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

Diplomová práce

Rekonstrukce API volaných webových služeb

Plzeň 2021 Bc. Tomáš Ballák

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Tomáš BALLÁK**
Osobní číslo:	**A19N0023P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Rekonstrukce API volaných webových služeb**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s typy webových služeb a způsoby implementace jejich klientské a servisní části. Vyhledejte a analyzujte nejpoužívanější technologie v jazyce Java pro vytváření klientů webových služeb.
2. Prostudujte metody a nástroje pro analýzu již existujících implementací softwarových modulů. Seznamte se s účelem úložiště CRCE a jím používanou reprezentací rozhraní softwarových komponent a služeb.
3. Navrhněte algoritmus pro získání informací o volaném („required“) rozhraní webových služeb pro nejběžnější případy užití technologií vybraných na základě bodu 1.
4. Implementujte rozšíření CRCE pro získání vhodné formy reprezentace rozhraní těchto vyžadovaných webových služeb a pro její uložení ve strojově čitelné podobě.
5. Ověřte funkčnost navrženého řešení vč. použití reálných implementací webových služeb jako testovacích dat.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Doc. Ing. Přemysl Brada, MSc., Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **11. září 2020**
Termín odevzdání diplomové práce: **20. května 2021**

L.S.

Doc. Dr. Ing. Vlasta Radová
děkanka

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 24. září 2020

Prohlášení

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

V Plzni dne 20. května 2021

Bc. Tomáš Ballák

Abstract

This master's thesis is focused on the reconstruction of the API the outgoing REST web services implemented in the Java programming language. The analytical part analyzes the most used web services and also compares possible approaches to code analysis together with available libraries. The experimental part deals with the design and implementation of an algorithm for the reconstruction of web service calls, which is also the goal of the thesis. The algorithm is implemented in the form of a module in the existing repository CRCE (Component Repository supporting Compatibility Evaluation) developed at the DCSE UWB. Thanks to the proposed algorithm, the implemented module can process the attached archives with already translated source codes and reconstruct information about the called web services from them. The obtained information is transformed into the required format and then stored in the CRCE repository.

Abstrakt

Diplomová práce se zaměřuje na rekonstrukci API volaných webových služeb typu REST implementovaných v programovacím jazyce Java. V teoretické části jsou analyzovány nejpoužívanější webové služby a zároveň jsou porovnány možné přístupy k analýze kódu společně s dostupnými knihovnami. Praktická část se zabývá návrhem a implementací algoritmu pro rekonstrukci volání webových služeb, který je zároveň cílem diplomové práce. Algoritmus je implementován formou modulu do existujícího úložiště CRCE (Component Repository supporting Compatibility Evaluation) vyvíjeného na KIV ZČU. Vytvořený modul dokáže díky navrhnutému algoritmu zpracovat přiložené archivy s již přeloženými zdrojovými kódy a rekonstruovat z nich informace o volaných webových službách. Získané informace jsou transformovány do požadovaného formátu a následně uloženy do CRCE repozitáře.

Poděkování

Rád bych poděkoval vedoucímu práce *doc. Ing. Přemysl Brada, MSc., Ph.D.* za vedení diplomové práce, užitečné rady a celkovou vstřícnost.

Obsah

1 Úvod **9**

2 Webové služby **11**
2.1 Typy webových služeb . . . 11
2.1.1 SOAP . . . 11
2.1.2 REST . . . 13
2.1.3 GraphQL . . . 20
2.1.4 Srovnání popularity . . . 23
2.2 REST frameworky . . . 24
2.2.1 Java . . . 24
2.2.2 Python . . . 27
2.2.3 Node.js . . . 28
2.2.4 Shrnutí . . . 29

3 CRCE **31**
3.1 Vize . . . 31
3.1.1 Repositáře pro SW komponenty . . . 31
3.2 Architektura . . . 33
3.3 Životní cyklus . . . 34
3.4 Struktura metadat . . . 35
3.5 Využití OSGi frameworku . . . 39
3.5.1 Apache Felix . . . 40
3.6 CRCE Indexery . . . 40
3.6.1 crce-metadata-indexer . . . 41
3.6.2 crce-metadata-osgi-bundle . . . 41
3.6.3 crce-webservices-indexer . . . 41
3.6.4 crce-restimpl-indexer . . . 41

4 Přístupy pro analýzu kódu **42**
4.1 Java . . . 42
4.1.1 Kompilace . . . 42
4.1.2 Java Virtual Machine a načtení třídy . . . 43
4.1.3 Just-In-Time kompilace . . . 44
4.1.4 Byte-kód . . . 44
4.2 Dekompilace . . . 53
4.3 Zpracování byte-kódu . . . 55

4.3.1 ASM 55
4.3.2 BCEL API 57
4.3.3 CRCE indexer webových služeb typu REST 58
4.3.4 Shrnutí 59

5 Návrh algoritmu pro analýzu API volaných WS 60

5.1 Analýza byte-kódu 60
5.1.1 Instrukce pro konstanty 60
5.1.2 Instrukce pro proměnné 61
5.1.3 Instrukce pro pole 62
5.1.4 Instrukce pro atributy třídy 63
5.1.5 Instrukce pro metody 64
5.1.6 Anonymní třídy a generické typy 65
5.2 Analýza tříd a rozhraní frameworků určených pro klienty WS 66
5.2.1 Spring 67
5.2.2 JAX-RS client API 74
5.2.3 Shrnutí 77
5.3 Algoritmus 78
5.3.1 Konfigurace 78
5.3.2 Zpracování byte-kódu 78
5.3.3 Interpretace 79
5.3.4 Rekonstrukce API volaných WS 80

6 Implementace analyzátoru 83

6.1 Struktura programu 83
6.1.1 ClassModel 84
6.1.2 Config 85
6.1.3 ClassProcessor 85
6.1.4 BasicProcessor 85
6.1.5 EndpointProcessor 85
6.1.6 Tools 85
6.1.7 Bean a Dependency Procesor 85
6.2 Konfigurační soubor 86
6.2.1 RequestParameters 86
6.2.2 ArgDefinitions 86
6.2.3 WsClientData 87
6.2.4 WsClient 88
6.3 Integrace 88

7 Ověření funkčnosti **90**
7.1 Testování funkcionality 90
7.1.1 Vlastní implementace 90
7.1.2 Testování na reálných implementacích 93
7.2 Shrnutí 95

8 Závěr **96**

A Literatura **98**

B Uživatelský manuál **103**
B.1 Příkazová řádka 103
B.2 Webové rozhraní 103
B.3 REST API 104

C Seznam obrázků **105**

D Seznam ukázek kódu **106**

E Seznam tabulek **108**

F Seznam zkratek **110**

G Struktura odevzdávaného archivu **112**

1 Úvod

Webová služba (WS) je software, který je navržen pro síťovou komunikaci mezi různými zařízeními. Ta probíhá typicky prostřednictvím aplikačního protokolu HTTP [25, 26]. Komplexní systémy, které tyto služby využívají, bývají často vzájemně provázány, ať už za účely agregace nebo decentralizace [52]. Pro udržení celkové stability systému je nutná kontrola kompatibility služeb, a to i napříč jejich verzemi.

Významný vliv na myšlenku zajištění kompatibility nebo funkčnosti programu má nepochybně proces Continuous integration (CI). Proces si klade za cíl zajistit, aby při každé změně kódu bylo možné rozpoznat narušení logiky programu [47], což by mohlo vést k neočekávanému chování. Součástí CI procesu je mimo jiné i automatizované sestavení zdrojového kódu. Nástroj pro kontrolu kompatibility se tak může stát jeho důležitou součástí. Zejména u WS typu REST je tento proces velmi důležitý. Právě pro ně byl v rámci projektu Component Repository supporting Compatibility Evaluation (CRCE) vytvořen indexer, který dokáže získat informace o poskytovaných REST službách [44], tedy o *serverové části* dané aplikace. K porovnání kompatibility je zapotřebí zpracovat data, jak ze strany serverové části, tak i ze strany klienta.

Cílem této práce je vytvořit nástroj, který je schopen získat informace o *volaných* REST službách v rámci *klientské* části. Informace, které tento nástroj může poskytnout, lze využít nejen pro kontrolu kompatibility mezi *klientem* a *serverem*, ale i k porovnání jejich verzí.

Kapitola *Webové služby* se zabývá jednotlivými typy webových služeb a frameworky, které se využívají pro jejich implementaci. Jsou zde porovnány jednotlivé typy jako je *SOAP*, *REST* a *GraphQL*. Porovnání se zaměřuje na jejich vlastnosti a jejich popularitu v roce *2021* (kdy je práce vytvořena). Další částí je představení dostupných frameworků pro různé programovací jazyky jako například *Java*, *Python*, ale i *JavaScript* (zaměřeno na Node.js). Po představení je vyhodnoceno jejich zastoupení na trhu spolu s nastíněním jejich klíčových vlastností.

V kapitole *CRCE* (Component Repository supporting Compatibility Evaluation) se nachází rozbor úložiště CRCE. Jeho původní myšlenkou je kontrola kompatibility mezi softwarovými komponentami. Kombinací nástroje pro *Rekonstrukci API volaných WS* a výše zmíněného indexeru se možnosti CRCE rozšíří o funkcionalitu kontroly kompatibility v rámci webových služeb. Jelikož je nástroj pro rekonstrukci API WS integrován do výše zmíně-

ného úložiště, stává se tak jeho nedílnou součástí. Tato kapitola se zároveň zaměřuje na celkovou vizi úložiště CRCE a na oblasti, kde by šlo tento software využít. Další částí je samotná architektura a použité technologie spolu s popisy modulů, které poskytují hlavní funkcionality.

Další kapitola *Přístupy pro analýzu kódu* je zaměřena na porovnání jednotlivých knihoven a přístupů, které jsou pro účely analýzy kódu v rámci této práce nejvhodnější. Při výběru jsou zhodnoceny nejen schopnosti získání informací příslušnými nástroji, ale i jejich případná výkonnost. Zároveň je zde v krátkosti představen výše zmíněný indexer pro WS v rámci serverové části aplikace.

Kapitola *Návrh algoritmu pro analýzu API volaných WS* se zabývá možnostmi řešení hlavní problematiky této práce, a to *Rekonstrukce API volaných WS*. Je zde vysvětlen konečný přístup, který bude implementován. Algoritmus je popsán ve formě diagramu spolu s popisem jeho nejdůležitějších částí.

Implementační část je řešena v kapitole *Implementace analyzátoru*. Zde je zastoupena architektura, popis jednotlivých částí a implementace algoritmu.

Závěrečná část práce se zabývá tím, jak se postupovalo v případě testování funkcionalit, robustnosti a do jaké míry byly splněny deklarované cíle.

2 Webové služby

Podle World Wide Web konsorcia (W3C) jsou webové služby definovány jako množina standardů [25], pomocí kterých lze uskutečnit komunikaci přes *HTTP*, za využití *XML* nebo *JSON*. Mezi jejich hlavní myšlenku patří komunikace mezi *poskytovateli* a *konzumenty* služeb. V této kapitole jsou popsány jejich výhody spolu s využitelností v praxi. Na služby typu REST se bude upínat celá tato práce. V rámci této kapitoly jsou vypsány a porovnány dostupné REST frameworky, které se v posledních letech stávají velmi populární.

2.1 Typy webových služeb

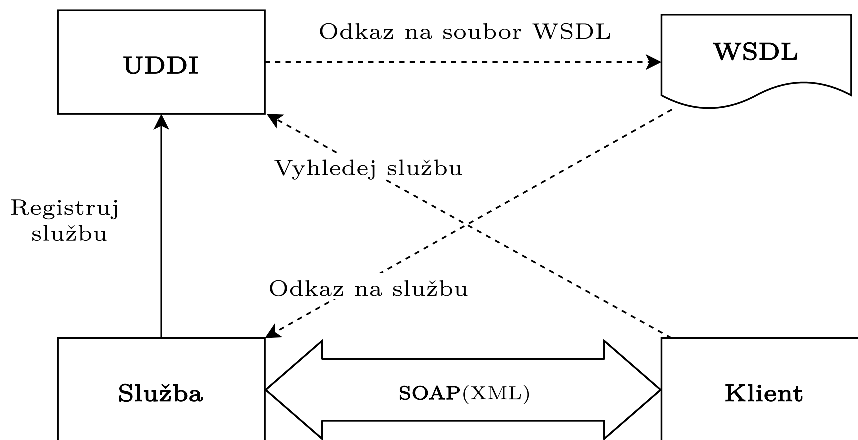
Webové služby mohou být rozděleny podle zvolených protokolů, architektur apod. Jako první příklad lze uvést využití protokolu *SOAP*, dále implementace založena na architektuře *REST* a v neposlední řádě použití dotazovacího jazyka *GraphQL*.

2.1.1 SOAP

SOAP je zkratka pro Simple Object Access Protocol. K přenosu zpráv využívá aplikační protokol *HTTP*. SOAP ve verzi 1.2 používá pro komunikaci Požadavek-Odpověď metodu *POST*. Další možností je Požadavek(bez těla zprávy)-Odpověď, kde může být požadavek tvořen pomocí metody *GET* (nemusí se chránit obsah zprávy). Uvažuje se i o možném použití dalších metod jako je *PUT*, *DELETE* atd. [50].

Mezi jeho důležité funkcionality lze zařadit poskytování popisu služeb pro klientský uzel. K němu se využívá *Web Services Description Language* (WSDL) [50].

Celý proces začíná tím, že se webová služba registruje do *UDDI* - správce informací o WS. Ten si drží základní údaje o poskytovateli a odkaz na WSDL soubor popisující danou službu. Po získání tohoto souboru může *klient* zahájit komunikaci prostřednictvím SOAP protokolu s poskytovatelem služby. Názorná ukázka těchto kroků zobrazuje obrázek 2.1.



Obrázek 2.1: SOAP a WSDL (zdroj: vlastní tvorba).

Pro komunikaci se používají speciální SOAP zprávy, které jsou předávány ve formě XML dokumentů. Struktura těchto zpráv je následující [11, 57].

- Obálka
  - `<soap:Envelope>`
  - kořen celé zprávy
- Hlavička
  - `<soap:Header>`
  - nepovinná část
  - aplikační informace spojené se zprávou
    - autentizace
    - platba
- Tělo
  - `<soap:Body>`
  - obsahuje samotnou zprávu
- Informace o chybě
  - `<soap:Fault>`
  - obsahuje informaci o chybové zprávě, její kód, kdo ji způsobil, informaci spojenou s tělem zprávy

WSDL

Aby byla zajištěna kompatibilita, se pro dvojici *služba-klient* v rámci protokolu SOAP používá Web Services Description Language (WSDL)[54]. Je to popisný jazyk využívající formát *XML*, který zajišťuje definici jednotlivých endpointů dané služby. Používá se k popisu jak Remote Procedure Call (RPC) služeb, tak ale i k popisu služeb *Dokumentových*.

2.1.2 REST

REST je architektonický styl pro distribuované hypermediální systémy (zvuk, video, text, propojené prostřednictvím odkazů) [39].

Definice 1 *Architektura softwaru je množina struktur potřebných pro popis systému včetně jeho jednotlivých prvků a jejich vzájemných vztahů spolu s jejich vlastnostmi [23] .*

Myšlenka REST je založena na následujících vlastnostech: separation of concerns (SOC); beze-stavovost; kešovatelnost (cache); uniformní rozhraní; systém rozdělen do vrstev; code-on-demand [39].

Vlastnosti architektury REST

V případě webových služeb je SOC myšleno jako oddělení uživatelského a datového rozhraní. To znamená převedení co největšího množství logiky na stranu klienta. Hlavní výhodou je tak možnost vyvíjet obě strany nezávisle na sobě. Zároveň musí být zachováno rozhraní, přes které tyto dvě strany komunikují.

REST komunikace je *beze-stavová*. Znamená to, že uchovávání stavu závisí pouze na klientovi. Každý požadavek tak musí nést potřebné informace, které dotvářejí chybějící kontext. Beze-stavovost sebou přináší řadu výhod jako je například *větší spolehlivost*. Komunikace není závislá na předešlých událostech a je kompletně nezávislá.

Pro urychlení komunikace se REST opírá o myšlenku mezi-paměti (*cache*). Odpovědi serveru, které jsou označeny jako kešovatelné, uživatel uchovává pro opakovatelná volání. Tento přístup částečně odbourává počet dotazů, které by musely být jinak provedeny. Zároveň to může způsobit problémy, kdy data uložená do mezi-paměti už nadále nejsou aktuální.

Komponenty mezi sebou využívají uniformní rozhraní. To se skládá ze tří základních prvků: *identifikace zdrojů*, *manipulace se zdroji* a *popisné zprávy* [39]. K identifikaci zdrojů se používá *Uniform Resource Locator* (URL) nebo *Uniform Resource Name* (URN) viz tabulka 2.1.

Datový část	Hodnoty
zdroj	cílový soubor označený identifikátorem
identifikace zdroje	URL, URN
popis zdroje	HTML dokument, obrázek (JPEG image) apod.
metadata popisu	media type, last-modified time
manipulace se zdroji	if-modified-since, cache-control

Tabulka 2.1: Omezení rozhraní pro REST (zdroj: [39])

Oddělení systému do vrstev je jedna z klíčových myšlenek architektury REST. Při vývoji komplexních systémů, které mohou fungovat několik jednotek až desítek let, je potřeba přemýšlet nad zpětnou kompatibilitou. Pro tyto účely mohou existovat dva druhy vrstev: *legacy vrstva* - starší funkcionalita, která musí být zachována a vrstva určená pro nové funkcionality. Další možností jak využít vrstvy je přesunutí často používaných služeb do nové společné vrstvy [39].

Code-on-demand řeší rozšíření funkcionality klienta za pomocí stažení kódu rozšiřující funkcionalitu (dříve Java applet). Dnes se pracuje především s JavaScriptovými aplikacemi, ať už v podobě komplexnějších (ReactJs) či jednodušších (JQuery) knihoven. Architektura řeší poměrně široce problematiku spojení klient-server. Udává, jakým směrem by se měla ubírat nejen serverová část, ale navrhuje i možné přístupy klienta. Tato řešení tak vedou ke stabilnějšímu a robustnějšímu systému, který si zároveň uchovává prvek agility.

HTTP

Hypertext Transfer Protocol (HTTP) je bezestavový aplikační protokol pro distribuované systémy, spolupracující a hypermediální informační systémy [40].

REST je často zmiňován pravě spolu s protokolem *HTTP*. Ten se dá z části označit jako RESTový [39]. V rámci webových služeb hrají oba dva důležitou roli. HTTP disponuje devíti metodami *GET*, *HEAD*, *POST*, *PUT*, *DELETE*, *CONNECT*, *OPTIONS*, *TRACE*, *PATCH* [40]. Každá z nich má sémantický význam odkazující na tzv. *Create, Read, Update and Delete* (CRUD) operace.

Pro výčet následujících HTTP metod je potřeba nejprve definovat následující pojmy *bezpečnost*, *idempotentnost*, *kešovatelnost*.

Definice 2 *HTTP metodu lze považovat za bezpečnou, pokud neovlivňuje celkový stav serveru, to znamená, že provádí pouze čtení [20].*

Definice 3 *HTTP metodu lze považovat za idempotentní, pokud si udržuje stejnou funkcionalitu i po několikátém volání v řadě. [16].*

Definice 4 *HTTP odpověď ze serveru lze považovat za kešovatelnou (cache), pokud jí lze uchovat pro pozdější využití u nového požadavku [14].*

Na následujících několika stranách jsou popsány jednotlivé HTTP metody spolu s jejich vlastnostmi.

- *GET*
  - jen pro získávání dat (*READ*)
  - příklad požadavku:

```
GET /page/image.png
```

    * bez parametrů

```
GET /page?num=1&string=ZCU
```

    * s parametry *num* a *string*

Vlastnost	**[ANO/NE]**
Požadavek má tělo	NE
Úspěšná odpověď má tělo	ANO
Bezpečné	ANO
Idempotentní	ANO
Kešovatelné (Cache)	ANO
Povolené v HTML formulářích	ANO

Tabulka 2.2: HTTP metoda GET (zdroj: [15])

- *HEAD*
  - požaduje odpověď identickou u požadovku *GET*, ale bez těla v odpovědi (*READ*)

```
GET /page?num=1&string=ZCU
GET /page/image.png
```

Vlastnost	**[ANO/NE]**
Požadavek má tělo	NE
Úspěšná odpověď má tělo	NE
Bezpečné	ANO
Idempotentní	ANO
Kešovatelné (Cache)	ANO
Povolené v HTML formulářích	ANO

Tabulka 2.3: HTTP metoda HEAD (zdroj: [15])

- *POST*
  - používá se pro posílání entit specifické službě, může ovlivnit stav serveru (*Create*, *Update*, *Delete*)

```
POST /page
Host: host.cz
Content-Type: application/x-www-form-
   urlencoded
Content-Length: 16

num=1&string=ZCU
```

    * parametry *num* a *string* předané v těle dotazu
    * *Content-Type* definuje typ předávané zprávy, v tomto případě data pro formulář (podobně jako u *GET*)
    * *Content-Length* popisuje délku poslané zprávy

Vlastnost	**[ANO/NE]**
Požadavek má tělo	ANO
Úspěšná odpověď má tělo	ANO
Bezpečné	NE
Idempotentní	NE
Kešovatelné (Cache)	Jen pokud je nastavený parametr
Povolené v HTML formulářích	ANO

Tabulka 2.4: HTTP metoda POST (zdroj: [15])

- *PUT*
  - požadavek kompletně *vymění* cílový zdroj vlastními daty, které jsou dodány v těle požadavku (*Delete*, *Create*)

```
PUT /school
Host: host.cz
Content-Type: application/json
Content-Length: 37

{name: ZCU, faculty: [{ name: FAV }]}

Odpověď:
201 Created - pokud záznam neexistoval
200 nebo 204 - pokud záznam byl aktualizován
```

* pokud neexistuje záznam o ZCU, tak vytvoří nový
* pokud záznam existuje, tak ho přepíše

Vlastnost	**[ANO/NE]**
Požadavek má tělo	ANO
Úspěšná odpověď má tělo	NE
Bezpečné	NE
Idempotentní	ANO
Kešovatelné (Cache)	NE
Povolené v HTML formulářích	NE

Tabulka 2.5: HTTP metoda PUT (zdroj: [15])

- *DELETE*
  - odstranění zdroje podle identifikátoru (*Delete*)

```
DELETE /school/1
Host: host.cz
```

Vlastnost	**[ANO/NE]**
Požadavek má tělo	MŮŽE
Úspěšná odpověď má tělo	MŮŽE
Bezpečné	NE
Idempotentní	ANO
Kešovatelné (Cache)	NE
Povolené v HTML formulářích	NE

Tabulka 2.6: HTTP metoda DELETE (zdroj: [15])

- *CONNECT*
  - vytvoří tunel mezi klientem a serverem identifikovaném v požadavku

```
CONNECT www.zcu.cz:443
Host: www.zcu.cz:443
Proxy-Authorization: basic abc123=
```

    * vytvoření tunelu mezi www.zcu.cz:443 s využitím přihlašovacích údajů, které se nachází v *proxy-authorization*.

Vlastnost	**[ANO/NE]**
Požadavek má tělo	NE
Úspěšná odpověď má tělo	ANO
Bezpečné	NE
Idempotentní	NE
Kešovatelné (Cache)	NE
Povolené v HTML formulářích	NE

Tabulka 2.7: HTTP metoda CONNECT (zdroj: [15])

- *OPTIONS*
  - používá se k popisu komunikace cílené na určitý zdroj

```
OPTIONS /school
Odpověď:
Allow: OPTIONS, GET, POST, HEAD
```

    * odpověď vrací možné HTTP metody, které lze na tento endpoint odeslat

Vlastnost	**[ANO/NE]**
Požadavek má tělo	NE
Úspěšná odpověď má tělo	ANO
Bezpečné	ANO
Idempotentní	ANO
Kešovatelné (Cache)	NE
Povolené v HTML formulářích	NE

Tabulka 2.8: HTTP metoda OPTIONS (zdroj: [15])

- *TRACE*
  - message loop-back test

```
TRACE /school
```

Vlastnost	[ANO/NE]
Požadavek má tělo	NE
Úspěšná odpověď má tělo	NE
Bezpečné	ANO
Idempotentní	ANO
Kešovatelné (Cache)	NE
Povolené v HTML formulářích	NE

Tabulka 2.9: HTTP metoda TRACE (zdroj: [15])

- *PATCH*
  - aplikuje jen určité změny v cílovém zdroji (*Update*)

```
PATCH /school/1
Host: host.cz
Content-Type: application/json
Content-Length: 41

{faculty: [{ name: FAV }, { name: FEL }]}

Odpověď:
201 nebo 204
```

  * aktualizace záznamů fakult školy pod ID 1

Vlastnost	[ANO/NE]
Požadavek má tělo	ANO
Úspěšná odpověď má tělo	ANO
Bezpečné	NE
Idempotentní	NE
Kešovatelné (Cache)	NE
Povolené v HTML formulářích	NE

Tabulka 2.10: HTTP metoda PATCH (zdroj: [15])

Komunikace

Komunikace klient-služba probíhá tak, že klient vyšle požadavek s nastavenými parametry a server odpoví zprávou ve formě JSONu nebo XML. Oproti komunikaci pomocí SOAP je tento způsob výrazně jednodušší a lze vidět, že klient je opravdu oddělen od dané služby.

