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

Diplomová práce

Určování nahraditelnosti a kompatibility webových služeb

Plzeň 2020 Zdeněk Valeš

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 5. května 2020

Zdeněk Valeš

Abstract

The text of the abstract (in English). It contains the English translation of the thesis title and a short description of the thesis.

Abstrakt

Text abstraktu (česky). Obsahuje krátkou anotaci (cca 10 řádek) v češtině. Budete ji potřebovat i při vyplňování údajů o bakalářské práci ve STAGu. Český i anglický abstrakt by měly být na stejné stránce a měly by si obsahem co možná nejvíce odpovídat (samozřejmě není možný doslovný překlad!).

Obsah

1 Úvod

- k čemu je práce dobrá - co text práce obahuje - use casy

2 Webové služby

V této kapitole jsou definovány webové služby a související pojmy. Jsou zde představeny strojově čitelné způsoby popisu rozhraní služeb a protokoly sloužící ke komunikaci s webovými službami. Tato kapitola také obsahuje popis architektonického stylu REST.

2.1 Webová služba

Pojem 'webová služba' může mít odlišné významy[4] a existuje pro něj také spousta definic, z nichž některé jsou více či méně vhodné pro tento text. Například konsorcium W3C[1] definuje pojem webová služba následovně[16]:

> Webová služba je softwarový systém navržený pro podporu mezistrojové komunikace po síti. Webová služba má rozhraní, které je popsáno ve strojově čitelném formátu (konkrétně WSDL). Ostatní systémy interagují s webovými službami předepsaným způsobem za použití zpráv protokolu SOAP, které jsou typicky zprostředkované protokolem HTTP s využitím serializace XML a dalších webových standardů.

Tato definice je pro účely mé práce problematická z několika důvodů. Prvním z nich je omezení formátu popisu služby pouze na WSDL. Strojově čitelný popis rozhraní sice snižuje náročnost vytvoření klienta pro danou službu, nicméně WSDL není jediným takovým formátem a netřeba tedy z definice vyloučit služby používající jiné formáty jako například WADL, nebo JSON-WSP, jež přímo souvisí s mou prací. Závislost na jednom konkrétním formátu také nemusí být výhodná z hlediska budoucího vývoje, protože nezohledňuje možnost příchodu jiného formátu, nebo změny přístupu k webovým službám obecně. WSDL do verze 1.1 například neumožňovalo popis REST služeb, ty byly zohledněny až ve verzi 2.0[12].

Definice dále svazuje implementaci webové služby s použitím konkrétního protokolu pro výměnu dat, v tomto případě SOAP, čímž opět nezahrnuje služby používající jakékoliv jiné protokoly. I zde platí nevýhoda takové vazby v kontextu budoucího vývoje, jež byla zmíněna v předchozím odstavci. Webová služba by například přestala být webovou službou, pokud by došlo k změně protokolu pro výměnu dat ze SOAP na JSON-RPC.

[1] World Wide Web Consorcium

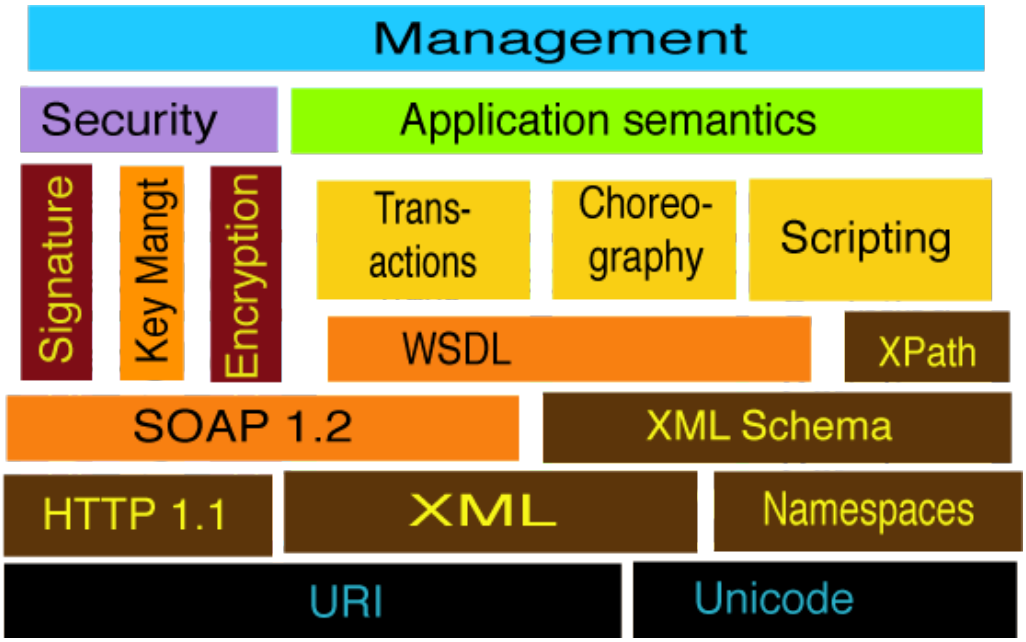
Posledním důvodem je vymáhání použití XML jako jediného formátu pro přenos dat i přes rostoucí popularitu jiných formátů, jako například JSON, nebo v poslední době YAML. Není také výjimkou, že webové služby podporují více formátů pro serializaci dat.

Z uvedených důvodů je potřeba zavést vhodnější definici s volnějšími kritérii tak, aby webové služby co možná nejméně omezovala. Definici, která služby popisuje primárně skrze komunikační protokol HTTP ve své knize nabízí R. Daigneau [7]:

> Webové služby poskytují prostředky pro integraci různorodých systémů a zpřístupňují znovupoužitelné business funkce skrze HTTP. HTTP je použito buď jako jednoduchý komunikační protokol, skrze který jsou přenášena data (např. SOAP/WSDL služby), nebo jako kompletní aplikační protokol, jež definuje sémantiku chování služby (např. RESTful služby).

Tato definice netrpí výše zmíněnými nedostatky a proto jsem se rozhodl ji využít pro účely mé práce. S webovými službami se váže několik dalších pojmů, které jsou v průběhu textu užívány, a pro úplnost jsou zde taktéž definovány:

- **Poskytovatel služby**: server na kterém je služba dostupná, případě organizace, která ji spravuje,
- **Konzument služby**: klient využívající službu, typicky se jedná o aplikaci,
- **Endpoint**: Jedna konkrétní operace služby, případě URI na které je dostupný nějaký zdroj,
- **Operace**: Operace webové služby, v tomto textu používáno ve stejném významu jako endpoint,
- **Service**: element `service` v popisu WSDL, při překladu do češtiny (služba) by mohl pojem kolidovat s pojmem 'webová služba', proto je pro element WSDL označující část SOAP služby ponechán anglický tvar service,
- **API**: Rozhraní webové služby.



Obrázek 2.1: Technologie zahrnuté ve webových službách

2.1.1 Použití webových služeb

Samotný pojem služba z technického pohledu typicky označuje funkcionalitu, jež vykonává nějakou podnikovou operaci, zajišťuje přístup k souborům, nebo obstarává generickou funkcionalitu jako přihlášení do systému. Webové služby umožňují tyto zpřístupnit různorodým klientům jako jsou například webový prohlížeč, nebo mobilní aplikace, nezávisle na technologiích použitých k jejich realizaci.

Ve srovnání s alternativami jako například distribuované objekty jsou webové služby snazší na použití, zejména kvůli jednoduché komunikaci skrze protokol HTTP a standardizované serializaci dat do formátů jako XML, nebo JSON. Díky tomu jsou služby snadno propojitelné a lze je tak skládat do větších celků za účelem realizace komplexních procesů[7].

2.2 Techniky webových služeb

vzory: - RPC API - Message API - Resource API

Faktory ke zvážení při návrhu API[8]: Zapouzdření, Kontrakt služby, Autonomnost

2.2.1 HTTP

HTTP[2] je bezestavový protokol aplikační vrstvy určený pro distribuované hypermediální systémy a tvoří základ informačního systému WWW[3], který je využívá od 90. let [2]. Celkem bylo specifikováno pět verzí protokolu: 0.9, 1.0, 1.1, 2.0 a 3.0. Tato sekce se zabývá protokolem HTTP/1.1, neboť právě vlastnosti představené v této verzi jsou relevantní mé práci. Komunikace přes HTTP typicky probíhá skrze TCP/IP spojení na portu 80.

Protokol je založen na principu volání-odpověď. Klient odešle žádost ve formě metody, identifikátoru a verze protokolu, následovanou tělem žádosti v MIME[4] formátu. Identifikátor žádosti posílané klientem je tvořen URI[5], což je sekvence znaků, která jasně určuje daný zdroj [3]. Tělo může obsahovat informace o klientovi, modifikátory žádosti a data pro příjemce. Server odpoví statusem, jež zahrnuje verzi protokolu a kód výsledku žádosti (chyba, nebo úspěch). Ten je následovaný zprávou, rovněž ve formátu MIME, obsahující informace o serveru a případně tělo odpovědi s daty pro klienta [2].

HTTP definuje množinu metod, jejichž účelem je specifikovat konkrétní operaci, která by měla být provedená na daným zdrojem, identifikovaným volaným URI. Výčet metod společně s popisem je uveden v tabulce 2.1.

Název	Popis užití	Idempotentní
GET	Získání informace náležící volanému URI	ANO
HEAD	Identická GET, ale odpověď neobsahuje tělo	ANO
PUT	Vytvoření nové, nebo aktualizace existující entity odpovídající volanému URI	ANO
DELETE	Smazání zdroje identifikovaným volaným URI	ANO
OPTIONS	Získání informací o možnostech komunikace	ANO
TRACE	Příjemce odešle zpět tělo přijaté zprávy	ANO
POST	Vytvoření nových dat vztažených ke zdroji identifikovaným volaným URI	NE
CONNECT	Rezervovaný název	NE

Tabulka 2.1: Metody definované v HTTP protokolu

Implementace obsluhy metod sice záleží na konkrétním poskytovateli, nicméně HTTP definuje tzv. bezpečné metody, jež by měly označovat pouze

[2]Hypertext Transfer Protocol
[3]World Wide Web
[4]Multipurpose Internet Mail Extensions
[5]Uniform Resource Identifier

operace určené k získání dat. Jedná se o metody GET a HEAD. Protokol u metod také zavádí vlastnost idempotentnosti pokud $N > 0$ identických volání způsobí stejné vedlejší efekty jako pouhé jedno volání [2].

2.3 Remote Procedure Call

Pojem Remote Procedure Call (dále jen RCP) obecně označuje technologii umožňující předání řízení mezi programy v různých adresních prostorech, které jsou mezi sebou propojeny komunikačním kanálem [13]. Tato sekce se zaměřuje na popis RPC zejména ve vztahu k webovým službám a komunikaci s nimi.

Obecně je RPC postaveno na principu volání-odpověď. Klient voláním předá poskytovateli informace o operaci, jež má vykonat, včetně vstupních dat. Příjemce (server) tuto operaci provede a v odpovědi vrátí zpracovaná data. Typickým příkladem použití je přesunutí náročného výpočtu z méně výkonného klienta na výkonný server.

V kontextu webových služeb se za 'RPC-like' považují služby založené na definici konsorcia W3C, tedy služby popsané WSDL předávající si data protokolem SOAP v těle HTTP volání a odpovědi. Klient takovou službu vnímá jako množinu operací které může volat, z nichž každá vykonává určitou úlohu [22]. Aby toto bylo možné, musí každá služba definovat jasné rozhraní pomocí kterého klient přistupuje k operacím a zpracovává vrácená data. Příkladem takového rozhraní je popis služby ve formátu WSDL.

Protože součástí mé práce bylo zpracování metadat služeb popsaných formátem WSDL, jež typicky komunikují skrze SOAP, zabývají se další odstavce právě tímto protokolem. Popisný formát WSDL je detailněji rozveden na konci této kapitoly.

2.3.1 SOAP

RCP označuje způsob komunikace, kdy klient předá serveru

- definováno v [13] - popsat obecně architekturu RPC - popsat RPC webové služby - popsat SOAP - specifikace SOAP: https://www.w3.org/TR/soap12-part1/#intro - protokol aplikační vrstvy SOAP (web service)

- XML pro popis datového modelu

2.4 REST

Representational State Transfer (dále jen REST) je architektonický styl tvorby webových služeb, který ve své disertační práci [9] popsal R. T. Fielding. Služby tvořené v tomto stylu se od služeb definovaných podle definice konsorcia W3C se liší zejména využitím protokolu HTTP pro sémantiku aplikace a vyznačuje se také absencí stavu.

R.T Fielding definoval styl REST skrze množinu omezení, jež jsou aplikována na systém vzájemně komunikujících komponent, což jsou v rámci této práce webové služby. Vybraná omezení jsou popsána v následujících odstavcích.

