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

Diplomová práce

Rozšíření úložiště komponent o externí zdroje dat

Plzeň, 2019 Bc. Roman Pešek

Prohlášení:

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

V Plzni dne 28. 4. 2019 . . . . . .

Poděkování:

Tímto vyjadřuji poděkování vedoucímu práce Doc. Ing. Přemyslu Bradovi, MSc., Ph.D., za cenné rady a připomínky, velkou vstřícnost a ochotu mi vždy poradit a pomoci.

Abstract:

The diploma thesis deals with the extension of the Component Repository supporting Compatibility Evaluation (CRCE) to include gathering artefacts from external data sources and compiling, maintaining, storing, versioning and distributing sets of components which contain artefacts from CRCE repository or sets of components previously created.

The theoretical part of the thesis addresses the properties of the component-based architecture, its advantages and disadvantages, especially in the area of the compatibility theory. The thesis introduces OSGi technology and presents implementation examples with emphasis on gathering artefacts from external sources, compiling, maintaining versions and applying sets of artefacts as a complex application.

In the practical part of the thesis, the extension of CRCE repository to include data sources with the support of Aether library, indexation and full-text search in the central Maven repository was designed and implemented. Furthermore, maintaining, versioning and distributing sets of components were carried out. For the user comfort, the forms in the new web application on Vaadin platform were extended to include the support of new functionalities.

Abstrakt:

Diplomová práce pojednává o rozšíření úložiště komponent podporující kontrolu kompatibility CRCE o získávání artefaktů z externích zdrojů dat a sestavování, správu, ukládání, verzování a distribuce množin komponent, které jako své části obsahují artefakty z úložiště CRCE nebo již dříve vytvořené množiny.

Teoretická část práce obsahuje charakteristiku komponentové architektury aplikací, její přednosti i zápory, a to zejména v oblasti teorie kompatibility. Pojednává o technologiích pro komponenty OSGi a uvádí přehled implementačních příkladů s důrazem na získávání artefaktů z externích zdrojů, sestavování, správu verzí a nasazování množin artefaktů jako celistvé aplikace.

V praktické části práce je navrženo a implementováno rozšíření úložiště CRCE o zdroje dat s podporou knihovny Aether, indexace a fultextové vyhledávání v centrálním úložišti Maven. Dále byla realizována správa, verzování a distribuce množin komponent. Pro uživatelský komfort jsou v nové webové aplikaci na platformě frameworku Vaadin doplněny a rozšířeny formuláře o podporu nových funkcionalit.

Obsah

1 Úvod

Informační technologie od svého vzniku procházely vždy dynamickým vývojem. V poslední době je tento trend zvláště patrný z pohledu vývoje software, respektive aplikací. Velmi obtížně by se dnes hledal program, který bude v nezměněné podobě užíván několik let. Pravděpodobně jej bude nutné přizpůsobit ať již požadavku uživatelů, úpravou dalších spolupracujících programů nebo nově objeveným bezpečnostním hrozbám.

1.1 Komponentově orientovaná architektura

Architektura aplikace by měla být zvolena k danému účelu a s ohledem na další potenciálně plánované využití. V poslední době lze zaznamenat trend vývoje aplikací, které po skončení své funkce, pro kterou byly vyvinuty, či při její změně, jsou kompletně nahrazeny aplikací novou. Odůvodněný tento směr vývoje je u mobilních aplikací a také v případě použití aplikačních rámců, které prochází dynamickým vývojem.

Kompromisem, který se snaží minimalizovat dopady dynamických změn vlivem požadavků na funkčnost a při změnách okolního prostředí (myšleny jsou i změny aplikačních rámců), je použití architektury, která rozdělí aplikaci na menší spolupracující celky (komponenty). Opodstatněné použití komponentové architektury aplikace je v případě větších např. podnikových aplikací, u kterých lze předpokládat časté úpravy vlivem měnících se požadavků zákazníka. Právě v tomto případně vyniknou výhody, jakými jsou zejména: znovupoužitelnosti samostatných celků aplikace, aktualizace pouze těch částí, kterých se to exaktně týká, a přehlednost pro vývojáře aplikace.