Obrázek 2.2: REST (zdroj: vlastní tvorba)

WADL

WADL je XML popis služeb, které jsou postaveny na architektuře *REST* [21]. Pomocí tohoto popisu se klient připojí na požadovanou službu a zároveň získá informace o možnosti její využití. Oproti klasickým (dokumentačním) popisům je WADL zaměřen primárně na strojový popis a z toho pramení řada výhod [41].

V rámci něho se dají získat informace o dostupných zdrojích nebo funkcionalitách, které daná služba nabízí. Tato informace může být například využita k vytvoření vizuálního modelu. Ten by mohl zobrazovat vztahy mezi jednotlivými zdroji a jak je možné k nim přistupovat.

Hlavní výhodou je také možnost automaticky vygenerovat klienty, kteří konzumují obsah dané služby. To zajišťuje maximální kompatibilitu a díky tomu se mohou omezit případné chyby a urychlit celkový vývoj klientské části.

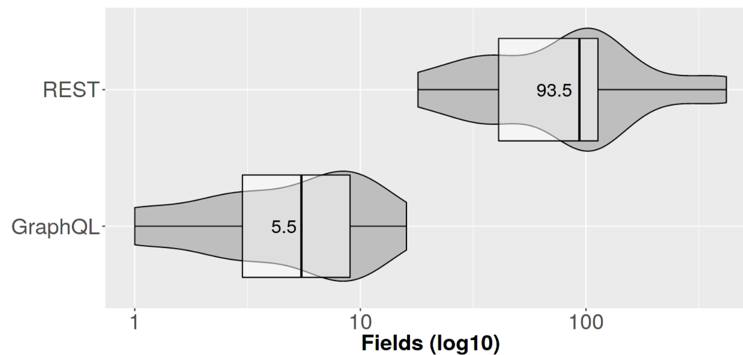
V neposlední řadě *WADL* umožňuje definovat jednotný a přenositelný způsob komunikace mezi serverem a klientem.

2.1.3 GraphQL

GraphQL je dotazovací jazyk pro Application Programming Interface (API), který byl v roce 2015 vytvořen společností Facebook [38, 45]. Není vázán na specifický typ databáze nebo engine úložiště.

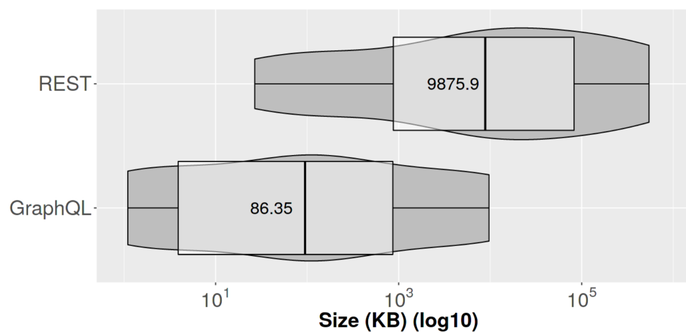
GraphQL razí výrazně odlišný přístup ve srovnání s REST nebo SOAP. Jednotlivé endpointy, přes které se získávají data, jsou tvořeny na straně klienta. To výrazně urychluje celkový vývoj nehledě na zkušenosti programátora [29]. Kromě urychlení vývoje, je i výrazně zlepšena efektivita v oblasti přenosu dat. Ukázalo se, že případný přepis klasické REST architektury

na GraphQL může výrazně zrychlit celkový přenos dat. To lze určit na základě mediánových hodnoty, kde velikost výsledných JSON objektů (počet záznamů v JSON objektu) byla snížena o *94%* (93.5 → 5.5) [30], jak je vidět na obrázku 2.3.



Obrázek 2.3: Porovnání vel. objektů uložených v JSON objektu (zdroj: [30]).

U přenosu JSON dokumentů bylo zjištěno celkové snížení velikosti o *99%* (bytů) [30]. To je možné vidět na obrázku 2.4.



Obrázek 2.4: Porovnání celkové velikosti JSON dokumentů (zdroj: [30])

GraphQL používá pro komunikaci *Queries* a *Mutations*. Queries se používají pro získávání dat. Klient si přímo nadefinuje požadovaný objekt a k němu další informace, které mají na tento objekt vazbu. Příklad takového požadavku lze vidět na ukázce kódu 2.1.

```
1 query School(){
2     school {
3         name
4         subjects {
```

```
5               name
6           }
7       }
8   }
```

Kód 2.1: Query s připojenými subjects (zdroj: vlastní tvorba)

Kromě těchto dotazů lze získat data podle nadefinovaných argumentů, jako je např. ID, jméno apod. viz ukázka kódu 2.2.

```
1 query School(){
2     school (name: "ZČU") {
3         name
4         subjects {
5             name
6         }
7     }
8 }
```

Kód 2.2: Argumenty dotazu (zdroj: vlastní tvorba)

Mutations (Mutace) se využívají pro vytváření, úpravu nebo odstranění záznamů. Je to způsob jak zavést konvence, aby byla jasně oddělena manipulace a získávání (čtení) dat. Vytvoření nového záznamu lze vidět na ukázce kódu 2.3. Při vkládání se dá určit jaká část argumentu ($school) bude použita, v tomto případě pouze *name*. $School je zde v roli proměnné, která si drží objekt typu *School*. Znamená to, že podle potřeby lze z tohoto typu vybrat pouze požadované části.

```
1 mutation CreateSchool($school: School!){
2     school (name: $school)") {
3         name
4     }
5 }
```

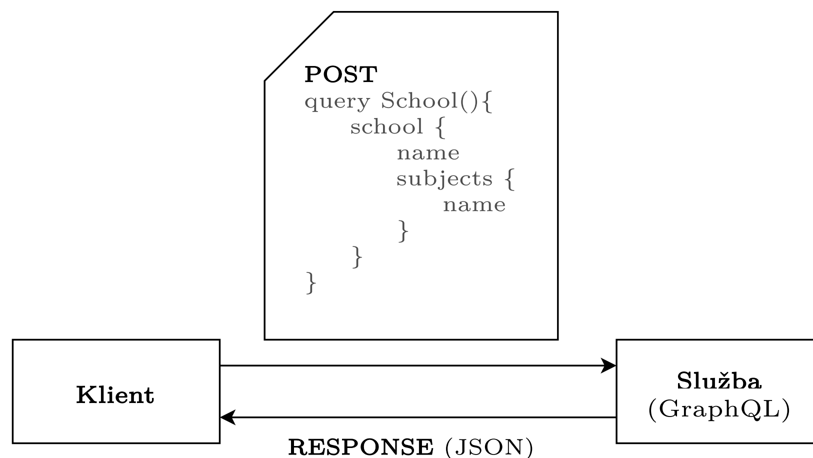
Kód 2.3: Mutation (zdroj: vlastní tvorba)

Pro tento způsob přístupu k datům je potřeba mít jejich popis (*schema*). GraphQL používá pro definici schémat jednotlivých datových typů *schema definition language* (SDL), to lze vidět na ukázce kódu 2.4.

```
type School {
    name: String
    subjects [Subject]
}
```

Kód 2.4: Schéma pro typ School (zdroj: vlastní tvorba)

Podobně jako u REST nebo SOAP, tak i GraphQL je nezávislé na výběru programovacího jazyka, avšak používá se především v rámci Node.js aplikací. Kromě JavaScriptu existují implementace např. v jazyce Java nebo Python. Interakce mezi klientskou částí a GraphQL službou znázorňuje obrázek 2.5.



Obrázek 2.5: GraphQL (zdroj: vlastní tvorba)

2.1.4 Srovnání popularity

Na základě srovnání popularity GraphQL, REST a SOAP viz tabulka 2.11 lze říci, že z porovnávané trojice je REST v GitHub repositářích nejvíce zastoupený. Při srovnání období *2004-2021* a *2015-2021* vynikne především výrazný nárůst v počtu vyhledávání slova "REST API"(architektury REST), a to prostřednictvím nejpoužívanějšího vyhledávače (Google).

Název	**GitHub**		**Google Trends**[1]	
	Search results	Topics	2004-2021[2]	2015-2021[3]
REST	497 740	7 708	33	74
GraphQL	105 419	16 177	8	24
SOAP	20 501	762	4	5

Tabulka 2.11: SOAP vs REST vs GraphQL (zdroj: vlastní tvorba)

GraphQL v posledních letech zažívá velký nárůst popularity viz tabulka 2.11. Jelikož je to poměrně nová technologie (vznik roku 2015), nelze s jistotou tvrdit, že si tento trend i do budoucna udrží.

Definice 5 *Software, jehož zdrojové kódy jsou neveřejné lze považovat jako uzavřeny (closed source).*

Definice 6 *Software, jehož zdrojové kódy jsou veřejné a volně distribuované lze považovat jako otevřený (open source).*

Není zde pokryta množina tzv. *closed source* projektů, pro které by byl vhodnější jiný druh analýzy. Nicméně můžeme konstatovat, že popularita u projektů typu *open source* bude do jisté míry odrážena i do uzavřených projektů.

2.2 REST frameworky

Definice 7 *Framework definuje to, jakým způsobem se musí členit struktura programu a zároveň určuje způsob interakce mezi jednotlivými komponentami. Snaží se vynutit styl komunikace, který je určen pro řešení domény daného problému [53].*

K vývoji WS se většinou využívá framework. Ten umožňuje lépe udržovat zdrojový kód a udržet si určený způsob komunikace napříč komponentami.

Tato práce je zaměřena primárně na RESTové služby. Předchozí analýza ukazuje, že ty jsou právě v dnešní době nejvíce využívané. A právě proto se tato práce ubírá stejným směrem jako zmiňovaný CRCE modul pro zpracování rozhraní WS (serverové části), tedy zpracováním webových služeb, které jsou implementovány pomocí REST frameworků.

Dále je potřeba získat informaci o tom, jaké konkrétní frameworky jsou nejpopulárnější. Tyto informace budou použity pro rozhodnutí o zaměření implementační části této práce.

2.2.1 Java

Tato sekce je zaměřena na frameworky určené pro jazyk Java.

[1] Google Trends: zobrazuje relativní (hodnocení od 0 do 100) trendy vyhledávání prostřednictvím vyhledávače od společnosti Google podle dodaných klíčových slov

[2] 2004-2021: průměrné hodnoty v rozmezí roku 2004 a 2021; rok 2004 je nejblíže roku, kdy vznikl SOAP a REST a kdy Google začal zaznamenávat data o vyhledávání

[3] 2015-2021: průměrné hodnoty v letech 2015 a 2021; v roce 2015 vzniklo GraphQL

Spring, Jersey, RESTeasy a Apache CXF jsou frameworky určené pro RESTovou komunikaci. Podle dostupných dat, které lze vidět v tabulce 2.12 se *Spring* řadí do řady nejpopulárnějších REST frameworků v rámci jazyka Java.

Název	**GitHub**		**Maven repository**
	Search results	Topics	Search results
Spring(boot)	253 898	27 959	10 381
Jersey	8 779	496	1 401
RESTeasy	1 134	95	969
Apache CXF	675	40	940

Tabulka 2.12: Java REST frameworky (zdroj: vlastní tvorba)

JAX-RS API a JAX-RS client API

JAX-RS je množina specifikací, rozhraní a anotací určené pro REST komunikaci [3, 22]. Mezi nejznámější implementace patří *Jersey*, *RESTeasy* a *Apache CXF* viz tabulka 2.12. Jejich popularita nedosahuje zdaleka takových hodnot jako má *framework Spring* viz tabulka 2.12, ale to nutně neznamená, že nemohou být více zastoupeny například v komerční sféře, která zde není zmapována. Pro jakoukoliv JAX-RS implementaci platí, že jsou navzájem zaměnitelné, všechny mají stejné *názvy* tříd a *balíčků* (package) viz ukázka kódu 2.5.

```
1    import javax.ws.rs.GET;
2    import javax.ws.rs.Path;
3    import javax.ws.rs.core.MediaType;
```

Kód 2.5: Importování JAX-RS implementací (zdroj: vlastní tvorba)

Do verze *JAX-RS 2.0* existovalo API pouze pro serverovou část. Spolu s novou verzí tak přišla podpora i pro *klienta* [34]. Znamená to, že případný analyzátor prostřednictvím *JAX-RS client API* pokryje všechny frameworky, které pro své klientské části kódu využívají právě toto API.

Klientské *JAX-RS API* tvoří následující rohraní `AsyncInvoker`, `Client`, `ClientRequestContext` , `ClientRequestFilter`, `ClientResponseContext`, `ClientResponseFilter`, `Invocation`, `Invocation.Builder`, `WebTarget` [18]. Instance tříd, které tyto rozhraní implementují, slouží pro komunikaci se službami viz ukázka kódu 2.6.

```
1    Client client = ClientBuilder.newClient();
```

```
2     Response res = client.target("http://example.org
        /hello").request("text/plain").get();
```

Kód 2.6: Použití třídy Client pro komunikaci (zdroj: [18])

Tyto třídy jsou tak hlavní kandidáti pro analýzu kódu v rámci detekce volaných WS.

V článku z *Journal of Telecommunication* [56] proběhlo porovnání JAX-RS implementací, kde pomocí relativně komplexní metriky byl vytvořen žebříček výše zmíněných frameworků. Pro lepší porozumění způsobu testování je nutné nejprve nadefinovat několik základních pojmů.

Definice 8 *Load test je takový test, který je zaměřen na chování služby pod definovaným počtem požadavků .*

Definice 9 *Stress test je test, který se zabývá chováním služby pod neustálým příchodem požadavků. Test je mimo jiné zaměřen na detekci chyb v paměti.*

Definice 10 *Stability testing je testování zaměřené na hledání množství požadavků, které naruší chod dané služby.*

Definice 11 *Peak test je test, který ukazuje chování systémů při velkých změnách v počtu příchozích požadavků* (500 $\rightarrow$ 12000) *v pěti minutových intervalech.*

Testování probíhalo pod určeným časovým intervalem, a tedy v případě *Load test* 10 minut, *Stability test* 10 minut a *Peak test* pouze 5 minut.

Název	**Load**	**Stability**	**Stress**	**Peaks**	**Hodnocení (0-10)**
Jersey	4	2	9	4	4.75
RESTeasy	8	9.7	3	5	4
ApacheCXF	10	10	4	10	8.5

Tabulka 2.13: Porovnání JAX-RS implementací (zdroj: [56])

Podle tabulky 2.13 se *Jersey* ukázal jako nejodolnější vůči neustále přicházejícím požadavkům. Celkově nejlepší téměř ve všech oblastech měření se ukázal framework *Apache CXF*. Oproti tomu *RESTeasy* má nejhorší celkové hodnocení. Získal ale dobré skóre v testu stability, kde předčil framework *Jersey*.

Spring

Spring se ukázal jako nejpoužívanější a pravděpodobně i nejznámější Java framework. Pro účely rekonstrukce volaných WS je tak velmi důležitý. Případné pokrytí frameworku Spring může mít dosah až na deseti-tisíce projektů, které ho využívají.

Oproti *Apache CXF, Jersey apod.*, Spring používá *vlastní klienty* pro komunikace s WS. Jako první můžeme zmínit dnes už zastaralý (někde stále používaný) `RestTemplate` viz ukázka kódu 2.7.

```
1 School expectedObject = restTemplate
2     .getForObject("http://localhost:8080/page/school
        /{id}", School.class,2)
```

Kód 2.7: Rest Template (zdroj: vlastní tvorba)

Další třídou určenou pro konzumování WS je `WebClient` viz ukázka kódu 2.8. Ten nahradil výše zmiňovaný `RestTemplate`, který je už považován jako *deprecated*.

```
1 School expectedObject = restTemplate
2     .getForObject("http://localhost:8080/page/school
        /{id}", School.class,2)
```

Kód 2.8: `WebClient` (zdroj: vlastní tvorba)

2.2.2 Python

Python stejně tak jako Java patří mezi významné programovací jazyky. I pro něj je vytvořeno mnoho frameworků určených pro RESTovou komunikaci. Ty nejznámější jsou zde vypsány spolu s porovnáním jejich popularity.

Podle dostupných dat viz tabulka 2.14 lze tvrdit, že *Django* patří mezi nejpopulárnější frameworky pro Python. Výrazně ale pokulhává ve výkonnosti (počet dotazů za vteřinu). *Falcon*, který ze zmíněných frameworků patří k těm rychlejším, zvládne oproti Djangu zpracovat až *osm krát více* požadavků [33]. Pro služby, kde je výkonnost klíčová, tak Django nebude nejlepší volba. Silné stránky má však v dostupných knihovnách a ve větší komunitě. To může vést k rychlejšímu a celkově levnějšímu vývoji.

Ze skupiny starších a známějších frameworků lze uvést například *Flask*. Ten je zhruba *čtyřikrát rychlejší* než výše zmiňované Django. Zároveň má poměrně solidní základnu v podobě dostupných knihoven viz tabulka 2.14. První release[4] měl již v roce *2010*. Jedenáct let je tak dostatečná doba pro

[4]Release - co je to release

vytvoření mnoha projektů a knihoven.

Fast API je oproti všem zmíněným výrazně rychlejší a nejmladší framework. *Dvanáctkrát rychleji* dokáže FAST API zpracovat příchozí požadavky oproti konkurenčnímu Djangu [33]. Podle oficiálních informací je tato rychlost dosažena hlavně tím, že využívá *Starlette* a *Pydantic*. Sarlette je nástupce *WSGI*, tedy standardu pro rozhraní mezi webovými servery a webovými aplikacemi napsanými v Pythonu [37]. Umožňuje oproti WSGI asynchronní operace a podporuje například výše zmíněné GraphQL [12]. Pydantic se zase stará o to, aby každý objekt dostal požadované argumenty ve správném datovém typu [9].

Falcon lze rozhodně zařadit do skupiny rychlejších frameworků. Nedá se ale říci, že za *osm let* (první release roku 2013) nabyl nějak výrazně na popularitě. Po srovnání s o šest let mladším FAST API, které je *jeden a půl krát rychlejší* [33] a o něco populárnější, lze konstatovat, že FAST API de-facto Falcon nahradil.

Název	**GitHub**		**PyPI**
	Search results	Topics	Search results
Django Rest framework	410 501	29 898	10 000+
Flask	218 217	23 317	5 357
FastAPI	6 574	1 259	455
Falcon	6 094	224	309

Tabulka 2.14: Python REST frameworky (zdroj: vlastní tvorba)

2.2.3 Node.js

Podle dostupných dat z *Google Trends (rok 2004-2021)* se Node.js v posledních pěti letech stává velmi populární. Node.js je *open-source* platforma založená na JavaScriptu. Ke svému běhu využívá *V8 JavaScript engine* (napsaný v C++), to je také jádrem velmi známého webového prohlížeče Google Chrome [5]. V porovnání s *Apache* (PHP-FPM) a *Nginx* (PHP-FPM) si Node.js vede velmi dobře. Všechny zmíněné serverové technologie jsou výrazně pomalejší v dodávání dynamického obsahu. Apache zvládá zpracovat až *tři a půl krát* méně požadavků za vteřinu oproti Node.js. Nginx je na druhou stranu vhodnější jako *static file server* [46].

Mezi nejpoužívanější frameworky v rámci Node.js lze zařadit *Express.js*. Má výrazně větší komunitu v porovnání s *Restify.js*, *Hapi.js* a *LoopBack*. Díky výrazně větší komunitě lze podstatně rychleji nalézt potřebné informace k řešení problémů spojené s tímto frameworkem. Jeho používání je

velmi jednoduché a jednoduše se konfiguruje, proto se může hodit i pro některé menší projekty. Dovoluje i volnější přístup pro udržování kódu, to ale nemusí být vhodné pro větší projekty složené z více programátorských týmů.

Dalším z frameworků určených pro *RESTové* služby je *Restify.js*. Ten využívají známé společnosti, jako je například Netflix nebo Pinterest. Podle syntetických benchmarků provedených v rámci porovnání frameworků se ukazuje, že Restify dokáže o něco rychleji zpracovávat příchozí požadavky [2].

Hapi.js je framework, který je udržován v rámci společnosti *Brave Software Inc.* Původně byl vyvíjen primárně pro účely Black Friday v rámci společnosti *Walmart*. Hapi využívá pluginy pro případné upravení chování a rozšíření celkové logiky. Ty mají po registraci jasné místo v životním cyklu celé aplikace [35]. Další výhodou je to, že dodává ve svém jádře funkcionality, které v případě využití Express.js musí být doinstalovány, např.: cookie-parser, body-parser apod. Jelikož má v sobě spoustu balíků zakomponovaných v jádře, a také nedovoluje tak velkou volnost, bude vhodný zejména pro větší projekty [49].

Jako poslední ve vytvořeném žebříčku se nachází *LoopBack* viz tabulka 2.15. Využívají ho známé firmy jako například *IBM* a *godaddy.com*. LoopBack je založený na výše zmíněném frameworku Express.js. Podle oficiální dokumentace se dá dobře využít pro agregaci nejen databázových systémů ale i SOAP a REST služeb [4]. Používá silně typovaný JavaScript (TypeScript). V roce 2019 vyhrál cenu pro nejlepší API middleware.

Název	**GitHub**		**npm**
	Search results	Topics	Dependents
Express.js	343 858	32 912	53 297
Restify.Js	2 485	275	1164
Hapi.js	9 336	661	447
LoopBack	6 461	532	273

Tabulka 2.15: Node.js REST frameworky (zdroj: vlastní tvorba)

2.2.4 Shrnutí

Nástroj pro analýzu dostupných služeb v rámci serverové části je zaměřený na výše zmiňované frameworky Spring a JAX-RS API. Proto se Spring a JAX-RS jeví jako nejlepší možnost pro analýzu *volaných WS* založených právě na těchto technologií. Analýza bude mít největší vliv na služby využívající framework Spring. To dokazuje nejen jeho popularita v rámci online

repositáře GitHub, ale také i významný počet programů nebo knihoven, které tento framework využívají a které jsou dostupné v Maven repositářích viz tabulka 2.12.

Nelze však opomenout i ostatní REST frameworky pro *Python* nebo *Node.js*. Některé z nich jsou v dnešní době velmi používané zejména k implementaci RESTových služeb. I pro ně by obdobné analyzátory mohly být do budoucna velmi přínosné.

3 CRCE

Component Repository supporting Compatibility Evaluation (CRCE) je rozšiřitelné úložiště postavené na technologii OSGi. CRCE je určeno pro ukládání softwarových komponent a kontrolu jejich kompatibility.

Tato práce se zabývá vytvořením nástroje pro analýzu kódu, jehož cílem je získat *informace* o *volaných WS* typu REST. Úložiště CRCE tyto informace může zkombinovat spolu s nástrojem pro analýzu dostupných RESTových služeb (serverové části). Ten je vytvořen v rámci diplomové práce G.H. a je zabudován jako OSGi modul do celého CRCE úložiště. Tyto dva nástroje budou schopny dodat kompletní informaci o provázání WS a jejich vzájemné kompatibility.

3.1 Vize

CRCE se snaží vyřešit problematiku kontroly kompatibility mezi softwarovými komponentami. Hlavní myšlenkou je přesunout takto výpočtově náročnou operaci od zařízení využívající dané komponenty do komponentového úložiště. Právě to lze lehce škálovat oproti cílovému zařízení, které má často v tomto omezené možnosti [28].

