kořenového elementu
@JsonPropertyOrder	umožňuje nastavit pořadí prvků obsažených v objektu
@JsonProperty	umožňuje nastavit atributu setter a getter, pokud nesplňuje jmenné konvence pro gettery a settery
@JsonGetter @JsonSetter	ekvivalent pro @JsonProperty aplikovatelný na getter a setter
@JsonAnyGetter @JsonAnySetter	umožňuje definovat getter a setter pro prvky v mapě, která představuje množinu atributů

3.3.3 Integrace v REST frameworcích

Java REST frameworky využívají frameworky pro mapování reprezentací zdrojů při konverzi těla HTTP požadavků do Java objektů a Java objektů do HTTP odpovědí.

JAX-RS

V JAX-RS tuto konverzi zajišťují tzv. *content handlery*, třídy implementující rozhraní MessageBodyWriter a MessageBodyWriter z balíku javax.ws.rs.ext. Tyto třídy se zaregistrují prostřednictvím anotace @Provider.

```
public interface MessageBodyReader<T> {

   boolean isReadable(Class<?> type, Type genericType,
                      Annotation annotations[], MediaType mediaType);

   T readFrom(Class<T> type, Type genericType,
              Annotation annotations[], MediaType mediaType,
              MultivaluedMap<String, String> httpHeaders,
              InputStream entityStream)
                        throws IOException, WebApplicationException;

}
```

```
public interface MessageBodyWriter<T> {

   boolean isWriteable(Class<?> type, Type genericType,
```

```
                        Annotation annotations[],
                                    MediaType mediaType);


   long getSize(T t, Class<?> type, Type genericType,
                 Annotation annotations[], MediaType mediaType);


   void writeTo(T t, Class<?> type, Type genericType,
                Annotation annotations[],
                MediaType mediaType,
                MultivaluedMap<String, Object> httpHeaders,
                OutputStream entityStream)
                       throws IOException, WebApplicationException;
}
```

Spring Web MVC

Ekvivalentem *content handlerů* je v případě frameworku Spring rozhraní org.springframework.http.converter.HttpMessageConverter.

Existuje sada výchozích konvertorů. Každá implementace HttpMessageConverter je asociována s jedním nebo více typy média. Která implementace bude použita, je určeno hlavičkami Content-Type a Accepts.

Konfiguraci konvertorů lze provést oddděděním třídy WebMvcConfigurerAdapter a překrytím metody configureMessageConverters().

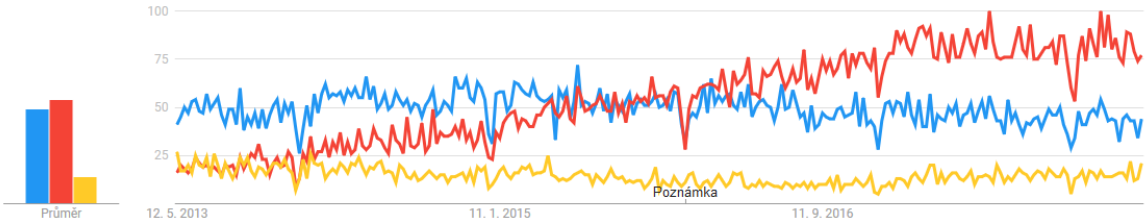
3.4 Shrnutí kapitoly

Není snadné nalézt vypovídající statistiky pro míru používanosti REST frameworků. Jistou hrubou představu nám může poskytnout srovnání četnosti hledaných výrazů dle Google Trends. Na následujícím grafu je znázorněn výsledek relativní četnosti vyhledávání výrazů "jax-rs" (modrá linka), "spring rest" (červená linka) a "restlet" (žlutá linka) za posledních pět let (vztažené k dubnu 2018).

Ačkoliv na výsledcích tohoto vyhledávání nemůžeme stavět přesné statistické analýzy, protože uživatelé mohou vyhledávat na základě jiných klíčových slov a srovnání není objektivní vzhledem k JAX-RS, což je pouhý standard a spadá pod něj více frameworků, jisté závěry z grafu vyvodit lze.

V současnosti jsou nejpoužívanějšími frameworky pro vytváření REST služeb v Javě

Spring Web MVC a frameworky splňující specifikaci JAX-RS. Na největší popularitě nabývá Spring Web MVC, čemuž přispívá fakt, že se jedná o komplexní framework pro použití ve webových aplikacích. Naopak popularita frameworku Restlet klesá.



Obrázek 3.4: Trendy ve vyhledávání klíčových slov pro REST frameworky

3.4.1 Společné vlastnosti JAX-RS a Spring Web MVC

Spring Web MVC a JAX-RS mají mnoho společných vlastností, mezi než patří následující:

- Jedná se o anotační frameworky.
- REST endpointy jsou reprezentovány metodami sdružovanými do zdrojových tříd.
- Všechny parametry dotazu jsou předávány endpointu v podobě parametrů metody, umožňují také sdružování parametrů do tříd, tzv. *parameter beans*
- Obsahují mechanismus pro nastavení výchozí hodnoty parametru, pokud není poslán.
- Tělo požadavku je endpointu předáno v podobě parametru metody.
- Mezi znaky určující endpoint patří informace o tom, jaké MIME typy požadavků endpoint akceptuje a jaké MIME typy odpovědí vrací. Oba frameworky podporují *content negotiation*.
- Jeden endpoint umožňuje vrátit více různých odpovědí. Toto vychází z konceptu jazyka Java, kdy jedna metoda může obsahovat téměř libovolný počet *return* instrukcí.
- Podporují nastavení stavového kódu odpovědi a připojení hlaviček včetně cookies.
- Podporují integraci s frameworky pro mapování obsahu.

3.4.2 Odlišnosti JAX-RS a Spring Web MVC

Některé odlišnosti zmiňovaných frameworků se týkají jen terminologie, jiné představují důležité implementační rozdíly. Pro přehlednost jsou odlišnosti uvedeny v následující tabulce.

Tabulka 3.4: Odlišnosti JAX-RS a Spring Web MVC

vlastnost	**JAX-RS**	**Spring Web MVC**
podstata	specifikace	framework - jedna implementace
zdrojová třída	zdroj (*resource*)	kontroler (*controller*)
parametr dotazu	query parameter	request parameter
položka formuláře	form parameter	request parameter (nerozlišuje mezi položkou formuláře a parametrem dotazu)
povinnost výskytu parametrů	V základní podobě neobsahuje mechanismus pro její zajištění, nutnost rozšíření pro podporu validace dle JSR 303[6]. Ve výchozím stavu jsou parametry nepovinné (kromě parametrů cesty).	Zajištěna prostřednictvím hodnoty params anotace RequestMapping, popř. hodnoty required anotací určující konkrétní parametry. Ve výchozím stavu jsou hodnoty parametrů povinné, výjimkou jsou parametry obsažené v *beanu*.
název parametru	určen anotací	určen anotací nebo názvem formálního parametru metody

[6] `http://beanvalidation.org/1.0/spec/`

Tabulka 3.5: Odlišnosti JAX-RS a Spring Web MVC - pokračování

vlastnost	**JAX-RS**	**Spring Web MVC**
parametry jako JavaBean	Atributy beanu parametrů mohou představovat všechny typy parametrů požadavku jako parametry metody a jsou označeny příslušnými anotacemi. Atributy musí být veřejné nebo k nim musí existovat příslušné *settery*. Proměnná představující bean je označena příslušnou anotací.	V beanu parametrů se mohou vyskytovat jen parametry dotazu, cesty a formulářové. Nejsou označeny žádnými zvláštními anotacemi. Ke všem atributům musí existovat příslušný *setter*. Proměnná představující bean není označena žádnou anotací.
tělo požadavku	Proměnná představující tělo není označena žádnou zvláštní anotací.	Proměnná označena anotací.
HTTP metoda	Endpoint je identifikován právě jednou HTTP metodou. Možné metody jsou GET, POST, PUT, DELETE, HEAD, OPTIONS.	Endpoint může obsluhovat více HTTP metod. V případě, že není žádná uvedena, obsluhuje všechny povolené; jmenovitě GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS.
implementační podpora hierarchie zdrojů	ano	ne

4 Úložiště CRCE a projekt JaCC

Cílem této diplomové práce je implementace rozšíření úložiště CRCE pro schopnost zachycení rozhraní REST služeb, které komponenty poskytují. Z toho důvodu je první část kapitoly zaměřena právě na CRCE a jeho datový model.

4.1 CRCE - koncept úložiště

Vyvíjení aplikace sestávající z komponent může být z hlediska udržování konzistence velmi náročné. Například aktualizace některé podmnožiny komponent může představovat složitý problém.

> CRCE (*Component Repository supporting Compatibility Evaluation*) je komponentové úložiště založené na *OSGi Bundle Repository* (OBR)[1] vytvořené výzkumnou skupinou ReliSA působící na Fakultě aplikovaných věd Západočeské univerzity. Jeho účelem je ukládání softwarových komponent a ověřování jejich vzájemné kompatibility. [Crc]

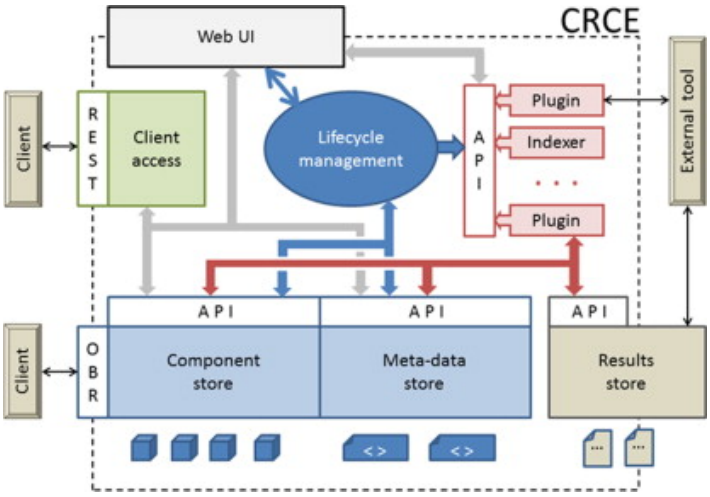
Pro bližší pochopení účelu úložiště uvedeme jednoduchý případ užití:

1. Uživatel nahraje do úložiště komponentu, typicky archiv obsahující Java bytecode.
2. Tím jsou aktivovány procedury, které provedou analýzu archivu a extrahují z něho metadata.
3. Uživatel spustí nad daným archivem testy kompatibility, které provedou analýzu metadat.

Na obrázku 4.1 je zobrazena architektura úložiště. Základními stavebními bloky jsou **úložiště komponent**, **úložiště metadat** a **úložiště výsledků testů**. [BJ15]

Úložiště je samo o sobě vyvinuto jako komponentová OSGi aplikace a je členěna na moduly. Některé z těchto modulů jsou tzv. *indexery*. Při nahrávání artefaktu do úložiště (typicky JAR nebo WAR souboru) jsou tyto indexery postupně volány, přičemž každý z nich provede analýzu artefaktu a získá z něj popisná data dle svého účelu. Tato data jsou následně uložena do úložiště metadat.

[1] OSGi je specifikace pro dynamické modulární systémy v Javě. V OSGi terminologii se modul nazývá *bundle*



Obrázek 4.1: Architektura CRCE

4.2 Datový model CRCE

Zdrojem pro následující text je diplomová práce Davida Pejřimovského z roku 2015 [Pej15], která se zabývá ukládáním popisu veřejně dostupných webových služeb v datovém modelu CRCE (viz 4.2.1).

