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

Diplomová práce

Integrace úložiště CRCE s Maven repository

Plzeň, 2015 Miroslav Brožek

Originál zadání

Prohlášení

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

V Plzni dne 13. května 2015 Miroslav Brožek

Poděkování

Ze všeho nejvíce bych rád poděkoval panu Doc. Ing. Přemyslu Bradovi, MSc., Ph.D. a Ing. Jiřímu Kučerovi za jejich ochotu, vstřícnost, čas strávený nad všemi otázkami a za všechny podklady k práci.

Děkuji svým prarodičům, svému otci Ing. Milanu Brožkovi a jeho rodině za neutuchající podporu v dobách těžkých i lepších a za poskytnutí zázemí k dosažení vzdělání a slušného postavení ve společnosti.

Abstrakt

Téměř každý programátor musí řešit otázku – kam a jak uložit různé druhy komponenty, podprogramy, balíky a ostatní soubory. Existuje mnoho variant řešení a každé má různé použití, výhody a nevýhody. Nejznámější široce známé Maven úložiště v Java ekosystému je od Apache Software Foundation. Cílem této práce je integrace experimentálního CRCE úložiště s lokálním a vzdáleným Maven úložištěm. Výsledkem této diplomové práce je OSGi plugin pro CRCE, který je schopný stáhnout a zpracovat artefakty z maven úložišť a uložit jejich meta-data do CRCE databáze. To umožňuje vývojářům získat cenné informace o požadovaných komponentách a v případě potřeby poskytnout artefakt ihned k dispozici.

Abstract

Almost every software developer needs to solve the question where and how to store different kind of components, subprograms, bundles and other files. There are many ways of doing this, each of them has different use cases, advantages and disadvantages. One of the most commonly known Maven repository in the Java ecosystem is from the Apache Software Foundation. The purpose of this work is the integration of the experimental CRCE repository with local and remote Maven repository. The result of this thesis is an OSGi-based plugin for CRCE, which is able to fetch and process artifacts from maven repositories and store their meta-data into the CRCE database. This enables to provide developers valuable information regarding specific components, and if needed, artifacts can be accessible immediately.

Obsah

1 Úvod 1

2 Komponentové programování 2

2.1 Komponenta 2
2.2 Komponentový model 3
2.3 OSGi 3

3 Úložiště komponent 8

3.1 CRCE 8
3.2 Maven 13
3.3 Maven Repository 19

4 Vyhledávání v Maven Repository 25

4.1 Nexus Indexer 25
4.2 Apache Lucene 26
4.3 Maven indexer 32
4.4 Aether 34

5 CRCE modul pro Maven repository 37

5.1 Specifikace 37
5.2 Návrh komponenty 45
5.3 Implementace 50
5.4 Testy a dosažené výsledky 57
5.5 Možnosti rozšíření 63

6 Závěr 65

Příloha A – Uživatelská příručka 68

Příloha B – Konfigurační soubor 72

Příloha C – ER diagram entity Resource 74

Příloha D - Návrh GUI pro maven repository 75

Příloha E – Obsah POM 76

Příloha F – Obsah přiloženého media 81

1 Úvod

Existují různé hmotné i nehmotné věci, které jsou pro každého z nás důležité a ty se snažíme uchovávat. Jedny z nehmotných věcí jsou právě také programy (subprogramy) a komponenty. Nejeden zdrojový kód či binární soubor má cenu milionů korun. V počátcích rozvoje výpočetní techniky a informačních technologií byla data rozložena na různých místech, dokonce i tam, kde by je nikdo nečekal. S postupem času přicházely různé koncepty, jak programy či komponenty efektivně uchovávat a spravovat. Nejjednodušším způsobem je určení vyhrazeného místa (úložiště), kam se všechny soubory budou ukládat.

Úložiště lze zjednodušeně chápat jako sklad věcí. Očekáváme tedy od něj veškeré běžné operace s uloženými prvky, mezi které patří přidání, odebrání, prohledání či získání popisu objektu. Úložiště by mělo splňovat nejrozšířenější standardy, čímž dosáhneme komplexního využití a širší distribuci. V dnešní době existuje několik skupin vydávající různé frameworky a programátorské konvence. Mezi největší z nich nepochybně patří komunita Apache. Právě ona vydala softwarový nástroj Apache Maven, který se rozletěl do světa programátorů v roce 2002.

Součástí projektu Apache Maven je úložiště artefaktů a jejich správa - Maven Repository (úložiště), které je stavebním kamenem při výrobě nejrůznějších programů. V těchto úložištích mohou být uloženy programy, komponenty a jiné soubory, kterým se také říká artefakty. Existují dvě varianty úložiště: lokální či vzdálené. Oba typy mají stejnou strukturu, což má za následek usnadnění práce a větší efektivitu.

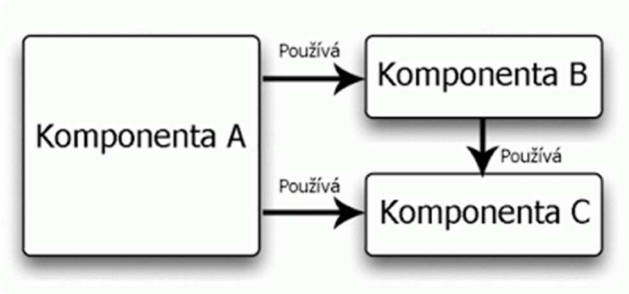
Z důvodu globálního rozšíření nástroje Maven a jeho repository je cílem této práce integrace aplikace CRCE s tímto typem úložišť. Projekt CRCE je rozšiřitelné experimentální úložiště, založené na OSGi Bundle Repository, k ukládání nejen softwarových komponent a kontrolování jejich kompatibility.

Integrace Maven Repository značně rozšíří využití CRCE a ulehčí správu potřebných artefaktů. Uživatelé aplikace budou mít tak přehled o uložených komponentách v požadovaných úložištích a budou je moci rychle a jednoduše získat. CRCE bude jednak využívat novou implementaci Maven Repository, ale také zůstane zachována možnost použití původního file-based úložiště.

Tato diplomová práce se zabývá zpracováním artefaktů uložených v Maven úložištích. V kapitole 2 budou nejprve vysvětleny všechny pojmy spojené s ukládáním a vyhledáváním komponent. Kapitola 3 je věnována úložišti komponent, kde je popsána také aplikace CRCE. Poslední teoretická část – kapitola 4 se věnuje vyhledávání v úložištích. Následně se v kapitole 5.1 čtenář seznámí se specifikací požadavků, které bude CRCE plugin poskytovat. Po implementaci nového pluginu, popsané v kapitole 5.4 bude mít čtenář v závěru práce možnost prostudovat výsledky aplikace.

2 Komponentové programování

Je-li funkcionalita programu rozdělena na nezávislé funkční jednotky, bavíme se o komponentově orientovaném programování. Pomocí objektů programátor vytvoří samostatnou komponentu, která splňuje několik zásadních vlastností. Ostatní vývojáři poté mají možnost sestavovat další komponenty pomocí jiných komponent, které spolu dokáží komunikovat a spolupracovat.



Obr. 2 - 1 Sestavení komponent

2.1 Komponenta

Softwarová komponenta je jednotka určená pro skládání do větších celků se specifickými rozhraními a pouze vnějším kontextem závislostí. Softwarová komponenta může být rozmístěna nezávisle a může být využita třetí stranou [SCCS]. Komponenta by měla splňovat následující vlastnosti.

2.1.1 Identifikace

Jednotka musí být jednoznačně identifikovatelná nejen v aplikaci ale třeba i v operačním systému. Díky tomu mohou existovat stejné komponenty s různým identifikačním číslem.

2.1.2 Zapouzdření

Vnitřní funkcionalita komponenty by měla být pro vnější svět neviditelná. Komponenty komunikují pomocí zpráv svého rozhraní.

2.1.3 Rozhraní

Komponenta poskytuje své veřejné rozhraní k její obsluze. Vnitřní mechanismy a algoritmy jsou skryté.

2.1.4 Připravenost k použití

Balík by měl být nezávisle nasaditelný a plně funkční. Má-li klient k němu přístup, měl by být schopen okamžitě využívat jeho služby pomocí veřejně přístupného API.

2.1.5 Znuvupoužitelnost

Je-li prvek vyvinutý, odladěný, otestován a optimalizován, můžeme ho použít tolikrát, kolikrát potřebujeme. Smyslem je značný nárůst produktivity vývojářů pomocí opakované použitelnosti jednou vytvořené softwarové entity.

2.2 Komponentový model

Komponentový model specifikuje, co je to komponenta, jak probíhá komunikace mezi komponentami, jak se komponenty skládají, jaké jsou povolené typy komponent, požadované služby, atd. Jde tedy o abstraktní definice struktur a pravidel, jako je tomu například u specifikací OSGi.

2.3 OSGi

OSGi (Open Service Gateway Initiative) je modulární systém, používaný v jazyce Java. Jeho dominantou je možnost správy jednotlivých modulů aplikace za běhu komponentového kontejneru. Model OSGi se vyskytuje ve více implementacích. Např. Eclipse Equinox, Concierge či Apache Felix. Právě poslední zmíněná implementace byla použita pro CRCE [RZ12].

Technologie OSGi [OIA11] tvoří soubor specifikací, které definují dynamický systém komponent pro platformu Java a poskytují modulární architekturu rozsáhlých distribuovaných systémů. OSGi umožňuje komponentizaci softwarových modulů a aplikací, zajišťuje vzdálenou správu a interoperabilitu aplikací či služeb. Při použití OSGi modulů lze dosáhnout zvýšení produktivity vývoje ve všech jeho fázích.

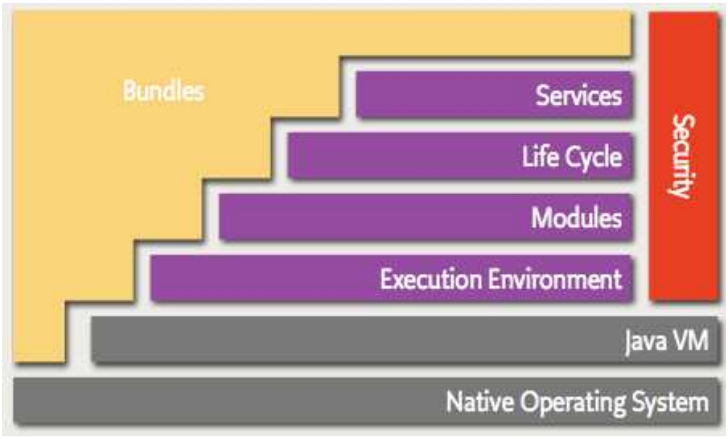
Jednotlivé moduly aplikace běžící v komponentovém kontejneru tvoří samostatné funkční celky. V prostředí OSGi se modul nazývá bundle. Toto pojmenování bude opakovaně používáno napříč prací. Bundly představují základní koncept, umožňující dekompozici systému. Modularitu lze stručně chápat, jako logické rozložení velkého systému do menších funkčních spolupracujících části.

Vlastnosti OSGi bundlů jsou specifikovány v jejich manifest souboru MANIFEST.MF. Tento soubor moduly nutně potřebují, pro možnost jejich provozování v OSGi kontejneru. V následující kapitole o architektuře OSGi je věnován odstavec modules, který popisuje samotný bundle detailněji.

2.3.1 OSGi Architektura

VRSTVENÍ

OSGi Framework implementuje model vrstvení:



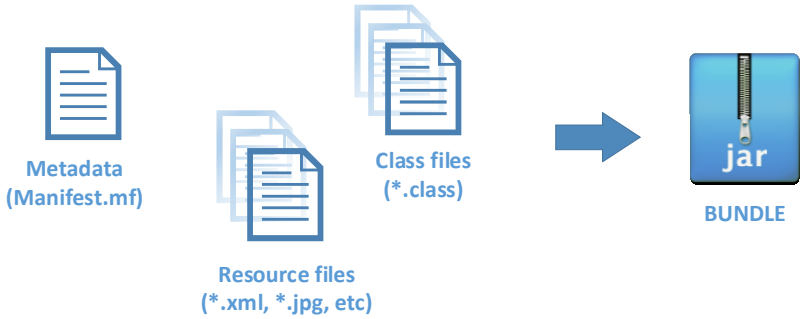
Obr. 2.3.1 - 1 Vrstvení OSGi [OSGA]

Bundles – jsou OSGi komponenty tvořeny vývojáři
Services – připojují dynamicky komponenty pro POJO objekty
Life Cycle – API pro instalaci, start, stop, aktualizaci a odinstalaci komponent
Modules – Vrstva definující jak může komponenta importovat nebo exportovat kód
Security – Vrstva starající se o bezpečnost
Execution Enfironment – Definuje metody a třídy dostupné pro danou platformu

MODULES

Tato vrstva definuje koncept OSGi modulu nazývaného bundle, což je JAR soubor s extra metadaty (data o datech). Bundle obsahuje třídy a jejich tzv. resources, viz obrázek 2.3.1 – 2. Bundle není celá aplikace zabalená do JAR souboru. Je to logická jednotka, která může explicitně deklarovat, které obsažené balíky modul poskytuje k veřejnému použití pro další moduly. Deklarace se zapisuje do manifest souboru pod hlavičku – exported packages.

Další výhodou bundlu oproti standartnímu JAR souboru, je to, že může explicitně definovat v manifestu, které externí balíky potřebují pro svojí činnost. Tyto závislosti jsou definované pod hlavičkou imported packages.



Obr. 2.3.1 – 2 Obsah bundlu

Výhoda explicitních definicí exported a imported packages je ta, že OSGi Framework dokáže řídit a ověřit jejich konzistenci automaticky. Tento proces se nazývá bundle resolution.

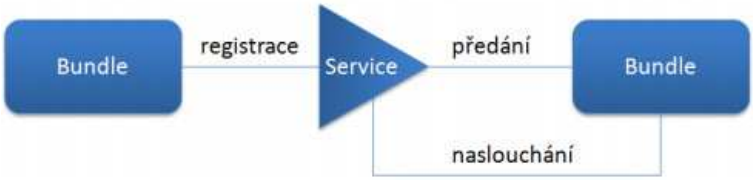
Manifest soubor obsahuje řadu důležitých atributů, ale za zmínku stojí ještě atribut – Bundle Activator. Aktivátor je rozhraní, které umožňuje definovat, jaké akce proběhnou před spuštěním a zastavením komponenty. Pro každou komponentu může existovat pouze jeden.

Ukázka obsahu souboru MANIFEST.MF:

Manifest-Version: 1.0
Bnd-LastModified: 1430488883272
Build-Jdk: 1.7.0_75
Built-By: M.Brozek
Bundle-Activator: cz.zcu.kiv.crce.plugin.internal.Activator
Bundle-Description: Component Repository supporting Compatibility Evaluation
Bundle-DocURL: http://www.kiv.zcu.cz
Bundle-ManifestVersion: 2
Bundle-Name: CRCE - Core - Plugin API
Bundle-SymbolicName: cz.zcu.kiv.crce.plugin
Bundle-Vendor: ZČU KIV
Bundle-Version: 2.1.1.SNAPSHOT
Created-By: Apache Maven Bundle Plugin
Export-Package: cz.zcu.kiv.crce.plugin;version="2.1.1.SNAPSHOT";
uses:="javax.annotation,org.osgi.framework,org.osgi.service.cm"
Import-Package: cz.zcu.kiv.crce.plugin;version="[2.1,3)",….

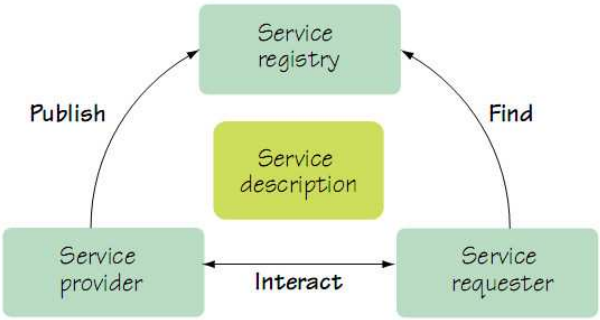
SERVICES

Bundle může vytvořit objekt a zaregistrovat ho u OSGi registru služeb jedním či více rozhraním. Ostatní komponenty mohou získat od registru seznam všech objektů, které jsou zaregistrovány pod specifickým rozhraním či třídou. Jakýkoliv bundle může zaregistrovat stejný typ služby a jakýkoliv počet komponent může získat tu samou službu.



Obr. 2.3.1 - 3 OSGi Services [OSGA]

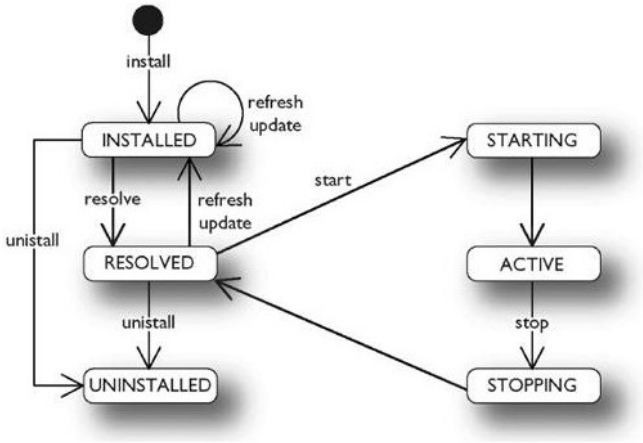
Základní vztah poskytování a hledání služeb v registru je vyjádřen na obrázku 2.3.1 – 4. Registr na základě požadavků vrací seznam služeb, které je možno použít. Stejně tak je schopen zaregistrovat novou službu a poskytnout ji ostatním modulům. Jedná se o tzv. service-oriented architekturu – SOA [OIA11].



Obr. 2.3.1 – 4 SOA [OIA11]

DEPLOYMENT

Komponenty jsou nasazeny do OSGi frameworku – bundle runtime prostředí. Neplést s kontejnerem jako v Java EE verzi. Jedná se o kolaborativní prostředí. Balíky běží pod stejným virtuálním strojem a mohou i sdílet kód. Framework používá jednoduché API [OSGA], které dovoluje komponenty instalovat, spouštět, vypínat či aktualizovat (Obr. 2.3.1 - 5).



Obr. 2.3.1 - 5 OSGi – Životní cyklus komponenty

LIFE CYCLE

Installed: Balík byl úspěšně nainstalován. Framework má veškeré informace a může se jej pokusit nahrát.

Resolved: Všechny zdroje potřebné pro tento balík byly nahrány úspěšně a balík je připraven ke spuštění. Do tohoto stavu se také ještě jednou dostane, je-li úspěšně zastaven.

Starting: Balík začíná nabíhat, ale ještě není dokončeno spuštění.

Active: Balík byl úspěšně aktivován, běží a je připraven k použití.

Stoping: Balík byl zastaven, ale ještě není zcela ukončen.

Uninstalled: Balík byl odinstalován. Jakmile je odebrán, s modulem nelze provádět žádné operace [OSFX10].

3 Úložiště komponent

Pojem komponenta a systém využívající komponenty jsme objasnili. Nyní je potřeba definovat, co znamená úložiště komponent. Tato kapitola vysvětluje, co je CRCE úložiště, co znamená Maven a jaká existují Maven úložiště neboli tzv. repository.

Obecné úložiště odkazuje na centrální místo, kde jsou uložena různorodá data. Může se jednat o dočasné nebo permanentní uskladnění. Nad obsahem úložiště, lze provádět různé operace, vkládat či vyjímat prvky apod. Podle obsahu úložiště (repository) můžeme definovat různé typy „skladů“ jako například:

- Úložiště software (CVS, SVN, Git)
- Databázové úložiště (Oracle DB, MongoDB)
- Úložiště komponent (CRCE, Maven Repository, Archiva)

3.1 CRCE

CRCE, v plném znění Component Repository supporting Compatibility Evaluation, je dlouhodobý projekt vyvíjený pod záštitou Katedry informatiky ZČU. Ačkoliv stojí na základech frameworku OSGi (kapitola 2.3) a je směřován především na vkládání OSGi komponent, lze do něj ukládat jakékoliv balíky, artefakty či moduly.

Hlavní úlohou úložiště je schopnost udržovat komponenty se zabudovanou kontrolou jejich kompatibility. Ke každému uskladněnému prvku jsou uchovávány další související informace jako XML metadata popisující komponenty, jejich vzájemné vazby a závislostí, jenž vycházejí z návrhu OBR úložiště – kapitola 3.1.2.

Tato sdružená data jsou k dispozici v databázi. Metadata si lze představit jako doplňující informace k uloženým prvkům. Jsou to data k datům. CRCE je schopné s těmito daty efektivně pracovat a vyhodnocovat jejich obsah.

Úložiště je navrženo jako modulární systém. Jednotlivé moduly zastávají samostatnou funkci a lze je vypnout nebo zapnout podle preferencí za běhu programu spolu s moduly třetích stran.

Základními vlastnostmi CRCE jsou:

a) **Rozšiřitelnost** – snadné přidání funkčnosti pomocí pluginů, díky technologii OSGi.

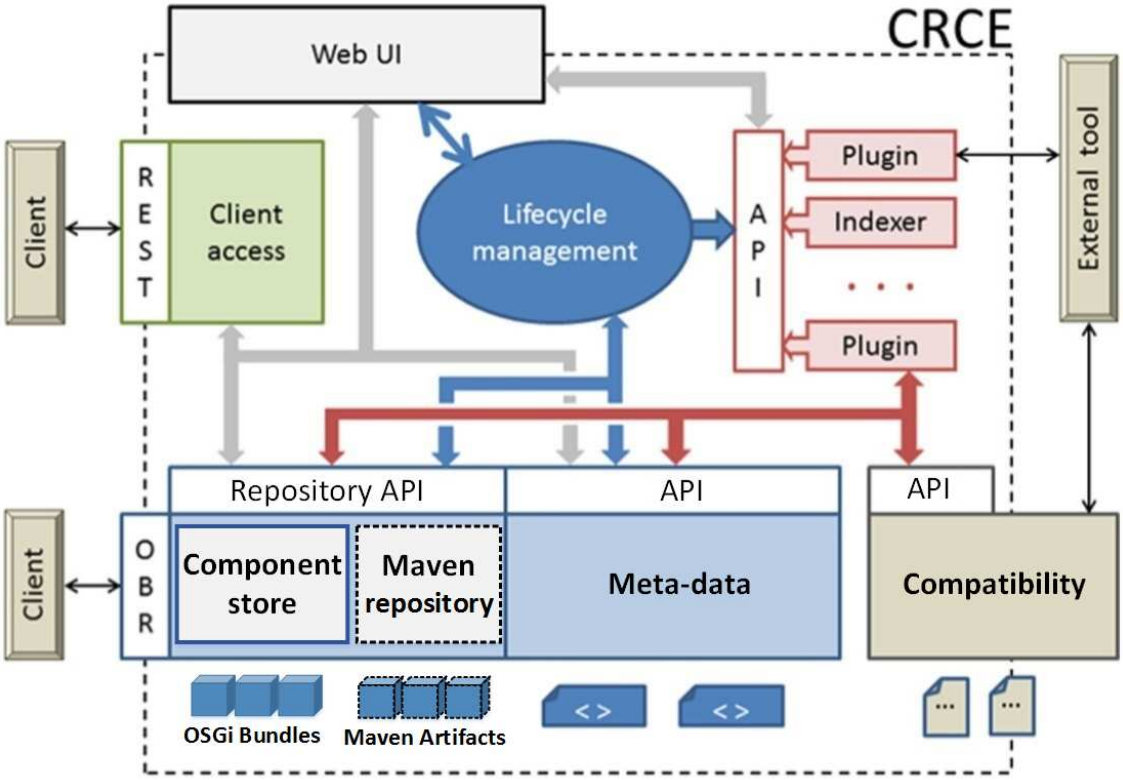
b) **Popisná metadata** – Umožňují popsání komponent, jejich závislostí, testů kompatibility, atd.

Uživatel má k dispozici následující funkce [RJ13]:

- Vkládání komponent do úložiště
- Získání seznamu uložených balíků
- Stažení komponenty z úložiště
- Získání metadat vybraného prvku
- Zpracování artefaktu během jeho životního cyklu

3.1.1 Architektura CRCE

CRCE je modulární projekt založený na OSGi balíkách, které lze různě přidávat či odebírat. Hlavní funkční celky mají své API definované v následujících kapitolách. Čárkovaná komponenta, Maven úložiště, je cíl této práce, která je popsána v kapitole 5. Component store, kterému se také říká File-based repository. Slouží k ukládání OSGi komponent a je mu věnován odstavec Repository API. Maven repository i File-based repository využívají rozhraní Store.



