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

Diplomová práce

Využití úložiště komponent pro podporu aktualizace aplikací

Plzeň 2013 Jan Řezníček

Prohlášení

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

V Plzni dne 13. května 2013

Jan Řezníček

Abstract

Component repository support for application updates

The aim of my work is to extend the Component Repository supporting Compatibility Evaluation (CRCE). CRCE repository is focused on describing components with metadata. It is modularized using the technology of OSGi components, which makes it extensible and enables creation of modules with additional functionality. The goal of this master thesis is to create a CRCE API for interaction with its clients.

The theoretical part contains information about Component-based software engineering, an overview of Java component repositories, a chapter about CRCE and the analysis of possibilities for the realisation of client-server communication between the repository and its clients. It was chosen to create REST API, so the last theoretical chapter places an emphasis on the REST software architecture style.

The practical part contains a chapter about the CRCE REST module that was created as part of this thesis. The chapter contains the aim of the module, usage scenarios, REST API design, module architecture and implementation. The last chapter is about the module that integrates tools for compatibility evaluation to CRCE.

Obsah

1 Úvod

Na Katedře informatiky ZČU již delší dobu probíhá vývoj úložiště komponent s názvem Component Repository supporting Compatibility Evaluation (CRCE). Úložiště slouží primárně k ukládání OSGi komponent, neomezuje se ale jen na ně, obecně do něj lze vložit jakýkoliv artefakt. Úložiště klade důraz na možnost popisu uložených artefaktů pomocí metadat a vlastní rozšiřitelnost prostřednictvím modulů. Rozšiřitelnost umožňuje integrování nástrojů pro kontrolu kompatibility komponent, generování komponent a dalších modulů.

Posláním této diplomové práce je rozšíření úložiště CRCE. Hlavním úkolem je umožnit úložišti interakci s klientskými aplikacemi, které potřebují využít jeho možnosti poskytování informací o kompatibilitě komponent. Pro realizaci interakce bylo vybráno použití REST API, což je moderní technologie umožňující snadný přístup k úložišti pomocí protokolu HTTP. Modul klientovi umožní získávat informace o komponentách v úložišti, distribuční balíky vlastních komponent a používat další funkce, které klientovi usnadní aktualizaci jeho komponent.

Dalším úkolem bylo začlenit do úložiště CRCE již existující nástroje, které umožňují kontrolu kompatibility komponent a automatické rozhodování o verzi komponent na základě rozsahu změny oproti předchozí verzi komponenty.

Členění této diplomové práce do kapitol bylo vytvořeno podle zadání. Kapitoly 2 - 5 jsou teoretické. Druhá kapitola slouží k představení komponentově orientovaného programování a OSGi. Třetí kapitola se věnuje úložištím softwarových komponent pro programovací jazyk Java. Na třetí kapitolu navazuje kapitola čtvrtá, která popisuje úložiště CRCE. Pátá kapitola rozebírá možnosti komunikace mezi klientem a serverem. Největší její část se zaměřuje na styl softwarové architektury REST.

Následují dvě praktické kapitoly o vytvoření dvou modulů do CRCE. Kapitola 6 popisuje modul poskytující úložišti CRCE REST API. Kapitola obsahuje představení účelu API, předpokládané scénáře použití i návrh jednotlivých REST API metod. Ve své druhé polovině kapitola zahrnuje informace o architektuře modulu a jeho implementaci. Sedmá kapitola se věnuje vytváření informací o kompatibilitě komponent.

2 Komponentově orientované programování

Program je podle komponentově orientovaného programování [8, 32] složený z komponent, což jsou nezávislé funkční jednotky. Základním kamenem tohoto systému je tedy softwarová komponenta.

"Softwarová komponenta je jednotka určená pro skládání do větších celků se specifikovanými rozhraními a pouze vnějším kontextem závislostí. Softwarová komponenta může být rozmístěna nezávisle a může být využita třetí stanou." [29]

Tato definice komponenty byla zformulována na Evropské konferenci o objektově orientovaném programování v roce 1996. Komponenta by měla mít svoji vnitřní strukturu od vnějšího světa pevně oddělenou určitým rozhraním, které určuje, co komponenta od okolí požaduje a jakou funkcionalitu poskytuje. Tento model se nazývá černá skříňka (black-box). Z vnějšího pohledu by neměla být vidět implementace komponenty. Měl by být znám pouze její popis a hlavně její rozhraní, pomocí kterého probíhá veškerá interakce komponenty s okolím.

Velkou výhodou komponentově orientovaného programování je znovupoužitelnost komponent, které by měly být tvořeny univerzálně. Znovupoužitelnost přináší další odvozené výhody. Častěji používaná komponenta je vícekrát testována, dá se tedy předpokládat, že je při tom odhaleno více chyb. Testování jednoho systému, který komponentu obsahuje, může pomoci i ostatním systémům, které ji používají. Často používanou komponentu se také vyplatí více optimalizovat, což zvyšuje kvalitu celého programu.

Existují samozřejmě i nevýhody, které se projevují ve vlastních nárocích komponenty na její tvorbu a běh. Tvorba komponenty je náročnější než tvorba odpovídajícího jednoúčelového kódu, protože musí být, s ohledem na znovupoužitelnost, psaná obecněji. K nákladům na vývoj komponent je nutné přičíst i náklady na navrhnutí komponentového modelu, pokud není použitý některý stávající. I to ale vyžaduje prostředky, zejména čas na seznámení programátorů s tímto modelem.

2.1 Komponentový model

Pokud bychom komponenty a jejich rozhraní tvořili zcela nahodile, jen těžko bychom je následně skládali do jednoho funkčního celku. Je třeba sada pravidel, která určuje, co to komponenta je a jak ji vytvořit. Tato sada pravidel se nazývá **komponentový model** [15]. Komponentový model definuje, jaká jsou pravidla pro tvorbu rozhraní komponent, jak spolu komponenty komunikují, jak se skládají, jaké mohou vyžadovat služby, atd.

Komponentový framework [15, 8] je implementací komponentového modelu. Je tvořen službami, jejichž hlavním úkolem je zajišťovat běh a komunikaci komponent. K jednomu modelu může být vytvořeno více frameworků a framework může dokonce implementovat více modelů, pokud mají slučitelné vlastnosti.

Některé příklady komponentových modelů či frameworků:

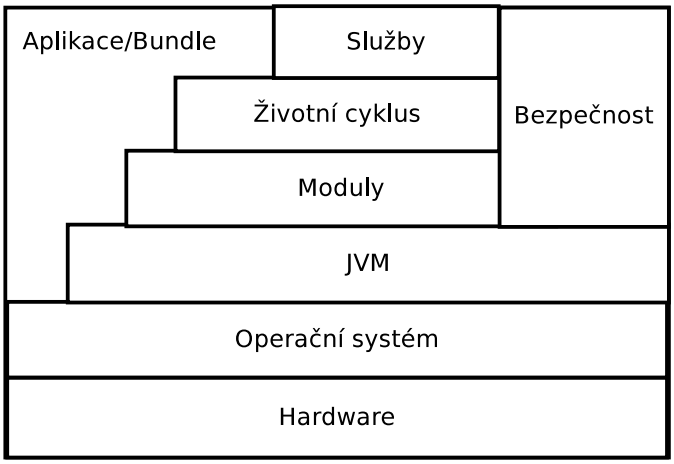
- **Enterprise JavaBeans (EJB)** - Komponentová architektura pro tvorbu modulárních systémů. EJB Java API je součástí platformy Java EE.
- **Component Object Model (COM)** - COM je standard, definující binární rozhraní pro softwarové komponenty. Je produktem firmy Microsoft. Umožňuje interakci komponent v různých programovacích jazycích. Více o jeho rozšíření pro komunikaci softwarových komponent DCOM najdete v sekci 5.1.2 na straně 26.
- **CORBA Component Model (CCM)** - Komponentový model umožňující interakci komponent za využití systému CORBA. Více o systému CORBA najdete v sekci 5.1.1 na straně 24.
- **Universal Network Objects (UNO)** - Příklad specializovaného komponentového modelu, který využívají aplikační balíky *OpenOffice.org* a *LibreOffice*. Umožňuje tvorbu rozšíření pro tyto aplikace v různých programovacích jazycích.
- **OSGi framework** - Je specifikací modulárního systému v programovacím jazyce Java. Je popsán v sekci 2.1.1 na straně 4.

2.1.1 OSGi framework

OSGI [20, 8] je specifikací modulárního systému v programovacím jazyce Java. OSGi je vyvíjeno sdružením OSGi Alliance, které vzniklo v roce 1999. Účelem OSGi je přidat do Javy dynamický komponentový model, který umožní aplikaci instalovat, startovat a zastavovat jednotlivé moduly za běhu.

Základní komponenty se v OSGi nazývají **bundle**. Z pohledu Javy jde o běžné JAR soubory. Bundle obsahuje manifest, který definuje základní parametry komponenty. Zatímco v čisté Javě je veškerý obsah JAR souboru dostupný uživatelům, bundle považuje obsah za privátní a na povrch vystavuje pouze to, co si v manifestu definuje jako veřejný obsah.

Architektura OSGi je zobrazena na obrázku 4.1. Framework se dělí na následující vrstvy:



Obrázek 2.1: OSGi - architektura frameworku (a vrstev pod ním)

- Aplikace/bundle - Vlastní aplikace se skládá z komponent (bundle).
- Services (Služby) - Vrstva služeb zajišťuje interakci mezi komponentami. Této vrstvě se věnuje podsekce Služby umístěná na straně 6.
- Life-Cycle (Životní cyklus) - API umožnující životní cyklus komponent. Tato vrstva je podrobněji probrána v podsekci Životní cyklus na straně 5.

- Modules (Moduly) - Vrstva definující, jak mohou komponenty importovat a exportovat kód. Více o této vrstvě najdete v podsekci Moduly níže.
- Security (Bezpečnost) - Vrstva zajišťující bezpečnostní aspekty.

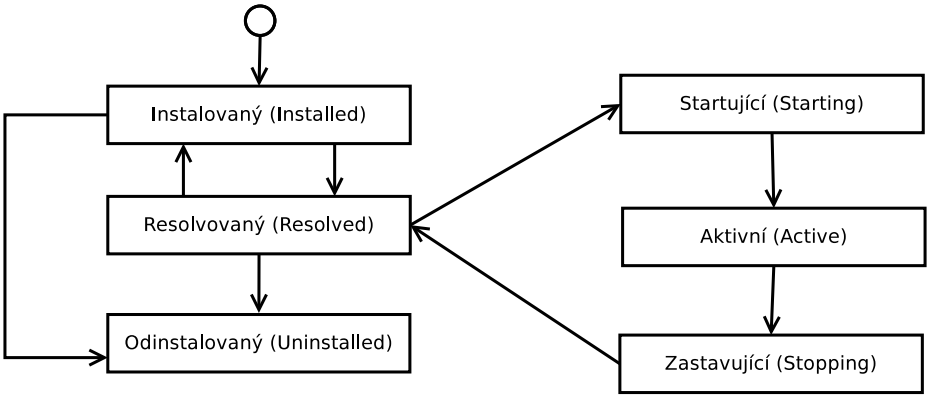
Moduly

Vrstva modulů (Module layer) definuje komponenty OSGi (bundle) a jejich možnosti zveřejňovat část svého kódu. Každý bundle může deklarovat, které jeho balíčky (package) budou viditelné navenek. Tyto balíčky jsou označovány pojmem *exportované balíčky* (v manifestu je najdete jako *Export-Package*). Je to vlastně rozšíření standardního přístupového mechanizmu Javy (modifikátory *public*, *protected*, *private*).

Druhou stanou tohoto mechanizmu je schopnost komponenty (bundle) deklarovat, na jakých exportovaných balíčcích závisí. Tyto balíčky se nazývají importované balíčky (v manifestu *Import-Package*).

OSGi poté při procesu vyhodnocování závislostí může ještě před spuštěním určit, zda jsou závislosti splněny. To výrazně omezuje problémy, kdy některá potřebná třída není nalezena (*ClassNotFoundException*).

Životní cyklus



Obrázek 2.2: OSGi - životní cyklus komponent(bundle)

Vrstva životního cyklu určuje, jak jsou komponenty dynamicky instalovány

a řízeny OSGi frameworkem. Životní cyklus komponent je zobrazen na obrázku 2.2. Komponenta prochází následujícími stavy:

- Instalovaný (Installed) - Pokud je komponenta v tomto stavu, byl již vykonán krok instalace. Nebyla ještě provedena analýza závislostí ani načítání tříd. Byly provedeny pouze základní kroky, jako je určení vlastností komponenty z jejího manifestu.
- Vyřešený (Resolved) - Stav, kdy již byly vyřešeny závislosti komponent.
- Startující (Starting) - Bundle je v tomto stavu, pokud byla spuštěna metoda *start* jeho aktivátoru a její činnost ještě neskončila .
- Aktivní (Active) - Bundle byl úspěšně nastartován a funguje ve frameworku.
- Zastavující (Stopping) - Bundle je v tomto stavu, pokud byla spuštěna metoda *stop* jeho aktivátoru a její činnost ještě neskončila.
- Odinstalován (Uninstalled) - Bundle byl odstraněn ze systému. Z tohoto stavu není návratu. Pokud bychom chtěli komponentu znovu použít, musíme ji znovu nainstalovat.

BundleActivator (aktivátor) je rozhraní, které umožňuje tvůrci komponenty určit akce, které proběhnou při startu a zastavení komponenty. Jeho poloha je specifikována v manifestu, pro každou komponentu může existovat pouze jeden. Je jednou ze základních možností, jak v OSGi spustit kód komponentu.

Služby

Princip služeb OSGi odpovídá konceptu servisně orientované architektury (SOA - zmíněna v úvodu kapitoly 5). Je třeba říci, že OSGi používá tento koncept déle, než se SOA stala populárním a často používaným termínem. Základem tohoto konceptu jsou vztahy mezi poskytovatelem, registrem a uživatelem služeb. Poskytovatel si může službu zaregistrovat v registru. Ostatní komponenty poté mohou získat seznam registrovaných služeb a vybrat si službu k použití. Zatímco dnes je pojem SOA většinou používán v souvislosti s webovými službami, v OSGi jsou služby lokální na jedné JVM.

Tato vrstva podporuje praktiku oddělení implementace služby od jejího rozhraní. OSGi služby jsou tedy v praxi Java rozhraními. Služby podporují flexibilitu komponent, mohou se objevovat a mizet během životního cyklu komponent.

Implementace OSGi

OSGi je standard, který naplňuje několik implementací:

- Apache Felix - implementace standardu OSGi od Apache Software Foundation. Poměrně malá implementace, kterou lze snadno použít.
- Equinox - vyvíjena týmem Eclipse, s čímž souvisí snadná integrace s tímto IDE. Pravděpodobně nejrozšířenější implementace.
- Knopflerfish - implementace vyvíjena Makewave.
- Concierge - velmi kompaktní implementace. Vhodná pro běh na mobilních zařízeních a vestavěných systémech.

2.2 Vývoj na ZČU

V oblasti komponentového programování probíhá na katedře informatiky a výpočetní techniky Západočeské university poměrně rozsáhlý výzkum a vývoj. Vzhledem k tomu, že tato práce se přímo či nepřímo několika těchto projektů dotýká, projdu zde některé tyto projekty, které jsou vyvíjeny na ZČU.

CoSi (Components Simplified) [11] - Komponentový model, jehož cílem je plně naplňovat základní principy komponentově orientovaného programování. Důraz je kladen na dodržení black-boxu. Podporuje komponenty ve stylu OSGi.

OBCC (OSGi Bundle Compatibility Checking) [13] - Sada nástrojů, která se zabývá nahraditelností komponent a kontrolou jejich kompatibility.

Základem je OSGi Bundle Compatibility Checker (nástroj pro kontrolu kompatibility), který zkoumá, zda je bezpečně možné nahradit komponentu jinou (novější verzí). Při nahrazování by se mohlo stát, že poskytovatel některé služby změní svoje rozhraní, na což nebude připraven uživatel služby. Tím by se systém mohl dostat do nefunkčního stavu.

Navazujícím nástrojem je OSGi Version Generator, který umožňuje automatické určení verze komponenty. Nástroj zjistí rozsah změn oproti předchozí verzi (zda se měnilo některé vnější rozhraní, nebo jsou změny jen vnitřní) a podle toho určí nové číslo verze.

CRCE (Component Repository supporting Compatibility Evaluation) [12] - Úložiště komponent, kterému je věnována sekce 4 na straně 14.

EFFCC (Extra Functional Property Featured Compatibility Checks) [21] - Je množina modulů, které umožňují obohatit existující komponentové frameworky založené na programovacím jazyce Java o mimo-funkční charakteristiky a jejich kontrolu kompatibility.

Integrace projektů - všechny zde zmíněné projekty do jisté míry souvisí s úložištěm CRCE. Do tohoto úložiště je možné vkládat komponenty CoSi. Pluginy úložiště využívají OBCC (více v sekci 7 na straně 69). Úložiště také podporuje komponenty s mimo-funkčními charakteristikami (EFFCC).

3 Ukládání softwarových komponent

Tato kapitola nabízí přehled možností pro ukládání softwarových komponent, tedy softwarových úložišť. Omezuje se pouze na úložiště komponent v programovacím jazyce Java. Podrobně je zde popsáno úložiště Apache Mavenu a úložiště OBR, které je definované v OSGi.

3.1 Apache Maven

Apache Maven [1, 23] je nástroj pro správu softwarových projektů. Mezi základní cíle Mavenu patří:

- zjednodušit proces sestavování (build) projektu - Přináší možnost sestavit projekt bez nutnosti znát detaily o vlastním průběhu sestavování.
- poskytnout jednotný sestavovací systém - Pokud se někdo seznámí se sestavováním jednoho Maven projektu, měl by se snadno v jakémkoliv jiném Maven projektu.
- poskytovat kvalitní informace o projektu
- vést ke správným programovacím praktikám
- umožnit snadné přijímání nových vlastností - Maven může být rozšiřován pomocí pluginů.

U **Mavenu** se nabízí srovnání se starším systémem pro sestavování projektů, kterým je **Ant**. Ant je procedurální povahy, určuje tedy postup, pomocí kterého se má projekt sestavit či spustit. Maven je oproti tomu deklarativní. Uživatele odstiňuje od přesného postupu a sám provede požadovanou akci, kterou může být např. kompilace zdrojových kódů či vytvoření JAR souboru.

K definici a popisu projektu Maven používá **POM (Project Object Model)**. Tento model se zapisuje do XML souboru (*pom.xml*) a lze si v něm definovat veškeré potřebné informace o projektu. Mezi základní informace, které obsahuje tento soubor patří identifikace projektu (groupId, artefactId, version), závislosti na ostatních knihovnách a seznam použitých Maven pluginů a jejich konfigurace. Minimální soubor *pom.xml*, obsahující pouze nutné informace, vypadá následovně:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

3.1.1 Maven úložiště

Správa závislostí na knihovnách je jednou z nejdůležitějších funkcí Mavenu při sestavovacím procesu. Při běžném sestavování projektu pomocí IDE či jiného nástroje se knihovny obvykle přidávají do některé složky projektu. Pokud více projektů používá stejnou knihovnu, je v adresáři každého z nich. Maven oproti tomu všechny knihovny či sestavené projekty ukládá do vlastního úložiště. Všechny objekty v tomto úložišti se nazývají artefakty (artefact). V tomto úložišti jsou artefakty dostupné všem projektům a každý artefakt v něm existuje pouze jednou.

Lokální úložiště je výše zmíněné úložiště, kam jsou na jednom počítači ukládány veškeré artefakty. Aby se nemusel uživatel o toto úložiště ručně starat, artefakty jsou do něj automaticky stahovány ze vzdáleného úložiště. Pokud je potřeba některý artefakt, Maven nejprve zkontroluje, zda je dostupný v lokálním úložišti. Pokud tomu tak není, Maven prohledá vzdálená úložiště (jejichž lokace lze určit v POM souboru projektu) a artefakt stáhne do lokálního úložiště.