Datový model uchovává metadata o softwarových komponentách a vychází z obecného konceptu OBR. Základem modelu (viz obr. 4.2) je úložiště (*repository*), které uchovává množinu zdrojů (*resources*).

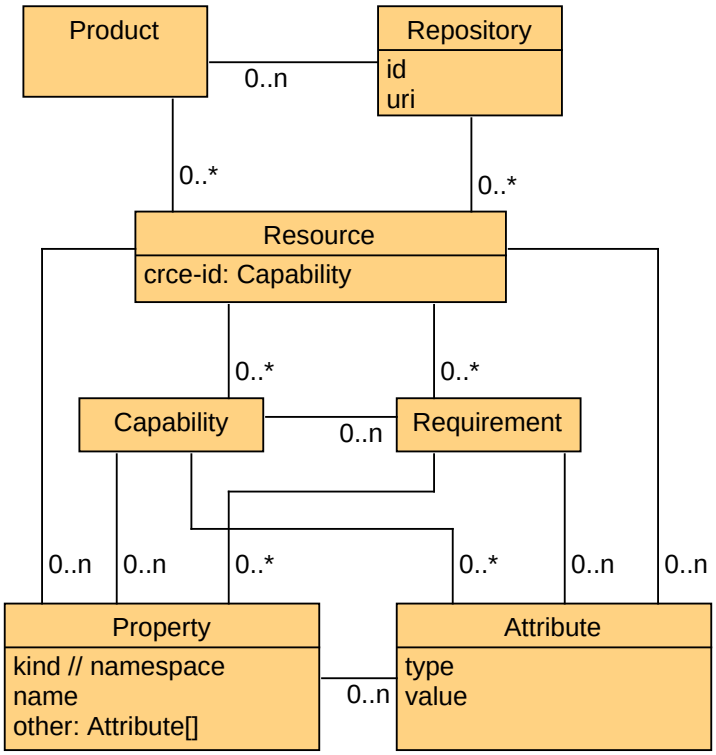
Zdroj obsahuje množinu metadat pro konkrétní komponentu. S jedním zdrojem může být svázáno libovolné množství požadavků (*requirements*), schopností (*capabilities*) a vlastností (*properties*). Jejich význam je následující:

- **requirement** - požadavek komponenty nezbytný pro její bezchybnou funkčnost,
- **capability** - schopnost komponenty, kterou poskytuje jiným komponentám (doplněk k požadavkům jiných komponent),
- **property** - dodatečná informace.

4.2.1 Reprezentace rozhraní webových služeb

Webové službě odpovídá zdroj (*resource*). Mluvíme-li o rozhraní webové skužby, myslíme tím schopnosti (*capabilities*), které poskytuje svému okolí.

[Pej15] ve své práci rozšířil stávají datový model o nové druhy metadat, konkrétně zavedením nových schopností (*capabilities*) a vlastností (*properties*) zdroje. Předmětem jeho práce jsou veřejně dostupné webové služby běžící na konkrétním serveru. Zpracování



Obrázek 4.2: Datový model CRCE

webových služeb probíhá parsováním jejich veřejně dostupného popisného dokumentu.

Nově zavedená metadata:

- schopnosti (capabilities)
  - webservice.identity
  - webservice.endpoint
- vlastnosti (properties)
  - webservice.endpoint.parameter
  - webservice.endpoint.response

Schopnost **webservice.identity**

Reprezentuje identifikační prvky běžící webové služby, jako jsou datum zpracování IDL dokumentu nebo URL, ze kterého je služba dostupná. Pro účely této diplomové práce, která se zabývá rekonstrukcí rozhraní z Java archivu s implementací REST služby, tudíž není relevantní.

Schopnost webservice.endpoint

Reprezentuje konkrétní endpoint webové služby, aneb dílčí webovou službu.

Poznámka: V případě REST služeb je endpoint identifikován kromě URL ještě řadou dalších vlastností HTTP požadavku.

Tabulka 4.1: Atributy webservice.endpoint

atribut	význam
name	název endpointu
url	URL požadavků, které endpoint zpracovává

Vlastnost webservice.endpoint.parameter

Reprezentuje jeden ze vstupních parametrů endpointu.

Tabulka 4.2: Atributy webservice.endpoint.parameter

atribut	význam
name	název parametru
type	datový typ parametru, popř. odkaz na gramatiku
order	pořadí v rámci všech parametrů endpointu
isOptional	zda je parametr nepovinný, obsahuje logickou 0 či 1
isArray	zda je parametr pole, obsahuje logickou 0 či 1

Vlastnost webservice.endpoint.response

Reprezentuje jednu z možných odpovědí endpointu.

Tabulka 4.3: Atributy webservice.endpoint.response

atribut	význam
type	datový typ odpovědi, popř. odkaz na gramatiku
isArray	zda odpověď obsahuje pole entit, obsahuje logickou 0 či 1

Tato kapitola pojednávala mimo jiné o podobě existujícího datového modelu úložiště.

V kapitole 5 je pak prezentován návrh úpravy modelu pro schopnost zachycení kompletního rozhraní služeb typu REST.

4.3 JaCC

JaCC (*Java Class Comparator*) je knihovna pro statickou analýzu bytecode vyvíjená na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Zdrojem pro následující text je diplomová práce Jana Ambrože z roku 2016, zabývající se optimalizací JaCC [Amb16].

4.3.1 Princip

V práci je uvedena jedna z klíčových myšlenek tohoto nástroje:

Bytecode vzniká při kompilaci zdrojového kódu v programovacím jazyce Java a obsahuje informaci o stavbě tříd coby základních stavebních prvcích při vývoji v tomto jazyce. Tuto informaci o stavbě tříd je možné s použitím reverzního inženýrství zpětně extrahovat z bytecode a uložit do vhodných datových struktur, se kterými je možné dále pracovat. Obsahem těchto datových struktur pak mohou být například informace o atributech, konstruktorech či metodách pro danou třídu.

> Reverzní inženýrství je proces analýzy předmětného systému s cílem identifikovat komponenty systému a jejich vzájemné vazby a/nebo vytvořit reprezentaci systému v jiné formě nebo na vyšší úrovni abstrakce. [Soc]

Nástroj JaCC umožňuje provést nejen analýzu bytecode, ale také porovnání různých verzí konkrétních tříd. Pro účely této práce je relevantní pouze první fáze zpracování, a to načtení souborů obsahujících bytecode a vytvoření datové reprezentace.

4.3.2 Získání modelu tříd z bytecode

JaCC pro načítání bytecode používá open-source framework ASM [**asms**]. ASM je víceúčelový framework nejen pro analýzu Java bytecode, ale také pro modifikaci existujících tříd či dynamické vytváření tříd nových.

Je založen na návrhovém vzoru *visitor*. Umožňuje s různými objektovými strukturami svázat události a obslužné metody, které se při nastalé události vykonají. ASM definuje několik rozhraní, jako jsou ClassVisitor, FieldVisitor, MethodVisitor a AnnotationVisitor.

Nástroj JaCC se skládá z několika modulů. Načtení bytecode zajišťuje modul javaypes-loader, konkrétně třída AsmDataParser. Po načtení bytecode je vytvořena datová reprezentace za využití tříd z modulu javatypes, odpovídajících základním stavebním elementům třídy v Javě, např. JClass, JField, JMethod, JAnnotation a další.

5 Návrh rekonstrukce rozhraní

V této kapitole je uveden návrh doménového modelu REST API, reprezentace v datovém modelu CRCE a návrh algoritmu pro jeho rekonstrukci.

5.1 Doménový model REST rozhraní

Při návrhu reprezentace rozhraní REST služeb je třeba zohlednit koncept protokolu HTTP přiblížený v kapitole 2 a principy implementace REST služby v jazyce Java, jimž byla věnována kapitola 3.

Struktury pro reprezentaci obecné webové služby uvedené v sekci 4.2.1 nebudou použity, protože některé z vlastností definovaných pro webové služby obecně nejsou pro REST služby relevantní a naopak z důvodu větší specifičnosti je pro REST služby možné zachytit rozhraní na větší úrovni detailu.

Pro reprezentaci rozhraní byla stejně jako u obecných webových služeb zvolena plochá struktura, kdy je rozhraní webové služby vnímáno jako množina endpointů, nesoucích kompletní informaci pro vytvoření HTTP požadavku a zpracování očekávané odpovědi.

Níže je uvedeno stromové schéma doménového modelu rozhraní REST služby. V další sekci je představena reprezentace tohoto modelu v datovém modelu CRCE, přičemž je podrobně vysvětlen význam jednotlivých položek.

```
endpoint
├── název endpointu (name)
├── cesty k endpointu (paths)
├── HTTP metody (methods)
├── MIME typy těla požadavku (consumes)
├── MIME typy těla odpovědi (produces)
└── parametry požadavku (request parameters) 0 - N
    ├── název (name)
    ├── kategorie (category)
    ├── datový typ (dateType)
    ├── je pole (isArray)
    └── je volitelný (isOptional)
```

- výchozí hodnota (defaultValue)
- tělo požadavku (request body) 0 - 1
  - struktura (structure)
  - je pole (isArray)
  - je volitelné (isOptional)
- odpovědi (responses) 1 - N
  - stavový kód (status)
  - struktura (structure)
  - je pole (isArray)
  - parametry odpovědi (response parameters) 0 - N
    - název (name)
    - kategorie (category)
    - datový typ (dateType)
    - je pole (isArray)

5.2 REST rozhraní v datovém modelu CRCE

Jak je uvedeno výše, Java archiv je v datovém modelu CRCE reprezentován entitou *resource*[1]. Pro zachycení REST rozhraní byly definovány dvě nové schopnosti a čtyři nové vlastnosti zdroje.

Schopnost restimpl.identity

Obsahuje jen jediný atribut, a to název frameworku (popř. specifikace), pomocí něhož je služba implementovaná. Tato schopnost je nadřazená všem schopnostem restimpl.endpoint.

Schopnost restimpl.endpoint

Capability restimpl.endpoint je základní stavební jednotkou rozhraní.

Vlastnost restimpl.endpoint.requestbody

Požadavek endpointu může nebo nemusí obsahovat tělo. To reprezentuje property restimpl.endpoint.requestbody.

Vlastnost restimpl.endpoint.requestparameter

Property představuje parameter požadavku. Sem řadíme i zvláštní druhy HTTP hlaviček (např. uživatelsky definované) a cookie.

Vlastnost restimpl.endpoint.response

Protože jazyk Java umožňuje více výstupních bodů z metody a REST frameworky podporují vytváření komplexních HTTP odpovědí, může jeden endpoint na základě

[1]Pozor: nezaměňovat s pojmem *resource* představujícím zdrojovou třídu logicky sdružující množinu endpointů.

Tabulka 5.1: Atributy restimpl.endpoint

atribut	**význam**
name	název endpointu
path	část cesty, na které je endpoint k dispozici
method	HTTP metoda požadavku
produces	seznam všech možných MIME typů odpovědí endpointu (hodnot hlavičky odpovědi Content-Type)
consumes	seznam všech možných MIME typů těla požadavku (hodnot hlavičky požadavku Content-Type)

Tabulka 5.2: Atributy restimpl.endpoint.requestbody

structure	struktura těla požadavku, plné jméno třídy reprezentující tělo, popř. primitivní datový typ (znak pro jeho bytecode reprezentaci) [3]
isOptional	zda je tělo vyžadováno ve všech případech
isArray	zda se jedná o pole či kolekci prvků s danou strukturou

Tabulka 5.3: Atributy restimpl.endpoint.requestparameter

name	**význam**
category	kategorie parametru (viz tab. 5.4)
dataType	plné jméno třídy reprezentující tělo, popř. primitivní datový typ
defaultValue	výchozí hodnota, pokud parameter není přítomen, nemusí být vyplněna
isOptional	zda je parametr povinný
isArray	zda se jedná o pole či kolekci prvků

vstupních parametrů teoreticky vracet naprosto odlišné odpovědi.