Klient-server

kvůli separation of concenrs; oddělení klienta od uložení dat umožňuje nezávislý vývoj jednotlivých komponent

Absence stavu

vše pro zpracování requestu musí být obsaženo v requestu a není možné využít žádný kontext uložený na serveru; tím dojde ke zpřehlednění a zlepšení spolehlivosti (protože je snazší se zotavit z částečných chyb); nevýhodou je možné snížení síťové performance a vzhledem k uložený kontextu na straně klienta je také na straně serveru nutná kontrola jeho konzistence

Cache

cache je na straně klienta, lze cacheovat pouze response, které jsou takto označené; tím lze zvýšit efektivitu

Uniformní rozhraní

(mezi komponentami) tím je docíleno oddělení služby, kterou komponenta poskytuje od její implementace; nevýhoda je zhoršení efektivity, protože informace jsou přenášeny ve standardním formátu, místo formátu, který by byl nativní aplikaci; 4 omezení rozhraní: identifikace zdrojů, manipulace se zdroji skrze reprezentace, sebe-popisující zprávy a hypermedia jako engine aplikačního stavu

Vrstvený styl

každý vrstva komunikuje pouze se seousendí vrstvou a nevidí vrstvy za nimi; to vede na větší nezávislost

2.4.1 Zdroj a jeho reprezentace

TODO: formulace Architektonický styl REST ignoruje implementační detaily jednotlivých komponent a zaměřuje se spíše na jejich roli, vzájemnou komunikaci interpretaci důležitých dat. Architektura vycházející z toho stylu se nazývá ROA[6] a ve své publikaci[15] ji popsal L. Richardson. Tato sekce v popisuje význam zdroje a jeho reprezentací v REST.

Klíčovou abstrakcí informace v REST je zdroj *resource*, jež může představovat jakoukoliv pojmenovatelnou informaci jako například dokument, obrázek, webovou službu, nebo kolekci jiných zdrojů. R. T. Fielding ve své práci[9] definuje zdroj R jako:

> funkci $M_R(t)$, která pro čas t vrací množinu ekvivalentních entit. Tato množina může být prázdná, nebo obsahovat reprezentace daného *resource*, případně jeho identifikátory

Pojem zdroj tedy neoznačuje konkrétní objekt, nebo data, ale abstraktní koncept, který je konkrétním objektem, nebo daty pouze reprezentován. Jako příklad lze uvést systém pro správu verzí, kde identifikátor 'poslední verze' označuje zdroj a samotná verze (0.8, 0.9, ...) jeho reprezentaci, která se s časem mění. Reprezentace různých zdrojů se mohou shodovat. V návaznosti na předchozí příklad by zdroje identifikované 'poslední verze' a 'verze 0.8' mohly být v určitý čas t reprezentovány stejnou verzí '0.8'.

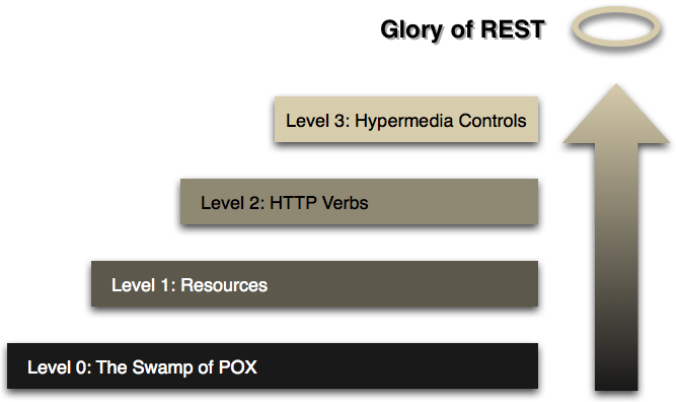
Každý zdroj má alespoň jeden identifikátor, v případě Webu se jedná o URI. Tento identifikátor by měl být co nejvíce popisný a v rámci jedné služby dodržovat stejný styl[15].

S reprezentacemi zdrojů je manipulováno HTTP voláním jejich identifikátorů (tedy URI) s nastavenou HTTP metodou, jež byly popsané v části 2.2.

2.4.2 RESTful služba

TODO: Formulace Jedním z primárních účelů webových služeb založených na stylu REST je manipulace s XML reprezentacemi zdrojů, skrze jednotnou množinu bezestavových operací, jimiž typicky jsou *create*, *retrieve*, *update* a *delete*[16]. Míra, kterou jednotlivé služby dodržují styl REST se různí a pojem 'RESTful' označuje ty, které REST dodržují nejvíce. Model zabývající se 'RESTful vyspělostí' dané služby, jež byl představen L. Richardsonem, se nazývá Richardson Maturity Model a ve svém online článku [10] jej rozvedl M. Fowler. Model, uvedený na obrázku 2.2, definuje čtyři níže popsané úrovně a pokud tyto služba splňuje, je označena jako RESTful (plně RESTová).

[6] Resource Oriented Architecture



Obrázek 2.2: Ilustrace Richardsonova modelu vyspělosti

0. úroveň

Služby v 0. úrovni se vyznačují použitím protokolu HTTP pro komunikaci, ale implementují vlastní sémantiku interakce, jež je typicky podobná stylu RPC. Služba je tedy například tvořena endpointem *bookingService* a na tento jsou směřovány všechny volání obsahující informace o tom, co má služba vykonat.

Zdroje

1. úroveň zavádí do služby zdroje, tak jak byly popsány v sekci 2.4.1. Místo volání jednoho univerzálního endpointu služby jsou nyní volány endpointy jednotlivých zdrojů.

HTTP metody

Předchozí úrovně používají HTTP metody nezávisle na jejich účelu. Endpoint každého zdroje z předchozí úrovně by tedy například očekával volání POST i když by pouze vracel data. 2. úroveň tento přístup mění a služby používají HTTP metody tak, jak jsou v HTTP definované. Volání GET endpointu zdroje tedy vrací data, zatímco POST zdroj vytváří, nebo mění.

Hypermediální řízení

Poslední úroveň zavádí princip HATEOAS[7], ve kterém je klientovi s reprezentacemi zdrojů navíc předána informace o další možné manipulaci s nimi. Tím je docíleno většího oddělení klienta od poskytovatele, neboť server může

[7]Hypertext As The Engine Of Application State

mimo jiné měnit schéma URI zdrojů a nenarušit tím schéma používané klientem.

2.4.3 Verzování REST služeb

Webové služby dodržující poslední úroveň Richardsonova modelu by za předpokladu statických vstupních bodů nemusely svou verzi vystavovat. Nicméně existuje nezanedbatelná část, jež na tuto úroveň nedosahují a tedy udržují statické rozhraní pro každou verzi, protože jakákoliv změna v schématu URI se negativně projeví na straně klienta. Ten totiž díky absenci HATEOAS nemůže tuto změnu dynamicky nijak zjistit je proto odkázaný na rozhraní služby.

Může tedy nastat případ kdy poskytovatel v zájmu udržení zpětné kompatibility udržuje více verzí jedné služby současně. Existují různé způsoby jak vystavit konkrétní verzi, jedním z nich je uvedení verze v URI zdroje. Jedná se o běžnou a snadno implementovatelnou praktiku [21][15]. Dopady použití této praktiky na mou práci jsou rozebrány v částech 4.2 a 5.3.3.

2.5 Formální popis rozhraní webových služeb

- existují různé, strojově čitelné, formáty pro popis API

- WSDL 1.1, 2.0, WADL (REST), JSON-WSP

- Swagger, Raml, OpenApi - nicméně těmi se v práci nezabývám, protože zatím neexistují indexery schopné z těchto extrahovat metadata

- v případě REST bohužel není nic formálně nutné (oproti třeba SOAP), takže specifikace API nemusí být kompatibilní, nemusí být úplné, nebo můžou být ad-hoc (např. slovní popis ve Word dokumentu) a tím pádem nemusí existovat univerzální způsob strojového čtení těchto specifikací

- v mé práci se věnuji především formátům WSDL, WADL a JSON-WSP, protože pro tyto v současnosti existují indexery

2.5.1 WSDL

Formát WSDL je využíván převážně pro popis webových služeb využívajících protokol SOAP k výměně dat, nicméně od verze 2.0 lze popsat i REST služby. Formát současně existuje ve verzích 1.1 a 2.0. Z důvodů staří verze 1.1, jež byla publikována v roce 2001 je tento popis zaměřen především na verzi 2.0.

WSDL je formát založený na XML popisující webové služby jako množiny endpointů operujících nad zprávami (*messages*), jež obsahují buď dokumentově, nebo procedurálně orientované informace. Abstraktně popsané operace a zprávy jsou spojeny s konkrétním síťovým protokolem a formátem zpráv, čímž je definován endpoint. Související endpointy jsou sdruženy do *services* [19]. Tabulka 2.2 obsahuje popis vybraných elementů použitých ve formátu WSDL 2.0.

Název	Popis
service	Kolekce endpointů, na kterých je služba dostupná
endpoint	Detaily endpointu, na kterém je služba dostupná
binding	Implementační detaily nezbytné k přístupu ke službě
operation	Implementační detaily konkrétní operace
types	Definice vlastních typů

Tabulka 2.2: Výběr elementů pro formát WSDL 2.0

2.5.2 WADL

WADL je strojově čitelný formát, jehož hlavním účelem je popis webových služeb založených na HTTP (například REST). Formát vznikl jako pokus o standardizaci popisu webových služeb, které jsou typicky popsané nestandardní, případě vzájemně odlišnou textovou dokumentací[17].

Formát je založený na XML a popisuje webové služby jako kolekce zdrojů, nad kterými definuje operace (*methods*). Zdroje *resource* je možné vnořovat a každý zdroj je identifikován URI sestaveným podle předem daného vzoru. Tabulka 2.3 obsahuje popis několika vybraných elementů, použitých k popisu webové služby.

Název	Popis
resources	Kolekce zdrojů poskytovaných službou
resource	Popis konkrétního zdroje
method	Popis operace, která může být nad zdrojem provedena

Tabulka 2.3: Výběr elementů pro formát WADL

2.5.3 JSON-WSP

Formát JSON-WSP je založený na JSON vychází z JSON-RPC což je protokol pro vzdálené volání procedur. JSON-WSP popisuje rozhraní webo-

vých aplikací a i když měl původně nahradit JSON-RPC, je zatím ve stavu neformálního návrhu a není nijak standardizovaný, ani neexistuje RFC s jeho specifikací, nebo JSON schéma. Neoficiální specifikace je udržována na Wikipedii[20].

JSON-WSP definuje službu pomocí objektů *description*, *request*, *response* a *fault*. Objekt *description* popisuje rozhraní služby, obsahuje mimo jiné název služby, URL služby a kolekci metod, které služba dokáže vykonat. Objekty *request*, *response* a *fault* definují podobu dat, které služba přijímá, respektive vrací. Tabulka 2.4 uvádí výběr objektů definovaných v *description* relevantních pro mou práci.

Název	Popis
methods	Kolekce metod vykonávaných službou
method	Popis konkrétní metody včetně parametrů a návratového typu
types	Definice vlastních typů

Tabulka 2.4: Výběr elementů pro formát WADL

3 Datové typy a porovnávání

TODO: formulace

Tato kapitola rozebírá vlastnosti datových typů se zaměřením na jejich vzájemné porovnání. Vzhledem ke kontextu práce jsou zde popsány typové systémy jazků Java a XSD.

3.1 Typové systémy

- přednášky z FJP - něco od taových typech obecně: https://www.cs.cmu.edu/ rwh/introsml/core/
- jak jazyky řeší datové typy - rekurzivní vs. nerekurzivní - možná fajn na rekurzivní typy: http://web.mit.edu/6.005/www/sp16/classes/16-recursive-data-types/recursive/ - built-in typy (xsd, Java) - primitivní typy vs. objekty

3.2 Porovnávání datových typů

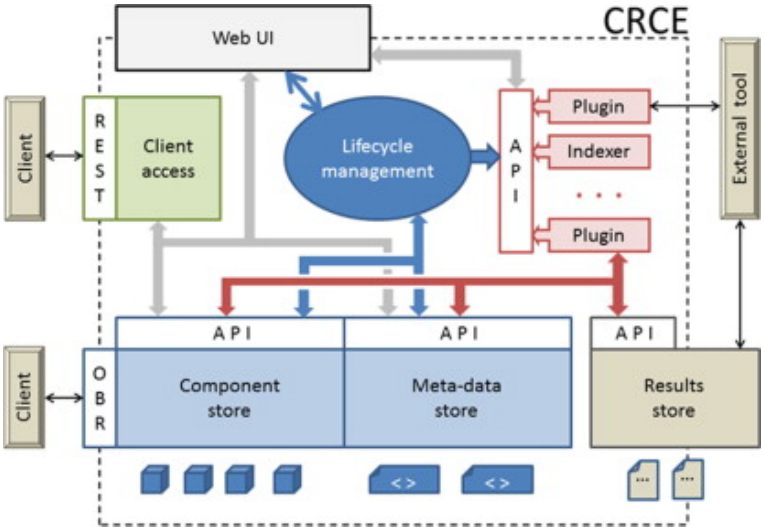
- jak to funguje - problémy při porovnání - subtyping vs. matching ([1]) - subtyping: A <: B <=> A může být použito v kdekoliv kde je očekáváno B - kontravariance: F'(A) <: F(B) <=> B <: A

4 Získávání metadat webových služeb v CRCE

Cílem této práce je vytvořit rozšíření pro úložiště CRCE[1], které bude schopno vyhodnocovat vzájemnou kompatibilitu indexovaných webových služeb. Tato kapitola představuje samotné úložiště, formát metadat a obecný způsob jejich získání. Na konci kapitoly je popsán princip fungování konkrétních rozšíření, která indexují webové služby (tzv. indexery).