Existují ospravedlnitelné nevýhody, jakými je udržení vzájemné kompatibility jednotlivých částí aplikace (komponent). Problematikou kompatibility komponent se zabývá výzkumná skupina ReliSA na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

1.2 Motivace a cíl práce

V rámci výzkumu skupiny ReliSA byla vytvořena aplikace úložiště komponent podporující jejich vzájemnou kompatibilitu (CRCE). V současné verzi CRCE je umožněno ukládat artefakty do úložiště pouze jako lokální soubory nebo dostupné url odkazem. Cílem diplomové práce je navrhnout rozšíření úložiště o možnost používat artefakty z jiných (externích) úložišť. Zamyšleny jsou zejména Maven

úložiště artefaktů, respektive splňující aplikační rozhraní Maven úložišť, ale toto není nutnou podmínkou. V současné době existuje množství nástrojů na sestavování aplikací z komponet a správu verzí. Cílem práce je analyzovat tyto nástroje a navrhnout způsob definování množin artefaktů z úložiště CRCE pro vytváření aplikací a spravování jejich verzí.

2 Historie architektur aplikací

Z hlediska softwarového inženýrství lze hovořit o architektuře aplikace v případě, kdy není tvořena jedním celistvým programem, ale více částmi (vrstvami), které spolu navzájem spolupracují. I v současné době ovšem existuje mnoho řešení aplikací, které potrádají definici spolupracujících částí. Je to zejména s ohledem na historicky vytvořené funkčnosti, které by bylo možné přeprogramovat do novějších technologií jen s velkými obtížemi a velkými finančními nároky.

Například úzce integrované aplikace s rozsáhlými databázovými technologiemi na historicky starší platformě stále ještě využívají některé instituce (Informix a jazyk 4gl). Tyto aplikace mohou spolehlivě poskytovat funkčnost po dobu několika let, ovšem jakékoliv rozsáhlejší změny či úpravy nejsou možné. Zejména s ohledem na specifický jazyk a nedostatek kvalifikovaných vývojářů dané platformy.

Speciální případem jsou jednoduché aplikace, které poslouží svému účelu a je možné je jednoduše nahradit novějšími. V současné době je tento trend patrný v oblasti aplikací pro mobilní zařízení. Za těchto okolností zřejmě není nutné vytvářet složitou architekturu.V následujících odstavcích bude uveden základní přehled architektur a možnosti jejich užití. Důvodem je stručné porovnání s komponentově orientovanou architekturou aplikací.

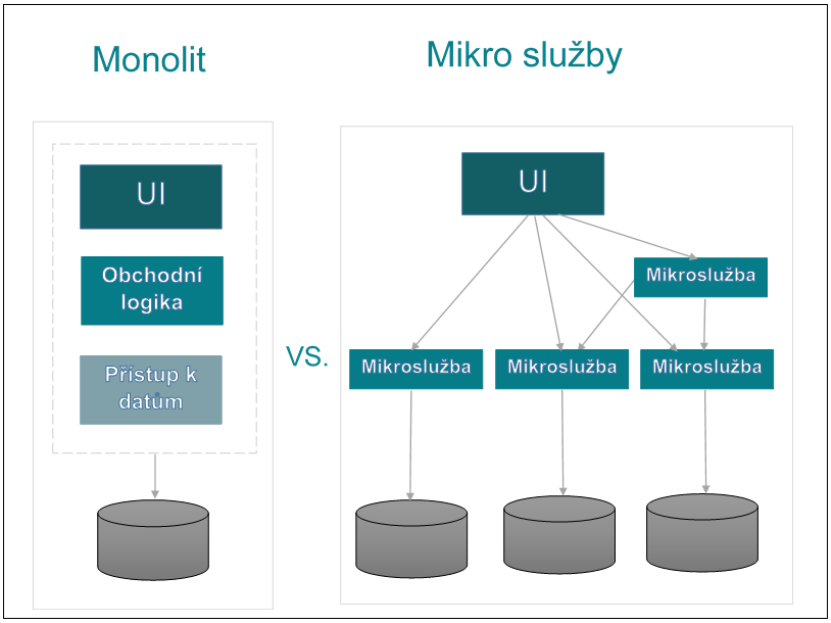
Jedním z historicky nejstarších uspořádání je **klient-server**. Jedná se technologii přenesení části výpočetní zátěže na výkonější počítač (server) a také rozdělení programu na sdílené části, které vykonává jeden (nebo i více) společný počítač (server). Tento model je aplikován například v síťových službách (DHCP, DNS), volání vzdálených procedur (RPC), webových stránkách (HTTP), webových službách (REST, SOAP). Výhodou je snadnější údržba, kdy při dodržení společného rozhraní a při změně na serverové části není klientská aplikace ovlivněna. Oproti samostatným klientským programům je zřejmou nevýhodou síťová zátěž a problém s případnou nefunkčností (výpadkem) serverové části programu.