Obr. 3.1.1 - 1 CRCE architektura [BRJZ]

METADATA API

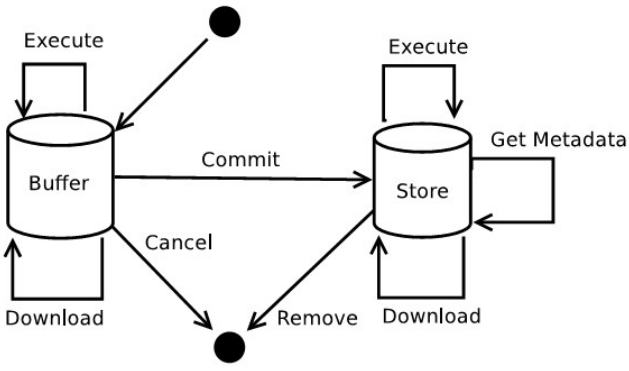
Rozhraní pro definici entit, struktura je převzata z OSGi Bundle Repository (dále jen OBR) (kapitola 3.1.2.). Tento modul se stará o obsluhu metadat komponenty. Umožňuje přidávat dodatečné elementy attribute do elementů capability, requirement a property. CRCE navíc přináší možnost rekurzivně deklarovat vnořené výše uvedené tři typy elementů. Obsahuje tato důležitá rozhraní [RJ13]:

- **Repository** – reprezentuje úložiště artefaktů
- **Resource** – vyjadřuje komponentu a její metadata
- **Capability** – popisuje schopnosti artefaktu, které může ostatním poskytovat
- **Requirement** – vyjadřuje požadavky, na kterých je komponenta závislá
- **Property** – vlastnost určité entity
- **Type** – definuje typ vlastnosti
- **Reason** – spojuje Requirement a Resource entity
- **Resolver** – vyhodnocuje závislosti mezi přidanými Resource

REPOSITORY API

Toto API definuje pro stávající implementaci OBR dva druhy úložišť - Buffer a Store. Buffer slouží jako vstupní krok, mezi-úložiště, kdy je možné s artefakty provádět různé operace před samotným finálním uložením do Store repository. Implementace CRCE pluginu pro Maven repository bude také využívat rozhraní Store viz kapitola 5.3.

Proces průchodu komponenty do OBR je znázorněn níže na obrázku 3.1.1. – 2.



Obr. 3.1.1 - 2 CRCE životní cyklus komponenty [RJ13]

REPOSITORY DAO API

Rozhraní Repository DAO slouží k načítání a ukládání metadat artefaktů. Obsluhuje pouze samotná metadata, nikoliv artefakty. Metadata jsou uchovávána v embedded H2 databázi. Zde se nachází prvky typu Resource, Requirement, Capability a další přidružené entity. Kompletní ER diagram je definován v příloze C. K identifikaci jednotlivých souborů se používá jejich URI.

WEB UI

Tento prvek vytváří webové uživatelské rozhraní, které umožňuje provádět základní operace s úložištěm CRCE. Vestavěný webový server vytváří webové stránky a obstarává jejich akce. Hlavní schopnosti modulu jsou [RJ13]:

- Zobrazení uložených artefaktů
- Zobrazení a editace metadat
- Vkládání a mazání artefaktů
- Spouštění testů kompatibility
- Zobrazení nainstalovaných pluginů

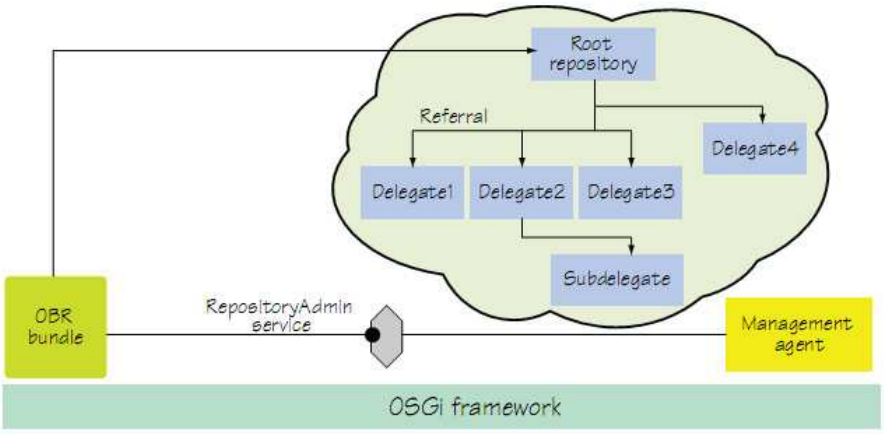
3.1.2 OSGi Bundle Repository - OBR

Dosavadní úložiště v CRCE využívá OSGi Bundle Repository (**OBR**), které není oficiálně považováno za specifikaci standardu. Řekněme, že se jedná spíše o návrh specifikace, který je k nalezení pod RFC 112 v OSGi Alliance. Tato kapitola se týká ukládání metadat – viz architektura CRCE , obr. 3.1.1 – 1, Meta-data API.

První verze OBR1 se nazývala Oscar Bundle Repository, která byla spojena s Oscar OSGi frameworkem, ze kterého se stal Apache Felix Framework. OBR je primárně cílené k řešení dvou aspektů:

- **Discovery** – poskytuje jednoduchý mechanismus k objevení modulů dostupných pro nasazení
- **Dependency deployment –** poskytuje mechanismu nasazení modulu jeho tranzitivní množinu závislostí

K dosažení prvního cíle OBR definuje jednoduché bundle repository s rozhraním k jeho přístupu a obecný XML formát zahrnuje popis prvků. Jedno OBR úložiště může odkazovat na jiná OBR úložiště. Jedním z hlavním cílů OBR byla jednoduchost. Výhodou úložiště založeného na XML formátu je, že není zapotřebí žádného serveru k jeho zpracování, i když tato varianta také připadá v úvahu [OIA11].

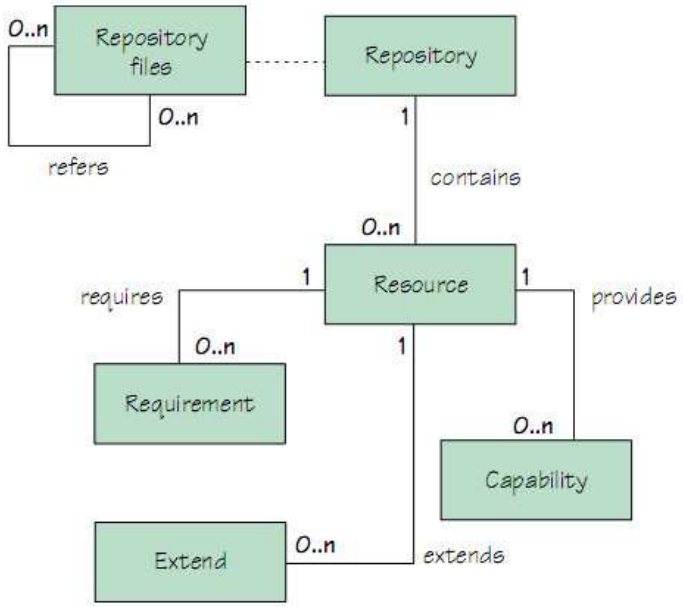


Obr. 3.1.2 - 1 Struktura OBR úložiště [OIA11]

Klíčový koncept OBR úložiště je obecný popis prvků a jejich závislostí. Prvek je abstraktní entita, reprezentující jakýkoliv typ artefaktu, např. bundle, certifikát, konfigurační soubor. Každý deskriptor prvku obsahuje:

- Žádný nebo více požadavků na jiný prvek či prostředí
- Žádné nebo více schopností, použité k uspokojení ostatních požadavků prvků

OBR mapuje informace z manifest souboru bundlu – kapitola 2.3.1 do metadat komponenty. Údaje z hlaviček Import-Package a Require-Bundle do požadavků prvků - Requirement. Z Export-Package a BundleSymblicName hlaviček mapuje na schopnosti prvku - Capability. Vztahy mezi entitami jsou znázorněny na následujícím obrázku [OIA11].



Obr. 3.1.2 - 2 Vztahy mezi OBR entitami [OIA11]

3.1.3 OSGi Manifest Indexer

CRCE vytváří výše uvedená metadata OSGi bundlu pomocí CRCE třídy OsgiManifestBundleIndexer. Tato třída implementuje rozhraní ResourceIndexer, jenž má definovanou metodu:

```
index (InputStream input, Resource resource)
```

Parameter input reprezentuje bundle soubor a resource vytvořenou entitu k dané komponentě. OsgiManifestBundleIndexer projde JAR soubor, přečte informace z manifest souboru a na základě získaných dat vytvoří entity odpovídající schématu na obrázku 3.1.2 – 2 a uloží metadata do databáze.

3.1.4 Nevýhody stávajícího file-based repository pluginu

O manipulaci s fyzickými soubory komponent se doposud staral plugin crce-repository-impl. Ten naráží na několik úskalí, jenž stručně vystihneme. Veškeré zpracované komponenty se ukládají do stejného adresáře bez jakékoliv adresářové struktury. Komponenty nejsou dle očekávání typu JAR, ale jsou převedeny do jiného typu souboru, který není snadné přečíst. Další nevýhodou je nutnost vkládání komponent jednotlivě. Není možné vložit více bundlů najednou. Poslední výraznou nevýhodou je fakt, že úložiště neumí zpracovávat Maven artefakty, což je cílem této práce.

3.2 Maven

Pojmy jako úložiště, komponenta a úložiště komponent jsou vysvětlené. Abychom mohli definovat pojem Maven repository, je nejprve nutné popsat, co znamená a k čemu slouží nástroj Apache Maven.

3.2.1 Maven definice

Nejčastější definice mezi uživateli je [MVNDG]: Maven je moderní nástroj pro sestavení (build) softwarových projektů. Projektoví manažeři naopak označují Maven jako projektový nástroj. Maven umí také generovat webové stránky či různé reporty. Je to univerzální nástroj, který si najde uživatele v širokém spektru.

FORMÁLNÍ DEFINICE

Maven je nástroj pro správu projektů, který zahrnuje objektový model projektu (POM soubor), soubor norem, životní cyklus projektu, systém řízení závislostí, a logiku pro provádění plugin úkonů ve stanovených fázích životního cyklu [MVNCR].

MAVEN ARTEFAKT

Artefakt je soubor, většinou typu JAR, který je nahrán do Maven úložiště. Maven proces sestavování vyprodukuje jeden či více artefaktů, jako například zkompilované JAR soubory, či JAR se zdrojovými kódy. Každý artefakt má své identifikátory – groupId, artifactId a verzi. Tyto tři parametry se také zapisují ve zkratce – GAV a tvoří unikátní identifikátor artefaktu. Více informací k tématu se lze dočíst v kapitole 3.2.3.

Výhodou Mavenu je možnost používat nepřeberné množství pluginů použitých např. při sestavování projektu. Díky tomuto nástroji lze z projektu sestavit JAR soubor nebo WAR (Web Application Resource), či vygenerovat třídy pro JAXB (práce s XML) nebo pro JAXWS (webové služby) a to až v průběhu sestavování. Dalším mocným mechanismem je práce s knihovnami. Stačí jen na určitou knihovnu či modul aplikace přidat závislost a Maven je automaticky přidá k projektu.

HLAVNÍ POUŽITÍ

- Usnadnění procesu sestavení (build) softwaru
- Poskytnutí jednotného sestavovacího systému
- Poskytnutí kvalitních projektových informací
- Vede ke správným programovacím praktikám
- Umožnění snadného získání nových vlastností, pomocí pluginů

Ačkoliv Maven neeliminuje nutnost znalosti sestavování systému, hodně detailů dokáže odstínit. Maven dovoluje sestavení projektů pomocí projektových objektových modelů (POM souborů), kterým je věnována kapitola 3.2.2. Mimo jiné poskytuje mnoho užitečných projektových informací, které jsou částečně převzaty z POM souboru a částečně vygenerovány ze zdrojových kódů.

Maven se zaměřuje na sbírání a využívání dnešních nejosvědčenějších vývojových procesů. Proto příklad, specifikace, exekuce a report unit testů jsou součástí normálního build cyklu. Maven umožňuje klientům jednoduchou aktualizaci a mohou tak vždy využívat nejnovější vylepšení pomocí různých rozšiřujících pluginů.

3.2.2 Project Object Model

K definici a popisu projektu Maven používá POM soubor, který obsahuje různé, nezbytné, ale také nepovinné informace o projektu. Tento model se zapisuje do XML souboru (pom.xml). POM soubory budou hrát klíčovou roli v této diplomové práci. Neboť jak již bylo zmíněno, obsahují veškeré důležité informace o projektu / komponentě / artefaktu.

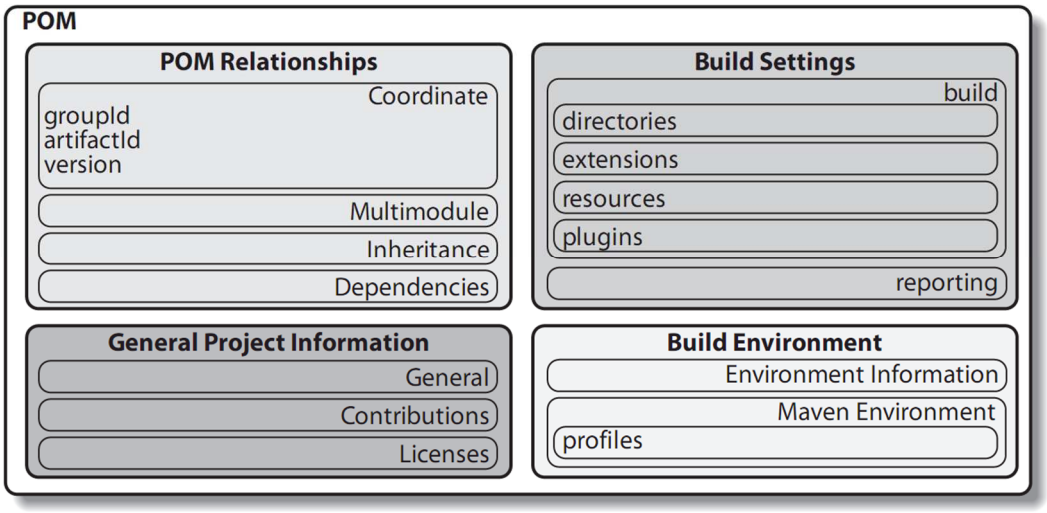
Každý projekt obsahuje právě jeden POM soubor. Protože projekty mohou být vnořené, Maven také umožňuje hierarchické vytváření pom.xml souborů a jejich dědičnost. Definicí globálních informací pro všechny sub projekty v rodičovském (parent) POM souboru se lze efektivně vyhnout duplicitním definicím.

POM říká Mavenu o jaký typ projektu se jedná a jak jej zpracovat k vygenerování finálního výstupu ze zdrojových souborů. Stejně jako je v Javě deskriptor webových aplikací - web.xml. Podobně si lze soubor pom.xml představit jako Makefile či And build.xml. POM soubor bývá také často nazýván jako projektový deskriptor či artefakt deskriptor. Detailnější rozbor tohoto deskriptoru je popsán v kapitole 3.2.3.

Ačkoliv je Maven projekt defaultně cílený k sestavování Java artefaktů ze zdrojových souborů, POM soubor není specifický pouze pro Java projekty [MVNCR].

3.2.3 Obsah POM

XML soubor obsahuje čtyři základní kategorie a konfigurace. Na následujícím obrázku 3.2.3 - 1 je znázorněna obecná struktura deskriptoru pom.xml. Aby čtenář dostal konkrétní představu, jaké elementy může POM soubor obsahovat, bude následovat detailnější popis [MVNDG] jednotlivých bloků deskriptoru spolu s příklady.



Obr. 3.2.3 - 1 Project Object Model [MVNDG]

POM RELATIONSHIPS

Projekt je málokdy samostatný. Závisí na ostatních projektech, přejímá konfigurace z rodičovských POM souborů, definuje vlastní koordinace a může obsahovat pod-moduly.

Nejjednodušší příklad obsahu deskriptoru:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch08</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

Maven Coordinates

Trojice elementů <groupId> <artifactId> a <version> se také často označují jako koordináty artefaktu / projektu. Tyto prvky musí obsahovat každý projekt a jsou také požadovány v případě definice rodičovského POM souboru, nebo při definici závislosti. Tyto tři prvky jasně identifikují projekt a symbolizují jakoby „adresu“ artefaktu. Proto tyto tři elementy uvidíme velmi často a jejich význam je následovný:

<groupId> je obecně unikátní napříč organizací nebo projektem. Například všechny hlavní Maven artefakty by měly mít groupId: org.apache.maven. GroupId nemusí nutně používat tečky jako oddělovač struktury balíku, ale je dobrý zvykem tuto zásadu dodržovat.

<artifactId> je většinou název samotného projektu. Spolu s elementem groupId tvoří unikátnost projektu, čímž je možné artefakt odlišit od všech artefaktů na světě.

<version> element je poslední část v pojmenování artefaktu. Každý projekt prochází vývojem a verzováním. Definicí verze v deskriptoru mají uživatelé jasno, který artefakt používají. Popis verze se dál dělí na menší části, viz příklad:

```
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>3.2.1.RELEASE</version>
```

Ze zápisu pak získáme dílčí prvky verze:

Major verze: 3
Minor verze: 2
Micro verze: 1
Qualifier: RELEASE

Nyní když máme identifikaci (adresu) artefaktu groupId:artifactId:version , existuje ještě jeden element, který nám uceluje popis artefaktu:

<packaging> může definovat různý typy artefaktů. Například: pom, jar, maven-plugin,ejb, war, ear, rar, par. Chybí-li tento element v POM souboru. Maven automaticky předpokládá, že se jedná o typ jar.

Multi-Module

Projekt obsahující další pod-moduly se nazývá multimodul nebo také agregátor. Moduly jsou definovány v seznamu modulů v POM souboru projektu a jsou skupinově spouštěny. Seznam modulů se někdy nazývá reaktor.

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>my-parent</artifactId>
 <version>2.0</version>

 <modules>
  <module>my-project</module>
  <module>another-project</module>
 </modules>
</project>
```

Dependencies

Základní kámen POM je jeho seznam závislostí. Téměř každý projekt závisí na jiném proto, aby mohl být spuštěn. Maven automaticky vyhodnotí a stáhne definované závislosti projektu. Jako bonus, Maven dokáže vyřešit také závislosti závislostí – tzv. **tranzitivní závislosti**. Závislost se definuje v elementu <dependency>. Ukázka zápisu v pom.xml souboru:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
              http://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <dependencies>
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.0</version>
   <type>jar</type>
   <scope>test</scope>
   <optional>true</optional>
  </dependency>
  ...
 </dependencies>
 ...
</project>
```

Verze závislostí lze také definovat jako rozsahy verzí a mohou být zapsány následovnou syntaxí:

1.0	"jemný" požadavek, doporučení na verzi 1.0 pokud Maven najde více verzí
[1.0]	"tvrdý" požadavek na verzi 1.0
(,1.0]	verze <= 1.0
[1.2,1.3]	1.2 <= verze <= 1.3
[1.0,2.0)	1.0 <= verze < 2.0
[1.5,)	verze >= 1.5
(,1.0],[1.2,)	verze <= 1.0 nebo verze >= 1.2
(,1.1),(1.1,)	definice výjimky konkrétní verze 1.1

Inheritance

Poslední důležitou části v tomto bloku je možnost dědění od předka / rodiče. Rodič se definuje v elementu <parent> a potomci dědí tyto prvky:

- závislosti
- vývojáři a přispěvatelé
- seznam pluginů
- seznam reportů
- spouštění pluginů spolu s jejich id
- konfiguraci pluginů

Chce-li uživatel definovat předka, musí jej jasně identifikovat pomocí již zmíněných identifikátorů v elementu <parent>:

```
<project>
 <modelVersion>4.0.0</modelVersion>

 <parent>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>my-parent</artifactId>
  <version>2.0</version>
  <relativePath>../my-parent</relativePath>
 </parent>

 <artifactId>my-project</artifactId>

</project>
```

Tímto končí blok POM Relationships , který je zásadní pro vypracování této práce, a proto byl rozebrán detailněji. Následující bloky jsou pouze stručně shrnuty a není potřeba je podrobně rozebírat. Chce-li čtenář získat více informací, doporučuji seznámit se literaturou [MVNDG] a [MVNCR], případně navštívit stránku s kompletní definicí Maven modelu https://maven.apache.org/ref/3.0.4/maven-model/maven.html.

BUILD SETTINGS

V této části se může modifikovat chování hlavního Maven buildu / sestavení. Lze změnit lokace zdrojových souborů, testů, lze přidat nový plugin, přiřadit úkol životního cyklu pluginu či upravit různé parametry.

GENERAL PROJECT INFORMATION

Tento blok obsahuje název či URL projektu, sponzorující organizace, seznam vývojářů a účastníků spolu s licencí projektu.

BUILD ENVIRONMENT

Zde je možno definovat profily, které mohou být aktivovány v různých situacích či prostředích. Počet profilů není omezen a jejich spuštění probíhá až při samotném buildu aktivací příslušným parametrem.

3.3 Maven Repository

Maven úložiště je místo – adresář, kde se nachází všechny jar, pluginy, knihovny a ostatní specifické projektové soubory zvané artefakty, které může Maven jednoduše použít. Objekty jsou systematicky ukládány do pevně dané a srozumitelné adresářové struktury – kapitola 3.3.5.

Tyto artefakty jsou dostupné všem projektům a jsou zpracovány jako závislosti v POM souborech jiných projektů. Každý artefakt existuje pouze jednou, ale může mít uloženo více svých verzí.

3.3.1 Lokální úložiště

Zde se nachází všechny artefakty nutné při sestavování aplikace pomocí Maven. Vytvoří se ve chvíli, kdy se poprvé zavolá jakýkoliv maven příkaz. Maven lokální úložiště obsahuje všechny projektové závislosti (knihovny, jar sobory, pluginy atd). Je-li vyžadován nějaký artefakt projektem, nejprve se zkontrolují dostupné artefakty v lokálním uložišti. V případě že zde není, maven prohledá vzdálená úložiště – kapitola 3.3.2, která lze definovat v POM souboru, a uloží jej lokálně.

Úložiště se ve Windows implicitně vytvoří v adresáři %USER_HOME%\.m2 . Cílový adresář úložiště lze změnit v Maven settings.xml souboru, který je defaultně dostupný v %M2_HOME%\conf adresáři. Po spuštění maven příkazu, se všechny závislosti budou ukládat do této nové cesty.

```
<settings>
   <localRepository>C:/MyLocalRepository</localRepository>
</settings>
```

3.3.2 Vzdálené úložiště

Je úložiště, které se nachází na jiném počítači, které je přístupné přes síť. V případě, že Maven nenajde požadovaný artefakt v centrálním uložišti – kapitola 3.3.3, zastaví se proces sestavování (build) a vyhodí se výjimka do konzole. K vyvarování se podobné situaci, Maven umožňuje definovat vzdálená úložiště v POM souboru popsaném v kapitole 3.2.2. Vývojář může nadefinovat vlastní cestu k uživatelskému uložišti, kde se nachází potřebné knihovny či artefakty.

Na příkladu níž (Obr. 3.3.2 – 1), je ukázka definice vzdálených uložišť v XML tagu <repositories> . Díky této definici je Maven schopen najít a stáhnout závislosti, které nenajde v lokálním či centrálním uložišti [MVNRP].