Vzdálená úložiště jsou umístěna na jiném počítači v síti. Jedním ze vzdálených úložišť je centrální repositář (*http://repo.maven.apache.org/*), ve kterém jsou artefakty představující nejznámější programy a pluginy pro Maven. Adresu centrálního repositáře není třeba v POMu definovat.

Struktura Maven úložiště

Jak lokální úložiště, tak vzdálené úložiště mají stejnou strukturu. Ta vychází z identifikátorů maven artefaktu:

- **groupId** - název pracovní skupiny nebo organizace

- **artefactId** - název artefaktu
- **version** - verze artefaktu

Konkrétní artefakt lze najít v úložišti na následující lokaci:

```
/$groupId[0]/../$groupId[n]/$artifactId/$version/$artifactId-$version.$extension
```

Kde *groupId*[] je pole řetězců, které vzniklo rozdělením názvu skupiny podle teček. *Extension* je přípona artefaktu.

3.2 OBR

OBR [3, 2, 23] je zkratkou zastupující výraz OSGi Bundle Repository. Jde o návrh specifikace od OSGi aliance pro službu, která umožní přístup do úložišť OSGi komponent. Každé takové úložiště musí nabízet základní funkce, jako je poskytnutí seznamu dostupných komponent ke stažení s informacemi o nich. Původní návrh OBR vznikl ve formě OSGi RFC-0112[3].

Logický model OBR je založen na artefaktu, jehož reprezentace je nazvaná *resource*. Resource mají své schopnosti (*capability*) a mohou také požadovat určité schopnosti od svého okolí. Schopnost je pojmenovaná množina vlastností (*property*) a její využití je tedy značné rozsáhlé. Prakticky každá vlastnost artefaktu se dá uložit jako schopnost. Požadavek artefaktu na schopnost (*requirement*) je pojmenovaný filtr, který určuje, zda určitá schopnost splňuje žádaný požadavek. Pro formát filtru jsou využity OSGi filtry, což jsou LDAP filtry. Filtr určuje hodnotu vlastností (*property*), které požadovaná schopnost musí obsahovat. LDAP filtry umožňují výrokovou logiku, pomocí níž lze skládat dohromady složitější požadavky.

3.2.1 OSGi 5 Repository Service

V OSGi verze 5 [2] byla Repository Service přidána do oficiální specifikace. Návrh z RFC-0112 je zde trochu změněn a stal se oficiální součástí OSGi specifikace. Tato specifikace definuje API pro přístup do repositáře, který může být vzdálený. Specifikace Repository Service úzce souvisí se specifikací

Resolver service, která je určena pro vyhodnocování závislostí komponent. Specifikace klade důraz na vyhledávání komponent a možnost přiřazení metadat ke komponentám.

Entity

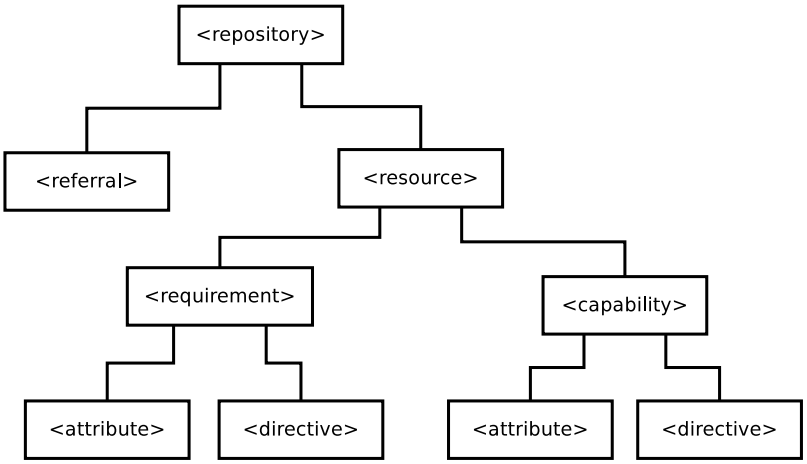
Specifikace definuje následující entity:

- Repository - fasáda pro vzdálený přístup k množině artefaktů (resource)
- Resource - artefakt
- Capability - schopnost artefaktu
- Requirement - požadavek artefaktu na schopnosti, které mu mají být poskytnuty
- Resource Content - poskytuje přístup k binární formě artefaktu ve standardním formátu (Input Stream na vlastní artefakt)

XML formát

Nepovinnou částí specifikace je návrh XML formátu pro metadata úložiště. Tento formát může být například využit pro výměnu dat mezi úložišti. Struktura XML je vidět na obrázku 3.1. Elementy tohoto XML formátu mají následující významy:

- repository - kořenový element pro úložiště
- referral - odkaz na jiný repositář. Podporuje vícenásobné repositáře se stromovou strukturou.
- resource - artefakt v úložišti
- capability - schopnost artefaktu
- requirement - požadavek artefaktu na schopnosti, které mu mají být poskytnuty
- attribute - vlastnost schopnosti nebo požadavku. Tvoří ji primitivní elementy *name*, *value* a *type*.



Obrázek 3.1: OSGi Repository Service - struktura XML

- directive - pokyn pro *Resolver*

4 Úložiště CRCE

Úložiště CRCE (Component Repository supporting Compatibility Evaluation) [12, 23] je obecné úložiště komponent. Přestože je zaměřené na OSGi komponenty, obecně do něj lze vkládat jakékoliv artefakty (různé druhy komponent, obrázky, atd). Jak již jeho název napovídá, CRCE je zaměřeno na možnost ověřovaní kompatibility komponent v úložišti. Zatímco ostatní úložiště se zaměřují na ukládání komponent a ověřování kompatibility nechávají na straně klienta, CRCE přenáší tyto výpočetně náročné operace na úložiště.

4.1 Poslání úložiště

Hlavním posláním úložiště je možnost skladování komponent se zabudovanou kontrolou jejich kompatibility. Při práci s obecným úložištěm je třeba při požadavku klienta o komponentu (například při updatu určité komponenty na straně klienta) vykonat následující kroky:

1. Klient si vybere komponentu z úložiště
2. Klient si ověří, zda komponenta vyhovuje aplikaci (plní všechny závislosti, atd..)
3. V případě že ověření selže, klient zkusí jinou komponentu

Úložiště CRCE přenáší proces kontroly kompatibility na stranu úložiště. Při vložení komponenty je pomocí nástroje OBCC vyhodnocena kompatibilita s ostatními komponentami téhož jména. Klient následně pomocí REST API získá informace o kompatibilitě a může se rozhodnout, zda spustí aktualizaci.

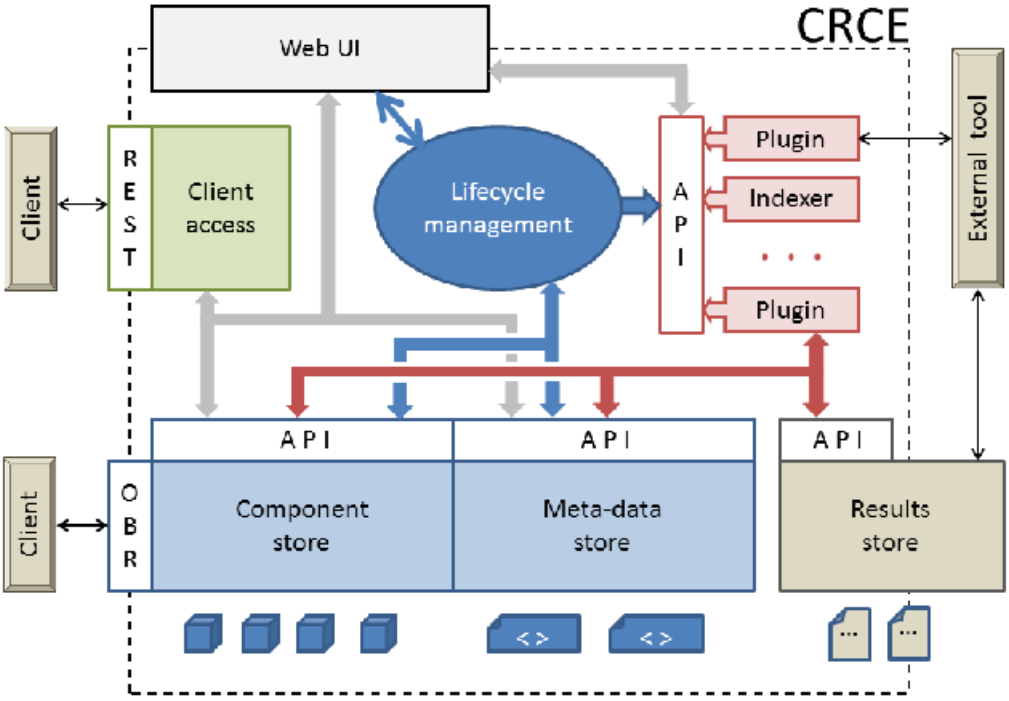
Základními vlastnostmi CRCE jsou:

- Rozšiřitelnost - Rozšiřitelnost umožňuje rozšiřovat funkčnost úložiště pomocí pluginů.
- Popisná metadata - Umožňují popsání komponent, jejich závislostí, testů kompatibility, atd..

Uživatelům úložiště poskytuje následující základní funkce:

- vložení artefaktu do úložiště
- získání seznamu artefaktů z úložiště
- stažení artefaktu z úložiště
- získání metadat artefaktu z úložiště
- zpracování artefaktu během jeho životního cyklu v úložišti (vyhodnocení kompatibility, testy, ...)

4.2 Architektura



Obrázek 4.1: CRCE - architektura úložiště (Zdroj: CRCE prezentace pro EuroMicro, Přemek Brada)

CRCE je modulárním projektem s podporou vytváření nových modulů. Základní oblasti mají své API definované v oddělených modulech:

- Metadata API - definice entit, jejichž základní struktura je převzata z OBR
- Plugin API - podpora rozšiřitelnosti
- Repository API - samotné úložiště
- Result API - výsledky testů
- Metadata DAO API - načítání a ukládání metadat

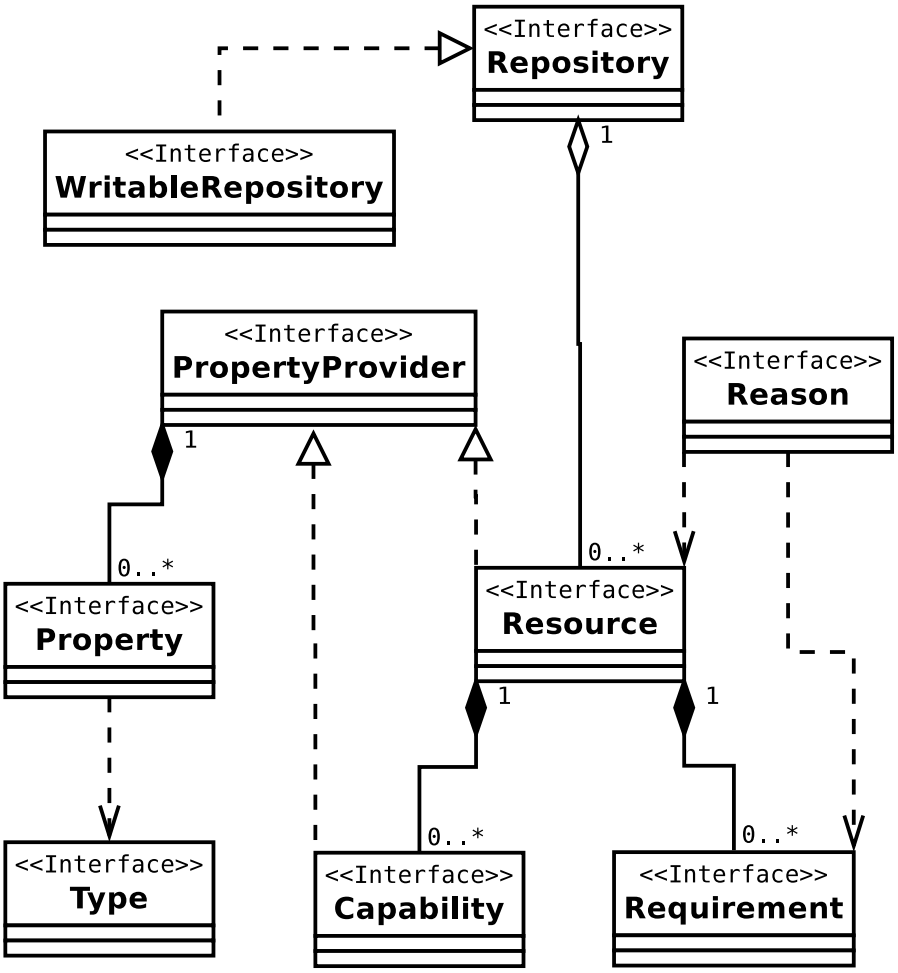
Pro jednotlivá API existují implementace, které jsou umístěny v samostatných modulech. Některé základní moduly jako jsou Metadata API, Plugin API, Repository API a Web UI budou v následujících sekcích popsány podrobněji.

4.2.1 Metadata API

Tento modul definuje rozhraní, které reprezentují entity metadat OBR. Základními rozhraními jsou:

- Repository - reprezentuje úložiště artefaktů
- Resource - reprezentuje artefakt a jeho popisná metadata
- Capability - reprezentuje schopnost artefaktu
- Requirement - reprezentuje požadavek na schopnost artefaktu
- PropertyProvider - reprezentuje entity, které mohou mít vlastnosti (Property)
- Property - reprezentuje vlastnost určité entity
- Type - určuje typ vlastnosti (Property)
- Reason - páruje Requirement a Resource. Indikuje důvod, proč je Resource vybraná Resolverem
- Resolver - vyhodnocuje závislosti mezi přidanými Resource.

UML diagram rozhraní a jejich vztahů je zobrazen na obrázku 4.2.



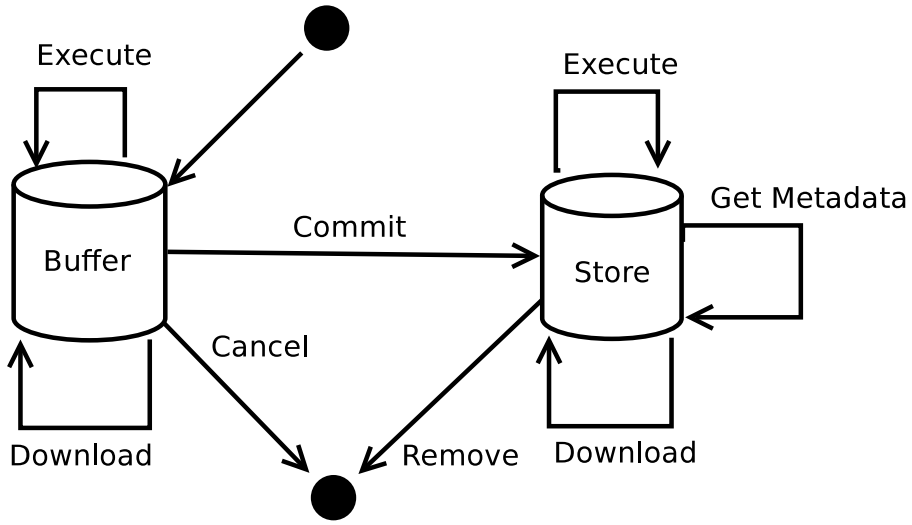
Obrázek 4.2: CRCE - UML diagram rozhraní z Metadata API

4.2.2 Plugin API

Plugin API umožňuje tvorbu nových modulů a správu existujících modulů. Základem je rozhraní **Plugin**, které musí každý modul implementovat. Tvůrce modulu může rovněž využít abstraktní třídu **AbstractPlugin**, která implementuje metody rozhraní *Plugin*, aby vraceli výchozí hodnoty. Další důležité rozhraní je **PluginManager**, což je správce pluginů, který umožňuje získávat všechny nainstalované pluginy (moduly). Tvorba nových modulů je podrobněji popsána v sekci 4.4 na straně 20.

4.2.3 Repository API

Repository API definuje dvě základní úložiště, kterými jsou **Buffer** a **Store**. Pokud by existovalo jen jedno úložiště, byly by artefakty hned po nahrání veřejně dostupné. Vstupní úložiště Buffer naopak umožní, aby artefakty mohly být před vložením do hlavního úložiště Store testovány, případně upravovány. Proces průchodu artefaktu úložišti je zobrazen na obrázku 4.3. Repository



Obrázek 4.3: CRCE - životní cyklus komponent

API umožňuje přiřazení pluginů na jednotlivé události životního cyklu artefaktu.

4.2.4 Repository DAO API

Repository DAO API slouží k ukládání a načítání metadat o artefaktech v úložišti. Toto API se orientuje pouze na metadata, neslouží k ukládání samotných artefaktů. Cílem API je odstínit artefakty a komponenty pracující s jejich metadaty od fyzického umístění souborů s metadaty artefaktů. Pro identifikaci artefaktu a jeho metadat se používá z důvodu obecnosti URI. Není tedy předepsáno, že soubory s metadaty se musí nacházet v lokálním souborovém systému.

4.2.5 Web UI

Tento modul tvoří webové uživatelské rozhraní, které umožňuje provádět základní funkce úložiště. Tento modul obsahuje zapouzdřený webový server, který vytváří webové stránky, pomocí nichž lze úložiště ovládat. Mezi základní možnosti webového rozhraní patří:

- zobrazování artefaktů v úložišti (v obou úložištích, Store i Buffer)
- možnost zobrazit metadata artefaktů
- možnost editovat metadata artefaktů
- vkládání artefaktů do úložiště (jak do dočasného Buffer, tak následně do Store)
- získávání artefaktů z úložiště
- mazání artefaktů z úložiště
- spouštění testů kompatibility
- zobrazení seznamu nainstalovaných pluginů (modulů)

4.3 Implementace

V předchozí sekci byla nastíněna architektura úložiště OSGi. Tato sekce pojednává o tom, jak a pomocí jakých technologií, byla architektura realizována.

Úložiště CRCE je postaveno na OSGi. Jednotlivé moduly se balí do OSGi komponent (bundle) a jsou poté spuštěny v OSGi frameworku. Pro balení modulů do OSGi komponent a další záležitosti jsou použity nástroje z projektu **Pax** (*https://ops4j1.jira.com/wiki/display/ops4j/Pax*). Pax runner je jedním z těchto nástrojů a zajišťuje spuštění modulů ve formě OSGi komponent ve frameworku Apache Felix.

Samotné moduly i celý projekt dohromady je sestavován pomocí Mavenu. Strukturu projektu a jeho modulů je možné najít na adrese:

`https://www.assembla.com/spaces/crce/wiki/Project_structure`

Apache Felix Dependency Manager je využit pro zajištění závislostí mezi moduly. Umožňuje deklarativně specifikovat závislosti pomocí jednoduchého Java API a odstiňuje tím programátora o nutnost starat se o registraci a používání OSGi služeb.

4.4 Tvorba modulů

Každý modul CRCE je OSGi komponentou, která je sestavovaná pomocí nástroje Maven.

Nový modul je třeba vytvořit jako projekt s Maven strukturou. V souboru *pom.xml* je třeba definovat rodiče následovně:

```
<parent>
  <groupId>cz.zcu.kiv.crce.crce-reactor.build</groupId>
  <artifactId>compiled-bundle-settings</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <relativePath>../poms/compiled/</relativePath>
</parent>
```

Rodičem je Maven artefakt *compiled-bundle-settings*, který je společným předkem pro všechny části CRCE, které jsou kompilované ze zdrojových kódů.