Třívrstvá architektura - již ze svého názvu je patrné, že aplikace je rozdělena na tři navzájem spolu komunikující celky. Jedná se o logické členění na databázovou, aplikační a prezentační vrstvu. Velký rozvoj zaznamenala tato architektura u webových aplikací v letech 2005-2010. Důvodem měla být jednodužší úprava, změna či úplné nahrazení kterékoliv vrstvy bez vlivu na ostatní. Avšak údržba rozsáhlé podnikové aplikace, která je založená na třívrstvé architektuře, je s ohledem na narůstající úpravy a rozsah implementovaných funkčností do budoucna značně obtížná.

Obdobná situace je u aplikací založených na modelu **MVC (Model-View-Controller)**. V tomto případě Model představuje aplikační a datovou vrstvu, View zobrazení dat uživateli na základě změn v modelu nebo na základě požadavků, které zpracovává vrstva Controller.

Z předchozích odstavců je zřejmé, že pro udržitelný rozvoj je nutné architekturu aplikace rozdělit na ještě více samostatných, ale navzájem komunikujících částí (komponent - **Komponentová architektura**). Míra dělení je závislá na daném kontextu. Situaci je možné představit na modelu stavby domu - pro porovnání stavba domu z cihel a oproti tomu z prefabrikovaných panelů. Komponenta, ve smyslu cihly, představuje malou jednotku. Stavba při použití této komponenty je složitější a náročnější, ale je možné dům kompletně přizpůsobit požadavkům. Oproti tomu komplexnější komponenty (myšleno připravené prefabrikáty) znamenají rychlejší a jednodušší stavbu, ale za cenu určitého přizpůsobení tvaru stavby. Analogicky si lze představit tuto situaci u aplikací tvořených z komponet.

Aplikace sestavená z malých částí znamená dále jednodušší aktualizaci na základě požadavků na změnu, samozřejmě za předpokladu dodržení standardů rozhraní mezi jednotlivými komponetami aplikace. Specifickým případem komponentové architektury je aplikace složená z mikroslužeb (microservices). Mikroslužby jsou architektonickým přístupem ke sestavování aplikací. Architekturu mikroslužeb tvoří rozdělení tradičních monolitických systémů do základních funkcí. Každá funkce se nazývá služba a může být nezávisle nasazena či odstraněna. Tento přístup znamená, že jednotlivé služby mohou fungovat (i selhat) bez negativního ovlivnění ostatních. [Mic] Na obrázku číslo 1 je uvedeno schematické porovnání klasické monolitické architektury s architekturou mikroslužeb.



Obrázek 1: Monolitická architektura vs. architektura mikroslužeb

Zvláštním typem architektury aplikací je **zpracovávání dat v reálném čase** nebo **zpracovávání velkých objemů dat**. Analýza dat v reálném čase se zabývá datovými proudy s žádnou nebo minimální latencí. Jedná se například o proces (kódování/dekódování) zvuků či videa nebo zpracovávání dat ze senzorů (monitoring). Architektura pro analýzu velkých objemů dat je speciálně navržená pro příjem, zpracování a vyhodnocování takového množství záznamů, které jsou pro tradiční databázové systémy příliš rozsáhlé a složité.