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.companyname.projectgroup</groupId>
   <artifactId>project</artifactId>
   <version>1.0</version>
   <dependencies>
      <dependency>
         <groupId>com.companyname.common-lib</groupId>
         <artifactId>common-lib</artifactId>
         <version>1.0.0</version>
      </dependency>
   <dependencies>
   <repositories>
      <repository>
         <id>companyname.lib1</id>
         <url>http://download.companyname.org/maven2/lib1</url>
      </repository>
      <repository>
         <id>companyname.lib2</id>
         <url>http://download.companyname.org/maven2/lib2</url>
      </repository>
   </repositories>
</project>
```

Obr. 3.3.2 - 1 Definice vzdáleného úložiště v pom.xml

3.3.3 Centrální úložiště

Centrální úložiště bylo vytvořeno komunitou Apache a obsahuje obrovské množství často používaných knihoven. Jedná se vlastně o vzdálené úložiště. V případě, že Maven nenajde artefakt v lokálním uložišti, začne hledat v centrálním uložišti, které je dostupné na URL:

http://repo1.maven.org/maven2/

K procházení tohoto úložiště je poskytována maven komunitou URL: http://search.maven.org/#browse . Zde si může vývojář vyhledat všechny dostupné knihovny včetně jejich starších verzí.

KLÍČOVÉ VLASTNOSTI

- Úložiště je spravováno Apache komunitou
- Nemusí být konfigurováno
- K jeho přístupu a prohledávání je zapotřebí připojení k internetu

3.3.4 Vyhledávací sekvence

Jakmile je zavolána maven akce sestavování, Maven začne vyhledávat závislosti v tomto pořadí [MVNRP]:

Krok 1: Prohledávání artefaktů v lokálním úložišti, nemůže-li být nějaká závislost vyhodnocena, následuje další krok.

Krok 2: Vyhledávání závislosti v centrálním úložišti. Nenajde-li Maven artefakt ani zde, nastávají dvě možnosti. Je-li nadefinováno vzdálené úložiště, pokračuje se krokem č. 4. Pokud není určeno další místo prohledávání, následuje krok 3. Najde-li Maven artefakt v centrálním uložišti, stáhne jej do lokálního úložiště pro budoucí použití.

Krok 3: Není-li definováno vzdálené úložiště, Maven zastaví proces sestavovaní a vyhodí chybu do konzole (Unable to find dependency).

Krok 4: Vyhledávání artefaktu ve vzdáleném úložišti, je-li nalezen, je stažen a uložen do lokálního úložiště, pro příští použití. Není-li závislost nalezena, Maven zastaví proces sestavování a vyhodí chybu.

3.3.5 Struktura úložiště

Všechny maven úložiště mají stejnou strukturu. Jedná se hierarchický soubor adresářů vycházející z identifikátorů maven artefaktu. Konkrétní artefakt lze v úložišti dohledat podle adresy z jeho koordinát:

```
<groupId>org.apache.maven.indexer</groupId>
<artifactId>maven-indexer</artifactId>
<version>5.1.1</version>
```

Adresa v operačním systému Windows:

```
.m2\repository\$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. [RJ13]. Výsledná adresa artefaktu je:

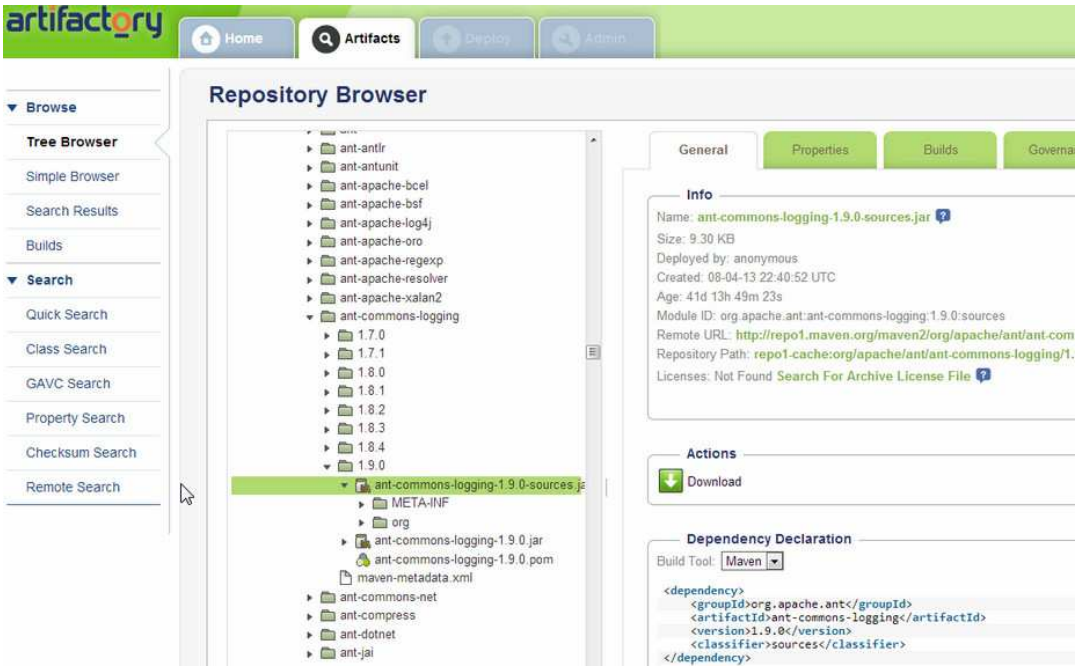
```
C:\Users\MBrozek\.m2\repository\org\apache\maven\indexer\maven-indexer\5.1.1\
```

3.3.6 Nástroje pro správu Maven repository

V této kapitole jsou zmíněny příklady správců maven úložišť. Každý bude pro představu stručně popsán spolu s přiloženým odkazem na produkt.

ARTIFACTORY

Artifactory je výkonný sestavovací server artefaktů, vyvinutý k virtuálnímu hostování jakéhokoliv binární artefaktu s podporou Javy a Mavenu. Uživatel může konfigurovat vzdálená úložiště nebo vytvářet virtuální úložiště a procházet jejich artefakty. Uživatel také může obsluhovat úložiště pomocí REST API.



Obr. 3.3.6 - 1 Artifactory GUI [ARTF]

VLASTNOSTI

Spolehlivost: Jako lokální proxy do vnějšího světa, Artifactory garantuje konzistentní přístup k potřebným komponentám.

Efektivita: Vzdálené artefakty jsou lokálně uloženy jako cache pro další znovupoužití. Nemusí být proto neustále stahovány.

Zabezpečení: Pokročilé zabezpečovací techniky dovolují kontrolovat přístup k artefaktu a definovat jeho rozmístění.

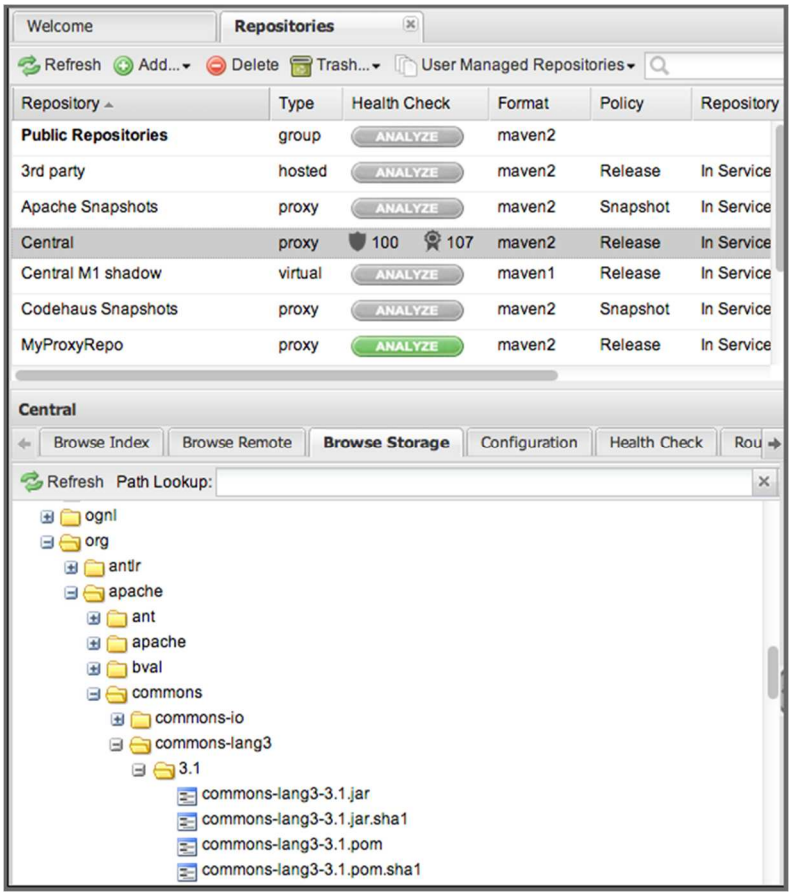
Stabilita: Podporuje velké zatížení spolu s extrémně vysokou souběžností a bezkonkurenční integraci dat.

Produktivita: Usnadnění programátorům nalezení jakéhokoliv artefaktu s vyhledávácími XML metadaty a uživatelským nastavením [ARTF].

http://www.jfrog.com/artifactory/

NEXUS

Správce Nexus od společnosti Sonatype byl jeden z prvních správců využívající Lucene Index – viz kapitola 4. Díky dlouhodobému vývoji je Nexus velice uživatelsky příjemný, intuitivní a efektivní nástroj.



Obr. 3.3.6 - 2 Nexus GUI

VLASTNOSTI

Jednoduchost: Snadné sdílení interních a externích komponent napříč vývojovým týmem.

Proxy a cache: Možnost využití správce jako proxy serveru. Komponenty ze vzdálených úložišť jsou ukládány do lokálních cache uložišť.

Průhlednost: Uživatel ví přesně, kde a jaké komponenty jsou použity. V případě defektu jsou artefakty snadno dohledatelné.

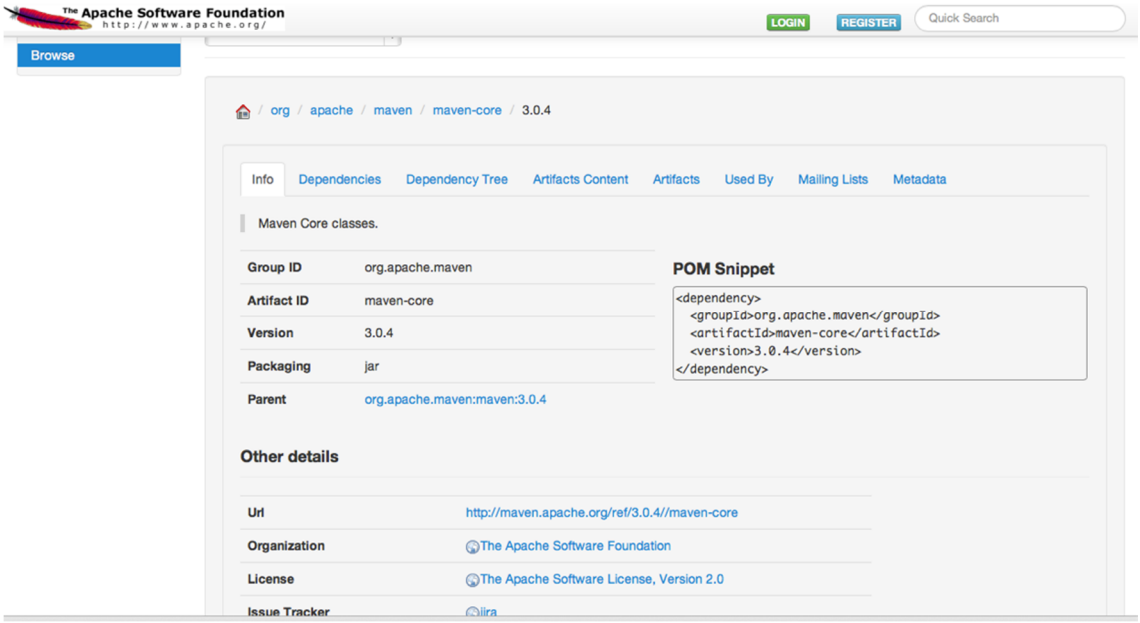
Efektivita: Pomáhá vývojářům vybírat lepší komponenty, a tím se jednoduše vyhnout problémům.

Aktuálnost: Možnost upozornění, je-li opravena chyba v open source projektech nebo je vydána nová verze.

http://www.sonatype.com/nexus/product-overview
http://books.sonatype.com/nexus-book/index.html

ARCHIVA

Tento správce je vyvíjen komunitou Apache. Stejně jako předchozí nástroje nabízí Archiva podobné možnosti jako procházení vzdálených úložišť, vyhledávání, přidávání či mazání artefaktů, nastavení pravidel pro proxy server či vytvoření virtuálních úložišť.



Obr. 3.3.6 - 3 Archiva GUI

http://archiva.apache.org/index.cgi
http://archiva.apache.org/docs/2.2.0/userguide/index.html

4 Vyhledávání v Maven Repository

Společnost Sonatype přišla jako první s myšlenkou vytvořit vyhledávací index pro Maven úložiště. Tím byli uživatelé ušetřeni nekonečnému vyhledávání v uložištích. Programátoři tudíž nemusejí obětovat terabajty přenosu stahováním tisíců artefaktů, které ani nikdy nepoužijí. Celá myšlenka indexace artefaktů byla víc než užitečná pro jakýkoliv projekt.

Na počátku byl proces vyhledávání v repository založen na několika nezávislých systémech analyzující vždy celé úložiště. K prohledávání úložiště bylo zapotřebí stáhnout celé úložiště a spustit sérii několika úkonů. Neexistoval žádný mechanismus, který by usnadnil vyhledávání Maven artefaktů kliknutím v prohlížeči či full text vyhledáváním.

První správci úložiště obsahovaly nezávislé implementace pro vyhledávání a indexaci. Počáteční verze programu Archiva měla nezávislý knihovní index, založený na Lucene (kapitola 4.2), kdežto Artifactory (kapitola 3.4) bylo závislé na JCR úložišti.

Java Content Repository API – JCR, je speciální typ úložiště, nazývané Content Repository a je specifikováno v JSR 283 a dříve v JSR 170. Nejznámější implementací je Apache Jackrabbit. Základní myšlenka je, že vše je „Content“ a JCR jej zpracovává jako strom nodů a parametrů za použití bohatých datových typů.

Ačkoliv bylo jasné, že úložiště odehrají klíčovou roli v poskytování jednoduchého vyhledávání artefaktů pomocí jejich metadat a názvů tříd, prvotní pokusy stále vyžadovaly stáhnutí celých uložišť a pustit časově náročné a CPU vytěžující procesy pouze k vytvoření vyhledávácího indexu. Zde využil Nexus Indexer svoji příležitost [STPB].

4.1 Nexus Indexer

Při vývoji správce Nexus - viz kapitola 3.3.6 bylo cíleno na vytvoření standardu a udržitelného směru vyhledávání artefaktů. Proto je Nexus jeden z nejpopulárnějších správců úložišť. Součástí Nexusu byl také vývoj komponenty Nexus Indexer, jenž definuje standart formátu úložiště, ale hlavně definuje přenosný formát, který obsahuje informace o uložišti. Nexus index je vlastně Lucene index, kterému je věnována kapitola 4.2.

Tento formát indexu, je to, co správce stáhne ze vzdáleného úložiště, a také zahrnuje důvod, proč správci jako Nexus či Archiva umožňují procházet obsah vzdáleného úložiště bez nutnosti stáhnutí celého obsahu repository.

Nexus, byl tedy první správce, který definoval standard formátu indexu repository. S tímto modelem mohly servery jako je Maven centrální úložiště a ostatní populární open source repository pravidelně indexovat úložiště a pohodlně poskytovat index uživatelům. Místo stahování terabajtů dat z internetu můžou klientské programy jako Maven, IDE vývojářů či jiný správce úložiště stáhnout pouze optimalizovaný index, obsahující všechna metadata hledaného artefaktu v uložišti.

Jak se formát indexu více šířil skrz Maven ekosystém a stával se více obecným v ostatních jazycích a systémech, společnost Sonatype, výrobce Nexus správce, došla k názoru, že by se měl Nexus Indexer oddělit od produktu Nexus. Nakonec bylo rozhodnuto, že Nexus Indexer bude darován komunitě Maven [STPB]. Nexus Indexer se přejmenoval na Maven Indexer (kapitola 4.3) a vývoj je nyní součástí projektu Apache Maven. K předání došlo v roce 2011 a k dnešnímu datu je volně dostupná verze 5.1.1 Maven Indexeru v centrálním uložišti.

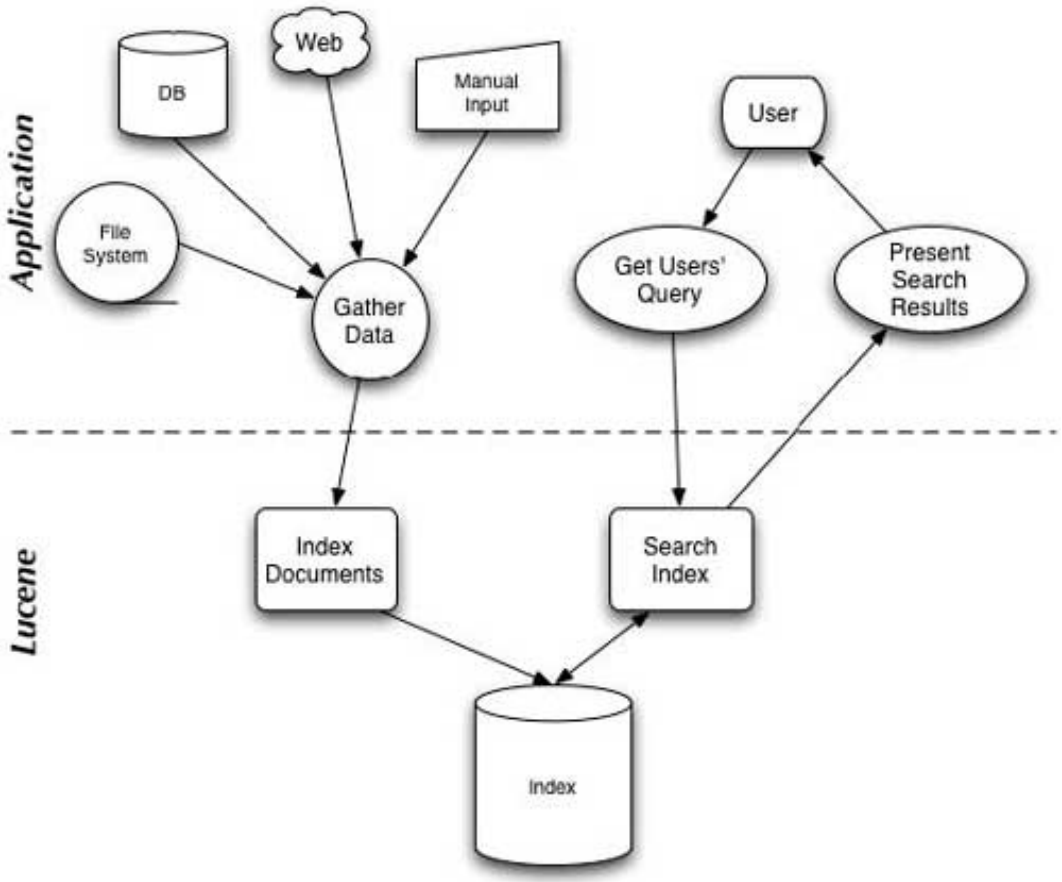
4.2 Apache Lucene

Apache Lucene je vysoce výkonný, full textový vyhledávač napsaný čistě v Javě. Lucene poskytuje jednoduché a silné API, které vyžaduje minimální znalosti fulltextového vyhledávání. Apache Lucene neslouží pouze k vyhledávání. Aby mohl prohledávat kontext, musí nejprve obsah oindexovat a vytvořit Lucene index.

Při prvním seznámení s tímto projektem dochází často k mýlce, že Lucene je stand-alone aplikace – program, či webový vyhledávač, což není. Implementací této softwarové knihovny získává aplikace schopnosti indexování a efektivního vyhledávání. [LIA05].

Lucene nezajímá zdroj dat, jeho formát ani jeho jazyk, pokud je možné data konvertovat do textu. Jelikož jsou v Maven repository především komponenty, obsahující různé soubory, Lucene při jejich indexaci použije analyzátor textových souborů.

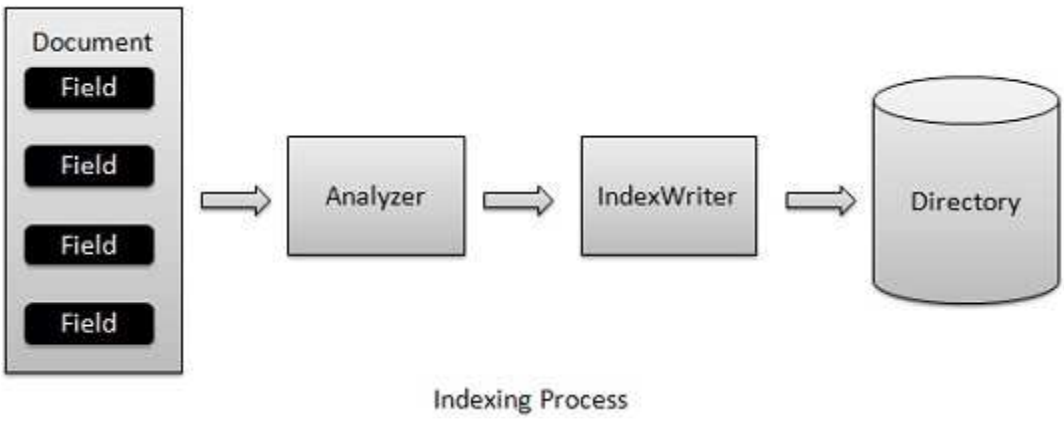
Na následujícím obrázku je zobrazena integrace Apache Lucene s různými aplikacemi. Úložiště komponent si lze představit jako File System. Lucene zpracuje data, vytvoří k nim index a uživatel může ve vytvořeném obsahu efektivně vyhledávat.



Obr. 4.2 - 1 Ukázka integrace Lucene [LIA05]

4.2.1 Efektivní indexování

Díky jednoduchému API vypadá indexování relativně jednoduše. Ve skutečnosti se skládá z několika komplexních operací, ze kterých můžeme vytýčit tři hlavní procesy. Konverze souborů do textových dokumentů, následně jejich analýza a nakonec vytvoření indexu [LIA05], viz obrázek 4.2.1 – 1.



Obr. 4.2.1 - 1 Proces vytvoření indexu [LIA05]

Následující příklad popisuje metodu pro vytváření indexů:

```
public static void createIndex() throws Exception {

    Analyzer analyzer = new StandardAnalyzer();
    boolean recreateIndexIfExists = true;
    IndexWriter iw = new IndexWriter(INDEX_DIRECTORY, analyzer, recreateIndexIfExists);
    File dir = new File(FILES_TO_INDEX_DIRECTORY);
    File[] files = dir.listFiles();
    for (File file : files) {
        Document document = new Document();
        String path = file.getCanonicalPath();
        document.add(new Field(FIELD_PATH, path, Field.Store.YES, Field.Index.UN_TOKENIZED));

        Reader reader = new FileReader(file);
        document.add(new Field(FIELD_CONTENTS, reader));
        iw.addDocument(document);
    }
    iw.optimize();
    iw.close();
}
```

Konstanta INDEX_DIRECTORY je cesta, kde bude vytvořen obsah formátu Lucene indexu, viz kapitola 4.2.4. Konstanta FILES_TO_INDEX_DIRECTORY je cesta k souborům, které požadujeme o oindexovat.

VLASTNOSTI INDEXOVÁNÍ

- Rychlost až 150GB za hodinu na moderním hardware
- Minimální požadavky na paměť – cca 1 MB haldy
- Inkrementální indexování stejně rychlé jako dávkové
- Velikost indexu je přibližně 20-30% indexovaného textu

4.2.2 Přesné vyhledávácí algoritmy

V prvních verzí Lucene, byly vráceny nejvíce důležité dokumenty jako první sestupně podle ohodnocení. Později bylo přidáno sofistikované řazení pro výsledky. V následujících odstavcích budou popsány hlavní vyhledávací třídy.

VLASTNOSTI VYHLEDÁVÁNÍ

- Ohodnocené vyhledávání – nejlepší výsledek vrácen jako první
- Více efektivních dotazů – fráze, wildcard, rozsahové …
- Typové vyhledávání – autor, název, obsah
- Řazení dle typu
- Multi-indexové vyhledávání se sloučením výsledků
- Souběžné vyhledávání s aktualizací

INDEXSEARCHER

Třída IndexSearcher [LIA05] obsahuje několik vyhledávácích metod, z nichž v následujícím příkladu bude nastíněna pouze základní search(Query), která vrací objekty Hits jako výsledky sestupně seřazené podle důležitosti. Pro aplikaci řazení je nutné použít syntaxi search(Query, Sort). IndexSearcher hledá to, co IndexWriter oindexuje.