Základní třídou modulu je jeho aktivátor (Activator), který je třeba umístit do interního balíku pluginu (*cz.zcu.kiv.crce.packageName.internal*). V této třídě je použit Apache Felix Dependency Manager, který se stará o správu OSGi služeb. Aby aktivátor mohl využívat služeb Dependency Managera, musí být potomkem abstraktní třídy *DependencyActivatorBase*. Tato abstraktní třída předepisuje implementaci metody *init()*, ve které jsou vytvořeny nové komponenty a vyřešeny závislosti jejich služeb. Tato metoda typicky obsahuje volání podobné následujícímu:

```
manager.add(createComponent()
                .setInterface(Plugin.class.getName(), null)
                .setImplementation(ExamplePlugin.class)
                .add(createServiceDependency().setService(Service.class).setRequired(false))
                );
```

Dependency Manager zde vytváří novou komponentu, v tomto případě se jedná o plugin, jehož rozhraní *Plugin* bylo definováno v Plugin API. Implementace je ve třídě *ExamplePlugin* a nejsnazší možností je tuto třídu vytvořit

jako potomka abstraktní třídy *AbstractPlugin* z Plugin API. Nová komponenta zde rovněž definuje svou závislost na službě, která jí poskytne třídu *Service*.

Nově vytvořený modul je třeba zabalit do OSGi komponenty. K tomu je využit Maven plugin *maven-bundle-plugin*, jehož společné nastavení je u Maven předka modulu (*compiled-bundle-settings*). Toto výchozí nastavení lze pro každý modul upravit v souboru *osgi.bnd*, který by měl být v kořenovém adresáři pluginu (spolu se souborem *pom.xml*).

Pak už stačí jen přidat nově vytvořený modul do kořenového *pom.xml* celého CRCE. Ten obsahuje seznam modulů uvnitř svého tagu s názvem <*modules*>. Do něj se nový modul přidá pod svým Maven artefactId:

```
<module>artefactId</module>
```

4.5 Plánovaný budoucí rozvoj

Úložiště CRCE je v neustálém vývoji, jehož částí je i tato diplomová práce. Vzhledem k použití specifikace OBR z OSGi je třeba reagovat na změny v této specifikaci. O tom se zmiňuje následující podsekce o CRCE 2.

Plánované vylepšení úložiště:

- plně integrovat do CRCE podporu mimo-funkčních charakteristik, které jsou vyvíjeny v projektu EFFCC[21].
- vyhodnocování kompatibility komponent při jejich vkládání do úložiště
- integrace s Maven repository
- rozšíření metadat. V plánu je přidání metadat o kompatibilitě, nebo právě metadat o mimo-funkčních charakteristikách.

4.5.1 CRCE verze 2

V současnosti probíhá vývoj nové verze CRCE, v které je změněno několik důležitých věcí. V první řadě je to změna Metadata API, aby odpovídalo

specifikaci Repository service v OSGi 5 (popsána v sekci 3.2.1 na straně 11). Původní návrh Metadata API vycházel ze staršího návrhu OBR (OSGi RFC-0112). V plánu pro tuto verzi jsou i další novinky, jako je implementace databázového Metadata DAO nebo vytvoření nového Resolveru, který by odpovídal specifikaci Resolver service z OSGi 5.

5 Možnosti komunikace klient-server

Úkolem této diplomové práce je rozšíření úložiště CRCE o možnost komunikace s klienty úložiště. Tato kapitola obsahuje porovnání technologií pro realizaci této komunikace. Jde tedy obecně o technologie s architekturou Klient-server, které umožňují přenos zpráv. U technologií používajících zprávy v XML jsou uvedeny ukázky těchto zpráv, aby je bylo možné porovnat mezi sebou. Důraz je kladen především na REST (viz podsekce 5.2 na straně 34), což je vybraný způsob komunikace realizovaný v CRCE REST modulu.

Některé často používané pojmy:

Klient-server je nejčastější přístup, s kterým se setkáváme v architektonických stylech pro síťové aplikace [17]. Server je část celku, která nabízí určité služby a čeká na žádosti o použití těchto služeb. Klient je další část, které chce tyto služby využívat. Klient odešle serveru žádost o použití služby. Server poté službu vykoná nebo odmítne vykonat a pošle klientovi odpověď s výsledky.

SOA - Service-oriented architecture je architektonický princip pro budování rozsáhlých softwarových systémů založených na službách. Pod pojmem služba je myšlena část systému, která poskytuje určitou funkčnost a poskytuje určité rozhraní pro komunikaci s ostatními službami. Jde v poslední době o módní a hojně používaný pojem, který není nijak přesně definovaný [27]. Často se pomocí tohoto pojmu označují systémy, které jsou propojené pomocí webových služeb.

RPC - Vzdálené volání procedur je popsáno v úvodu následující podsekce (5.1)

Webová služba - Pod pojmem webová služba (Web service) zde budou označeny služby, které je možno využívat prostřednictvím World Wide Webu. V knize RESTful Web Services [27] autor rozděluje webové služby do tří skupin:

- RESTful služby, které popisuje sekce 5.2 na straně 34

- služby, jejichž architektury jsou založeny na RPC (sekce (5.1))
- hybridní REST-RPC služby, což jsou formálně REST služby, které ale nedodržují principy správného návrhu RESTful služeb. O zásadách správného návrhu REST API pojednávájí sekce 5.3 (na straně 38) a 5.4 (na straně 42)

5.1 Možnosti komunikace založené na RPC

Vzdálené volání procedur (Remote procedure call) [25] umožňuje počítačovému programu zavolat proceduru (funkci), která se může vykonat v jiném místě adresního prostotu (na jiném místě v síti).

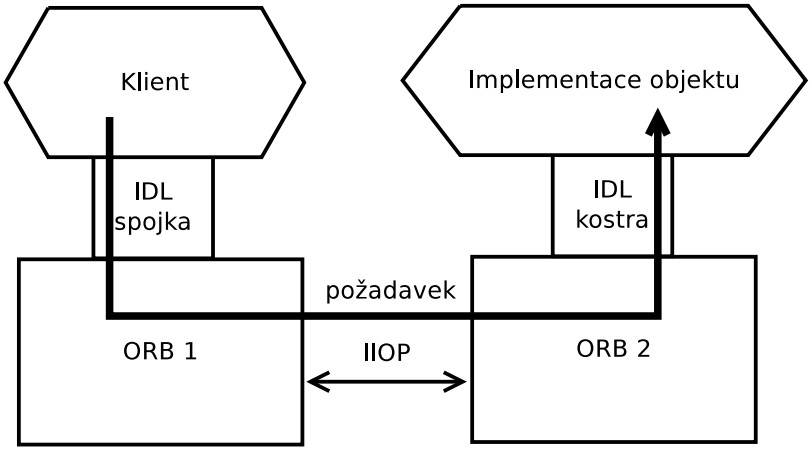
Volání vzdálené procedury z hlediska volajícího probíhá stejně, jako volání lokální procedury. Veškeré záležitosti přenosu zajišťuje implementace protokolu RPC.

Volající proces vyvolá lokální proceduru, která se nazývá spojka (stub). Parametry jsou jí předány stejným způsobem, jako lokální proceduře. Spojka na straně klienta zabalí proceduru včetně parametrů do zprávy pro partnerskou spojku na straně serveru. Proces balení parametrů procedury se nazývá marshalling. Spojka serveru zprávu přijme a rozbalí (unmarshalling). Spojka serveru poté na serveru proceduru vykoná jako běžnou lokální proceduru. Výstup je stejným způsobem pomocí spojek odeslán zpátky na klienta, kde klientská spojka vrátí výsledky a skončí, přičemž předá řízení původnímu procesu.

Tato sekce popisuje možnosti využití principu RPC k realizaci Klient-server komunikace

5.1.1 CORBA

Common Object Request Broker Architecture [19, 16, 24, 26] je standard pro tvorbu distribuovaných objektově orientovaných aplikací. Vznikl už v roce 1991 a je spravován konsorciem OMG (Object Management Group). Základem CORBA jsou distribuované objekty, které jsou jazykově nezávislé. Klient může použít objekt bez ohledu na programovací jazyk, v kterém je naprogramován i bez ohledu na jeho umístění v síti.



Obrázek 5.1: CORBA - Požadavek z klienta k implementaci objektu

IDL - Interface description language je jazyk využívaný pro popis rozhraní, které objekty poskytují. Vychází z jazyka C++. Mapování z IDL existuje pro mnoho programovacích jazyků, mimo jiné pro jazyky C, C++, Java, Smalltalk, Python a Ada. Použití IDL umožňuje oddělení rozhraní od jeho implementace.

Ukázka IDL:

```
interface salestax  {
     float  calculate_tax ( in float taxable_amount );
}
```

ORB - Object Request Broker zajišťuje komunikaci mezi objekty. V původní verzi nebyl formát přenosu dat definován, což bylo problematické. Řešení (pro sítě TCP/IP) přinesla až CORBA 2 vydání v roce 1996, kde je definován protokol IIOP (Internet Inter-ORB Protocol), který je implementací obecného protokolu GIOP (General InterORB Protocol).

Použití architektury CORBA spočívá v tom, že programátor popíše rozhraní vytvářeného objektu v IDL a pomocí mapování do svého programovacího jazyka si nechá vygenerovat IDL spojku a rozhraní. Poté stačí programátorovi dopsat implementaci objektu. Ukázka poslání požadavku vzdáleně umístěné implementaci objektu je znázorněna na obrázku 5.1.

Výhody:

- Mnoho podporovaných programovacích jazyků
- IDL a s ním spojené oddělení rozhraní od implementace
- Vhodný pro spojování různých (a různorodých) objektů a systémů
- CORBA systémy mohou nabídnout velký výkon

Nevýhody:

- Nutnost naučit se další jazyk (IDL)
- Nepodporuje přenos objektů
- Složitější použití

5.1.2 DCOM

Distributed Component Object Model [24] je proprietární obdobou standardu CORBA určenou pro vzdálenou komunikaci komponent standardu COM (Component Object Model). Ten byl vytvořen v roce 1993 společností Microsoft. Na rozdíl od CORBA se zaměřuje spíše na stolní počítače, než na průmyslové využití. Jeho protokol pro přenos dat se nazývá ORPC (Object Remote Procedure Call).

Výhody a nevýhody:
DCOM je obdobný standardu CORBA s nevýhodami, vyplývajícími z omezení na jednu platformu a jednoho výrobce. Výhodou je relativní rozšířenost, vzniklá dominancí produktů firmy Microsoft na desktopech.

5.1.3 Java RMI

Java Remote Method Invocation [6, 26] je implementace RPC v Javě. Umožňuje z jednoho virtuálního stroje (JVM) volat metody objektu na jiném

virtuálním stroji. Důležitá je vlastnost RMI, umožňující přenos nejen primitivních datových typů, ale i celých Java objektů. Klientovi může být tímto způsobem zaslán nový kód, který může následně dynamicky spouštět.

Remote Object (vzdálený objekt) je objekt, jehož metody mohou být vyvolány objektem z jiného virtuálního stroje (JVM). Tento objekt musí implementovat rozhraní *java.rmi.Remote*. Při přenosu vzdáleného objektu je na cílovou JVM předána spojka (stub), která se chová jako lokální reprezentace objektu

Přenos obecných objektů - Obecným objektem jsou myšlený ostatní objekty, které nejsou vzdálené. Přenos obecných objektů jako parametrů je také možný. Na cílovou JVM jsou předány jako kopie. Objekt je před přenosem serializován (Serialization), při čemž dojde k přeměně objektu na seřazený proud bytů, které jsou následně přeneseny po síti.

RMIregistry je jmenná služba sloužící k registraci vzdálených RMI objektů. Na každé JVM může být spuštěn pouze jeden RMIregistry. Server si v registru registruje svoji službu pomocí JNDI a klient poté může z registru získat spojku (stub), která mu umožní tuto službu používat. JNDI (Java Naming and Directory Interface) je API umožňující svázaní (bind) objektu se jménem a následně klientům umožňuje vyhledávat objekty podle jména.

Security manager je objekt definující bezpečnostní omezení, které jsou individuální pro každou aplikaci. Může omezovat akce spojky (stub).

Výhody:

- Podpora různých platforem
- Možnost zasílání objektů. Klientovi může být zaslán nový kód a poté na jeho JVM dynamicky spuštěn. To přináší velkou flexibilitu.
- Přítomen v Javě od JDK1.02, velké množství vývojářů má s touto technologií zkušenosti

Nevýhody:

- Vázaný na platformy, které podporuje Java
- Může pracovat jen s Java systémy. Silná vazba na jeden programovací jazyk.
- Bezpečnostní hrozby spojené se vzdáleným vykonáváním kódu. Omezení vynucená bezpečnostními opatřeními.

Poznámka - pro odstranění závislosti na Javě vznikl i RMI-IIOP, což je implementace RMI na CORBA základu.

5.1.4 XML-RPC

XML-RPC [30] je implementací RPC používající XML a HTTP transportní mechanismus (více o HTTP v sekci 5.2.1 na straně 35). Je to přímý předchůdce SOAP(5.1.5). Požadavek je odeslán jako HTTP metoda POST. V požadavku je identifikace metody a její parametry. Pro parametry má protokol definované vlastní datové typy. Odpověď je vrácena ve formě HTTP odpovědi s přiloženým XML, který obsahuje výstup metody, nebo popis nastalé chyby.

Příklad požadavku:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
      </param>
   </params>
</methodCall>
```

A odpovědi:

```
HTTP/1.1 200 OK
```

```
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
      </param>
   </params>
</methodResponse>
```

Výhody a nevýhody do značné míry odpovídají jeho nástupci, protokolu SOAP.

Výhody:

- První protokol využívající HTTP
- Jednoduchá struktura, poměrně snadné použití

Nevýhody:

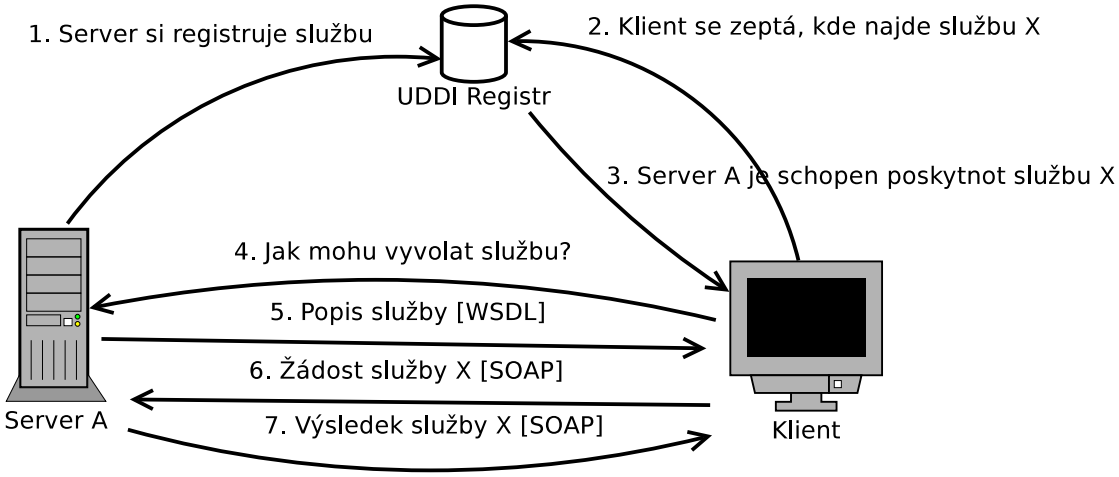
- SOAP je mnohem mocnější, umožňuje více možností
- Oproti RESTu obsahuje zbytečnou vrstvu abstrakce
- XML zpráva je velká v poměru k datům, které obsahuje.
- Je pomalejší než CORBA

5.1.5 Webové služby - SOAP/WSDL

Tato sekce je věnuje jedné skupině webových služeb, které využívají protokoly SOAP a WSDL [28, 22, 14, 31]. V této sekci bude tento konkrétní typ webových služeb označován jen pod pojmem webové služby. Princip těchto webových služeb je založen na kooperaci třech navzájem nezávislých technologií. Tyto technologie mají samostatně definované protokoly, ale jejich funkčnost se doplňuje.

Princip webových služeb je založen na třech hlavních technologiích:

- UDDI (Universal Description, Discovery and Integration) - registr umožňující vyhledávání webových služeb
- WSDL (Web Services Description Language) - Prostředek pro popis rozhraní webové služby
- SOAP (Simple Object Access Protocol) - protokol pro komunikaci



Obrázek 5.2: Vyvolání webové služby

Příklad použití technologií je znázorněn na obrázku 5.2:

1. Server poskytující službu se registruje v UDDI registru.
2. Klient hledá webovou službu, která splňuje jeho požadavky. Pošle tedy dotaz UDDI registru. UDDI má definované API [5], které umožňuje různé druhy dotazů a definuje odpovědi. UDDI registr pracuje také jako webová služba. Požadavky a odpovědi jsou obvykle ve formě SOAP zpráv.
3. UDDI registr odpoví, který server poskytuje příslušnou službu. Odpověď typicky obsahuje adresu serveru ve formě URI.
4. Klient se zeptá webového serveru, jak má službu použít. Dotaz je ve formě HTTP GET dotazu(více o HTTP v sekci 5.2.1 na straně 35) na URI serveru. Tento a následující krok někdy neproběhnou, pokud klient získá z UDDI serveru přímo popis služby v jazyce WSDL.

5. Server popíše službu v jazyce WSDL
6. Klient vyvolá službu (příklad SOAP žádosti na na straně 33).
7. Server odešle SOAP odpověď s požadovaným výsledkem nebo s chybovou zprávou. (příklad SOAP odpovědi na na straně 33)

WSDL je protokol, který definuje popis webových služeb ve formátu XML. Klient se pomocí popisu služby například dozví, jaké RPC metody může použít, jaké argumenty tyto metody očekávají a jaké datové typy vracejí. WSDL XML se skládá z následujících elementů:

- Types – obsahuje definice datových typů. Obvykle jsou zapsané v XSD formátu, i když může být použit i jiný formát.
- Message – popisuje formát zprávy pomocí dříve definovaných datových typů (v elementu *Types*)
- Operation – abstraktní popis akcí, které služba vykonává.
- Port Type – množina podporovaných operací
- Binding – přiřazuje portu konkrétní protokol a formát přenosu zpráv.
- Port – koncový bod, definovaný jako kombinace elementu *Binding* a síťové adresy.
- Service – kolekce koncových bodů

Ukázka WSDL souboru:

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2000/10/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
```

```
                </all>
            </complexType>
        </element>
        <element name="TradePrice">
            <complexType>
                <all>
                    <element name="price" type="float"/>
                </all>
            </complexType>
        </element>
    </schema>
</types>

<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
        <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>

<service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
    </port>
</service>

</definitions>
```

SOAP je protokolem pro výměnu zpráv ve formátu XML. Obecně nedefinuje, jaké zprávy bude obsahovat. Jeho typické použití je pro realizaci webových služeb, kdy nese žádost o použití služby a výsledek volání služby. SOAP ve verzi 1.0 vznikl v roce 1999. V současnosti je aktuální verze 1.2 z roku 2007. Přenos zpráv je obecně umožňuje přes HTTP(S), SMTP nebo i jiné transportní protokoly. Kořenovým elementem SOAP zprávy je vždy

SOAP Envelope (*soap:Envelope*). Ta obsahuje vyjma jmenných prostorů jeden nebo dva další elementy. Prvním je nepovinný element *Header* a druhým je povinný element *Body*. Tento element Body obsahuje jmenný prostor a neurčený počet elementů nebo atributů v tomto jmenném prostoru. Právě tyto atributy jsou nositeli informace.

Příklad SOAP žádosti [4]:
Tato žádost je požadavkem na vykonání metody *GetStockPrice*. Metoda má za úkol zjistit cenu akcií a její parametr *StockName* určuje, o jaké akcie jde.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Příklad SOAP odpovědi [4]:
Odpověď na předchozí žádost o vrácení ceny akcií. Obsahem SOAP Body je odpověď *GetStockPriceResponse*, která vrací hodnotu ceny akcií IBM v elementu *Price*.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Výhody:

- Podpora různých transportních protokolů (nejrozšířenější je HTTP)
- Pokud využije HTTP, není obvykle filtrována firewally a prochází proxy servery. Tím se liší oproti technologiím jako CORBA a DCOM.
- Je poměrně rozšířen v komerčním využití

Nevýhody:

- XML zpráva je velká v poměru k datům, které obsahuje. Není moc lidsky čitelná.
- Je pomalejší než CORBA
- Oproti RESTu obsahuje zbytečnou vrstvu abstrakce

5.2 Základní koncepty RESTu

REST Representational State Transfer (REST) [10, 27] je styl softwarové architektury využitelný v distribuovaných systémech. REST byl navrhnut v roce 2000 Royem Fleldingem v jeho disertační práci [17]. Roy Fielding je jeden z autorů specifikace HTTP. S tím také souvisí základní myšlenka REST, kterým je širší využití HTTP protokolu, nejen pro přenos HTML stránek. Pro většinu účelů je snazší a elegantnější použít prosté HTTP metody, než používat složitější mechanizmy, jako je CORBA, RPC nebo SOAP.