3 Softwarové komponenty

Již v roce 1968 Doug Mcllroy[1] na konferenci softwarového inženýrství (SE) uvedl v tématu „Mass-Produced Components“ koncept programových komponent. Softwarové komponenty mají historicky své místo v SE, respektive software složený z komponent je jedním ze základních inženýrských principů. Na mezinárodní konferenci o softwarovém inženýrství v roce 1998 byla představena koncepce komponentové softwarové techniky (CBSE) jako specifická oblast v rámci SE. Byla řešena témata specifická pro oblast CBSE, jako je kompatibilita, předvídatelnost funkčních i extrafunkčních vlastností, modelování komponentních systémů, opakovatelnost použití, nasazení, běhové prostředí (middleware). [CSS11]

3.1 Specifika CBSE

Cílem softwarového inženýrství založeného na komponentách je vytváření aplikací z již existujících součástí nebo vzájemným nahrazováním těchto součástí, tzn. vyvíjet komponenty jako opakovaně použitelné entity. Tento přístup znamená určité změny z pohledu technického i obchodního. Je nutné se zamyslet jednak nad přístupem „zhora dolů“, tzn. analýza dané problematiky a její rozdělení na dílčí problémy, a jednak nad přístupem „zdola nahoru“, tzn. opětovného použití již hotových produktů a jejich skládání do funkčních komplexních celků.

Softwarové inženýrství zaznamenalo v tomto smyslu v posledních letech významný pokrok ve vývoji komponent. Technologie jako OSGi a .NET jsou široce používány v mnoha aplikačních oblastech. Mnoho softwarových společností si vytvořilo vlastní technologie založené na komponetách. Zájem o CBSE se přesunul do oblasti služeb. Jedná se o integraci společných funkčností do sběrnice (ESB) a vystavení veřejných rozhraní komponent. Také vzrůstá praktické využití oblasti dynamických aplikací, které tvoří základní prvky architektury orientované na služby, webové služby, distribuované systémy a cloud computing.

I přes výše uvedené výhody zaznamenalo CBSE určité neúspěchy. Zejména koncepty komponentových trhů, které by umožnily automatické vyhledávání a nasazování komponent, zůstávají realizovány v malém měřítku a mnoho konceptů vyvinutých ve výzkumné komunitě, jako jsou důkazy formálních součástí, byly jen částečně uvedeny do praxe.

[1] Malcom Douglas Mcllroy (nar. 1937) - matematik a programátor. Od roku 2007 profesorem informatiky na Dartmonth College. Průkopník a propagátor softwarového inženýrství založeného na komponentách.

3.2 Terminologie CBSE a základní koncepce

Definovat softwarovou součást není jednoduché. Není zřejmé, zda se jedná o část navrhnutého UML diagramu, součást systému či databáze. Termín softwarová součást je stejně starý jako softwarové inženýrství samotné, ale jeho definice a související terminologie zůstávají předmětem intenzivních diskusí.

3.2.1 Terminologie

Softwarová komponenta a komponentový model - nejčastěji citovaná definice softwarových komponent je kompoziční jednotka se smluvně specifikovanými rozhraními a pouze explicitní závislostí na kontextu. Softwarová součást nasazena nezávisle a využita třetími stranami. Jiná definice zdůrazňuje roli komponentového modelu: „Softwarová součást je softwarový prvek, který náleží danému modelu komponent a může být samostatně nasazen nebo odstaněn bez modikace dle kompoziční normy“. Model se zde definuje jako soubor norem pro implementaci komponent, pojmenování, vzájemnou součinnost, přizpůsobení, složení, vývoj a nasazení. Druhá definice dává součásti odlišný význam - individuální jednotka komponentového modelu. [CSS11]

Rozhraní - definuje soubor funkčních vlastností součásti neboli sadu akcí, které jsou srozumitelné jak poskytovateli rozhraní (vlastní komponenty), tak i uživateli (jiné komponenty nebo jinému software, který komunikuje s poskytovatelem). Rozhraní má dvojí roli - specifikuje komponentu a je také prostředkem pro interakci mezi komponentou a prostředím. V modelu CBSE se rozlišují komponenty, které poskytují své součásti okolnímu prostředí (poskytované rozhraní) a dále činnosti, které komponenty od tohoto prostředí vyžadují (požadované rozhraní). V některých komponentových modelech rozhraní specifikují také extra-funkční vlastnosti, např. časovou závislost, využití zdrojů, spolehlivost a bezpečnost.