CRCE může být použito mnoha způsoby. Jako první lze uvést čistě lokální použití. Konkrétně to znamená, že by CRCE bylo využito v rámci firem nebo vědeckých skupin na univerzitách. V kontextu procesu CI může tento nástroj výrazně přispět ke zlepšení stability vývoje a celkové kvality vytvářeného programu.

Dalším způsobem je využití CRCE v rámci sdíleného online repositáře. Ten by mohl poskytovat informace o kompatibilitě knihoven a jejich SW komponentách.

3.1.1 Repositáře pro SW komponenty

Mezi nejvíce využívané repositáře můžeme uvést například *Maven* a *Npm*. Ani jeden z nich neposkytuje informace o kompatibilitách jednotlivých komponent. A ani neumožňuje tyto *metadata* automaticky vytvářet.

Maven repositář využívá pro ukládání SW komponent adresářovou strukturu spolu s popisem jednotlivých verzí v metadata *XML* souboru [28, 36]. Popisný soubor je předveden na ukázce kódu 3.1. Každá verze má vlastní

adresář, ve kterém se nachází samotný *Jar* soubor spolu s *pom.xml* obsahující informace jako např. název a verzi modulu, závislosti apod. Názorný příklad *pom* souboru popisující program je zobrazen na ukázce kódu ref-code:maven:pomxml.

```
<metadata modelVersion="1.1.0">
  <groupId>some.group.id</groupId>
  <artifactId>name-of-the-library</artifactId>
  <versioning>
    <latest>1.1.1</latest>
    <release>1.1.1</release>
    <versions>
        <version>1.0.0</version>
        ...
        <version>1.1.1</version>
    </versions>
  </versioning>
</metadata>
```

Kód 3.1: maven-metadata.xml (zdroj: vlastní tvorba)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>1.0.0</modelVersion>
  <artifactId>name-of-this-module</artifactId>
  <name>Title of module</name>
  <description>Description of module<description>
  <dependencies>
    <dependency>
      <groupId>GroupIdOfSomeLibraray</groupId>
      <artifactId>ArtefactID</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
  <build>
    ...
  </build>
</project>
```

Kód 3.2: pom.xml (zdroj: vlastní tvorba)

Npm má oproti Maven repositáři popsané komponenty ve formátu *JSON* [6]. Popis je v podstatě ekvivalentní výše zmíněnému *pom.xml*. Obsahuje *tagy* pro případné klienty, kteří budou chtít jen určitou verzi. Dále má v sobě informace o *názvu*, *datumu modifikace* a informace o *aktuální verzi*. Tento konfigurační soubor je zobrazen na ukázce kódu 3.3.

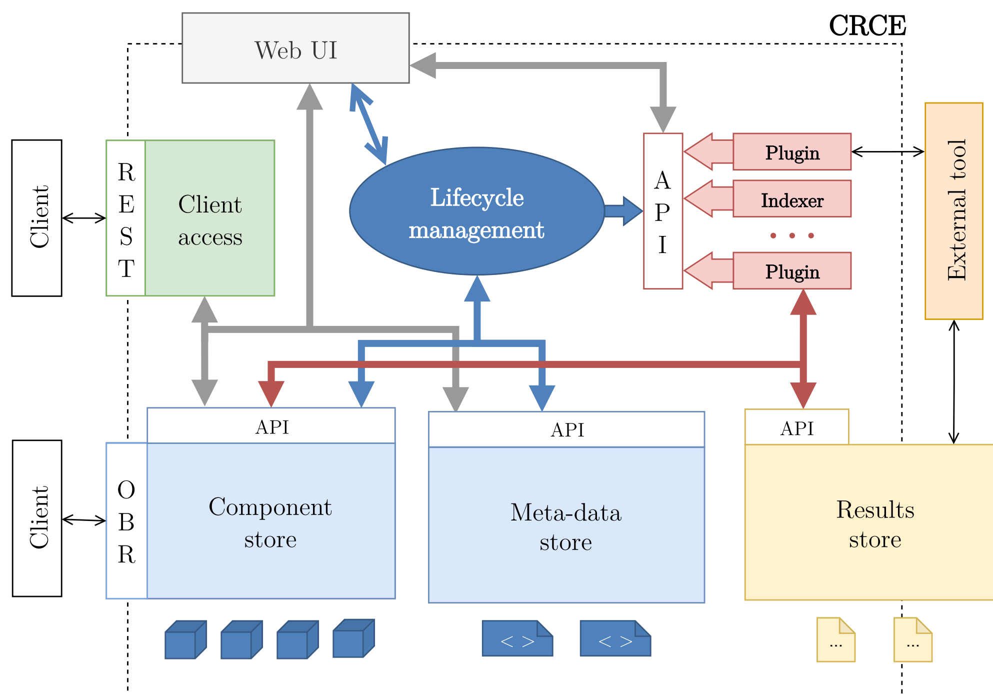
```
{
```

```
2       "dist-tags": {
3           "latest": "1.0.0"
4       },
5       "modified": "2015-05-16T22:27:54.741Z",
6       "name": "tiny-tarball",
7       "versions": {
8           "1.0.0": {
9               ...
10              "name": "tiny-tarball",
11              "version": "1.0.0"
12          }
13      }
14  }
```

Kód 3.3: Package metadata (zdroj: [6])

3.2 Architektura

Architektura úložiště CRCE je rozdělena do *tří* hlavních komponent [28]: *Component Store* (úložiště komponent), *Meta-data Store* (úložiště metadat) a *Results Store* (úložiště výsledku testů).



Obrázek 3.1: Architektura CRCE (zdroj: [28])

Tato trojice spolu s jejich vazbami na ostatní části systému se nachází na obrázku 3.1.

Definice 12 *Artefakt je v rámci CRCE úložiště definován jako libovolný soubor, který byl uživatelem nahrán do systému (Jar, zip atd.).*

Úložiště komponent se stará o ukládání fyzických komponent (artefaktů). Jeho správa probíhá v rámci modulu *crce-repository-impl*. Ten řeší celý proces ukládání dodaných souborů [48].

Definice 13 *Metadata jsou v rámci CRCE úložiště definovány jako vlastnosti, závislosti nebo výsledky testů jednotlivých komponent [48].*

Úložiště metadat hraje klíčovou roli v celém CRCE projektu. Cílem tohoto úložiště je poskytnout metadata, která lze využít pro kontrolu kompatibility SW komponent. Cílem je tak mít metadata, která obsahují následující informace [28].

- výsledek kompatibility porovnaných komponent
- vazby mezi dvojicí *klient-služba* a informace o požadované službě na straně klienta
- informace o částech, které nejsou kompatibilní

Poslední klíčovou komponentou je *úložiště výsledku testů*. To umožňuje následující [48]

- uložení výsledků
- propojení výsledků s testovaným artefaktem
- propojení výsledků s testem, který výsledek vygeneroval
- vyhledat výsledek a test, který jej vygeneroval

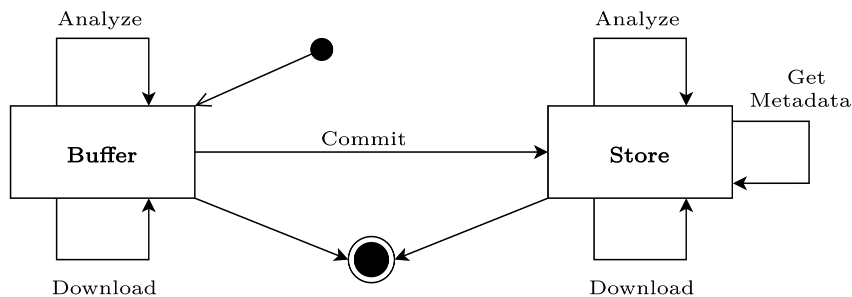
3.3 Životní cyklus

Životní cyklus artefaktu se skládá ze *čtyř* hlavních fází, a to *nahrání CRCE*, *indexace*, *vložení do úložiště*, *stažení artefaktu nebo jeho smazání*. Jedním z důležitých modulů je *web-ui*. To zajišťuje interakci s uživateli prostřednictvím webového grafického rozhraní. Pomocí něho se provede nahrání artefaktu do systému.

Po nahrání artefaktů dochází ke kontrole konzistence a jeho indexování za pomocí zaregistrovaných *indexerů* [28]. Aby byl modul detekován jako

indexer, musí být umístěn pod složkou *modules* a zároveň musí obsahovat třídu *Activator* pro OSGi. Další požadavek je, aby hlavní třída implementovala *AbstractResourceIndexer*. Potom tento modul bude brán jako jeden z indexerů. V této fázi je artefakt pouze v tzv. *Bufferu* viz obrázek 3.2. To znamená pouze to, že po restartu aplikace zmizí z databáze.

Po úspěšné indexaci lze artefakt uložit do permanentního úložiště. V této fázi mohou případní *CRCE klienti* přistupovat k těmto metadatům.



Obrázek 3.2: Životní cyklus CRCE artefaktu (zdroj: [28])

3.4 Struktura metadat

Pro strukturu metadat byl hlavní inspirací *OSGi Bundle Repository* (OBR).

Definice 14 *OSGi bundle lze definovat jako Jar archiv obsahující: spustitelnou implementaci v Javě, zdroje (resource) spolu s manifestem, který tento bundle a jeho závislosti popisuje [7].*

Cílém *OBR* je poskytnout službu pro automatickou instalaci daného bundlu. Repositář pro OSGi bundly, který se stará o jejich poskytování aplikacím, nemusí existovat jako veřejný server. Tato funkcionalita může být pouze na straně klienta, jelikož OBR dokáže číst *XML* metadata soubory, jenž jednotlivé bundly poskytují [7].

OBR definuje následující entity [7]:

- *Administrative repository*
  - hlavní část, která se stará o všechny repositáře
- *Repository*
  - poskytuje přístup ke zdrojům (resources)
- *Resource*

  - popis daného artefaktu, který má být nainstalován na dané zařízení

- *Capability*
  - pojmenovaná množina vlastností

- *Requirement*
  - požadavek na určité capability

- *Resolver*
  - objekt, který se stará o závislosti jednotlivých resources a jejich nasazení

- *Repository file*
  - XML soubor obsahující metadata pro resource

Na obrázku 3.3 je vidět, že CRCE se inspirovalo nejvíce při tvorbě komponent, jako je *Repository*, *Resource*, *Capability* a *Requirement*. Význam těchto komponent v rámci úložiště CRCE zde bude v krátkosti popsán [28, 44, 51]. Součástí jsou i ukázky metadat, která byla získána z indexace WS využívající framework *Spring*.

- *Repository*
  - poskytuje možnost ukládání jednotlivých artefaktů (resources)
  - na ukázce kódu 3.4 lze vidět konkrétní záznam
  - záznam je tvořen jedinečným identifikátorem (id) a *uri*, které odkazuje na místo, kde je artefakt uložen

```
1 {
2   "_id": "657fd85d-eaf0-4e9f-928d-
      f9b26510f03e",
3   "uri": "file:/felix/felix-framework
      -6.0.3/./felix-cache/bundle48/data/
      node0zabnn8jl2hamlkkt25rjrkdn0/"
4 }
```

Kód 3.4: Repository záznam (zdroj: vlastní tvorba)

- *Capability*

  - popisuje nabízené schopnosti pro ostatní komponenty
  - konkretní výsledek indexace WS Spring je zobrazen na ukázce kódu 3.5. Zde je uloženo jaký typ dat endpoint produkuje, podle jaké cesty se k němu přistupuje a případně jakou HTTP metodu dokáže zpracovat

```
{
...
[
    {
        "name": "produces",
        "type": "java.util.List",
        "value": ["application/json"]
    },
    {
        "name": "path",
        "type": "java.util.List",
        "value": ["/api/v2/users/getData"]
    },
    ...
    {
        "name": "method",
        "type": "java.util.List",
        "value": ["GET"]
    }
]
...
}
```

Kód 3.5: Capability záznam (zdroj: vlastní tvorba)

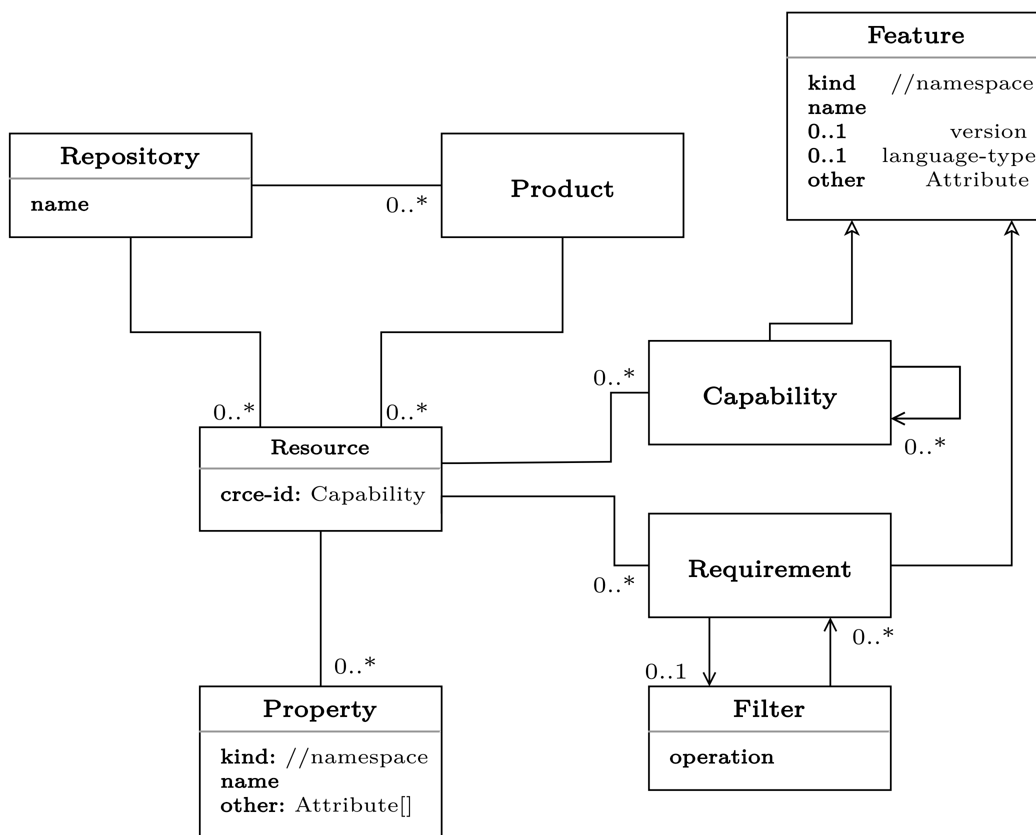
- *Requirement*
  - popisuje požadovanou funkcionalitu od jednotlivých komponent
  - bez naplnění tohoto požadavku nebude daná komponenta správně fungovat
  - ukázka není dostupná, jelikož získání požadavků na WS je cílem této práce
- *Resource*

- popisuje jednotlivé artefakty nahrané do CRCE úložiště (Jar, zip atd.)
- resource obsahuje kromě *názvu soubor* a jiných atributů také důležité informace jako *status* (v jaké fázi životního cyklu se nachází), *uri* (odkaz na umístění artefaktu) a *categories* (informace o indexerech použitých na tento artefakt) viz ukázka kódu 3.6

```
[
    {
        "name": "status",
        "type": "java.lang.String",
        "value": "buffered"
    },
    {
        "name": "uri",
        "type": "java.net.URI",
        "value": "file:/felix/felix-framework-6.0.3/felix-cache/bundle48/data/node0zabnn8jl2hamlkkt25rjrkdn0/res6741387206448579694.tmp"
    },
    {
        "name": "categories",
        "type": "java.util.List",
        "value": ["zip", "rest.client", "restimpl"]
    },
],
```

Kód 3.6: Resource záznam (zdroj: vlastní tvorba)

Na obrázku 3.3 si lze všimnout propojenosti jednotlivých komponent.



Obrázek 3.3: CRCE metadata model (zdroj: [28])

3.5 Využití OSGi frameworku

CRCE je modulární systém využívající framework *OSGi*. Ten je založen na myšlence modularity. Takto vyvíjený kód, u kterého je dbáno na dostatečnou modularitu, může být výrazně jednodušší udržovat.

OSGi

OSGi je framework, který umožňuje vyvíjet modulární Java aplikace. Celý systém je složený z tzv. *bundelů*. Ty jsou načítány za běhu pouze pokud je program aktuálně potřebuje [8]. Potom, co se systém rozhodne, že daný bundle už nechce mít dále načtený, může pomocí *Aktivátoru* tento bundle odebrat.

Obrázek 3.4 popisuje celý životní cyklus *OSGi* bundelu. Ten se skládá z pěti stavů, a to *INSTALLED*, RESOLVED, *STARTING*, *STOPPING* a *ACTIVATE*.

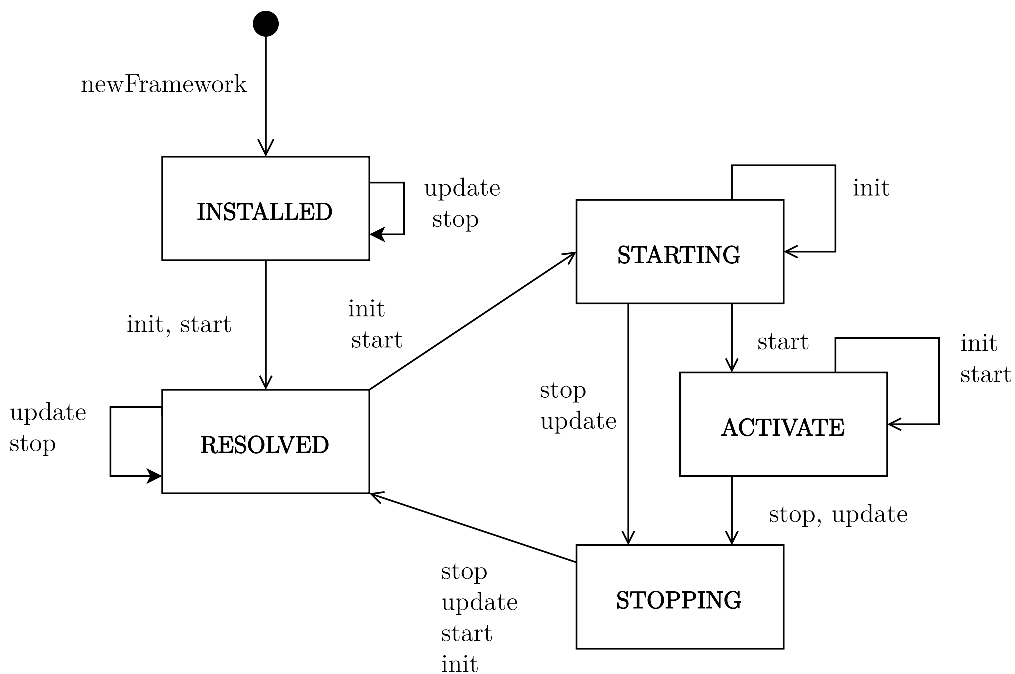
Vysvětlení jednotlivých stavů se nachází v tabulce 3.1.

Stav	Popis
INSTALLED	bunde byl úspěšně nainstalován
RESOLVED	Java třídy, které bundle potřebuje, jsou dostupné
STARTING	bundle je spouštěn pomocí BundleActivator.start
ACTIVATE	bundle je aktivován (BundleActivator dokončen)
STOPPING	metoda BundleActivator.stop je zavolána
UNINSTALLED	bundle je kompletně odstraněn

Tabulka 3.1: OSGi stavy

3.5.1 Apache Felix

Apache Felix je implementace OSGi frameworku, která rozšiřuje jeho stávající možnosti. Umožňuje navíc deklarativně registrovat, získávat a spravovat *OSGi služby*. Další funkcionalitu, kterou v rámci OSGi nástrojů pro správu služeb Felix dodává, je monitoring kontrolující jejich aktuální stav.



Obrázek 3.4: OSGi životní cyklus (zdroj: [8])

3.6 CRCE Indexery

Mezi nejdůležitější komponenty CRCE úložiště se nepochybně řadí *Indexery*. Ty mají za úkol zpracovat dodaný artefakt a získat z něj potřebná metadata.

3.6.1 crce-metadata-indexer

Tento indexer byl vytvořen v rámci původní implementace CRCE Jiřím Kučerou [48]. *Crce-metadata-indexer* se zabývá získáváním *typu* (zip, PNG atd.) indexovaného artefaktu.

3.6.2 crce-metadata-osgi-bundle

Crce-metadata-osgi-bundle se zabývá indexací výše zmíněných OSGi bundelů. Získává potřebná metadata z *manifest* souborů, které jsou přiloženy do Jar archivů [48].

3.6.3 crce-webservices-indexer

Tento modul je zaměřen na zpracování popisných souborů pro WS a získání informace o *Requirements* a *Capabilities*. Je kompatibilní například s JSON-WSP description object, WSDL a WADL [51].

3.6.4 crce-restimpl-indexer

Crce-restimpl-indexer je modul, který je zaměřen primárně na zpracování dat z anotací nacházejících se v přeloženém Java kódu. V anotacích jsou ukryty informace o endpointech, které indexovaná služba poskytuje [44].

4 Přístupy pro analýzu kódu

Hlavním cílem práce je vytvořit nástroj pro rekonstrukci volaných WS. Pro získání těchto informací existují dva přístupy. Prvním přístupem je analýza *přeloženého kódu* a druhým je zpracování *zdrojových kódů*.

Tato kapitola se zabývá popisem dostupných možností analýzy kódu pro jazyk Java. Existují dvě možné metody řešení, a to *dekompilace* a *zpracování byte-kódu (interpretace)*. Každý z těchto přístupů má své výhody, to samé platí i pro dostupné nástroje, které lze k tomu využít. Nejprve je však nutné uvést samotný jazyk Java a upřesnit jeho životní cyklus, který začíná kompilací a končí spuštěním výsledného kódu.

4.1 Java

Java je vysokoúrovňový programovací jazyk, který se řadí do skupiny objektově orientovaných jazyků založených na třídách. Tento jazyk je silně a staticky typovaný. Pro správu paměti využívá tzv. *garbage collector*, který si hlídá počet referencí na objekty. V případě, že objekt není používán, provede následnou dealokaci z paměti. Java se kompiluje do *byte-kódu*, který se spouští přes konkrétní implementaci Java Virtual Machine (JVM) [27]. Následující text popisuje životní cyklus skládající se z kompilace a spuštění v rámci JVM spolu s případnou optimalizací kódu.

4.1.1 Kompilace

První částí životního cyklu je kompilace kódu. Popis překladu čerpá z oficiálního popisu *OpenJDK*(implementace Javy) kompilace [17]. Překlad kódu se skládá ze *tří* fází [17]. V první fázi jsou všechny zdrojové soubory přečteny a převedeny do *syntaktických stromů*. Každý z těchto stromů je předán *Enteru* [17]. Ten se postará o načtení symbolů spojených s definicemi názvů metod, tříd apod. Takto propojené syntaktické stromy s odkazy (symboly) jsou předány do tzv. TODO listu.

Druhou fází je zpracování anotací za použití procesorů, které jsou tomu určené.

Poslední fází je samotná analýza kódu a generování výsledných přeložených (*class*) souborů. Ta se provede za pomocí pěti visitorů *Attr*, *Flow*, *TransTypes*, *Lower* a *Gen*.

Attr zpracuje syntaktické stromy, a to nejprve třídám na nejvyšší úrovni. V rámci této části jsou také detekovány syntaktické chyby.

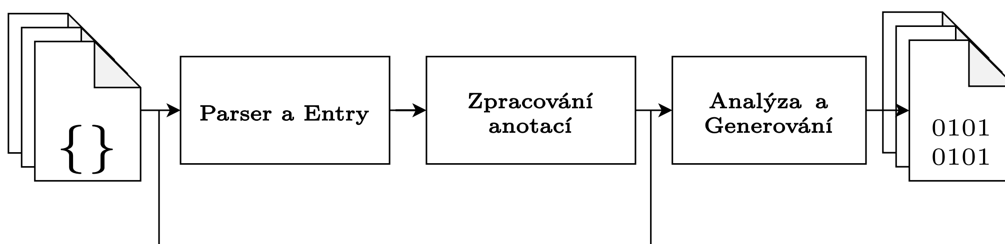
Pokud se neprojevily žádné nesrovnalosti v předchozí fázi, přijde na řadu *Flow* visitor. Ten zpracovává už určité třídy a kontroluje přiřazení proměnných a případný nedosažitelný kód.

TransTypes se stará o přeložení *generických* typů do již specifický tříd.

V dalším visitoru se zpracovává "syntaktický cukr". Zasahuje se tak do syntaktických stromů, kde probíhá nahrazení jejich částí (podstromů) za stromy, které jsou jednodušší a zároveň funkčně ekvivalentní. V rámci této části se také vytváří syntetické třídy, které si drží dílčí anotace pro každý *balík* (package) zvlášť.

Kód pro jednotlivé metody je generován pomocí *Gen* visitoru. Pokud tento proces bude úspěšně dokončen, dojde k zapsání výsledného byte-kódu do výstupního souboru.

Kompilaci spolu s jednotlivými kroky popisuje obrázek 4.1.



Obrázek 4.1: Kompilace Java (zdroj:[17])

4.1.2 Java Virtual Machine a načtení třídy

Java Virtual Machine (JVM) se snaží interpretovat přeložené zdrojové kódy (byte-kód). Díky tomu *byte-kód* není závislý na platformě, kde byl zkompilován. Je závislý pouze na kompilátoru, který generuje byte-kód a konkrétní implementaci JVM.

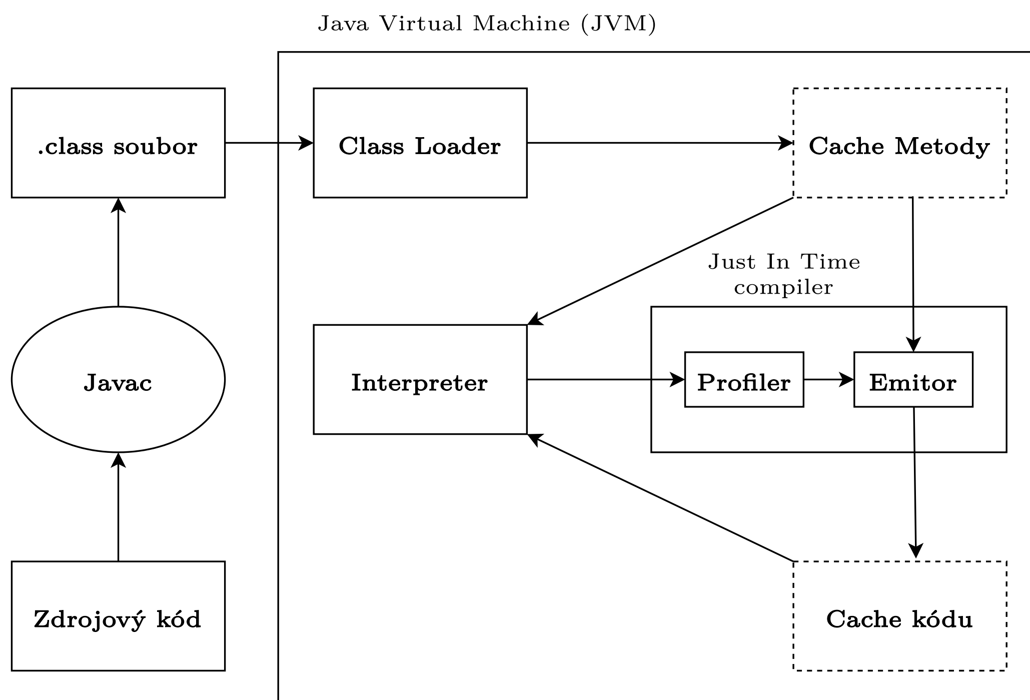
Před samotným spuštění aplikace se musí nastartovat JVM. O toto spuštění se postará operační systém. Nastaví se virtuální prostředí a provede se inicializace. Po inicializaci procesu přijde na řadu načtení hlavní třídy (Main) pomocí *Bootstrap classloaderu*. Ten obsahuje základní třídy jako je `java.lang.Object`, *Class* a *ClassLoader* [24, 55]. Bootstrap vytvoří nový *aplikační* classloader, který se stará o další načítání uživatelských tříd.

V rámci optimalizace některých částí kódu, JVM využívá Just-In-Time kompilaci.

4.1.3 Just-In-Time kompilace

S příchodem virtuálního stroje (VS) *HotSpot* byla zavedena Just-In-Time (JIT) kompilace. JIT se snaží optimalizovat jen určité části kódu. Případná optimalizace se zaměřuje pouze na *metody* a *smyčky*. Detekce částí pro optimalizaci probíhá v rámci interpretace kódu. Tehdy se začne provádět tzv. *monitoring aplikace*, který celé JIT kompilaci předchází. V rámci jeho běhu se kontroluje počet volání metod a smyček. Pokud dojde k překročení určité hranice počtu volání, přijde na řadu přeložení těchto často volaných částí z byte-kódu do strojového kódu [24]. Takto přeložený kód je poté uložen do mezi-paměti (cache) spolu s nativním kódem JVM [24]. Po naplnění mezipaměti dojde k přerušení JIT kompilací, a to znamená, že další optimalizace již není možné provést.

Průběh celého procesu od kompilace až po spuštění programu zobrazuje obrázek 4.2.



Obrázek 4.2: Životní cyklus zdrojového (Java) kódu (zdroj: [24])

4.1.4 Byte-kód

Jak bylo již v úvodu této kapitoly zmíněno, *byte-kód* je výsledkem kompilace zdrojového kódu v Javě. Přeložený zdrojový kód se skládá z jednotlivých instrukcí. Ty potom zpracovává JVM, které je interpretuje a případně ho

přeloží do strojového kódu. Z toho vyplývá, že informace, které instrukce prezentují, jsou dostatečné pro budoucí spuštění programu. Kompilátor Javy se kromě klasického přeložení do *byte-kódu* zabývá i vyřešením jednoduchých výrazů v rámci přímo vkládaných konstant (text. řetězec, čísla, boolean).

Následující část textu se zabývá samotnými *instrukcemi* byte-kódu. Mezi ty, které mohou nejvíce pomoci této práci patří instrukce pro *metody*, *proměnné*, *atributy třídy* a *konstanty*.

Definice 15 *Zásobník operandů je datová struktura typu LIFO, která se využívá pro ukládání dočasných výsledků instrukcí a pro přípravu parametrů metod [55].*

Definice 16 *Linkování je proces, který vezme třídu nebo rozhraní a začlení ho do run-time prostředí. Potom s nimi běžící program může pracovat [55].*

Definice 17 *Constant-pool se nachází v každém zkompilovaném souboru (class). Jedná se o tabulku, která má v sobě uloženy veškeré konstanty důležité pro chod programu jako například: názvy metod, názvy tříd, řetězce apod. viz ukázka kódu 4.1.*

Definice 18 *Rámec se využívá k ukládání dat, částečných výsledků nebo pro dynamic linking. Každý rámec vlastní pole lokálních proměnných, operandový zásobník a odkaz do run-time constant-poolu [55].*