World Wide Web stojí na třech základních technologiích:

- URI (Uniform Resource Identifier) - prostředek k unikátní identifikaci zdrojů na síti
- HTTP (HyperText Transfer Protocol) - protokol sloužící k vyžádání zdroje z URI a odpovědi na tuto žádost
- HTML (HyperText Markup Language) - jazyk, který je vlastně formátem vrácené odpovědi od zdroje

REST nabízí možnost použití prvních dvou již zavedených technologií k přenosu jiného obsahu, než jsou stránky v HTML.

Základem je tedy definovat si zdroje, označit je pomocí URI a poté s nimi pracovat pomocí HTTP. Právě tím se REST přístup liší od SOAP a jeho předchůdce RPC-XML. Ti také využívají HTTP, ale pod něj si přidávají další vrstvu abstrakce. SOAP je navržen k činnosti pod jednou URI, kdy jsou jednotlivé metody rozlišeny v textu XML SOAP zprávy. REST přístup říká, že HTTP je třeba využít tak, jak bylo navrženo. Další vrstva pod ním přináší jen zvýšené množství zbytečně přenesených dat.

Terminologie: REST je styl softwarové architektury. Architektury webových služeb, které tento styl používají, jsou označovány jako RESTful.

5.2.1 HTTP

Hyper Text Transfer Protocol (HTTP)[18] je protokol aplikační vrstvy, který se ve World Wide Webu využívá od roku 1990. Používá model Klient-server se dvěma základními typy zpráv: žádostí (Request) a dopovědí (Response). Klient požádá o určitý zdroj a server mu v odpovědi vrátí výsledek žádosti. Rest přístup se snaží o co nejširší využití protokolu HTTP.

HTTP metody

Existují 4 HTTP metody, které odpovídají základním CRUD metodám. CRUD (Create/Read/Update/Delete) je akronym pro základní funkce trvalého úložiště dat. Právě tyto metody jsou nejčastěji využívány v REST architekturách.

- POST - vytvoření (create) zdroje
- GET - získání (read) zdroje
- PUT - změna (update) zdroje
- DELETE - smazání (delete) zdroje

Ostatní HTTP metody jako HEAD, TRACE, OPTIONS a CONNECT nejsou v RESTful systémech tak často využívané.

Stavové kódy HTTP

Stavový kód je součást hlavičky HTTP odpovědi. Stavový kód je odpovědí na to, jak byla žádost vyřízena (kladně, chyba serveru ..). Stavové kódy jsou rozděleny do 5 kategorií (v každé kategorii jsou ze uvedeny jen některé příklady stavových kódů):

Informační 1xx

- **100 Continue** - Server obdržel hlavičky žádosti a čeká na klienta, až pošle tělo zprávy.

Úspěch 2xx

- **200 OK** - Standardní odpověď na požadavek, který byl úspěšně vyřízen.

Přesměrování 3xx

- **301 Moved Permanently** - Tato a všechny budoucí žádosti by měly být směrovány na dané URI.

Chyba klienta 4xx

- **400 Bad Request** - Požadavek byl podán nesprávně.
- **404 Not Found** - Požadovaný zdroj nebyl nalezen.

Chyba serveru 5xx

- **500 Internal Server Error** - Obecná zpráva o chybě na serveru.
- **501 Not Implemented** - Nebyla rozpoznána metoda v požadavku, nebo server tuto metodu neovládá.
- **503 Service Unavailable** - Služba je dočasně nedostupná

5.2.2 Požadavky na REST architektury

Požadavky na architektury, které je možno považovat za architektury REST stylu, definoval R.Fielding ve své práci [17] takto:

1. Klient-server - Oddělení klienta a serveru.
2. Beze-stavovost - Žádost od klienta musí obsahovat všechny potřebné informace. Server si nepotřebuje udržovat žádný stav komunikace s klientem.
3. Možnost použití mezipaměti (cache) - Princip snižující zátěž sítě. Klient či mezilehlý uzel si může uchovávat odpověď a použít ji v případě další žádosti. Nevýhodou je, že obsah mezipaměti nemusí odpovídat obsahu odpovědi, kterou by v tom okamžiku získal klient ze serveru.
4. Jednotné rozhraní - požadavek, který nejvíce odlišuje REST architektury od ostatních síťových architektur. Rest architektury by měli mít společné rysy vzniklé dodržováním základních principů návrhu. Příkladem jsou identifikace zdrojů a vhodné použití HTTP metod. Více o jednotném rozhraní najdete v sekci 5.3 na straně 38.
5. Vrstvený systém - Klient nemůže říci, zda komunikuje přímo s konečným serverem nebo některým jiným zařízením. Mezilehlé servery mohou snižovat zátěž sítě (škálovatelnost).
6. Kód na vyžádání (nepovinný požadavek) - Možnost, při které si klient stáhne ze serveru vykonavatelný kód, který rozšíří jeho funkčnost.

Tyto požadavky jsou značně obecné a s většinou z nich nemusí tvůrce REST architektury moc zápolit. Již samotné použití protokolu HTTP prakticky zajišťuje splnění požadavků 1,2,3 i 5. Protokol HTTP je založen na principu klient-server, je beze-stavový, podporuje použití mezipaměti a jeho viditelná hlavička plní požadavek na vrstvený systém. Je třeba se tedy zaměřit na čtvrtý požadavek, kterým je využívání jednotného rozhraní. Tímto požadavkem se zabývá následující sekce 5.3. Samotnému návrhu REST architektur se věnuje sekce 5.4 na straně 42.

5.3 Jednotné rozhraní

Jednotné rozhraní (Uniform interface) označuje společné rysy, které by měli mít systémy založené na REST stylu architektury. Jednotné rozhraní velmi zjednodušuje práci s potenciálně neznámým systémem. Například vždy, když máme k dispozici URI odkazující na zdroj, můžeme tušit, že můžeme získat reprezentaci tohoto zdroje pomocí metody GET. V této sekci postupně projdeme důležité principy, které je třeba dodržovat pro udržení jednotného rozhraní při návrhu RESTful API. Obsah této sekce je z podstatné části založen na knihách RESTful Web Services Cookbook [10] a RESTful Web Services [27].

5.3.1 Identifikace zdrojů

Základem abstrakce informace v RESTu je zdroj (resource). Každá informace, která může být pojmenována, je zdroj. Zdrojem může být dokument, obrázek, služba vracející kurz měny, kolekce dalších zdrojů nebo i nevirtuální objekt (například osoba). Zdroj je jednoznačně identifikován URI. Server nevrací zdroje, ale jejich reprezentaci, která odráží současný nebo zamýšlený stav zdroje. Například pokud je zdroj osobou, může být jeho reprezentací XML s jeho jménem, adresou, datem narozením, atd. Jak už bylo naznačeno, každý zdroj je pevně svázán s jednou URI. Pravidla:

- Význam mapování zdroje na URI by se neměl měnit. Obsah *example.com/users* se tedy měnit může, ale skutečnost, že tato URI odkazuje na seznam uživatelů by se měnit neměla.
- Identita zdroje není závislá na jeho hodnotě. Dva zdroje mohou obsahovat (odkazovat na) stejnou hodnotu, ale nejsou jedním zdrojem. Identita zdroje je tedy definována jeho URI.

Z těchto důvodů je velmi důležitý návrh URI pro zdroje, který by se už neměl měnit. Zásady pro návrh URI najdete v sekci 5.4.3 na straně 44.

5.3.2 Viditelnost

Viditelnost je definována jako schopnost prostředníka monitorovat nebo zasahovat do komunikace jiných dvou komponent [17]. Viditelnost může zlepšit výkon systému (použití mezipaměti - cache), škálovatelnost (vrstvený systém), spolehlivost (umožnění sledování systému) i bezpečnost (umožnění práce firewallům). Návrh se tedy musí snažit, aby prostředník mohl ze zprávy získat co nejvíce informací bez znalosti konkrétních informací specifických pro danou službu. Podstatné je co nejširší využití HTTP hlavičky. Dále je nutné používat vhodné operace. Metody GET, PUT a DELETE je vhodné používat k účelu, ke kterému byli vytvořeny (popsáno v sekci 5.3.4 na straně 40). Metoda POST je obecně využitelná nejen k vytvoření zdroje, ale k vykonání všech potenciálně nebezpečných a neidempotentních operací (o těch více v sekci 5.3.3 na straně 39).

Viditelnost je někdy nutné obětovat, pokud je potřeba měnit více zdrojů. Nebo existuje více zdrojů, které sdílejí data.

5.3.3 Bezpečnost a idempotentnost

Bezpečnost a idempotentnost jsou vlastnosti HTTP metod, které je bezpodmínečně nutné při využití metod zachovat.

- Bezpečné (safe) metody - nezpůsobují změnu stavu serveru. Jde tedy pouze o čtení určitých dat.
- Idempotentní (idempotent) metody - opakování žádosti má stejný efekt, jako když je odeslána poprvé. Příkladem je žádost o změnu určité hodnoty na X za použití metody put. At' už byla hodnota jakákoliv, po použití této metody bude vždy stejná.

Metoda	je bezpečná?	je idempotentní?
GET	ano	ano
HEAD	ano	ano
OPTIONS	ano	ano
PUT	ne	ano
DELETE	ne	ano
POST	ne	ne

5.3.4 Použití jednotlivých metod

Tato sekce obsahuje doporučené použití HTTP metod v RESTful systémech.

GET

Metoda GET je vhodná pro bezpečné a idempotentní získání informace. Jejím úkolem je získat reprezentaci stavu zdroje. Tato metoda je ze své podstaty obvykle použita správně. Jen je si třeba dát pozor, aby metoda nebyla použita pro nebezpečné a neidempotentní operace.

Příklady nesprávného použití této metody GET:

- Přidání stánky do záložek:

```
GET /bookmark/add_bookmark?www.example.org%2Fpage.html HTTP/1.1
Host: www.example.org
```

  Tato operace není bezpečná. Pro tento účel by tedy neměla být použita metoda GET, ale metoda POST.

- Smazání poznámky:

```
GET /notes/delete?id=123 HTTP/1.1
Host: www.example.org
```

  Ani tato operace není bezpečná. Ukázku správně použité metody DELETE pro tento příklad najdete v sekci 5.3.4 na straně 41.

POST

Vhodné použití metody POST:

- vytvoření nového zdroje
- úprava jednoho nebo více zdrojů prostřednictvím kontrolujícího zdroje (více v 5.4.2 na straně 44)

- operace, pro které by URI obsahovala příliš dlouhou nehierarchickou část. Nehierarchickou částí je myšlen především dotaz (query). I když protokol HTTP neurčuje žádné omezení délky URI, prohlížeče a webové servery mohou v zpracovávat pouze omezeně dlouhou URI (Apache web server ve výchozím nastavení omezuje délku URI na 8190 bytů).
- vykonání jakékoliv nebezpečné a neidempotentní operace, pro které se nezdá vhodná jiná HTTP metoda

Chybné použití metody POST většinou spočívá v jejím použití pro operaci, kterou by mohla uskutečnit jiná HTTP metoda.

PUT

Operace se používá pro změnu některého zdroje. V určitých případech je možné tuto metodu použít i k vytváření nového zdroje. K tomu je ale nutné, aby klient mohl rozhodnout o URI tohoto zdroje. Příkladem může být server, který vyhradí klientovi určitou kořenovou URI pro každého klienta a alokuje si pro ně místo na svém pevném disku. V tomto případě je vytváření idempotentní. Ve většině případů je ale při vytváření vhodné, aby tento proces i návrh URI kontroloval server. Proto je pro vytváření zdroje častěji používána metoda POST

DELETE

Využívá se k mazání zdroje. Příkladem může být správné smazání poznámky (uvedena v příkladu chybného použití metody GET v sekci 5.3.4 na straně 40):

```
DELETE /notes/123 HTTP/1.1
Host: www.example.org
```

Nestandardní metody

Proběhlo několik pokusů o rozšíření množství HTTP metod. Nejznámější je projekt WebDAV, který navrhl použití dalších HTTP metod, jako jsou PROPFIND, COPY, MOVE, LOCK a další. Je doporučováno vyhnout se

použití těchto metod, jelikož se nejde spolehnout na to, že je budou podporovat všechny servery a prostředníci na cestě k nim. Pokud by se ale staly v budoucnu standardem, mohly by tyto metody přispět ke snazšímu zachování viditelnosti a jednotnosti rozhraní REST architektur.

5.4 Návrh REST architektury

Tato sekce obsahuje postupy, které je možné a vhodné použít při návrhu REST architektury. Obsah této sekce je z podstatné části založen na knihách RESTful Web Services Cookbook [10] a RESTful Web Services [27].

5.4.1 HTTP hlavička

V hlavičce HTTP žádostí (request) a odpovědí (response) je vhodné použít co nejvíce hlaviček, jelikož to pomáhá zachovávat viditelnost. Příklady některých, které by měly být pokud možno užity:

- Content-Type - Popisuje typ reprezentace pomocí MIME typu (*text/html*, *image/png*, ...). U textových typů je doporučeno přidat za MIME typ i použité kódování:

  ```
  Content-Type: application/xml;charset=UTF-8
  ```

- Content-Length - velikost těla reprezentace v bytech. Zbytečné v HTML 1.1, které používá *chunked transfer encoding*.
- Content-Language - dvoupísmenná zkratka jazyka (podle RFC 5646).
- Content-MD5 - md5 hash těla pro kontrolu
- Content-Encoding - používá se v odpovědi, pokud je třeba její tělo dekódovat.
- Last-Modified - časová známka, kdy server naposledy změnil reprezentaci nebo zdroj

U GET žádostí může být důležitá hlavička *Cache-Control*, pomocí které lze nastavit, zda může být odpověď uložena do mezipaměti. To je důležité zejména u dotazů na proměnné zdroje.

5.4.2 Práce se zdroji

V této sekci jsou uvedena doporučená řešení situací, které mohou nastat při návrhu metod a systému zdrojů.

Kolekce - Collection

Kolekce [10] je spojení zdrojů, na které se lze hromadně dotázat. Zdroj může být zároveň ve více kolekcích. U větších kolekcí je vhodné podpořit stránkování, kdy se v odpovědi vrací určený pouze počet prvků kolekce, současně s URI odkazy na současnou, předchozí a následující stránku.

Reprezentace kolekce by měla obsahovat:

- odkaz na sebe sama

```
<link rel="self" href="http://www.example.org/users"/>
```

- odkaz na další stránku, pokud je kolekce stránkovaná
- odkaz na předchozí stránku, pokud je kolekce stránkovaná
- indikátor velikosti kolekce

Komposity - Composite

Pokud máme více zdrojů různých druhů, které bychom často vraceli za sebou (mají společný logický význam), je možné je spojit dohromady do kompositu [10]. Příkladem budiž aplikace, která zobrazuje k zákazníkovi jeho 10 posledních objednávek a jeho 10 posledních reklamací. Aby nebylo vždy po zobrazení zákazníka nutné posílat postupně GET žádosti na zákazníka, na jeho objednávky a na jeho reklamace, je vhodné navrhnout zdroj spojující tyto údaje dohromady. Tento komposit je dotázán jednou GET žádostí a vrácen jako jeden zdroj, který je složen ze 3 dalších zdrojů (zákazník, kolekce objednávek, kolekce reklamací).

Kontroler - Controller

Použití kontroleru je vhodné v operacích, které atomicky mění více než jeden zdroj. Pro každou takovou operaci je třeba vytvořit kontrolující zdroj [10] (contoller resource), který operaci obstará. Klient požádá o spuštění kontroleru metodou POST. Některá typická použití tohoto vzoru:

- Účelem operace je vytvořit nový objekt. V odpovědi je vrácen stavové kód 201 (Created) a hlavička *Location* s URI nově vzniklého objektu.
- Operace mění jeden nebo více objektů. V odpovědi je vrácen stavové kód 303 (See Other) s URI, kterou klient může použít k zobrazení změněných objektů (URI kolekce, viz sekce 5.4.2).
- Výsledek operace není možno vrátit jako URI. Je tedy vrácen kód 200 (OK) a v těle odpovědi je reprezentace, která klientovi umožní dostat se na změněné objekty.

Praktickým příkladem budiž situace, kdy uživatel má v mobilním telefonu adresář, který potřebuje synchronizovat s adresářem na serveru. Klient pošle POST s výpisem svého adresáře. Server zpracuje synchronizaci (adresáře podle určitých pravidel spojí) a vrátí kód 303 (See Other) s URI v hlavičce *Location*, která odkazuje na kolekci adres tvořící výsledný adresář.

5.4.3 Návrh URI

Při navrhování URI je vhodné zachovávat ustálené zvyklosti [10], které podporují jednotnost rozhraní. Mezi ně patří:

- použití domén a subdomén k logickému seskupení nebo rozdělení zdrojů za účelem lokalizace, šíření, nebo z bezpečnostních či jiných důvodů. Příkladem je nabízení lokalizace podle subdomény:

  ```
  http://en.example.org/book/1234
  http://cz.example.org/book/1234
  ```

  nebo dvě subdomény, jedna pro prohlížeče a druhá pro ostatní klienty:

  ```
  http://www.example.org/book/1234
  http://api.example.org/book/1234
  ```

- užití dopředného lomítka (/) v URI k vyjádření hierarchického vztahu zdrojů
- užití čárky (,), středníku (;) k vyjádření nehierarchických prvků

```
http://www.example.org/axis;x=0,y=2
```

- použití pomlčky (-) a podtržítka (_) k zlepšení čitelnosti dlouhých názvů.
- užití ampresandu (&) k oddělení parametrů v dotazové (query) části URI

```
http://www.example.org/search?word=Antarctica&limit=30
```

- vyhnutí se příponám souborů v URI

Velká písmena mohou v URI občas způsobit problém, podle RFC 3986 URI rozlišuje velká a malá písmena, kromě částí schématu a jména počítače (host). Následující URI jsou tedy shodné:

```
http://www.example.org/doc.txt
http://WWW.EXAMPLE.org/doc.txt
```

Ale odlišná je už URI:

```
http://www.example.org/Doc.txt
```

Mezera je validní znak a podle RFC 3986 by se měla v URI kódovat jako %20. Ve formulářích podle MIME typu *application/x-www-form-urlencoded* se však místo mezery používá znak '+'. Je třeba tedy dát pozor, aby nedošlo k záměně.

URI v reprezentaci zdroje - Kdykoliv je to možné, je vhodné vrátit klientovi URI jakou součást těla HTTP odpovědi. Při použití xml se doporučuje tag

```
<link href=URI rel="self"/>
```

Nepovinný parametr *rel* určuje vztah odkazovaného objektu s vraceným.