Nasazení, vazby a kompozice - nasazení je proces, který umožňuje integraci komponent do systému (běhového prostředí). Nasazená součást je v systému registrována a připravena poskytovat své služby. Vazba je proces, který vytváří spojení mezi jednotlivými komponentami skrze jejich rozhraní. V CBSE se vazba nazývá složená komponenta, která předpokládá kompozici funkcí vzájemně spojených komponent. Kromě funkčních vlastností se CBSE také zabývá skládáním extra-funkčních vlastností spojených komponent.

3.2.2 Základní koncepce a zásady

Samotný název CBSE předpokládá budování systémů z komponent jako opakovatelně použitelných jednotek a udržování vývoje komponent odděleně od rozvoje

systému. Toto oddělení má významné důsledky pro obchodní strategie (např. trh s komponentami software), technologické důsledky (zavádění nových funkčností za provozu) a právní či sociální důsledky (důvěra, bezpečnost a údržba). Aby CBSE dosáhl svých hlavních cílů, tj. zvýšení efektivity a kvality rozvoje, zkrácení doby uvedení do provozu či prodeje, je založen na následujících zásadách.

Opakovatelnost - celý přístup CBSE je plně využíván pouze tehdy, jsou-li komponenty vyvinuty jen jednou a mají potenciál pro opakované použití v různých aplikacích. Může se jednat o komerční komponenty, ale také součásti s otevřeným kódem. Odůvodněné jsou i principy CBSE pro vytváření architektonických komponent bez záměru o jejich znovupoužití v jiných systémech. Jedná se např. o dekompozici složitého systému na menší jednodušeji udržovatelné a aktualizovatelné součásti.

Nahrazení - systémy složené z komponent si udržují správnou funkci, i když je nějaká jejich část nahrazena jinou. Tento požadavek se shoduje se zásadou nahrazení podle Liskové[2]: Nechť $q(x)$ je vlastnost prokazatelná u objektů x typu T. Potom $q(y)$ platí také pro objekty y typu S, kde S je podtyp T. Tento princip je možný pro funkční vlastnosti, ale nikoliv pro extra-funkční vlastnosti komponenty, protože ty závisí na jiných faktorech, např. systémovém kontextu.

Rozšiřitelnost - v CBSE je rozšiřitelnost zaměřena na podporu rozvoje přidáním nových komponent nebo vyvíjením stávajích, aby se rozšířila funkčnost systému. Typickým řešením pro podporu rozvoje součástí je poskytnout komponentě více rozhraní.

Schopnost být skládán - skládání menších součástí do větších celků je základní zásadou CBSE. Každá technologie založená na komponentách podporuje složení funkčních vlastností (slučování komponent). Málokdy existuje podpora pro složení extra-funkčních vlastností, například složení spolehlivosti komponent nebo doby provádění nebo využití paměti. Sloučení extra-funkčních vlastností zůstává jedním z hlavních úkolů výzkumu CBSE.

[2]Barbara Lisková (nar.1939) - americká informatička a profesorka na Massachusettském technologickém institutu.

3.3 Přínos CBSE

Během několika let vývoje se softwarové komponenty změnily ve své formě a účelu. Zejména v koncepci prvků zdrojového kódu, jako jsou rutiny, funkce a procedury, moduly a objekty, které se transformují na architektonické jednotky a bloky připravené k okamžitému použití. Ty je možné ihned dynamicky zapojit do běžících systémů. CBSE přispěl k pokroku ve vývoji tím, že poskytl komponentám formální specifikace a identifikoval jejich jasnou roli jako opakovatelně použité jednotky.

Souběžně s CBSE jsou komponenty základními prvky jiných přístupů softwarového inženýrství - opětovné použití software, produktové řady software a modelem řízený vývoj. Toto přijetí v různých disciplínách ukazuje, že softwarové komponenty zůstanou na dlouhou dobu klíčovým aspektem principů softwarového inženýrství.