```
1  #1 = Methodref     #6.#15 // java/lang/Object."<init
   >":()V
2  ...
3  #3 = String        #18
4  #5 = Class         #21
5  #6 = Class         #22
6  #7 = Utf8          <init>
7  ...
8  #15 = NameAndType #7:#8
9  ...
10 #18 = Utf8         Hello world
11 ...
12 #21 = Utf8         HelloWorld
13 #22 = Utf8         java/lang/Object
```

Kód 4.1: Constant Pool (zdroj: vlastní tvorba)

Instrukce pro konstanty

Instrukce pro konstanty jsou využívány napříč celým byte-kódem. Využívají se ať už k přístupu proměnné dané metody nebo například pro manipulaci s polem. *CONST* se řadí mezi hlavní instrukce pro práci s konstantami. Skládá se z pěti instrukcí, kde každá pracuje s jiným datovým typem, a to je možné pozorovat na tabulce 4.1. Využitím této instrukce se zabývá ukázka kódu 4.2 přeložená do kódu 4.3, která je zaměřena na práci s poli a instrukcí *LDC*.

<table>
<tr><th>Skupina</th><th>Význam</th><th>Instrukce</th><th>Typ</th><th>Omezení</th></tr>
<tr><td rowspan="5">CONST</td><td rowspan="5">Vložení hodnoty na zásobník</td><td>ACONST_NULL</td><td>Reference na objekt</td><td>null</td></tr>
<tr><td>DCONST</td><td>Double</td><td>0.0, 1.0</td></tr>
<tr><td>FCONST</td><td>Float</td><td>0.0f, 1.0f, 2.0f</td></tr>
<tr><td>ICONST</td><td>Integer</td><td>-1, 1, 2, 3, 4, 5</td></tr>
<tr><td>LCONST</td><td>Long</td><td>0L, 1L</td></tr>
</table>

Tabulka 4.1: Druhy instrukcí CONST (zdroj: vlastní tvorba)

Další velmi důležitou instrukcí je *LDC*. Ta se stará o načtení řetězců, čísel a symbolických odkazů tříd z *constant-poolu* přímo do zásobníku operandů. Tyto konstanty jsou důležité pro výsledný *byte-kód*, jelikož jsou zde uloženy i *symbolické odkazy na třídy* viz tabulka 4.2. *LDC* se dělí na tři instrukce, přičemž *LDC2_W* je omezena jen na typ *long* nebo *double*, to zobrazuje tabulka 4.2.

<table>
<tr><th>Skupina</th><th>Význam</th><th>Instrukce</th><th>Typ</th></tr>
<tr><td rowspan="3">LDC</td><td rowspan="3">Vložení konstanty na zásobník</td><td>LDC</td><td rowspan="2">int, float, reference na string, symbolický odkaz na třídu</td></tr>
<tr><td>LDC_W</td></tr>
<tr><td>LDC2_W</td><td>long nebo double</td></tr>
</table>

Tabulka 4.2: Druhy instrukcí LDC (zdroj: vlastní tvorba)

```java
public class MainConst {
    public static void main(String[] var0) {
        String[] var1 = new String[]{"
            LDC_STORED_STRING"};
        String var2 = var1[0];
    }
}
```

Kód 4.2: Zdrojový kód pracující s konstantami (zdroj: vlastní tvorba)

```

//String[] var1 = new String[]{"LDC_STORED_STRING"};
iconst_1
anewarray java/lang/String
dup
iconst_0
ldc "LDC_STORED_STRING" (java.lang.String)
aastore
astore1
//String var2 = var1[0]
aload1
iconst_0
aaload
astore2

return
```

Kód 4.3: Přeložený kód vycházející z ukázky kódu 4.2 (zdroj: vlastní tvorba)

Instrukce pro proměnné

Instrukce, které se zaměřují na manipulaci s proměnnými jsou instrukce *STORE* a *LOAD*. Ty jsou složeny z dalších *pod-instrukcí* upřesňující typ dat, se kterými tato dvojice pracuje.

Kontext těchto instrukcí je definován tzv. rámcem. Jak je z definice 18 patrné, jednotlivé instrukce a proměnné jsou provázány vždy s určitou metodou. Každá proměnná v rámci tohoto kontextu má svůj vlastní *index*. Ten je reprezentován *celým kladným číslem*. Tímto indexem se jednotlivé proměnné načítají a ukládají.

Instrukce *LOAD* provádí načtení proměnných a jejich obsah vloží do *zásobníku operandů*. Tabulka 4.3 ukazuje, že tato instrukce se skládá z dílčích

pod-instrukcí zaměřených na načítání dat z *pole* nebo klasické *proměnné*. Typ těchto dat je určen názvem příslušné instrukce např. *ILOAD*, který načítá hodnotu typu *Integer*.

Skupina	**Význam**	**Instrukce**	**Práce s typem**
LOAD	Načtení hodnoty z proměnné	ILOAD	Integer
		LLOAD	Long
		FLOAD	Float
		DLOAD	Double
		ALOAD	Reference na objekt
	Načtení hodnoty z pole	IALOAD	Integer
		LALOAD	Long
		FALOAD	Float
		DALOAD	Double
		AALOAD	Reference na objekt
		BALOAD	Boolean
		CALOAD	Char
		SALOAD	Short

Tabulka 4.3: Druhy instrukcí LOAD (zdroj: vlastní tvorba)

Pro lepší představu toho, jak tato instrukce funguje, je uvedena ukázka přeloženého kódu 4.5 spolu s *LOAD* instrukcemi. Zdrojový kód, který byl do této podoby přeložen, zobrazuje ukázka kódu 4.4. Proces nejprve začíná deklarací velikosti pole pomocí `iconst_3`. Poté následuje instrukce `anewarray`, která se postará o vytvoření objektu. Následně jsou do pole přiřazeny objekty pod určitými indexy. V poslední části lze vidět využití instrukce `aaload`, která načte referenci na pole do zásobníku operandů.

```
1 Object arrayOfObj[] = {new Object(), new Object(),
    new Object()};
2 anotherArrayOfObj = arrayOfObj;
```

Kód 4.4: Zdrojový kód pracující s polem (zdroj: vlastní tvorba)

```
1 iconst_3
2 anewarray java/lang/Object
3 dup
4 iconst_0
5 new java/lang/Object
6 dup
7 invokespecial java/lang/Object.<init>()V
```

```
8  aastore
9  dup
10 iconst_1
11 new java/lang/Object
12 dup
13 invokespecial java/lang/Object.<init>()V
14 aastore
15 dup
16 iconst_2
17 new java/lang/Object
18 dup
19 invokespecial java/lang/Object.<init>()V
20 aastore
21 astore2
22
23
24 aload2
25 iconst_0 //arrayOfObj
26 aload2
27 iconst_1 //anotherArrayOfObj
28 aaload
29 aastore //anotherArrayOfObj = arrayOfObj
```

Kód 4.5: Přeložený kód vycházející z ukázky kódu 4.4 (zdroj: vlastní tvorba)

Instrukce *STORE* je určena pro ukládání hodnot nacházející se aktuálně na *zásobníku operandů*. Tato instrukce je členěna totožně jako instrukce *LOAD*, to ostatně ukazuje tabulka 4.4. Pro upřesnění jejího fungování je určena ukázka kódu 4.4. Na té je vidět, jakým způsobem se provádí přiřazení reference na pole do jiné proměnné (`anotherArrayOfObj = arrayOfObj`). Průběh je takový, že se přečte ze zásobníku reference na pole spolu s indexem a přiřadí se do příslušné proměnné.

Tabulka 4.4 ukazuje jasný vzor, jakým jsou instrukce pojmenovány. Počáteční písmena *I, L, F, D, B, C, S* označují práci s primitivními datovými typy. Pokud instrukce začíná znakem *A*, bude pracovat s objektem. V případě předpony *AA* je instrukce určená pro činnost zaměřenou na pole.

Skupina	Význam	Instrukce	Práce s typem
STORE	Uložení hodnoty do proměnné	ISTORE	Integer
		LSTORE	Long
		FSTORE	Float
		DSTORE	Double
		ASTORE	Reference na objekt
	Uložení hodnoty do pole	IASTORE	Integer
		LASTORE	Long
		FASTORE	Float
		DASTORE	Double
		AASTORE	Reference na objekt
		BASTORE	Boolean
		CASTORE	Char
		SASTORE	Short

Tabulka 4.4: Druhy instrukcí STORE (zdroj: vlastní tvorba)

Instrukce pro volání metod

K volání metod se využívají instrukce *INVOKE*. Tabulka 4.5 zobrazuje dělení jejich jednotlivých typů instrukcí. Typ instrukce je závislý na druhu metody, podle které jí překladač generuje. Analýza chování instrukce pro volání metod vychází z třídy, která se nachází v ukázce kódu 4.6.

Při inicializaci objektu se nejprve volá konstruktor. Ten se zpravidla v byte-kódu ukazuje pod názvem `init`. Pro zavolání této metody se vytvoří instrukce typu *INVOKESPECIAL*. Ta zajistí inicializaci potřebných dat daného objektu. V případě, že objekt obsahuje *statické* atributy, vytvoří překladač pro jejich inicializaci metodu `cinit`. Její volání není v byte-kódu vidět. Je to z toho důvodu, že je tato metoda spuštěna za běhu programu, při prvním využití třídy. Není tak pro ni vygenerována instrukce, která by mohla být viděna v přeloženém kódu. Chování instrukce *INVOKESPECIAL* může být pozorováno na ukázce kódu 4.7.

```
1 class Test implements TestI {
2     public static String test = "TEST";
3     @Override
4     public String interfaceMethod() {
5         return "TEST";
6     }
7     public String virtualMethod(){
8         return "TEST";
```

```
9     }
10    public static String staticMethod(){
11        return "TEST";
12    }
13    static { //<cinit> uvnitr tridy Test, pridano po
          prekladu
14        ldc "TEST" (java.lang.String)
15        putstatic Test.test:java.lang.String
16        return
17    }
18
19 }
```

Kód 4.6: Ukázková třída (zdroj: vlastní tvorba)

Instrukce *INVOKEVIRTUAL* je vytvořena pro volání metody inicializovaných objektů. Tato instrukce se nachází v ukázce kódu 4.7. Další lze zmínit například instrukci *INVOKEINTERFACE*. Proměnná obsahující referenci na objekt, může být otypována na rozhraní, které třída tohoto objektu implementuje. Pokud se metoda tohoto rozhraní zavolá, překladač vytvoří instrukci *INVOKEINTERFACE*.

Poslední instrukcí pro volání metod je *INVOKESTATIC*. Jak lze z názvu vyčíst, tato instrukce je generována překladačem při volání statické metody. Ukázka kódu 4.7 zobrazuje, jak vygenerovaná instrukce vypadá.

Ještě existuje instrukce *INVOKEDYNAMIC* pro vytváření dynamických metod (lambda metody).

```
1  //Test test = new Test();
2  invokespecial Test.<init>()V
3  astore1
4  aload1
5  astore2 //TestI interfaceCall = test;
6
7  //test.virtualMethod()
8  aload1
9  invokevirtual Test.virtualMethod()Ljava/lang/String;
10
11 //interfaceCall.interfaceMethod()
12 aload2
13 invokeinterface TestI.interfaceMethod()Ljava/lang/
       String;
14
```

```
15 //Test.staticMethod();
16 invokestatic Test.staticMethod()Ljava/lang/String;
17 pop
```

Kód 4.7: Vytváření pole objektů včetně byte-kódu (zdroj: vlastní tvorba)

Skupina	Význam	Instrukce	Typ
INVOKE	Volání metody	INVOKEINTERFACE	volání prostřednictvím objektu, který je otypován pomocí rozhraní
		INVOKESPECIAL	inicializace
		INVOKESTATIC	statická metoda
		INVOKEVIRTUAL	volání metody přes objekt
		INVOKEDYNAMIC	vytvoření lambda metody

Tabulka 4.5: Druhy instrukcí INVOKE (zdroj: vlastní tvorba)

Instrukce pro atributy třídy

Instrukce pro práci s atributy tříd jsou rozděleny do dvou skupin *GET* a *PUT*. Tabulka 4.6 zobrazuje další dělení těchto metod na *statické* a *dynamické*.

GET a *GETSTATIC* jsou určeny pro získání atributu dané třídy. *GETSTATIC* se využívá pro získání statických atributů. Obdobně se chovají *PUT* a *PUTSTATIC*, ty ale na-rozdíl od zmíněných instrukcí provádějí nastavení atributů. Chování v praxi naznačuje zdrojový kód 4.8 a jeho přeložená verze 4.9.

Význam	Instrukce	Typ
Získání atributu třídy	GETFIELD	dynamický
	GETSTATIC	statický
Nastavení atributu třídy	PUTFIELD	dynamický
	PUTSTATIC	statický

Tabulka 4.6: Instrukce pro práci s atributy třídy (zdroj: vlastní tvorba)

```
1 class MainNext {
2   public static void main(String[] var0) {
```

```
3         Field var1 = new Field();
4         String var2 = var1.field1;
5         String var3 = Field.field2;
6         Field.field2 = "TEST";
7         var1.field1 = "TEST2";
8     }
9 }
```

Kód 4.8: Zdrojový kód pro práci s atributy (zdroj: vlastní tvorba)

```
1  //Field var1 = new Field();
2  invokespecial Field.<init>()V
3  astore1
4
5  //String var2 = var1.field1;
6  aload1
7  getfield Field.field1:java.lang.String
8  astore2
9
10 //String var3 = Field.field2;
11 getstatic Field.field2:java.lang.String
12 astore3
13
14 //Field.field2 = "TEST";
15 ldc "TEST" (java.lang.String)
16 putstatic Field.field2:java.lang.String
17
18 //var1.field1 = "TEST2";
19 aload1
20 ldc "TEST2" (java.lang.String)
21 putfield Field.field1:java.lang.String
```

Kód 4.9: Přeložený kód vycházející z ukázky kódu 4.8 (zdroj: vlastní tvorba)

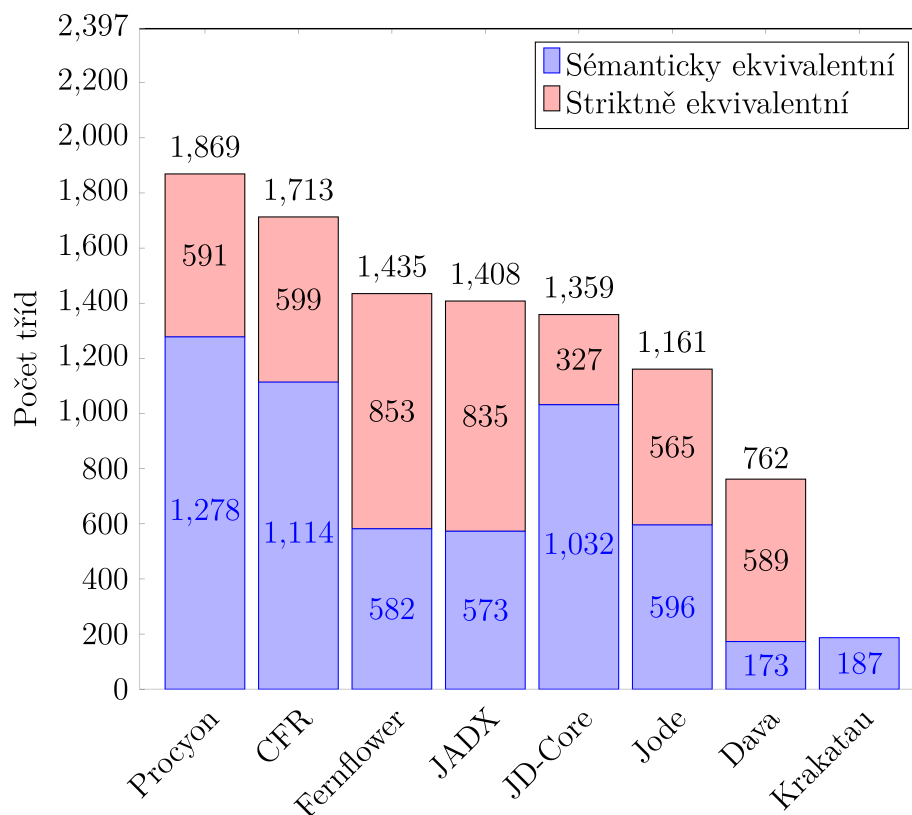
4.2 Dekompilace

Definice 19 *Dekompilace je inverzní proces k překladu, který zpětně formuje instrukce byte-kódu do původní podoby (zdrojového kódu) [42].*

Rekonstrukce API volaných WS se může ubírat směrem k analýze zdrojových kódů. Ty, ale většinou nemusí být k dispozici, proto se nabízí možnost

využít proces dekompilace pro získání zdrojových kódů.

Pro jazyk Java to znamená, přeložený kód (byte-kód) převést zpět do původní podoby zdrojového kódu. Jak se ukázalo, tento proces není stoprocentní [43]. Jeden z důvodu může být například způsob kompilování kódu a jeho optimalizace v rámci překladu. Obrázek 4.3 zobrazuje porovnání jednotlivých dekompilátorů. Na něm je patrné, že žádný z nich nedokáže program dekompilovat do podoby, kdy výsledkem rekompilace takto získaného kódu je totožný byte-kód. Pro lepší pochopení obrázku 4.3 je potřeba si nejprve nadefinovat následující pojmy.



Obrázek 4.3: Výsledky dekompilace (zdroj: [43])

Definice 20 *Sémanticky ekvivalentní kód, je kód, pro který platí, že zkompilovaný kód projde stejnými funkčními testy jako ten, se kterým je srovnáván. Přeložený kód (byte-kód) nemusí obsahovat stejnou posloupnost instrukcí.*

Definice 21 *Striktně ekvivalentní kód, je takový kód, který po kompilaci má totožný byte-kódu jako ten, se kterým je srovnáván.*

Procyon má nejvyšší celkovou úspěšnost dekompilace. Z celkových 2397 *class* souborů dokázal úspěšně dekompilovat 1869, tedy necelých *78%*. Jak

ukazují data, *Procyon* má problém dekompilovat byte-kód do podoby striktně ekvivalentního kódu, to má ve výsledcích pouze minoritní zastoupení (*32%*).

Dostupné výsledky ukazují, že nejpřesněji oproti původnímu kódu dokáže dekompilovat dvojice *Fernflower* a *JADX* (*61%* striktně ekvivalentní a *41%* sémanticky ekvivalentní).

Shrnutí

Dekompilaci Java byte-kódu nelze na základě předchozích informací brát jako spolehlivé řešení, které zvládne pokaždé dokonale zrekonstruovat původní kód. Pro účely této práce je tak tento přístup *nevhodný*. Může být využit například v těch oblastech, kde není vyžadována spolehlivost.

Pokud by existoval ideální dekompilátor, přispělo by to k obecnějšímu přístupu v rámci *Rekonstrukce API volaných webových služeb*, který by mohl pokrýt širší skupinu programovacích jazyků. V porovnání ať už analýzy byte-kódu nebo strojového kódu se přístup zaměřený na zdrojový kód může jevit jako rychlejší řešení. Nemusí se procházet jednotlivé instrukce, a to může výrazně urychlit celkový chod analýzy. Tento přístup lze uplatnit pouze na oblast, kde jsou zdrojové kódy vždy k dispozici.

4.3 Zpracování byte-kódu

Jak bylo již v úvodu této kapitoly zmíněno, byte-kód je výsledkem překladu zdrojového kódu v Javě. Ukázalo se, že je dekompilace tohoto kódu velmi nespolehlivá. Poslední možností tak zbývá *analýza byte-kódu*. Jak ukázala předchozí sekce zaměřená na jazyk Java, byte-kód v sobě uchovává všechny potřebná data pro účely této práce. Lze z něj získat informace např. o konstantách, volaných metodách a vytvářených objektů. Ty tak mohou být využity v rámci analýzy API volaných WS.

4.3.1 ASM

Projekt ASM byl vytvořen během *PhD* studia Eric Brunetonem [32]. Celý projekt spadá do kategorie *open source* a může ho tak šířit a využít kdokoliv.

Knihovna ASM má za cíl usnadnit *analýzu*, *generování* a *transformaci* Java byte-kódu. Podle oficiální dokumentace je tento nástroj vhodný zejména pro *syntaktickou* nebo *sémantickou* analýzu [31]. Pro řešení rekonstrukce API volaných WS je tak tento přístup ideální.

ASM je postaveno na tzv. *ClassVistorech*, které jsou znázorněny na následující ukázce kódu 4.10. Tyto třídy jsou určeny pro postupné prochá-

zení jednotlivých tříd a metod v rámci byte-kódu. Pro získání informací je potřeba vytvořit vlastní třídu, která bude ClassVisitor rozšiřovat o vlastní funkcionalitu.