QUERY

Konkrétní subtřídy obsahující logiku pro vybraný typ query – viz kapitola 4.2.3. Instance Query jsou předány vyhledávácí metodě třídy IndexSearcher.

QUERY PARSER

Zpracovává zadaný (a čitelný) výraz do konkrétního objektu Query.

HITS

Reprezentuje výsledek poskytnutý metodou search třídy IndexSearcher. Třída Hits nabízí čtyři metody týkající se výsledku:

length() - počet dokumentů v kolekci Hits
doc (n) - instance Dokumentu n-tého top-ohodnoceného dokumentu
id(n) - ID n-tého top-ohodnoceného dokumentu
score(n) - normalizované skóre

Příklad metody vyhledávání pomocí Lucene indexu:

```
public static void searchIndex(String searchString) throws IOException, ParseException {

        Directory directory = FSDirectory.getDirectory(INDEX_DIRECTORY);
        IndexReader indexReader = IndexReader.open(directory);
        IndexSearcher indexSearcher = new IndexSearcher(indexReader);

        Analyzer analyzer = new StandardAnalyzer();
        QueryParser queryParser = new QueryParser(FIELD_CONTENTS, analyzer);
        Query query = queryParser.parse(searchString);
        Hits hits = indexSearcher.search(query);

        Iterator<Hit> it = hits.iterator();
        while (it.hasNext()) {
                Hit hit = it.next();
                Document document = hit.getDocument();
                String path = document.get(FIELD_PATH);
        }
}
```

Konstanta INDEX_DIRECTORY je cesta k adresáři, obsahující oindexované dokumenty popsané v kapitole 4.2.4.

4.2.3 Druhy Lucene Query

Lucene používá k vyhledávání různé druhy query (dotazů), které budou nyní stručně vysvětleny [LIA05].

TERMQUERY

Nejjednodušší způsob vyhledání indexu je přes specifický term. Term je nejmenší oindexovaný díl, skládající se z názvu pole a jeho textové hodnoty.

```
Term t = new Term("contents", "java");
Query query = new TermQuery(t);
```

Všechny dokumenty, které obsahují slovo java budou obsaženy v navráceném objektu Hits po dokončení vyhledávání. Při vyhledávání je zapotřebí brát v potaz, že TermQuery je case-sensitive.

RANGEQUERY

Termy jsou v indexu seřazeny lexikograficky, což umožňuje efektivní vyhledávání termů v rozsahu. RangeQuery vyhledává od počátečního termu do koncového. Počáteční i koncové definice mohou být započítány nebo vyloučeny z rozsahu.

```
Term begin = new Term("pubmonth","198805");
Term  end = new Term("pubmonth","198810");

RangeQuery query = new RangeQuery(begin, end, true);
IndexSearcher searcher = new IndexSearcher(directory);
Hits hits = searcher.search(query);
```

PREFIXQUERY

Vyhledávání pomocí PrefixQuery najde dokumenty obsahující Term (termín) začínající specifickým prefixem.

```
IndexSearcher searcher = new IndexSearcher(directory);
Term term = new Term("category", "/technology/computers/programming");
PrefixQuery query = new PrefixQuery(term);
Hits hits = searcher.search(query); // hledání knih „programming“, včetně podkategorií

hits = searcher.search(new TermQuery(term));  //hledání bez podkategorii
```

BOOLEANQUERY

Zmíněné query mohou být komplexně kombinovány pomocí BooleanQuery. Obsahují pod-query, které mohou být volitelné, požadované či zakázané. Do BooleanQuery se přidávají pomocí API metody:

```
public void add(Query query, boolean required, boolean prohibited)
```

Příklad:

```
TermQuery searchingBooks = new TermQuery(new Term("subject","search"));
RangeQuery currentBooks = new RangeQuery(new Term("pubmonth","200401"),
new Term("pubmonth","200412"), true);

BooleanQuery currentSearchingBooks = new BooleanQuery();
currentSearchingBooks.add(searchingBooks, true, false);
currentSearchingBooks.add(currentBooks, true, false);

IndexSearcher searcher = new IndexSearcher(directory);
Hits hits = searcher.search(currentSearchingBooks);
```

Díky výše uvedeným vlastnostem bude právě tento druh dotazů nejvíce využíván při implementaci nového CRCE pluginu pro Maven repository. Pomocí BooleanQuery lze tvořit dotazy podobné dotazům pro SQL databázi.

4.2.4 Formát souboru indexu

Při indexování dokumentů Apache Lucene vytvoří v definovaném místě adresář, obsahující některé z následujících souborů. Tomuto adresáři se říká indexovaný kontext. Obsahuje indexy všech zpracovaných dokumentů a na základě vhodných dotazů může Apache Lucene dokumenty efektivně prohledávat.

Název	Přípona	Popis
Soubory segmentů	segments.gen segments_N	Obsahuje informace o segmentech
Zámek souboru	write.lock	Zabraňuje vice instancím IndexWriter zápis do stejného souboru
Složený soubor	.cfs	Nastavitelný „virtuální“ soubor, skládající se z ostatních indexů pro systémy, které často nemají dostatek správců souborů
Vstupní tabulka složených souborů	.cfe	„Virtuální“ tabulka obsahující všechny vstupy v konkrétních .cfs souborech
Pole	.fnm	Obsahuje informace o polích
Index pole	.fdx	Obsahuje ukazatel na pole dat
Pole dat	.fdt	Uložená pole pro dokumenty
Informace o termínech	.tis	Část slovníků termínů, uloženy informace termínů
Index informací termínů	.tii	Index souboru informací termínů
Frekvence	.frq	Obsahuje seznam dokumentů, které obsahují každý termín s jeho frekvencí
Pozice	.prx	Ukládá pozici, kde se termín nachází v indexu
Normy	.nrm	Zakódované délky a faktory dokumentů a polí
Index vektoru termínů	.tvx	Uložený offset do souboru dokumentu dat
Vektor termínu dokumentu	.tvd	Obsahuje informace o každém dokumentu, který má vektor termínů
Pole vektoru termínu	.tvf	Pole úrovní informací o vektoru termínu
Smazané dokumenty	.del	Informace o smazaných souborech

Tab. 4.2.4 – 1 Typy souborů Lucene [LUCA]

4.3 Maven indexer

Apache Maven Indexer (původně Nexus Indexer) je ve skutečnosti fasáda nad knihovnou Apache Lucene. Rozšiřuje její funkcionalitu a nabízí příjemnější rozhraní. Obsahuje tři základní moduly [MVIX], z nichž právě první, ten nejdůležitější, detailněji rozebereme.

4.3.1 Indexer Core

Tato část obsahuje všechny potřebné třídy ke zpracování informací indexů z lokálních nebo vzdálených úložišť. Pomocí tohoto modulu lze prohledávat úložiště a také vytvářet jejich indexy. V následujících odstavcích budou vysvětleny některé významné třídy a rozhraní.

INDEXERCREATOR

Rozhraní zodpovědné za ukládání a čtení dat z Lucene indexu. Toto rozhraní implementuje třída AbstractIndexCreator. Následující třídy dědí tuto abstraktní třídu a každá má specifické využití.

JarFileContentsIndexCreator

- Používá se k indexování názvů Java tříd z Maven artefaktu (JAR, WAR)

MinimalArtifactInfoIndexCreator

- Poskytuje základní informace o Maven artefaktu jako groupId, artifactId, version, classifier, packaging. Aby indexování bylo co nejrychlejší, neotvírají se žádné soubory. Získané informace mohou být někdy vyhodnoceny jako „best-effort“ a nemusí odpovídat realitě.

MavenArchetypeArtifactInfoIndexCreator

- Využívá se k opravě hodnoty packaging na „maven-archetype“ pokud prozkoumaný JAR soubor je Archetype. Jelikož packaging vyhodnocuje již předchozí třída, tento indexer slouží pouze jako alternativa.

MavenPluginArtifactInfoIndexCreator

- Poskytuje informace o Maven pluginu. Získává hodnoty prefix a goals.

OsgiArtifactIndexCreator

- Indexuje některá OSGi metadata. Zpracovává všechny JAR soubory, nejen ty s hodnotou „bundle“ v elementu packaging.

- Indexovaná metadata OSGi:
    - Bundle-SymbolicName
    - Bundle-Version
    - Export-Package
    - Export-Service

```
// Vytvoření indexerů
List<IndexCreator> indexers = new ArrayList<>();
indexers.add(plexusContainer.lookup(IndexCreator.class, "min"));
indexers.add( plexusContainer.lookup( IndexCreator.class, "maven-archetype" ) );
indexers.add( plexusContainer.lookup( IndexCreator.class, "osgi-metadatas" ) );
```

INDEXINGCONTEXT

Toto rozhraní reprezentuje úložiště artefaktů (viz kapitola 4.2.4), které je možno indexovat a prohledávat. Indexovaný kontext je stavová komponenta, udržující stav čtenářů a tvůrců indexu. Poskytuje například metody:

getIndexCrators() – vrací seznam indexerů

getRepository() – vrací lokaci úložiště

getRootGroups() – vrací aktuální množinu názvů groupId v kontextu

getSize() - vrací nijak specifickou hodnotu která reprezentuje „velikost“ obsahu, nikoli však počet artefaktů. Slouží čistě k porovnání více kontextů, ke zjištění který je větší.

```
// Vytvoření indexovaného kontextu
IndexingContext createIndexingContext( String id, String repositoryId, File repository, File
        indexDirectory, String repositoryUrl, String indexUpdateUrl, boolean searchable,
        boolean reclaim, List<? extends IndexCreator> indexers )
    throws IOException, ExistingLuceneIndexMismatchException, IllegalArgumentException;
```

INDEXER

Hlavní komponenta z Apache Maven Indexeru. Poskytuje metody pro vytvoření a zavření objektu IndexingContext či metody k jeho správě a prohledávání.

```
// Vytvoření vyhledávací boolean query a vrácení výsledku
Indexer indexer = plexusContainer.lookup(Indexer.class);
Query gidQ = indexer.constructQuery(MAVEN.GROUP_ID, new SourcedSearchExpression(ai.groupId));
Query aidQ = indexer.constructQuery(MAVEN.ARTIFACT_ID, new SourcedSearchExpression(ai.artifactId));
Query pckQ = indexer.constructQuery(MAVEN.PACKAGING, new SourcedSearchExpression("bundle"));

BooleanQuery bq = new BooleanQuery();
bq.add(gidQ, Occur.MUST);
bq.add(aidQ, Occur.MUST);
bq.add(pckQ, Occur.MUST);

FlatSearchRequest fsr = new FlatSearchRequest(bq, indexingContext));
FlatSearchResponse response = indexer.searchFlat(fsr);
Set<ArtifactInfo> results = response.getResults();
```

Jakmile je vytvořen indexovaný kontext reprezentující uložiště, uživatel může vytvářet různé dotazy, podle kterých Indexer vrací vyhledané výsledky.

4.3.2 Artifact Utils

Obsahuje malou množinu nástrojů užitečných především pro zpracování artefaktů. V balíku jsou třídy ke zpracování a poskytování základních informací cílového objektu. Například třída GAV slouží k reprezentaci unikátních identifikátorů artefaktu.

4.3.3 Indexer CLI

Tento modul slouží jako command-line interface, čili rozhraní využívající příkazovou řádku k indexaci a zveřejňování Maven úložišť.

4.4 Aether

Ve vzdálených úložištích se může vyskytovat větší množství artefaktů. Například v Central Maven Repository je k aktuálnímu dni - 27.4.2015 přes 2 461 856 dokumentů, z toho 82 544 artefaktů s packaging „bundle“ – čili komponent. Existuje mnoho způsobů, jak dané prvky získat, ale právě Aether nabízí efektivní řešení.

Mnoho uživatelů si myslí, že Aether je knihovna, která se stará především o poskytování obsahu POM souborů artefaktů, což je mylná domněnka. Aether sice pracuje s POM soubory, ale to není jeho hlavní činnost.

Eclipse Aether je knihovna napsaná v programovacím jazyce Java, sloužící primárně ke zpracování artefaktů ze vzdálených Maven úložišť [WIKEC]. Poskytuje silné a jednoduché rozhraní pro přenos artefaktů, jejich zveřejňování, řešení přímých (1st level) či tranzitivních závislostí (kapitola 3.2.3 – Dependencies). Proto je rozšířen a integrován ve velkých projektech jako je Apache Maven, m2eclipse plugin, úložiště Sonatype Nexus a další.

4.4.1 Repository system

Rozhraní RepositorySystem je klíčový bod k vytvoření systému úložiště a umožnění jeho funkcionality. Vytvoření patřičné implementace (přes dependency injection, service locator, atd.) závisí na požadavcích aplikace, jenž toto rozhraní využívá. Komponenty Aetheru implementují org.eclipse.aether.spi.locator.Service ,umožňující snadné propojení s instancí DefaultServiceLocator. Pomocí servis lokátoru lze zaregistrovat konektor úložiště a přenosové třídy.

```
public static RepositorySystem newRepositorySystem() {

    DefaultServiceLocator locator = MavenRepositorySystemUtils.newServiceLocator();
    locator.addService(RepositoryConnectorFactory.class,BasicRepositoryConnectorFactory.class);
    locator.addService(TransporterFactory.class, FileTransporterFactory.class);
    locator.addService(TransporterFactory.class, HttpTransporterFactory.class);

    return locator.getService(RepositorySystem.class);
}
```

Repository systém představuje opravdu nejdůležitější objekt z knihovny Aether. Poskytuje mnoho užitečných metod jako například:

readArtifactDeskriptor() – dává informace o artefaktu, jako přímé závislosti apod.

resolveArtifact() – stažení artefaktu do lokálního úložiště v případě potřeby Existuje-li artefakt již v úložišti, je přeskočen a není znovu stahován

deploy() – nahrání kolekce artefaktů a jejich metadat do vzdáleného úložiště

Aby mohl uživatel využívat metody z instance RepositorySystem, musí být k systému vytvořena také jeho session.

4.4.2 Repository session

Definuje nastavení a komponenty, které kontrolují systém úložiště. Jakmile je jednou objekt inicializován, je neměnný a může být sdílen napříč celou aplikací a jakýchkoliv vláken.

```
public static DefaultRepositorySystemSession newRepositorySystemSession
                                                (RepositorySystem system) {

    LocalRepository localRepo = new LocalRepository("target/local-repo");
    DefaultRepositorySystemSession session = MavenRepositorySystemUtils.newSession();
    session.setLocalRepositoryManager(system.newLocalRepositoryManager(session, localRepo));

    return session;
}
```

4.4.3 Artifact resolve

Nyní nám nic nebrání vytvořit instanci vzdáleného úložiště, nadefinovat požadavek artefaktu a vyřešit – stáhnout, specifický artefakt. V následujícím příkladu jsou popsány potřebné kroky k vyřešení artefaktu:

```
public static List<RemoteRepository> newRepositories () {

      ArrayList<RemoteRepository> remoteRepos = new ArrayList<RemoteRepository>();
      RemoteRepository central = new RemoteRepository.Builder("central", "default",
                                    "http://central.maven.org/maven2/").build();
      remoteRepos.add(central);
      return remoteRepos;
}


public Artifact resolveArtifact() {

   RepositorySystem system = Booter.newRepositorySystem();
   RepositorySystemSession session = Booter.newRepositorySystemSession( system );
   Artifact artifact = new DefaultArtifact( "args4j:args4j-tools:2.0.23" );

   ArtifactRequest artifactRequest = new ArtifactRequest();
   artifactRequest.setArtifact( artifact );
   artifactRequest.setRepositories( newRepositories() );

   ArtifactResult artifactResult = system.resolveArtifact( session, artifactRequest );
   return artifactResult.getArtifact();
}
```

Metoda newRepositories() vrací seznam vzdálených úložišť, jenž Aether bude prohledávat, nenajde-li artefakt v lokálním úložišti. Při samotném řešení artefaktu je zapotřebí nejprve vytvořit systém úložiště, k němu odpovídající session a nadefinovat požadovaný artefakt. Poté vytvoříme požadavek na vyřešení artefaktu a předáme ho systému, který nám vrátí výsledek.

5 CRCE modul pro Maven repository

Cílem této práce je vytvoření CRCE pluginu (modulu), který by rozšířil funkcionalitu CRCE o možnost zpracování Maven artefaktů, uložených buď v lokálním či vzdáleném Maven repository. Výsledkem práce bude OSGi komponenta, kterou bude možno zapínat a vypínat dle libosti uživatele.

Stávající CRCE plugin (crce-repository-impl), též nazývaný file-based repository, obstarávající veškeré procesy nad vkládanými OSGi komponentami zůstane zachován a souběžně s ním bude aktivní nový CRCE maven repository plugin.

Každý požadovaný artefakt bude předán k vyhodnocení metadat, která budou zapsána do databáze. Protože se v úložištích můžou vyskytovat statisíce artefaktů, bude kladen důraz především na časovou efektivitu aplikace. Podrobná funkcionalita nového pluginu je popsána v následující kapitole věnované specifikaci.

5.1 Specifikace

V této kapitole budou podrobně popsány požadované vlastnosti aplikace, ze kterých dostaneme ucelený pohled využitelnosti CRCE maven repository modulu. Na základě těchto požadavků navrhneme architekturu a implementuje řešení. Ovládání nového CRCE pluginu bude muset být řešeno přes konfiguračního soubor, jelikož stávající GUI plugin není přizpůsoben pro obsluhu maven úložišť.

5.1.1 Přístup k lokálnímu maven úložišti

Aplikace bude schopna získat ucelený přehled o obsahu lokálního maven repository. Z vytvořeného obsahu bude možnost číst artefakty a vytvářet k nim potřebná metadata. Vytvořená metadata pojící se k artefaktu budou zapsána v konkrétním formátu popsaném v kapitole 5.2.3 do embedded databáze. Místo lokálního úložiště bude konfigurovatelné.

5.1.2 Přístup ke vzdálenému úložišti

Aplikace bude schopna získat ucelený přehled o obsahu vzdáleného maven repository. Z vytvořeného obsahu bude možnost číst artefakty a vytvářet k nim potřebná metadata Vytvořená metadata pojící se k artefaktu budou zapsána v konkrétním formátu popsaném v kapitole 5.2.3 do embedded databáze. Místo vzdáleného úložiště bude konfigurovatelné.

5.1.3 Nastavení aktualizace obsahu úložiště

Stahování indexu vzdáleného úložiště bude možné spouštět na základě definovaného parametru. Tím dojde k výraznému ušetření času, jelikož se vyhneme stahováním i 100MB repository indexu. Obsah úložišť může být dlouhodobě stálý. Pokud ne, některá vzdálená uložiště (např. Central Maven Repository) aktualizují svůj obsah například po týdnu.

5.1.4 Nastavení cesty k index kontextu

Pro každé nadefinované úložiště vytvoří Apache Maven Indexer během indexace repository adresář, reprezentující indexovaný kontext – kapitola 4.2.4. Jakmile je tento obsah vytvořen, je schopen jej aktualizovat inkrementálně a nemusí se pouštět celá analýza od začátku. V těchto adresářích je uložen kompletní popis úložišť, nad kterým lze získávat různé požadavky k zjištění požadovaného výsledku. Umístění adresáře s indexovaným kontextem bude načteno z konfiguračního souboru při spuštění aplikace.

5.1.5 Výběr primárního úložiště

Je-li nakonfigurováno vzdálené úložiště, neznamená to nutně jeho použití. Konfigurační soubor bude obsahovat definici pouze jednoho lokálního úložiště, neboť Aether během řešení artefaktu neumí implicitně používat více jak jedno lokální úložiště. Při konstrukci úložiště je také zapotřebí nadefinovat jeho název a boolean hodnotu, zda má být aktualizován kontext.

Aether umí pracovat s více vzdálenými úložišti najednou, ale zpracování více úložišť z konfiguračního souboru by s sebou neslo výraznou změnu kódu. Bylo by zapotřebí změnit business logiku k vytvoření a vyhledávání v indexovaných kontextech.

Jakmile bude vytvořeno nové grafické rozhraní pro CRCE maven repository, vytvoření seznamu vzdálených úložišť a vyhledávání v nich bude jednoduché pomocí volání již existujících metod pro jedno úložiště.

Na rozdíl od vzdáleného úložiště, se lokální repository využívají pokaždé. Neboť toto úložiště využívá Aether při stahování artefaktů a jejich závislostí. Pokud aplikace získá parametr na čtení artefaktů primárně ze vzdáleného repository, CRCE bude řešit zpracování artefaktů ze zvoleného vzdáleného úložiště. Bude-li parametr pro primární vzdálené úložiště nastaven na false: use.remote.maven.store.default=false , CRCE bude zpracovávat artefakty z definovaného lokálního úložiště.

5.1.6 Vytvoření metadat artefaktu

Jakmile bude aplikace schopna zpracovávat artefakty z lokálního či vzdáleného úložiště, je zapotřebí vytvořit metadata k artefaktu. OsgiManifestBundleIndexer – kapitola 3.1.3, není vhodné ke zpracování metadat artefaktu použít, jelikož je přizpůsoben k indexaci OSGi bundlů pomocí čtení informací z manifest souborů, nikoliv maven artefaktů.

Bude zapotřebí vytvořit novou třídu MavenArtifactMetadataIndexer, která bude využívat existující CRCE komponentu Metadata API k obsluze metadat ,a proto se musí dodržet OBR schéma entit – viz kapitola 3.1.2. Entity budou tvořeny na základě získaných informací z artefakt deskriptorů – POM souborů. Protože aplikace bude zpracovávat veliké množství artefaktů, vyhodnocení všech metadat může být časové náročné, tudíž CRCE plugin pro maven repository bude nabízet následující dva způsoby indexace metadat artefaktů, jenž každý z nich má své výhody a nevýhody.

POM INDEXACE

Pomocí této konfigurace, se vyhodnotí pouze POM soubory artefaktů z úložiště. Při čtení artefaktů z lokálního repository není toto nastavení natolik prioritní jako při řešení artefaktů ze vzdálených úložišť. Neboť artefakt deskriptory mívají velikost řádově několika kB oproti celému balíku JAR komponenty, který může mít desítky MB.

VÝHODY:

- rychlý přenos POM souborů ze vzdálených úložišť
- malá velikost soboru, řády kB
- malá síťová zátěž
- vhodné pro vysoký počet artefaktů

NEVÝHODY:

- fyzicky chybějící komponenta v lokálním cache repository
- ostatní CRCE indexery neumí získat informace o komponentě
- je-li artefakt požadován, musí se nakonec stejně stáhnout JAR soubor
- a tím vznikne potřeba kontaktovat vzdálené úložiště dvakrát

JAR INDEXACE

Při řešení metadat artefaktů se bude zpracovávat celý balík komponenty v souborovém formátu – JAR. V případě řešení artefaktů ze vzdáleného repository se sice zvýší datový přenos, ale komponenta bude fyzicky dostupná k použití. Tento typ indexace má opačné vlastnosti oproti POM indexaci.