4.1 CRCE

CRCE je komponentové úložiště dlouhodobě vyvíjené a spravované výzkumnou skupinou ReliSA na Katedře Informatiky ZČU, jehož primárním účelem je indexace a následná kontrola vzájemné kompatibility komponent. Úložiště je postaveno na modulární architektuře (viz obrázek 4.1) a je tedy možné přidat rozšíření pro indexaci a zpracování vlastních dat.



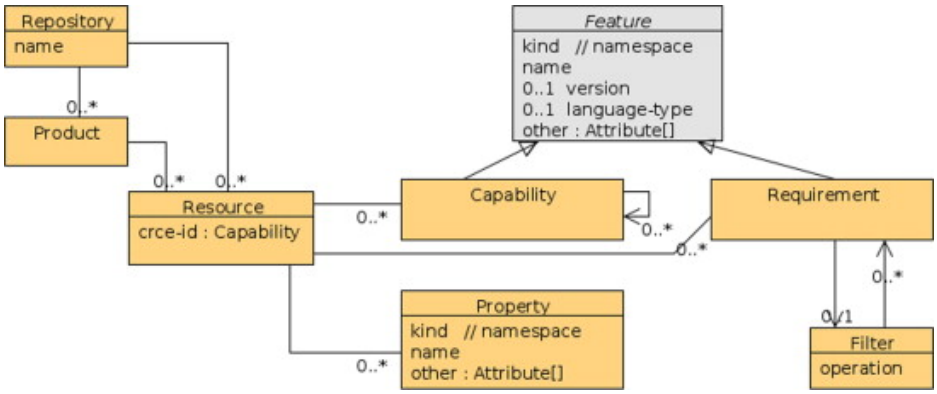
Obrázek 4.1: Architektura CRCE

[1]Component Repository supporting Compatibility Evaluation

4.1.1 Metadata komponent

Data, která vzniknou indexací komponenty a případným dalším zpracováním (např. porovnáním) jsou uložena do souboru metadat a představují klíčový element systému CRCE. Návrh struktury těchto metadat, který je naznačen na obrázku 4.2, vychází z konceptu OBR[2] jehož základními entitami jsou mimo jiné *Resource*, *requirements* a *capabilities*[5].

Entita *Resource* reprezentuje komponentu uloženou v CRCE, *requirements* a *capabilities* jsou množiny vlastností, popisující co komponenta ke své správné funkci vyžaduje, respektive co naopak poskytuje. Model také umožňuje přidání key-value atributů k jednotlivým vlastnostem a jejich detailům.



Obrázek 4.2: Reprezentace metadat v CRCE

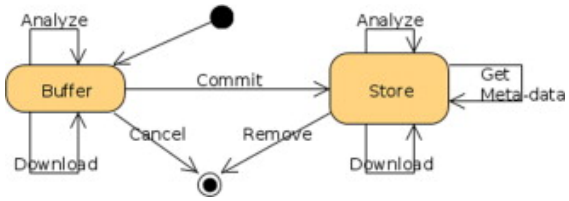
V mé práci jsem pracoval především s poskytovanými vlastnostmi (množina *capabilities*) a proto zde popíši hlavně jejich strukturu. Každá konkrétní vlastnost je reprezentována elementem *Capability* a od ostatních je odlišena identifikátorem *namespace*. Detaily konkrétní vlastnosti jsou popsány elementy *Property* a *Attribute*, kde *Property* reprezentuje logický celek několika atributů. Například parametr endpointu REST služby shlukuje atributy popisující jeho jméno, datový typ atp. *Attribute* pak představuje pár klíč-hodnota, který nese konkrétní informace jako např. jméno endpointu, nebo datový typ parametru.

Z obrázku 4.2 je vidět rekurzivní povaha elementu *Capability*, čehož je využito ke skládání jednodušších vlastností do složitějších celků. Vznikne tím stromová struktura, která je vhodná k modelování hierarchických dat mezi něž patří například popisy webových služeb. V případě takto komplexních vlastností je ke komponentně (*Resource*) přiřazena pouze jedna, tzv. kořenová, *Capability*, která reprezentuje celou vlastnost.

[2]OSGi bundle repository

4.1.2 Životní cyklus komponenty v CRCE

Úložiště bylo původně navrženo pro ukládání OSGi komponent, nicméně indexovat lze jakoukoliv komponentu. Komponenta je v CRCE popsána již zmíněnými metadaty a prochází vlastním životním cyklem naznačeným na obrázku 4.3.



Obrázek 4.3: Životní cyklus komponenty v CRCE

Životní cyklus má dvě hlavní fáze, jimiž jsou *Buffer* a *Store*. Komponenta po nahrání do úložiště nejprve prochází fází *Buffer*, v níž dojde k indexaci obsahu, kontrole vnitřní konzistence a kompatibility komponenty. Pokud touto fází projde bez chyb, může uživatel pomocí operace *commit* komponentu nahrát do trvalého úložiště, čímž dojde k přechodu do fáze *Store*.

V obou fázích jsou nad komponentou prováděny operace z nichž *analyze* je nejvíce relevantní mé práci, protože právě během této operace dochází ke sběru metadat (fáze *Buffer*) a dalším výpočtům nad nimi (fáze *Store*). Modul s rozšířením, který je předmětem mé práce bude zařazen mezi výpočty prováděné nad metadaty během *analyze*, kde bude vyhodnocovat vzájemnou kompatibilitu webových služeb. Způsoby sběru těchto metadat, ke kterým dochází ve fázi *Store*, jsou popsány v následující sekci.

4.2 Indexování webových služeb

V této podkapitole je krátce popsán obecný způsob indexace komponent v CRCE. Následně je podrobněji rozebráno indexování webových služeb konkrétními moduly a reprezentace popisu API metadaty v CRCE. Na závěr jsou také uvedeny limity indexování.

4.2.1 Obecná indexace komponenty

Indexace komponenty a související sběr metadat je proveden ve fázi *Buffer* k tomu určenými moduly. Ty jsou vzájemně nezávislé a obecně platí, že každý nich je zaměřen na sběr nějaké logicky ucelené části dat jako například informace o OSGi bundlu, maven koordináty, nebo popis webových služeb.

Vlastní data komponenty zůstávají během tohoto procesu nezměněná což v kombinaci s řetězením indexerů zaručuje mimo snadnou rozšiřitelnost také transparentní přístup ke komponentě každému z nich.

4.2.2 Indexace komponenty s webovou službou

Soubor obsahující implementaci, nebo popis API je v CRCE vnímán jako komponenta a prochází tedy zmíněným životním cyklem včetně výše popsané indexace. Pro popis webových služeb existuje mnoho standardních i nestandardních způsobů, jak již bylo zmíněno v kapitole 2. Z tohoto důvodu je není možné všechny analyzovat jedním indexerem a je nutné zaměřit se pouze na část z nich.

V současné době tedy existují dva moduly podporující několik popisných formátů a implementací. Konkrétně se jedná o modul pro indexaci webových služeb založených na architektonickém stylu REST[11] a o modul pro indexaci webových služeb s popisem ve formátu WSDL, WADL, nebo Json-WSP [14]. Oba dva vznikly v rámci diplomových prací a jsou stručně popsány v následujících sekcích.

Obrázek 4.4: Příklad indexované SOAP webové služby pro komix Dilbert

4.2.3 Struktura metadat popisující webovou službu

Jak již bylo zmíněno v části 4.1.1, hierarchickou strukturu popisu API lze vhodně vyjádřit metadaty CRCE. Během indexování komponenty reprezentující API jsou shromážděny různé typy popisných vlastností. Jedním z těchto typů je i samotný popis webové služby, který je reprezentován stromem metadat a ke komponentě je přiřazen skrze kořenovou *Capability*.

I když jsou různé druhy API indexovány rozdílnými moduly, výsledná metadata mají podobnou strukturu. Příklad metadat API je zobrazen na objektovém diagramu 4.4, jedná se o webovou službu, která vrací strip komixu Dilbert pro dnešní den.

Z uvedeného obrázku je vidět, že klíčové elementy API jako web service, nebo endpoint jsou reprezentovány objektem *Capability*. Detaily těchto elementů jsou popsány objekty *Property*. Jedná se zejména o parametry endpointů, těla requestů a response. Objekt *Attribute* pak představuje konkrétní hodnoty, jež jsou na obrázku naznačeny jen jako páry "klíč=hodnota". *Attribute* nemusí být vázaný jen na *Property* a lze jej použít i pro popis *Capability*, jak je tomu např. u objektu *WebService1*.

4.2.4 Indexování REST služeb

Modul pro indexování REST API vznikl v rámci diplomové práce Bc. Gabriely Hessové. Princip sběru dat je založen na binární analýze java archivů (JAR) obsahujících implementaci REST služeb pomocí frameworků splňujících specifikaci JAX-RS a frameworku Spring Web MVC. Modul byl testován na frameworcích Jersey verze 2.26, RESTEasy verze 3.0.16 a Spring Boot verze 1.5.9 [11].

Z implementace REST služby modul rekonstruuje kolekci endpointů s jejich parametry, tělem requestu, response a případnými parametry response. Každý endpoint je reprezentován entitou *Capability*, všechny další jeho vlastnosti pak entitami *Property*. Výčet všech indexovaných elementů rozhraní je uveden v tabulce 4.1.

4.2.5 Indexování webových služeb na základě popisu

Modul pro indexování webových služeb vznikl v rámci práce Bc. Davida Pejřimovského. Oproti předchozímu modulu pro indexaci REST služeb nepracuje tento s její implementací, ale s popisným souborem služby, tak jak bylo uvedeno v kapitole 2. Podporované formáty popisu služeb jsou WSDL (verze 1.1 i 2.0) pro SOAP webové služby, WADL a Json-WSP pro REST služby[14].

Element API	Entita v metadatech	Vstaženo k
Endpoint	Capability	-
Request body	Property	Endpoint
Request parameter	Property	Request
Response	Property Endpoint	Endpoint
Response parameter	Property	Response

Tabulka 4.1: Seznam indexovaných elementů REST služby a jejich reprezentací v metadatech

Struktura dat vytvořených pro REST služby (tedy z popisu WADL, nebo JSON-WSP) je podobná struktuře dat vytvořenou předchozím indexerem. Endpoint je tedy reprezentován entitou *Capability*, jeho parametry entitami *Property*. Z popisu Json-WSP je ještě vytvořena reprezentace response pro daný endpoint (entita *Property*). Z popisu WADL se žádné další vlastnosti endpointů nezískávají.

Z WSDL popisu je vytvořena reprezentace služeb, jež jsou popsány xml elementy `<wsdl:service>` a jejich vnořených endpointů. Endpoint je ve WSDL popsán elementem `<wsdl:port>` a má definované operace (elementy `<wsdl:operation>`), nicméně modul tyto nevnořuje a vytváří zjednodušenou reprezentaci. Model endpointu tedy obsahuje metadata získaná z elementů `<wsdl:operation>` a url definovanou v elementu `<wsdl:port>`. Služby i endpointy jsou v metadatech reprezentovány entitami *Capability*. Oproti REST službám, které mají jednu úroveň vnoření *Capability*, zde vznikají úrovně dvě. Výčet elementů API a jejich reprezentace v metadatech je uveden v tabulce 4.2.

Element API	Entita v metadatech	Vztaženo k
Service	Capability	-
Endpoint	Capability	Service
Endpoint parameter	Property	Endpoint
Response	Property	Endpoint
Response parameter	Property	Endpoint

Tabulka 4.2: Seznam indexovaných elementů webové služby a jejich reprezentací v metadatech

Logika parsování WSDL souborů (verze 1.1. i 2.0) obsahovala chybu ve čtení adresy endpointu. Ta byla očekávána v atributu `action` elementu `<wsdl:operation>`, který ale není uveden ve specifikaci WSDL 1.1 [18] ani WSDL 2.0 [19]. Tuto chybu jsem v rámci mé práce opravil.

4.2.6 Limity indexování

Současný proces indexování webových služeb naráží na dva známé problémy týkající se datových typů. Jedná se o indexování rekurzivních datových typů a absenci samotných definic typů.

První problém se týká zejména indexování webových služeb podle popisných souborů, protože ty definice typů obsahují. Způsoby rozvoje a ukládání datových typů jsou popsány v [1], nicméně logika zatím není implementována. Druhý problém se týká binární analýzy REST služeb, protože archiv s implementací služby nemusí nutně obsahovat definice tříd. Ty mohou být například v jiném artefaktu, na který se archiv pouze odkazuje skrze závislost.

Z těchto důvodů je do metadat uložen pouze název datového typu což snižuje možnosti porovnávání služeb.

5 Porovnávání webových služeb

Cílem této práce je rozšířit systém CRCE o funkcionalitu pro porovnání webových služeb. Tato kapitola detailně rozebírá porovnávací algoritmus společně s daty, nad kterými je možné porovnávač použít. Zároveň je zde popsán způsob vyhodnocení výsledků porovnání a formát uložení takto získaných dat.