Vazba mezi restimpl.endpoint a restimpl.endpoint.response je tedy 1:N.

[3] Nezaměňovat s MIME typem těla požadavku. Java frameworky vnímají MIME typ těla požadavku jako jednu z vlastností identifikujících endpoint, proto je tato vlastnost zachycena jako atribut consumes u restimpl.endpoint.

Tabulka 5.4: Kategorie parametrů požadavku

kategorie	**význam**
query	Parametr dotazu je součástí URL a vztahuje se na celý požadavek. server/path**?name1=value1&name2=value2**
matrix	Maticový parametr je součástí URL a vztahuje se pouze na část cesty. server/pathPart**;name1=value1&name2=value2**/pathPart
path	Parametr cesty je součástí URL a představuje proměnnou část cesty. server/path**/pathParam**
form	Položka formuláře je obsažena v těle požadavku jako jedna z dvojic klíč=hodnota. Content-Type požadavku je application/x-www-form-urlencoded. **name1=value1&name2=value2**
header	HTTP hlavička požadavku. **MyHeader: headerValue**
cookie	Cookie je v případě HTTP požadavku obsažena v hlavičce Cookie. **Cookie: name=value; name2=value2; name3=value3**

Tabulka 5.5: Atributy restimpl.endpoint.response

responseId	
status	HTTP stavový kód
structure	struktura těla odpovědi, plné jméno třídy reprezentující tělo, popř. primitivní datový typ (znak pro jeho bytecode reprezentaci)
isArray	zda se jedná o pole či kolekci prvků s danou strukturou

Vlastnost restimpl.endpoint.responseparameter

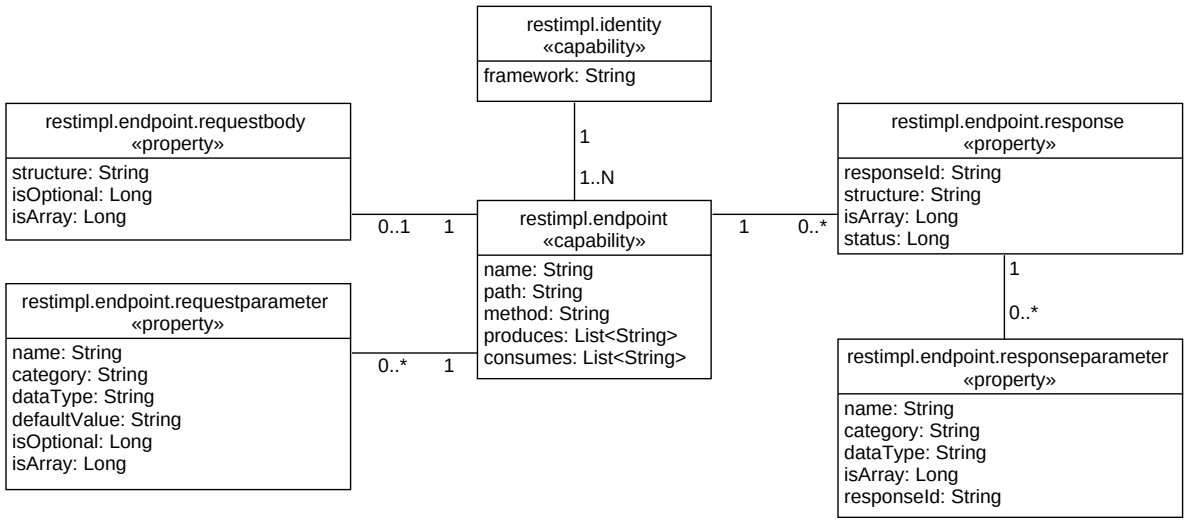
Odpověď stejně jako požadavek může obsahovat hlavičky, např. Set-Cookie nebo Location při vytvoření nového zdroje. Hlavičky odpovědi zachycuje property endpoint.responseparameter.

Celkové schéma

Na obrázku 5.1 je zobrazeno schéma vazeb mezi capability restimpl.endpoint a příslušnými properties.

Tabulka 5.6: Atributy restimpl.endpoint.responseparameter

name	**význam**
category	kategorie parametru, pouze hodnoty header nebo cookie
dataType	plné jméno třídy reprezentující tělo, popř. primitivní datový typ
isArray	zda se jedná o pole či kolekci prvků



Obrázek 5.1: Rozhraní REST služby v CRCE

5.3 Rekonstrukce rozhraní

V rámci této práce princip rekonstrukce rozhraní spočívá ve zpracování Java archivu obsahujícího implementaci REST služby, přičemž výstupem tohoto procesu jsou metadata datového modelu úložiště CRCE.

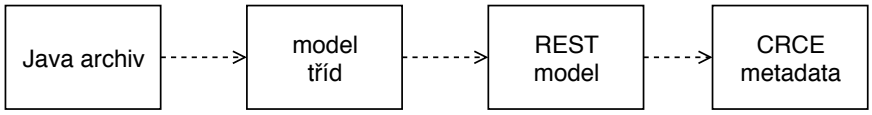
Byly zvažovány dva možné přístupy:

1. zpracování archivu a vytvoření reprezentace rozhraní ve formátu WADL s následnou úpravou modulu pro zpracování IDL dokumentů webových služeb, který je schopen zpracovat WADL a uložit do databáze příslušná metadata,
2. zpracování archivu a vytvoření reprezentace rozhraní ve formě metadat bez mezikroku spočívajícího ve vytváření WADL dokumentu.

První přístup by umožnil využití již existujícího indexeru od Pejřimovského, ale zároveň by obnášel nutnost jeho modifikace, protože je velice obecný a jeho vstupem je popisný dokument běžící webové služby. Použití tohoto přístupu s sebou nepřináší žádné

významné výhody, proto bylo rozhodnuto použít přístup druhý a vytvořit nový nezávislý indexer.

Program bude rozčleněn do tří fází. První fází je získání modelu tříd. Na základě jeho analýzy z něj následně bude vytvořen model REST rozhraní. Nakonec bude obraz rozhraní v objektech Javy převeden do podoby CRCE metadat a uložen do databáze.



Obrázek 5.2: Konverze datových modelů v průběhu zpracování programem

V následujícím textu je popsán požadovaný model tříd a návrh algoritmu pro rekonstrukci rozhraní.

5.3.1 Požadavky na vstupní archiv

Vstupní archiv pro rekonstrukci rozhraní musí být JAR nebo WAR soubor obsahující implementaci REST služby za využití frameworku Spring Web MVC nebo některého z JAX-RS frameworků, které byly vybrány na základě analýzy nejpoužívanějších technologií uvedené v kapitole 3.2.

Pro úspěšnou rekonstrukci rozhraní musí archiv splňovat několik předpokladů:

1. Webová služba musí být implementována jen pomocí jednoho z frameworků, ne jejich kombinací.
2. **Zdroj** je reprezentován třídou a označen anotací této třídy.
3. Pokud framework podporuje implementaci **podřízených zdrojů**, jsou tyto opět reprezentovány třídou a určeny tím, že obsahují metody odpovídající endpointům.
4. **Endpoint** představuje jedna metoda Javy, je označen anotací.
5. **Cesta k endpointu** je určena konkatenací cesty zdroje (třídy) a endpointu (metody), popř. více tříd a metod v případě podřízených zdrojů.
6. **HTTP metoda** endpointu je určena anotací nebo parametrem anotace metody, popř. známým výchozím nastavením.
7. **MIME typy požadavků a odpovědí** endpointu jsou určeny speciálními anotacemi, popř. hodnotami anotace. Tyto anotace mohou být přítomny u zdrojových tříd i u metod, výsledná hodnota je určena agregací dle vlastností použitého frameworku.

8. **Parametr požadavku** je určen parametrem metody nebo atributem tzv. beanu parametrů, přičemž parametry i atributy jsou označeny speciální anotací.

   - **Kategorie parametru** je určena názvem anotace parametru či atributu, popř. parametrem této anotace.
   - **Název parametru** je určen buď hodnotou anotace parametru či atributu, nebo přímo názvem parametru či atributu.
   - **Datový typ parametru** je určen datovým typem parametru či atributu.
   - **Výchozí hodnota parametru** je určena hodnotou anotace parametru či atributu.

9. **Tělo požadavku** je určeno buď samotným parametrem metody, nebo parametrem metody označeným příslušnou anotací.

10. **Odpověď** a všechny parametry s ní spojené jsou určeny buď přímo návratovým typem metody, nebo hodnotami atributů instance obecné třídy pro reprezentaci odpovědi. Tyto vlastnosti jsou definovány instrukcemi těla metody.

5.3.2 Požadavky na model tříd

Pro rekonstrukci rozhraní je potřeba z bytecode vytvořit model tříd. Je potřeba, aby obsahoval následující informace:

Třídy

Třídy webové služby lze rozdělit do těchto pěti kategorií:

- zdrojové třídy (hlavní a podřízené),
- další poskytovatelé (*providers* - filtry, mapovače obsahu atd.),
- doménové třídy představující těla zpráv,
- *beany* parametrů,
- ostatní.

V případě, že je třída tzv. *beanem parametrů* nebo doménovou třídou představující těla požadavků a odpovědí, potřebujeme znát všechny její atributy, a tudíž i atributy jejího předka.

V případě, že se jedná o zdrojové třídy, potřebujeme znát informace o jejich metodách.

Deklarace metod

Parametry endpointů představují parametry požadavků, těla požadavku a informace.

Návratový typ určuje odpověď. Modifikátor přístupu je jedním z ukazatelů, zda se vůbec jedná o endpoint.

Anotace

Velkou část informace o webových službách nesou anotace. Označují zdrojové třídy, endpointy, určují cestu požadavků, MIME typy požadavků a odpovědí, HTTP metody endpointů, označují parametry požadavků (včetně hlaviček a cookie), určují jejich kategorii, název, výchozí hodnotu i povinnost výskytu, dále označují beany parametrů, těla požadavků i speciální třídy sdružující informace o kontextu požadavku. Proto je potřeba získat název anotace i všechny její parametry.

Názvy atributů a formálních parametrů metod

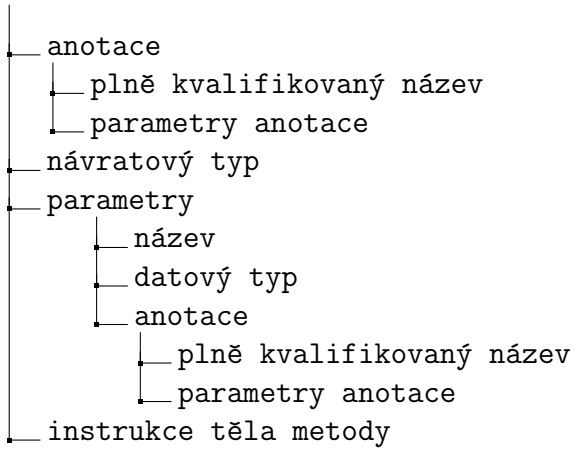
Kvůli vlastnosti Spring Web MVC, která umožňuje navázání parametru požadavku přímo na parametr metody na základě stejného jména, je potřeba z bytecode získávat i názvy atributů beanů parametrů a formálních parametrů metod.