```
1 public abstract class ClassVisitor {
2     public ClassVisitor(int api);
3     public ClassVisitor(int api, ClassVisitor cv);
4     public void visit(int version, int access,
        String name,String signature, String
        superName, String[] interfaces);
5     public void visitSource(String source, String
        debug);
6     public void visitOuterClass(String owner, String
         name, String desc);
7     AnnotationVisitor visitAnnotation(String desc,
        boolean visible);
8     public void visitAttribute(Attribute attr);
9     public void visitInnerClass(String name, String
        outerName,String innerName, int access);
10    public FieldVisitor visitField(int access,
        String name, String desc,String signature,
        Object value);
11    public MethodVisitor visitMethod(int access,
        String name, String desc,String signature,
        String[] exceptions);
12    void visitEnd();
13 }
```

Kód 4.10: ClassVisitor (zdroj: [31])

Jak lze vyčíst z ukázky kódu 4.10, ASM poskytuje při analýze kódu podrobné informace spojené s danou třídou. Z těchto dat může být získané např. *jméno*, *popis*, *vlastník*, *atributy* apod.

Pro podrobnější informace o *atributech* třídy musí být rozšířena třída *FieldVisitor*, která bude využita v rámci ClassVisitoru. FieldVisitor lze použít pro zpracování dat z anotací.

Mezi nejdůležitější visitory patří `MethodVisitor`. Ten umožňuje analyzovat *deklarace lokálních proměnných*, *vytváření pole volání metod* a *návratovou hodnotu*.

Díky tomu, že ASM dodává zmíněným informacím kontext např. o umístění dané třídy, kde je volaná metoda atd., mohou být tak vytvořeny komplexní modely chování zpracovávaného programu.

Projekt ASM je stále aktivní a téměř na denní bázi se objevují aktualizace zlepšující tuto knihovnu.

4.3.2 BCEL API

Byte Code Engineering Library (BCEL API) patří mezi nejstarší byte-kód analyzátory [1]. Používá se zejména pro statickou analýzu a dynamické vytváření nebo transformace Java *class* souborů.

Zpracování byte-kódu zde probíhá ve třech fázích [32]. V první fázi se pole bytů reprezentující danou třídu převede do struktury, která je uložena v paměti. Tato struktura je složena z jednotlivých uzlů, které jsou zabaleny do objektů. Tato abstrakce jde až na úroveň instrukcí. Každá instrukce je tak obalená objektem, který ji reprezentuje. Ve druhé fázi je tento model zpracován a jsou nad ním dále prováděny různé operace. V poslední fázi proběhne opětovná serializace těchto objektů [32].

Eric Bruneto se v rámci vývoje nástroje ASM zabýval i měřením toho, jak jsou konkurenční knihovny rychlé. Měření probíhalo pro *1155* tříd na zařízení s *JDK1.3.1 Hotspot VM on Linux 2.4.9,on a Pentium III 1 GHz.* Zároveň zavedl *čtyři* případy, podle kterých se měřila celková doba běhu [32].

- Případ (a) - využití standardního *ClassLoaderu*
- Případ (b) - serializace a deserializace každé třídy před jejím načtením, včetně (a)
- Případ (c) - přepočítání maximální velikosti zásobníku, včetně (b)
- Případ (d) - přidání počítadla do každé třídy na začátek každé metody spolu s instrukcí pro inkrementaci, včetně (c)

Knihovna	**Případy [s]**			
	(a)	**(b)**	**(c)**	**(d)**
ASM	1,98	3,22	3,29	3,26
BCEL	1,98	16,6	19,2	16,8

Tabulka 4.7: Porovnání knihoven (zdroj: [32])

Výsledky z tabulky 4.7 jasně ukazují, že *BCEL* za *ASM* velmi zaostává a to téměř ve všech případech.

4.3.3 CRCE indexer webových služeb typu REST

Tento modul, který je určen pro indexaci RESTových WS, byl vyvinut v rámci CRCE úložiště. Snaží se získat informace o dostupných WS (serverová část) procházení byte-kódu. Při zpracování se zaměřuje primárně na *anotace* tříd. Tato práce se snaží i o částečnou interpretaci těla metody, pro případné nastavení dalších parametrů endpointu. Touto interpretací se získávají informace o parametrech hlavičky nebo těla požadavku [44].

Cílem práce je vytvořit nástroj, který bude tento modul doplňovat o informace z protější strany, tedy ze strany klienta. Nabízí se tak možnost využít část stávající implementace a spolu s tím zajistit porovnatelnost dat dodaných z obou modulů.

Visitor

Indexer využívá pro analýzu byte-kódu výše zmíněnou knihovnu *ASM*. Pomocí vlastních tříd `MyClassVisitor`, `MyMethodVisitor`, `MyFieldVisitor` doluje dostupná data ukryta v byte-kódu.

```
1 visit(int version, int access, String name,
2 String signature, String superName,
3 String[] interfaces)
4
5 visitField(int access, String name, String desc,
     String signature, Object value)
6
7 visitMethod(int access, String name, String desc,
     String signature, String[] exceptions)
```

Kód 4.11: MyClassVisitor (zdroj: [44])

`MyClassVisitor`, rozšiřuje abstraktní třídu `ClassVisitor`. Cílem této třídy je naplnit vytvořené struktury (třídy) `ClassStruct`, `Method` atd. Třída `ClassStruct` v sobě uchovává *seznam metod*, *atributy* a *rozhraní, které implementuje*.

Třída `Method` je oproti ClassStruct zaměřená na data, která lze získat `MethodVisitorem` viz ukázka kódu 4.12.

```
1 visitLocalVariable(String name, String desc,...)
2 visitFieldInsn(int opcode, String owner,...)
3 visitMethodInsn(int opcode, String owner,..)
4 visitParameterAnnotation(int paramIndex, String desc
     ,...)
```

```
5 visitIntInsn(int opcode, int operand)
6 visitVarInsn(int opcode, int var)
7 visitInsn(int opcode)
```

Kód 4.12: MyMethodVisitor (zdroj: [44])

Veškeré informace o možnostech `MethodVisitoru` jsou popsány v sekci zaměřené na knihovnu ASM.

Další třídy jsou již určeny na zpracování anotací, částečné interpretace kódu a vnitřní logice tohoto modulu.

4.3.4 Shrnutí

Pro udržení podobného směru, který byl určen tímto indexer je zvolena knihovna ASM. Tato volba je nejen vhodná pro udržení konzistence celého projektu, ale je i významně rychlejší než zmíněná knihovna *BECL*.

Výše představený indexer pro WS má určitou část kódu, která by šla využít i v rámci řešení rekonstrukce volaných API WS. Platí to zejména pro visitory, kde logika bude v podstatě totožná. Lišit se budou jen struktury pro uchovávání dat. Část kódu u visitorů nebude pro řešení vůbec potřeba. Například anotace se u klientských částí WS nevyskytují.

5 Návrh algoritmu pro analýzu API volaných WS

Cílem této práce je vytvořit nástroj pro *Rekonstrukci API volaných webových služeb*. Rekonstrukce je zaměřena na programovací jazyk Java, a na základě předchozích analýz ubírá směrem zpracování *byte-kódu*. Tato kapitola se zaměřuje na výsledný algoritmus a analýzu, která vedla k jeho vytvoření.

5.1 Analýza byte-kódu

Tato část se zabývá hledáním informací v byte-kódu, které by mohly být využity v rámci *Rekonstrukce API volaných WS*. Tyto informace jsou porovnány vůči tomu, co nabízí zdrojový kód dané implementace. Následující analýza bude zaměřena na instrukce, které jsou určeny pro *metody*, *proměnné*, *konstanty*, *pole* a *atributy třídy*.

5.1.1 Instrukce pro konstanty

Konstanty mají klíčovou roli v celé práci. Veškeré *URL* adresy endpointů, nastavení hlaviček apod. mají původ v konstantách uložených v *constant-poolu*. Ukázka byte-kódu 5.2 zobrazuje případné využití konstant pro účely nastavení hlavičky dotazu. Zdrojový kód přeložené části je možné vidět na ukázce kódu 5.1.

```
1 public List getInfo() {
2 String url = "http://localhost:8090/api/user/users";
3 HttpHeaders headers = new HttpHeaders();
4 headers.add("Content-Type", "application/json");
5 }
```

Kód 5.1: Příklad využití konstant (zdroj: vlastní tvorba)

```
1  ldc "Content-Type"
2  ldc "application/json"
3  invokevirtual org/springframework/http/HttpHeaders.
      add(Ljava/lang/String;Ljava/lang/String;)
```

Kód 5.2: Nastavení hlavičky požadavku (zdroj: vlastní tvorba)

5.1.2 Instrukce pro proměnné

Instrukce pro práci s proměnnými tvoří dvojice *LOAD* a *STORE*. *LOAD* provádí načtení dat z lokálních proměnných do zásobníku. *STORE* oproti tomu je určen pro ukládání ze zásobníku do proměnných. Pro případ využití těchto příkazů je vhodné uvést následující ukázku kódu 5.3.

```
1  public List getInfo() {
2  String url = "http://localhost:8090/api/user/users";
3  ...
4  HttpHeaders headers = new HttpHeaders();
5  headers.add("Content-Type", "application/json");
6  ...
7  ResponseEntity<List<User>> result =
8          restTemplateBuilder
9          .build()
10         .exchange(url,
11                 HttpMethod.GET, request, new
                       ParameterizedTypeReference<List<
                       User>>() {});
12 ...
13 }
```

Kód 5.3: Příklad využití proměnné (zdroj: vlastní tvorba)

Ukázka kódu 5.4 reprezentuje případ, kdy je konstanta uložená do lokální proměnné pro pozdější využití.

```
1  ldc "http://localhost:8090/api/user/users"
2  astore1
```

Kód 5.4: Uložení URL do proměnné (zdroj: vlastní tvorba)

Instrukce *LOAD* jak zobrazuje ukázka kódu 5.4 se využívá i pro načtení případných *klientů*, kteří jsou uloženi v lokální proměnné. Tato informace je důležitá, jelikož dodává analyzátoru případný kontext u zpracování volaných WS.

```
1  aload2  //HttpHeaders  head;
2  ldc "Content-Type" (java.lang.String)
3  ldc "application/json" (java.lang.String)
4  invokevirtual org/springframework/http/HttpHeaders
5      .add(Ljava/lang/String;Ljava/lang/String;)
```

Kód 5.5: Načtení klienta (zdroj: vlastní tvorba)

Argumenty metod jsou v podstatě lokální proměnné. To samé platí pro referenci na sebe v rámci dané třídy. Pro následující popis platí.

- *arg* - argument metody
- *var* - lokální proměnná

Indexy jsou pro dílčí proměnné rozděleny následujícím způsobem: *odkaz na sebe*(index 0), $arg_1 \ldots arg_n$ (n - počet argumentů metody) a $var_{n+1} \ldots var_{n+m}$ (m - počet proměnných metody). Reference na sebe sama se nachází pouze v metodách, které nejsou statické.

5.1.3 Instrukce pro pole

V rámci ukázky kódu 5.6 lze pozorovat, že případný vytvářený požadavek může mít argument přímo jako pole nebo jako tzv. *variabilní argument* (Varargs). Tento typ argumentu zobrazuje ukázka kódu 5.6. *Varargs* a *klasické pole* jsou v podstatě zaměnitelné. V případě *Varargs* se místo nedodaných argumentů vytvoří následující pole `new Object[0]` viz ukázka kódu 5.8.

```
1 patchForObject(
2     String url,
3     Object request,
4     Class<T> responseType,
5     Object... uriVariables
6 )
```

Kód 5.6: Metoda pro RestTemplate klienta (zdroj: vlastní tvorba)

```
1 ResponseEntity <List <User >> result =
    restTemplateBuilder.build().exchange(url,
    HttpMethod.GET, request,
    newParameterizedTypeReference <List <User >>()
    {});
```

Kód 5.7: Volání metody bez argumentu pro *Varargs* (zdroj: vlastní tvorba)

```
1 // reference to self
2 aload0
3 // this.restTemplateBuilder
4  getfield boot.web.client.RestTemplateBuilder
5 // .build ()
6 invokevirtual org/springframework/boot/web/client/
    RestTemplateBuilder
```

```
7     .build()Lorg/springframework/web/client/
        RestTemplate;
8
9 ...
10
11 // new Object[0]
12 iconst_0
13 anewarray java/lang/Object
```

Kód 5.8: Instrukce pro volání metody `exchange` (zdroj: vlastní tvorba)

Pro získání kompletní informace v rámci této práce se musí zpracovat i tyto konstrukce. Jelikož tento typ argumentu uchovává případné parametry dané URL adresy a další důležité informace.

5.1.4 Instrukce pro atributy třídy

Instrukce pro atributy třídy jsou *GETFIELD, GETSTATIC* a *PUTFIELD, PUTSTATIC*. Pro tuto práci jsou tyto instrukce důležité v případech, kdy jsou parametry požadavků uložené v některém z atributu třídy. Takovou konstrukci je možné pozorovat na ukázce kódu 5.9.

```
1 public Mono<String> test(){
2     RequestBodyUriSpec test = this.webClient.put();
3     return test
4             .uri(Uri.FIRST)
5             .retrieve()
6             .bodyToMono(String.class);
7 }
```

Kód 5.9: Atribut třídy držící URI požadavku (zdroj: vlastní tvorba)

Z předchozího zdrojového kódu je generován byte-kód, který prezentuje ukázka kódu 5.10. Na ní je možné vidět použití instrukcí *GETSTATIC* a *GETFIELD*. GETSTATIC zde provádí získání dané URI, to představuje objekt typu `String`. Instrukce *GETFIELD* v tomto případě načítá atribut aktuální třídy, ve které je tato instrukce spuštěna. *GETFIELD* potřebuje kontext do jakého objektu se bude přistupovat. To dodává instrukce `aload0`, kterou je možné pozorovat na ukázce kódu 5.10.

```
1 aload0 // reference to self
2 getfield com/baeldung/reactive/service/
    EmployeeService
```

```
3       .webClient:org.springframework
4       .web.reactive.function.client.WebClient
5   invokeinterface org/springframework/web
6       /reactive/function/client/WebClient
7           .put()Lorg/springframework/web/reactive/
8               function/client/WebClient$
9                   RequestBodyUriSpec;
10  astore1
```

Kód 5.10: Instrukce pro získání atributu třídy (zdroj: vlastní tvorba)

Obdobně fungují i instrukce *PUTFIELD* a *PUTSTATIC*, jenom s tím rozdílem, že provádí vkládání ze zásobníku operandů do atributů třídy.

5.1.5 Instrukce pro metody

Nejdůležitější sadou instrukcí pro tuto práci jsou instrukce určené pro metody. Ty lze rozdělit do dvou skupin *RETURN* a *INVOKE*.

RETURN je zaměřena na návratovou hodnotu dané metody a *INVOKE* je instrukce určená pro její volání. Pro popis těchto instrukcí je využita ukázka kódu 5.11.

```
1   public Mono<String> test(Integer employeeId) {
2       String test = "/dalsi/uri/s/argumentem/{id}";
3       if (employeeId == 1){
4           test = "test";
5       }
6       String test2 = "/bla/uri/s/argumentem/{id}";
7       return webClient
8               .delete()
9               .uri(test,employeeId)
10              .retrieve()
11              .bodyToMono(String.class);
12  }
```

Kód 5.11: Příklad vytvoření požadavku (zdroj: vlastní tvorba)

V rámci volání metody překladač dodá do byte-kódu její popis. Ten se nazývá *signature*. Obsahuje následující: *vlastníka (třídu, ve které je metoda obsažena), popis typů argumentů, typ návratové hodnoty a název samotné metody.*

Signaturu, která je obsažena v INVOKE instrukci je možné vidět na následující ukázce kódu 5.12.

```
1 //invokeintrface <Signatura metody>
2 invokeinterface org/springframework/web/reactive/
    function/
3     client/WebClient.delete()
4     Lorg/springframework/web/reactive/function/
        client/
5       WebClient$RequestHeadersUriSpec;
```

Kód 5.12: Příklad volání klientské metody (zdroj: vlastní tvorba)

Výše zmíněné informace, které lze získat z volání metody, mohou být využity při případné detekci volané WS. Zároveň instrukce *RETURN* poslouží pro informaci o návratové hodnotě dané metody, případně jejího typu.

5.1.6 Anonymní třídy a generické typy

Anonymní třídy jsou zanořeny do jiné třídy. Při překladu tříd, které je obalují, jsou vytvořeny nové *class* soubory. Ty mají název podle vlastníka (třídy, ve které se nachází). K tomuto názvu je nakonec přidáno znaménko dolaru např. `ApiService.class` → `ApiService$1.class`.

Generické typy jsou určeny pro vytváření typů, které mohou být upřesněny přidanými parametry. Právě ty jsou hojně zastoupeny mezi argumenty metod jednotlivých klientů. Generické typy mohou způsobovat problémy, jelikož typ, který je jim předáván prostřednictvím parametru nelze určit z byte-kódu.

Ke klasickým generickým typům existují alternativy v podobě generické abstraktní třídy. Ta je určena pro udržení datových typů. Neobsahuje žádnou *abstraktní metodu*. Používají se jako *inline anonymní třídy*. Z překladačem vygenerované třídy je možné později určit, jaký datový typ obsahují.

Tyto konstrukce je možné pozorovat na ukázce kódu 5.13.

```
1 ResponseEntity<List<User>> result =
    restTemplateBuilder.build().
2       exchange(url,
3               HttpMethod.GET, request, new
                  ParameterizedTypeReference<List<
                  User>>() {});
```

Kód 5.13: Předávání typu jako argument metody (zdroj: vlastní tvorba)

Na ukázce kódu 5.14 lze pozorovat, jak vytvoření výsledného objektu anonymní třídy probíhá.