5.1 Reprezentace rozdílů a kompatibility

Webové služby jsou popsány komplexní strukturou metadat, která byla uvedena v předchozí kapitole. Rozdíl mezi těmito strukturami lze vyjádřit pouhou pravdivostní hodnotou (stejné, nebo nestejné), nicméně takový přístup skrývá před klientem většinu informací, na základě kterých by se mohl rozhodnout o dalším postupu a tím značně snižuje použitelnost aplikace. Mimo to, samotná skutečnost, že metadata webových služeb se neshodují nemusí nutně z pohledu klienta znamenat nekompatibilitu. Z těchto důvodů je k reprezentaci rozdílů mezi službami třeba použít vhodnější metodu.

Článek [6] zabývající se možnostmi evaluace kompatibility komponent na základě relace subtypingu, nahlíží na komponentu skrze její rozhraní jako na datový typ a popisuje odlišnosti v různých úrovních stromu metadat (třída, operace, parametr, ...). Na základě těchto odlišností (a relace subtypingu) je pak určena míra kompatibility dvou komponent. Protože je webová služba v CRCE reprezentována komponentou a na její rozhraní se dá také nahlížet jako na kontrakt datového typu, je tento přístup vhodný k reprezentaci rozdílů mezi webovými. Oproti pouhé pravdivostní hodnotě je navíc klientovi poskytnuta detailní informace o konkrétních rozdílech mezi službami. Z těchto důvodů jsem se rozhodl použít výše zmíněný způsob reprezentace rozdílů v mé práci.

Datové struktury použité pro reprezentaci rozdílů dvou entit a jejich vzájemné kompatibility navržené v rámci citovaného článku se nazývají *Diff* a *Compatiblity*. *Diff* je definován jako rekurzivní typ, který uchovává jak konkrétní informace o rozdílu skrze podřazené *Diff* tak i úroveň odlišnosti zvanou *Difference*. Tyto úrovně jsou popsány tabulkou 5.1 a v citovaném článku tvoří obor hodnot funkce $diff(a, b) : Type \times Type \rightarrow Difference$.

Třída *Compatibility* uchovává informace o kompatibilitě dvou komponent (reprezentovány entitami *Resource*) společně s detaily jejich rozdílů, jež jsou reprezentovány stromem *Diff*.

Název úrovně	Zkratka	Váha	Popis
None	NON	1	$a = b$
Insertion	INS	2	a není definováno, ale b ano
Deletion	DEL	2	a je definováno, ale b ne
Specialization	SPE	3	b je subtyp a ($b <: a$)
Generalization	GEN	3	a je subtyp b ($a <: b$)
Mutation	MUT	4	kombinace *INS*/*SPE* a *DEL*/*GEN*
Unknown	UNK	5	a nelze porovnat s b

Tabulka 5.1: Popis úrovní rozdílů

Způsob vyhodnocení rozdílů dvou webových služeb společně s významem jednotlivých úrovní *Difference* a jejich vah pro klienta je popsán na konci této kapitoly. Pro přehlednost jsou jednotlivé úrovně rozdílů v průběhu kapitoly nazývané jejich zkratkami.

5.2 Algoritmus porovnání

Porovnávací algoritmus pracuje s metadaty popsanými v části 4.1.1. Jedná se o stromovou strukturou, jejíž uzly tvoří instance tříd *Capability*, *Property* a *Attribute*, kde objekty *Attributes* jsou listy této struktury . Soubor metadat může obsahovat další vlastnosti komponenty, která představuje webovou službu, ta však zůstanou nedotčena, protože algoritmus pracuje pouze s daty, která byla vytvořena indexery popsanými v části 4.2.

Moduly pro indexování webových služeb popsané v předchozí kapitole používají dvě různé množiny *namespace* identifikátorů po pojmenování entit *Capability*, *Property* a *Attribute*. Do budoucna je zároveň plánované rozšíření těchto modulů o funkcionalitu pro indexování datových typů a datová struktura metadat je tedy předmětem změny. Z těchto důvodů je algoritmus schopen porovnat pouze metadata vytvořená stejným indexerem a používající stejné *namespace* identifikátory. Není tedy možné vzájemně porovnat například metadata REST služby získaná binární analýzou JAR s metadaty získanými čtením JSON-WSP dokumentu i když by se mohlo jednat o jednu službu.

5.2.1 Popis algoritmu

Vstupem algoritmu jsou reprezentace obou webových služeb v podobě objektů *Resource*, z kterých jsou následně k porovnání vybrány kolekce endpointů, případě kolekce *service* obsahující endpointy. Výstupem algoritmu je objekt *Compatibility*, jež obsahuje detailní popis rozdílů mezi webovými službami ve formátu popsaném v sekci 5.1.

Před porovnáváním datových struktur je zkontrolována jejich kompatibilita. Ta je dána následujícími vlastnostmi:

- typ popisu: z čeho byla metadata získána (implementace, formát popisného souboru),
- typ komunikace: jakým způsobem lze službu volat,
- *identity capability*: obsahuje informace o identitě komponenty v CRCE.

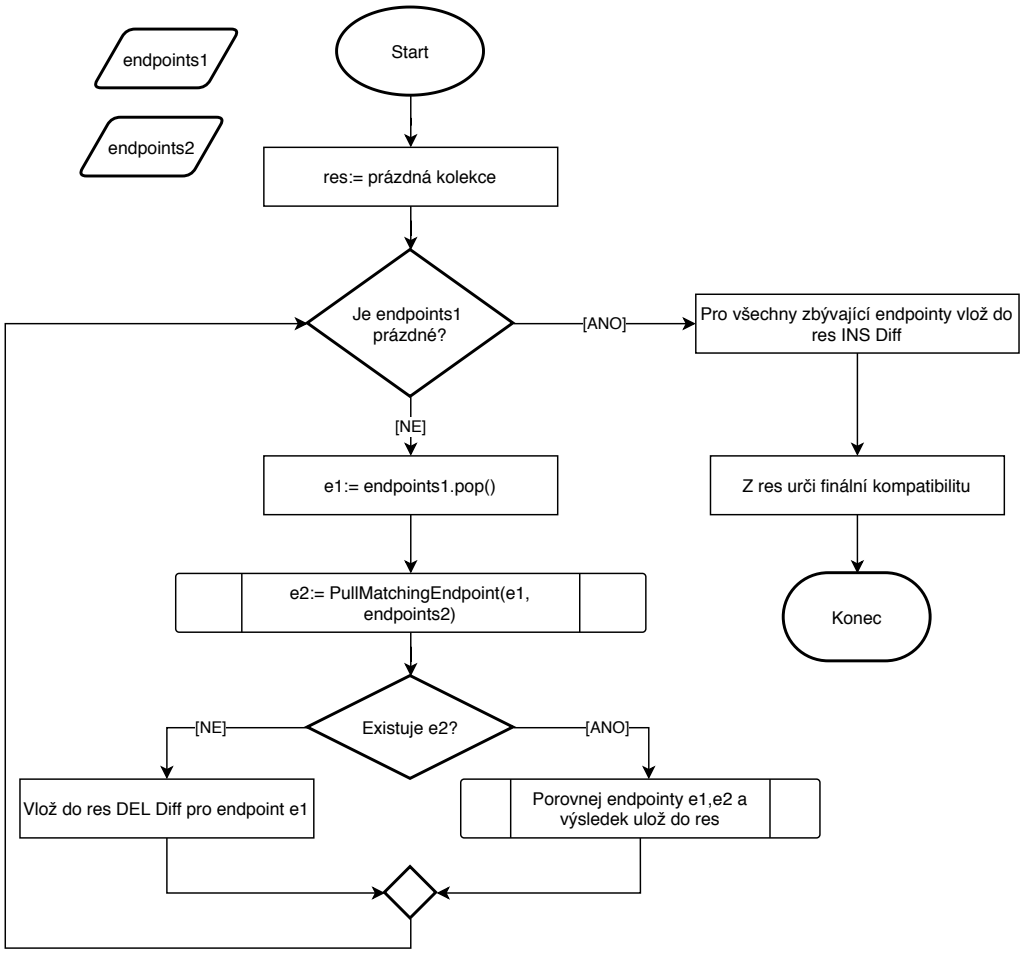
Pokud kontrola proběhne úspěšně, je spuštěn samotný algoritmus jehož průběh je naznačen na vývojovém diagramu 5.1. Vstupní data na obrázku označená jako *endpoints1* a *endpoints2* označují množiny endpointů první a druhé webové služby.

Tento postup analogicky platí i pro porovnání *service* v případě SOAP webových služeb s tím rozdílem, že krok 'Porovnej endpointy e1,e2 a výsledek ulož do res' by obsahoval porovnání endpointů definovaných v rámci jedné *service*, která je v CRCE reprezentována jako kolekce endpointů. V případě porovnávání *services* by množiny vstupních dat byly označené jako *services1* a *services2*.

Mezivýsledky porovnání jsou ukládány v datové struktuře *Diff*, která je detailně popsaná v sekci 5.1. Algoritmus postupuje po stromu metadat od kořene směrem k listům, ve kterých dojde k porovnání konkrétních hodnot a vyhodnocení rozdílů mezi nimi. Rozdíl mezi dvěma ne-koncovými uzly je vyhodnocen až po porovnání všech jejich potomků (nezávisle na pořadí). Algoritmus tedy postupuje zpět ke kořeni stromu, který na konci algoritmu obsahuje finální údaj o rozdílu obou služeb. Jednotlivé fáze porovnání jsou popsány v následujících odstavcích.

Výběr entit vhodných k porovnání

Tento odstavec se týká endpointů a *service*, protože tyto nejsou na pořadí závislé (na rozdíl např. od parametrů operace) a jejich pořadí v metadatech nelze ani předpokládat. Proto je nutné před samotným porovnáním nejprve vybrat dvojici entit k tomu vhodnou. V diagramu 5.1 je zde popsaný výběr reprezentován krokem 'PullMatchingEndpoint()'.



Obrázek 5.1: Vývojový diagram porovnávacího algoritmu

Endpoint *e1* (*service s1*) z první webové služby je vybrán sekvenčně, jak je naznačeno na diagramu 5.1. Druhý endpoint *e2* (*s2*) je pak vybrán na základě určité shody s metadaty endpointu *e1*, respektive *service s2*. V případě endpointů se konkrétně jedná o počet povinných parametrů, jméno a URL na které je daný endpoint dostupný. URL nemusí být shodné úplně. Pokud tak nastane, je nastavena vlajka MOV, která je detailně popsána v sekci 5.3.

O porovnatelnosti a rozdílu dvou *services* je rozhodnuto na základě úplné shody jejich jmen a typu (současně je používán pouze typ "`rpc/messaging`").

Porovnání dvou endpointů

Po výběru vhodné dvojice endpointů dojde k jejich porovnání. Postupně se porovnají (pokud jsou pro daný typ webové služby definovány) parametry, response a těla request. Způsob porovnání parametrů a response je rozveden

v následujících odstavcích. Metadata těla request jsou dostupná pouze v případě REST služeb indexovaných na základě implementace a detail jejich porovnání je znázorněn v tabulce 5.2. U atributu *isOptional* může dojít ke změně *GEN* pokud se povinný atribut stane nepovinným. V opačném případě se jedná o změnu *SPE*.

Název atributu	Možné výsledky
isArray	*NON*, *UNK*
isOptional	*NON*, *GEN*, *SPE*
dataType	*NON*, *GEN*, *SPE*, *UNK*

Tabulka 5.2: Porovnání atributů těl requestů endpointů

Porovnání response dvou endpointů

Element response je definovaný ve všech případech kromě služeb popsaných WADL. V případě REST služeb indexovaných na základě implementace jsou navíc definovány i parametry response a může také existovat více elementů response pro jeden endpoint. Ty jsou navzájem odlišeny atributem *id*, jehož hodnotu vytváří indexer. Detail porovnání response (včetně parametrů) REST služeb je uveden v tabulce 5.3, tabulka 5.4 pak obsahuje detail porovnání response ostatních služeb.

Název atributu	Možné výsledky
isArray	*NON*, *UNK*
dataType	*NON*, *GEN*, *SPE*, *UNK*
status	*NON*, *UNK*
parameterName	*NON*, *UNK*
parameterCategory	*NON*, *UNK*
parameterIsArray	*NON*, *UNK*
parameterDataType	*NON*, *GEN*, *SPE*, *UNK*

Tabulka 5.3: Porovnání atributů response a parametrů response endpointů REST služby

Název atributu	Možné výsledky
isArray (pouze JSON-WSP)	*NON*, *UNK*
dataType	*NON*, *GEN*, *SPE*, *UNK*

Tabulka 5.4: Porovnání atributů response ostatních služeb

Porovnání parametrů endpointů

Množiny porovnávaných atributů parametrů se navzájem liší podle typu metadat služby (viz kapitola 4.2). Kompletní přehled je uveden v tabulce 5.5. Parametry endpointů vhodné k porovnání jsou vybrány na základě jejich jména (atribut *name*). Po výběru dojde k porovnání zbylých atributů a rozdíl mezi parametry je vyhodnocen na základě rozdílů mezi jejich atributy.