Těla metod

Obě zkoumané technologie umožňují jako návratový typ endpointu nastavit objekt představující obecnou odpověď. Všechny vlastnosti odpovědi, jako její datový typ či struktura, návratový kód a hlavičky jsou pak nastavovány v těle metody. Z tohoto důvodu je potřeba získat posloupnost operací těla metody.

Model tříd by měl mít následující logickou strukturu.

```
třída
├── plně kvalifikovaný název
├── předek
├── rozhraní
├── anotace
│   ├── plně kvalifikovaný název
│   └── parametry anotace
├── atributy
│   ├── název
│   ├── datový typ
│   ├── modifikátor přístupu
│   └── anotace
│       ├── plně kvalifikovaný název
│       └── parametry anotace
└── metody
    ├── název
    ├── modifikátor přístupu
    └── deklarované výjimky
```



5.3.3 Existující řešení pro získání modelu tříd

JaCC a Javatypes

Jak už bylo krátce zmíněno v kapitole 4.3.2, JaCC vytáří vlastní model tříd definovaný v modulu javatypes. V těchto datových typech je zachyceno mnoho informací, v některých případech nepotřebných pro účely rekonstrukce rozhraní, např. seznam všech importů třídy. Na druhou stranu JaCC neuchovává sadu instrukcí představující tělo metod. Ve verzi poskytnuté v době psaní této práce (1.0.10) navíc nenačítá anotace tříd, metod, parametrů metod ani třídních atributů, které jsou pro pro rekonstrukci rozhraní klíčové. Zároveň nenačítá ani názvy formálních parametrů metod, což ale není zásadní nedostatek, protože názvy často nejsou v bytecode přítomny.

ASM Tree API

Model tříd nástroje JaCC vzniká konverzí z modelu tříd frameworku ASM, který je poskytován jeho rozšířením zvaným ASM Tree API. Tento model opět zachycuje mnoho informací, ale vyžaduje další zpracování. U metod jsou uvedeny jejich deskriptory a signatury, ale už ne konkrétní návratové typy a typy parametrů, natož jejich jména či příslušné anotace. Anotace parametrů jsou uvedeny v ve zvláštním poli zvaném visibleParameterAnnotations. Metoda obsahuje seznam instrukcí jejího těla.

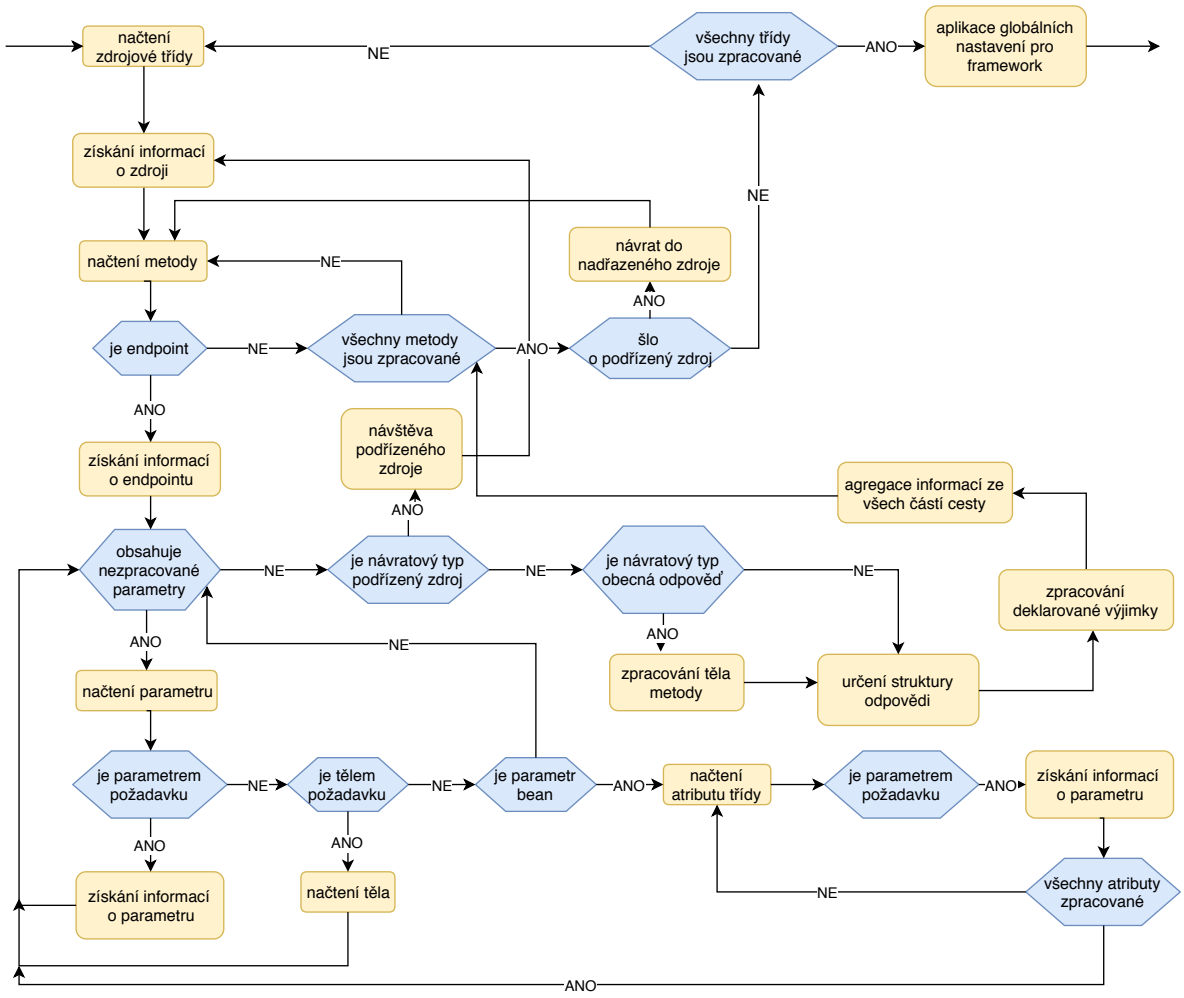
5.3.4 Rekonstrukce rozhraní z modelu tříd

Proces rekonstrukce rozhraní sestává z několika dílčích procedur, jmenovitě:

- identifikace zdrojů - tříd,
- identifikace dalších tříd potřebných pro rekonstrukci rozhraní,
- získání informací o zdrojích,
- identifikace endpointů uvnitř zdrojů,
- získání informací o endpointu,

- zpracování parametrů metody představující endpoint - získání parametrů požadavku a těla požadavku,
- zpracování návratového typu metody - získání odpovědi, popř. zpracování podřízeného zdroje,
- zpracování instrukcí těla metody - určení odpovědí, případně některých dalších parametrů požadavku,
- agregace atributů cesty zdroje a endpointu,
- aplikace výchozího nastavení pro daný framework v případě neurčení některých vlastností a jiných globálních hodnot pro všechny endpointy.

Jejich provázání je znázorněno na diagramu 5.3. Implementační detaily algoritmu jsou uvedeny v následující kapitole.



Obrázek 5.3: Proces rekonstrukce rozhraní

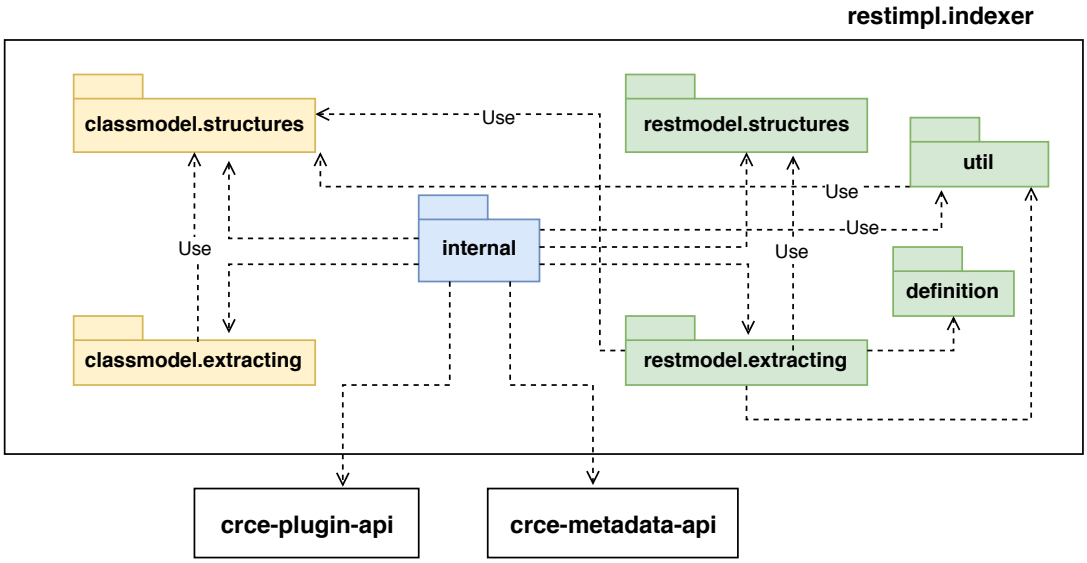
6 Implementace

Cílem práce je vytvořit nástroj, který bude schopen provést automatizovanou rekonstrukci rozhraní REST služby z Java archivu a toto rozhraní následně uloží v podobě metadat do komponentového úložiště. Tato kapitola popisuje implementaci nástroje. V první části jsou uvedeny detaily implementace nástroje jako celku, následují části věnované popisu získání modelu tříd z archivu, procesu rekonstrukce rozhraní REST ze získaného modelu a jeho převedení do výsledných metadat.

6.1 Indexer

Jak bylo uvedeno v kapitole 4, projekt CRCE je členěn na moduly. Rozšiřující moduly, které nejsou součástí jádra CRCE, se nacházejí v nadřazeném modulu crce-modules-reactor. Sem spadá i nově vytvořený modul/indexer crce-restimpl-indexer.

Obrázek 6.1 znázorňuje strukturu modulu a vztahy s ostatními moduly, se kterými přímo interaguje.



Obrázek 6.1: Kontext a struktura modulu crce-restimpl-indexer

Třídy lze rozdělit do tří skupin.

- **Balík internal**. V tomto balíku se nacházejí hlavní třídy modulu zajišťující aktivaci modulu a volání kódu tříd z ostatních balíků.
- **Třídy pro získání modelu tříd**. Jedná se třídy z balíku classmodel.structures a classmodel.extracting dále popsané v sekci 6.2.
- **Třídy pro rekonstrukci REST rozhraní.** Třídy z balíků restmodel.structures, restmodel.extracting, definition a util popsané v sekci 6.3.

Aktivaci modulu zajišťuje třída internal.Activator, která vytvoří instanci bundlu a předá mu OSGi kontext. Aktivátor musí být uveden v konfiguračním souboru osgi.bnd, ve kterém se nachází také informace o veřejných a soukromých balících modulu.

```
Bundle-Activator: ${bundle.namespace}.internal.Activator
Private-Package: ${bundle.namespace}.internal, ${bundle.namespace}.classmodel...
Export-Package: ${bundle.namespace}
```

Vstupním bodem pro indexaci je metoda index() třídy internal.RestimplResourceIndexer. V té je nejdříve načten model tříd, na kterém je provedena rekonstrukce REST rozhraní. Výsledkem rekonstrukce je množina endpointů. Na konec přichází na řadu třída RestimplMetadataManager, která z objektů reprezentujících endpointy vytvoří CRCE metadata a přiřadí je *resource* objektu představujícímu vstupní archiv.