```
...
new com/app/demo/service/ApiService$1
dup
aload0 // reference to self
invokespecial com/app/demo/service/ApiService$1.<
    init>(Lcom/app/demo/service/ApiService;)V
...
```

Kód 5.14: Část překladu anonymních generik (zdroj: vlastní tvorba)

5.2 Analýza tříd a rozhraní frameworků určených pro klienty WS

V této části probíhá analýza dostupných zdrojových kódů frameworků pro webové klienty zaměřených na REST komunikaci. Rozbor kódů je zaměřen na klientské části frameworku *Spring* a *JAX-RS client API*. Mezi frameworkem *Spring* a *JAX-RS client API* jsou hledány metody spolu s jejich argumenty, které jsou si vzájemně velmi podobné.

Popsání podobnosti těchto frameworků bude probíhat pomocí popisných hlaviček a případných argumentů. Hlavička je odvozena od *HTTP* metody vytvářeného požadavku. Pokud není specifikována, může být metoda pojmenovaná jako `GENERIC`. Další možností je pojmenování metody podle její činnosti např. `ACCEPT` (Accept pro hlavičku), `ACCEPT_LANGUAGE` apod.

Jako příklad lze uvést popis pro `put(String url, Map<String,?> uriVariables)`.

- put
    - HTTP metoda - PUT
    - popisy
        * `PUT(URL,URL_PARAMETERS)`

Způsoby, kterými lze označit jednotlivé argumenty metod jsou vidět v tabulce 5.1.

Argumenty metody	Vysvětlení
HTTP	GET | HEAD | POST | PUT | DELETE | CONNECT | OPTIONS | TRACE | PATCH
PATH	cesta k dané webové stránce nebo souboru
BASEURL	základní část celé URL (např. https://www.zcu.cz)
URL	celková URL složena z BASEURL a PATH
URL_PARAMS	parametry dané URL (ať už v podobě klasické query nebo použití matrix parametrů)
EXPECT	očekávaný objekt získaný z odpovědi serveru
SEND	odesílaný objekt v požadavku klienta
ENDPOINT_DATA	informace o daném endpointu, který bude volán (může obsahovat veškerá data zmíněna v této tabulce)
CALLBACK_REQ	callback metoda pro zpracování požadavku
EXTRACTOR	objekt starající se o zpracování odpovědi
HEADER_TYPE	typ hlavičky jako např. *Content-Type*
HEADER_VALUE	hodnota hlavičky jako např. *application/json*

Tabulka 5.1: Symboly argumentů klientů WS (zdroj: vlastní tvorba)

5.2.1 Spring

Spring je komplexní framework poskytující třídy a rozhraní pro SOAP a REST komunikaci. Pro klienty WS, kteří využívají výhradně RESTovou komunikaci existují dvě hlavní rozhraní `WebClient` a `RestTemplate`.

RestTemplate

V následujících tabulkách jsou uvedeny metody, které `RestTemplate` rozhraní využívají. Tabulka 5.2 je zaměřena na metody určené k volání specifických požadavků. Tyto metody zároveň neočekávají odpověď v podobě určitého objektu. Metody `put`, `headForHeaders`, `delete` a `optionsForAllow` mají společné argumenty. Pro přehled stačí popsat pouze metodu `put`.

- `put`
  - HTTP metoda - PUT
  - popisy
    * `PUT(URL, URL_PARAMS)`
    * `PUT(URL)`

Název	**Parametry**	**Typ**
put	(String url, Map<String,?> uriVariables)	PUT
	(String url, Object... uriVariables)	
	(URI url)	
headForHeaders	(String url, Map<String,?> uriVariables)	HEADER
	(String url, Object... uriVariables)	
	(URI url)	
delete	(String url, Map<String,?> uriVariables)	DELETE
	(String url, Object... uriVariables)	
	(URI url)	
optionsForAllow	(String url, Map<String,?> uriVariables)	OPTIONS
	(String url, Object... uriVariables)	
	(URI url)	

Tabulka 5.2: Metody s typem požadavku (zdroj: [10])

Metody v tabulce 5.3 se od první tabulky liší zejména tím, že očekávají objekt (uživatel, článek apod.) jako odpověď. Ten je specifikovaný třídou, která je přiložena do parametru metody. Za zmínku stojí, že typ dané odpovědi nebo požadavku je obalen generickou abstraktní třídou. Informace o případné HTTP metodě je uložena v názvu metody.

Název	**Parametry**	**Typ**
getForEntity getForObject	(String url, Class<T>responseType, Map<String,?>uriVariables)	GET
	(String url, Class<T>responseType, Object... uriVariables)	
	(URI url, Class<T>responseType	
postForEntity postForObject	(String url, Object request, Class<T>responseType, Map<String,?>uriVariables)	POST
	(String url, Object request, Class<T>responseType , Object... uriVariables)	
	(URI url, Object request, Class<T>responseType)	

Tabulka 5.3: Požadavky s generikami pro *entity* a *object* (zdroj: [10])

- `getForObject` a `getForEntity`

  - HTTP metoda - GET
  - popisy
    * `GET(URL)`
    * `GET(URL, URL_PARAMS)`

Pro HTTP metodu *POST* jsou navíc předávány informace o odesílaném objektu. Popis pro tuto metodu bude jen nepatrně odlišný od zmíněné HTTP metody *GET*. Metody `postForEntity`, `postForObject` a `postForLocation` jsou popsány níže.

- `postForEntity` a `postForObject`
  - HTTP metoda - POST
  - popisy
    * `POST(URL, SEND || ENDPOINT_DATA, EXPECT, URL_PARAMS)`
    * `POST(URL, SEND || ENDPOINT_DATA, EXPECT)`

Další je metoda `postForLocation` nacházející se v tabulce 5.4, ta je oproti předchozí dvojici ochuzena o typ očekávaných dat.

<table>
<tr><th>Název</th><th>Parametry</th><th>Typ</th></tr>
<tr><td rowspan="3">postForLocation</td><td>(String url, Object request, Map<String,?>uriVariables)</td><td rowspan="3">POST</td></tr>
<tr><td>(String url, Object request, Object... uriVariables)</td></tr>
<tr><td>(URI url, Object request)</td></tr>
<tr><td rowspan="3">patchForObject</td><td>(String url, Object request, Class<T>responseType, Map<String,?>uriVariables)</td><td rowspan="3">PATCH</td></tr>
<tr><td>(String url, Object request, Class<T>responseType, Object... uriVariables)</td></tr>
<tr><td>(URI url, Object request, Class<T>responseType)</td></tr>
</table>

Tabulka 5.4: Požadavky s gener. třídami pro *location* a *object* (zdroj: [10])

Popis metody `postForLocation` je možné vidět na následujícím seznamu.

- `postForLocation`
  - HTTP metoda - POST
  - popisy
    * `POST(URL, SEND || ENDPOINT_DATA, URL_PARAMS)`
    * `POST(URL, SEND || ENDPOINT_DATA)`

Dále lze zmínit metodu `patchForObject`, ta je v následující části vysvětlena.

- `patchForObject`
  - HTTP metoda - PATCH
  - popisy
    * `PATCH(URL, SEND || ENDPOINT_DATA, EXPECT, URL_PARAMS)`
    * `PATCH(URL, SEND || ENDPOINT_DATA, EXPECT)`

Poslední množinou metod, které jsou zobrazeny na tabulce 5.5, jsou generické metody. Jejich chování je striktně dáno přiloženými parametry. Pro ukázku je zmíněna pouze metoda `exchange` určená pro vytváření požadavků. Tato metoda používá složitější objekty jako například *callback metody* pro správu příchozích odpovědí. Jelikož je to relativně komplexní metoda, existuje více způsobů, kterými ji lze popsat.

- `exchange`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(URL, HTTP, CALLBACK_REQ, EXTRACTOR, URL_PARAMS)`
    * `GENERIC(URL, HTTP, ENDPOINT_DATA, EXPECT, URL_PARAMS)`
    * `GENERIC(SEND, EXPECT)`

Název	**Parametry**	**Typ**
exchange	(RequestEntity<?>entity, Class<T>responseType)	GENERIC
	(RequestEntity<?>entity, ParameterizedTypeReference<T> responseType)	
	(String url, HttpMethod method, HttpEntity<?>requestEntity, Class<T>responseType, Map<String,?>uriVariables)	
	(String url, HttpMethod method, HttpEntity<?>requestEntity, Class<T>responseType, Object... uriVariables)	
	(String url, HttpMethod method, HttpEntity<?>requestEntity, ParameterizedTypeReference<T>responseType, Map<String,?>uriVariables)	
	(String url, HttpMethod method, HttpEntity<?>requestEntity, ParameterizedTypeReference<T>responseType, Object... uriVariables)	
	(URI url, HttpMethod method, HttpEntity<?>requestEntity, Class<T>responseType)	
	(String url, HttpMethod method, RequestCallback requestCallback, ResponseExtractor<T>responseExtractor, Map<String,?>uriVariables)	
	(String url, HttpMethod method, RequestCallback requestCallback, ResponseExtractor<T>responseExtractor, Object... uriVariables)	
	(URI url, HttpMethod method, RequestCallback requestCallback, ResponseExtractor<T>responseExtractor)	

Tabulka 5.5: Generické volání (zdroj: [10])

Další entity, které zasahují do celkové komunikace jsou *obalovací třídy*. Ty udržují informace pro klientské metody, jako je *HTTP metoda*, *hlavičky*

apod. Několik klientů může využívat stejná data pro vytváření požadavků. Mezi tyto třídy patří například `HttpHeaders`. Ta je zaměřena primárně na nastavování hlaviček požadavků. Některé ze zmíněných metod jsou nastíněny v tabulce 5.6. Popis těchto metod bude teď v krátkosti uveden.

- `set` a `setAll`
  - HTTP metoda - není určena
  - popisy
    - `GENERIC(HEADER_KEY, HEADER_VALUE)`
- `setAccept`
  - HTTP metoda - není určena
  - popisy
    - `ACCEPT(HEADER_VALUE)`
- `setAcceptCharset`
  - HTTP metoda - není určena
  - popisy
    - `ACCEPT(HEADER_VALUE)`
- `setAcceptLanguage`
  - HTTP metoda - není určena
  - popisy
    - `ACCEPT_LANGUAGE(HEADER_VALUE)`

Název	**Parametry**	**Endpoint Data**
konstruktor	(MultiValueMap<String, String>headers)	GENERIC
set	(String headerName, String headerValue)	GENERIC
setAll	(Map<String,String>values)	GENERIC
setAccept	(List<MediaType> acceptableMediaTypes)	Accept
setAcceptCharset	(List<Charset> acceptableCharsets)	Accept-Charset
setAcceptLanguage	(List<Locale.LanguageRange> languages)	Accept-Language

Tabulka 5.6: Obalovací třída `HttpHeaders` (zdroj: [10])

WebClient

`WebClient` nahrazuje staré rozhraní `RestTemplate`. Princip fungování a syntaxe volání metod `WebClienta` je velmi podobný. Nebudou zde tak podrobně rozepsány jednotlivé metody, jako tomu bylo v případě `RestTemplate`. Pro analýzu může být důležité zmínit využití tzv. `WebClientBuilderu`. Ten se stará o postupné vytváření klienta, který potom bude volat konkrétní endpoint.

Z tabulky 5.7 je patrné, že se `Builder` chová velmi podobně jako třída `HttpHeaders` zmíněná v předchozí sekci.

Název	**Parametry**	**Endpoint Data**
baseUrl	(String baseUrl)	URL
defaultCookie	(String cookie, String... values)	COOKIE
defaultHeader	(String header, String... values)	HEADER

Tabulka 5.7: `Builder` (zdroj: [13])

Třída `WebClient.RequestBodySpec` je zaměřená primárně na nastavení těla dotazu. Názornou ukázkou je trojice metod vyobrazena v tabulce 5.8. Tato trojice je zaměřena na nastavování těla požadavku.

Název	**Parametry**	**Endpoint Data**
bodyValue	(Object body)	BODY
body	(P publisher,Class<T>elementClass)	
body	(Object producer,Class<?>elementClass)	

Tabulka 5.8: `RequestBodySpec` (zdroj: [13])

V krátkosti teď budou popsány metody vycházející z tabulky 5.7.

- `baseUrl`
  - HTTP metoda - není určena
  - popisy
    - `GENERIC(URL)`
- `defaultCookie`
  - HTTP metoda - není určena
  - popisy

    * `GENERIC(COOKIE_NAME, COOKIE_VALUE)`

- `defaultCookie`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(COOKIE_NAME, COOKIE_VALUE)`
- `defaultHeader`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(HEADER_TYPE, HEADER_VALUE)`

Jako další lze uvést metody `bodyValue` a `body` z tabulky 5.8.

- `bodyValue`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(SEND)`
- `body`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(SEND, SEND)`

5.2.2 JAX-RS client API

Jak již bylo zmíněno v kapitole *Webové služby*, JAX-RS client API se zaměřuje na klienty RESTového typu. Je to API, které může být implementováno různými frameworky například *Apache CXF*, *Jersey* nebo *RESTeasy*. Všechny implementace JAX-RS client API mají stejné názvy balíků (package) a tříd. Mezi nejdůležitější rozhraní zajišťující komunikaci nepochybně patří `Client` a `Invocation$Builder`. Rozhraní `Client` poskytuje jenom velmi omezené množství metod. Metody `invocation` a `target` poskytují základní nastavení URL, to lze vyčíst z tabulky 5.9.

Název	**Parametry**	**Endpoint Data**
invocation	(Link link)	URL
target	(Link link)	URL
	(String uri)	
	(URI uri)	
	(UriBuilder uriBuilder)	
defaultHeader	(String header, String... values)	HEADER

Tabulka 5.9: Metody třídy `Client` (zdroj: [19])

Metody `invocationm` `target` a `defaultHeader` lze popsat následujícím způsobem.

- `invocation` a `target`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(URL)`
- `defaultHeader`
  - HTTP metoda - není určena
  - popisy
    * `GENERIC(HEADER_TYPE, HEADER_VALUE)`

Další třídou, která se nachází uvnitř třídy `Invocation` je `Builder`. Ta agreguje velkou část REST komunikace. Obsahuje jak generické metody, jejichž chování je upřesněno pomocí dodaných parametrů, tak i metody sémanticky definované jejich názvem. Z tabulky 5.10 lze vyčíst seznam jednotlivých metod, které `Builder` poskytuje. Principiálně pracuje velmi podobně jako `RestTemplate` a `WebClient`. Jednotlivé metody vycházející z tabulky 5.10 je možné popsat následujícím způsobem.

- `head`
  - HTTP metoda - HEAD
  - popisy
    * `HEAD()`
- `get`
  - HTTP metoda - GET

  - popisy
    * `GET()`
    * `GET(EXPECT)`

- `delete`
  - HTTP metoda - DELETE
  - popisy
    * `DELETE()`
    * `DELETE(EXPECT)`

- `post`
  - HTTP metoda - POST
  - popisy
    * `POST(ENDPOINT_DATA)`
    * `POST(ENDPOINT_DATA, EXPECT)`

- `put`
  - HTTP metoda - PUT
  - popisy
    * `PUT(ENDPOINT_DATA)`
    * `PUT(ENDPOINT_DATA, EXPECT)`

- `trace`
  - HTTP metoda - TRACE
  - popisy
    * `TRACE()`
    * `TRACE(EXPECTT)`

Název	Parametry	Typ
get	()	GET
	(Class<T>responseType)	
	(GenericType<T>responseType)	
delete	()	DELETE
	Class<T>responseType)	
	(GenericType<T>responseType)	
head	()	HEAD
post	(Entity<?>entity)	POST
	(Entity<?>entity, Class<T>responseType)	
	(Entity<?>entity, GenericType<T>responseType)	
put	(Entity<?>entity)	PUT
	(Entity<?>entity, Class<T>responseType)	
	(Entity<?>entity, GenericType<T>responseType)	
trace	()	TRACE
	(Class<T>responseType)	
	(GenericType<T>responseType)	

Tabulka 5.10: Metody třídy `Builder` (zdroj: [19])

5.2.3 Shrnutí

Způsoby volání webových služeb mohou být rozděleny do dvou následujících kategorií: *generické* a *upřesněné názvem metody*. *Generické* metody mají svou činnost definovanou vstupními argumenty metody. Ty mohou určit HTTP metodu, hlavičku, tělo dotazu apod.

Z analýzy je patrné, že spousta popisů definující jednotlivé klientské metody se v některých třídách a dokonce frameworcích opakují.

Tato informace je využita v rámci návrhu algoritmu pro řešení získání volaných API WS.

5.3 Algoritmus

Algoritmus pro analýzu API volaných WS se zaměřuje na *byte-kód* jazyka Java. Vychází z předchozích znalostí o tom, jak byte-kód pracuje a jaké informace z něj lze získat. Skládá se ze *tří* hlavních částí, a to *konfigurace*, *zpracování byte-kódu* a jeho *interpretace*. Logiku a návaznost jednotlivých procesů, na kterých je algoritmus postaven, je možné vidět na obrázku 5.1.

5.3.1 Konfigurace

Pro získání informací o požadavcích klientů WS musí existovat způsob, jakým metody provádějící dotazování budou detekovány. Na základě analýzy z předchozí sekce je možné vytvořit konfigurační soubor. Ten je postaven na sémantických informacích vyobrazených v tabulce 5.1. Každý framework potřebuje vlastní konfiguraci, pokud nesdílí stejné rozhraní. Součástí konfiguračního souboru pro klienty WS musí být následující.

- název *třídy/rozhraní*
- název *metody* spolu s jejím sémantickým významem (typ HTTP metody)
- druhy *parametrů* např.:
  - hlavička
  - tělo požadavku
  - očekávaná odpověď
  - ...

5.3.2 Zpracování byte-kódu

Před samotnou interpretací se byte-kód musí nejprve zpracovat. Zpracováním byte-kódu do struktur se získá kontext volání metod (vlastníka metod - třída) a lepší popis metod, proměnných a tříd. Proces je postaven na průchodu kódem pomocí tzv. *visitorů*. Z parametrů metod se získají informace, které jsou následně uloženy do připravených struktur.

Nejprve se zpracuje třída, ta obaluje veškeré další prvky jako je metoda a její operace spolu s atributy třídy.

- *Zpracování třídy*
  - atributy třídy

  - inicializace

Po zpracování základních částí třídy se algoritmus zaměří na zpracování samotných metod.

- *Zpracování metod*
  - název a popis
  - návratový typ
  - parametry metody
  - operace
  - vlastník (třída, ve které se metoda nachází)

Jednotlivé operace jsou uzavřeny v rámci každé metody a obsahují následující.

- *Zpracování operace*
  - opcode - unikátní kód operace
  - typ operace
  - vlastníka
  - jméno volané metody
  - popis
  - ...

5.3.3 Interpretace

Z diagramu prezentovaném na obrázku 5.1 je možné vyčíst další část algoritmu interpretace byte-kódu. Ty hrají klíčovou roli v získávání informací o volané WS. Interpretace využívá *zásobník*, *simulaci proměnných* a *atributů* tříd. Skládá se ze dvou částí *základní operace* a *volání metod*.

Základní operace

Tato část interpretace řeší zpracování operací jako je *získání atributů třídy*, *zpracování ldc (načtení konstanty)*, *načtení spolu s uložením do proměnné*, *vytvoření pole* atd.

Pro manipulaci s atributy tříd je nutné vytvořit *map* objekt. Ten uchovává veškeré atributy pod konkrétním klíčem (název atributu), který je uložen v struktuře dané operace. Případné operace *(GET/PUT)_FIELD* mapu využívají pro ukládání nebo načtení hodnoty z dané třídy.

Proměnné jsou ukládány do pole, které vlastní každá metoda. Pole musí být dynamické, jelikož není dopředu známa jeho velikost (počet proměnných). Prvních n pozic zabírají argumenty metod. Proměnné určené pro kontext dané metody nebo vnitřních cyklů inkrementují ten samý index a jsou proto ukládány společně do jednoho pole. Logika manipulace s proměnnými je pak řízena samotnými instrukcemi *(X)STORE* a *(X)LOAD*. Ty si drží index, který určuje pozici proměnné. Potom se proměnná pod určitým indexem načte nebo uloží.

Volání metod

Volání metod zajišťují instrukce *INVOKE* s příponami *STATIC*, *VIRTUAL*, *DYNAMIC*, *STATIC* a *INTERFACE*. Interpretace volání metod je značně omezená. Pro získání přesné informace by bylo potřeba simulovat reálný běh aplikace. Proto se algoritmus zaměřuje na *návratové typy* jednotlivých metod. Algoritmus je schopný interpretovat jednoduché skládání řetězců. Ty jsou pak důležité zejména pro získání URL volaného endpointu.

5.3.4 Rekonstrukce API volaných WS

Samotná rekonstrukce API je řešena v poslední části algoritmu. Za využití konfigurace se detekuje volání klientů WS. Při každé instrukci *INVOKE_(X)* se hledá shoda s popisem metody a dodanou konfigurací. Pokud se metoda nachází v konfiguraci, algoritmus podle ní získá data ze zásobníku a následně je zpracuje. Takto získané informace jsou dále přiděleny záznamům o volaných WS nebo se případně vytvoří záznam úplně nový. Zmíněné operace zobrazuje diagram na obrázku 5.1.

Aby byla dotazovaná služba nebo endpoint jednoznačně identifikováni, musí existovat *URL adresa*, která je umožní odlišit. URL může využívat buď klasické *query* nebo *matrix* parametry.

- *URL*
  - základ adresy
  - určitá cesta
  - parametry (matrix nebo query)

Další důležitou součástí dotazů jsou jejich hlavičky. Ty lze podle oficiálních standardů rozdělit do *šesti* kategorií [40].

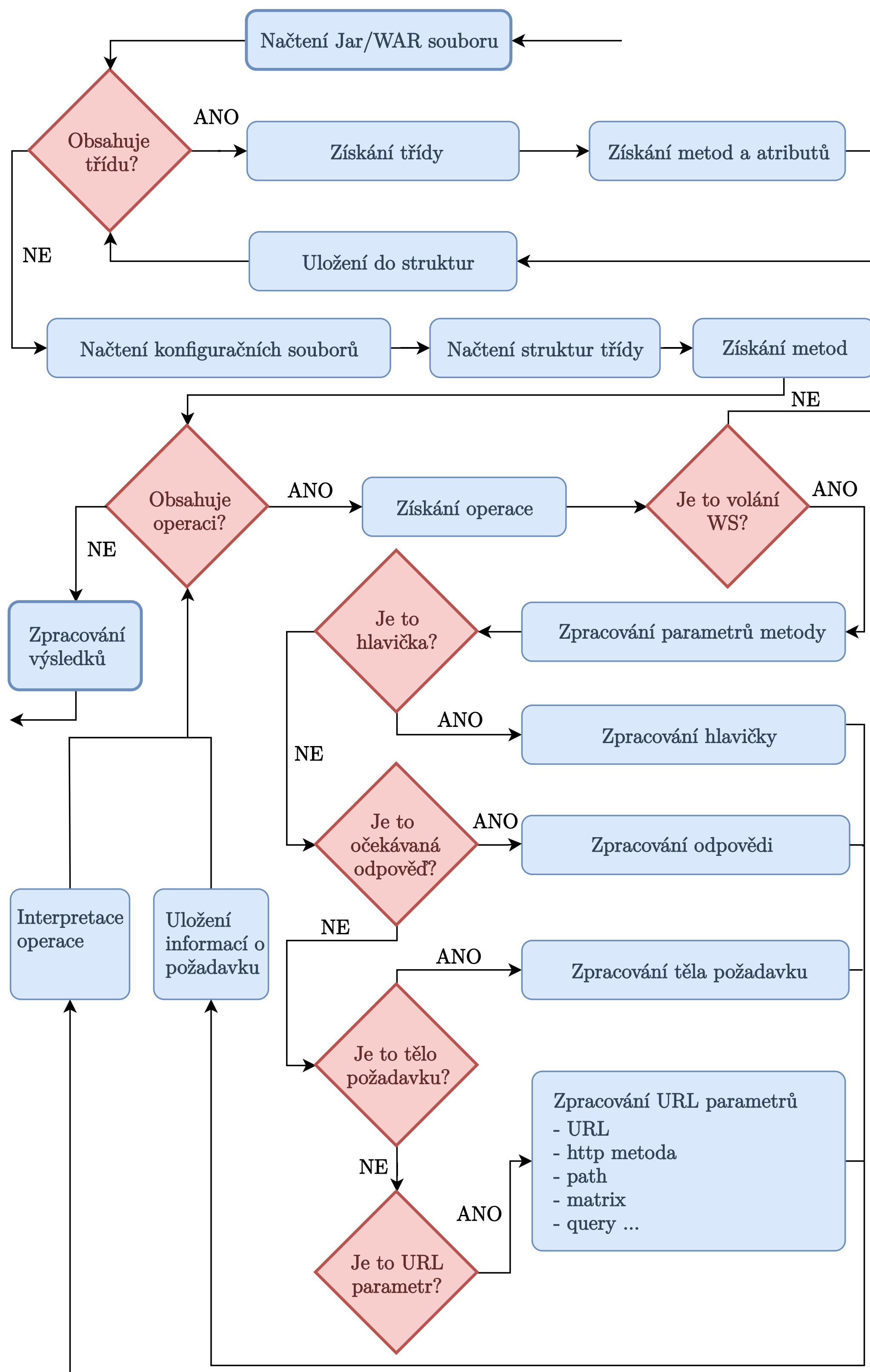
- *Hlavičky*

  - controls: `Cache-Control`, `Expect`, `Host` ...
  - conditionals: `If-Match`, `If-None-Match`, `If-Range` ...
  - content negotiation: `Accept`, `Accept-Charset`, `Accept-Encoding` ...
  - authentication credentials: `Authorization` a `Proxy-Authorization`
  - request context: `From`, `Referer` a `User-Agent`
  - representation: `Content-Type` a `Content-Length`
  - responses - ostatní hlavičky, které nejsou zařazeny pod výše zmíněné skupiny

- *HTTP* typ požadavku (GET, POST ...)

Tělo požadavku ve formě dodaného *beanu* je zpracováno a transformováno do *JSON* struktury, které je přenositelné a může být využito pro porovnání kompatibility. To samé platí pro očekávanou odpověď služby.

Tyto informace jsou ukládány do záznamů jednotlivých endpointů WS.



Obrázek 5.1: Algoritmus (zdroj: vlastní tvorba)

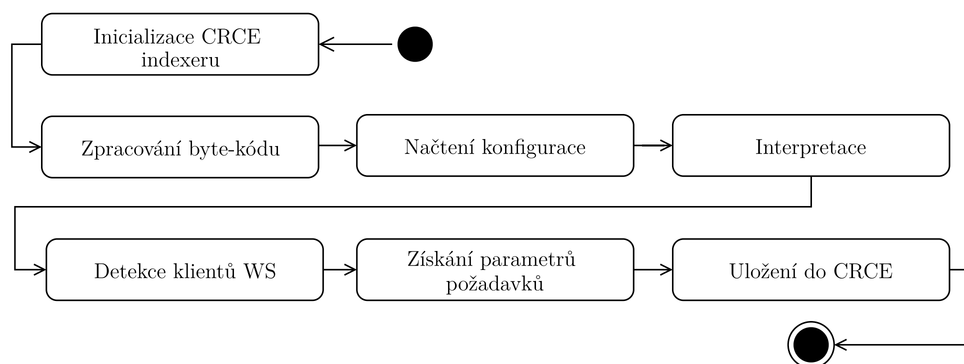
6 Implementace analyzátoru

Cílem této práce je vytvoření nástroje pro *Rekonstrukci API volaných webových služeb*. Rekonstrukce se zaměřuje na jazyk Java a jeho byte-kód. Analýza byte-kódu se snaží získat informace o volaných WS z pohledu klientské části aplikace. Celá analýza probíhá v rámci CRCE úložiště, které získané informace agreguje za účelem budoucí kontroly kompatibility jednotlivých SW komponent.

Tato kapitola popisuje implementaci zmíněného nástroje. Zároveň je zde vysvětlen postup, který musel být vynaložen pro integraci vytvořeného indexeru do úložiště CRCE. V poslední řadě je zde předvedená krátká ukázka toho, jak tento nástroj pracuje.

6.1 Struktura programu

Struktura programu se odvíjí od algoritmu zmíněného v předchozí kapitole a je závislá na celkovém procesu analýzy byte-kódu, který je možné pozorovat na obrázku 6.1.

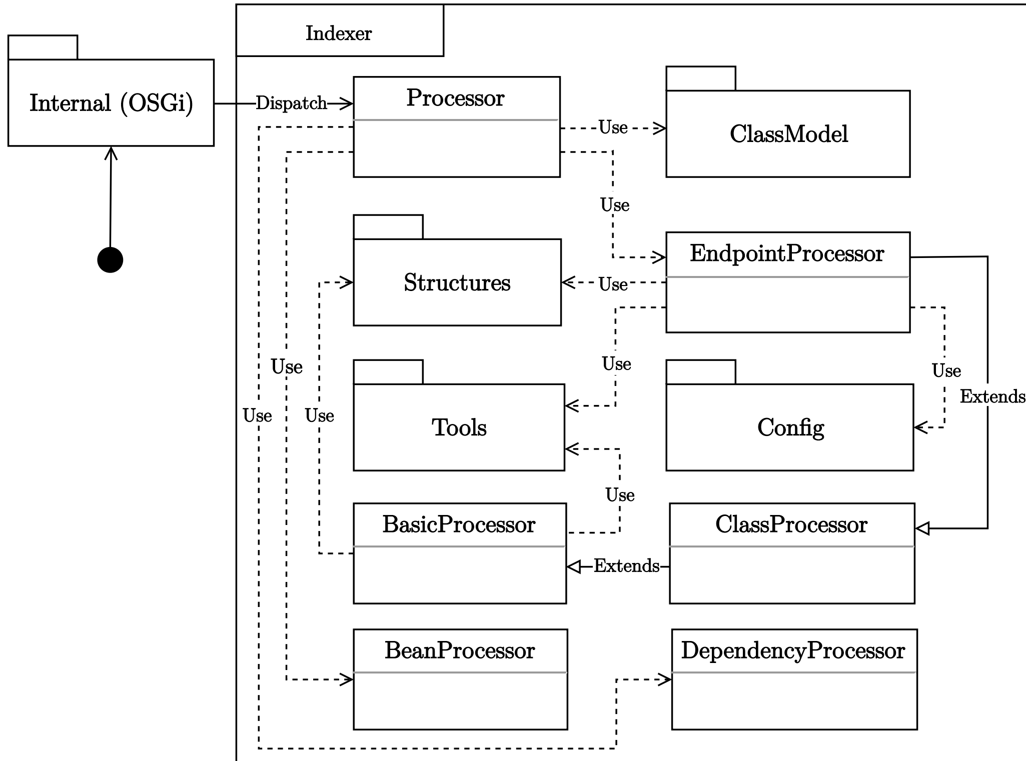


Obrázek 6.1: Popis fungování výsledné aplikace (zdroj: vlastní tvorba)

`Internal` je balík, jehož hlavním úkolem je propojení *indexeru* s úložištěm CRCE. Řeší to, v jakém formátu budou data budou uložena do CRCE a jak budou pojmenována.

Primární komponentou, která spouští celý proces je třída `Processor`. Prostřednictvím `FieldProcessoru` a `MethodProcessoru` se zpracují jednotlivé třídy a jejich metody. Balík `Tools` obsahuje nástroje pro *detekci hlaviček*, *zpracování popisu metod*, *manipulaci s endpointy* atd. Tento balík je využíván napříč jednotlivými procesory.

Třída `EndpointDataFactory` řeší samotné získávání dat z *klientů WS*, která jsou uložena v pomocném zásobníku. Využívá se zde *konfigurace* jednotlivých frameworků, pomocí kterých je možné přidat kontext dodaným parametrům. Konfigurace je připravená pro frameowork *Spring* a rozhraní *JAX-RS client API*.



Obrázek 6.2: crce-rest-client-indexer (zdroj: vlastní tvorba)

Z obrázku 6.2 lze vyčíst nejdůležitější třídy a balíky, na kterých závisí chod analyzátoru.

6.1.1 ClassModel

Tento balík se stará o zpracování *byte-kódu* pomocí knihovny *ASM*. Ta poskytuje abstraktní třídy, jako je například `ClassVisitor` a `MethodVisitor`. Ty jsou využity pro vlastní implementaci.

Pomocí ASM jsou obaleny *třídy*, *metody* a *operace* vyskytující se v byte-kódu. Operace mají připojeny důležité informace, jako např. *místo odkud se volají*, typ (*volání metody*, *načtení konstanty z const. poolu* apod.).

Balík se skládá ze dvou částí. První část je zaměřena na samotné visitory a druhá na struktury, do kterých se data ukládají.

6.1.2 Config

O načtení a zpracování konfiguračních souborů se stará balík `config`. Jednotlivé konfigurační soubory jsou načteny podle názvu třídy a metody do zanořené *mapy*. Tato data jsou využita při určení toho, jestli daná operace pracuje s klientem WS.

6.1.3 ClassProcessor

`ClassProcessor` dědí funkcionalitu od třídy `BasicProcessor` a provádí samotnou *interpretaci*, která je klíčovou funkcionalitou výsledného indexeru. Oproti `BasicProcessoru` navíc přínáší možnosti zpracování volání metod *INVOKESTATIC*, *INVOKEVIRTUAL* apod. Zároveň provádí inicializaci tříd (`<init>` a `<cinit>`) zpracovávaného byte-kódu. Při zpracování metod obstarává i kontrolu rekurze, aby nedošlo k případnému přetečení zásobníku.

6.1.4 BasicProcessor

Základní instrukce pro práci s proměnnými, konstantami a atributy třídy zpracovává třída `BasicProcessor`.

6.1.5 EndpointProcessor

`EndpointProcessor` zpracovává data jednotlivých klientů. K tomu se využívá *zásobník*. Do něj se ukládají uměle vytvořené proměnné se specifickým typem, ten je závislý na operaci, ze které vznikly. Mohou tak v sobě uchovávat data o volaném *endpointu*, *řetězci*, *návratovém typu* volané metody apod.

6.1.6 Tools

Tento balík poskytuje množinu nástrojů, kterou dále využívají ostatní procesory. Jsou zde nástroje pro konverzi do JSON struktury, zpracování popisů metod atd.

6.1.7 Bean a Dependency Procesor

`BeanProcessor` převádí tzv. *beany* do JSON objektů. `DependencyProcessor` se oproti tomu zabývá hledáním metod, ze kterých je klient volán, ať už přímo nebo nepřímo. To je pak využito k tomu, aby se zjistila závislost mezi

jednotlivými endpointy (na straně služby). Tato třída je využita v rámci modulu pro indexaci dostupných WS a rozšiřuje tak stávající informace v rámci *capability* dat o závislosti endpointů na jiné službě (*provided-required*).

6.2 Konfigurační soubor

Kompatibilita s různými frameworky je zajištěna pomocí konfiguračního souboru. Způsob jakým se jednotliví klienti WS zapisují je dostatečně obecný, aby se pokrylo co největší množství kombinací argumentů, různých implementací rozhraní a dědění tříd. Konfigurační soubor je odvozen od potřeb algoritmu, který se nachází v předchozí kapitole. Konfigurační soubor se skládá ze *čtyř* částí, a to `requestParameters`, `argDefinitions`, `wsClientData` a `wsClient`.

6.2.1 RequestParameters

`RequestParameters` je určen pro mapování, ať už *výčtových typů* nebo *atributů* tříd na hodnoty, které jsou vhodné pro analyzátor. Skládá se z `classes` - názvy tříd a `fields` - názvů atributů tříd. Pro mapování hodnot se využívá kombinace <fieldName> a <fieldMappingValue>, která je nastíněna v ukázce kódu 6.1.