VÝHODY:

- ostatní CRCE indexery mají možnost získat další informace o artefaktu
- komponenta se nachází fyzicky v lokálním repository
- vhodné pro malý počet artefaktů

NEVÝHODY:

- při větším počtu artefaktů dochází k vysoké síťové zátěži
- tím se zvyšuje časová zátěž výpočtu
- prvotní indexace tisíců artefaktů může trvat i dny až týdny
- při klonování úložišť je zapotřebí velké místo na disku GB až TB

5.1.7 Vyhledávání závislostí artefaktu

Aplikace umožní vyhodnocovat přímé nebo hierarchické (tranzitivní) závislosti určitého artefaktu. Tyto závislosti budou řádně oindexovány a uloženy spolu s metadaty artefaktu do databáze. Časová náročnost bude záležet na zvoleném typu řešení závislostí. Rozdíly jsou následující:

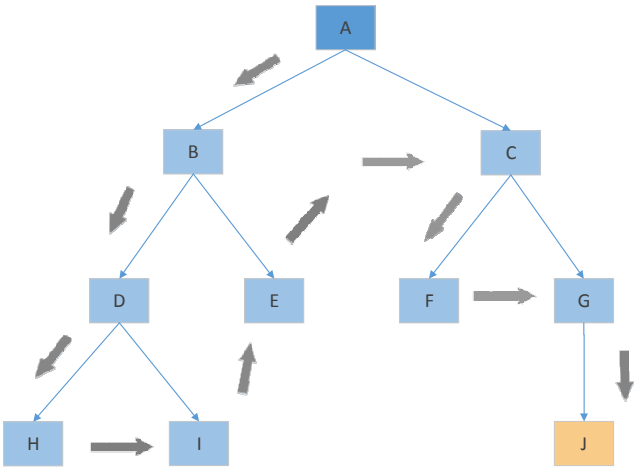
PŘÍMÉ ZÁVISLOSTI

CRCE Maven repository modul umožní řešení přímých závislostí specifického artefaktu. Řešení závislostí bude možné nejen za přítomnosti JAR souboru, ale také při nastavení využívající pouze indexaci přes POM soubory. Neboť jeli aplikace nastavena k indexaci využívající JAR soubor, Aether automaticky vyhodnotí také POM soubor, který obsahuje veškeré informace o artefaktu – kapitola 3.2.3.

Řešení přímých závislostí je velmi rychlé, neboť se vyhodnocuje pouze lineární seznam závislostí, definovaný v pom.xml souboru artefaktu, v kořenovém elementu <dependencies>. Tyto závislosti o indexuje MavenArtifactMetadataIndexer, který k nim vytvoří entity Requirement a připojí je k entitě Resource, reprezentující zpracovávaný artefakt.

HIERARCHICKÉ (TRANZITIVNÍ) ZÁVISLOSTI

Aplikace bude schopna vytvořit hierarchickou strukturu závislostí k danému artefaktu získaného z lokálního či vzdáleného úložiště při řešení artefaktu pomocí JAR nebo POM souboru. Znamená to, že se nejprve vyhledají všechny potomci závislostí pro danou závislost artefaktu a až poté se řeší další potomci další závislosti artefaktu. Jedná se o analogii prohledávání do hloubky – viz obrázek 5.1.7-1. Výsledná struktura se předá k indexaci metadat, stejně jako u přímých závislostí.



Obr. 5.1.7 – 1 DFS

Řešení tranzitivních závislostí je velice časově náročné, jelikož může dojít k hlubokému zanoření během vyhledávání. Spojení se vzdálenými úložišti může být vytvořeno i tisíckrát v rámci řešení závislostí jednoho artefaktu. Může dojít k zacyklení či odmítnutí spojení od některého vzdáleného úložiště a v nejhorším případě se tranzitivní závislosti nemusí vyhodnotit vůbec, nebude-li možné najít některou ze zanořených závislostí. Konkrétní implementace bude popsána v kapitole 5.3.1.

5.1.8 Filtrace množiny artefaktů ke zpracování

Jednu z nejdůležitějších funkcí aplikace tvoří funkce filtrace artefaktů určených ke zpracování. Úložiště mohou obsahovat veliká množství artefaktů, od deseti tisíců až po miliony. Zpracovat takový počet by bylo nepochybně velice časově náročné. Proto je zapotřebí specifikovat různé druhy filtrací, které zúží výslednou množinu prvků, jenž aplikace zpracuje.

Bude zapotřebí vymyslet a vytvořit algoritmy využívající vyhledávací dotazy z knihovny Apache Maven Indexer – kapitola 4.3.1. Použití knihovny Aether pro zjištění například všech verzí pro statisíce artefaktů by bylo silně neefektivní, jelikož vytváří spojení se vzdálenými servery. Má-li například navíc nadefinováno více úložišť, Aether k získání výsledků jednotlivých artefaktů kontaktuje každé z nich. Logika prořezávacích filtrů je popsána v následujících kapitolách.

ALL

Tato konfigurace nebude ve skutečnosti filtr. Bude-li tento parametr aktivní, CRCE zpracuje všechny komponenty v nadefinovaném úložišti. Při dalších nevhodně zvolených parametrech může celý proces zpracování trvat i několik dnů. Proto by si měl být uživatel jist počtem zpracovávaných artefaktů a případně zvážit užití tohoto argumentu.

NEWEST

Tento filtr umožní modulu CRCE – maven repository prohledat cílené maven úložiště a vyhledat poslední verzi pro každou komponentu. Výslednou množinu pak předá ke zpracování jako list. V tomto seznamu není obsažen žádný artefakt víckrát než jednou.

HIGHEST MAJOR

Aktivací tohoto parametru CRCE obdrží příkaz k prohledání obsahu lokálního či vzdáleného repository. Zjistí pro každý artefakt všechny jeho dostupné verze v daném indexovaném kontextu a z nich přidá do výsledného seznamu pouze ty, které mají aktuálně nejvyšší číslo major verze.

HIGHEST MINOR

Pokud se v úložišti vyskytuje více verzí artefaktů, parameter highest minor spustí algoritmus k vyhledání všech verzí komponenty s aktuálně největší hodnotou minor verze. Major a micro verze mohou být jakékoliv. Výsledná množina bude předána k procesu indexace artefaktu a následnému uložení metadat do integrované databáze.

HIGHEST MICRO

Filtr highest micro bude vyhodnocovat, obdobně jako předešlé parametry, pouze artefakty s největší hodnotou micro verze. Bude-li například největší hodnota v části micro verze ‘4’, ostatní verze s hodnotou menší, nebudou přidány do množiny výsledků ke zpracování.

LOWEST MINOR

Tento filtr je opakem k filtru – highest minor. CRCE modul prohledá množinu všech artefaktů a pro každý artefakt zjistí nejmenší hodnotu minor verze. Major a micro verze mohou být opět jakékoliv. Artefakty splňující podmínku nejmenší verze jsou přidány do výsledné množiny, která bude předána k procesu indexace artefaktu a následnému uložení metadat do integrované databáze.

LOWEST MICRO

Filtr sloužící k nalezení nejmenších micro verzí pro jednotlivé artefakty. Velice efektivní prořezávání pro vyhledání prvních releases daných artefaktů.

UKÁZKA VÝSLEDKŮ PŘEDCHOZÍCH FILTRŮ

Na základě definovaných požadavků můžeme vytvořit imaginární seznam verzí jednoho artefaktu, který by se mohl nacházet ve zpracovávaném úložišti. Pro lepší pochopení znázorníme jednotlivé množiny každého parametru do sloupců jim odpovídajícím. Z výsledků zobrazených v tabulce 5.1.8 - 1 je vidět, že největší množina artefaktů bude s přepínačem ‘all’ a šance na nejmenší možný počet prvků od každého artefaktu je při použití filtru ‘newest’.

Verze	All	Newest	Highest Major	Highest Minor	Highest Micro	Lowest Minor	Lowest Micro
		3.2.1	(3)	(2)	(5)	(0)	(0)
1.0.0	x					x	x
1.0.1	x					x	
1.0.2	x						
1.1.0	x						x
1.1.1	x						
1.1.2	x						
1.1.5	x				x		
1.2.0	x			x			x
2.0.0	x					x	x
2.0.2	x					x	
2.1.0	x						x
2.1.1	x						
2.2.0	x			x			x
3.0.0	x		x			x	x
3.1.0	x		x				x
3.1.2	x		x				
3.2.1	x	x	x	x			

Tab. 5.1.8 – 1 Filtrování artefaktů

GROUPID – ARTIFACTID – VERSION

Zadá-li uživatel filtr GAV, musí k němu přidat také odpovídající parametr s upřesněním, který artefakt požaduje. Aplikace poté prohledá specifický požadavek právě na jeden artefakt s group-id , artefakt-id a přesnou verzí. Pokud taková komponenta v uložišti existuje, bude zpracována. Zápis artefaktu musí být v následujícím formátu, kde jsou identifikátory artefaktu odděleny dvojtečkou:

```
artifact.resolve=gav
artifact.resolve.param=args4j:args4j-tools:2.0.23
```

GROUPID

Bude-li aktivní filtr group-id a k němu bude existovat platný parametr s definicí požadovaného group-id, CRCE prohledá obsah úložiště pro daný název a vrátí množinu všech artefaktů s daným group-id a nejnovější verzí. Zápis parametru artifact resolve musí mít definovaný alespoň groupId artefaktu. Bude-li mít parametr definováno více koordinát, CRCE zpracuje pouze první část z parametru – čili groupId.

```
artifact.resolve=groupid
artifact.resolve.param=org.apache.camel:camel-spring:2.8.2
```

GROUPID – ARTIFACTID

Tímto nastavením může uživatel požadovat vrácení množiny všech existujících verzí daného artefaktu v úložišti. Zápis parametru artifact resolve k tomuto filtru musí mít definovaný alespoň groupId a artifactId artefaktu. Poslední část s definicí verze filtr zahodí.

```
artifact.resolve=groupid-artifactid
artifact.resolve.param= org.codehaus.jackson:jackson-xc

artifact.resolve=groupid-artifactid
artifact.resolve.param= org.codehaus.jackson:jackson-xc:1.9.0
```

GROUPID – ARTIFACTID – MINVERSION

Poslední prořezávací filtr vyhledá všechny varianty artefaktu od definované minimální verze včetně. Výslednou množinu poté aplikace zpracuje.

```
artifact.resolve=groupid-artifactid-minversion
artifact.resolve.param=org.sonatype.nexus:nexus-api:1.2.0
```

5.1.9 Definice vstupů a výstupů aplikace

Ke správnému návrhu komponenty je zapotřebí definovat všechny její vstupy a určit výstupy na konci procesu aplikace. Tím dostaneme očekávané chování modulu a spolu se specifikací můžeme navrhnout architekturu.

VSTUPNÍ BOD PRÁCE

Jediný vstup CRCE pluginu pro maven repository je konfigurační soubor, obsahující veškerá nastavení pro zpracování požadovaných artefaktů. Tento soubor předaný Aktivátoru je dále vyhodnocen a na základě příslušné konfigurace je spuštěna požadovaná větev procesů.

Použití konfiguračního souboru bylo zvoleno jako dočasné řešení ovládání pluginu, dokud nebude připraveno nové grafické rozhraní pro CRCE maven repository. Po přepracování GUI funkcionalita metod maven repository pluginu zůstane zachována.

VÝSTUPY PROCESU

Na konci celého procesu zpracování komponenty bude na lokálním disku v definovaném lokálním repository minimálně deskriptor artefaktu, případně další soubory. Ke zpracovaným artefaktům budou vytvořena metadata popisující artefakt, která budou uložena do entit v interní databázi – kapitola 5.1.6, 5.1.7. Pro každé zpracovávané úložiště zůstane dostupný indexovaný kontext, v definovaném místě podle konfiguračního parametru indexing.context.uri, pro další použití.

5.2 Návrh komponenty

Jak naznačuje specifikace aplikace v kapitole 5.1, bude se jednat o vcelku rozsáhlou aplikaci s mnoha třídami. Jako první krok návrhu musíme nejdříve vyjasnit dílčí procesy zpracování artefaktu, od jeho získání až po uložení metadat do databáze. Tento proces bude obecně popsán v nadcházející kapitole.

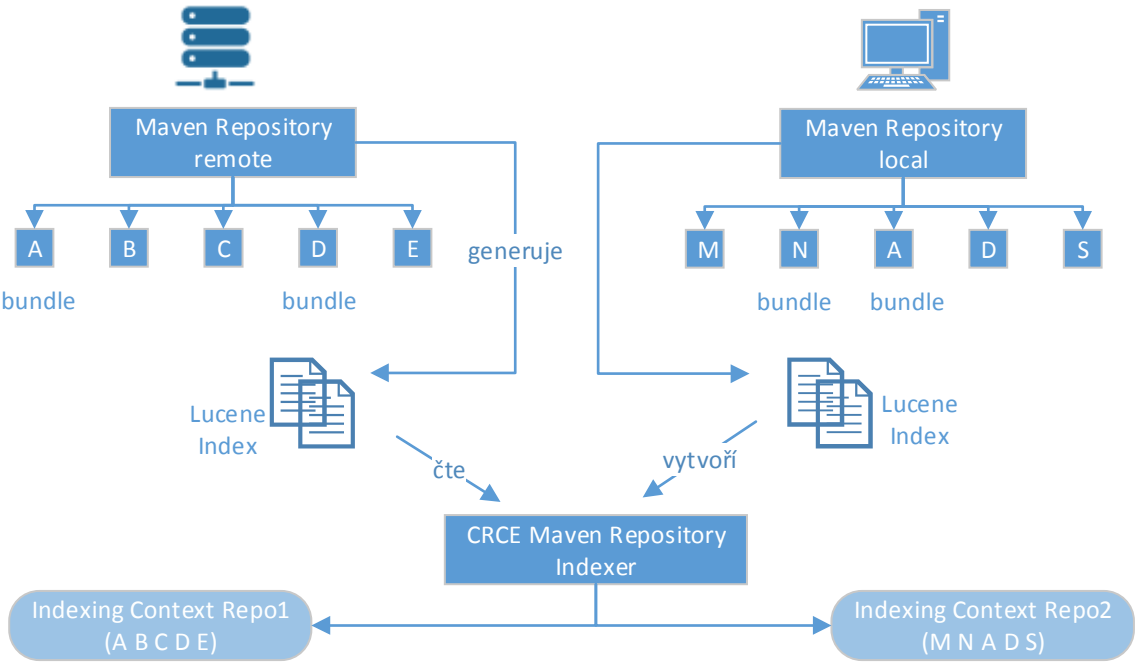
5.2.1 Proces zpracování artefaktu

Detailní zobrazení průchodu artefaktu aplikací by bylo nepřehledné, proto celý proces zobecníme na tři dílčí sub-procesy:

1. Vytvoření prohledavatelného obsahu maven repository (indexing context)
2. Určení množiny požadovaných artefaktů
3. Zpracování informací artefaktů a vytvoření metadat

Krok 1 – Indexing context

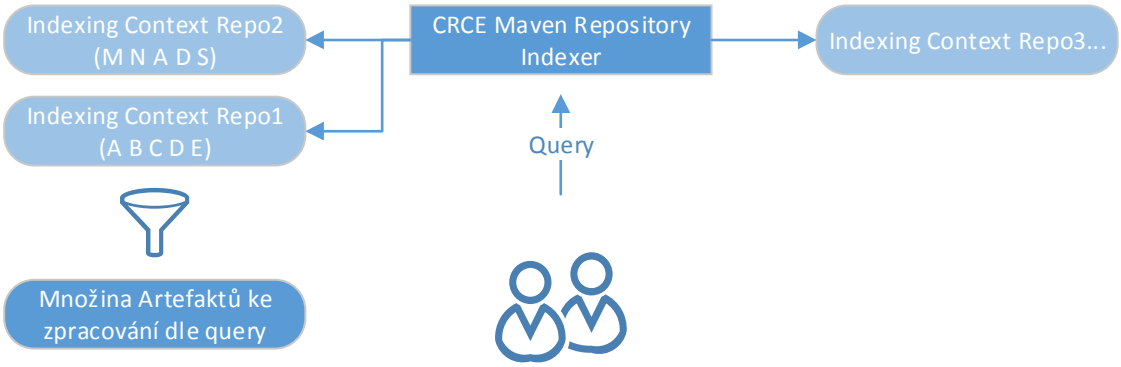
Některá vzdálená úložiště periodicky generují Lucene index, jenž reprezentuje obsah úložiště, čímž je uživatel ušetřen od stahování terabajtů dat z celého uložiště – viz kapitola 4. Není-li Lucene index k dispozici, je zapotřebí ho nejprve vytvořit. Primární krok je tedy implementovat proces, který zpracuje Lucene index úložiště a poskytne uživateli snadno prohledávatelný obsah repository. K tomu bude sloužit třída CRCE pluginu MavenRepositoryIndexer využívající knihovnu Apache Maven Indexer, který vytvoří čtenáři již známý – indexing context. První podproces je znázorněn na nadcházejícím obrázku.



Obr. 5.2.1 – 1 Indexing context

Krok 2 – Filtrace obsahu

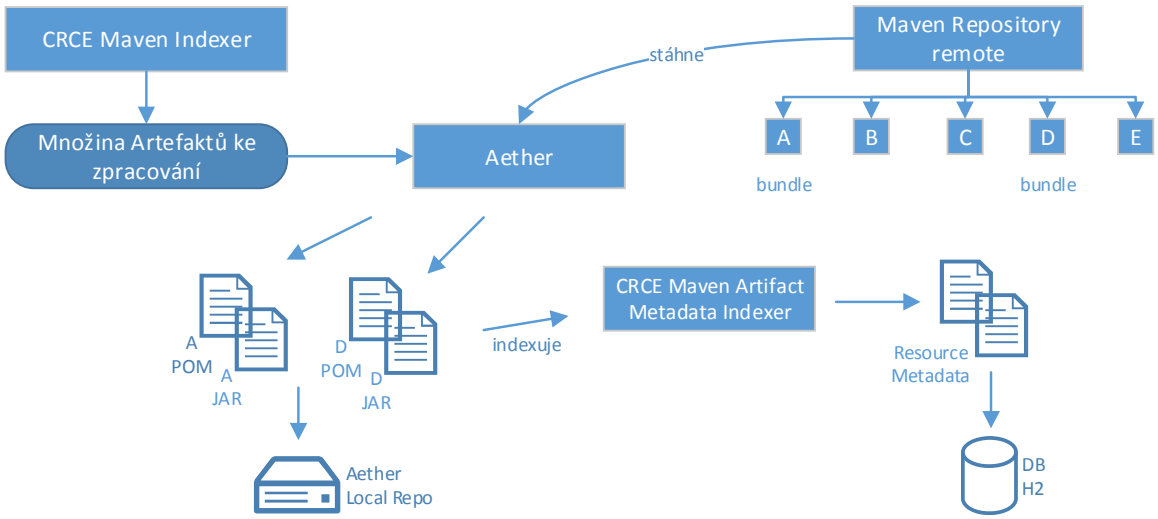
Jakmile dojde k vytvoření kontextu a CRCE získá obsah repository pomocí Lucene indexu, může uživatel definovat jednoduché dotazy nad obsahem, které budou vracet specifické výsledky. Konkrétně budou vytvářeny dotazy podle požadovaných filtrací v kapitole 5.1.8. CRCE Maven Repository Indexer prohledá kontext a vrátí množinu tříd ArtifactInfo, které obsahují základní informace artefaktu – groupId, artifactId a verzi. Tyto výsledky se v dalším kroku předají ke zpracování. Dílčí proces filtrace je zobrazen na obrázku 5.2.1 – 2.



Obr. 5.2.1 – 2 Filtrace artefaktů

Krok 3 – Zpracování artefaktu

Po vyhodnocení množiny artefaktů podle nastaveného filtru prořezávání, zbývá popsat poslední krok, ten nejdůležitější, zpracování artefaktů a uložení jejich metadat do databáze. Objekt CRCE Maven Repository Indexer vyhodnotí každý prvek z množiny pomocí Aetheru, který stáhne artefakt do lokálního úložiště. Druhá fáze procesu se bude odehrávat v třídě CRCE MavenArtifactMetadataIndexer, která obdrží ucelené informace o artefaktu, včetně jeho závislostí. Tyto informace zapíše jako metadata artefaktu do entit a uloží do databáze, viz kapitola 5.1.6. Proces si lze stručně představit podle obrázku níže.



Obr. 5.2.1 – 3 Zpracování metadat

Zpracování artefaktu pomocí nového CRCE pluginu pro maven repository je objasněno. V následujících kapitolách rozebereme technologie a možností sestavení nového CRCE modulu.

5.2.2 Architektura

Z informací získaných v předchozích kapitolách víme, že modul zpracovává příchozí data a musí projít několika procesy, aby bylo možné vyhodnotit entity reprezentující artefakt. Výsledná data bude zapotřebí uživateli nějak prezentovat.

Pokud bychom začínali psát aplikaci z prázdného projektu, nabízelo by se použití vícevrstvé architektury. Postačující by byla třívrstvá architektura, která je obecný typ vícevrstvé architektury.

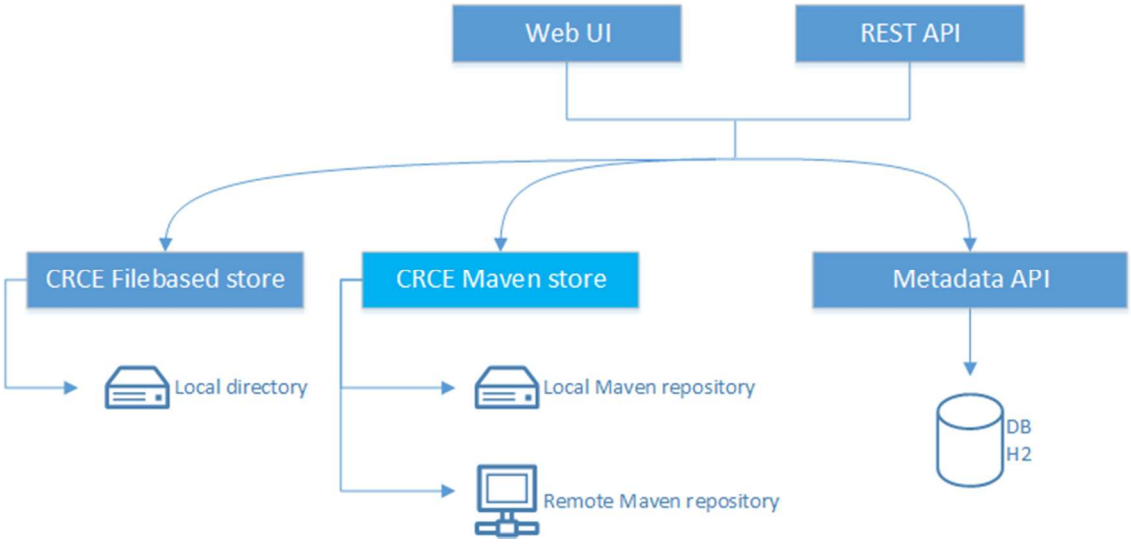
Naštěstí CRCE je postaveno na využití technologie OSGi, popsané v kapitole 2.3. Každá komponenta je samostatný nezávislý celek, který má svoji specifickou úlohu. Komponenty je možné různě přidávat, odebírat, zapínat a vypínat. Díky tomu můžeme jednoduše využívat dosavadní CRCE moduly, které zastávají funkci různých vrstev.

Z odborného pohledu můžeme říct, že CRCE používá vícevrstvou architekturu. Data můžeme prezentovat pomocí existujícího modulu CRCE Web UI, který je aktuálně přizpůsoben původnímu file-based úložišti a není zatím schopen poskytnout plnou obsluhu nového pluginu CRCE pro maven repository. Nicméně základní a nejdůležitější funkcionalita je zachována, a to zobrazení artefaktů a možnost jejich mazání.

Ukládání a načítání dat z databáze má na starosti CRCE komponenta Metadata API. Proto nám nic nebrání použití tohoto rozhraní, k obsluze metadat artefaktů získaných pomocí nového indexeru MavenArtifactMetadataIndexer, který bude zastávat podobnou funkci jako původní OsgiManifestBundleIndexer – kapitola 3.1.3.

Prezentační a datová část je tedy vyřešena a zbývá nám pouze část aplikační, což bude naše komponenta. Modul bude vyvíjen jako OSGi komponenta, která bude konkurovat originálnímu file-based úložišti a bude se nacházet na stejné úrovni.

Uživatel bude moci modul přidat nebo odebrat z obsahu CRCE, případně pouze pozastavit jeho běh a následně opět spustit. Na následujícím obrázku 5.2.2-1 je zobrazen hlavní výřez použití komponenty pro maven repository.



Obr. 5.2.2 – 1 Použití Maven repository pluginu

5.2.3 Formát metadat artefaktu

Jelikož CRCE vyhodnocuje primárně artefakty typu ‘bundle’ můžeme metadata ukládat do základní struktury metadat vycházející z OSGi 5. K perzistenci využijeme rozhraní komponenty Metadata API, která má ER model popsaný v příloze C.