6.2 Získání modelu tříd

V předchozí kapitole v sekci 5.3.2 byly uvedeny požadavky na model tříd potřebný pro rekonstrukci REST API. V sekci 5.3.3 pak byla představena existující řešení pro rekonstrukci modelu tříd z bytecode.

Model tříd ASM Tree API neobsahuje názvy formálních parametrů metod, je nepřehledný a vyžaduje další zpracování, např. extrakci parametrů metod z jejího deskriptoru či signatury. Nástroj JaCC rovněž nenačítá názvy formálních parametrů, navíc nezachycuje anotace a instrukce tvořící těla metod.

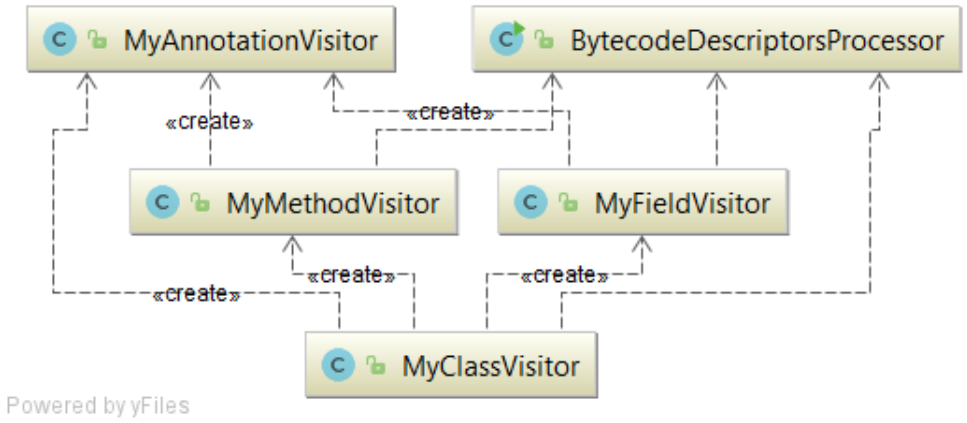
Z tohoto důvodu byl navržen vlastní model tříd a implementován nástroj pro jeho vytvoření z bytecode. Vše zajišťuje balík classmodel, který obsahuje podbalíky structures a extracting.

Pro zpracování bytecode je využit framework ASM a jeho způsob načítání tříd s využitím vlastní implementace třídy ClassVisitor a dalších dílčích *visitorů*.

- MyClassVisitor. V této třídě jsou získávány základní informace o třídě - její jméno, předek a názvy anotací. Jsou zde zpracovávány také základní informace o jejích členech z deskriptorů či signatur polí a deklarovaných metod.

- MyMethodVisitor. Slouží k zaznamenání důležitých operací těla metody a také ke zjištění názvů formálních parametrů metody. Ty jsou totiž ukládány jako první lokální proměnné metody. Pokud class soubor názvy formálních parametrů nezahrnuje, jsou místo nich přítomny proměnné s názvy arg0, arg1 apod.[1]
- MyFieldVisitor. Slouží k detailnímu zpracování atributu, konkrétně k nalezení jeho anotací.
- MyAnnotationVisitor. Slouží ke zpracování parametrů anotace.

Třídy vzájemně vytvářejí své instance, jak je znázorněno na obrázku **??**. Třída BytecodeDescriptorsProcessor zajišťuje zpracování deskriptorů a signatur pomocí regulárních výrazů na základě specifikace pro class formát[2].



Obrázek 6.2: Extrakce modelu tříd

V balíku structures jsou jednotlivé stavební prvky modelu odpovídající logickému modelu tříd uvedenému v předchozí kapitole, oddílu 5.3.2. , ClassStruct reprezentující třídu, Method metodu, Variable parametr metody, Field atribut třídy Annotation anotaci, DataType datový typ.

6.3 Rekonstrukce rozhraní z modelu tříd

Třídy zajišťující rekonstrukci rozhraní se nacházejí v balíku restmodel.extracting. Hlavní logika je obsažena ve třídách RestApiReconstructor a ClassModelProcessor. Algoritmus rekonstrukce je znázorněn v předchozí kapitole na diagramu 5.3 a v následující sekci jsou popsány jednotlivé jeho části.

[1] Vložení názvů do bytecode lze vynutit přepínačem -parameters kompilátoru pro Javu 8, popř. kompilací kódu pro účely debugování.

[2] viz `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html`

Konfigurace programu

Přes veškerou podobnost se zpracovávané frameworky liší v mnoha podstatných vlastnostech. Algoritmus pro rekonstrukci rozhraní je navržen tak, aby pracoval s oběma frameworky na základě jejich *definice rozhraní*. Definice rozhraní je představována třídou RestApiDefinition a je získána před zahájením rekonstrukce z konfiguračních souborů ve formátu YAML. Byly vytvořeny dva soubory s definicemi rozhraní - jeden pro specifikaci JAX-RS a druhý pro RESTovou část frameworku Spring Web MVC. V případě existence jiného frameworku fungujícho na podobném principu je možné pro jeho definici rozhraní vytvořit další konfigurační soubor a algoritmus s ním bude umět pracovat. Soubory se nacházejí v adresáři config/api přítomném v kořenovém adresáři projektu.

Dále existuje konfigurační soubor pro zpracování *deployment descriptoru* web.xml - config/dispatcher.yml, ve kterém je identifikován dispečer požadavků a jeho mapování na URL, popř. označení balíků obsahující zdroje.

Ukázka konfiguračního souboru s definicí API je uvedena v příloze A.

Zpracování deployment descriptoru

Pro zpracování *deployment descriptoru* slouží třída util.WebXmlParser. Ta v souboru vyhledává deklaraci dispečeru, jeho mapování a cestu k aktivním zdrojovým třídám a dalším poskytovatelům. Pokud specifikace zdrojů není nalezena, jsou za aktivní považované všechny nalezené zdrojové třídy.

Identifikace zdrojových tříd

Prvním krokem rekonstrukce rozhraní je nalezení zdrojových tříd. Zdrojová třída je vždy označena anotací z množiny resource_annotations.

Identifikace tříd pro mapování výjimek

Kromě zdrojů jsou při analýze tříd vyhledány také třídy pro mapování výjimek na odpovědi (na základě položky exception_handler). Tyto třídy jsou využity později při zpracovávání deklarací metod endpointů. Pokud metoda deklaruje vyhození výjimky, jejíž obslužná třída je k dispozici, je k množině možných odpovědí přidána i odpověď, na kterou je výjimka mapována.

Získání informací o zdroji

Zdroj u sebe může obsahovat globální nastavení MIME typů, část cesty a někdy i globální HTTP metodu. Související položky definice jsou produces, consumes a http_method.

Identifikace endpointů

Po identifikaci třídy jakožto zdroje jsou postupně zpracovány všechny její veřejné metody a je zjišťováno, jestli se jedná o endpointy. Endpointy jsou podobně jako zdroje

označeny anotacemi, tentokrát z množiny endpoint_annotations.

Zpracování parametrů metody

Algoritmus postupně prochází všechny parametry metody a určuje jejich druh. Vstupem do metody může být parametr požadavku (sem řadíme i hlavičky a cookie), tělo požadavku, pomocná třída nesoucí informace o kontextu požadavku nebo bean parametrů.
Na základě funkce parametru je každý z nich dále zpracován.

- **Parametr požadavku**. Je určena jeho kategorie, název, datový typ, výchozí hodnota a povinnost výskytu. Parametry cesty jsou vždy povinné. Související položky jsou endpoint_parameters, parameter_name, parameter_requirement, default_parameter_value. Někdy není kategorie parametru z anotací jasně určena a může být ovlivněna hodnotou consumes endpointu. Důvodem je fakt, že Spring nerozlišuje mezi parametry typu query a form a parametry s kategorií form jsou ve skutečnosti předány v těle požadavku.

- **Tělo požadavku.** To, jestli parametr představuje tělo požadavku, určuje položka konfigurace body. V té je uvedeno, zda je tělo označeno speciální anotací, nebo v případě, že anotací označeno není, jsou zde vyjmenovány vylučující anotace, které parametru přidávají jiný význam.

- **Bean parametrů**. Bean parametrů je třída, jejíž atributy představují parametry požadavků. Proměnná představující bean může být označena speciální anotací (viz parameter_bean_annotations). Samotná třída pak už žádnou anotací označena být nemusí. Způsob zpracování beanu se pro jednotlivé frameworky opět liší. Např. pro Spring musí mít parametry definované příslušné settery a mohou zde být uvedeny je parametry s kategoriemi query, form a path. Protože atribut představující parametr nemusí být označen žádnou anotací, která určí jeho kategorii, je potřeba zpracovat také cestu k endpointu, která může obsahovat proměnné, tzv. parametry cesty. Výchozí hodnoty parametrů z beanu mohou mít jiné výchozí vlastnosti než parametry získané z parametrů metody, např. může být jiná výchozí povinnost výskytu. Související položky: fieldParamAnnotations, fieldParamSetterRequired, popř. všechny položky související s parametry metody.

- **Objekt nesoucí informace o kontextu**. Metodě může být předána instance třídy obsahující informace o kontextu, např. hlavičkách, URI, popř. celý požadavek či odpověď.

Zpracování návratového typu

Návratový typ metody může v kontextu endpointu REST služby nést velmi rozdílnou informaci:

- **Podřízený zdroj**. Pokud se framework podporuje podřízené zdroje (položka subresources) a návratovým typem metody je známá třída obsahující endpointy, jedná se o podřízený zdroj.

- **Obecná třída pro odpověď**. Pokud je návratovým typem metody obecná třída reprezentující odpověď, nezjistíme o struktuře ani statusu HTTP odpovědi nic a musíme následně zpracovat tělo metody.
- **Jiná třída.** Pokud je návratovým typem jiná třída než výše uvedené, je zpracována jako odpověď. Je určena její struktura a příznak, zda se jedná o pole, resp. kolekci.

Zpracování instrukcí těla metody

Ke zpracování těla metody, tzn. posloupnosti instrukcí dochází v případě, že je návratovým typem obecná třída reprezentující HTTP odpověď. Toto zajišťuje třída MethodBodyInterpreter, která představuje velmi zjednodušený interpret instrukcí. V těle metody jsou vyhledávána volání metod pro nastavování těl a stavových kódů odpovědí. Tento algoritmus funguje na základě heuristiky předpokládající základní scénáře pro vytváření instancí tříd pro odpovědí představené v sekci 3.2 o REST frameworcích.

Některé informace z bytecode však získat nelze. Příkladem je vrácení generické kolekce jakožto odpovědi. Při kompilaci generických typů dochází k tzv. vymazání typů, což znamená, že místo seznamu prvků konkrétní třídy se v bytecode nachází jen obecný seznam objektů. Jediným případem, kdy se informace o generických typech objeví v bytecode jsou tzv. signatury.

Položky definice: status_setting_methods, entity_setting_methods, status_fields, cookie_class, cookie_setting_method, header_setting_method.

Agregace atributů cesty zdroje a endpointu

Agregace atributů cesty zdroje a endpointu spočívá v konkatenanci částí cesty, nastavení správných hodnot produces a consumes, přičemž hodnoty u zdroje představují globální hodnoty v případě nespecifikování u endpointu. V opačném případě se hodnoty přepíší. Pokud lze uvádět HTTP metody u zdroje, jsou tyto metody přidány k metodám u endpointů.

Aplikace globálních hodnot pro framework

Posledním krokem je aplikace výchozích hodnot specifických pro daný framework, např. nastavení HTTP metod v případě frameworku Spring či přidání URL prefixů k cestám k endpointům.