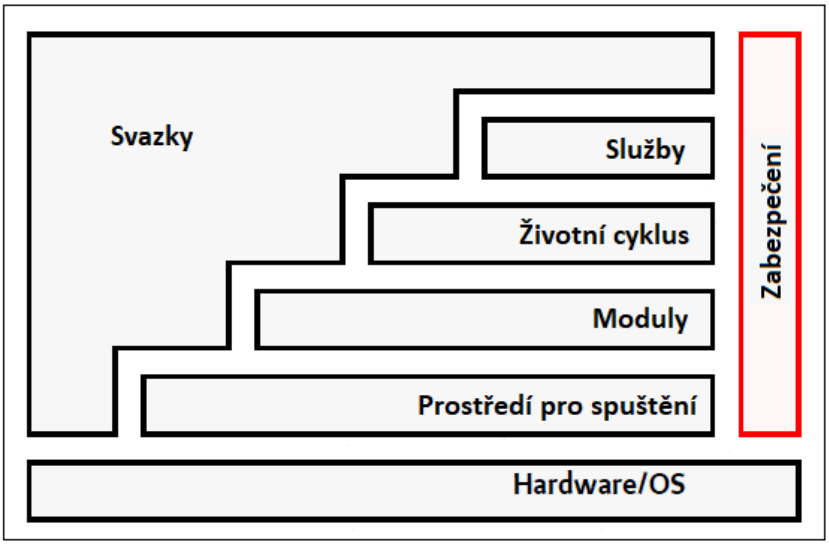
4 Technologie pro softwarové komponenty

Realizaci integrace komponent je možné provést různými způsoby. V současné době jsou nepoužívanější zejména technologie war v Java EE, Microsoft .NET Framework, softwarová knihovna NPM pro platformu node.js, OSGi. S ohledem implementaci úložiště CRCE, která je orientována na platformu OSGi, bude v dalším textu tato technologie podrobněji charakterizována.

Technologie OSGi se skládá ze souboru specifikací, referenční implementace pro každou specifikaci a souboru testů pro každou specifikaci, které společně definují dynamický modulární systém pro programovací jazyk Java. V následujících odstavcích bude stručně uveden základní popis technologie. Důvodem je využití konceptu OSGi pro aplikaci úložiště komponent podporující kontrolu kompatibility a jeho rozšíření o nové funkčnosti.

4.1 Specifikace rámce OSGi

Rámec (framework) OSGi tvoří základní jádro. Poskytuje obecnou, bezpečnou a spravovatelnou strukturu v programovacím jazyku Java pro nasazení dynamicky rozšiřitelných a stahovatelných aplikací známých jako svazky (bundle). OSGi komatibilní systémy mohou svazky dynamicky stahovat, instalovat a odstraňovat (pokud již nejsou potřeba). K tomuto účelu spravuje závislosti mezi svazky a službami. Jednotlivé komponenty spolu komunikují lokálně nebo s okolím díky zmíněným službám. [Hal11], [Osg]