Metadata představující zpracovaný artefakt se uloží do entity Resource. Primární údaje o artefaktu se zapíší do entity Capability. Závislosti se namapují jako Requirement a přiřadí se k entitě Resource. CRCE mírně rozšiřuje OBR databázové schéma a díky tomu není problém vytvořit vnořené Requirementy v případě zápisu tranzitivních závislost. Mapování by mělo být snáze pochopitelné z příkladu zjednodušeného zápisu modelu metadat artefaktu:

Capability – pro popis artefaktu samotného
namespace: 'maven.artifact'
attributes:
'group-id' (String)
'artifact-id' (String)
'version' (Version)
'packaging' (String)
'classifier' (String) – volitelné

Requirement - pro popis závislostí (dependencies) i odkazu na parenta
namespace: 'maven.artifact'
attributes:
'group-id' (String)
'artifact-id' (String)
'version' (Version)
'packaging' (String) – volitelné
classifier (String) – volitelné
directives:
'scope' (String) – nezapsáno v případě 'compile'
'optional' (String) – zapsáno jen v případě 'true'

Protože využíváme existující řešení, metadata můžou v databázi již existovat, ale také je bude stále možné přidat pomocí původního file-based modulu. K odlišení uložených metadat slouží parametr namespace. Každá capabilita a requirement pojené k maven artefaktu bude mít namespace = maven.artifact.

5.2.4 Technologické nástroje pro vývoj

Každá rozsáhlejší aplikace se pohodlně tvoří ve vysokoúrovňovém jazyce, mezi které patří i programovací jazyk Java. Proto i CRCE je napsán tímto jazykem. Mimo Javu CRCE využívá modulární systém OSGi , konkrétně implementaci Apache Felix. Celý projekt je sestavován pomocí nástroje Apache Maven. A každá komponenta má jasně definovaný packaging v pom.xml souboru jako ‘bundle’.

Proto můj CRCE modul pro maven repository byl také vyvíjen v jazyce Java jako OSGi komponenta a sestavován pomocí Mavenu. Aplikace v Jave je možné vyvíjet v různých prostředích, mezi které patří Eclipse nebo Netbeans. Pro tvorbu komponenty byl zvolen Eclipse Luna, z jasného důvodu. Eclipse se nemusí instalovat, je velice modulární, osobně jsem s ním v kontaktu od prvního ročníku na fakultě a používám ho několik let v práci.

Celý projekt využívá decentralizovaného verzovacího systému Git a veškeré zdrojové soubory jsou dostupné na githubu : https://github.com/ReliSA/crce.git . Proto jsem využil Eclipse pluginu EGit , který mi značně ulehčil obsluhu gitu.

Jaké knihovny jsem použil, rovněž postup vytvoření aplikace bude popsán v následující kapitole věnované samotné implementaci.

5.3 Implementace

Každý CRCE modul je OSGi komponentou, která je sestavována pomocí nástroje Apache Maven. CRCE je interně rozděleno na několik podskupin – reaktorů. Hlavní reaktor má následující sub reaktory:

```
<module>pom</module>
<module>build</module>
<module>core</module>
<module>modules</module>
```

Proto je zapotřebí nový modul vytvořit jako projekt s Maven strukturou a v pom.xml deskriptoru je nutné definovat rodiče a veškeré ostatní nutné atributy jako závislosti apod. Nakonec musíme přidat náš modul do reaktoru modules. Kompletní obsah pom.xml souboru nového CRCE pluginu je k dispozici v příloze E. V kořenovém elementu dependencies se nachází přes padesát závislostí, což celkem vypovídá o složitosti pluginu. Představím alespoň základní atributy z deskriptoru.

```
<parent>
   <relativePath>../pom</relativePath>
   <groupId>cz.zcu.kiv.crce</groupId>
   <artifactId>crce-modules-parent</artifactId>
   <version>2.1.0-SNAPSHOT</version>
</parent>

<artifactId>crce-repository-maven-impl</artifactId>
<packaging>bundle</packaging>

<name>CRCE - Maven Repository Implementation</name>
```

Nemá-li artefakt v deskriptoru definovaný element groupId a má odkaz na předka (element parent), groupId se přebírá z rodičovského pom.xml souboru.

PŘEHLED BALÍKŮ

Nejprve je zapotřebí charakterizovat dílčí funkce jednotlivých balíků, do nichž se aplikace člení. Jejich struktura zachycuje základní architekturu komponenty. Poté budou stručně popsány **některé** objekty, které se v nich nachází a v případě nutnosti bude popsána jejich konkrétní funkčnost.

5.3.1 cz.zcu.kiv.crce.repository.maven.internal

Ve výchozím balíku se nacházejí třídy, které reprezentují maven repository, nebo je s ním něco spojuje. Zároveň je zde umístěn Aktivátor modulu a také implementace rozhraní Store.

ACTIVATOR

Aktivátor načte konfigurační soubor, inicializuje příslušné parametry z konfiguračního souboru a uloží je do statických proměnných v třídě MavenStoreConfig, zjistí defaultní URI lokálního či vzdáleného úložiště a vytvoří instanci MavenStoreImpl, která implementuje rozhraní Store.

ARTIFACTRESOLVE

Jedná se o datový typ Enum, který reprezentuje všechna nastavení filtrace zmíněná v kapitole 5.2.8

MAVENREPOSITORYINDEXER

Nejdůležitější třída v balíku, která se stará o vytvoření kontextu úložiště a indexaci artefaktů. Dědí od objektu cz.zcu.kiv.crce.concurency.model.Task a překrývá tak metodu run, čímž je zajištěno asynchronní volání. Díky tomu CRCE nemusí čekat na dokončení procesů mého maven repository modulu a může provádět ostatní činnosti, jako je třeba zobrazování grafického rozhraní.

Ke správné funkcionalitě této třídy bylo zapotřebí přidat některé závislosti do pom.xml souboru, například knihovny Maven Indexeru, Aether, Lucene a další. Maven build sice proběhne v pořádku, ale bylo ještě zapotřebí definovat Embed-Dependency v soboru osgi.bnd, neboť sestavování osgi komponenty probíhá odlišným způsobem.

Třída nejprve vytvoří indexovaný kontext k definovanému úložišti. Zjistí nastavený stupeň filtrace a na základě vytvořeného dotazu (query) předá výsledky k indexaci metadat. Nasledující zápis představuje dotaz na artefakt v indexovaném kontextu s konkrétním groupId, artifactId a požadavkem na packaging typu bundle.

```
Query gidQ = indexer.constructQuery(MAVEN.GROUP_ID, new SourcedSearchExpression(ai.groupId));
Query aidQ = indexer.constructQuery(MAVEN.ARTIFACT_ID, new SourcedSearchExpression(ai.artifactId));
Query pckQ = indexer.constructQuery(MAVEN.PACKAGING, new SourcedSearchExpression("bundle"));

BooleanQuery bq = new BooleanQuery();
bq.add(gidQ, Occur.MUST);
bq.add(aidQ, Occur.MUST);
bq.add(pckQ, Occur.MUST);
```

Dále se v této třídě analyzuje, zda má proběhnout indexace metadat artefaktu přes POM soubory či pomocí stažení celé komponenty ve formě JAR. Jeli zvolena první varianta, vytvořený objekt typu Artifact, nemá přiřazený soubor, neboť JAR soubor neexistuje fyzicky v lokálním Aether repository. Bez cesty k souboru není možné přidat entitu artefaktu do databáze. Proto je zapotřebí zavolat metodu setPOMfile, která nejprve vyhledá soubor podle maven specifikace, viz kapitola 3.3.5 a předá referenci k cestě.

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

V důsledku kladení maximálního důrazu na efektivitu komponenty, bylo zapotřebí volat Aether pouze z jednoho místa, a tím značně ušetřit čas, omezit počet volání na vzdálené servery. Proto jsou během vyhodnocování artefaktu z úložiště, viz kapitola 5.2.1 – krok 3, také současně řešeny přímé nebo tranzitivní závislosti. Výsledný seznam závislostí je spolu s artefaktem zabalen do objektu MavenArtifactWrapper, který je dál předán k procesu zpracování běžné komponenty v aplikaci CRCE.

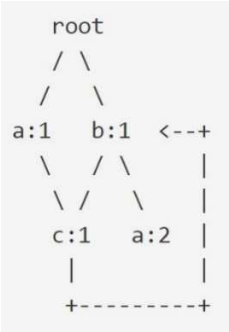
Přímé závislosti vyhodnotí Aether velice snadno, přečtením lineárního seznamu elementu dependencies v deskriptoru artefaktu. Vyřešení tranzitivních (hierarchických) závislostí je poněkud složitější, jelikož může dojít ke smyčkám a tím k zacyklení. Proto Aether tento úkol rozdělí na procesy:

1) Vyhodnocení koordinát artefaktů tvořící tranzitivní závislosti
2) Stažení souborů, které byly nalezeny v kroku 1

V prvním kroku se musí nejprve sestavit graf závislostí, kde kořenový element bude artefakt, k němuž se hledají tranzitivní závislosti. Sestavení grafu probíhá následovně.

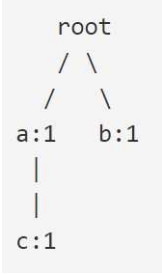
Systém úložiště (kapitola 4.4.1) nejdřív přečte POM soubor každého artefaktu. Z deskriptoru se zjistí závislosti, a jaká další úložiště se mají vzít v úvahu pro vyřešení závislostí. Každou přímou závislost může dependency selector vyloučit z grafu, najde-li například kolizi verzí, duplicitu nebo zacyklení. Selektor může také odebrat závislosti, které nesplňují nadefinovaný scope, nebo pokud je závislost na seznamu výjimek.

Graf závislostí se skládá z instancí DependencyNode, kde každý node reprezentuje jednu závislost s potomky její přímých závislostí. V prvních krocích jsou proto v grafu obsaženy duplicitní závislosti nebo dokonce smyčky:



Obr. 5.3.1 - 1 Zacyklený graf

Jakmile tento graf projde algoritmem řešení konfliktů, duplicitní závislosti se odstraní a dostaneme výsledný strom závislostí.



Obr. 5.3.1 - 2 Strom závislostí

Jako jistou nevýhodu v řešení hierarchických závislostí vidím větší časovou složitost a možnost finálního selhání. Nemá-li Aether prvky v lokálním úložišti, musí vytvořit dotaz pro každou závislost na vzdálený server. Může tak dojít až k tisícům volání serveru při řešení tranzitivních závislostí jednoho artefaktu.

Jeli server přetížen, nebo z nějakého jiné důvodu má responseTime delší než bychom chtěli, může být vyřešení závislostí velmi časově náročné a nakonec také skončit i chybou, pokud nebude možné nějaký deskriptor závislosti z různých důvodů lokalizovat. Potencionálním rizikem jsou většinou různé artefakty s verzí obsahující znaky 'RC' (Release Candidate) kdy obsah POM souborů není ještě ustálený.

MAVENSTOREIMPL

Tato třída implementuje rozhraní Store a s ním spojené přetížené metody. Pomocí dependency manažeru se zavolá metoda start, která asynchronně spustí proces indexace vzdáleného nebo lokálního úložiště.

5.3.2 cz.zcu.kiv.crce.repository.maven.internal.aether

Tento balík obsahuje všechny třídy, které převážně pracují s knihovnou Eclipse Aether. Zde byla zapotřebí pouze jedna třída:

REPOSITORYFACTORY

Obsahuje statické metody, jenž se starají o vytvoření Aether repository systému a k němu vrací příslušnou session. V této třídě se také může vytvářet seznam vzdálených úložišť.

Během vývoje jsem zde narazil na zajímavost. Aether využívá k vytvoření spojení se vzdálenými servery knihovnu Apache HttpClient, kterému předává defaultní parametry:

```
DEFAULT_CONNECT_TIMEOUT = 10 * 1000

DEFAULT_REQUEST_TIMEOUT = 1800 * 1000
```

Hodnoty jsou v milisekundách. Pozornost jistě upoutá druhý parametr DEFAULT_REQUEST_TIMEOUT , který znamená 1 800 sekund, což je v přepočtu až 30 minut. Proto je vhodné při vytváření session nastavit také tyto hodnoty manuálně, například:

```
session.setConfigProperty(ConfigurationProperties.REQUEST_TIMEOUT, "2000");//2s
session.setConfigProperty(ConfigurationProperties.CONNECT_TIMEOUT, "1000");//1s
```

5.3.3 cz.zcu.kiv.crce.repository.maven.internal.metadata

Neméně důležitý balík maven repository modulu. Obsahuje všechny třídy týkající se metadat vyhodnocených artefaktu. Některé třídy budou popsány v následujících kapitolách.

MAVENARTIFACTMETADATAINDEXER

Primární třída má na starosti zjištění a vyplnění informací o získaných artefaktech. K příchozímu artefaktu se zjistí základní koordináty a uloží se jako Capability do entity Resource.

```
Capability cap = metadataFactory.createCapability(NAMESPACE__CRCE_MAVEN_ARTIFACT);
cap.setAttribute(ATTRIBUTE__GROUP_ID, a.getGroupId());
cap.setAttribute(ATTRIBUTE__ARTIFACT_ID, a.getArtifactId());
Version v = new MavenArtifactVersion(a.getBaseVersion()).convertVersion();
cap.setAttribute(ATTRIBUTE__VERSION, v);

if (!(a.getClassifier().equals(""))) {
   cap.setAttribute(ATTRIBUTE__CLASSIFIER, a.getClassifier());
}
cap.setAttribute(ATTRIBUTE__EXTENSION, a.getExtension());
cap.setAttribute(ATTRIBUTE__PACKAGING, "bundle");
```

Metoda createDependencyRequirement vytvoří metadata pro každou příchozí závislost a zjišťuje, zda nemá závislost definovanou verzi jako interval verzí. Dostane-li kladnou odpověď, přidá k entitě Requirement dva nové atributy spolu s příslušným operátorem, například >, <=, <, >=. Výsledný zápis je následovný:

```
r.addAttribute( ATTRIBUTE__VERSION, new MavenArtifactVersion(v.getvMin()).convertVersion(),
                                                                    v.getvMinOperator());

r.addAttribute(ATTRIBUTE__VERSION, new MavenArtifactVersion(v.getvMax()).convertVersion(),
                                                                    v.getvMaxOperator());
```

Zjistí-li třída, že je nastaven parametr na vyhodnocení hierarchických závislostí, zavolá se metoda createDependencyHierarchy, která vytvoří požadovanou stromovou strukturu metadat, pomocí metody solveChildren.

```
private void solveChildren(DependencyNode dn,Requirement requirement) {
    Requirement child = metadataFactory.createRequirement
                                    (NAMESPACE__CRCE_MAVEN_ARTIFACT);
    createDependencyRequirement(dn.getDependency(),child);

    requirement.addChild(child);
    child.setParent(requirement);

    Iterator<DependencyNode> it = dn.getChildren().iterator();

    while(it.hasNext()){
        solveChildren(it.next(), child);
    }
}
```

MAVENARTIFACTVERSION

Neboť implementujeme modul do existující architektury a využíváme rozhraní Metadata API, musíme také dodržovat formát metadat, především ukládání verze. Obsah elementu <version> může být značně jiný oproti specifikaci OSGi. Problém nastává především v zápisu závislostí, kdy verze může obsahovat i takovéto zápisy:

2.23 – aplha-01
(2.1 , 2.2.3)
[1.6,)
0.0.0.4.6.3
0.2_test
1.3.1-SNAPSHOT
4.8.1.SNAPSHOT

Z tohoto důvodu se musí nejprve verze z artefakt deskriptorů vhodně zpracovat a vyhodnotit. Pro každý artefakt se z příchozí verze ve formátu String vyhodnotí části major, minor, micro a qualifier a pokud třída zjistí, že se jedná o definici intervalu verzí, uloží rozsahy verzí do svých proměnných, které vrací na základě jejich žádosti.

Vývoj CRCE pluginu pro maven repository by neměl měnit stávající komponenty, což nebylo úplně možné. Pokud verze artefaktu neobsahuje všechny tři části – major.minor.micro, chybějící části jsou nahrazeny hodnotou -1, nikoliv 0. Neboť například verze 1.3 maven nevyhodnotí stejně jako 1.3.0. K persistenci metadat využíváme existující komponentu Metadata API.

Proto bylo zapotřebí třídu Version z balíku cz.zcu.kiv.crce.metadata.type lehce modifikovat. Konkrétně metodu validate, která kontroluje formát verze. Změna se dotknula podmínek pro části major, minor a micro, kde se hodnota -1 považuje za validní. Dále se musela modifikovat podmínka pro qualifier, kde se znak tečky vyhodnocuje také jako korektní znak.

5.4 Testy a dosažené výsledky

Po implementaci CRCE pluginu pro Maven repository je zapotřebí provést několik testů k ověření zpracování artefaktů a zjištění náročnosti aplikace. Abychom mohli výsledky aplikace považovat za validní, musíme nejprve vypracovat takzvané funkční testy.

5.4.1 Testovací sestava

Veškeré testy probíhaly na počítači s procesorem Intel Core i5 2500K, upraveným na frekvenci 4,6 GHz. Sestava byla osazena čtyřmi 4GB DDR3 paměťovými moduly a systémovým pevným diskem SSD Samsung 120 GB 840 EVO. Tyto komponenty byly osazeny v základní desce ASUS P8P67 B3. Na disku byl nasazen primárně operační systém Windows 7 Professional 64bit. Pro druhý operační systém Ubuntu 14 byl vytvořen virtuální stroj. Celou sestavu napájel zdroj Seasonic 560 Watt. Počítač byl připojen do internetové sítě gigabitovým spojem a realná maximální rychlost dosahovala až 40 MB/s download i upload.

5.4.2 Funkční testy

Tyto testy ověřují základní funkcionalitu jednotlivých úkonů pluginu. Budou zpracovávat předem připravená data. V lokálním úložišti bude malý počet artefaktů pro lepší validaci výsledků. Každý test bude mít přiložené logovací soubory a entity z databáze s detailním vyhodnocením výsledku. Z důvodu obsáhlosti testů, budou k dispozici pouze na přiloženém médiu této diplomové práce. Testy byly cílené k ověření především:

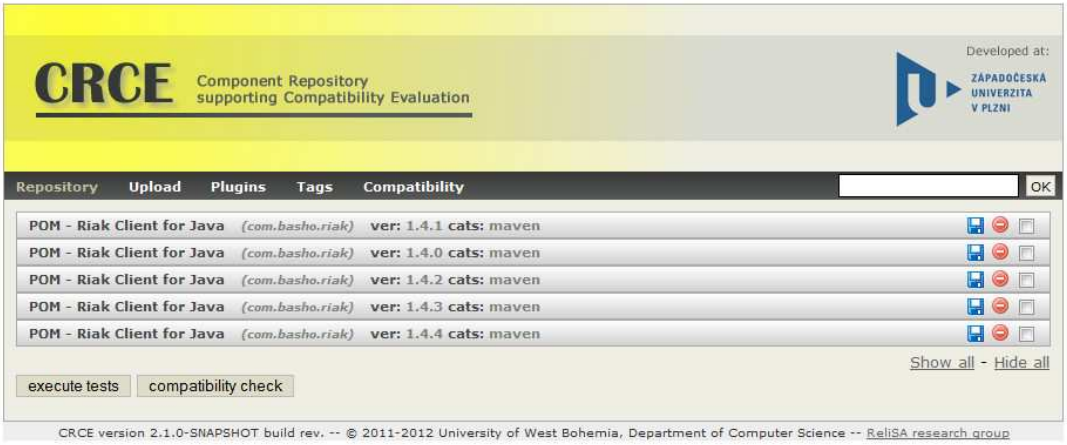
1) Ukládání metadat artefaktů do DB
2) Zpracování metadat pomocí pom.xml souborů
3) Zpracování metadat pomocí *.jar souborů
4) Vyhodnocení přímých závislostí artefaktu
5) Vyhodnocení tranzitivních závislostí artefaktu
6) Získání množiny artefaktů pomocí prořezávacích filtrů

FILTRACE GAV-MINVERZE

Jako ukázku jednoho konkrétního funkčního testu jsem vybral použití filtru groupId-artifactId-minVersion, který vrátí množinu artefaktů od zadané minimální verze. Příklady budou dva. V prvním případě se bude jednat o řešení artefaktů pomocí pom.xml souborů a v druhém případě pomocí souborů *.jar. Pod příklady bude stručné vysvětlení rozdílů.

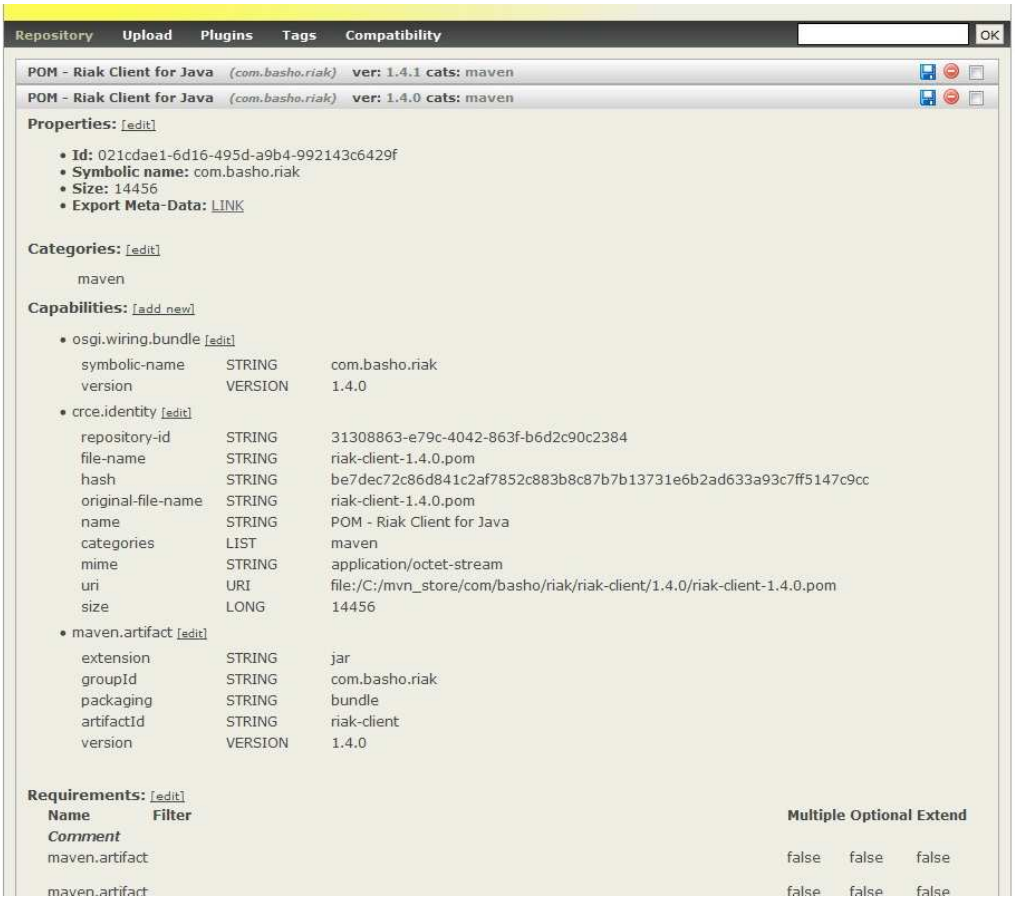
groupId: `com.basho.riak`
artifactId: `riak-client`
min verze: `1.3.6`
dostupné verze: `[1.0.7, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.4.0, 1.4.1, 1.4.2, 1.4.3, 1.4.4]`