6.4 Shrnutí kapitoly

V implementační části práce byl vytvořen indexer, který dovede zpracovat vstupní archiv v implementací REST služby, analyzovat ho a zrekonstruovat z něj rozhraní REST služby. Model rozhraní je nakonec převeden do podoby metadat CRCE.

Algoritmus umí pracovat s anotačními frameworky fungujícími na principech JAX-RS a Spring Web MVC a je programátorsky konfigurovatelný.

Analyzátor má několik známých omezení:

- Vytváření modelu tříd z bytecode nemusí fungovat přesně v případě složitých signatur, např. při kombinaci obsahující deklaraci výjimek, lambda výrazy a další prvky. V takovém případě většinou nefunguje správná identifikace formálních parametrů metody.
- Analyzátor není schopen získat skutečné názvy parametrů požadavku, jestliže nejsou uvedeny jako hodnota anotace a v bytecode nejsou přítomny názvy formálních parametrů metod.
- Dále není schopen správně určit strukturu odpovědi, jestliže je nastavena v těle metody jako typový parametr kolekce z důvodu vymazání typů při kompilaci zdrojového kódu.
- Nejsou rozpoznávány parametry požadavku zpracovávané pouze ve *filtrech* bez deklarace v metodě endpointu. Tyto parametry jsou získávány programově v těle metod a existuje příliš mnoho možností pro jejich získání. Zároveň není snadné určit, který filtr se vztahuje na konkrétní endpoint. Toto by vyžadovalo komplexní interpretaci těl metod s vyhodnocováním podmínek a dalších instrukcí, což by prakticky obnášelo vytvoření kompletního interpretu bytecode.
- Parametry požadavku získávané dynamicky v těle metody (tzn. ne deklarované v podobě formálních parametrů) nejsou zpracovávány ze stejného důvodu jako v předchozím bodě.
- Konfigurace služby ve smyslu registrace zdrojových tříd a definice prefixu URL všech požadavků jsou zpracovávány jen v případě, že jsou uvedeny v souboru web.xml. Dynamická konfigurace v kódu není zpracovávána.
- Není řešeno získávání popisných schémat pro třídy modelu předávané v tělech požadavků a odpovědí, z toho důvodu, že v případě CRCE jsou všechny třídy archivu přítomny v úložišti, a tak stačí uchovávat jen odkaz na ně.

7 Ověření funkčnosti

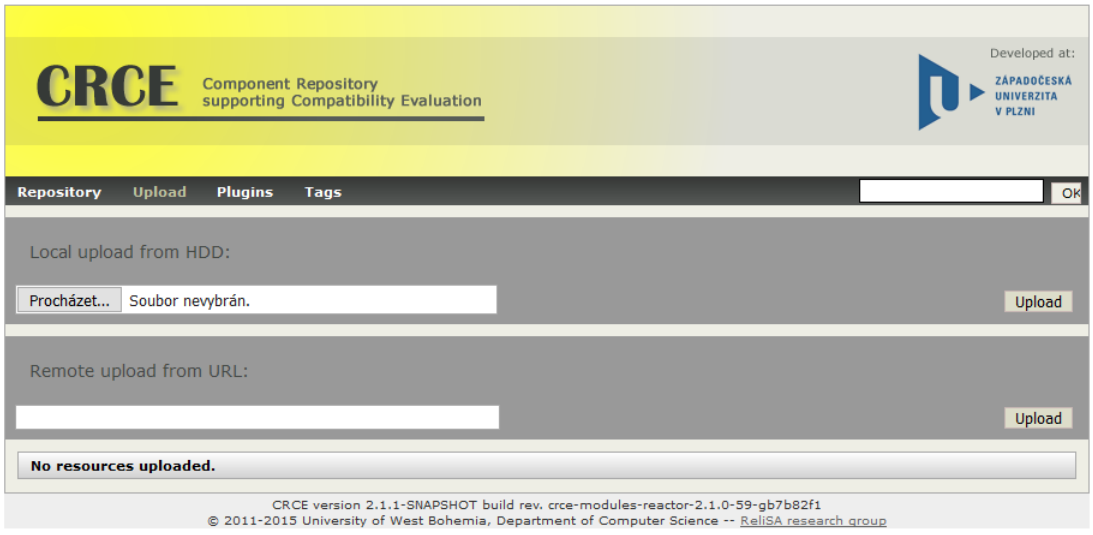
Tato kapitola se dělí na tři části. V první je uveden stručný návod pro nahrání archivu do úložiště a zobrazení metadat nalezených vytvořeným indexerem, ve druhé jsou uvedeny výstupy aplikace indexeru pro rekonstrukci rozhraní na testovací archivy a ve třetí se nachází shrnutí výsledků testování.

7.1 Použití indexeru

Indexer lze otestovat na běžící instanci úložiště CRCE s aktivovaným modulem CRCE - Rest implementation Indexer. Podrobné instrukce ke zprovoznění CRCE a aktivaci modulu jsou k dispozici v uživatelském manuálu, který je přílohou této diplomové práce.

7.1.1 Nahrání archivu do úložiště

Archiv s Java bytecode se do úložiště nahraje přes webové uživatelské rozhraní na záložce Upload.



Obrázek 7.1: Nahrání archivu přes webové rozhraní

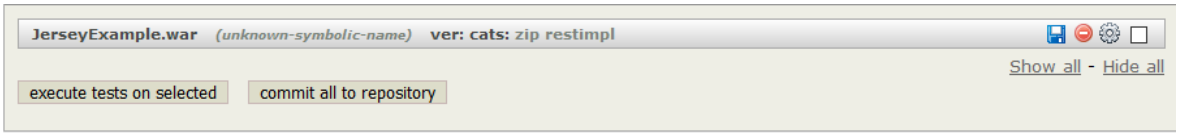
Po výběru archivu a stisknutí tlačítka Upload je archiv nahrán do vyrovnávací paměti a jsou na něj postupně aplikovány aktivní indexery. Pokud je archiv rozpoznán jako

implementace webové služby typu REST, je mu přiřazena kategorie restimpl.

Po rozkliknutí názvu archivu v seznamu jsou zobrazena příslušná metadata nalezená indexery, konkrétně jeho schopnosti (*capabilities*), požadavky (*requirements*) a vlastnosti (*properties*). Vlastnosti příslušné schopnostem a požadavkům zde uvedeny nejsou. REST endpointy jsou uvedeny v seznamu Capabilities jako položky typu restimpl.endpoint.

Obrázek 7.2: Zobrazení metadat ve vyrovnávací paměti

Uložení metadat do databáze proběhne až po kliknutí na tlačítko commit all to repository v dolní části obrazovky.



Obrázek 7.3: Uložení dat do databáze

7.1.2 Zobrazení uložených metadat

Na záložce Repository je k dispozici seznam všech archivů (*resources*) nacházejících se v databázi, pro jednotlivé položky jsou však zobrazeny jen základní informace.

Podrobné informace o nahraném archivu je možné získat pomocí webových služeb.

Seznam artefaktů v úložišti je přístupný na relativní cestě /metadata, v případě instance CRCE běžící na lokálním stroji na URL

http://localhost:8080/rest/v2/metadata/.

Stejně jako v grafickém webovém rozhraní jsou zde jen základní informace o artefaktech.

V obou případech je u artefaktu zobrazen jeho unikátní identifikátor, který lze použít jako parametr id v další webové službě na URL

http://localhost:8080/rest/v2/metadata/{id}.

Zde se nachází podrobný popis artefaktu se všemi jeho schopnostmi, požadavky a vlastnostmi. Výchozí formát je XML, při přidání hlavičky požadavku Accepts s hodnotou application/json, je možné data zobrazit ve formátu JSON.

Zde je uvedena ukázka výstupu:

```
<capability uuid="8ad1bec9-48ca-4c7a-ba33-d09829430d23"
namespace="restimpl.identity">
  <attribute name="name" type="java.lang.String" value="jaxrs" />
  <capability uuid="372cb806-f65c-4dce-8d3d-386e15119a4c"
  namespace="restimpl.endpoint">
    <attribute name="path" type="java.util.List"
    value="[/rest/test/paramReallyRequired]" />
    <attribute name="method" type="java.util.List" value="[GET]" />
    <attribute name="name" type="java.lang.String"
    value="resources/TestResource.paramReallyRequired" />
    <attribute name="produces" type="java.util.List" value="[application/xml]" />
    <attribute name="consumes" type="java.util.List"
    value="[application/json]" />
    <property uuid="1d41552b-7356-4cf6-87a6-098862e9e4b1"
     namespace="restimpl.endpoint.requestparameter">
      <attribute name="datetype" type="java.lang.String" value="java/lang/String" />
      <attribute name="name" type="java.lang.String" value="param" />
      <attribute name="isArray" type="java.lang.Long" value="0" />
      <attribute name="isOptional" type="java.lang.Long" value="1" />
      <attribute name="category" type="java.lang.String" value="QUERY" />
    </property>
    ...
```

7.2 Testování

Pro ověření funkčnosti byly vytvořeny tři testovací archivy pokrývající široké spektrum možností konstrukce endpointů REST služeb v jazyce Java. Dva testovací archivy obsahují implementaci webových služeb pomocí frameworků splňující JAX-RS, konkrétně Jersey a RESTEasy, třetí obsahuje implementaci pomocí frameworku Spring Web MVC. Všechny archivy jsou zkompilované pomocí Javy 8 a jsou k dispozici na přiloženém CD spolu se zdrojovými kódy.

Následuje popis testovacích archivů s ukázkami případů rekonstrukce konkrétních endpointů. V testovacích případech je zahrnuto získání všech charakteristik endpointů REST služby s ohledem na nejčastější způsob jejich implementace v REST frameworcích, a to konkrétně následující:

- HTTP metoda/y,
- cesta/y (hierarchická část URL),
- MIME typ/y těla požadavku / odpovědi,
- struktura těla požadavku,
- parametry požadavku deklarované jako parametry metody,
- parametry jako JavaBean ,
- povinnost výskytu parametrů ,
- parametry odpovědi,
- status odpovědi,
- mapování výjimek na odpovědi,
- struktura těla odpovědi, získaná také interpretací těla odpovědi,
- zpracování podřízeného zdroje,
- agregace vlastností zdroje a endpointu.

7.2.1 Jersey

První testovací archiv obsahuje implementaci REST služby prostřednictvím frameworku Jersey verze 2.26 od společnosti Glassfish. V souboru web.xml jsou definovány balíky poskytovatelských tříd (*providers*) exception a resources. Servlet představující dispečer požadavků je mapován na URL /rest/*.

Archiv obsahuje čtyři zdrojové třídy nejvyšší úrovně (HelloWorldService, ArticlesResource, TestResource, TestGlobalSettingsResource) a jeden podřízený zdroj (CommentsResource), všechny uvedeny v balíku resources.

Dále obsahuje jednu třídu pro mapování výjimek na odpovědi, exception/DataNotFoundExceptionMapper.

Celkem je v archivu obsaženo 39 endpointů.