Název atributu	Typ metadat služby	Možné výsledky
name	všechny	*NON*, *UNK*
dataType	všechny	*NON*, *GEN*, *SPE*, *UNK*
order	všechny krom WADL	*NON*, *UNK*
isOptional	všechny krom WSDL	*NON*, *GEN*, *SPE*, *UNK*
isArray	REST a JSON-WSP	*NON*, *UNK*
category	REST	*NON*, *UNK*

Tabulka 5.5: Porovnání parametrů endpointů

5.2.2 Porovnání datových typů

V současnosti jsou největším omezením porovnávacího algoritmu datové typy. Jak již bylo zmíněno v kapitole 4.2, jméno datového typu je jedinou dostupnou informací a tedy také jediným kritériem, podle kterého je lze porovnávat. To je dostatečné v případě vestavěných typů jako například třídy z balíku `java.lang`, nebo typy definované v xsd.

Nad těmito lze provádět plnohodnotné porovnání včetně kontroly generalizace (změny *GEN*, *SPE*). U vestavěných typů Javy je generalizace určena na základě dědičnosti a lze například určit, že datový typ s názvem `java.lang.Number` je generalizací typu s názvem `java.lang.Long`, protože třída *Number* je rodičem třídy *Long*. Obdobně je tomu i u vestavěných typů definovaných v XSD, kde je generalizace $a <: b$ definována v případě, že typ b je dost velký na pojmutí typu a. Například typ `xsd:long` dokáže reprezentovat číslo typu `xsd:int` a proto platí, že `xsd:int` $<:$ `xsd:long`.

Porovnání vestavěných typů je založeno na podmínce správně zaindexovaného jména datového typu. V případě analýzy byte kódu je tato podmínka splněna, nicméně u metadat získaných čtením popisných XML souborů tomu tak vždy nemusí být. Problém spočívá v předponě datového typu, která je součástí jeho jména a porovnávací algoritmus očekává její určitou hodnotu (konkrétně `xs`). Hodnota předpony je však určena definicí namespace v XML popisu a ta se může od očekávané lišit. Tato definice není indexerem ukládána a není proto možné ji během porovnání přesně určit.

Pro uživatelsky definované typy je z těchto důvodů porovnání omezeno pouze na úplnou shodu názvu včetně prefixu, protože na základě pouhého jména datového typu nejde s jistotou usoudit nic dalšího. Změny pro jednotlivé rozdíly mezi datovými typy jsou uvedeny v tabulce 5.6.

Vztah typů	Výsledek
$A = B$	*NON*
$A <: B$	*GEN*
$B <: A$	*SPE*
$A \neq B$	*UNK*

Tabulka 5.6: Rozdíly mezi datovými typy A a B

5.2.3 Časová složitost algoritmu

Algoritmus v zásadě porovnává kolekce endpointů webových služeb a jejich počet je tedy hlavní parametr, od kterého se odvíjí časová složitost. V případě SOAP webových služeb jsou operace definované v rámci *service* a je tedy třeba brát v úvahu i jejich počet. Elementární operací algoritmu je výběr vhodného páru entit (*service*, nebo endpoint) a jejich následné porovnání.

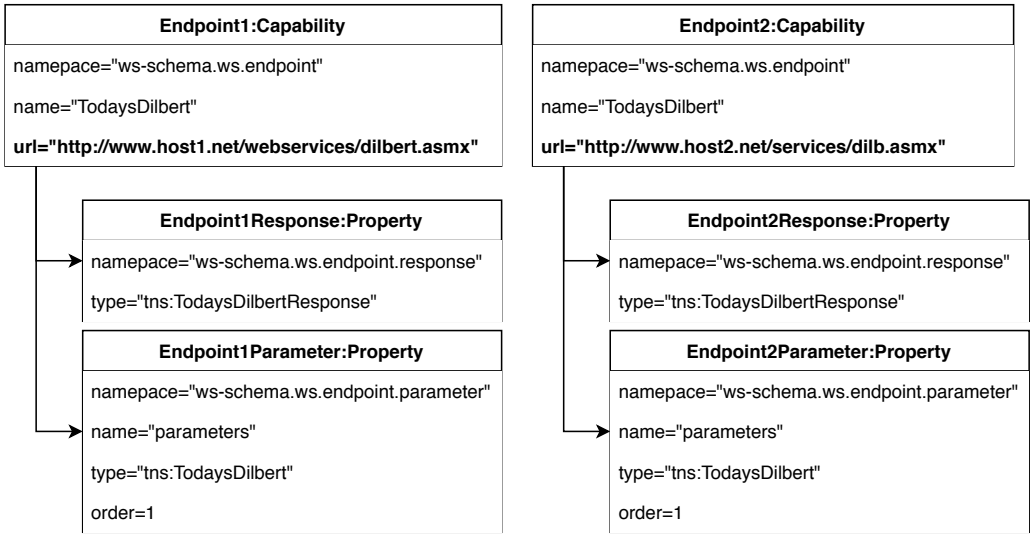
Proces výběru entit k porovnání, jež byl popsán v sekci 5.2.1, hledá k prvku z množiny první webové služby (*endpoint1*, *services1*) vhodný prvek v množině druhé webové služby (*endpoints2*, *services2*). Z obrázku 5.1 je vidět, že v případě nevhodného řazení entit bude potřeba zkontrolovat n prvků z množiny *endpoints2* (*services2*), než bude nalezen vhodný protějšek k porovnání. Pokud taková situace nastane pro každý prvek z množiny první webové služby, bude potřeba provést $n * n$ výběrů a následných porovnání. V případě vhodného řazení bude naopak potřeba provést pouze n výběrů, protože vhodné protějšky budou v obou množinách na stejném místě a k jejich nalezení bude vždy potřeba pouze jedno porovnání. Časová náročnost algoritmu pro jednotlivé typy služeb je shrnuta v tabulce 5.7.

Typ služby	Složitost
REST, WADL, JSON-WSP	$\Omega(n)$, $O(n^2)$ kde n je počet endpointů
WSDL	$\Omega(mn)$, $O((mn)^2)$ kde m je počet *service* a n je počet endpointů

Tabulka 5.7: Složitost algoritmu

5.3 Migrace webových služeb

V popisu porovnání metadat endpointů bylo zmíněno použití URL endpointu jako identifikátoru. Tento přístup naráží na problém v případě, že poskytovatel ponechá nezměněnou implementaci a webovou službu přesune, nebo provede změny v cestě k danému endpointu. Na obrázku 5.2 je uveden příklad dvou shodných endpointů, jež se liší pouze v URL. Aplikujeme-li výše popsaný algoritmus na tato data, skončí negativním výsledkem i přes to, že endpointy mají totožné rozhraní a klient by k nim mohl bez obtíží přistoupit.



Obrázek 5.2: Příklad shodných endpointů s rozdílnou URL

Podobným příkladem je i verze API v cestě k endpointu. Klient může mít požadavek na zjištění kompatibility API dvou různých verzí, ale algoritmus vrátí rozdíl *MUT*, protože endpointy z prvního API vyhodnotí jako chybějící v API druhém a naopak, právě kvůli rozdílným cestám. Tím vznikne kombinace rozdílů *DEL* a *INS*, která vede na *MUT*. Takové chování není žádané a je potřeba těmto problémům předcházet, proto je nutné případné změny v URL detekovat a brát je při porovnávání v potaz.

Jako řešení popsaného problému byl zaveden příznak "MOV", který je ortogonální k úrovni změny (*Difference*) popsané v předchozích sekcích a je součástí objektu *Diff*. Díky ortogonalitě je možné stále určit méně nebezpečné rozdíly jako generalizaci v příznaku parametru (SPE) a zároveň předat klientovi informaci o změně v URL.

Příznak MOV má smysl brát v úvahu jen u případů porovnání jejichž úroveň změny je v podmnožině NON, GEN, SPE, která je v rámci této sekce

označována jako bezpečná. Všechny ostatní úrovně představují v kontextu migrace příliš velkou změnu. Je tedy například nesmyslné nastavit příznak MOV u rozdílu dvou endpointů s úrovní změny *DEL*, protože samotná úroveň změny říká, že endpoint nebyl v druhé webové službě nalezen a proto nelze ani určit zda byl přemístěn.

5.3.1 Detekce změn v URL

Součástí URL endpointu je také jeho název, změny v URL se tedy dají detekovat na třech místech. Prvním z nich je doména (včetně protokolu), druhým je cesta k endpointu a třetím jeho jméno. URL všech endpointů obou webových služeb jsou podle těchto částí rozděleny, čímž vznikne 6 množin (2 pro každou část URL). Pokud platí, že $url_{i,1} \subseteq url_{i,2}$ kde $url_{i,j}$ je i-tá část url j-té webové služby, je daná část URL považována za nezměněnou. Příklad rozdělení URL dvou webových služeb vycházející z obrázku 5.2 je uveden v tabulce 5.8. V tomto příkladu by byla detekována změna v částech URL "Doména"a "Cesta".

$url_{i,j}$	Služba 1 (j=1)	Služba 2 (j=2)
Doména (i=1)	$\{http://host1.net\}$	$\{http://host2.net\}$
Cesta (i=2)	$\{webservices/dilbert.asmx\}$	$\{services/dilb.asmx\}$
Operace (i=3)	$\{TodaysDilbert\}$	$\{TodaysDilbert\}$

Tabulka 5.8: Příklad rozdělění URL na části podle obrázku 5.2

Výsledek detekce změn v URL nese tři příznaky, kde každý z nich určuje, zda byla v dané části URL detekována změna. Vzhledem k tomu, že algoritmus detekce pracuje pouze s URL a jmény endpointů, může dojde k pozitivnímu výsledku i v případě, že jsou porovnávány dvě odlišné webové služby. Aby se redukoval počet false-positives, je potřeba určit, které kombinace příznaků změny mohou vést na MOV a které představují příliš velké odchýlení. Tyto kombinace jsou uvedeny v tabulce 5.9, kde proměnné h, p, n označují změnu v doméně, cestě a jménu endpointu. Hodnota *true* značí změnu.

Z tabulky je vidět, že změna ve jménech endpointů vždy vede na záporný výsledek. Jedním z důvodů je fakt, že změna ve jméně operace je mutací webové služby a jedná se tedy o nekompatibilní změnu, jejíž řešení bylo popsáno v předchozích sekcích. Dalším důvodem je použití jména endpointu jako druhého kritéria pro výběr vhodného páru endpointů (prvním je URL). Bez něj by algoritmus zdegradoval na porovnání "každý s každým", což by značně snížilo efektivitu.

Kombinace	MOV	Zdůvodnění
$!h \wedge !p \wedge !n$	ne	Nebyla detekována žádná změna.
$h \wedge !p \wedge !n$	ano	Změna v doméně, může se jednat o migraci webové služby.
$!h \wedge p \wedge !n$	ano	Změna v cestě k endpointům, může se jednat o restrukturalizaci služby.
$h \wedge p \wedge !n$	ano	Může se jednat o kombinaci obou předchozích.
všechny ostatní	ne	Změna je příliš velká.

Tabulka 5.9: Kombinace změn vedoucích na MOV

5.3.2 Výběr entit k porovnání s příznakem MOV

Protože jedním z kritérií pro výběr vhodného páru endpointů je i URL, je potřeba upravit logiku výběru tak, aby byly brány v potaz výsledky detekce popsané v předchozí sekci. Při porovnávání URL dvou endpointů je podle kombinace změn daná část URL ignorována a pracuje se pouze s nezměněnou částí u které je vyžadována striktní rovnost.

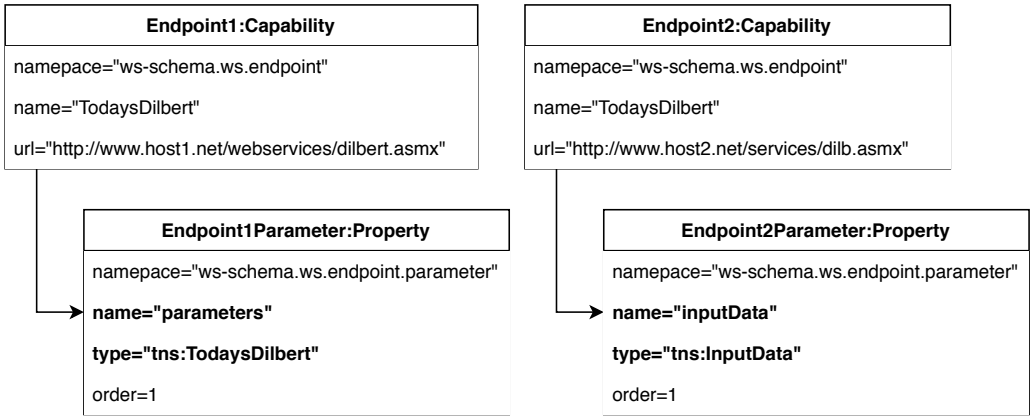
Tento způsob může vést na případy, kdy je dvojce endpointů vyhodnocena jako potencionálně vhodná k porovnání (s nastaveným příznakem MOV), nicméně porovnání skončí negativním výsledkem, například *UNK*. Pouhá akceptace takového výsledku a pokračování algoritmu by mohlo zapříčinit negativní vyhodnocení kompatibility webových služeb v případech, kdy lze mezi službami nají kompatibilní dvojice endpointů.

Příklad na obrázku 5.3 tento problém znázorňuje. Oba endpointy mají stejný název (atribut *name*) a proto by byly vybrány jako vhodné k porovnání s nastavenými MOV příznaky $h = t$, $p = t$, $o = f$. Hlubší porovnání by však skončilo výsledkem *UNK*, protože se neshodují jména a datové typy jejich parametrů a algoritmus by služby správně vyhodnotil jako nekompatibilní. V druhé webové službě by se nicméně mohl nacházet kompatibilní endpoint a je zde tedy možnost získání lepšího výsledku.