Příklad 1: Řešení pomocí pom.xml souborů



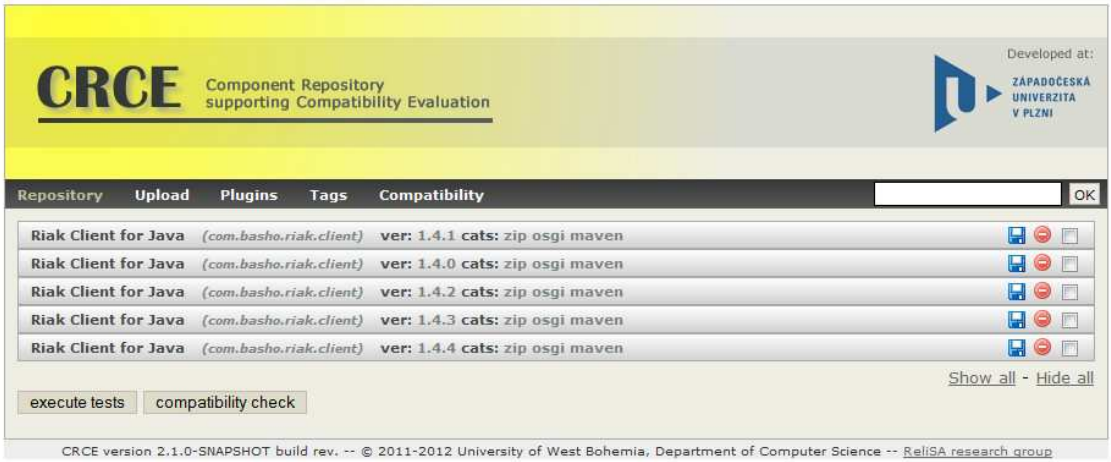
Obr. 5.4.2 – 4 Riak Client seznam (pom.xml)

Detail vybraného artefaktu:



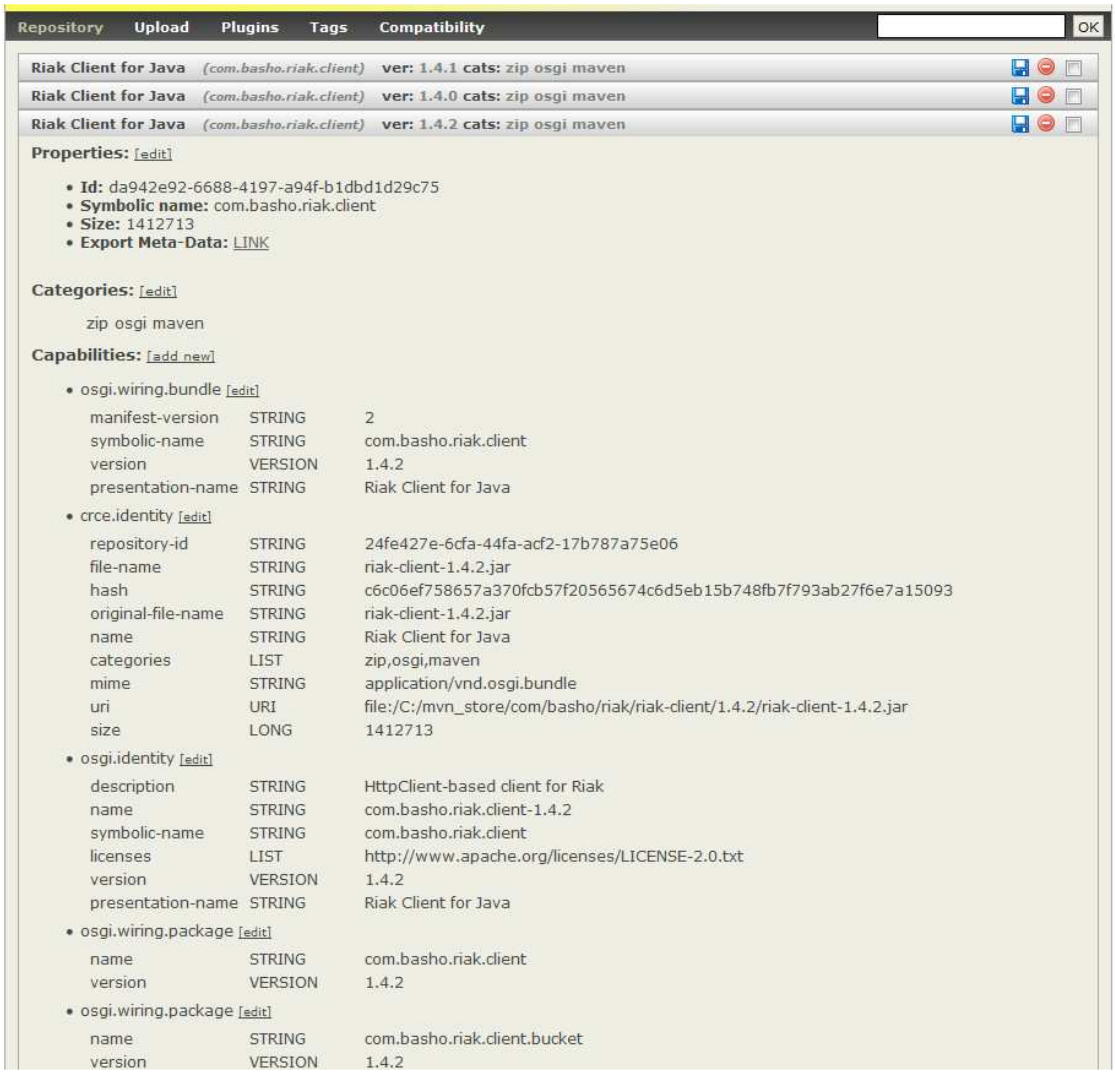
Obr. 5.4.2 – 5 Riak Client detail (pom.xml)

Příklad 2: Řešení pomocí *.jar souborů



Obr. 5.4.2 – 6 Riak Client seznam (*.jar)

Detail vybraného artefaktu:



Obr. 5.4.2 – 7 Riak Client (*.jar)

CRCE zjistilo, že požadavek minimální verze artefaktu je 1.3.6. Nejbližší verze splňující podmínku je 1.4.0, tudíž výsledný interval verzí je [1.4.0, 1.4.4]. V druhém příkladu je vidět, že metadata artefaktů obsahují více informací. Neboť jsou k dispozici JAR soubory a ostatní CRCE indexery, můžou například vyhodnotit další informace z manifestu komponenty. Indexování pomocí pom.xml souborů je pouze výsada CRCE pluginu pro Maven repository.

5.4.3 Výkonnostní testy aplikace

Po ověření funkčnosti aplikace, můžeme spustit hromadné zpracování artefaktů ze vzdálených úložišť a vyhodnotit získané výsledky. K ověření výkonnosti CRCE pluginu pro Maven repository jsem se rozhodl otestovat aplikaci zpracováním artefaktů v nejznámějším a nejdůležitějším Maven úložišti – Central Maven Repository:

http://repo1.maven.org/maven2

Přiložené testy se dělí na dvě části a byly zvoleny tak, aby byly viditelné rozdíly možných parametrů z konfiguračního souboru. První skupina obsahuje testy, kdy aplikace zpracování komponent řešila pouze pomocí pom.xml souborů artefaktů. Druhá část zobrazuje výsledky řešení artefaktů pomocí stahování JAR souborů. Na závěr budou výsledky podrobně analyzovány a porovnány. Sloupce tabulek mají tento význam:

New context - boolean hodnota, zda se při testu vytvářel nový kontext úložiště
- to znamená, jestli se stahoval a zpracovával Lucene index repository

Local empty - boolean hodnota, zda bylo při testování prázdné lokální úložiště
- to znamená, že se musely stáhnout POM nebo JAR sobory artefaktů

Dependencies - tento sloupec zobrazuje, zda byly řešeny přímé nebo tranzitivní závislosti

Filter - definuje, který parametr filtrace byl zvolen (kapitola 5.1.8)

Hits - počet artefaktů ke zpracování, které splňují zvolený filtr

Repo size - velikost lokálního úložiště obsahující zpracované artefakty

Time - celkový čas od vyhodnocení prvního artefaktu až po poslední
- údaje je jsou ve formátu HH:mm:ss

Je-li ve sloupci Local empty hodnota false, znamená to, že v lokálním úložišti byla dostupná požadovaná množina artefaktů, čímž se artefakty nemusely stahovat ze vzdáleného úložiště. Před spuštěním každého testu byl obsah databáze vždy vymazán.

Zaznamenával jsem také dobu, jak dlouho trvalo stáhnout Lucene index vzdáleného úložiště a vytvoření indexovaného kontextu. Tento čas byl téměř konstantní pro všechny filtrace stejného vzdáleného úložiště. Proto jsem pro většinu testů volil parametr remote.repository.update=false (sloupec New context). Výsledný čas vytvoření nového kontextu Central Maven repository byl přibližně **5-7 minut**. Adresář obsahující indexovaný kontext **855 MB**. Protože všechny výsledky se týkají stejného úložiště, tyto parametry jsem do tabulky nepřidával.

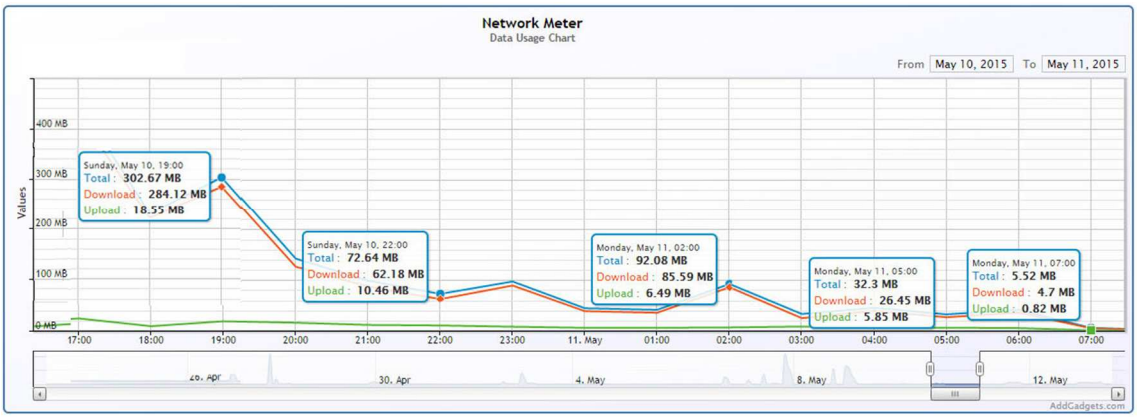
POM INDEXACE

Zobrazené údaje v tabulce reprezentují variace testů, při nichž byly získávány pouze pom.xml soubory artefaktů.

Test	Filter	New context	Local empty	Dependencies	Hits	Repo size	Time
1	All	True	True	Direct	82 779	573MB	11:58:43
2	All	False	False	Direct	82 779	573MB	4:46:09
3	Newest	True	True	Direct	9 505	113 MB	0:29:54
4	Newest	False	True	Direct	9 505	113 MB	0:23:03
5	Newest	False	False	Direct	9 505	113 MB	0:03:49
6	Lowest-micro	False	True	Transitive	34 206	427MB	6:42:04
7	Lowest-micro	False	False	Transitive	34 206	427MB	3:53:27

Tab. 5.4.2 – 1 Výsledky s POM soubory

Test 1 trval nejdéle, protože CRCE muselo zpracovat a stáhnout největší počet artefaktů. Graf síťové zátěže 5.4.2-2 ukazuje počet přenesených dat během průběhu testu. Z obrázku je jasně vidět, že na začátku procesu se nejdříve musel stáhnout Lucene index Central repository. To posunulo křivku značně výše. Vyhodnocováním závislostí artefaktů se plnilo lokální úložiště soubory pom.xml, a proto se také snižoval přenos dat, neboť je při řešení zbývajících artefaktů nemusela aplikace získávat ze vzdáleného úložiště.



Graf 5.4.2 – 2 Síťová zátěž testu 1

Oproti tomu test 2 měl již k dispozici jak připravený indexovaný kontext, tak lokální úložiště se všemy artefakty, které byly získány prvním testem, tudíž trval výrazně kratší dobu. Test 3 ukazuje, že vyhodnocení 9 505 artefaktů, do prázdného úložiště, včetně přímých závislostí a vytvoření kontextu, trvalo přibližně 30 minut. V testu 5 se již nevytvářel kontext ani nebylo zapotřebí stahovat komponenty a proces byl 10x rychlejší oproti testu 3. Poslední dva testy zpracovávaly pouze artefakty, které splňovaly podmínku nejmenší hodnoty micro verze včetně jejich tranzitivních závislostí.

JAR INDEXACE

Tato kapitola popisuje výsledky zpracování artefaktů pomocí celých komponent – JAR souborů, které můžou mít desítky megabajtů.

Test	Filter	New context	Local empty	Dependencies	Hits	Repo size	Time
8	Newest	False	True	Direct	9 505	10,2 GB	4:40:28
9	Newest	False	False	Direct	9 505	10,2 GB	0:13:52
10	Highest-micro	False	True	Transitive	15 254	13,8 GB	7:57:43

Tab. 5.4.2 – 3 Výsledky s JAR soubory

V testu 8 je názorně vidět, že zpracování 9 505 artefaktů včetně JAR souborů a jejich přímých závislostí trvalo téměř pět hodin a získané soubory zabraly 10,2 GB na disku. Při indexaci 15 254 artefaktů (filtr highest-micro) včetně jejich tranzitivních závislostí trval proces téměř osm hodin a vyžadoval 13,8 GB místa na pevném disku.

5.5 Možnosti rozšíření

Aplikace CRCE prošla velkým vývojem za posledních několik let. Nicméně díky výhodám modulárního systému OSGi ji lze kdykoliv jednoduše rozšířit o novou funkčnost pomocí nové komponenty. V nadcházejících odstavcích popíšu některé úpravy, jenž by pro maven repository modul stály za zmínku do budoucna promyslet a případně implementovat.

5.5.1 Repository update trigger

Obsah vzdálených úložišť je aktualizován v intervalech. Proto není zapotřebí při každém spuštění modulu stahovat Lucene index s obsahem repository. Díky tomu existuje parametr pro aktualizaci úložiště v konfiguračním souboru. Tento parametr by bylo vhodné nahradit například inteligentně navrženým triggerem, který by aktualizaci indexu úložiště nastavoval automaticky.

Trigger by se mohl použít i v případě čtení artefaktů z lokálního úložiště, ale to nemá takovou prioritu, jelikož při opakovaném vytváření indexovaného kontextu se nestahují megabajty dat. Nicméně by se i tak ušetřil čas pro vytvoření a překopírování obsahu lokálního repository.

5.5.2 Reindex local cache repository

Během vyhodnocování závislostí, ať už přímých nebo tranzitivních, se zároveň tyto závislosti ukládají do lokálního úložiště. Chceme-li zpracovat také metadata těchto artefaktů, musíme vhodnou metodou zjistit, zda vyhodnocené závislosti byly už o indexovány během procesu čtení úložiště, nebo přibyly jako nové. Například porovnáním obsahu indexované kontextu před a po čtení artefaktů z úložiště metodou IndexingContext.getSize , jenž vrátí hodnotu, která reprezentuje aktuální stav, nikoliv celkový počet prvků v kontextu.

5.5.3 Seznam úložišť a artefaktů

K zvýšení efektivity aplikace by také pomohla implementace možnosti nadefinování seznamu jednotlivých úložišť, případně zpracování nakonfigurovaného seznamu jednotlivých artefaktů pomocí kombinace groupId , artifactId a verze. Samotná implementace by neměla být příliš složitá, jelikož jde pouze o načtení seznamu prvků a samotný proces vyhodnocování nemusí být modifikován.

5.5.4 GUI

Nejzásadnější změnu bych doporučoval v grafickém rozhraní. Celé GUI musí být přepracováno. Implementací mého modulu získalo CRCE jiné možnosti, které by bylo snadnější ovládat pomocí grafického rozhraní.

Například u prvků vyhodnocených pouze za pomoci pom.xml souborů by mohlo být tlačítko ‚Stáhnout JAR‘, čímž by uživatel získal plnohodnotnou komponentu až v okamžiku, kdy by ji opravdu potřeboval.

Úprav v grafickém rozhraní se nabízí nespočet množství, proto jsem nastínil jednoduchý koncept v příloze D. Jde pouze o náčrt, ze kterého je možné vycházet a dále jej modifikovat. Prezentuje využití všech možností implementovaného maven repository modulu.

6 Závěr

Hlavním cílem této práce bylo vytvoření rozšíření aplikace CRCE, které umožní interakci s maven úložišti lokálních či vzdálených. Toto rozšíření mělo umožnit získávání artefaktů z úložišť a z nich zpracované informace ukládat do metadat. Metadata zůstanou uložena v databázi a mohou být dále poskytnuta ostatním komponentám k vyhodnocování dalších informací.

V teoretické části jsem objasnil veškeré pojmy, které bylo nutné znát před vypracováním práce. Největší překážku představovala samotná implementace knihoven Maven Indexer a Eclipse Aether, kde bylo téměř nemožné získat obsáhlou dokumentaci či praktické informace. Celý projekt byl tak vytvářen velmi nepředvídatelně. V závěru implementace se mi však podařilo odstranit veškeré nejasnosti a překážky. Díky tomu jsem byl schopen provést velký refactoring tříd, čímž jsem dosáhl vetší časové efektivity aplikace.

Z výsledků je vidět, že modul pracuje velice rychle. Viz příklad indexace pomocí POM souborů – kapitola 5.4.3. Bohužel grafické rozhraní poskytuje pouze základní obsluhu Maven artefaktů a vyžaduje rozsáhlejší přepracování. Nicméně všechny potřebné funkce ke zpracování maven artefaktů jsou implementovány a připraveny k dalšímu použití.

Práci jsem vypracoval podle požadavků přiloženého zadání a otestoval funkčnost maven repository modulu. Aplikace byla také testována a optimalizována v prostředí Unix systému, neboť na referenčním stroji katedry, na kterém bude CRCE spouštěn, je nasazen operační systém Ubuntu. Tímto považuji práci za úspěšnou a dokončenou.

Literatura

[SCCS] Szyperski Clemens. *Component Software, Beyond Object Oriented Programming SE* Addison-Wesley Longman Publishing Co., 2002, ISBN 0-201-74572-0 2nd edition

[RJ13] Řezníček Jan. *Diplomová práce: Využití uložiště komponent pro podporu aktualizace aplikací.* Západočeská univerzita. Plzeň 2013.

[RZ12] Růžička Zbyněk. *Bakalářská práce: Rozšíření frameworku pro ověřování kompatibility softwarových komponent.* Západočeská univerzita. Plzeň 2012.

[BRJZ] Brada P., Ježek K., *Repository and Meta-Data Design for Efficient Component Consistency Verification.* Science of Computer Programming. Elsevier 2015. http://doi.org/10.1016/j.scico.2014.06.013

[OSFX10] Walid Joseph Gédéon. *OSGi and Apache Felix 3.0 Beginner's Gouide.* Packt Publishing Ltd, 5. 11. 2010 ISBN 978-1849511384

[OIA11] R.S.Hall, K.Pauls, S.McCulloch, D.Savage. *OSGi in Action.* Manning Publications Co. 2011, ISBN 9781 93398 8917

[OSGA] OSGi Alliance. *OSGi Architektura* Dostupné na: http://www.osgi.org/Technology/WhatIsOSGi

[MVNDG] Mike Loukides, Sarah Schneider, *Maven , The Definitive Guide* O'Reilly Media, 2008, ISBN 978-0-596-51733-5

[MVNCB] Maven. *The Complete Reference 2011* Dostupné na: http://books.sonatype.com/mvnref-book/pdf/mvnref-pdf.pdf

[MVNRP] Tutorialspoint. *Maven Repositories* Dostupné na: http://www.tutorialspoint.com/maven/maven_repositories.htm

[STPB] Sonatype Blog. *Maven Indexer: Sonatype's Donation to Repository Search* Dostupné na: http://blog.sonatype.com/2011/02/maven-indexer-sonatypes-donation-to-repository-search/

[LIA05] O.Gospodnetic, E.Hatcher. *Lucene in Action.* Manning Publications Co. 2005, ISBN 1-932394-28-1

[LUCA] Lucene Wiki. *Apache Lucene* Dostupné na: http://lucene.apache.org/core/3_6_2/fileformats.html

[ARTF] JFrog. *Artifactory* Dostupné na: http://www.jfrog.com/artifactory/

[MVIX] Maven. *Maven Indexer* Dostupné na: http://maven.apache.org/components/maven-indexer/

[WIKEC] Eclipse Wiki. *Aether* Dostupné na: http://wiki.eclipse.org/Aether

Přehled zkratek

CRCE	Component Repository supporting Compatibility Evaluation
OBR	OSGi Bundle Repository
API	Application programming interface
POM	Project Object Model
GAV	GroupId ArtifactID Version
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
BFS	Breadth First Search
DFS	Depth First Search
GUI	Graphical User Interface
ER	Entity Relationship
URI	Uniform Resource Identifier
OSGI	Open Service Gateway Initiative
JCR	Java Content Repository

Seznam příloh

A – Uživatelská příručka
B – Konfigurační soubor
C – ER diagram entity Resource
D – Návrh GUI pro Maven repository
E – Obsah POM souboru
F – Obsah přiloženého media

Příloha A – Uživatelská příručka

V této příloze je uveden stručný manuál k používání pluginu CRCE pro Maven repository. Obsah je rozdělen na tyto kapitoly:

1) Instalace
2) Konfigurace pluginu
3) Použití aplikace

A. 1 Instalace

Pro používání CRCE úložiště je nejprve nutné získat zdrojové soubory aplikace a sestavit program pomocí nástroje Maven. K sestavení aplikace je nutné dodržet následující kroky:

1) Nainstalujte JDK, Mongo DB and Apache Maven

- Aplikace byla vyvíjena pomocí Oracle's JDK 1.7 a Maven 3.2.3

2) Proveďte checkout zdrojových souborů z GitHubu projektu (https://github.com/ReliSA/crce.git) z větve maven-based-repository

   Pozn.: Zdrojové soubory je možné také získat z adresáře ‘/src’ z přiloženého media práce

3) Nakopírujte obsah adresáře modules/conf.default do modules/conf a upravte konfigurační soubory – především cesty k úložištím v souboru cz.zcu.kiv.crce.repository.maven-store.cfg

- nakopírujte obsah adresáře ‘/bin/jacc’ z přiloženého media práce do lokálního Maven repository /.m2

- nakopírujte obsah adresáře ‘/jaxb’ z přiloženého media práce do modules/crce-rest/src/main/java/cz/zcu/kiv/crce/rest/internal/jaxb

4) Sestavte aplikaci příkazem `'mvn clean install'` v kořenovém adresáři projektu.

- V případě že se sestavení nezdaří z důvodu negativních testů, spusťte sestavování příkazem `'mvn -Dmaven.test.skip=true -Dfindbugs.skip=true -Dpmd.failOnViolation=false clean install'`

5) Spusťte aplikaci příkazem `'mvn pax:run'` v adresáři modules/

6) CRCE v defaultním nastavení běží na adrese http://localhost:8080/crce.

7) Apache Felix Web Console je dostupná na http://localhost:8080/system/console (login: admin, password: admin)

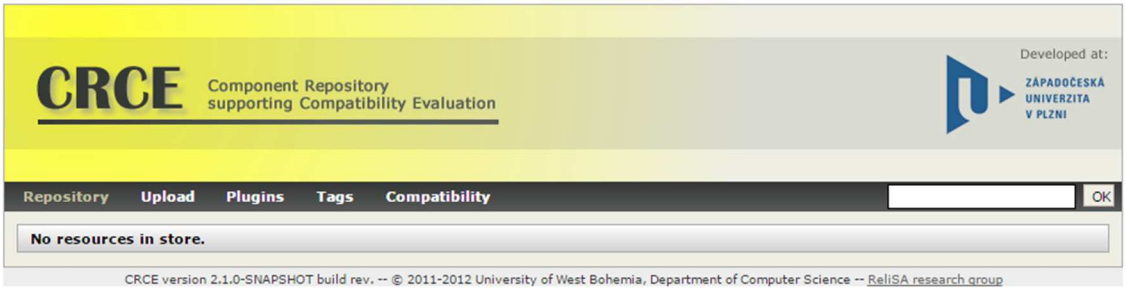
A. 2 Konfigurace pluginu

CRCE plugin pro Maven repository při svém spuštění zpracovává konfigurační soubor cz.zcu.kiv.crce.repository.maven-store.cfg - viz krok 3 v kapitole A.1. Ukázka obsahu souboru je k dispozici v příloze B. Význam jednotlivých parametrů je obsažen v konfiguračním souboru samotném a také v kapitole 5.1. Pomocí této konfigurace se ovládá chování CRCE pluginu pro Maven repository.