```
1 requestParameters:
2   - classes:
3       - <className>
4       - ...
5     fields:
6       <fieldName>: <fieldMappingValue>
7       ...
```

Kód 6.1: Konfigurace - RequestParameters (zdroj: vlastní tvorba)

6.2.2 ArgDefinitions

Argumenty metod jsou definovány v rámci jednoho bloku `ArgDefinitions`. Každý argument má jedinečné jméno - <argName>, typ - `type` a název tříd nebo rozhraní - `classes`, `interfaces`. Argumenty je možné využít pro každou konfiguraci argumentů metody.

```
1 argDefinitions:
2     <argName>:
```

```
        type: <argType>
        #např.:URL,HEADER,REQUEST_BODY atd.
        classes:
            - <className>
            - ...
        interfaces:
            - <interfaceName>
            - ...
    <argNameX>
        ...
```

Kód 6.2: Konfigurace - ArgDefinitions (zdroj: vlastní tvorba)

6.2.3 WsClientData

Konfigurace `wsClientData` je určená pro definici objektů, které drží informace o případném endpointu, ale nevytváří žádný konkrétní požadavek. Skládá se z názvu tříd nebo rozhraní - `classes`, `interfaces`. Pod klíčem `settings` se nachází jednotlivé konfigurace metod. Jejich typ je určen hodnotou `<methodType>`. Konfigurace argumentů je předávána ve formě reference (názvu) `<argDefinitionReference>`.

```
wsClientData:
    - classes:
        - <className>
        - ...
      interfaces:
        - <interfaceName>
        - ...
          settings:
                <methodType>:
                #HEADER, GENERIC, atd.
                - names:
                    - <methodName>
                    - ...
                  args:
                    - - <argDefinitionReference>
                    #odkaz na referenci - název z
                       argDefintions
        type: <argType>
        #např.:URL,HEADER,REQUEST_BODY atd.
```

```
19          classes:
20              - <className>
21              - ...
22          interfaces:
23              - <interfaceName>
24              - ...
25      <argNameX>
26          ...
```

Kód 6.3: Konfigurace - wsClientData (zdroj: vlastní tvorba)

6.2.4 WsClient

Konfigurace `wsClient` je velmi podobná `wsClientData`. Je zde navíc možnost nastavení skupiny metod, a to na `settings` nebo `request` viz ukázka kódu 6.4. Skupina `request` se využívá pro metody, jejichž zavolání nastaví *HTTP* metodu případného požadavku. `Settings` je pak určeno pro případy, kdy volání metody má význam nastavení *hlavičky*, cookies apod.

```
1 wsClient:
2 ...
3       <groupType>: #settings nebo request
4             <methodType>:
5             #HEADER, GENERIC, atd.
6 ...
```

Kód 6.4: Konfigurace - wsClient (zdroj: vlastní tvorba)

6.3 Integrace

Vytvářený modul je součástí *CRCE úložiště*. S tím jsou spojeny určité integrační procesy. Jelikož je CRCE založeno na technologii *OSGi*, musí být připraveny příslušné soubory jako *Activator* a *osgi.bnd*. Tento soubor obsahuje metadata pro OSGi bundle viz ukázka kódu 6.5.

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

Kód 6.5: Ukázka osgi.bnd souboru (zdroj: vlastní tvorba)

Pro přidání nového indexeru do úložiště CRCE musí být ještě nastaven samotný indexer. Aby tento modul CRCE zaregistrovalo, musí třída nacházející se v balíku *Internal* rozšířit abstraktní třídu `AbstractResourceIndexer`.

7 Ověření funkčnosti

Práce se zaměřuje na analýzu Java byte-kódu. Java je multiparadigmatický[1] jazyk, to znamená, že pro analýzu kódu je potřeba pokrýt široké spektrum konstrukcí kódu. Nástroj má určeny jisté hranice, jelikož se nejedná o plnohodnotnou interpretaci kódu. Tato kapitola má tak za cíl tyto hranice otestovat a určit, jaké části kódu je možné analyzovat.

7.1 Testování funkcionality

V průběhu samotného vývoje, byl výstup nástroje průběžně testován prostřednictvími *jednotkových testů*. Testovala se schopnost tohoto nástroje získat informace o volaných WS spolu s testováním dílčích části kódu. Ty provádí složitější operace, jako je např. aplikace regulárních výrazů na názvy a popisy metod. V této části budou ukázány jednotlivé případy, na kterých bylo provedeno testování.

7.1.1 Vlastní implementace

V této části je otestováno chování indexeru vůči implementaci REST klientů v rámci vlastní implementace. Otestovány jsou následující konstrukce.

- přepisování proměnné
- chování pro podmíněné přiřazení hodnoty
- využití *výčtových typů*
- *abstraktní generické třídy* jako argument

Přepisování proměnné

Data získaná indexerem je možné pozorovat na ukázce kódu 7.2 a následujícím seznamu. Z těchto dat lze určit, že nástroj v tomto případě funguje správně.

- získaná data
  - URL - /123

[1]Multiparadigmatický jazyk - jazyk, který poskytuje, jak funkcionální tak i objektový přístup vývoje programů

– HTTP metody - GET
– URL parametry - `java/lang/Integer`

```
public Mono<Employee> getEmployeeById(Integer
    employeeId) {
    String test = "TEST";
    test = "/123";
    return webClient
            .get()
            .uri(test, employeeId)
            .retrieve()
            .bodyToMono(Employee.class);
}
```

Kód 7.1: Test - přepisování hodnoty proměnné (zdroj: vlastní tvorba)

Převedený bean `Employee` na JSON reprezentaci je obsažen v ukázce kódu 7.2.

```
{
{
    "structure" : {
      "type" :
      "com/baeldung/reactive/model/Employee",
      "data" : {
        "lastName" : "java/lang/String",
        "firstName" : "java/lang/String",
        "role" :
        [ "ENGINEER", "LEAD_ENGINEER", "
           SENIOR_ENGINEER" ],
        "arrayOfUO" : [ "java/util/Map" ],
        "jobs" : [ "java/lang/String" ],
        "employeeId" : "java/lang/Integer",
        "arrayOfArraysOfJobs" :
        [ [ "java/lang/String" ] ],
        "friends" :
        [ "com/baeldung/reactive/model/Employee" ],
        "age" : "java/lang/Integer"
      }
    },
    "isArray" : false
```

```
22 }
```

Kód 7.2: Získaný typ odpovědi (zdroj: vlastní tvorba)

Podmíněné nastavení proměnné

Ukázka kódu 7.3 zobrazuje případ, kdy je proměnná v podmíněné větvi přepsána jinou hodnotou.

- získaná data
  - URL - test
  - HTTP metody - GET
  - URL parametry - `java/lang/Integer`

```
1  public Mono<String> test(Integer employeeId) {
2      String test = "/dalsi/uri/s/argumentem/{id}";
3      if (employeeId == 1){
4          test = "test";
5      }
6      String test2 = "/bla/uri/s/argumentem/{id}";
7      return webClient
8              .delete()
9              .uri(test,employeeId)
10             .retrieve()
11             .bodyToMono(String.class);
12 }
```

Kód 7.3: Test - podmíněné přiřazení (zdroj: vlastní tvorba)

```
1 {
2     "structure" : "java/lang/String",
3     "isArray" : false
4 }
```

Kód 7.4: Test - získaný typ odpovědi (zdroj: vlastní tvorba)

Využití výčtových typů a generických tříd

Využití výčtového typu pro určení *HTTP* metody prezentuje ukázka kódu 7.5. Zde je testováno to, jestli nakonfigurovaný výčtový typ nebo třída předává očekávanou hodnotu.

- získaná data
  - URL - http://localhost:8090/api/user/users
  - HTTP metody - GET
  - očekávané odpovědi - pole objektů `com/app/demo/model/User`

```
1 public List<Employee> empList() {
2     WebTarget target = client.target(BASE_URI);
3     List<Employee> emplist =
4     target.path("/emp").request().
5     header(HttpHeaders.ACCEPT, MediaType.
         APPLICATION_JSON).
6     header(HttpHeaders.CONTENT_TYPE,MediaType.
         APPLICATION_JSON).
7     get(new GenericType<List<Employee>>() {});
8         return emplist;
9 }
```

Kód 7.5: Test - výčtové typy (zdroj: vlastní tvorba)

Indexer správně zpracoval, jak výčtový typ, tak i bean obalený generickou třídou. Část výstupu je možné vidět na obrázku 7.6.

```
1 {
2   "httpMethods" : [ "GET" ],
3     "responses" : [ {
4     "type" : "com/app/demo/model/User",
5     "structure" : {
6       "type" : "com/app/demo/model/User",
7       "data" : {
8         "name" : "java/lang/String",
9         "email" : "java/lang/String"
10      }
11    },
12 }
```

Kód 7.6: Test - získaná HTTP metoda a odpověď (zdroj: vlastní tvorba)

7.1.2 Testování na reálných implementacích

Součástí této práce je i testování na reálných implementacích. Testování probíhalo nad dvěma průmyslově využívanými implementacemi.

První implementace využívá pro klientské části *JAX-RS API*. Kód je podle testovatele velmi komplexní se spoustou abstrakcí a zanoření. Samotné volání *JAX-RS* je velmi zanořené, a proto indexer nebyl schopný získat informace o volaných službách. Zároveň se zde ukázala nestabilita indexeru, který tento kód nedokázal zpracovat. Nalezené chyby v kódu byly po této zkušenosti opraveny.

Druhá implementace vytvořena v rámci pracoviště ZČU - CIV využívá framework *Spring*, a to konkrétně `WebClient`. Každá metoda obalující klienta využívala pro vytváření *URL* vlastní metodu. Pomocí předávaných parametrů se prováděli vnitřní operace nad objektem, který si udržuje informace o URL. Zde se podařilo podchytit část *cesty* každé URL spolu se správcováním očekávaných požadavků (konverze bean na JSON). Ukázku výsledku indexeru na jednom z několika požadavků je možné pozorovat na ukázce 7.7.

```
1   "baseUrl" : null,
2   "path" : "getPredmetyByUcitel",
3   "httpMethods" : [ "GET" ],
4   "requestBodies" : [ ],
5   "responses" : [ {
6     "type" : "cz/zcu/civ/web/rest/domain/stag/
         courses/STAGPredmetyUcitele",
7     "structure" : {
8       "type" : "cz/zcu/civ/web/rest/domain/stag/
           courses/STAGPredmetyUcitele",
9       "data" : {
10        "predmetUcitele" : [ "cz/zcu/civ/web/rest/
             domain/stag/courses/STAGPredmetyUcitele"
             ]
11      }
12    },
13    "isArray" : false
14  } ],
15  "parameters" : [ {
16    "name" : "outputFormat",
17    "category" : "QUERY",
18    "dataType" : "java/lang/String",
19    "isArray" : false
20  } ],
21
```

```
22 }
```

Kód 7.7: Test - Výstup z indexece průmyslové aplikace (zdroj: vlastní tvorba)

V rámci této implementace se indexer potýká s *URL*, které jsou dynamicky vytvářeny, to je možné pozorovat na ukázce kódu 7.8. Pomocí *znovuprovedení instrukcí* pro každé volání metody, je indexer schopen relativně přesně získat vytvářené URL požadavků spolu s dalšími informacemi.