Algoritmus 'pick best'

Řešením zmíněného problému je algoritmus 'pick best', který postupně porovnává dvojice endpointů (k tomu předem vybrané) a jako výsledek vrátí dvojici s nejlepším rozdílem. Vstupem algoritmu je tedy endpoint z první webové služby *e1* a množina endpointů druhé webové služby *endpoints2*.

Algoritmus postupně vybírá elementy *e2* z *endpoints2* a pokud je pár $e1, e2$ vyhodnocen jako porovnatelný, dojde k detailnímu porovnání. V pří-



Obrázek 5.3: Příklad dvou rozdílných endpointů, pro které by byl nastaven příznak MOV

padě výsledku spadajícího do bezpečné množiny (NON, GEN, SPE) je pár $e1, e2$ vrácen a porovnání pro endpoint *e1* je ukončeno. V opačném případě je negativní výsledek uložen a z množiny *endpoints2* jsou vybírány další porovnatelné endpointy dokud algoritmus nedojde ke dvojici, jejíž rozdíl spadá do bezpečné množiny, nebo dokud není *endpoints2* vyčerpána. Pokud je množina *endpoint2* vyprázdněna, znamená to (alespoň částečnou) nekompatibilitu obou webových služeb a je vrácen první porovnávaný pár $e1, e2$.

5.3.3 Verze REST API v cestě k endpointu

Jednou ze speciálních změn detekovatelných v cestě k endpointu je verze API. Jedná se o běžnou praktiku [21] a k jejímu zpracování lze použít jednodušší přístup, než byl dosud popsán. Část cesty, která obsahuje verzi lze jednoduše vypustit a porovnat URL bez verze, což případě stejného API (s odlišností verze) znamená porovnání dvou identických URL. Příznak MOV je potom nastaven pokud jsou URL s verzemi rozdílné a URL bez verzí stejné.

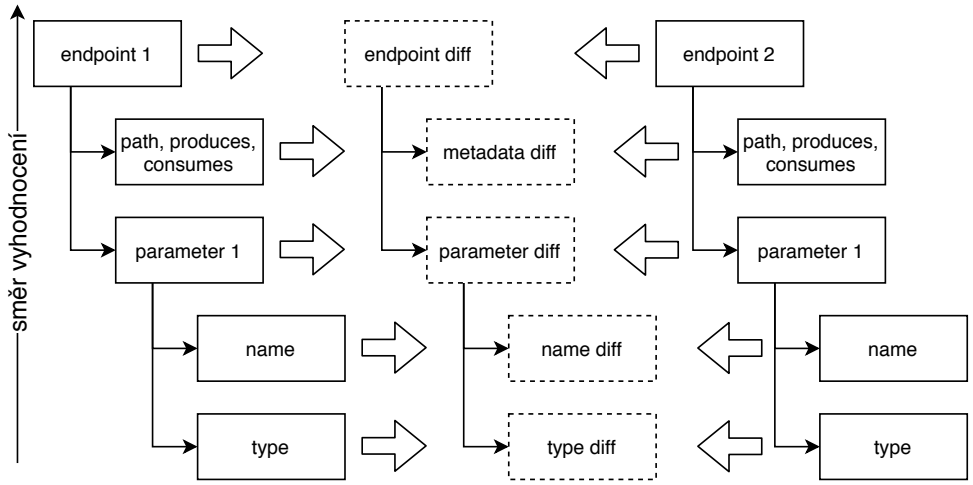
Algoritmus podporuje standardní verzovací formát `major.minor.micro`, který je vyjádřen regulárním výrazem: `/[vV][0-9]+(?:[.-][0-9]+){0,2}/`.

5.4 Vyhodnocení výsledků

Mezivýsledky porovnání jednotlivých elementů stromu metadat jsou ukládány do hierarchické struktury *Diff* a vždy po vyhodnocení všech rozdílů potomků dvou uzlů, dojde na základě těchto i k vyhodnocení rozdílů uzlů samotných. Úroveň rozdílu jež určuje finální kompatibilitu webových služeb

je dána úrovní kořenového *Diff*, který drží celou strukturu popisující detailní rozdíly mezi službami.

Příklad tvorby *Diff* při porovnání dvou operací je znázorněn na obrázku 5.4. Nejdříve dojde k vyhodnocení rozdílů listů, tedy atributů *name* a *type* parametrů obou operací, čímž vzniknou *type diff* a *name diff*. Na základě těchto se určí rozdíl mezi parametry samotnými (*parameter diff*) a po porovnání metadat operací (*metadata diff*) je vyhodnocen i výsledný rozdíl obou operací jež je reprezentován objektem *endpoint diff*.



Obrázek 5.4: Tvorba rozdílů mezi dvěma endpointy

Vyhodnocení úrovně rozdílu *Diff* na základě jeho potomků je řízeno prioritou. Každá z úrovní rozdílů *Difference* popsaných v tabulce 5.1 má určitou prioritu, která reprezentuje závažnost rozdílu. Uzel od svých potomků vždy přejímá *Difference* s nejvyšší prioritou, čímž je zaručeno, že se rozdíly narušující kompatibilitu projeví na finálním verdiktu.

Pokud tedy například dojde k rozdílu *UNK* (maximální priorita) při porovnání dvou atributů parametru endpointu, postupným vyhodnocováním se tato *Difference* dostane až ke kořenovému uzlu a výsledná kompatibilita bude mít hodnotu *UNK*. Hodnoty jsou v tabulce 5.1 seřazeny od nejnižší priority po nejvyšší.

5.4.1 Dopad na klienta

Výše popsané úrovně *Difference* mají rozdílný dopad na klienta. Ten se dá rozdělit do skupin bezpečné, potenciálně nebezpečné a nebezpečné, tak jak je tomu v tabulce 5.10. Následující odstavce popisují dopad na klienta pro jednotlivé úrovně a zdůvodňují jejich zařazení do dané skupiny.

Difference	Dopad na klienta
None (NON)	bezpečné
Specialization (SPE)	bezpečné
Insertion (INS)	bezpečné
Deletion (DEL)	potenciálně nebezpečné
Generalization (GEN)	potenciálně nebezpečné
Mutation (MUT)	nebezpečné
Unkown (UNK)	nebezpečné

Tabulka 5.10: Dopad jednotlivých úrovní rozdílu na klienta

Bezpečné rozdíly

Pokud je rozdíl mezi službami v bezpečné skupině, může klient transparentně volat obě služby, aniž by došlo k jakékoliv chybě v rámci kontraktu. Úroveň *SPE* sice označuje specializaci, nicméně změny *SPE* a *GEN* mohou nastat pouze v případě neshody datových typů parametrů, nebo odpovědi operace. V takovém případě je aplikován princip kontravariance [1] podle kterého platí, že $F'(a') <: F(a) <=> a <: a'$. *SPE* tedy znamená bezpečnou změnu *GEN* u parametrů, nebo odpovědi operace.

Změna *INS* nastává v případě, že druhá webová služba obsahuje operace, které nejsou v první definované. Vzhledem k tomu, že je klient nemohl volat a nemůže tak dojít k porušení kontraktu webové služby, je tato změna brána jako bezpečná.

Potenciálně nebezpečné rozdíly

Změny v této skupině mohou mít nebezpečný dopad na klienta ve smyslu volání operace webové služby s neplatným kontraktem, nicméně nejsou tak závažné jako změny nebezpečné. Posouzení reálného rizika změny provádí klient na základě vráceného objektu popisujícího kompatibilitu služeb.

Úroveň *DEL* označuje element definovaný v první službě, ale chybějící v druhé. Může se jednat například o *service*, nebo operaci služby. Úroveň *GEN* je získána, stejně jako *SPE*, na základě principu kontravariance a označuje změnu v datovém typu (parametru, odpovědi operace, ...), například z *long* na *int*.

Nebezpečné rozdíly

Změny v této skupině představují buď kombinace několika méně závažných změn (*MUT*), nebo neporovnatelnost obou webových služeb (*UNK*). Stejně

jako u předchozí skupiny, i zde může klient použít výsledky porovnání a sám rozhodnout o míře nekompatibility. Úroveň *MUT* může například nastat i kombinací rozdílů endpointů, které klient nepoužívá a druhá webová služba tak pro něj může být stále kompatibilní.

Úroveň *UNK* označuje buď neporovnatelnost metadat a v takovém případě nelze o kompatibilitě jednoznačně rozhodnout, nebo nerovnost metadat v případě, kde možným výsledkem je pouze 'stejné', nebo 'nestejné'.

6 Implementace rozšíření

Cílem mé práce bylo vytvořit rozšíření CRCE schopné automatické kontroly kompatibility indexovaných webových služeb. Tato kapitola obsahuje stručný popis implementace rozšíření, jež realizuje algoritmus popsaný v předešlé kapitole. Druhá část kapitoly obsahuje popis integrace rozšíření do systému CRCE.

6.1 Porovnávač

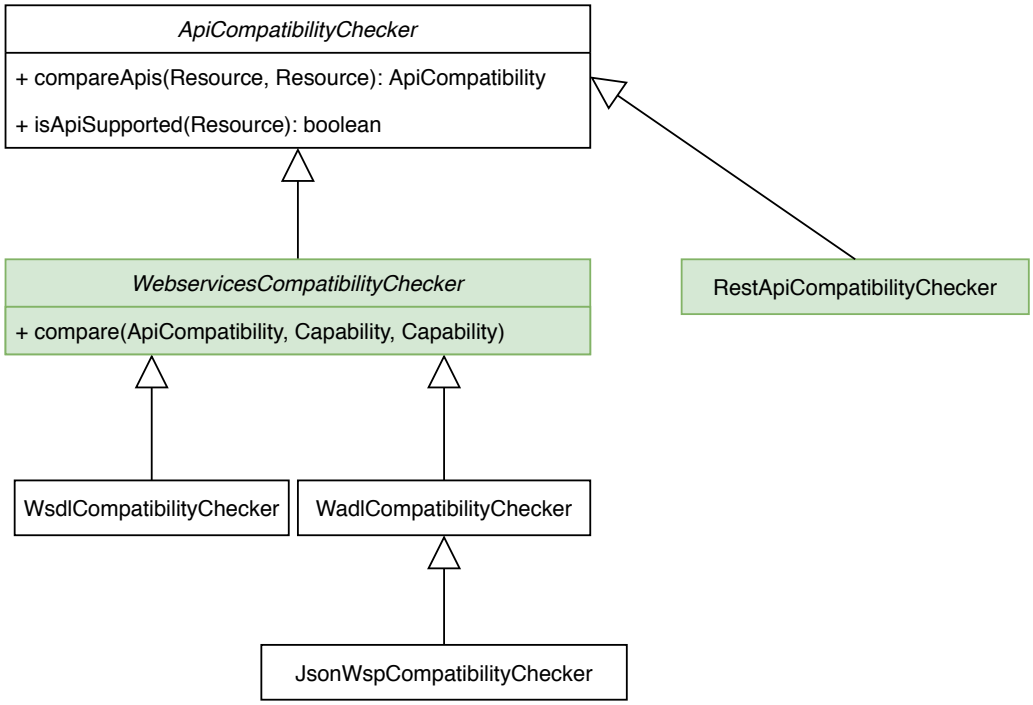
V kapitole 4 bylo zmíněna modulární architektura CRCE. Moduly představující nadstavbou jádra jsou podřazené modulu `crce-modules-reactor` a zde je umístěno i mé rozšíření. Samotný modul obsahující rozšíření je tvořen maven artefaktem a je implementovaný jako OSGi bundle, což je standardní postup v rámci CRCE.

Veřejná funkcionalita modulu je deklarovaná v rozhraní *ApiCompatibilityCheckerService*, jehož implementace je zpřístupněná formou OSGi služby (anotace *Component*). Zde jsou obsaženy všechny porovnávače webových služeb a implementace sama je schopná vybrat správný porovnávač pro zadanou dvojici *Resource*.

6.1.1 Struktura tříd porovnávačů

Diagram 6.1 znázorňuje strukturu tříd, jež slouží k porovnávání webových služeb. Každý porovnávač dědí od abstraktní třídy *ApiCompatibilityChecker*, která obsahuje deklaraci metody určené k porovnání služeb (*compareApis()*). Z důvodů zmíněných v sekci 5.2 je algoritmus implementován ve dvou verzích (na obrázku zeleně vyznačené třídy) podle původu metadat. První verze (na obrázku *RestApiCompatibilityChecker*) pracuje s metadaty získanými z implementace, druhá (na obrázku *WebservicesCompatibilityChecker*) pak s metadaty získanými z popisu webové služby.

I když je struktura metadat podobná, rozhodl jsem se vytvářet co možná nejméně společné implementace, protože bych tím vytvořil silnou vazbu mezi jinak nezávislými indexery. Ta by v případě změny struktury metadat u jednoho z indexerů přinášela obtíže při následné refaktorizaci porovnávacího modulu. Jediným společným předkem je tedy obecná abstraktní třída pro porovnávač webových služeb.