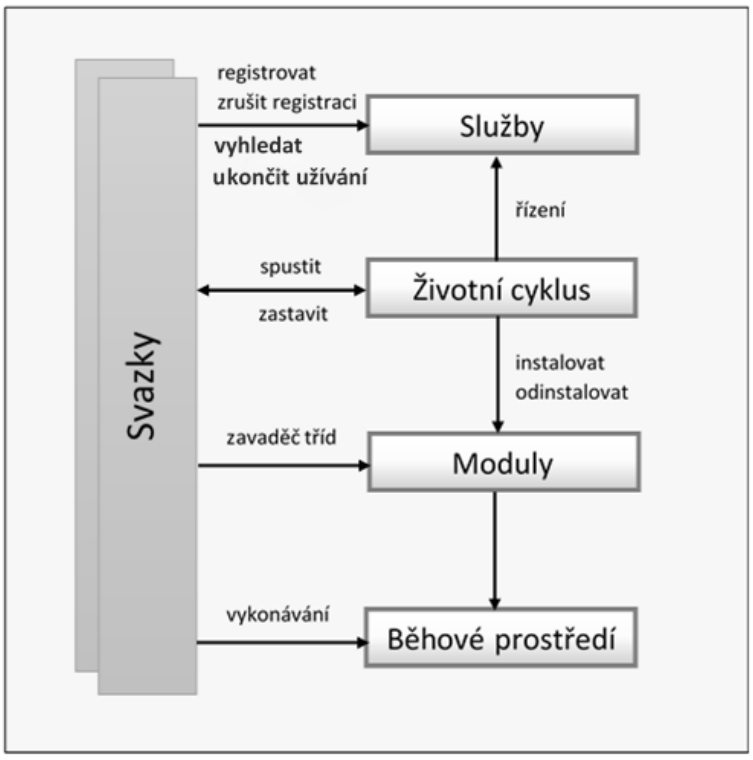


Obrázek 2: Znázornění vrstev OSGi modelu

Funkčnost rámce je rozdělena do následujících vrstev:

- **Svazky** - jsou součásti OSGi, které vytvářejí vývojáři.
- **Služby** - dynamicky spojují svazky tím, že aplikují model *publikování - nalezení - propojení* pro jednoduché objekty POJO.
- **Životní cyklus** - rozhraní API pro instalaci, spuštění, zastavení, aktualizaci a odinstalaci svazků.
- **Moduly** - definují, jak může svazek importovat a exportovat kód.
- **Zabezpečení** - vrstva, která zpracovává bezpečnostní aspekty.
- **Prostředí pro spuštění** - definuje, jaké metody a třídy jsou k dispozici v konkrétní platformě.

Pro přehlednost je uvedeno schéma jednotlivých vrstev obrázku číslo 2. Spolupráce jednotlivých vrstev je zřejmá z obrázku číslo 3.

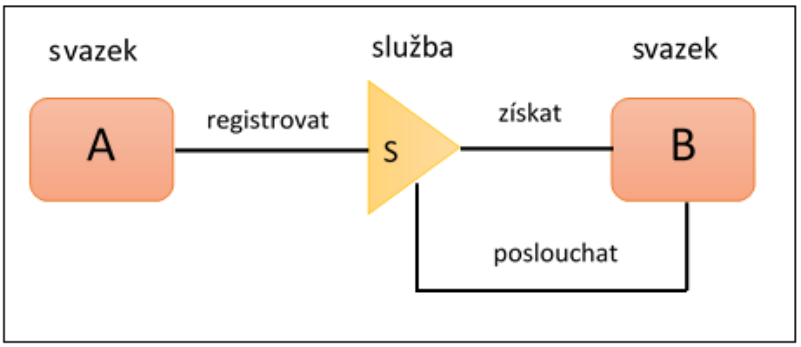


Obrázek 3: Interakce vrstev OSGi modelu

Model služeb znamená velkou výhodu specifikace OSGi proti klasickému pojetí sdílení tříd v programovacím jazyku Java. Při volání tovární metody třídy není

možné jednoduchým způsobem ovlivnit výběr implementace podtříd. Problém je, že mechanismus továrních tříd je konvence používaná ve stovkách míst ve VM, kde každá továrna má své vlastní unikátní API a konfigurační mechanismy. Neexistuje centralizovaný přehled implementací, ke kterým je kód vázán. Implementační kód (v klasickém pojetí) nemá možnost, jak se propagovat, a uživatel nemá možnost seznámit se s implementačním seznamem a vybrat z něj nejvhodnější variantu.

Řešení výše uvedeného problému poskytuje registr služeb OSGi. Svazku lze přiřadit objekt a registrovaj jej do registru služeb s jedním nebo více rozhraními. Jiné svazky mohou uvést do registru všechny objekty, které jsou registrovány pod určitými rozhraními nebo třídou. Balíček, který obsahuje implementaci rozhraní, při svém spuštění vytvoří instatnci třídy a přidá ji do seznamu služeb. Jiný balíček, který potřebuje službu (implementaci třídy), požádá registr o všechny možné rozšíření dané třídy. Dokonce může počkat, až bude daná služba dostupná. Svazek tedy může zaregistrovat službu, může ji získat a může naslouchat, až se služba objeví nebo zmizí. Jakýkoli počet svazků může zaregistrovat stejný typ služby a libovolný počet svazků může získat stejnou službu. Toto je znázorněno na obrázku číslo 4, což je obecně nazýváno makléřským vzorem (broker pattern).