```
1  private URI getUrl(String category, String method,
     Map other) {
2    UriComponentsBuilder builder =
3    UriComponentsBuilder.
4      fromUri(this.uri).
5          pathSegment(new String[]{category, method}).
6              queryParam("outputFormat", new Object[]{
                 "json"});
7    Objects.requireNonNull(builder);
8    other.forEach((x$0, xva$1) -> {
9       builder.queryParam(x$0, new Object[]{xva$1});
10   });
11   return builder.build().encode().toUri();
12 }
```

Kód 7.8: Test - dynamické vytváření URL (zdroj: vlastní tvorba)

Omezení indexeru je v tom, že indexer není schopen simulovat smyčky, jako je `foreach` nebo `while` obsažený v ukázce kódu 7.8.

7.2 Shrnutí

Výsledný indexer splňuje očekávání. Je omezený tím, že neprovádí reálnou interpretaci. Nedokáže zpracovat složitější konstrukce, jako jsou smyčky a správně vyhodnotit podmíněné bloky. Data, která jsou generována dynamicky, je program schopný do jisté míry zpracovat tím, že se umožňuje opakované volání interpretovaných metod.

Pokud se vytvoří podrobný konfigurační soubor, může indexer zjistit poměrně přesně informace o volaných WS.

8 Závěr

Cílem této práce bylo vytvořit nástroj pro *rekonstrukci API volaných webových služeb*. Kombinací dat z tohoto nástroje a indexeru WS (z pohledu serverové části), který je součásti úložiště CRCE se potenciál této práce ještě více umocnil. Kontrola využívající zmíněná data výrazně pomůže případnému vývoji malých i velkých webových služeb, které si potřebují zajistit vysokou míru stability. Zajištění stability může probíhat například prostřednictvím *CI*, kde se při každé změně kódu spustí oba indexery. Takto získaná data lze porovnat a detekovat případnou nekompatibilitu serveru a klienta. Případná kontrola umožní podchytit doposud skryté problémy. Ty se mohou neočekávaně objevit až na produkční verzi aplikace.

Indexer WS razí směr zaměřený na služby typu REST, které jsou implementovány pomocí Java frameworků. Analýza provedená v kapitole *Webové služby* potvrdila, že tento směr je správný, a může tak pokrýt velké množství WS.

Výsledný nástroj není zaměřený pouze na určitou skupinu frameworků. Přistupuje k řešení obecně a umožňuje chování programu měnit prostřednictvím konfigurace. Může být nastaven například na vlastní implementace klientů nebo na doposud nenakonfigurované REST frameworky.

Výstup, který nástroj dodává, je velmi podobný indexeru dostupných WS. Rozdíly lze pozorovat například ve formátu *konzumovaných* nebo *dodávaných* dat. Ty jsou nově obohacena o podrobnější informace. Předchozí data nesoucí pouze název předávaného nebo konzumovaného *beanu* jsou nově serializována do formátu JSON. Takto získaná data jsou přenositelná napříč různými projekty.

Nástroj je zaměřen na indexaci *byte-kódu*. To přínáší řadu výhod. Indexer může zpracovávat například i samotné knihovny. Zároveň dokáže zpracovávat přeložené Java soubory, které tak nemusí být složitým (navíc nepřesným) procesem dekompilovány.

Nový rozměr této práci dodává samotný fakt, že je nástroj zakomponován do *CRCE úložiště*. To je právě zaměřeno na výše zmíněnou kontrolu kompatibility. Data je tak možné porovnávat i spolu s ostatními indexery, které jsou součástí CRCE úložiště.

V rámci rozšíření bylo implementováno i získávání informací o vzájemných závislostech WS (*provided-required*). Vzájemná závislost WS může hrát klíčovou roli v celkové stabilitě programu. Díky tomuto přehledu je možné zjistit, jak moc zasáhne program výpadek jedné služby.

Nástroj si přesto udržuje vysokou míru nezávislosti. Přináší možnost využití prostřednictvím příkazové řádky a de facto pracovat jako samostatná aplikace.

Schopnost pracovat i s reálnými implementacemi byla otestována v rámci dvou průmyslových aplikací. Kromě toho probíhalo i průběžné testování na vlastních testovacích datech.

Diplomová práce splnila veškeré cíle, které jsou uvedeny v jejím zadání.

A Literatura

[1] *BCEL manual* [online]. The Apache Software Foundation, 2005. [cit. 2021/05/14]. Introduction into BECL library. Dostupné z: http://epic-beta.kavli.tudelft.nl/share/doc/bcel-5.2/manual.html.

[2] *Benchmarks* [online]. Fastify, 2021. [cit. 2021/05/08]. Dostupné z: https://www.fastify.io/benchmarks/.

[3] *30.1 Overview of the Client API* [online]. Oracle, 2014. [cit. 2021/05/06]. JAX-RS client API. Dostupné z: https://docs.oracle.com/javaee/7/tutorial/jaxrs-client001.htm.

[4] *LoopBack 4* [online]. LoopBack, 2021. [cit. 2021/05/07]. Dostupné z: https://loopback.io/doc/en/lb4/.

[5] *Introduction to Node.js* [online]. Node.js, 2021. [cit. 2021/05/07]. Dostupné z: https://nodejs.dev/learn/introduction-to-nodejs.

[6] *Package Metadata* [online]. Npm, 2020. [cit. 2021/05/14]. Dostupné z: https://github.com/npm/registry/blob/master/docs/responses/package-metadata.md.

[7] *OSGi bundles* [online]. IBM, 2020. [cit. 2021/05/15]. Dostupné z: https://www.ibm.com/docs/en/wasdtfe?topic=overview-osgi-bundles.

[8] *OSGi Core Release 8* [online]. OSGI Alliance, 2021. [cit. 2021/05/15]. Dostupné z: https://docs.osgi.org/specification/osgi.core/8.0.0/framework.introduction.html.

[9] *Overview* [online]. Pydantic, 2021. [cit. 2021/05/06]. Introduction to Pydantic. Dostupné z: https://pydantic-docs.helpmanual.io/.

[10] *RestTemplate* [online]. Spring, 2021. [cit. 2021/05/16]. Javadoc pro RestTemplate. Dostupné z: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html.

[11] *Structure of a SOAP message* [online]. IBM, 2021. [cit. 2021/05/06]. Dostupné z: https://www.ibm.com/docs/en/cics-ts/5.4?topic=format-structure-soap-message.

[12] *Introduction* [online]. Starlette, 2021. [cit. 2021/05/06]. Introduction to Starlette. Dostupné z: https://www.starlette.io/.

[13] *WebClient* [online]. 2021. [cit. 2021/5/16]. Javadoc pro WebClient. Dostupné z: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html.

[14] *Cacheable* [online]. Mozilla Corporation, 2021. [cit. 2021/05/10]. Definition of cachable Http response. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/cacheable.

[15] *HTTP request methods* [online]. Mozilla Corporation, 2021. [cit. 2021/05/10]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[16] *Idempotent* [online]. Mozilla Corporation, 2021. [cit. 2021/05/10]. Definition of Idempotentess Http method. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Idempotent.

[17] *Compilation Overview* [online]. Oracle, 2021. [cit. 2021/05/14]. Dostupné z: https://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html.

[18] *Package javax.ws.rs.client* [online]. Oracle, 2015. [cit. 2021/05/10]. Dostupné z: https://docs.oracle.com/javaee/7/api/javax/ws/rs/client/package-summary.html.

[19] *JAXRS client* [online]. Oracle, 2015. [cit. 2021/05/16]. Javadoc pro javax.ws.rs.client. Dostupné z: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html.

[20] *Safe* [online]. Mozilla Corporation, 2021. [cit. 2021/05/10]. Definition of Safeness Http method. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Safe.

[21] *Chapter 18. WADL Support* [online]. Jersey, 2021. [cit. 2021/05/11]. Dostupné z: https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/wadl.html.

[22] Andronache, M. *JAX-RS is just an API!* [online]. Baeldung, 2019. [cit. 2021/05/10]. Dostupné z: https://www.baeldung.com/jax-rs-spec-and-implementations.

[23] Bass, L. – Clements, P. – Kazman, R. *Software Architecture in Practice.* Addison-Wesley Professional, 3rd edition, 2012. ISBN 0321815734.

[24] Benjamin J Evans, C. N. J. G. *Optimizing Java.* O'Reilly Media, Inc., 2018. ISBN 9781492025795.

[25] Berners-Lee, T. *Web Services* [online]. W3C, 2009. [cit. 2021/04/29]. Dostupné z: https://www.w3.org/DesignIssues/WebServices.html.

[26] Berners-Lee, T. *Web Services* [online]. W3C, 2004. [cit. 2021/05/13]. Dostupné z: https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice.

[27] Bierman, G. *Chapter 1. Introduction* [online]. Oracle, 2021. [cit. 2021/05/07]. The Java Langauge Specification, Java SE 16 Edition. Dostupné z: https://docs.oracle.com/javase/specs/jls/se16/html/jls-1.html.

[28] Brada, P. – Jezek, K. Repository and meta-data design for efficient component consistency verification. *Science of Computer Programming.* 2015, 97, s. 349–365. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2014.06.013. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0167642314002925.

[29] Brito, G. – Valente, M. T. REST vs GraphQL: A Controlled Experiment. In *2020 IEEE International Conference on Software Architecture (ICSA)*, s. 81–91, 2020. doi: 10.1109/ICSA47634.2020.00016.

[30] Brito, G. – Mombach, T. – Valente, M. T. Migrating to GraphQL: A Practical Assessment. s. 140–150, 2019. doi: 10.1109/SANER.2019.8667986.

[31] Bruneton, E. *A Java bytecode engineering library* [online]. ASM project, 2007. [cit. 2021/05/14].

[32] Bruneton, E. – Lenglet, R. – Coupaye, T. ASM: A code manipulation tool to implement adaptable systems. 2002, s. 1–12.

[33] Buczkowski, M. *Python REST frameworks performance comparison* [online]. Grandmetric, 2020. [cit. 2021/05/06]. Dostupné z: https://www.grandmetric.com/2020/08/10/python-rest-frameworks-performance-comparison/.

[34] Burke, B. *RESTful Java with JAX-RS 2.0, 2nd Edition.* O'Reilly Media, Inc., 2013. ISBN 9781449361341.

[35] Dave Swersky. *Hapi vs. Express in 2018: Node.js Framework Comparison* [online]. dzone, 2018. [cit. 2021/05/07]. Dostupné z: https://dzone.com/articles/hapi-vs-express-in-2018-nodejs-framework-compariso.

[36] Capote, J. *How does a maven repository work?* [online]. pacakgecloud:blog, 2017. [cit. 2021/05/14]. Dostupné z: https://blog.packagecloud.io/eng/2017/03/09/how-does-a-maven-repository-work/.

[37] Eby, P. J. *PEP 333 – Python Web Server Gateway Interface v1.0* [online]. Python, 2003. [cit. 2021/05/06]. Description of WSGI. Dostupné z: https://www.python.org/dev/peps/pep-0333/.

[38] Facebook. *GraphQL* [online]. 2021. [cit. 2021/05/02]. Current Working Draft - GraphQL. Dostupné z: https://spec.graphql.org/draft/.

[39] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, IRVINE, 2000.

[40] Fielding, R. T. – Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014. Dostupné z: https://rfc-editor.org/rfc/rfc7231.txt.

[41] Hadley, M. *Web Application Description Language* [online]. W3C, 2009. [cit. 2021/05/11]. Dostupné z: https://www.w3.org/Submission/wadl/.

[42] Harrand, N. et al. The Strengths and Behavioral Quirks of Java Bytecode Decompilers. *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Sep 2019. doi: 10.1109/scam.2019.00019. Dostupné z: http://dx.doi.org/10.1109/scam.2019.00019.

[43] Harrand, N. et al. Java decompiler diversity and its application to meta-decompilation. *Journal of Systems and Software*. 05 2020, 168, s. 110645. doi: 10.1016/j.jss.2020.110645. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0164121220301151.

[44] Hessová, G. Automatizovaná rekonstrukce rozhraní webových služeb reverzním inženýrstvím. Master's thesis, Západočeská univerzita v Plzni, 2018.

[45] *Introduction to GraphQL* [online]. Facebook, 2021. [cit. 2021/05/02]. Dostupné z: https://graphql.org/learn/.

[46] Ioannis K. Chaniotis, N. D. T. K.-I. D. K. Is Node.js a viable option for building modern webapplications? A performance evaluation study. *Computing*. March 2014, 97, s. 1023–1044. ISSN 1436-5057. Dostupné z: https://doi.org/10.1007/s00607-014-0394-9.

[47] Jez Humble, D. F. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Video Enhanced Edition*. Addison-Wesley Professional, 2010. ISBN 9780321670250.

[48] Kučera, J. Úložiště komponent podporující kontroly kompatibility. Master's thesis, Západočeská univerzita v Plzni, 2011.

[49] Lum, M. *Is Express the Best Option* [online]. Medium, 2020. [cit. 2021/05/07]. Express vs Koa vs Hapi. Dostupné z: https://medium.com/@mklum88/is-express-the-best-option-c5990ac9232e.

[50] Nilo Mitra, Y. L. *SOAP Version 1.2* [online]. W3C, 2007. [cit. 2021/04/29]. Dostupné z: https://www.w3.org/TR/soap12-part0/.

[51] Pjeřimovský, D. Vytváření a ukládání popisu webových služeb v úložišti CRCE. Master's thesis, Západočeská univerzita v Plzni, 2015.

[52] Rania Khalaf, F. L. *On Web Services Aggregation.* Springer, 2003. ISBN 978-3-540-39406-8.

[53] Riehle, D. *Framework Design: A Role Modeling Approach.* PhD thesis, Universität Hamburg, Hamburg, 2000.

[54] Roberto Chinnici, A. R. S. W. J.-J. M. *Web Services Description Language (WSDL) Version 2.0* [online]. W3C, 2007. [cit. 2021/05/01]. Dostupné z: https://www.w3.org/TR/wsdl/.

[55] Tim Lindholm, G. B. A. B. D. S. F. Y. *Chapter 1. Introduction* [online]. Oracle, 2021. [cit. 2021/05/07]. Java Virtual Machine Specification, Java SE 16 Edition. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se16/html/index.html.

[56] Velandia, J. et al. JAX-RS Implementations: A Performance Comparison. *Journal of Telecommunication.* 03 2018, 10, 1-8, s. 139–144. ISSN 2289-8131.

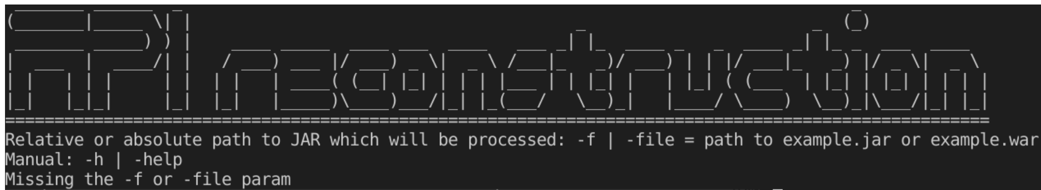
[57] Virender Ranga, A. S. API Features Individualizing of Web Services: REST and SOAP. *International Journal of Innovative Technology and Exploring Engineering.* July 2019, 8, s. 664–671. ISSN 2278-3075. doi: 10.35940/ijitee.I1107.0789S19. Dostupné z: https://www.ijitee.org/wp-content/uploads/papers/v8i9S/I11070789S19.pdf.

B Uživatelský manuál

Vytvořený indexer je možné ovládat třemi způsoby: *přes příkazovou řádku*, pomocí *grafického rozhraní CRCE úložiště* nebo přes *REST API*.

B.1 Příkazová řádka

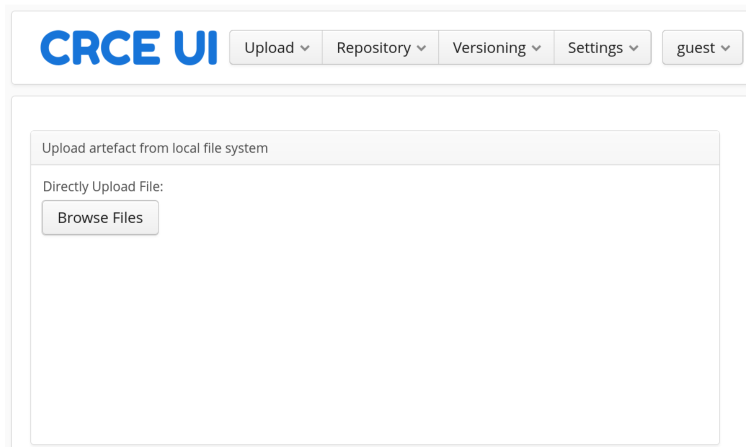
Parametr `-f` určuje cestu k *Jar/WAR* souboru. Pro zobrazení nápovědy, kterou je možné vidět na obrázku B.1, musí být program spuštěn bez parametru nebo s parametrem `-h`.



Obrázek B.1: Spuštění bez parametrů (zdroj: vlastní tvorba)

B.2 Webové rozhraní

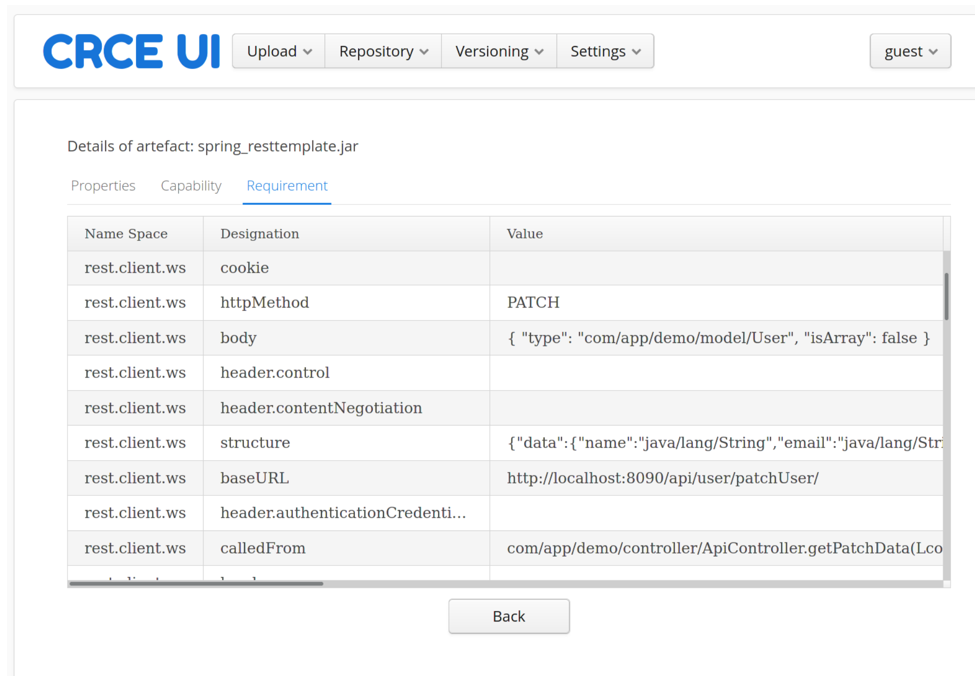
Úvodní obrazovka webového rozhraní, kterou je možné vidět na obrázku B.2, umožňuje nahrát soubor, který bude následně zpracován.



Obrázek B.2: Úvodní obrazovka (zdroj: vlastní tvorba)

Na obrázku B.3 je zobrazen obsah, který jednotlivé indexery dodaly do

CRCE úložiště. Pod sekcí `Requirement` jsou potom konkrétní výsledky indexeru klientů WS.



Obrázek B.3: Výsledek indexace (zdroj: vlastní tvorba)

B.3 REST API

Pro ukázku jsou zde uvedeny dvě adresy, přes které lze komunikovat prostřednictvím REST API s CRCE úložištěm.

- Dostupná metadata - `http://localhost:8080/rest/v2/metadat`
- Konkrétní výběr - `http://localhost:8080/rest/v2/metadata/<id>`

C Seznam obrázků

2.1 SOAP a WSDL (zdroj: vlastní tvorba). 12
2.2 REST (zdroj: vlastní tvorba) 20
2.3 Porovnání vel. objektů uložených v JSON objektu (zdroj: [30]). 21
2.4 Porovnání celkové velikosti JSON dokumentů (zdroj: [30]) . 21
2.5 GraphQL (zdroj: vlastní tvorba) 23

3.1 Architektura CRCE (zdroj: [28]) 33
3.2 Životní cyklus CRCE artefaktu (zdroj: [28]) 35
3.3 CRCE metadata model (zdroj: [28]) 39
3.4 OSGi životní cyklus (zdroj: [8]) 40

4.1 Kompilace Java (zdroj:[17]) 43
4.2 Životní cyklus zdrojového (Java) kódu (zdroj: [24]) 44
4.3 Výsledky dekompilace (zdroj: [43]) 54

5.1 Algoritmus (zdroj: vlastní tvorba) 82

6.1 Popis fungování výsledné aplikace (zdroj: vlastní tvorba) . . 83
6.2 crce-rest-client-indexer (zdroj: vlastní tvorba) 84

B.1 Spuštění bez parametrů (zdroj: vlastní tvorba) 103
B.2 Úvodní obrazovka (zdroj: vlastní tvorba) 103
B.3 Výsledek indexace (zdroj: vlastní tvorba) 104

D Seznam ukázek kódu

2.1 Query s připojenými subjects (zdroj: vlastní tvorba) 21
2.2 Argumenty dotazu (zdroj: vlastní tvorba) 22
2.3 Mutation (zdroj: vlastní tvorba) 22
2.4 Schéma pro typ School (zdroj: vlastní tvorba) 23
2.5 Importování JAX-RS implementací (zdroj: vlastní tvorba) . 25
2.6 Použití třídy Client pro komunikaci (zdroj: [18]) 25
2.7 Rest Template (zdroj: vlastní tvorba) 27
2.8 `WebClient` (zdroj: vlastní tvorba) 27
3.1 maven-metadata.xml (zdroj: vlastní tvorba) 32
3.2 pom.xml (zdroj: vlastní tvorba) 32
3.3 Package metadata (zdroj: [6]) 32
3.4 Repository záznam (zdroj: vlastní tvorba) 36
3.5 Capability záznam (zdroj: vlastní tvorba) 37
3.6 Resource záznam (zdroj: vlastní tvorba) 38
4.1 Constant Pool (zdroj: vlastní tvorba) 45
4.2 Zdrojový kód pracující s konstantami (zdroj: vlastní tvorba) 47
4.3 Přeložený kód vycházející z ukázky kódu 4.2 (zdroj: vlastní tvorba) 47
4.4 Zdrojový kód pracující s polem (zdroj: vlastní tvorba) 48
4.5 Přeložený kód vycházející z ukázky kódu 4.4 (zdroj: vlastní tvorba) 48
4.6 Ukázková třída (zdroj: vlastní tvorba) 50
4.7 Vytváření pole objektů včetně byte-kódu (zdroj: vlastní tvorba) 51
4.8 Zdrojový kód pro práci s atributy (zdroj: vlastní tvorba) . . 52
4.9 Přeložený kód vycházející z ukázky kódu 4.8 (zdroj: vlastní tvorba) 53
4.10 ClassVisitor (zdroj: [31]) 56
4.11 MyClassVisitor (zdroj: [44]) 58
4.12 MyMethodVisitor (zdroj: [44]) 58
5.1 Příklad využití konstant (zdroj: vlastní tvorba) 60
5.2 Nastavení hlavičky požadavku (zdroj: vlastní tvorba) 60
5.3 Příklad využití proměnné (zdroj: vlastní tvorba) 61
5.4 Uložení URL do proměnné (zdroj: vlastní tvorba) 61

5.5 Načtení klienta (zdroj: vlastní tvorba) 61
5.6 Metoda pro RestTemplate klienta (zdroj: vlastní tvorba) . . 62
5.7 Volání metody bez argumentu pro *Varargs* (zdroj: vlastní tvorba) 62
5.8 Instrukce pro volání metody `exchange` (zdroj: vlastní tvorba) 62
5.9 Atribut třídy držící URI požadavku (zdroj: vlastní tvorba) . 63
5.10 Instrukce pro získání atributu třídy (zdroj: vlastní tvorba) . 63
5.11 Příklad vytvoření požadavku (zdroj: vlastní tvorba) 64
5.12 Příklad volání klientské metody (zdroj: vlastní tvorba) . . . 65
5.13 Předávání typu jako argument metody (zdroj: vlastní tvorba) 65
5.14 Část překladu anonymních generik (zdroj: vlastní tvorba) . . 66
6.1 Konfigurace - RequestParameters (zdroj: vlastní tvorba) . . 86
6.2 Konfigurace - ArgDefinitions (zdroj: vlastní tvorba) 86
6.3 Konfigurace - wsClientData (zdroj: vlastní tvorba) 87
6.4 Konfigurace - wsClient (zdroj: vlastní tvorba) 88
6.5 Ukázka osgi.bnd souboru (zdroj: vlastní tvorba) 88
7.1 Test - přepisování hodnoty proměnné (zdroj: vlastní tvorba) 91
7.2 Získaný typ odpovědi (zdroj: vlastní tvorba) 91
7.3 Test - podmíněné přiřazení (zdroj: vlastní tvorba) 92
7.4 Test - získaný typ odpovědi (zdroj: vlastní tvorba) 92
7.5 Test - výčtové typy (zdroj: vlastní tvorba) 93
7.6 Test - získaná HTTP metoda a odpověď (zdroj: vlastní tvorba) 93
7.7 Test - Výstup z indexece průmyslové aplikace (zdroj: vlastní tvorba) 94
7.8 Test - dynamické vytváření URL (zdroj: vlastní tvorba) . . . 95

E Seznam tabulek

2.1 Omezení rozhraní pro REST (zdroj: [39]) 14
2.2 HTTP metoda GET (zdroj: [15]) 15
2.3 HTTP metoda HEAD (zdroj: [15]) 16
2.4 HTTP metoda POST (zdroj: [15]) 16
2.5 HTTP metoda PUT (zdroj: [15]) 17
2.6 HTTP metoda DELETE (zdroj: [15]) 17
2.7 HTTP metoda CONNECT (zdroj: [15]) 18
2.8 HTTP metoda OPTIONS (zdroj: [15]) 18
2.9 HTTP metoda TRACE (zdroj: [15]) 19
2.10 HTTP metoda PATCH (zdroj: [15]) 19
2.11 SOAP vs REST vs GraphQL (zdroj: vlastní tvorba) 23
2.12 Java REST frameworky (zdroj: vlastní tvorba) 25
2.13 Porovnání JAX-RS implementací (zdroj: [56]) 26
2.14 Python REST frameworky (zdroj: vlastní tvorba) 28
2.15 Node.js REST frameworky (zdroj: vlastní tvorba) 29

3.1 OSGi stavy 40

4.1 Druhy instrukcí CONST (zdroj: vlastní tvorba) 46
4.2 Druhy instrukcí LDC (zdroj: vlastní tvorba) 46
4.3 Druhy instrukcí LOAD (zdroj: vlastní tvorba) 48
4.4 Druhy instrukcí STORE (zdroj: vlastní tvorba) 50
4.5 Druhy instrukcí INVOKE (zdroj: vlastní tvorba) 52
4.6 Instrukce pro práci s atributy třídy (zdroj: vlastní tvorba) . 52
4.7 Porovnání knihoven (zdroj: [32]) 57

5.1 Symboly argumentů klientů WS (zdroj: vlastní tvorba) . . . 67
5.2 Metody s typem požadavku (zdroj: [10]) 68
5.3 Požadavky s generikami pro *entity* a *object* (zdroj: [10]) . . . 68
5.4 Požadavky s gener. třídami pro *location* a *object* (zdroj: [10]) 69
5.5 Generické volání (zdroj: [10]) 71
5.6 Obalovací třída `HttpHeaders` (zdroj: [10]) 72
5.7 `Builder` (zdroj: [13]) 73
5.8 `RequestBodySpec` (zdroj: [13]) 73

5.9 Metody třídy `Client` (zdroj: [19]) 75
5.10 Metody třídy `Builder` (zdroj: [19]) 77

F Seznam zkratek

- API - Application Programming Interface
- BCEL - Byte Code Engineering Library
- CRCE - Component Repository supporting Compatibility Evaluation
- CRUD - create, read, update a delete
- HTTP - Hypertext Transfer Protocol
- ID - Identification Data
- JIT - Just In Time compilation
- JVM - Java Virtual Machine
- JSON - JavaScript Object Notation
- LIFO - Last in, first out
- OBR - OSGi Bundle Repositor
- OSGi - Open Services Gateway initiative
- REST - Representation State Transfer
- SOAP - Simple Object Access Protocol
- SOAC - Separation Of Concercns
- UDDI - Universal Description, Discovery, and Integration
- URI - Uniform Resource Identifier
- URL - Uniform Resource Locator
- URN - Uniform Resource Name
- Varargs - Variable arguments
- VM - Virtual Machine
- VS - Virtuální Stroj
- WADL - Web Application Description Language

- WS - Webová Služba
- WSDL - Web Services Description Language
- WSGI - Web Server Gateway Interface
- W3C - World Wide Web Consortium
- RPC - Remote Procedure Call
- XML - Extensible Markup Language
- ZIP - Formát pro kompresy dat
- PNG - Portable Network Graphics

G Struktura odevzdávaného archivu