Ukázka 7.1: Základní příklad endpointu

```
@Path("/articles")
public class ArticlesResource {
   ...
    @GET
    @Produces({ MediaType.APPLICATION_JSON})
    public List<Article> getArticlesJSON(@QueryParam("offset")
                                         @DefaultValue("0") int offset,
                                         @QueryParam("limit")
                                         @DefaultValue("10") int limit) {
        return articlesDAO.getArticles(offset, limit);
    }
```

Očekáváná struktura endpointu:

```
endpoint
├── name: resources/ArticlesResource.getArticlesJSON
├── paths: /rest/articles
├── httpMethods: GET
├── produces: application/json
├── requestParameters
│   ├── parameter
│   │   ├── name: offset
│   │   ├── category: QUERY
│   │   ├── dataType: int
│   │   ├── isOptional: true
│   │   ├── isArray: false
│   │   └── defaultValue: 0
│   └── parameter
│       ├── name: limit
│       ├── category: QUERY
│       ├── dataType: int
│       ├── isOptional: true
│       ├── isArray: false
│       └── defaultValue: 10
└── responses
    └── response
        ├── structure: model/Article
        └── isArray: true
```

Cesta k endpointu má prefix získaný z hodnoty mapování dispečeru požadavků uvedeného ve web.xml, následuje cesta ke zdroji. Jsou správně rozpoznány dva parametry

dotazu, včetně jejich názvu a výchozích hodnot. Protože v JAX-RS jsou parametry ve výchozí hodnotě nepovinné, je nastaven příznak isOptional. Zároveň je identifikována možná odpověď jako kolekce článků.

Vstupem následujícího endpointu je třída představující bean parametrů.

Ukázka 7.2: Parametry jako JavaBean

```
@POST
@Path("childBean/{path}")
public String childBean(@BeanParam ChildParamBean bean) {
        return bean.toString();
}
public class ChildParamBean extends ComplexParamBean {
    @QueryParam("child") private int child;
    public int getChild() {return child;}
    public void setChild(int child) {this.child = child;
}

public class ComplexParamBean {
    @QueryParam("query") @DefaultValue("0") public int query;
    @PathParam("path") @DefaultValue("10") public int path;
    @CookieParam("cookie") public String cookie;
    @HeaderParam("header")  public String header;
}
```

Tato třída obsahuje parametr dotazu child, další parametry obsahuje také její předek ComplexParamBean. V případě JAX-RS jsou jako parametry beanu rozpoznány všechny parametry označené speciální anotací a přístupné pomocí veřejného modifikátoru nebo *setteru*.

Očekávaný výsledek:

```
requestParameters
├── parameter
│   ├── name: header
│   ├── category: HEADER
│   ├── dataType: String
│   ├── isOptional: true
│   └── isArray: false
└── parameter
    ├── name: cookie
    ├── category: COOKIE
    └── dataType: String
```

```
    |__isOptional: true
    |__isArray: false
  |__parameter
    |__name: query
    |__category: QUERY
    |__dataType: int
    |__isOptional: true
    |__isArray: false
    |__defaultValue: 0
  |__parameter
    |__name: path
    |__category: PATH
    |__dataType: int
    |__isOptional: false
    |__isArray: false
    |__defaultValue: 10
  |__parameter
    |__name: child
    |__category: QUERY
    |__dataType: int
    |__isOptional: false
    |__isArray: false
```

Je rozpoznáno 5 parametrů. Parametr cesty je vždy označen za povinný. Výsledek odpovídá předpokladu. Povinnost výskytu parametrů jiných než parametru cesty lze ovlivnit prostřednictvím anotací JSR 303.

Ukázka 7.3: Povinnost výskytu parametrů

```
@GET
@Path("paramReallyRequired")
  public String paramReallyRequired(@NotNull @QueryParam("param")
                                                    String param) {
  return "paramReallyRequired was called with param " + param;
}
```

Očekávaný výsledek:

```
requestParameters
  |__parameter
    |__name: param
    |__category: QUERY
    |__dataType: String
```

```
    ├── isOptional: false
    └── isArray: false
```

Výsledek odpovídá předpokladu.

Následující metoda endpointu deklaruje vyhození výjimky, jejíž obslužná třída patří do balíku poskytovatelů registrovaných ve web.xml.

Ukázka 7.4: Odpověď mapovaná na výjimku

```
@PATCH
public String exception(@FormParam("file") String name,
                        @FormParam("value") int value)
                        throws DataNotFoundException {

  return String.valueOf("uploaded");
}

@Provider
public class DataNotFoundExceptionMapper
implements ExceptionMapper<DataNotFoundException> {
    @Override
    public Response toResponse(DataNotFoundException e) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}
```

Očekávaný výsledek:

```
responses
├── response
│   ├── structure: String
│   └── isArray: false
└── response
    ├── status: 404
    ├── structure: null
    └── isArray: false
```

S endpointem jsou svázány dvě známé podoby odpovědi. Výsledek odpovídá předpokladu.

Už na předchozím příkladu byla vidět ukázka rekonstrukce HTTP odpovědi z instrukcí těla metody. Následující příklad je o něco složitější:

Ukázka 7.5: Získání odpovědí interpretací těla metody

```
@Path("/responsesShort")
@GET
public Response responsesShort() {
   int random = new Random().nextInt(2);
   Response defaultResponse = createResponse();
   if (random == 0) {
     Response response = Response.accepted().header("myHeader", 111).build();
     Response response2 = Response.status(Response.Status.BAD_REQUEST)
                                  .build();
     return response;
   } else if (random == 1) {
       return Response.ok("Hi there.").build();
   } else {
      return defaultResponse;
   }
 }
private Response createResponse() {
   // CONFLICT: 409
   return Response.status(Response.Status.CONFLICT).entity(new Comment())
                  .build();
}
```

Očekávaný výsledek:

```
responses
├── response
│   ├── status: 202
│   └── parameters
│       └── parameter
│           ├── name: myHeader
│           ├── category: HEADER
│           └── dataType: int
├── response
│   ├── status: 409
│   ├── structure: model/Comment
│   └── isArray: false
└── response
    ├── status: 200
    ├── structure: String
    └── isArray: false
```

Algoritmus správně rozpoznal tři možné odpovědi. Pokud je známá implementace vo-

lané metody (createResponse()), která vrací objekt představující obecnou odpověď, je rekurzivně prozkoumáno i tělo této metody a výsledek zahrnut do množiny možných odpovědí. Hodnota proměnné response2 je ignorována, protože není možnou návratovou hodnotou. Výsledek odpovídá předpokladu.

Archiv obsahuje další testovací případ identifikovaný jako endpoint s názvem resources/TestResource.responses, který identifikuje 9 různých odpovědí definovaných v těle metody.

Ukázka 7.6: Agregace vlastností zdrojové třídy a endpointu

```
@Path("global")
@Produces(MediaType.APPLICATION_XML)
public class TestGlobalSettingsResource {
  @Path("object")
  @Produces(MediaType.APPLICATION_JSON)
  @GET
  public User object() {
    return new User("Lil", "Anderson");
  }
}
```

Ačkoliv zdrojová třída deklaruje, že MIME typy odpovědí jejích endpointů jsou application/xml, endpoint tuto hodnotu může přepsat. Očekávaný výsledek:

```
endpoint
├── name: resources/TestGlobalSettingsResource.object
├── paths: /rest/global/object
├── httpMethods: GET
├── produces: application/json
├── requestParameters
└── responses
    └── response
        ├── structure: model/User
        └── isArray: false
```

Výsledek odpovídá předpokladu.

Následuje ukázka zpracování endpointů nacházejících se v podřízeném zdroji.

Ukázka 7.7: Podřízený zdroj

```
@Path("/articles")
public class ArticlesResource {
 @Path("/{articleId}/comments")
    public CommentsResource getComments() {
        return new CommentsResource();
    }
}
public class CommentsResource {
    @GET
    public List<Comment> getAllComments(@PathParam("articleId")
                                                    long articleId) {
        return commentsDAO.getAllComments(articleId);
    }
}
```

Očekávaný výsledek:

```
endpoint
├── name: resources/CommentsResource.getAllComments
├── paths: /rest/articles/{articleId}/comments
├── httpMethods: GET
├── produces: application/xml
├── requestParameters
│   └── parameter
│       ├── name: articleId
│       ├── category: PATH
│       ├── dataType: long
│       ├── isOptional: false
│       └── isArray: false
└── responses
    └── response
        ├── structure: model/Comment
        └── isArray: true
```

Výsledek odpovídá předpokladu.

7.2.2 RESTEasy

Další archiv je implementován pomocí frameworku RESTEasy verze 3.0.16. Jeho struktura velmi podobná struktuře testovacího archivu pro Jersey. Používané zdrojové třídy

jsou zaregistrovány programově ve třídě app/MyApplication. Archiv obsahuje 28 endpointů.

U RESTEasy uvedeme ukázku konstrukce (a následné rekonstrukce) endpointu, který umožňuje nahrávání souborů.

Ukázka 7.8: Binární soubor jako vstup endpointu

```
@POST
@Path("/uploadFile2")
@Consumes("multipart/form-data")
public String uploadFile2(@MultipartForm FileUploadForm form) {
  return String.valueOf("uploaded");
}
```

Očekávaný výsledek:

```
endpoint
├── name: resources/TestResource.uploadFile2
├── paths: /rest/test/uploadFile2
├── httpMethods: POST
├── consumes: multipart/form-data
├── body
│   ├── structure: util/FileUploadForm
│   ├── isOptional: true
│   └── isArray: true
└── responses
    └── response
        ├── structure: String
        └── isArray: false
```

7.2.3 Spring Web MVC

Archiv vytvořený pomocí Spring Web MVC je implementován pomocí frameworku Spring Boot verze 1.5.9.

Obsahuje pět zdrojových tříd, zde nazývaných *kontrolery*: GreetingController, ArticlesController, CommentsController, TestController, TestGlobalSettingsController.

Framework Spring umožňuje u endpointu uvádět více HTTP metod, nebo naopak žádnou. Není-li uvedena žádná, jsou akceptovány všechny metody povolené frameworkem.

Ukázka 7.9: HTTP metody ve Spring Web MVC

```
@RequestMapping("/testNoMethod")
  public ResponseEntity testNoMethod() {
     return ResponseEntity.ok("test no method OK");
}
```

```
endpoint
├── name: app/TestController.testNoMethod
├── paths: test/testNoMethod
├── httpMethods: GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS
├── produces: application/json
└── responses
    └── response
        ├── status: 200
        ├── structure: String
        └── isArray: false
```

Ukázka 7.10: HTTP metody ve Spring Web MVC 2

```
@RequestMapping(value = "moreMethods2", method = {RequestMethod.GET,
                                                  RequestMethod.DELETE})
public String moreMethods2() {
   return "OK";
}
```

Očekávaný výsledek:

```
endpoint
├── name: app/TestController.moreMethods2
├── paths: test/moreMethods2
├── httpMethods: DELETE, GET
├── produces: application/json
└── responses
    └── response
        ├── structure: String
        └── isArray: false
```

Ve Springu jsou všechny parametry, které nejsou součástí beanu, implicitně vyžadované. Povinnost výskytu však lze ovlivnit hodnotou anotace. Název hlavičky `h` je získán přímo

z názvu formálního parametru.

Ukázka 7.11: Parametry ve Springu

```
@GetMapping("required")
public String requiredParam(@RequestParam(name="r", required=false)  int r,
                            @RequestHeader int h) {
  return String.valueOf(r) + String.valueOf(h);
}
```

Očekávaný výsledek:

```
parameters
├── parameter
│   ├── name: r
│   ├── category: QUERY
│   ├── dataType: int
│   ├── isOptional: true
│   └── isArray: false
└── parameter
    ├── name: h
    ├── category: HEADER
    ├── dataType: int
    ├── isOptional: false
    └── isArray: false
```

Výsledek odpovídá předpokladu.

Jestliže je hodnota consumes u endpointu application/x-www-form-urlencoded, je parametrům požadavku (RequestParam) přirazena kategorie FORM.