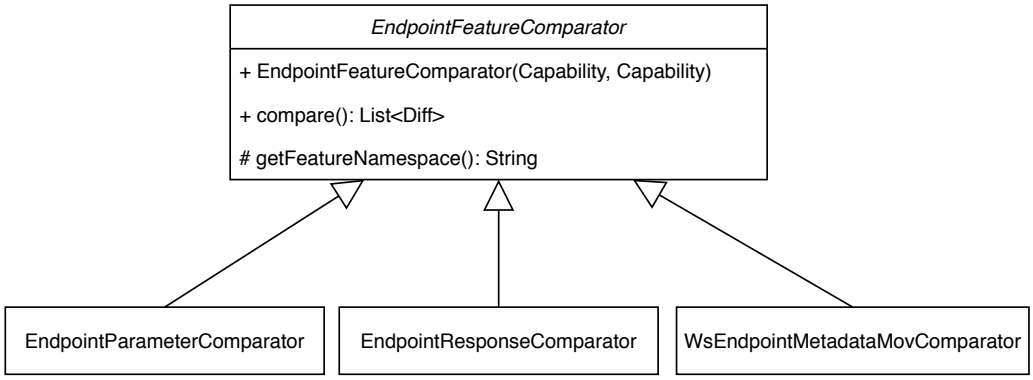
Obrázek 6.1: Diagram tříd popisující strukturu porovnávače

Protože indexer pracující s popisem je schopen indexovat více typů služeb, je pro každý z těchto typů vytvořen zvláštní porovnávač. Na obrázku to jsou třídy dědící od *WebservicesCompatibilityChecker*. Tato obsahuje porovnávací logiku a abstraktní metody pro získání konkrétních detailů jako například jmen některých atributů, identifikátorů *Capability*, nebo instancí porovnávačů metadat, které jsou volané během porovnání. Daná implementace pak tyto metody překrývá čímž poskytuje data specifická danému typu služby.

Porovnávání endpointů

Jak bylo zmíněno v kapitole 5.2, porovnávání endpointů služeb je rozděleno na několik částí. Z důvodů čistoty kódu a principu jedné odpovědnosti je logika porovnání každé části umístěna ve zvláštní třídě a pro každé jedno porovnání je vytvořena nová instance této třídy. Obrázek 6.2 obsahuje diagram těchto tříd společně s abstraktní třídou od které dědí každý z porovnávačů.

Abstraktní třída *EndpointFeatureComparator* obsahuje pouze extrakci daných vlastností endpointů z objektů *Capability* a abstraktní metodu *compare()*, kterou volá klientský kód. Tím je dosaženo snadné rozšiřitelnosti v případě nutnosti porovnání dalších vlastností endpointu, například pokud by se rozšířila struktura metadat.



Obrázek 6.2: Struktura porovnávače vlastností endpointů

Vestavěné datové typy

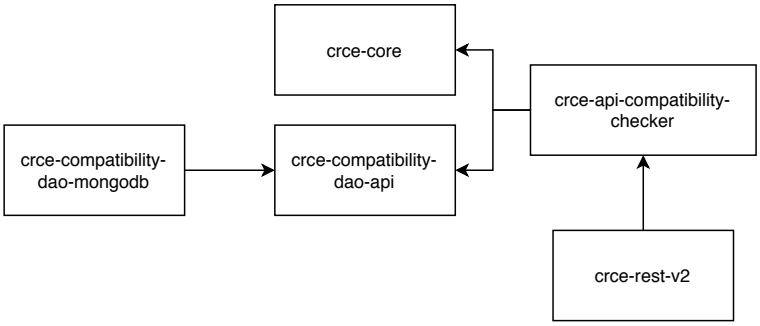
Porovnávání vestavěných datových typů je řešeno obalovací třídou, jejíž vstupním parametrem je jméno obalovaného datového typu. Tato třída obsahuje logiku která je schopná vyhodnotit, zda se doopravdy jedná o jméno vestavěného typu a také logiku porovnání dvou vestavěných typů. V současné době je toto porovnání omezena na rovnost a generalizaci (jak bylo uvedeno v sekci 5.2.2). Konkrétní obalové třídy jsou *JavaTypeWrapper* pro vestavěné typy Javy a *XsdDataType* pro vestavěné typy XML.

Obalové třídy nepokrývají všechny vestavěné typy, ale pouze jejich podmnožinu. Zejména se jedná o čísla (i s plovoucí desetinnou čárkou) a řetězce.

6.2 Integrace modulu

Modul ke své správné funkčnosti vyžaduje několik dalších komponent, tato závislost je znázorněna na obrázku 6.3. Konkrétně se jedná o moduly *crce-core*, který sdružuje všechny komponenty jádra (takže není nutné definovat závislosti jednotlivě) a *crce-compatibility-dao-api*, který obsahuje rozhraní pro přístup k datovému úložišti objektů *Compatibility*.

Výsledky porovnání webových služeb jsou ukládány do perzistentního úložiště a není tedy nutné znovu počítat rozdíly pro již porovnané dvojice. Ukládání těchto dat obstarává modul *crce-compatibility-dao-mongodb*, jež implementuje výše zmíněné rozhraní. V rámci mé práce jsem tyto moduly rozšířil o funkcionalitu pro získání objektu *Compatibility* podle zadané dvojice *Resource* a mazání existujících objektů *Compatibility*.



Obrázek 6.3: Interakce rozšíření se zbytkem CRCE

6.2.1 REST API

Na rozdíl od indexerů zmíněných v kapitole 4.2 není modul pro počítání kompatibility přímo napojen na žádnou z fází životního cyklu komponenty v CRCE a je potřeba jej manuálně spouštět. V současné době je rozšíření zpřístupněno skrze REST API implementovaném v modulu *crce-rest-v2*. Zde je skrze dependency injection zpřístupněná služba *ApiCompatibilityChecker-Service*, která na základě předaných vstupních parametrů provede porovnání a vrátí výsledek v podobě objektu *Compatibility*. Ten je pak vrácen klientovi.

Modul je dostupný na adrese `/apicomp/compare`. Parametry volání jsou uvedeny v tabulce 6.1. Nepovinný parametr *force*, jehož výchozí hodnota je *false* byl přidán z důvodů nutnosti přepočítání výsledků v případě změny implementace. Pokud tedy dojde například k opravě chyby díky které jsou v databázi uložena špatná data, není nutné je ručně mazat, ale stačí porovnání spustit znovu s nastaveným příznakem *force*.

Název	Popis	Povinný
id1	ID 1. *Resource* k porovnání	Ano
id2	ID 2. *Resource* k porovnání	Ano
force	Ignorovat existující *Comaptibility*	Ne

Tabulka 6.1: Parametry volání endpointu pro porovnání služeb

7 Ověření funkčnosti

Rozšíření do systému CRCE, jež bylo předmětem mé práce bylo testováno skrze jednotkové a integrační testy. Tato kapitola se postupně ve třech částech zabývá použitím modulu a ověřením jeho správné funkčnosti. V první části je rozebráno použití modulu od nahrání artefaktů do CRCE až po získání výsledků porovnání. Druhá část této kapitoly obsahuje popis integračních testů, včetně testovacích dat. Ve třetí části je uvedeno shrnutí provedených testů.

7.1 Sestavení a nasazení modulu

Jedním z hlavních bodů oborového projektu, který předcházel této diplomové práci bylo zprovoznění automatického sestavení a nasazení CRCE do vývojového prostředí. Toho jsem využil při vývoji porovnávacího modulu zejména k rychlému nasazení nových verzí, které pak bylo možné pohodlně otestovat. Sestavením CRCE na nezávislém stroji jsem zároveň ověřil, že mnou vytvořený kód netrpí chybami typu "funguje na mém stroji".

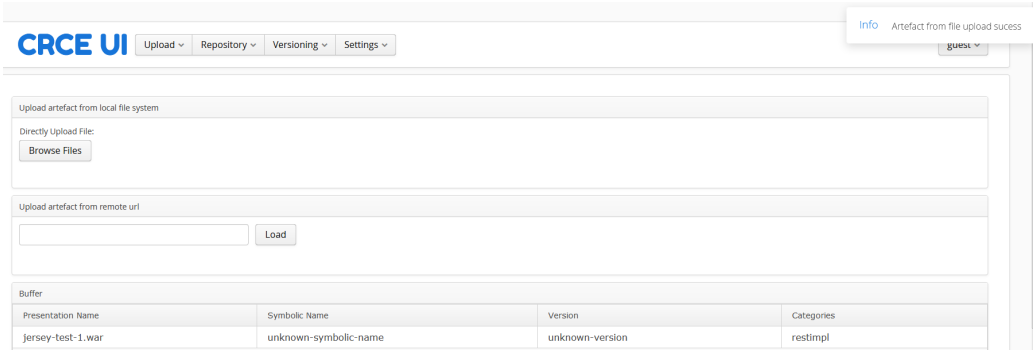
Kód CRCE je udržován na univerzitním Gitlabu, který kromě podpory verzování nástrojem GIT umožňuje také kontinuální integraci a nasazení pomocí Gitlab Pipelines[1]. Vstupem tohoto nástroje je deklarativní konfigurace jednotlivých kroků od překladu až po nasazení. Pipeline se automaticky spustí vždy při nahrání nového commitu do úložiště. Součástí nakonfigurované pipeline je také spuštění unit testů.

7.2 Použití modulu

Aby bylo možné modul použít, je nejprve potřeba nahrát testovací data. Tato sekce popisuje proces nahrání dat do CRCE a následného volání porovnávací služby. Pro jednoduchost je nahrán pouze jeden artefakt obsahující implementaci REST služby a následně je porovnán sám se sebou.

Nahrání artefaktu do CRCE Testovací artefakt *jersey-test-1.war* lze do CRCE nahrát skrze uživatelské rozhraní v záložce *File/url* v menu *Upload*, viz obrázek 7.1. Po dokončení nahrání se v pravém horním rohu zobrazí zpráva oznamující úspěch.

[1] https://docs.gitlab.com/ee/ci/pipelines/index.html

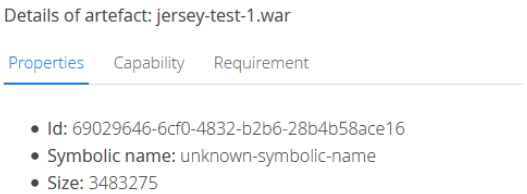


Obrázek 7.1: Nahrání artefaktu s implementací webové služby do CRCE

Nahrání artefaktu do Store V tuto chvíli se nahraná komponenta nachází ve fázi *Buffer*. Došlo tedy k sebrání metadat a indexaci artefaktu, nicméně pro porovnání je nutné provést operaci *commit* a tím komponentu posunout do fáze *Store*. To je lze udělat tlačítkem *Push to Store* v záložce *Plugins* pod menu *Repository* tak je znázorněno na obrázku 7.2. Tato záložka mimo jiné obsahuje přehled všech komponent nacházejících se ve fázi *Buffer* svého životního cyklu.

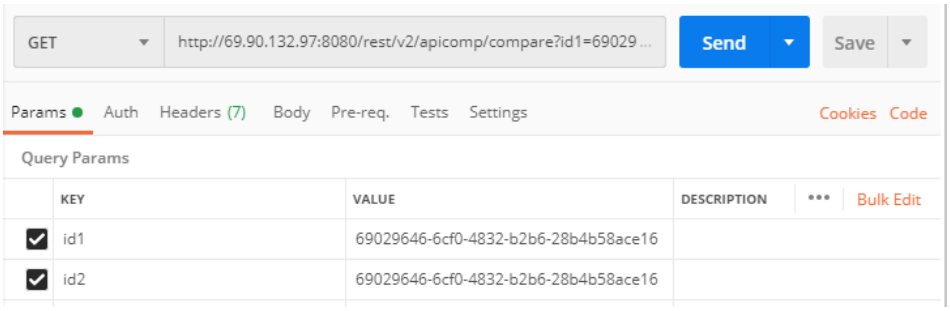
Obrázek 7.2: Nahrání artefaktu s implementací z Buffer do Store

Získání ID artefaktu K porovnání webových služeb je nutné znát ID *Resource*, jež reprezentují nahrané artefakty. To je možné zjistit označením daného artefaktu v uživatelském rozhraní a následným kliknutím na tlačítko *Detail*. Detail nahraného artefaktu je na obrázku 7.3.



Obrázek 7.3: Detail nahraného artefaktu

Volání porovnávací služby Identifikátor získaný v předchozím kroku lze použít k volání REST služby pro porovnávání. K tomu jsem pro jednoduchost použil nástroj Postman. Příklad volání služby s danými identifikátory je na obrázku 7.4.



Obrázek 7.4: Volání porovnávací služby

Výsledky porovnání Výsledek vrácený porovnávací službou je uveden na obrázku 7.5. Pro jednoduchost jsou některé části výsledného JSON dokumentu vypuštěny. Ve výsledku je obsažena finální úroveň rozdílu (*diffValue*), příznaky nastavené během porovnání (*additionalInfo*) a také strukturu *diffDetails* tvořenou stromem *Diff* detailně popisující rozdíly mezi službami.

```
{
    "id": "5eaad2db0cf23066dcf07704",
    "resourceName": "69029646-6cf0-4832-b2b6-28b4b58ace16",
    "baseResourceName": "69029646-6cf0-4832-b2b6-28b4b58ace16",
    "diffValue": "NON",
    "diffDetails": [...],
    "contract": "INTERACTION",
    "additionalInfo": {}
}
```

Obrázek 7.5: Volání porovnávací služby

Pokud by během porovnání došlo k nastavení příznaku MOV, byl by tento vrácený v *additionalInfo*.