5.4.4 Reprezentace zdroje

Reprezentace zdroje není v REST nijak omezena a může být prakticky libovolná. Nejjednodušší možností je prostý text. Druh reprezentace by měl být vždy zapsán v hlavičce HTML jako MIME typ. Příklad odpovědi s označeným typem reprezentace zdroje:

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
```

Nejčastěji se ale pro popis zdroje používají následující prostředky.

XML

Extensible Markup Language je značkovací jazyk, který klade důraz na zachování logické struktury dokumentu. Je určen především pro výměnu dat mezi aplikacemi a uchovávání dat. XML značky (tagy) popisují význam jednotlivých částí dat. Je tedy vhodný pro popis prakticky jakéhokoliv zdroje. Pro popis jeho konkrétní struktury pro určité zdroje je možné využít XML schémata (XSD).

JSON

JavaScript Object Notation je textový formát dat, který se zaměřuje na jednoduchost a snadnou čitelnost lidmi. Vznikl z JavaScriptu, ale jako textový formát je na konkrétním jazyce zcela nezávislý. Výhodou je jeho jednoduchost. Informace v něm zapsané většinou zabírají méně místa než odpovídající informace v XML.

Ukázka:

```
{
"employees": [
{ "firstName":"John" , "lastName":"Doe" },
{ "firstName":"Anna" , "lastName":"Smith" },
{ "firstName":"Peter" , "lastName":"Jones" }
]
}
```

Reprezentace smíšených dat

K reprezentaci smíšených dat je možné použít MIME typy *multipart/mixed*, *multipart/related* nebo *multipart/alternative*. Tyto typy umožňují definovat si oddělovač (v příkladu je nastavena jako *abcd*), který části dat různých typů odděluje. Pokud je některá část dat binárním, je pro tuto část třeba nastavit kódování na *Base64*, aby se neposílala textově.

Příklad:

```
{
Content-Type: multipart/mixed boundary ="abcd"

--abcd
Content-Type: application/xml;charset=UTF-8

<movie> ... </movie>

--abcd
Content-Type: video/mpeg

...... zde je binární video .........

--abce--
```

5.5 REST a Java

RESTful API je v Javě samozřejmě možné vytvořit pomocí Java Servletů, jako jiné webové servery. Aby ale byla tvorba REST API v Javě co nejjednodušší, vzniklo prostřednictvím JCP (Java Community Process) speciální API specifikace pro tvorbu RESTful Webových služeb - JAX-RS. Při tvorbě RESTful webových služeb je tedy možné využít některou implementaci této specifikace nebo i jiné technologie, které tvorbu usnadňují.

5.5.1 JAX-RS

Java API for RESTful Web Services [9] je Java API , které poskytuje podporu pro vytváření webových služeb používajících REST. Používá anotace, které usnadňují mapovaní Java objektu (POJO) na webový zdroj.

Základními anotacemi jsou:

- @Path umožňuje nastavení relativní cesty ke zdroji (část URI)
- @GET, @PUT, @POST, @DELETE a @HEAD určují použití HTTP metod
- @Produces určuje MIME typ, který metoda produkuje
- @Consumes určuje MIME typ, který metoda umí přijmout

6 CRCE modul pro REST API

Hlavním úkolem této práce bylo vytvoření CRCE modulu, který by přidal CRCE úložišti REST API a umožnil tak jeho interakci s klienty. Klienti potřebují mít možnost získávat komponenty (obecně artefakty) z úložiště a také získávat metadata artefaktů, které jim umožní správně si vybrat potřebné komponenty. V této kapitole je modul popsán od jeho úkolu a specifikace, po vlastní implementaci.

6.1 Účel - specifikace

Základním úkolem REST API je umožnění komunikace s klienty úložiště. Možnosti komunikace jsou probrány v následujících scénářích použití REST API.

6.1.1 Scénáře použití

Pro komunikaci CRCE s klientem bylo navrženo několik scénářů použití. Tyto scénáře jsou pojmenovány zkratkami ve formátu SG-SN, kde SG určuje skupinu scénářů a SN vlastní jméno scénáře. Příkladem budiž scénář BO-SM, jehož zkratka je složena ze skupiny BO (základní operace - basic operations) a jména SM (jedna komponenta - single bundle).

Navržené scénáře včetně důležitosti jejich implementace jsou zobrazeny v následující tabulce:

Zkratka	Anglický název	implementace
BO-SB	Obtain a single bundle	musí být
BO-SM	Obtain a single bundle metadata	musí být
BO-AM	Obtain metadata about all available bundles	musí být
BO-OM	Obtain metadata about ”other”bundles	měla by být
SM-RS	Which bundle versions can replace this single one?	musí být
SM-RA	Which bundle versions can replace this one within an application?	měla by být
SM-OS	Which other bundles could replace this one?	nemusí být
SM-OA	Which other bundles can replace this one within an application?	nemusí být
SM-FP	Find bundles, that are providers of given capability	měla by být
SM-FC	Determine transitive closure satisfying requirements	nemusí být
SM-UA	Metadata for Application Update	měla by být

Základní operace

Základní operace jsou operace umožňující získání komponenty a jejích metadat.

Scénář BO-SB: Obtain a single bundle - získání komponenty Scénář, kdy klient žádá CRCE, aby mu úložiště poskytlo konkrétní komponentu.

Klient musí v žádosti určit, kterou komponentu žádá. Provede to pomocí identifikačního čísla (id) komponenty nebo pomocí její jména a verze.

CRCE v odpovědi buď pošle komponentu nebo odpoví, že požadovaná komponenta nebyla nalezena.

Scénář BO-SM: Obtain a single bundle metadata - získání metadat jedné komponenty Klient žádá CRCE o poskytnutí metadat jedné komponenty. Komponentu určí podle identifikačního čísla nebo podle jména a verze. Může rovněž přidat kritérium, které určí, o které skupiny metadat má zájem.

CRCE nazpátek pošle vyžádaná metadata komponenty (v XML nebo JSON formátu) nebo odpoví, že neobsahuje požadovanou komponentu.

Scénář BO-AM: Obtain metadata about all available bundles - získání metadat o všech komponentách Klient požádá CRCE o poskytnutí metadat všech komponent dostupných v úložišti. Může rovněž přidat kritérium, které určí, o které skupiny metadat má zájem.

CRCE klientovi pošle vyžádaná metadata komponent (v XML nebo JSON formátu).

Scénář BO-OM: Obtain metadata about ”other”bundles (repo contents diff) - získání metadat o ostatní komponentách Scénář slouží k získání metadat komponent, o kterých klient neví. Klient pošle CRCE seznam identifikátorů komponent, které zná.

CRCE v odpovědi zašle seznam následujících komponent:

- nové komponenty - seznam komponent v jeho úložišti, které nejsou na seznamu klienta
- smazané komponenty - komponenty, které jsou v seznamu komponent klienta a byly smazány z úložiště. (Tento seznam obsahuje pouze identifikátory smazaných komponent, ne ostatní metadata)
- neznámé komponenty - komponenty, které jsou v seznamu zaslaném klientem, ale nejsou v úložišti (neexistuje záznam, že byly smazány). U těchto komponent bude hodnota atributu *crce.status* "unknown".

Metadata pro náhradu

Tato skupina scénářů popisuje situace, kdy klient chce nahradit jeden nebo více svých komponent. Pomocí následujících scénářů získá potřebné informace ve formě metadat o vzájemné kompatibilitě, aby věděl, jaké komponenty si má vyžádat

Scénář SM-RS: Which bundle versions can replace this single one? - Jaká komponenta může nahradit tuto komponentu? Klient chce nahradit jednu ze svých komponent, kterou v žádosti popíše identifikačním číslem. Může rovněž popsat, o jakou operaci náhrady žádá.

CRCE nazpátek pošle metadata (*core* část) o komponentách, které jsou kompatibilní s komponentou určenou v žádosti a jejichž verze splňuje žádanou operaci náhrady.

Operace náhrady jsou následující:

- upgrade [výchozí] = náhrada za vyšší verzi
- lowest = náhrada za nejnižší možnou verzi
- highest = náhrada za nejvyšší možnou verzi
- downgrade = náhrada za nižší verzi
- any = náhrada za jinou verzi

Scénář SM-RA: Which bundle versions can replace this one within an application? Jaká komponenta může nahradit tuto komponentu v rámci aplikace? Scénář je obdobný předchozímu scénáři SM-RS, pouze klient do žádosti přidá kontext aplikace, což jsou metadata o všech komponentách v aplikaci.

CRCE při vyhodnocování musí brát v úvahu, zda komponenta v odpovědi bude vhodná do aplikace (závislosti musí být splněny).

Scénář SM-OS: Which other bundles could replace this one? - Která jiná komponenta může nahradit tuto komponentu? Scénář je obdobný scénáři SM-RS, pouze CRCE hledá pouze kompatibilní komponenty s jiným jménem nebo poskytovatelem.

Scénář SM-OA: Which other bundles can replace this one within an application? - Která jiná komponenta může nahradit tuto komponentu v rámci aplikace? Scénář je obdobný scénáři SM-OA, pouze CRCE misí vzít v úvahu kontext aplikace stejně jako ve scénáři SM-RA.

Scénář SM-FP: Find bundles, that are providers of given capability - Najdi komponenty, které poskytují schopnost Situace, kdy klient má komponentu s požadavkem na schopnost (requirement), který chce uspokojit. Zeptá se tedy úložiště, zda mu může poskytnout komponenty s touto schopností. Žádost od klienta je tedy požadavek na schopnost. Úložiště CRCE poté v odpovědi pošle seznam komponent, které tuto schopnost mají.

Scénář SM-FP: SM-FC: Determine transitive closure satisfying requirements - Urči tranzitivní uzávěr splňující požadavky Klient žádá CRCE o množinu komponent, které uspokojí množinu požadavků (requirement). Klient v žádosti poskytne metadata požadavků a může rovněž přidat hodnotu kritéria, která určí, jaký tranzitivní uzávěr hledat.

Úložiště CRCE vrátí množinu metadat komponent, které dohromady mají vlastnosti splňující požadavky v žádosti a zároveň jsou interně konzistentní (nemají sami žádné nesplněné požadavky).

Hodnota kritéria může být:

- fastest [výchozí] = najde tranzitivní uzávěr co nejrychleji
- minimum = najde takový tranzitivní uzávěr, který obsahuje nejméně komponent

Scénář SM-UA: Metadata for Application Update - Metadata pro aktualizaci aplikace Situace, kdy klient má nainstalovanou konzistentní aplikaci složenou z komponent revizí $b\_1, b\_2, ..., b\_N$ a chce tuto aplikaci aktualizovat.

Klient v žádosti pošle CRCE kompletní popis aplikace, který tvoří metadata všech komponent aplikace.

Úložiště CRCE následně vrátí množinu metadat komponent o revizích $b\_1', b\_2', ..., b\_N'$, kde $b\_i' > b\_i$, tedy konzistentní aplikaci, kde jsou jednotlivé komponenty aktualizovány na vyšší verzi.

6.1.2 Formát metadat

Pro výměnu informací pomocí REST API byl jako výchozí formát vybrán XML. Zvažován byl i JSON, ale převážně z důvodu částečné kompatibility s formátem OBR byl zvolen XML, i přestože jsou v něm ekvivalentní zprávy větší. Návrh formátu metadat vychází ze specifikace OSGi 5 Repository Service, která je popsána v sekci 3.2.1 na straně 12. Formát ale není přebrán beze zbytku, jelikož CRCE má některé jiné požadavky. Ukázku tohoto formátu je možno najít v příloze A na straně 79.

Základní struktura vychází z OSGi 5, kořenovým elementem je **repository**, reprezentující množinu artefaktů z úložiště. Pro artefakty jsou zde elementy **resource**. Element *referral* z OSGi 5 specifikace byl vynechán, jelikož CRCE zatím nepodporuje vícevrstvé úložiště. Element *resource* obsahuje elementy tří druhů:

- capability - schopnost komponenty, převzatá z OSGi 5.
- requirement - požadavek komponenty na schopnost, převzatý z OSGi 5.
- property - pouze popisný údaj (vlastnost), který se nepoužívá ve vyhodnocování závislostí

Elementy *capability*, *requirement* a *property* mohou obsahovat následující elementy:

- attribute - atribut, který je tvořen až čtveřicí hodnot (*name*, *value*, *type* a *op*). Hodnota *type* určuje Java typ hodnoty uložené v elementu *value*. Hodnota *op* byla přidána oproti specifikaci OSGi 5 je to jméno operátoru. Používá se například u atributu *version* požadavku (requirement), kde určuje operátor požadavek na verzi komponenty (nabývá hodnoty porovnávacích operátorů, například greater-than (větší než)).
- directive - pokyn pro Resolver
- link - odkaz (obvykle odkaz na jinou část metadat)
- jiný element s namespace *crce* - Obecně jsou povoleny elementy s jiným namespace. CRCE bude využívat například elementy *crce:filter* pro pokročilé vyhodnocování požadavků nebo *crce:difference* pro výsledky vyhodnocování kompatibility.

Části metadat

Metadata jsou rozdělena do jednotlivých částí. Toho využívá REST API, kdy klient může při určitém dotazu žádat pouze určitá metadata.

Jednotlivé skupiny metadat jsou:

- core = základní metadata o komponentě, tvořeny jsou schopnostmi s názvy končícími řetězcem *.identity* nebo *.content*
- cap = všechny elementy schopností (capability)
- req = všechny elementy požadavků (requirement)
- prop = všechny elementy vlastností (property)

Základní (core) metadata - Základní metadata se v současnosti skládají ze 3 elementů: *osgi.identity*, *osgi.content* a *crce.identity*. Schopnost *osgi.identity* obsahuje základní identifikační údaje komponenty v OSGi, kterými jsou jméno a verze:

```
<capability namespace='osgi.identity'>
     <attribute name="name" value="obcc.parking.gate" />
     <attribute name="version" type="Version" value="2.0.0.build-3622" />
</capability>
```

Schopnost *osgi.content* obsahuje základní informace o souboru s komponentou. Atribut *url* udává URL na komponentu, kterou lze získat pomocí REST API. Následuje ukázka *osgi.content*:

```
<capability namespace='osgi.content'>
     <attribute name="hash" value="---the-SHA-256-hex-encoded-digest-for-this-resource---" />
     <attribute name="url"
        value="http://crce.kiv.zcu.cz/cz/zcu/kiv/obcc/parking/gate/2.0.0" />
     <attribute name="size" type="Long" value="12345" /><!-- in bytes -->
     <attribute name="mime" value="application/vnd.osgi.bundle" />
     <attribute name="crce.original-file-name" value="obcc-parking-example.gate.jar" />
</capability>
```

Schopnost *crce.identity* obsahuje základní identifikaci komponenty v CRCE. Atribut *version.original* obsahuje původní verzi komponenty při vložení do CRCE. Tato verze mohla být následně změněna pomocí automatického verzovacího modulu, který určuje číslo verze na základě rozsahu změn oproti původní verzi. Atribut *crce.categories* obsahuje kategorie, do kterých artefakt zařadil CRCE Indexer. Atribut *crce.status* udává, v jakém úložišti je artefakt uložen. Trvale uložené artefakty ve Store budou mít tento atribut nastaven na "stored". Dalšími možnými hodnotami jsou: "buffer", "deleted", "unknown". Následuje ukázka *crce.identity*:

```
<capability namespace='crce.identity'>
     <attribute name="name" value="obcc-parking-example.gate-2.0.0" />
     <attribute name="resource.type" value="osgi" />
     <attribute name="provider" value="cz.zcu.kiv" />
     <attribute name="version.original" type="Version" value="2.0.0.build-3622" />
     <attribute name="crce.categories" value="initial-version,versioned,osgi"
        type="List<String>" />
     <attribute name="crce.status" value="stored" />
</capability>
```

6.2 Návrh REST API

Návrh REST API byl proveden tak, aby REST API umožňovalo řešení všech scénářů použití. Jednotlivé metody mají vždy uvedeno, jaký scénář použití řeší a jak u nich vypadá žádost a následná odpověď CRCE serveru. U všech následujících metod platí, že předpokládají práci s OSGi komponentami, které jsou uloženy v trvalém úložišti *Store* (Viz sekce 4.2.3 na straně 18). Všechny metody využívají HTTP kódy a vrací je při chybách (400 - interní chyba serveru, 300 - chybně formulovaná žádost, atd.). Metody si vyměňují zprávy v XML formátu, který je popsán v sekci 6.1.2 na straně 53. Tento formát je v dotazech a odpovědích uveden jako:

```
+ XML metadata format
```

6.2.1 Get Bundle

Metoda vrátí OSGi komponentu uloženou v úložišti.

Řeší Scénář: BO-SB

Žádost: Komponenta je určená identifikačním číslem. Alternativně může být určena jménem a verzí nebo pouze jménem (v tom případě je použita nejvyšší dostupná verze).

```
GET /bundle/id
GET /bundle?name=name&version=version
GET /bundle?name=name
```

Odpověď: Vrácen je binární soubor s komponentou. Pokud nebyl nalezen, je vrácen kód 404. Kladná odpověď ze souborem vypadá následovně:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.osgi.bundle
+ bundle
```

6.2.2 Get Metadata

Metoda vrátí metadata o OSGi komponentách.

Řeší Scénář: BO-SM a BO-AM

Žádost: Metoda bez určení vrátí metadata všech komponent. V případě URL vedoucí přímo na id komponenty jsou vrácena metadata této komponenty. Další možností je určit komponenty LDAP filtrem. V tom případě se vrátí metadata těchto komponent.

Nepovinnými parametry lze určit, jaké skupiny metadat (viz 6.1.2 na straně 54) mají být vráceny.

- core [výchozí] - základní metadata
- cap - všechny schopnosti
- cap=name - schopnosti určitého jména (cap=osgi.wiring.package vrátí metadata se všemi požadavky OSGi závislostí)
- req - všechny požadavky
- req=name - požadavky určitého jména
- prop - všechny vlastnosti
- prop=name - vlastnosti určitého jména

Tyto parametry lze kombinovat, tudíž můžeme žádat vrácení metadat všech schopností a zároveň s metadaty požadavků určitého jména.

```
GET /metadata
GET /metadata/id
GET /metadata?filter=(name=xyz)
```

Příklad metody vracející určité skupiny metadat komponent, které jsou určeny LDAP filtrem:

```
GET /metadata?filter=(&(name=examplebundle)(version=1.0))
       &core&cap=osgi.wiring.package
```

Odpověď: Vrácen je soubor s metadaty komponent:

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
+ XML metadata format
```

6.2.3 Other Bundles Metadata

Vrací rozdíl stavu repositáře od stavu předaného v žádosti.

Řeší Scénář: BO-OM

Žádost: Žádost obsahuje seznam komponent, které klient zná. Může též obsahovat parametry určující, které části metadat má server vrátit (stejně jako u předchozí metody GET METADATA).

```
POST other-bundles-metadata HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
+ XML metadata format
```

Odpověď: CRCE v odpovědi zašle seznam následujících komponent:

- nové komponenty - seznam komponent v jeho úložišti, které nejsou na seznamu klienta
- neznámé komponenty - komponenty, které jsou v seznamu zaslaném klientem, ale nejsou v úložišti (neexistuje záznam že byli smazané). U těchto komponent bude hodnota atributu *crce.status* "unknown".

V budoucnu bude vracet i informace o smazaných komponentách, v současnosti si ale CRCE neudržuje záznamy o komponentách, které byly smazány.

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
+ XML metadata format
```

6.2.4 Replace Bundle

Nabídne klientovi náhradu za jeho komponentu

Řeší Scénář: SM-RS

Žádost: Žádost obsahuje identifikaci komponenty a žádaný způsob náhrady (parametr *op*). Hodnoty parametru *op* mohou být následující:

- upgrade [výchozí] = náhrada za vyšší verzi
- lowest = náhrada za nejnižší možnou verzi
- highest = náhrada za nejvyšší možnou verzi
- downgrade = náhrada za nižší verzi
- any = náhrada za jinou verzi

```
GET replace-bundle?id=id(&op=upgrade OR downgrade
        OR lowest OR highest OR any)
```

Odpověď: CRCE zašle metadata o komponentě, která může klientovu komponentu nahradit. V případě že komponenta, kterou klient chce nahradit není na serveru nalezena, je vrácen kód 404. V případě že neexistuje komponenta, která vyhovuje požadované operaci (například při operaci *highest* neexistuje na serveru komponenta s vyšší verzí než ta v žádosti), jsou navráceny metadata komponenty identifikované v žádosti.