Ukázka 7.12: Parametry ve Springu 2

```
@PostMapping(value = "formParams",
             consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public String formParams(@RequestParam(name = "name", defaultValue = "unknown")
                         int param) {
     return String.valueOf(param);
}
```

Očekávaný výsledek:

```
parameters
 ├─ name: name
 ├─ category: FORM
 ├─ dataType: int
 ├─ isOptional: false
 └─ isArray: false
```

Výsledek odpovídá předpokladu.

Parametry v beanu nejsou označeny anotacemi a jsou všechny považovány za parametry dotazu (popř, formulářové) nebo parametry cesty, jestli se vyskytují jako proměnné v některé z cest k endpointu. Na rozdíl od parametrů metody jsou implicitně nepovinné a nelze jim nastavit výchozí hodnotu.

Ukázka 7.13: Parametry jako JavaBean

```
@PostMapping(path = "childBean/{path}")
public String childBean(ChildParamBean bean) {
  return bean.toString();
}
-------------------------------------------------------
public class ChildParamBean extends ComplexParamBean {
    private int child;
    public void setChild(int child) {this.child = child;
}
public class ComplexParamBean {
    private String cookie;
    private String header;
    private int path;
    private int query;
    /* setters for all but query*/
}
```

Třída ChildParamBean dědí od ComplexParamBean, jehož parametr query nemá příslušný *setter*.

Očekávaný výsledek:

```
requestParameters
 └─ parameter
    ├─ name: header
    ├─ category: QUERY
    ├─ dataType: String
    └─ isOptional: true
```

```
    isArray: false
  parameter
    name: cookie
    category: QUERY
    dataType: String
    isOptional: true
    isArray: false
  parameter
    name: path
    category: PATH
    dataType: int
    isOptional: false
    isArray: false
  parameter
    name: child
    category: QUERY
    dataType: int
    isOptional: true
    isArray: false
```

Parametr cesty je stejně jako u JAX-RS vždy povinný, jinak endpoint není nalezen. U uvedeného endpointu byly nalezeny čtyři parametry a byla jim nastavena správná kategorie. Výsledek odpovídá předpokladu.

7.3 Shrnutí kapitoly

Byly provedeny testy analyzátoru pokrývající množinu nejběžnějších případů konstrukce REST služeb v platformě Java. Analyzátor funguje v rozsahu daném návrhem a známými omezeními (viz předchozí kapitoly, oddíly 5.3.1 a 6.4). Je schopen zrekonstruovat rozhraní REST služby na základě informací o stavbě tříd a znalostech vlastností frameworků. V základní podobě dovede také interpretovat instrukce těla metody za účelem získání množiny možných HTTP odpovědí endpointu.

Na přiloženém CD jsou k dispozici tři výše popsané testovací archivy s implementacemi REST služeb v platformě Java.

8 Závěr

Práce navazuje na diplomovou práci Davida Pejřimovského [Pej15], který se zabýval analýzou instancí obecných webových služeb a získáváním jejich metadat z veřejně dostupných popisných dokumentů.

Předmětem této práce nebyla analýza běžících webových služeb, ale analýza Java archivů obsahujících implementaci služeb. Konkrétně se jednalo o služby typu REST.

V rámci práce byl navržen model rozhraní REST služeb a implementován algoritmus pro jeho rekonstrukci z archivů obsahujících implementace REST služeb pomocí frameworků splňujících specifikaci JAX-RS a frameworku Spring Web MVC, což jsou v současnosti nejpoužívanější technologie pro implementaci REST služeb v platformě Java.

Byl vytvořen nový modul komponentového úložiště CRCE vyvíjeného na Katedře informatiky a výpočetní techniky na Západočeské univerzitě. Modul funguje jako tzv. *indexer*, jehož kód je aktivován při nahrávání JAR nebo WAR komponenty do úložiště. Modul provádí analýzu bytecode a na jejím základě rekonstruuje rozhraní REST služby.

Výstupem indexeru je reprezentace rozhraní REST služby v podobě metadat o archivu. Předpokládá se další využití těchto metadat pro analýzy a navazující výzkumné práce zabývající se problematikou ověřování kompatibility komponent. Indexer dále poskytuje prostor pro vylepšení, např. v oblasti vyhledávání konstrukcí pro tvorbu rozhraní pomocí interpretu instrukcí.

Bibliografie

[Amb16] Jan Ambrož. "Výkonnostní a paměťová optimalizace nástroje JaCC". Dipl. Západočeská univerzita v Plzni, 2016.

[Api] Apiary. *API Blueprint*. URL: https://apiblueprint.org/ (cit. 02. 12. 2017).

[BJ15] Přemek Brada a Kamil Ježek. "Repository and meta-data design for efficient component consistency verification". In: (2015).

[BSB08] Bryan Basham, Kathy Sierra a Bert Bates. *Head First Servlets and JSP*. O'Reilly Media, 2008. ISBN: 978-0-596-51668-0.

[Cam] *Apache Camel*. URL: http://camel.apache.org (cit. 31. 01. 2018).

[Crc] *CRCE wiki*. URL: https://app.assembla.com/spaces/crce/wiki (cit. 30. 01. 2018).

[DIKR] CSC. Doc. Ing. Karel Richta. *Metody integrace aplikací*. URL: https://edux.fit.cvut.cz/oppa/BI-SI1/prednasky/BI-SI1-P12m.pdf.

[Fow10] Martin Fowler. *Richardson Maturity Model*. 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (cit. 07. 10. 2017).

[Fra] *Framework*. URL: https://cs.wikipedia.org/wiki/Framework (cit. 31. 01. 2018).

[FT00] Roy T Fielding a Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[Jaxa] *Creating a RESTful Root Resource Class*. 2017. URL: https://docs.oracle.com/javaee/7/tutorial/jaxrs002.htm (cit. 07. 10. 2017).

[Jaxb] *JAX-RS: Java^{TM} API for RESTful Web Services*. Oracle Corporation, 2013.

[Jaxc] *Mapping XML types to Java types*. URL: https://en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding (cit. 24. 01. 2018).

[Mox] *EclipseLink MOXy*. URL: http://www.eclipse.org/eclipselink/#moxy (cit. 31. 01. 2018).

[Pas13] Michael Pasternak. *The Markup Conference*. 2013. URL: http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html (cit. 02. 12. 2017).

[Pej15] David Pejřimovský. "Vytváření a ukládání popisu webových služeb v úložišti CRCE". Dipl. Západočeská univerzita v Plzni, 2015.

[Ram] *RAML*. URL: `https://raml.org/` (cit. 02. 12. 2017).

[Resa] *RESTEasy*. URL: `http://resteasy.jboss.org` (cit. 31. 01. 2018).

[Resb] *Restlet*. URL: `https://restlet.com/open-source/documentation/user-guide/2.3/introduction/overview` (cit. 20. 01. 2018).

[Ric16] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, 2016. ISBN: 978-1-491-94161-4.

[RR07] Leonard Richardson a Sam Ruby. *RESTful Web Services*. O'Reilly Media, 2007. ISBN: 978-0-596-5296-0.

[Soc] Jiří Sochor. *ÚVT MU*. URL: `http://webserver.ics.muni.cz/zpravodaj/articles/61.html` (cit. 30. 01. 2018).

[Spra] *Spring OXM*. URL: `https://docs.spring.io/spring-ws/sites/1.5/reference/html/oxm.html` (cit. 31. 01. 2018).

[Sprb] *Spring Web MVC*. 2017. URL: `https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc` (cit. 07. 10. 2017).

[Swa] Swagger. *OpenAPI Specification*. URL: `https://swagger.io/specification/` (cit. 02. 12. 2017).

[W3C04] W3C. *Web Services Architecture*. 2004. URL: `https://www.w3.org/TR/ws-arch/#whatis` (cit. 02. 12. 2017).

[W3C07] W3C. *Web Services Description Language (WSDL) Version 2.0*. 2007. URL: `https://www.w3.org/TR/2007/REC-wsdl20-20070626/` (cit. 02. 12. 2017).

[W3C09] W3C. *Web Application Description Language*. 2009. URL: `https://www.w3.org/Submission/wadl/` (cit. 02. 12. 2017).

Seznam zkratek

- **API** - Application Programming Interface
- **CRCE** - Component Repository supporting Compatibility Evaluation
- **CRUD** - Create, Replace, Update, Delete
- **DAO** - Data Access Object
- **HATEOAS** - Hypertext As The Engine Of Application State
- **HTTP** - HyperText Transfer Protocol
- **IDL** - Interface Definition Language
- **JaCC** - Java Class Comparator
- **JAR** - Java ARchive
- **Java EE** - Java Enterprise Edition
- **Java SE** - Java Standard Edition
- **JAX-RS** - Java API for RESTful Web Services
- **JAXB** - Java Architecture for XML Binding
- **JSON** - JavaScript Object Notation
- **JSP** - JavaServer Pages
- **MIME** - Multipurpose Internet Media Extensions
- **MVC** - Model View Controller
- **OBR** - OSGi Bundle Repository
- **OSGi** - Open Services Gateway initiative
- **POJO** - Plain Old Java Object
- **POX** - Plain Old XML
- **RAML** - RESTful API Modeling Language
- **ReliSA** - Reliable Software Architectures
- **REST** - REpresentational State Transfer
- **ROA** - Resource Oriented Architecture

- **RSDL** - RESTful Service Description Language
- **SOA** - servisně orientovaná architektura
- **SOAP** - Simple Object Access Protocol
- **URI** - Unified Resource Identifier
- **URL** - Unified Resource Locator
- **W3C** - World Wide Web Consorcium
- **WADL** - Web Application Description Language
- **WAR** - Web Application Resource nebo Web application ARchive
- **WSDL** - Web Services Description Language
- **XML** - eXtensible Markup Language
- **YAML** - YAML Ain't Markup Language

Seznam tabulek

A Ukázka konfiguračního souboru

Zde je ukázka části konfiguračního souboru pro tvorbu API pomocí frameworků implementujících JAX-RS.

```
framework: jaxrs

resource_annotations:
  - javax/ws/rs/Path

endpoint_annotations:
  - javax/ws/rs/GET
  - javax/ws/rs/POST
  - javax/ws/rs/PUT
  - javax/ws/rs/DELETE
  - javax/ws/rs/HEAD
  - javax/ws/rs/OPTIONS
  - javax/ws/rs/PATCH
  - javax/ws/rs/Path

url:
  - annotation: javax/ws/rs/Path
    processing_way: from_value
    valueKeys:
      - value

produces:
  - annotation: javax/ws/rs/Produces
    processing_way: from_value
    valueKeys:
      - value

consumes:
  -  annotation: javax/ws/rs/Consumes
     processing_way: from_value
     valueKeys:
```

```
       - value

default_http_methods: []

http_method:
  -   annotation: javax/ws/rs/GET
      processing_way: from_name
      result: GET
  -   annotation: javax/ws/rs/POST
      processing_way: from_name
      result: POST
  -   annotation: javax/ws/rs/PUT
      processing_way: from_name
      result: PUT
          ...
parameter_requirement:
  annotationProcessors:
    -   annotation: javax/validation/constraints/NotNull
        processing_way: from_name
        result: true
  parametersDefault: false
  fieldsDefault: false

default_parameter_value:
   -   annotation: javax/ws/rs/DefaultValue
       processing_way: from_value
       valueKeys:
         -  value

default_object_mime: application/xml
default_primitive_mime: text/plain

fieldParamAnnotations: true
fieldParamSetterRequired: false

exception_handler:
  annotations:
    - javax/ws/rs/ext/Provider
  interfaces:
    - javax/ws/rs/ext/ExceptionMapper
  method: toResponse
```