7.3 Testování

K ověření funkčnosti porovnávacího modulu je připraveno několik testovacích scénářů. Každý z nich je zaměřen na jeden typ indexované služby a všechny s CRCE komunikují skrze REST API, tak jak bylo popsáno v předchozí sekci.

K testování jsem použil nástroj Postman, konkrétně jeho Collection Runner, který umožňuje vybrat šablonu volání REST API a soubor testovacích

dat ve formátu csv, jehož každá řádka představuje jeden test v podobě porovnávané dvojice služeb a očekávaného výsledku. Příklad těchto dat je uveden na obrázku 7.6.

```
id1,id2,res-diff,mov-flag
4047c969-5abe-441b-9d49-6c614bce13bb,4047c969-5abe-441b-9d49-6c614bce13bb,NON,0
4047c969-5abe-441b-9d49-6c614bce13bb,ecc04b3a-8018-4d78-87f8-5390a3c330b8,SPE,0
ecc04b3a-8018-4d78-87f8-5390a3c330b8,4047c969-5abe-441b-9d49-6c614bce13bb,GEN,0
ecc04b3a-8018-4d78-87f8-5390a3c330b8,ecc04b3a-8018-4d78-87f8-5390a3c330b8,NON,0
```

Obrázek 7.6: Příklad testovacích dat pro nástroj Postman

Princip každého z testovacích scénářů spočívá ve vytvoření několika verzí jedné služby a následném vzájemném porovnání. Tím lze otestovat jak vybrané změny tak i pořadí porovnávaných služeb. Jednotlivé scénáře včetně výsledků jsou detailně popsány v následujících sekcích.

7.3.1 REST služba založená na Jersey

V tomto scénáři jsou testovány dvě verze REST služby. První z nich (*v1*) obsahuje základní implementaci, druhá verze (*v2*) pak mění datový typ parametru u jednoho z endpointů z *Long* na *Number*.

Účelem tohoto testovacího scénáře je kromě ověření funkčnosti modulu pro REST služby také ověření správné aplikace principu kontravariance v případě změn datových typů parametrů endpointů. Změna datového typu z *Long* na *Number* je změnou *GEN*. Po použití kotrnavariance by změna na úrovni endpointu měla být *SPE* a protože je to jediný rozdíl mezi službami, měla by tato změna označovat také celkový rozdíl.

Výsledky vrácené voláním porovnávací služby jsou zobrazeny v tabulce 7.1. Výsledky odpovídají očekávání a princip kontravariance v případě změn *GEN* a *SPE* u datového typu parametru endpointu služby je tedy aplikován správně.

	v1	v2
v1	*NON*	*SPEC*
v2	*GEN*	*NON*

Tabulka 7.1: Výsledky porovnání metadat získaných z implementace

7.3.2 Služba popsaná JSON-WSP

Testovací scénář je zaměřen na služby popsané formátem JSON-WSP. Data pro testování tvoří čtyři verze uměle vytvořeného rozhraní založeného na

příkladu [2]. Odlišnosti mezi jednotlivými verzemi, označenými *v1* až *v4*, jsou rozepsány v tabulce 7.2.

Verze služby	Popis změny
v1	Základní verze
v2	Modifikace datového typu *User*
v3	Přidána operace *deleteUser*
v4	Operace *listGroups* změněna na *getUserInGroup*

Tabulka 7.2: Přehled změn ve verzích služby

Účelem tohoto scénáře je ověřit porovnávání vlastních datových typů a mutací mezi endpointy. Verze služby *v2* mění definici typu *User*, ale ne jeho název. Vzhledem k faktu, že definice typů nejsou indexovány, mělo by porovnání skončit výsledkem *NON*, protože názvy typů se shodují. Verze *v3* a *v4* přidávají endpointy, čímž mutují rozhraní služby. Očekávané výsledky pro tyto verze jsou *INS* pro *v3* (přidaný endpoint) a *MUT* pro *v4* (změna názvu operace).

Tabulka 7.3 obsahuje výsledky vrácené porovnávací službou, které odpovídají předpokladům uvedeným v předchozím odstavci.

	v1	v2	v3	v4
v1	*NON*	*NON*	*INS*	*MUT*
v2	*NON*	*NON*	*INS*	*MUT*
v3	*DEL*	*DEL*	*NON*	*MUT*
v4	*MUT*	*MUT*	*MUT*	*NON*

Tabulka 7.3: Výsledky porovnání metadat získaných z JSON-WSP popisu

7.3.3 Služba FuelEconomy

Tento scénář je zaměřen na porovnání REST služeb popsaných formátem WADL. Data pro testování tvoří popis služby Fuel Economy[3] a z něj vycházející tři verze. Popis verzí včetně změn je uveden v tabulce 7.4.

Scénář je zaměřen na mutaci operací a změny v URL na kterých jsou dostupné. Verze *v2* a *v3* testují mutaci, konkrétně přidávání a odebírání operací. Očekávané výsledky jsou *DEL* v případě *v2* a *MUT* V případě *v3*. Verze služby *v4*, kde jedinou změnou je URL a cesta k endpointu *fuelprices*, testuje správné nastavení příznaku *MOV* a očekávaný rozdíl je *NON*

[2] https://en.wikipedia.org/wiki/JSON-WSP

[3] https://www.fueleconomy.gov/ws/rest/application.wadl

Verze služby	Popis změny
v1	Základní verze
v2	Odebrán resource *labelvehicle*
v3	Odebrán resource *labelvehicle*, přidán resource *somethingDifferent*
v4	Změna URL a změna cesty k resource */fuelprices* na *fuel-prices*

Tabulka 7.4: Přehled změn ve verzích služby

	v1	v2	v3	v4
v1	*NON*	*DEL*	*MUT*	*NON,MOV*
v2	*INS*	*NON*	*INS*	*INS*
v3	*MUT*	*DEL*	*NON*	*MUT*
v4	*NON,MOV*	*DEL*	*MUT*	*NON*

Tabulka 7.5: Výsledky porovnání metadat získaných z WADL popisu

Tabulka 7.5 obsahuje výsledky vzájemného porovnání jednotlivých verzí služby. Pro přehlednost je v tabulce uveden příznak MOV pouze v případě, že byl vrácen. Pokud nastaven nebyl, je v tabulce uveden pouze rozdíl služeb.

7.3.4 Služba STAG-Číselníky

Testovací scénář je zaměřen na SOAP služby popsané formátem WSDL. Základem dat pro testování je popis služby STAG-Číselníky[4] který je rozšířen o operaci *testOperation*. Z této základní podoby jsou vytvořeny další čtyři verze. Celkem je tedy porovnáno pět verzí, jejichž rozdíly jsou popsané v tabulce 7.6.

Základní verze *v1* definuje operaci *testOperation*, která má nastavený parametr typovaný na *xs:int*. Porovnání verze *v1* s verzemi *v2* a *v5* je využito k testování principu kontravariance společně se změnami v URL. Verze *v3* mění definici typu *insertPracoviste*. Tato změna by neměla porovnání nijak ovlivnit, protože detaily vlastních datových typů nejsou indexovány. Verze *v4* mění název *service*, což by mělo být vyhodnoceno jako mutace.

Tabulka 7.7 obsahuje výsledky vzájemných porovnání všech verzí služby. Tyto výsledky, včetně nastavených příznaků MOV odpovídají předpokladům.

[4]https://stag-ws.zcu.cz/ws/services/soap/ciselniky?wsdl

Verze služby	Popis změny
v1	Základní verze s přidanou operací *testOperation*
v2	Změna hostname v url operací
v3	Změna typu *insertPracoviste*
v4	Změna názvu *service* z *CiselnikyServiceImplService* na *CiselnikyServiceImplServiceUpdate*
v5	Změna typu parametru operace *testOperation* z *xs:int* na *xs:long*

Tabulka 7.6: Přehled změn ve verzích služby

	v1	v2	v3	v4	v5
v1	*NON*	*NON,MOV*	*NON*	*MUT*	*SPE*
v2	*NON,MOV*	*NON*	*NON,MOV*	*MUT*	*SPE,MOV*
v3	*NON*	*NON,MOV*	*NON*	*MUT*	*SPE*
v4	*MUT*	*MUT*	*MUT*	*NON*	*MUT*
v5	*GEN*	*GEN,MOV*	*GEN*	*MUT*	*NON*

Tabulka 7.7: Výsledky porovnání metadat získaných z WSDL popisu

7.4 Shrnutí testů

Testovací scénáře popsané v předchozích sekcích ověřily základní funkčnost porovnávacího modulu na všech typech indexovaných služeb, včetně jeho dostupnosti skrze REST API. V případě testování webové služby STAG-Číselníky byla také ověřena kombinace změn URL a datového typu parametru operace.

Na přiloženém CD jsou k dispozici testovací data k jednotlivým scénářům, včetně exportovaných metadat pro jednotlivé verze služeb. Stejně tak jsou přiloženy konfigurace testů pro nástroj Postman.

8 Závěr

Literatura

[1] Abadi, M. – Cardelli, L. On Subtyping and Matching. In *European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*, 952, s. 145–167. ACM Press, January 1995. Dostupné z: https://www.microsoft.com/en-us/research/publication/on-subtyping-and-matching/.

[2] Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, July 1999. Dostupné z: https://tools.ietf.org/html/rfc2616.

[3] Berners-Lee, T. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, RFC Editor, January 2005. Dostupné z: https://tools.ietf.org/html/rfc3986.

[4] Berners-Lee, T. *Web Services - Program Integration across Application and Organization boundaries* [online]. aug 2009. [cit. 2020-05-01]. Dostupné z: https://www.w3.org/DesignIssues/WebServices.html.

[5] Brada, P. – Jezek, K. Repository and Meta-Data Design for Efficient Component Consistency Verification. *Science of Computer Programming*. 2015, 97, part 3, s. 349–365. ISSN 0167-6423. doi: 10.1016/j.scico.2014.06.013. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167642314002925.

[6] Brada, P. – Valenta, L. Practical Verification of Component Substitutability Using Subtype Relation. s. 38 – 45, 10 2006. doi: 10.1109/EUROMICRO.2006.50.

[7] Daigneau, R. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, From Objects to Web Services. Addison-Wesley, 2011.

[8] Daigneau, R. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, Web Service API Styles. Addison-Wesley, 2011.

[9] FIELDING, R. T. Architectural Styles and the Design of Network-based Software Architectures [online]. Disertační práce, University of California, Irvine, 2000 [cit. 2020-02-22]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[10] Fowler, M. *Richardson Maturity Model* [online]. [cit. 2020-05-01]. Dostupné z: https://www.martinfowler.com/articles/richardsonMaturityModel.html.

[11] HESSOVÁ, G. Automatické získání historických údajů z webových zdrojů [online]. Bakalářská práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2015 [cit. 2020-02-22]. Dostupné z: https://theses.cz/id/pzbgj7/.

[12] Mandel, L. *Describe REST Web services with WSDl 2.0* [online]. [cit. 2020-05-01]. Dostupné z: https://www.ibm.com/developerworks/webservices/library/ws-restwsdl/.

[13] Nelson, B. J. *Remote Procedure Call*. PhD thesis, Palo Alto Research Center, USA, 5 1981. Dostupné z: https://dl.acm.org/doi/book/10.5555/910306.

[14] PEJŘIMOVSKÝ, D. Vytváření a ukládání popisu webových služeb v úložišti CRCE [online]. Diplomová práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2015 [cit. 2020-02-22]. Dostupné z: https://theses.cz/id/bb74eq/.

[15] Richardson, L. *RESTful Web Services*. O'Reilly, 2007. ISBN 978-0-596-52926-0.

[16] W3C. *Web Services Architecture* [online]. feb 2004. [cit. 2020-05-01]. Dostupné z: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[17] W3C. *Web Application Description Language* [online]. aug 2009. [cit. 2020-05-01]. Dostupné z: https://www.w3.org/Submission/wadl/.

[18] W3C. *Web Services Description Language (WSDL) 1.1* [online]. mar 2001. [cit. 2020-05-01]. Dostupné z: https://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[19] W3C. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* [online]. jun 2007. [cit. 2020-05-01]. Dostupné z: https://www.w3.org/TR/wsdl/.

[20] Wikipedia. *JSON-WSP* [online]. april 2020. [cit. 2020-05-01]. Dostupné z: https://en.wikipedia.org/wiki/JSON-WSP.

[21] Wood, T. *How are REST APIs versioned?* [online]. sep 2014. [cit. 2020-05-01]. Dostupné z: http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/.

[22] Xinyang Feng – Jianjing Shen – Ying Fan. REST: An alternative to RPC for Web services architecture. In *2009 First International Conference on Future Information Networks*, s. 7–10, 2009.

Seznam zkratek

API	Application Programming Interface
CRCE	Component Repository supporting Compatibility Evaluation
HATEOAS	Hypertext As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
JSON-WSP	JSON Web-Service Protocol
OSGi	Open Services Gateway initiative
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WADL	Web Application Description Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

Příloha A - WSDL webové služby pro komix Dilbert

Tady bude WSDL