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
+ XML metadata format
```

6.2.5 Replace Bundle within Application

Nabídne klientovi náhradu za jeho komponentu kompatibilní v kontextu aplikace.

Řeší Scénář: SM-RA

Žádost: Žádost je obdobná žádosti metody **Replace Bundle**, pouze je použita metoda POST, aby mohl být přenesen i kontext aplikace v XML formátu.

```
POST replace-bundle-context?id=id(&op=upgrade OR downgrade
         OR lowest OR highest OR any) HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
+ XML metadata obsahující kontext aplikace
```

Odpověď: Odpověď má stejnou formu jako u metody **Replace Bundle**. Pokud není nalezena na serveru nějaká komponenta předaná v žádosti, je vrácen kód 404.

6.2.6 Compatible Bundles

Nabídne klientovi kompatibilní náhradu za jeho komponentu, která má jiné jméno či jiného poskytovatele.

Řeší Scénář: SM-OS

Žádost

```
GET compatible-bundles?id=id
```

Odpověď: Odpověď má stejnou formu jako u metody **Replace Bundle**.

6.2.7 Compatible Bundles within Application

Nabídne klientovi kompatibilní náhradu za jeho komponentu v rámci aplikace, která má jiné jméno či jiného poskytovatele.

Řeší Scénář: SM-OA

Žádost:

```
POST compatible-bundles-context?id=id HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
+ XML metadata obsahující kontext aplikace
```

Odpověď Odpověď má stejnou formu jako u metody **Replace Bundle**.

6.2.8 Providers of a Capability

Najde schopnost vyhovující požadavku.

Řeší Scénář: SM-FP

Žádost: Žádost obsahuje XML s požadavkem.

```
POST provider-of-capability/ HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
+ XML metadata požadavku
```

Ukázka XML metadat požadavku:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<requirement namespace='osgi.wiring.package'>
    <attribute name='name'
       value='cz.zcu.kiv.cosi.parkoviste.parkoviste' />
    <attribute name='version' value='1.0.0' op='less-than' />
    <directive name='filter' value=
     '(&(osgi.wiring.package=cz.zcu.kiv.cosi.parkoviste.parkoviste)
     (version&gt;=1.0.0))'/>
</requirement>
```

Odpověď: Odpovědí je seznam komponent, které mají schopnost vyhovující požadavku.

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
+ XML metadata
```

6.2.9 Transitive Closure

Vrátí množinu komponent, která uspokojí množinu požadavků a jejíž komponenty nemají žádné další nevyhodnocené požadavky. Navrácená množina komponent se nazývá tranzitivní uzávěr.

Řeší Scénář: SM-FC

Žádost: Žádost obsahuje množinu požadavků a může obsahovat kritérium, jaký tranzitivní uzávěr zvolit (viz scénář SM-FC).

```
POST transitive-closure/(criterium=fastest OR minimum) HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
+ XML metadata požadavků
```

Odpověď: Odpovědí je seznam komponent tranzitivního uzávěru nebo kód 404, pokud tranzitivní uzávěr nelze vytvořit.

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
+ XML metadata
```

6.2.10 Metadata for Application Update

Klient chce aktualizovat aplikaci a žádá server o seznam komponent, které budou touto aktualizací.

Řeší Scénář: SM-UA

Žádost: Žádost obsahuje informace o komponentách, které tvoří aplikaci.

```
POST /update-application HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
+ XML metadata
```

Odpověď: Odpovědí je seznam komponent, které společně tvoří aktualizaci aplikace.

```
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
+ XML metadata
```

6.3 Technologie

Po návrhu metod a před samotnou implementací bylo třeba učinit rozhodnutí, jaké technologie budou v modulu použity. Bylo potřeba vybrat framework, který usnadňuje vytvoření REST API. Dále bylo třeba vybrat knihovnu pro převod údajů do a z XML formátu, který byl vybrán jako výchozí formát pro komunikaci.

6.3.1 Frameworky pro REST API

Pro realizaci REST API bylo na výběr několik frameworků, některé z nich byly implementací JAX-RS (viz sekce 5.5.1 na straně 47), jiné ne. Příklady některých frameworků pro tvorbu REST API:

- Apache CXF - implementace JAX-RS od Apache. Kromě REST API ji lze využít i k tvorbě SOAP webových služeb, nebo i jiných protokolů.
- Jersey - referenční implementace JAX-RS

- RESTeasy - jBoss implementace JAX-RS
- Restlet - první REST framework, který existoval ještě před JAX-RS. Práce s ním je tedy odlišná. V současnosti má rozšíření, pomocí kterého implementuje JAX-RS.

Vzhledem k tomu, že tyto frameworky implementují stejné API, je práce s nimi takřka totožná. Pro vytvoření REST API modulu CRCE bylo zvoleno použití referenční implementace **Jersey**. Důvodem byla existence přehledné dokumentace na internetových stránkách frameworku.

6.3.2 Převod do XML

Pro vytvoření metadat v XML formátu bylo nutné převádět informace obsažené v třídách Javy do XML a opačně. Porovnání možností prováděl ve své diplomové práci zaměřené na REST klienta CRCE David Švamberk a výsledkem bylo, že nejvhodnější je použití Java Architecture for XML Binding (**JAXB**). Přednosti této knihovny se projevily především při vytváření velkých XML, kde měla JAXB nejlepší výsledky. JAXB umožňuje mapování Java tříd do XML reprezentace. Následně umožňuje obousměrný převod mezi reprezentací dat v Java třídách a reprezentací XML.

6.4 Implementace

Tato sekce obsahuje informace o implementaci REST API CRCE modulu, zaměřuje se zejména na jeho architekturu.

Následující tabulka nabízí přehled scénářů použití, REST API metod které tyto scénáře řeší, základních URI těchto metod a zda metody jsou v současné době implementované:

Scénář	REST API Metoda	URI	implem.
BO-SB	Get Bundle	GET bundle/id	ano
BO-SB	Get Bundle	GET bundle/id	ano
BO-SM	Get Metadata	GET metadata/id	ano
BO-AM	Get Metadata	GET metadata	ano
BO-OM	Other Bundles Metadata	POST other-bundles-metadata	ano
SM-RS	Replace Bundle	GET replace-bundle?id=id	ano
SM-RA	Replace Bundle within Application	POST replace-bundle-context?id=id	ne
SM-OS	Compatible Bundles	GET compatible-bundles?id=id	ne
SM-OA	Compatible Bundles within Application	POST compatible-bundles-context?id=id	ne
SM-FP	Providers of a Capability	POST provider-of-capability	ano
SM-FC	Transitive Closure	POST transitive-closure	ne
SM-FP	Metadata for Application Update	POST update-application	ne

6.4.1 Architektura

Tato sekce popisuje architekturu CRCE REST API modulu.

Přehled balíků

Zde následuje přehled funkcí jednotlivých balíků, do nichž se aplikace člení. Jejich struktura zachycuje základní architekturu modulu.

cz.zcu.kiv.crce.rest.internal.rest Ve výchozím balíku modulu se nacházejí rozhraní definující jednotlivé REST API metody. Název těchto rozhraní vždy začíná názvem použité HTTP metody:

- GetBundle
- GetMetadata
- GetReplaceBundle
- PostOtherBundlesMetadata
- PostProviderOfCapability

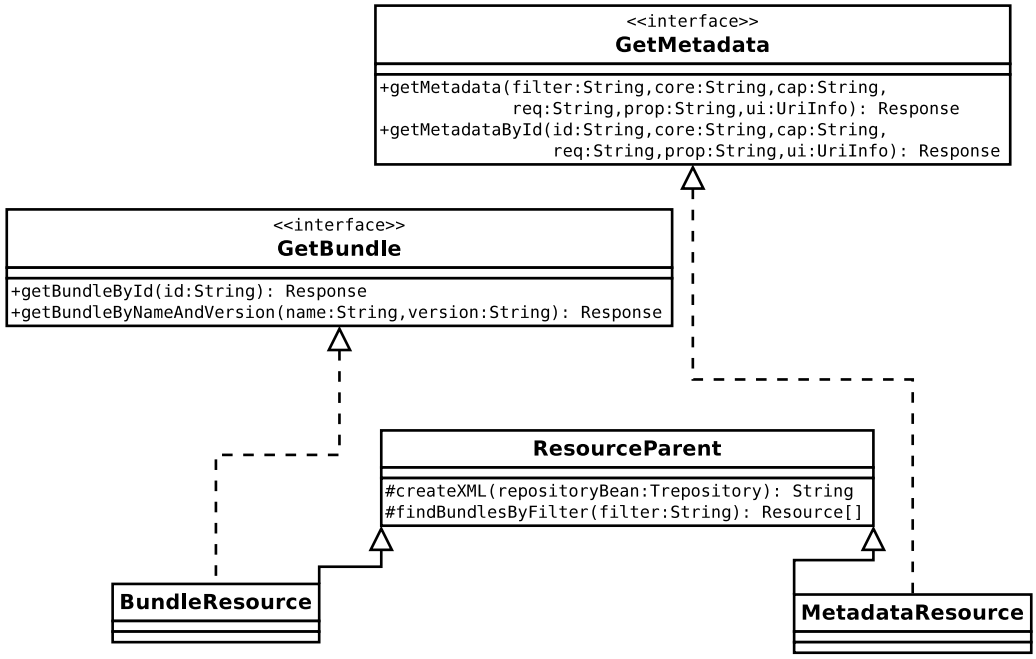
cz.zcu.kiv.crce.rest.internal.rest.convertor Balík obsahující třídy pro převod dat v reprezentaci Java tříd CRCE do JAXB reprezentace, kterou je následně možné pomocí JAXB mapovat na XML formát.

cz.zcu.kiv.crce.rest.internal.rest.generated Balík obsahuje JAXB třídy reprezentující data, které je možné převádět do XML. Tyto třídy byli automaticky vygenerovány z XML Schema souboru, který definuje formát XML popsaný v sekci 6.1.2 na straně 53.

cz.zcu.kiv.crce.rest.internal.rest.structures Balík obsahující pomocné struktury, které jsou využity v ostatních balících.

cz.zcu.kiv.crce.rest.internal.rest.xml Balík obsahující implementaci rozhraní REST API metod využívající pro přenos XML formát.

Implementace metod



Obrázek 6.1: CRCE REST modul - architektura metod REST API

Každá REST API metoda tedy má své rozhraní a následně i implementaci. Implementace REST API metod v balíku *cz.zcu.kiv.crce.rest.internal.rest.xml* jsou vždy potomky třídy *ResourceParent*, která jim předává společné metody. Ukázka této architekturu pro dvě vybrané REST API metody je na obrázku 6.1. Zde je ukázáno, jak metody implementují své

rozhraní a dědí od předka *ResourceParent*. Třída *ResourceParent* jim ve skutečnosti poskytuje více metod, než zde zobrazené 2 ukázkové metody (převod JAXB reprezentace do XML a vyhledávání komponent v úložišti podle filtru).

URI - výsledná URI jednotlivých REST API metod je složena ze jména stroje kde CRCE běží, portu na kterém je puštěný příslušný webový server a přípony *rest/* před všemi URI metod, jak byly navrženy v sekci 6.2 na straně 55. Příkladem budiž URI metody pro získání metadat všech komponent v úložišti za situace, kdy CRCE běží na stejném stroji, ze kterého je poslán dotaz:

```
http://localhost:8080/rest/metadata
```

6.5 Možné rozšíření

Předpokládaným rozšířením modulu do budoucna je především implementace zbylých navržených REST API metod. Tyto metody umožní klientům větší komfort při používání úložiště.

Modul je též připraven o rozšíření počtu podporovaných formátů pro přenos dat. I když formát XML plní veškeré potřeby CRCE a umožňuje kompatibilitu s OSGi Repository Service XML formátem, je možné, že budoucí nároky na provoz budou vyžadovat zmenšení velikosti přenášených zpráv. V tom případě by bylo vhodné, doplnit pro přenos i podporu formátu JSON.

6.6 Ověření funkčnosti

Po implementaci REST API modulu proběhlo nutné ověření funkčnosti jednotlivých metod.

Jednoduché je ověření metod funkčnosti GET metod. U nich stačí zadat URI do prohlížeče. Pomocí tohoto způsoby byla ověřena funkčnost všech implementovaných GET REST API metod.

Pro ověření POST metod je nutné využít některého klienta. Klienta pro

CRCE vytváří v současnosti David Švamberk v souběžné diplomové práci. V době psaní tohoto textu však oficiální klient není hotový. Proto bylo ověření uskutečněno pomocí jednoduchého jednoúčelového klienta, který umožňuje sestavení požadovaného dotazu a zobrazí odpověď. Pro lepší možnost ověření funkčnosti jsou v současné době vytvářeny automatické testy, které budou funkčnost REST API metod kontrolovat.

Ověření POST metod lze rovněž provést pomocí univerzálního REST klienta, kterým je rozšíření pro Mozzila Firefox **Poster**. Tohoto klienta lze stáhnout z následující adresy:

`https://addons.mozilla.org/en-US/firefox/addon/poster/`

Ukázku použití REST API metod a odpovědí na ně najdete v příloze B na straně 87.

7 CRCE - Vytváření informací o kompatibilitě

Vytváření informací o kompatibilitě v úložišti CRCE probíhá pomocí již existujících nástrojů **OBCC** (OSGi Bundle Compatibility Checking).

Základem OBCC je OSGi Bundle Compatibility Checker (nástroj pro kontrolu kompatibility), který zkoumá, zda je bezpečně možné nahradit komponentu jinou (novější verzí). Při nahrazování by se mohlo stát, že poskytovatel některé služby změní svoje rozhraní, na což nebude připraven uživatel služby. Tím by se systém mohl dostat do nefunkčního stavu.

Dalším nástrojem OBCC OSGi Version Generator, který umožňuje automatické určení verze komponenty. Nástroj zjistí rozsah změn oproti předchozí verzi (zda se měnilo některé vnější rozhraní nebo jsou změny jen vnitřní) a podle toho určí nové číslo verze.

Součástí této diplomové práce bylo zprovoznění modulu, který určuje verzi u komponent vkládaných do úložiště.

7.1 Versioning Action Handler

Version Action Handler je CRCE modul, který má na starost určení verze vkládané komponenty. Je umístěn v adresáři *crce-handler-versioning*. Má na starost určení verze komponenty, která je vkládána do repositáře. K tomu mu slouží dvě implementace rozhraní *ActionHandler*:

- IncreaseVersionActionHandler - inkrementace verze
- VersioningActionHandler - určení verze

Rozhraní *ActionHandler* z CRCE Plugin API slouží k realizaci pluginů, které zareagují na některou akci z životního cyklu komponenty v úložišti. Tyto pluginy se mohou například spustit vždy před tím, než je komponenta vložena do trvalého úložiště *Store*. Obě implementace rozhraní *ActionHandler* budou popsány v následujících sekcích.

7.1.1 Prerekvizity modulu

Modul využívá části projektu CRCE, který zase využívá projektu JACC (Java Class Comparator - nástroj pro porovnávání Java tříd). Pro spuštění modulu musí být tyto závislosti splněny.

Jelikož v současnosti nejsou projekty OBCC a JACC dostupné v žádném Maven repositáři, musí před spuštěním dojít k jejich sestavení ze zdrojových kódů do lokálního Maven repositáře. Je třeba si tedy stáhnout pomocí SVN projekty z následujících lokací

```
https://subversion.assembla.com/svn/jacc/trunk/
https://subversion.assembla.com/svn/obcc/trunk/
https://subversion.assembla.com/svn/obcc/file-utils
https://subversion.assembla.com/svn/obcc/version-generator
```

Projekty je nutno v uvedeném pořadí postupně sestavit pomocí příkazu *maven install* spuštěném vždy v kořenovém adresáři projektu (kde je umístěn soubor pom.xml).

7.1.2 IncreaseVersionActionHandler

Tato implementace rozhraní *ActionHandler* slouží k upravení verze u artefaktu, který je vkládán do úložiště, ve kterém již existuje artefakt se stejným jménem a verzí. Pokud tato situace nastane, je připojena na konec kvalifikátoru verze vkládaného artefaktu přípona *_2*. V případě, že již existuje artefakt se stejným jménem i verzí i s touto příponou, je přípona postupně zvyšována, dokud nemá artefakt unikátní verzi mezi artefakty stejného jména. Tato funkce tedy umožňuje, aby byl do úložiště znovu vkládán stejný artefakt.

IncreaseVersionActionHandler je spouštěn pro obě úložiště (Store i Buffer), vždy když je nějaký artefakt vkládán do příslušného úložiště. Pokud tedy při vkládání artefaktu se jménem *exempleArt* a verzí 1.0.0 do úložiště Buffer již v úložišti artefakt s příslušným jménem a verzí existuje, je verze změněna na (1.0.0_2 - předpokládejme, že nyní je v úložišti Buffer v rámci artefaktů se stejným jménem verze unikátní). Při následném vkládání do trvalého úložiště Store však může nastat konflikt jména a verze znovu a verze artefaktu může být změněna například na 1.0.0_4.

7.1.3 VersioningActionHandler

Tento nástroj se poprvé aktivuje před vložením artefaktu do dočasného úložiště Buffer, kdy uloží původní jméno a verzi artefaktu do jeho metadat. Před vložením do stálého úložiště Store pak nástroj provede určení verze komponenty na základě rozdílu od základní komponenty. Základní komponentou je komponenta se stejným jménem, která je již uložena v úložišti (pokud žádná taková neexistuje, určení verze není provedeno). Pokud je v úložišti více komponent se stejným jménem, je zvolena ta z nejvyšší verzí jako základní.

Nástroj Version generator určí verzi na základě změn od základní komponenty. Při určení verze jsou dodržována pravidla OSGi SemanticVersioning [7], kde je verze rozdělena na části major(hlavní), minor(vedlejší), micro(mikro) a qualifier(kvalifikátor). Zápis verze vypadá následovně:

```
version ::= <major> [ '.' <minor> [ '.' <micro> [ '.' <qualifier> ]]]
```

Jednotlivé části odpovídají následujícím změnám:

- major - balíky s odlišnou hlavní (major) verzí, jsou z hlediska poskytovatele i uživatele nekompatibilní
- minor - API uživatele (consumer) je kompatibilní s komponentami, které exportují službu s stejnou major verzi a stejnou nebo vyšší minor verzi. API poskytovatele (provider) je kompatibilní s komponentami, které mají stejnou major i micro verzi. Například verze 1.2 je zpětně kompatibilní s verzí 1.1 pro uživatele, ale nekompatibilní pro poskytovatele.
- micro - Rozdíl v micro části verze nesignalizuje žádné problémy se zpětnou kompatibilitou. Používá se k opravě chyb nebo pro vnitřní změny, které nepostihují API.
- qualifier - Rozdíl v qualifier části verze nesignalizuje žádné problémy se zpětnou kompatibilitou. Obvykle se používá u stejných balíků, které se liší například jen jinou časovou známkou sestavení.

Realizace - Realizace určení verze proběhla za pomoci jež existujícího nástroje Version Generator z OBCC. Po spuštění VersioningActionHandler je

nejprve zkontrolováno, zda již nebyla verze komponenty určována (o čemž existuje zápis v metadatech komponenty). Pokud verze ještě nebyla určována, je vyhledána základní komponenta a pokud tato komponenta existuje, je provedeno volání služby, kterou poskytuje Version Generator.

8 Závěr

Hlavním cílem práce bylo vytvoření rozšíření úložiště CRCE, které umožní interakci úložiště s jeho klienty. Toto rozšíření mělo umožnit získávání artefaktů z úložiště a získávání metadat o artefaktech v úložišti, včetně metadat kontrol kompatibility komponent.

V teoretické části práce jsem se seznámil s úložišti komponent včetně CRCE. Poté jsem zjišťoval možnosti realizace komunikace úložiště s jeho klienty. Pro uskutečnění vzdáleného přístupu bylo zvoleno vytvoření REST API. Jeho výhodou je využívání protokolu HTTP, což umožňuje ovládání úložiště prostřednictvím webu. Jako klienta je možné pro základní metody (využívající HTTP metodu GET) použít jakýkoliv webový prohlížeč. Pátá kapitola této diplomové práce se poměrně rozsáhle věnuje REST stylu architektury a poskytuje návod pro návrh RESTful API.