A. 3 Použití aplikace

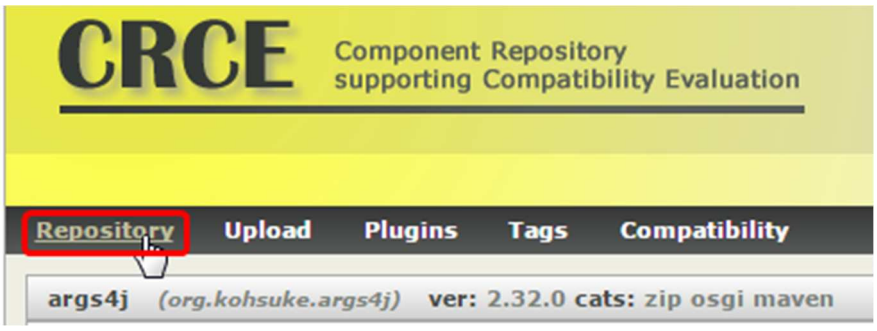
Jakmile je aplikace spuštěna příkazem 5) v kapitole A.1, CRCE plugin pro Maven úložiště začne vyhodnocovat artefakty podle zvolené konfigurace. K zobrazení informací zpracovaných artefaktů slouží grafické rozhraní CRCE.

Aplikace běží na adrese http://localhost:8080/crce , kterou je zapotřebí otevřít v libovolném webovém prohlížeči.



Obr. A.3 – 1 Grafické rozhraní CRCE

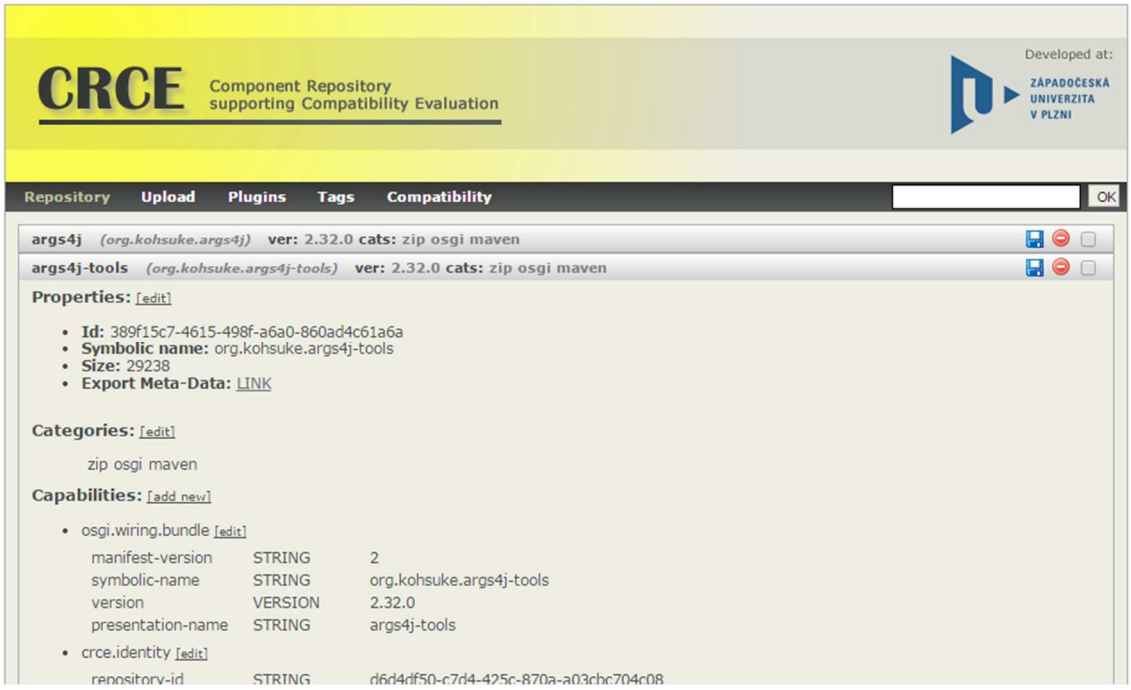
Záložka Repository obsahuje nejen Maven artefakty, ale všechny zpracované komponenty. Kliknutím na odkaz dostaneme obsah CRCE úložiště:



Obr. A.3 – 2 Odkaz - Repository CRCE

Chce-li uživatel získat detailní informace o vybrané komponentě, stačí na ni kliknout a zobrazí se podrobný popis prvku.

Obr. A.3 – 3 Výběr komponenty



Obr. A.3 – 4 Detail komponenty

Vybranou komponentu lze smazat kliknutím na ikonu smazání v pravé části záložky. Komponenta bude odstraněna fyzicky z CRCE repository včetně její metadat.

Obr. A.3 – 5 Smazání komponenty

Seznam komponenty v CRCE úložišti je aktualizován a položka už není dale k dispozici.

Obr. A.3 – 6 Aktualizovaný seznam komponent

Současné grafické rozhraní není přizpůsobeno CRCE pluginu pro Maven repository, proto víc operací s Maven artefakty přes toto rozhraní není umožněno. V příloze D jsem navrhnul koncept nového GUI, pomocí kterého by ovládání Maven artefaktů bylo výrazně příjemnější.

Příloha B – Konfigurační soubor

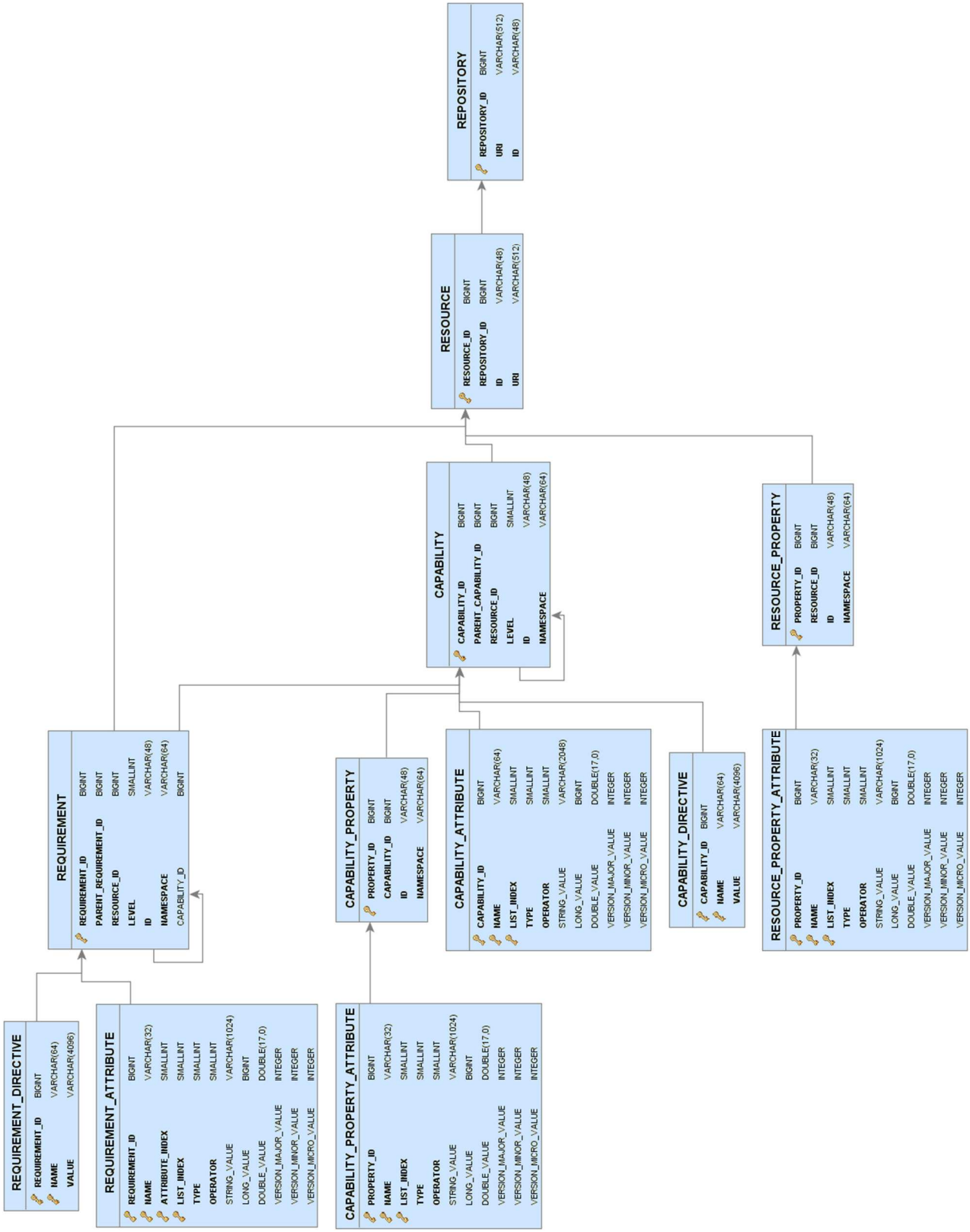
Tato příloha obsahuje ukázku konfiguračního souboru, nutného ke správnému běhu CRCE pluginu pro Maven repository. Reflektuje nastavení parametrů z kapitoly 5.1.

```
# URI of CRCE local maven repository
#
# Valid URIs:
# file:/c:/some/dir         <- on windows
# file:///c:/dir/path        <- on windows
# file:///home/user/  <- on linux
#
# Examples Local:
# local.maven.store.uri=file:/C:/mvn_store
# local.maven.store.uri=mvn_store/test
#
local.maven.store.uri=file:///C:/mvn_store

# Name of local store
local.store.name=local_repository

# Boolean value for update default set repository
# Examples
# update.repository=false
# update.repository=0
local.repository.update=true


#============================================================#
# URI of CRCE remote maven repository
#
# Examples Remote:
# remote.maven.store.uri=http://maven.kalwi.eu/repo/releases
# remote.maven.store.uri=http://repo1.maven.org/maven2
# remote.maven.store.uri=http://relisa-dev.kiv.zcu.cz:8081/nexus/content/groups/public
remote.maven.store.uri=http://repo1.maven.org/maven2

# Name of remote store
remote.store.name=central

# Boolean value for update default set repository
# Examples
# update.repository=false
# update.repository=0
remote.repository.update=true


#============================================================#
# URI of CRCE maven repository indexing context

# Valid URIs:
# file:/c:/some/dir         <- on windows
# file:///c:/dir/path        <- on windows
# file:///home/user/  <- on linux
#
# Examples:
# indexing.context.uri=file:/C:/mvn_store_index
# indexing.context.uri=mvn_store_index
#
indexing.context.uri=file:///C:/mvn_store_index
```

```
#============================================================#
# Define main maven repository local or remote
#
# Examples
# use.remote.maven.store.default=1
# use.remote.maven.store.default=true
use.remote.maven.store.default=true


# Set finding dependencies only direct or nested hierarchy
# true = DFS - not recomended, very time consuming, can fail
aether.find.dependency.hierarchy=false


# Aether switch to download only POM or JARs
# false = aether will try solve artifact by POMS - faster
# true =  aether will try solve artifact by JAR
aether.resolve.artifacts=false


#============================================================#
# Enum which Artifact versions will be resolved
#
# Enum variants
# artifact.resolve=all
# artifact.resolve=newest
# artifact.resolve=highest-major
# artifact.resolve=highest-minor
# artifact.resolve=highest-micro
# artifact.resolve=highest-qualifier
# artifact.resolve=lowest-minor
# artifact.resolve=lowest-micro
#
# Must be set also resolve parameter #
# artifact.resolve=gav
# artifact.resolve.param=org.sonatype.nexus:nexus-api:1.5.0
#
# artifact.resolve=groupid
# artifact.resolve.param=org.sonatype.nexus
#
# artifact.resolve=groupid-artifactid
# artifact.resolve.param=org.sonatype.nexus:nexus-api
#
# Resolve all artifacts with GID an AID and minimal def.version
# artifact.resolve=groupid-artifactid-minversion
# artifact.resolve.param=org.sonatype.nexus:nexus-api:1.2.0

artifact.resolve=newest
```

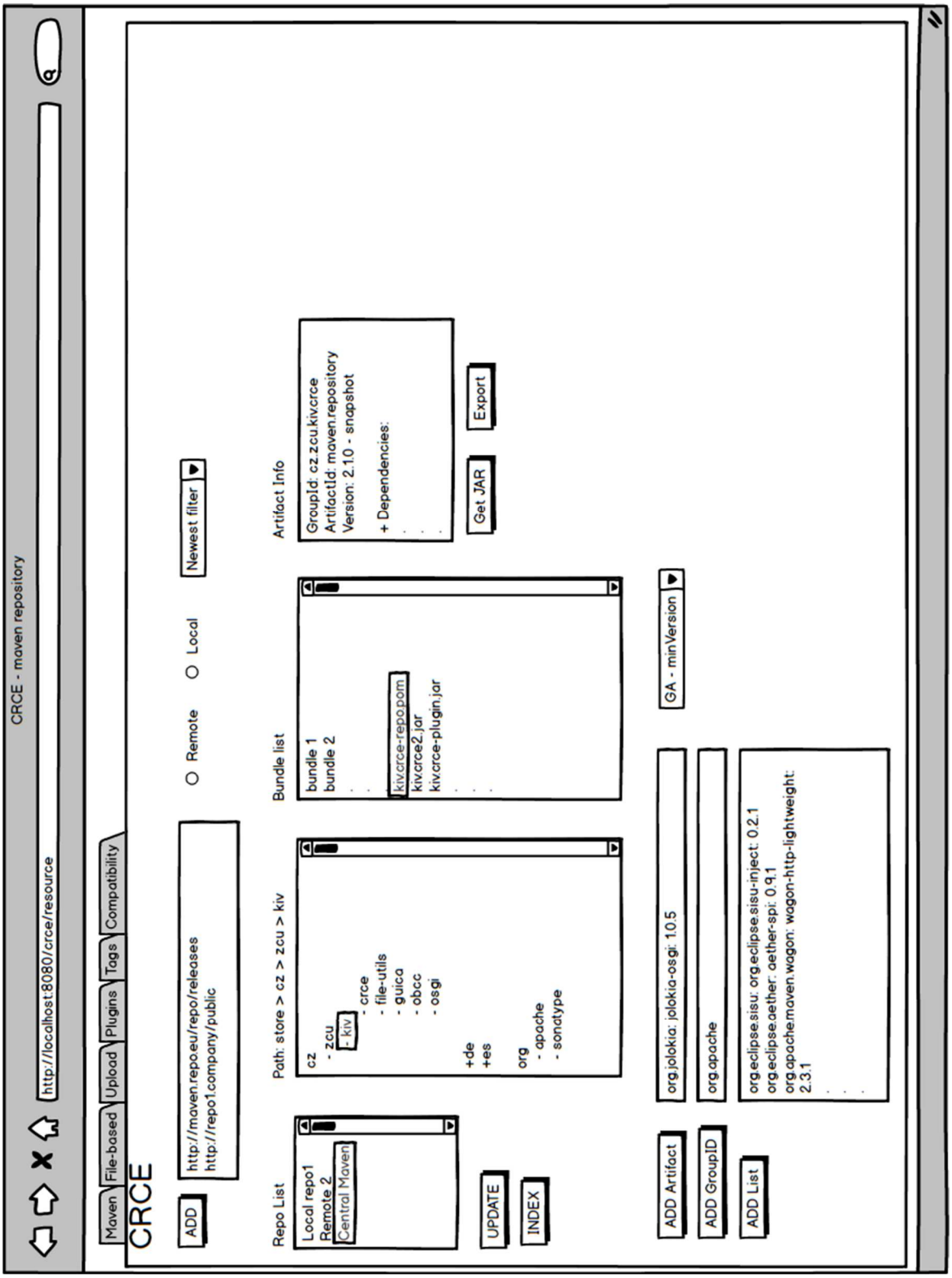
Příloha C – ER diagram entity Resource

Zobrazený diagram reprezentuje vazby mezi entitami metadat artefaktů uložených v databázi.



Příloha D - Návrh GUI pro maven repository

Na následujícím obrázku je navržen koncept možné podoby GUI pro správu CRCE maven úložiště.



Příloha E – Obsah POM

V této části je kopie 1:1 pom.xml souboru CRCE pluginu pro maven repository.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <relativePath>../pom</relativePath>
    <groupId>cz.zcu.kiv.crce</groupId>
    <artifactId>crce-modules-parent</artifactId>
    <version>2.1.0-SNAPSHOT</version>
  </parent>

  <artifactId>crce-repository-maven-impl</artifactId>
  <packaging>bundle</packaging>


  <name>CRCE - Maven Repository Implementation</name>

  <properties>
    <bundle.symbolicName>${namespace}.repository.maven.impl</bundle.symbolicName>
    <bundle.namespace>${namespace}.repository.maven</bundle.namespace>
    <version.org.apache.maven.indexer>5.1.1</version.org.apache.maven.indexer>
    <version.org.eclipse.aether>0.9.1.v20140329</version.org.eclipse.aether>
    <version.org.apache.maven>3.2.1</version.org.apache.maven>
    <findbugs.skip>true</findbugs.skip>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>org.apache.felix.dependencymanager.annotation</artifactId>
      </plugin>
    </plugins>
  </build>

  <dependencies>

    <!-- Felix -->

    <dependency>
      <groupId>org.apache.felix</groupId>
      <artifactId>org.osgi.service.obr</artifactId>
    </dependency>

    <!-- 3rd party dependencies -->

    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
    </dependency>

    <dependency>
      <groupId>org.eclipse.sisu</groupId>
      <artifactId>org.eclipse.sisu.plexus</artifactId>
      <version>0.2.1</version>
      <exclusions>
        <exclusion>
          <groupId>org.codehaus.plexus</groupId>
          <artifactId>plexus-classworlds</artifactId>
        </exclusion>
      </exclusions>
```

```
</dependency>

<dependency>
  <groupId>org.eclipse.sisu</groupId>
  <artifactId>org.eclipse.sisu.inject</artifactId>
  <version>0.2.1</version>
</dependency>
<dependency>
  <groupId>org.sonatype.sisu</groupId>
  <artifactId>sisu-guice</artifactId>
  <version>3.2.1</version>
</dependency>
<dependency>
  <groupId>org.codehaus.plexus</groupId>
  <artifactId>plexus-classworlds</artifactId>
  <version>2.5.1</version>
</dependency>
<dependency>
  <groupId>com.google.inject</groupId>
  <artifactId>guice</artifactId>
  <version>3.0</version>
</dependency>

<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-api</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-impl</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-spi</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-connector-basic</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-transport-file</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-transport-http</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>
<dependency>
  <groupId>org.eclipse.aether</groupId>
  <artifactId>aether-util</artifactId>
  <version>${version.org.eclipse.aether}</version>
</dependency>


<!-- 3rd party wrapped dependencies -->

<dependency>
  <groupId>cz.zcu.kiv.crce.wrapper</groupId>
  <artifactId>org.apache.httpcomponents.httpclient</artifactId>
  <version>4.2.6</version>
</dependency>
<dependency>
  <groupId>cz.zcu.kiv.crce.wrapper</groupId>
  <artifactId>org.apache.httpcomponents.httpcore</artifactId>
  <version>4.2.5</version>
</dependency>
<dependency>
```

```
    <groupId>cz.zcu.kiv.crce.wrapper</groupId>
    <artifactId>org.codehaus.plexus.plexus-utils</artifactId>
    <version>3.0.17</version>
</dependency>
<dependency>
    <groupId>cz.zcu.kiv.crce.wrapper</groupId>
    <artifactId>org.codehaus.plexus.plexus-interpolation</artifactId>
    <version>1.19</version>
</dependency>
<dependency>
    <groupId>cz.zcu.kiv.crce.wrapper</groupId>
    <artifactId>org.codehaus.plexus.plexus-component-annotations</artifactId>
    <version>1.5.5</version>
</dependency>

<!-- Embedded dependencies -->

<dependency>
    <groupId>org.apache.maven.indexer</groupId>
    <artifactId>indexer-artifact</artifactId>
    <version>${version.org.apache.maven.indexer}</version>
</dependency>
<dependency>
    <groupId>org.apache.maven.indexer</groupId>
    <artifactId>indexer-core</artifactId>
    <version>${version.org.apache.maven.indexer}</version>
</dependency>
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-model</artifactId>
    <version>${version.org.apache.maven}</version>
</dependency>
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-model-builder</artifactId>
    <version>${version.org.apache.maven}</version>
</dependency>
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-aether-provider</artifactId>
    <version>${version.org.apache.maven}</version>
</dependency>
<dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-repository-metadata</artifactId>
    <version>${version.org.apache.maven}</version>
</dependency>
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-core</artifactId>
    <version>3.6.2</version>
</dependency>
<dependency>
    <groupId>org.glassfish.hk2.external</groupId>
    <artifactId>javax.inject</artifactId>
    <version>2.2.0</version>
</dependency>

<dependency>
    <groupId>org.sonatype.aether</groupId>
    <artifactId>aether-api</artifactId>
    <version>1.13.1</version>
</dependency>
<dependency>
    <groupId>org.sonatype.aether</groupId>
    <artifactId>aether-util</artifactId>
    <version>1.13.1</version>
</dependency>

<!-- Project dependencies -->

<dependency>
    <groupId>${project.groupId}</groupId>
```

```
    <artifactId>crce-core</artifactId>
    <type>pom</type>
  </dependency>

  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>crce-concurrency</artifactId>
    <version>${project.version}</version>
  </dependency>

  <!-- Test dependencies -->

  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  </dependency>
  <dependency>
    <artifactId>org.apache.felix.dependencymanager.annotation</artifactId>
    <groupId>org.apache.felix</groupId>
    <exclusions>
      <exclusion>
        <artifactId>maven-project</artifactId>
        <groupId>org.apache.maven</groupId>
      </exclusion>
    </exclusions>
  </dependency>


  <dependency>
        <groupId>cz.zcu.kiv.crce</groupId>
        <artifactId>crce-metadata-api</artifactId>
        <version>2.1.0</version>
        <type>bundle</type>
  </dependency>
  <dependency>
        <groupId>cz.zcu.kiv.crce</groupId>
        <artifactId>crce-metadata-dao-api</artifactId>
        <version>2.1.0</version>
        <type>bundle</type>
  </dependency>
  <dependency>
        <groupId>cz.zcu.kiv.crce</groupId>
        <artifactId>crce-plugin-api</artifactId>
        <version>2.1.0</version>
        <type>bundle</type>
  </dependency>
  <dependency>
        <groupId>cz.zcu.kiv.crce</groupId>
        <artifactId>crce-repository-api</artifactId>
        <version>2.1.0-SNAPSHOT</version>
        <type>bundle</type>
  </dependency>
  <dependency>
        <groupId>cz.zcu.kiv.crce</groupId>
        <artifactId>crce-resolver-api</artifactId>
        <version>2.1.0-SNAPSHOT</version>
        <type>bundle</type>
  </dependency>
  <dependency>
        <groupId>cz.zcu.kiv.crce</groupId>
        <artifactId>crce-metadata-indexer-api</artifactId>
        <version>2.1.0-SNAPSHOT</version>
        <type>bundle</type>
  </dependency>

  <dependency>
         <groupId>org.apache.maven</groupId>
        <artifactId>maven-artifact</artifactId>
        <version>3.2.5</version>
    </dependency>

<!-- For ResourceFetcher implementation, if used -->
 <dependency>
```

```
            <groupId>org.apache.maven.wagon</groupId>
            <artifactId>wagon-http-lightweight</artifactId>
            <version>2.3</version>
            <scope>compile</scope>
    </dependency>

    <dependency>
            <groupId>org.apache.maven.wagon</groupId>
            <artifactId>wagon-provider-api</artifactId>
            <version>2.3</version>
    </dependency>

    <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-highlighter</artifactId>
            <version>4.8.1</version>
    </dependency>

    <dependency>
            <groupId>cz.zcu.kiv.crce</groupId>
            <artifactId>crce-metadata-api</artifactId>
            <version>3.0.0-SNAPSHOT</version>
    </dependency>

  </dependencies>

</project>
```

Příloha F – Obsah přiloženého media

Součástí této práce je rovněž i paměťové médium (DVD) obsahující tyto adresáře a soubory:

- readme.txt – textový soubor obsahující popis a význam jednotlivých adresářů
- /bin – obsahuje jacc maven artefakty, potřebné k sestavení aplikace
- /doc – obsahuje PDF verzi této práce
- /jaxb – obsahuje vygenerované jaxb třídy potřebné k sestavení aplikace
- /src – obsahuje zdrojové kódy aplikace CRCE včetně pluginu pro Maven repository
- /test – obsahuje výsledky funkčních testů