Scénáře pro komunikaci úložiště CRCE s klienty mi poskytl zadavatel práce Přemek Brada. Na základě těchto scénářů a nastudované literatury jsem vytvořil návrh REST API pro CRCE. Důležité metody navrženého REST API jsem poté implementoval v nově vytvořeném REST modulu pro CRCE. Jelikož klient, který byl vytvářen v rámci souběžné diplomové práce nebyl dokončen včas, byla funkčnost tohoto úložiště otestována pomocí prohlížeče a univerzálního REST klienta *Poster*. Následně jsem zprovozňoval další CRCE modul, který provádí automatické určování verzí komponent pomocí nástroje OBCC.

Přestože se cíl této práce v průběhu trochu vzdálil původnímu zadání, práce alespoň částečně plní všechny body zadání. V rámci práce nakonec nebylo uskutečněno doplnění webového rozhraní úložiště o možnost řízení kontrol kompatibility komponent. Podle informací od zadavatele práce bude v budoucnu pro samotné kontroly kompatibility vypsána ještě jedna diplomová práce.

Seznam zkratek

CRCE - Component Repository supporting Compatibility Evaluation
OBCC - OSGi Bundle Compatibility Checking
OBR - OSGi Bundle Repository
REST - Representational State Transfer
HTTP - Hypertext Transfer Protocol
URI - Uniform resource identifier
XML - Extensible Markup Language
CORBA - Common Object Request Broker Architecture
RPC - Remote procedure call
RMI - Remote method invocation
JVM - Java Virtual Machine
SOAP - Simple Object Access Protocol
XSD - XML Schema Definition
SMTP - Simple Mail Transfer Protocol
RFC - Request for Comments
MIME - Multipurpose Internet Mail Extensions
JSON - JavaScript Object Notation
API - Application programming interface
JAXB - Java Architecture for XML Binding
JaCC - Java Class Comparator
SOA - Service Oriented Architecture

Literatura

[1] Apache Maven Project. [cit. 24.4.2013].
URL http://maven.apache.org/

[2] OSGi Enterprise Release 5. [cit. 22.4.2013].
URL http://www.osgi.org/Download/File?url=/download/r5/osgi.enterprise-5.0.0.pdf

[3] OSGi RFC-0112 Bundle Repository. [cit. 22.4.2013].
URL http://www.osgi.org/download/rfc-0112_BundleRepository.pdf

[4] SOAP Example. [cit. 10.4.2013].
URL http://www.w3schools.com/soap/soap_example.asp

[5] UDDI Version 3.0.2. 2004, [cit. 22.4.2013].
URL http://uddi.org/pubs/uddi_v3.htm

[6] Java Remote Method Invocation. 2010, [cit. 24.4.2013].
URL http://docs.oracle.com/javase/6/docs/platform/rmi/spec/rmiTOC.html

[7] Semantic Versioning. 2010, [cit. 6.5.2013].
URL http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf

[8] UvodDoKomponent. 2010, [cit. 6.4.2013].
URL http://wiki.kiv.zcu.cz/UvodDoKomponent/HomePage

[9] JAX-RS: Java API for RESTfulWeb Services. 2013, [cit. 19.4.2013].
URL http://jcp.org/en/jsr/detail?id=339

[10] Allamaraju, S.: *RESTful Web Services Cookbook.* O'Reilly, 2010, ISBN 978-0-596-80168-7, I-XV, 1-293 s.

[11] Brada, P.: The CoSi Component Model: Reviving the Black-box Nature of Components. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE)*, *Lecture Notes in Computer Science*, ročník 5282, Karlsruhe, Germany: Springer Verlag, October 2008.

[12] Brada, P.; Jezek, K.: Ensuring Component Application Consistency on Small Devices: A Repository-Based Approach. In *38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, editace V. Cortellessa; H. Muccini; O. Demirors, IEEE Computer Society, September 2012, s. 109–116, doi:10.1109/SEAA.2012.48.

[13] Brada, P.; Valenta, L.: Practical Verification of Component Substitutability Using Subtype Relation. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*, IEEE Computer Society Press, 2006, ISSN 1089-6503, s. 38–45, doi: http://doi.ieeecomputersociety.org/10.1109/EUROMICRO.2006.50.

[14] Christensen, E.; Curbera, F.; Meredith, G.; aj.: *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium, 2001, [cit. 10.4.2013].
URL `http://www.w3.org/TR/wsdl`

[15] Councill, B.; Heineman, G. T.: Component-based software engineering. kapitola Definition of a software component and its elements, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN 0-201-70485-4, s. 5–19.
URL `http://dl.acm.org/citation.cfm?id=379381.379438`

[16] Fay-Wolfe, V.; DiPippo, L. C.; Cooper, G.; aj.: Real-Time CORBA. *IEEE Transactions on Parallel and Distributed Systems*, ročník 11, č. 10, 2000: s. 1073–1089, ISSN 1045-9219, doi: http://doi.ieeecomputersociety.org/10.1109/71.888646.

[17] Fielding, R.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, University of California, Irvine, 2000.

[18] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Červen 1999, [cit. 13.4.2013].
URL `http://www.ietf.org/rfc/rfc2616.txt`

[19] Group, O. M.: *CORBA BASICS*. Object Management Group, 2013.
URL `http://www.omg.org/gettingstarted/corbafaq.htm`

[20] Hall, R.; Pauls, K.; McCulloch, S.; aj.: *OSGi in Action: Creating Modular Applications in Java*. Manning Publications, první vydání, Duben 2011, ISBN 1933988916.

[21] Ježek, Kamil and Brada, P.: Formalisation of a Generic Extra-Functional Properties Framework. In *Evaluation of Novel Approaches to Software Engineering*, *Communications in Computer and Information Science*, ročník 275, editace K. Maciaszek, Leszek A. and Zhang, Springer Berlin Heidelberg, 2013, s. 203–21, doi:10.1007/978-3-642-32341-6_14.

[22] Kosek, J.: *Inteligentní podpora navigace na WWW s využitím XML*. Diplomová práce, Vysoká škola ekonomická v Praze, 2002.

[23] Kučera, J.: *Úložiště komponent podporující kontroly kompatibility*. Diplomová práce, Západočeská univerzita v Plzni, 2011.

[24] Nowak, F.; Qasim, M.: A Comparison of Distributed Object Technologies CORBA vs DCOM. 2005.

[25] Peterka, J.: RPC I. 1993, [cit. 8.4.2013].
URL `http://www.earchiv.cz/a93/a350c110.php3`

[26] Reilly, D.: Java RMI a CORBA. 2006, [cit. 12.4.2013].
URL `http://www.javacoffeebreak.com/articles/rmi_corba/`

[27] Richardson, L.; Ruby, S.: *Restful web services*. O'Reilly, první vydání, 2007, ISBN 9780596529260.

[28] Sotomayor, B.: *The Globus Toolkit 3 Programmer's Tutorial*. 2013, [cit. 10.4.2013].
URL `http://gdp.globus.org/gt3-tutorial/multiplehtml/ch01s02.html`

[29] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN 0201745720.

[30] Winer, D.: *XML-RPC Specification*. 1999, [cit. 8.4.2013].
URL `http://xmlrpc.scripting.com/spec`

[31] World Wide Web Consortium: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. 2007, [cit. 10.4.2013].
URL `http://www.w3.org/TR/soap12-part1/`

[32] Řezníček, J.: *Tvorba rozšíření v komponentovém modelu OpenOffice.org.* Bakalářská práce, Západočeská univerzita v Plzni, 2010.

Příloha A - XML formát metadat

Tato příloha obsahuje ukázku metadat komponent v XML formátu. Tento formát je použit k výměně zpráv pomocí REST API. Formát XML je stále ve vývoji, REST API v současnosti používá jeden jeho stav. Předpokládá se, že se bude při dalším vývoji úložiště ještě měnit. Pro definici formátu jsem vytvořil jsem vytvořil XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:crce="TBD-CRCE-METADATA-XSD-URI"
  targetNamespace="TBD-CRCE-METADATA-XSD-URI" elementFormDefault="unqualified"
  attributeFormDefault="unqualified" version="1.0.0">

   <element name="repository" type="crce:Trepository"/>
   <complexType name="Trepository">
      <sequence>
         <choice minOccurs="0" maxOccurs="unbounded">
            <element name="resource" type="crce:Tresource"/>
         </choice>
         <!-- It is non-deterministic, per W3C XML Schema 1.0:
         http://www.w3.org/TR/xmlschema-1/#cos-nonambig
         to use name space="##any" below. -->
         <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string">
         <annotation>
            <documentation xml:lang="en">
               The name of the repository. The name may contain
               spaces and punctuation.
            </documentation>
         </annotation>
      </attribute>
      <attribute name="increment" type="long">
         <annotation>
            <documentation xml:lang="en">
               An indication of when the repository was last changed. Client's can
               check if a
               repository has been updated by checking this increment value.
            </documentation>
         </annotation>
      </attribute>
```

```
   <anyAttribute/>
</complexType>

<complexType name="Tresource">
   <annotation>
      <documentation xml:lang="en">
         Describes a general resource with
         requirements and capabilities.
      </documentation>
   </annotation>
   <sequence>
      <element name="requirement" type="crce:Trequirement" minOccurs="0"
         maxOccurs="unbounded"/>
      <element name="capability" type="crce:Tcapability" minOccurs="1"
         maxOccurs="unbounded"/>
      <element name="property" type="crce:Tproperty" minOccurs="0"
         maxOccurs="unbounded"/>
      <!-- It is non-deterministic, per W3C XML Schema 1.0:
      http://www.w3.org/TR/xmlschema-1/#cos-nonambig
      to use name space="##any" below. -->
      <any namespace="##other" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
   </sequence>
   <attribute name="id" type="string" form="qualified">
      <annotation>
         <documentation xml:lang="en">
            Crce id of the resource.
         </documentation>
      </annotation>
   </attribute>
</complexType>



<complexType name="Tcapability">
   <annotation>
      <documentation xml:lang="en">
         A named set of type attributes and directives. A capability can be
         used to resolve a requirement if the resource is included.
      </documentation>
   </annotation>
   <sequence>
      <choice minOccurs="0" maxOccurs="unbounded">
         <element name="directive" type="crce:Tdirective"/>
         <element name="attribute" type="crce:Tattribute"/>
         <element name="capability" type="crce:Tcapability"/>
      </choice>
      <!-- It is non-deterministic, per W3C XML Schema 1.0:
      http://www.w3.org/TR/xmlschema-1/#cos-nonambig
      to use name space="##any" below. -->
      <any namespace="##other" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
   </sequence>
   <attribute name="namespace" type="string">
      <annotation>
         <documentation xml:lang="en">
            Name space of the capability. Only requirements with the
            same name space must be able to match this capability.
         </documentation>
      </annotation>
   </attribute>
   <anyAttribute/>
```

```
   </complexType>

   <complexType name="Trequirement">
      <annotation>
         <documentation xml:lang="en">
            A filter on a named set of capability attributes.
         </documentation>
      </annotation>
      <sequence>
         <choice minOccurs="0" maxOccurs="unbounded">
            <element name="directive" type="crce:Tdirective"/>
            <element name="attribute" type="crce:Tattribute"/>
            <element name="requirement" type="crce:Trequirement"/>
         </choice>
         <!-- It is non-deterministic, per W3C XML Schema 1.0:
         http://www.w3.org/TR/xmlschema-1/#cos-nonambig
         to use name space="##any" below. -->
         <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="namespace" type="string">
         <annotation>
            <documentation xml:lang="en">
               Name space of the requirement. Only capabilities within the
               same name space must be able to match this requirement.
            </documentation>
         </annotation>
      </attribute>
      <anyAttribute/>
   </complexType>

   <complexType name="Tproperty">
<annotation>
         <documentation xml:lang="en">
            A filter on a named set of property attributes.
         </documentation>
      </annotation>
      <sequence>
         <choice minOccurs="0" maxOccurs="unbounded">
            <element name="directive" type="crce:Tdirective"/>
            <element name="attribute" type="crce:Tattribute"/>
            <element name="link" type="crce:Tlink"/>
            <element name="property" type="crce:Tproperty"/>
         </choice>
         <!-- It is non-deterministic, per W3C XML Schema 1.0:
         http://www.w3.org/TR/xmlschema-1/#cos-nonambig
         to use name space="##any" below. -->
         <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="namespace" type="string">
         <annotation>
            <documentation xml:lang="en">
               Name space of the property.
            </documentation>
         </annotation>
      </attribute>
      <anyAttribute/>
   </complexType>

   <complexType name="Tattribute">
      <annotation>
```

```
      <documentation xml:lang="en">
         A named value with an optional type that decorates
         a requirement or capability.
      </documentation>
   </annotation>
   <sequence>
      <any namespace="##any" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
   </sequence>
   <attribute name="name" type="string">
      <annotation>
         <documentation xml:lang="en">
            The name of the attribute.
         </documentation>
      </annotation>
   </attribute>
   <attribute name="value" type="string">
      <annotation>
         <documentation xml:lang="en">
            The value of the attribute.
         </documentation>
      </annotation>
   </attribute>
   <attribute name="type" type="string">
      <annotation>
         <documentation xml:lang="en">
            The type of the attribute.
         </documentation>
      </annotation>
   </attribute>
   <attribute name="op" type="string">
      <annotation>
         <documentation xml:lang="en">
            The operation of the attribute.
         </documentation>
      </annotation>
   </attribute>
   <anyAttribute/>
</complexType>

<complexType name="Tdirective">
   <annotation>
      <documentation xml:lang="en">
         A named value of type string that instructs a resolver
         how to process a requirement or capability.
      </documentation>
   </annotation>
   <sequence>
      <any namespace="##any" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
   </sequence>
   <attribute name="name" type="string">
      <annotation>
         <documentation xml:lang="en">
            The name of the directive.
         </documentation>
      </annotation>
   </attribute>
   <attribute name="value" type="string">
      <annotation>
         <documentation xml:lang="en">
            The value of the directive.
```

```
            </documentation>
         </annotation>
      </attribute>
      <anyAttribute/>
   </complexType>

   <complexType name="Tlink">
      <annotation>
         <documentation xml:lang="en">
            A link.
         </documentation>
      </annotation>
      <sequence>
         <any namespace="##any" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
      </sequence>
      <attribute name="rel" type="string">
         <annotation>
            <documentation xml:lang="en">
               relationship
            </documentation>
         </annotation>
      </attribute>
      <attribute name="type" type="string">
         <annotation>
            <documentation xml:lang="en">
               type
            </documentation>
         </annotation>
      </attribute>
      <attribute name="name" type="string">
         <annotation>
            <documentation xml:lang="en">
               name
            </documentation>
         </annotation>
      </attribute>
      <anyAttribute/>
   </complexType>

</schema>
```

Následující ukázka obsahuje předpokládaná plná data o jedné komponentě. Je to ukázka, jak by formát mohl vypadat, až bude jeho vývoj dokončen. Vytvořena byla zadavatelem práce Přemkem Bradou:

```
<repository name='CRCE Repository' increment='13582741' xmlns="TBD-CRCE-METADATA-XSD-URI">

<resource crce:id='obcc-parking-example.gate-2.0.0' >

    <!-- this is the primary identification of the resource in CRCE -->
    <capability namespace='crce.identity'>
      <attribute name="name" value="obcc-parking-example.gate-2.0.0" />
      <attribute name="resource.type" value="osgi" />
      <link rel="detail" type="capability" name="osgi.identity" />
      <attribute name="provider" value="cz.zcu.kiv" />
      <attribute name="version.original" type="Version" value="2.0.0.build-3622" />
      <!-- CRCE classification put here as they can be resolved-against -->
      <attribute name="crce.categories" value="initial-version,versioned,osgi"
```

```
  type="List<String>" />
  <attribute name="crce.status" value="stored" />
  <!-- buffer / stored / removed (or similar, according to lifecycle) -->
</capability>

<!-- component type-dependent information -->
<capability namespace='osgi.identity'>
  <!-- 'name' attribute is used instead of OSGi R5 repetition of namespace -->
  <!-- 'type' attribute denotes the language type name, e.g. Java class name;
       is used by resolver to interpret the cap/req data,
       only primitive types + typed collections are allowed for
       capabilities/reqts (as in OSGi) -->
  <!-- the 'name' and 'type' attribute names are "well known" -->
  <attribute name="name" value="obcc.parking.gate" />
  <attribute name="version" type="Version" value="2.0.0.build-3622" />
</capability>
<capability namespace='osgi.content'>  <!-- (from original CRCE "capability: file")
  <attribute name="hash" value="---the-SHA-256-hex-encoded-digest--" />
   <!-- name='osgi.content' for OBR -->
  <attribute name="url"
     value="http://crce.kiv.zcu.cz/cz/zcu/kiv/obcc/parking/gate/2.0.0" />
  <attribute name="size" type="Long" value="12345" /><!-- in bytes -->
  <attribute name="mime" value="application/vnd.osgi.bundle" />
    <!-- mime for unknown: application/octet-stream -->
  <attribute name="crce.original-file-name"
     value="obcc-parking-example.gate.jar" />
</capability>
<capability namespace='osgi.wiring.package'>
  <attribute name='name' value='cz.zcu.kiv.obcc.parking.gate.stats' />
  <attribute name='version' type='Version' value='2.0.0' />
  <directive name='uses' value='cz.zcu.kiv.obcc.parking.gate.base' />
</capability>

<!-- how component-wide extra functional properties
     according to EFFCC schema are declared -->
<capability namespace='effcc.efp'>
  <attribute name='name' value='GR.memory.allocation' />
  <attribute name='value' value='LR.desktop.low' type="cz.zcu.kiv.effcc.types.Long" />
</capability>

<!-- recursive capability declaration for hierarchical structures -->
<capability namespace='osgi.runtime.service'>
  <attribute name='name' value='base-stats-1' />
  <attribute name='type' value='cz.zcu.kiv.obcc.parking.gate.stats.BaseStatistics' />
  <capability namespace='effcc.efp' >
    <attribute name='name' value='GR.throughput.cars' />
    <attribute name='value' value="=LR.carpark.high" type="cz.zcu.kiv.effcc.types.Int" />
  </capability>
  <capability namespace='effcc.efp' >
    <attribute name='name' value="GR.throughput.trucks" />
    <attribute name='value' value="LR.carpark.low" type="cz.zcu.kiv.effcc.types.Int" />
  </capability>
</capability>

<!-- requirements ie dependencies of the current resource -->

<requirement namespace='osgi.wiring.package'>
  <attribute name='name' value='cz.zcu.kiv.obcc.example.carpark.arrivals' />
  <attribute name='version' value='1.0.0' op='greater-than' />
     <!-- ^^^ "op"s are: greater-than, less-than, greater-equal,
          less-equal, equal, not-equal -->
  <!-- the OSGi representation of the above via LDAP filter,
```

```
        as direct resolver directive -->
  <directive name='filter' value=
  '(&(osgi.wiring.package=cz.zcu.kiv.obcc.example.carpark.arrivals)(version&gt;=1.0.0))'/>
</requirement>

<!-- referencing hierarchical structures, declared by recursive capability elements -->
<requirement namespace='xyz.service'>
  <attribute name='xyz.service' value='bundleA::cz.zcu....stats::svc1' />
  <!-- unambiguous -->
  <attribute name='xyz.service' value='svcXyz' /> <!-- ambiguous, more candidates -->
  <!-- override of default 'and'-ing of attributes in the requirement -->
  <directive name='operator' value='or' />
</requirement>

<!-- requirements with attribute filters (keeping attributes as first-class citizens) -->
<requirement namespace='osgi.wiring.package'>
  <attribute name='name' value="cz.zcu.kiv.obcc.example.carpark.arrivals" />
  <crce:filter op='or'>
  <!-- this filter says "version in <1.0.0, 2.0.0) or equal to 0.2.1" -->
    <crce:filter op='and'>
      <attribute name='version' value='1.0.0' crce:op='greater-equal'/>
      <attribute name='version' value='2.0.0' crce:op='less-than'/>
    </crce:filter>
    <attribute name='version' value='0.2.1' />  <!-- default 'op' is "equal" -->
  </crce:filter>
</requirement>

<!-- nested requirements, i.e. we want a given component
which has 1.2.13 version and contains a given bean -->
<requirement namespace='sofa.component'>
  <attribute name='name' value="daoComponent" />
  <crce:filter>  <!-- op='and' by default -->
    <attribute name='version' value='1.2.13.build1299' />
    <requirement namespace='sofa.component.bean'>
      <attribute name='name' value='CorePosDao' />
    </requirement>
  </crce:filter>
</requirement>

<!-- descriptive properties -->

<property namespace='crce.description'>
  <attribute name="crce.description"
     value="An example bundle from the Parking Place app." />
</property>

<!-- results modeled as crce properties referencing the related capabilities/requirements,
     which corresponds to the Results meta-model -->
<property namespace='crce.result'>
  <attribute name="name" value="obcc-parking--2.0.0--loadtest" />
  <!-- refers to crce.identity -->
  <attribute name="uri" value=
  "http://crce.kiv.zcu.cz/rest/obcc-parking--2.0.0--loadtest.log" type="java.net.Url" />
  <!-- CRCE properties allow structured types, unlike caps/reqts -->
  <attribute name="creator" value="cz.zcu.kiv.crce.plugin.simco.runner" />
  <attribute name="creator-version" value="1.5.0" />
  <link rel="reason" type="capability" name="1.memory.allocation" />
  <link rel="reason" type="capability" name="base-stats-1::GR.throughput.cars" />
  <!-- references sub-capability -->
</property>
```

```
    <property namespace="crce.compatibility" >
      <attribute name="predecessor-version" value="1.2.6" />

      <!-- below, 'base' refers to osgi.identity:version / crce.identity:version capability
           of the resource with the same osgi.identity / crce.identity value;
           the difference/diff 'value' is computed as the change from
           that base to the current resource -->

      <crce:difference base="1.2.6" contract="extra-func" syntax="EFFCC" value="SPEC" />
        <!-- TODO supply detailed diff example for EFFCC -->

      <crce:difference base="1.2.6" contract="signature" syntax="java" value="MUT" >
        <crce:result>
          <attribute name="name" value="obcc-parking--1.2.6-vs-2.0.0--diff" />
          <!-- refers to crce.identity -->
          <attribute name="uri" value=
          "http://crce.kiv.zcu.cz/data/results/obcc-parking--1.2.6-vs-2.0.0--diff.log" />
          <attribute name="creator" value="cz.zcu.kiv.crce.plugin.obcc.crce-comparator" />
          <attribute name="creator-version" value="1.0.1" />
        </crce:result>
        <!-- the generic tree structure of 'diff's is used because
          it fits any contract syntax comparison result.
             "capability"/"requirement" references the given element on the resource "surface".
             "part" value depends on the contract syntax used. -->
        <crce:diff part="osgi.wiring.package" name="cz.zcu.kiv.obcc.parking.gate.stats"
        type="capability" value="SPEC" />
        <crce:diff part="osgi.wiring.package" name="cz.zcu.kiv.obcc.parking.gate.base"
        type="capability" value="GEN" >
          <attribute name='version' value='2.6.0' />
          <!-- if we need to reference a concrete variant of the capability -->
          <crce:diff part="class" name="SomeClass" value="GEN">
            <crce:diff part="operation" name="myConstructor" value="GEN" />
          </crce:diff>
          <crce:diff part="class" name="cz.zcu.kiv.obcc.parking.gate.base.OtherClass"
             value="GEN" />
        </crce:diff>
        <crce:diff part="osgi.wiring.package" name="cz.zcu.kiv.obcc.parking.carpark.arrivals"
        type="requirement" value="GEN" />
      </crce:difference>

      <crce:difference base="1.2.0" contract="signature" syntax="java" value="SPEC" >
        <crce:diff part="osgi.wiring.package" name="cz.zcu.kiv.obcc.parking.gate.stats"
        type="capability" value="SPEC" >
          <crce:diff part="class" name="OtherClass" value="SPEC">
            <crce:diff part="operation" name="newOperation" value="SPEC" />
          </crce:diff>
        </crce:diff>
      </crce:difference>

    </property>

  </resource>
</repository>
```

Příloha B - Ukázka použití REST modulu CRCE

Tato příloha obsahuje ukázky použití REST modulu CRCE. V úložišti bylo v době používání 5 komponent.

Get Metadata

První použití metody získáme základní OSGi identifikace komponent v úložišti. Metoda se aktivuje GET http dotazem na URI:

```
http://localhost:8080/rest/metadata?cap=osgi.identity
```

Výsledkem je vrácené XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI">
    <resource ns2:id="testJar.jar-0.0.0">
        <capability namespace="osgi.identity">
            <attribute name="name" value="testJar.jar"/>
            <attribute name="version" value="0.0.0" type="Version"/>
        </capability>
    </resource>
    <resource ns2:id="ParkovisteOsgiParkoviste-0.9.0">
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="0.9.0" type="Version"/>
        </capability>
    </resource>
    <resource ns2:id="ParkovisteOsgiParkoviste-1.0.0.5">
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="1.0.0.5" type="Version"/>
        </capability>
    </resource>
    <resource ns2:id="ParkovisteOsgiParkoviste-1.0.0.1">
```

```
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="1.0.0.1" type="Version"/>
        </capability>
    </resource>
    <resource ns2:id="ParkovisteOsgiParkoviste-2.0.1.1">
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="2.0.1.1" type="Version"/>
        </capability>
    </resource>
</ns2:repository>
```

Kompletní metadata o jedné komponentě včetně URI pro její stažení lze získat metodou s URI:

```
http://localhost:8080/rest/metadata/ParkovisteOsgiParkoviste-0.9.0
```

Výsledkem následně je:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI">
    <resource ns2:id="ParkovisteOsgiParkoviste-0.9.0">
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
               value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
            <attribute name="name" value="org.osgi.service.event"/>
            <attribute name="version" value="1.2.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
               value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
            <attribute name="name" value="org.osgi.framework"/>
            <attribute name="version" value="1.3.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
               value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
        </requirement>
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="0.9.0" type="Version"/>
        </capability>
        <capability namespace="osgi.content">
            <attribute name="hash"
             value="f360984836f877e2ad7fc18787f1b2c86e37f791155358af69054aff0980eae3"/>
            <attribute name="url"
             value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-0.9.0"/>
            <attribute name="size" value="5673" type="Long"/>
            <attribute name="mime" value="application/vnd.osgi.bundle"/>
            <attribute name="crce.original-file-name" value="OSGi_without_EFP2.jar"/>
        </capability>
        <capability namespace="crce.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste-0.9.0"/>
            <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
```

```
            <attribute name="crce.status" value="stored"/>
        </capability>
        <capability namespace="osgi.wiring.package">
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
            <attribute name="version" value="0.0.0" type="Version"/>
        </capability>
    </resource>
</ns2:repository>
```

Replace Bundle

Metodu pro nahrazení komponenty lze aktivovat například příkazem:

```
http://localhost:8080/rest/replace-bundle?id=ParkovisteOsgiParkoviste-1.0.0.1&op=highest
```

V tomto případě žádáme o nahrazení komponenty se jménem *ParkovisteOsgiParkoviste* a verzí *1.0.0.1* komponentou s nejvyšší dostupnou verzí. Výsledkem je XML s metadaty této komponenty:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI">
    <resource ns2:id="ParkovisteOsgiParkoviste-2.0.1.1">
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
            <attribute name="name" value="org.osgi.service.event"/>
            <attribute name="version" value="1.2.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
            <attribute name="name" value="org.osgi.framework"/>
            <attribute name="version" value="1.3.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
        </requirement>
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="2.0.1.1" type="Version"/>
        </capability>
        <capability namespace="osgi.content">
            <attribute name="hash"
             value="d0b7aa75d0db7c04b74ccb0b7d1cc0b6944e1258e179563d546207fc0977d93c"/>
            <attribute name="url"
             value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-2.0.1.1"/>
            <attribute name="size" value="5675" type="Long"/>
            <attribute name="mime" value="application/vnd.osgi.bundle"/>
            <attribute name="crce.original-file-name" value="OSGi_without_EFP3.jar"/>
        </capability>
```

```
        <capability namespace="crce.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste-2.0.1.1"/>
            <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
            <attribute name="crce.status" value="stored"/>
        </capability>
        <capability namespace="osgi.wiring.package">
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
            <attribute name="version" value="0.0.0" type="Version"/>
        </capability>
    </resource>
</ns2:repository>
```

Other Bundles Metadata

Metodu otestujeme odesláním žádosti HTTP metodou POST na adresu:

```
http://localhost:8080/rest/other_bundles_metadata
```

Žádost obsahuje XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI" name="testRep">
    <resource ns2:id="testJar.jar-0.0.0"/>
</ns2:repository>
```

Odpovědí je XML, které ukazuje rozdíl úložiště oproti stavu úložiště uvedeného v žádosti:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI">
    <resource ns2:id="ParkovisteOsgiParkoviste-0.9.0">
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
            <attribute name="name" value="org.osgi.service.event"/>
            <attribute name="version" value="1.2.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
            <attribute name="name" value="org.osgi.framework"/>
            <attribute name="version" value="1.3.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
        </requirement>
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
```

```
        <attribute name="version" value="0.9.0" type="Version"/>
    </capability>
    <capability namespace="osgi.content">
        <attribute name="hash"
         value="f360984836f877e2ad7fc18787f1b2c86e37f791155358af69054aff0980eae3"/>
        <attribute name="url"
         value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-0.9.0"/>
        <attribute name="size" value="5673" type="Long"/>
        <attribute name="mime" value="application/vnd.osgi.bundle"/>
        <attribute name="crce.original-file-name" value="OSGi_without_EFP2.jar"/>
    </capability>
    <capability namespace="crce.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste-0.9.0"/>
        <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
        <attribute name="crce.status" value="stored"/>
    </capability>
    <capability namespace="osgi.wiring.package">
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
        <attribute name="version" value="0.0.0" type="Version"/>
    </capability>
</resource>
<resource ns2:id="ParkovisteOsgiParkoviste-1.0.0.5">
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
        <attribute name="name" value="org.osgi.service.event"/>
        <attribute name="version" value="1.2.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
        <attribute name="name" value="org.osgi.framework"/>
        <attribute name="version" value="1.3.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
    </requirement>
    <capability namespace="osgi.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste"/>
        <attribute name="version" value="1.0.0.5" type="Version"/>
    </capability>
    <capability namespace="osgi.content">
        <attribute name="hash"
         value="b0fc8021d1a16482ba7c423631c674a19eb6a4694ffcef4512ceb72bd3a50030"/>
        <attribute name="url"
         value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-1.0.0.5"/>
        <attribute name="size" value="5674" type="Long"/>
        <attribute name="mime" value="application/vnd.osgi.bundle"/>
        <attribute name="crce.original-file-name" value="OSGi_without_EFP1.jar"/>
    </capability>
    <capability namespace="crce.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste-1.0.0.5"/>
        <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
        <attribute name="crce.status" value="stored"/>
    </capability>
    <capability namespace="osgi.wiring.package">
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
        <attribute name="version" value="0.0.0" type="Version"/>
    </capability>
</resource>
```

```
<resource ns2:id="ParkovisteOsgiParkoviste-1.0.0.1">
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
        <attribute name="name" value="org.osgi.service.event"/>
        <attribute name="version" value="1.2.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
        <attribute name="name" value="org.osgi.framework"/>
        <attribute name="version" value="1.3.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
    </requirement>
    <capability namespace="osgi.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste"/>
        <attribute name="version" value="1.0.0.1" type="Version"/>
    </capability>
    <capability namespace="osgi.content">
        <attribute name="hash"
         value="6a40356e478508e92544bd17dccb2ef142899e3460a1cfae07d1fdeb52a3781c"/>
        <attribute name="url"
         value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-1.0.0.1"/>
        <attribute name="size" value="5674" type="Long"/>
        <attribute name="mime" value="application/vnd.osgi.bundle"/>
        <attribute name="crce.original-file-name" value="OSGi_without_EFP.jar"/>
    </capability>
    <capability namespace="crce.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste-1.0.0.1"/>
        <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
        <attribute name="crce.status" value="stored"/>
    </capability>
    <capability namespace="osgi.wiring.package">
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
        <attribute name="version" value="0.0.0" type="Version"/>
    </capability>
</resource>
<resource ns2:id="ParkovisteOsgiParkoviste-2.0.1.1">
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
        <attribute name="name" value="org.osgi.service.event"/>
        <attribute name="version" value="1.2.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
        <attribute name="name" value="org.osgi.framework"/>
        <attribute name="version" value="1.3.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
    </requirement>
    <capability namespace="osgi.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste"/>
        <attribute name="version" value="2.0.1.1" type="Version"/>
```

```
        </capability>
        <capability namespace="osgi.content">
            <attribute name="hash"
             value="d0b7aa75d0db7c04b74ccb0b7d1cc0b6944e1258e179563d546207fc0977d93c"/>
            <attribute name="url"
             value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-2.0.1.1"/>
            <attribute name="size" value="5675" type="Long"/>
            <attribute name="mime" value="application/vnd.osgi.bundle"/>
            <attribute name="crce.original-file-name" value="OSGi_without_EFP3.jar"/>
        </capability>
        <capability namespace="crce.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste-2.0.1.1"/>
            <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
            <attribute name="crce.status" value="stored"/>
        </capability>
        <capability namespace="osgi.wiring.package">
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
            <attribute name="version" value="0.0.0" type="Version"/>
        </capability>
    </resource>
</ns2:repository>
```

Provider of Capability

Metodu otestujeme odesláním žádosti HTTP metodou POST na adresu:

```
http://localhost:8080/rest/provider-of-capability
```

Žádost obsahuje XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<requirement namespace='osgi.wiring.package'>
<attribute name='name' value='cz.zcu.kiv.cosi.parkoviste.parkoviste' />
<attribute name='version' value='1.0.0' op='less-than' />
<directive name='filter' value=
  '(&amp;(osgi.wiring.package=cz.zcu.kiv.cosi.parkoviste.parkoviste)(version&gt;=1.0.0))'/>
</requirement>
```

Odpovědí je XML, které obsahuje metadata komponent, které nabízí schopnost:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI">
    <resource ns2:id="ParkovisteOsgiParkoviste-0.9.0">
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
               value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
            <attribute name="name" value="org.osgi.service.event"/>
            <attribute name="version" value="1.2.0" op="greater-equal"/>
```

```
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
        <attribute name="name" value="org.osgi.framework"/>
        <attribute name="version" value="1.3.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
    </requirement>
    <capability namespace="osgi.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste"/>
        <attribute name="version" value="0.9.0" type="Version"/>
    </capability>
    <capability namespace="osgi.content">
        <attribute name="hash"
         value="f360984836f877e2ad7fc18787f1b2c86e37f791155358af69054aff0980eae3"/>
        <attribute name="url"
         value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-0.9.0"/>
        <attribute name="size" value="5673" type="Long"/>
        <attribute name="mime" value="application/vnd.osgi.bundle"/>
        <attribute name="crce.original-file-name" value="OSGi_without_EFP2.jar"/>
    </capability>
    <capability namespace="crce.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste-0.9.0"/>
        <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
        <attribute name="crce.status" value="stored"/>
    </capability>
    <capability namespace="osgi.wiring.package">
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
        <attribute name="version" value="0.0.0" type="Version"/>
    </capability>
</resource>
<resource ns2:id="ParkovisteOsgiParkoviste-1.0.0.5">
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
        <attribute name="name" value="org.osgi.service.event"/>
        <attribute name="version" value="1.2.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
        <attribute name="name" value="org.osgi.framework"/>
        <attribute name="version" value="1.3.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
    </requirement>
    <capability namespace="osgi.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste"/>
        <attribute name="version" value="1.0.0.5" type="Version"/>
    </capability>
    <capability namespace="osgi.content">
        <attribute name="hash"
         value="b0fc8021d1a16482ba7c423631c674a19eb6a4694ffcef4512ceb72bd3a50030"/>
        <attribute name="url"
         value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-1.0.0.5"/>
```

```
        <attribute name="size" value="5674" type="Long"/>
        <attribute name="mime" value="application/vnd.osgi.bundle"/>
        <attribute name="crce.original-file-name" value="OSGi_without_EFP1.jar"/>
    </capability>
    <capability namespace="crce.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste-1.0.0.5"/>
        <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
        <attribute name="crce.status" value="stored"/>
    </capability>
    <capability namespace="osgi.wiring.package">
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
        <attribute name="version" value="0.0.0" type="Version"/>
    </capability>
</resource>
<resource ns2:id="ParkovisteOsgiParkoviste-1.0.0.1">
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
        <attribute name="name" value="org.osgi.service.event"/>
        <attribute name="version" value="1.2.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
        <attribute name="name" value="org.osgi.framework"/>
        <attribute name="version" value="1.3.0" op="greater-equal"/>
    </requirement>
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
    </requirement>
    <capability namespace="osgi.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste"/>
        <attribute name="version" value="1.0.0.1" type="Version"/>
    </capability>
    <capability namespace="osgi.content">
        <attribute name="hash"
          value="6a40356e478508e92544bd17dccb2ef142899e3460a1cfae07d1fdeb52a3781c"/>
        <attribute name="url"
          value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-1.0.0.1"/>
        <attribute name="size" value="5674" type="Long"/>
        <attribute name="mime" value="application/vnd.osgi.bundle"/>
        <attribute name="crce.original-file-name" value="OSGi_without_EFP.jar"/>
    </capability>
    <capability namespace="crce.identity">
        <attribute name="name" value="ParkovisteOsgiParkoviste-1.0.0.1"/>
        <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
        <attribute name="crce.status" value="stored"/>
    </capability>
    <capability namespace="osgi.wiring.package">
        <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
        <attribute name="version" value="0.0.0" type="Version"/>
    </capability>
</resource>
<resource ns2:id="ParkovisteOsgiParkoviste-2.0.1.1">
    <requirement namespace="osgi.wiring.package">
        <directive name="filter"
          value="(&amp;(package=org.osgi.service.event)(version&gt;=1.2.0))"/>
        <attribute name="name" value="org.osgi.service.event"/>
        <attribute name="version" value="1.2.0" op="greater-equal"/>
    </requirement>
```

```
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=org.osgi.framework)(version&gt;=1.3.0))"/>
            <attribute name="name" value="org.osgi.framework"/>
            <attribute name="version" value="1.3.0" op="greater-equal"/>
        </requirement>
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
              value="(&amp;(package=cz.zcu.kiv.cosi.parkoviste.konfigurace))"/>
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.konfigurace"/>
        </requirement>
        <capability namespace="osgi.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste"/>
            <attribute name="version" value="2.0.1.1" type="Version"/>
        </capability>
        <capability namespace="osgi.content">
            <attribute name="hash"
              value="d0b7aa75d0db7c04b74ccb0b7d1cc0b6944e1258e179563d546207fc0977d93c"/>
            <attribute name="url"
              value="http://localhost:8080/rest/bundle/ParkovisteOsgiParkoviste-2.0.1.1"/>
            <attribute name="size" value="5675" type="Long"/>
            <attribute name="mime" value="application/vnd.osgi.bundle"/>
            <attribute name="crce.original-file-name" value="OSGi_without_EFP3.jar"/>
        </capability>
        <capability namespace="crce.identity">
            <attribute name="name" value="ParkovisteOsgiParkoviste-2.0.1.1"/>
            <attribute name="crce.categories" value="zip,osgi" type="List&lt;String&gt;"/>
            <attribute name="crce.status" value="stored"/>
        </capability>
        <capability namespace="osgi.wiring.package">
            <attribute name="name" value="cz.zcu.kiv.cosi.parkoviste.parkoviste"/>
            <attribute name="version" value="0.0.0" type="Version"/>
        </capability>
    </resource>
</ns2:repository>
```