Obrázek 4: Schéma registrace služeb v OSGi

Služby jsou dynamické. Balíček, který je poskytuje, může nabízení služeb v registru ukončit. Ostatní balíčky, které využívaly danou službu, si musí samy zajistit, aby ukončenou službu přestaly využívat. Splnění této vlastnosti je základním předpokladem přidávání a odebírání svazků za běhu.

4.2 Přehled technologií a jejich vlastností, metadata, verzování

V předchozí kapitole byla ve stručné formě nastíněna specifikace OSGi. V následujících podkapitolách budou uvedeny příklady implementace, zejména bude uveden způsob získávání artefaktů a správa množin komponent.

4.2.1 Apache Felix

Jedná se o komunitní úsilí o implementaci OSGi specifikace šestého vydání pod Apache licencí. [Fel] V základní verzi se jedná o kontainer komponent, který umožňuje řešit závislosti mezi jejich požadavky (requirements) a schopnostmi (capabilities). Pro rozšíření možností je projekt Felix organizován do dílčích projektů, kde je každý dílčí projekt zaměřen na konkrétní specifikaci OSGi nebo technologii související s OSGi. S ohledem na cíl práce je důležité zmínit zejména podprojekty **Gogo**, **File Install**, **OSGi Bundle Repository**, **Maven Bundle Plugin** a **Web Console**.

Projekt **Gogo** je pokročilý shell pro interakci s OSGi rámcem. Obsahuje tři svazky, které zabezpečují funkci zpracování příkazů jádra rámce a poskytují jednoduché uživatelské rozhraní pro komunikaci s příkazovým procesorem. Stažení a instalaci svazku (dostupného na url adrese) lze provést příkazem `install [url odkaz]`.

File Instal je agent, který sleduje v konfiguraci nastavený adresář, z nějž instaluje a spouští svazky. Instalace svazků proběhne, když je soubor do adresáře umístěn poprvé. Aktualizace svazku bude provedena automaticky, když bude soubor z adresáře odstraněn a následně do něj znovu nakopírován.

Cílem projektu **OSGi Bundle Repository** je zjednodušení nasazení a používání modulů, zvýšení viditelnosti dostupných svazků a zajištění přístupu k spustitelnému balíčku a jeho zdrojovému kódu. Implementace OBR s podporou kompatibility uložených komponent je projekt CRCE. Podrobnosti budou uvedeny v kapitole číslo 6.

Projekt **Maven Bundle Plugin** je založen na nástroji BND[3]. V následujícím zdrojovém kódu je na obrázku číslo 5 uvedena ukázka použití pluginu. Instrukcemi uvedenými v tagu `<instructions>` je možné specifikovat zkopírování balíčků do svazku, nastavení jejich exportu, požadavky na import, uvedení spuštěcí třídy při aktivaci svazku a uvedení vložených závislostí. Předpokládá se užití pluginu

[3]BND - nástroj pro vytváření popisných metadat pro OSGi svazky (soubor manifest) při sestavování projektu do spustitelného souboru jar `https://bnd.bndtools.org/`

při sestavování projektu nových funkčností rozšíření úložiště komponent CRCE.

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <manifestLocation>META-INF</manifestLocation>
        <instructions>
            <Bundle-Activator>${bundle.namespace}.internal.Activator</Bundle-Activator>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
            <Private-Package>cz.zcu.kiv.crce.crce_external_repository.internal.*</Private-Package>
            <Export-Package>
                cz.zcu.kiv.crce.crce_external_repository.api,
                org.apache.maven.index.*,
                org.apache.maven.wagon.*,
                javax.inject.*
            </Export-Package>
            <DynamicImport-Package>*</DynamicImport-Package>
            <Import-Package>
                !${bundle.namespace}.internal.*
            </Import-Package>
            <Embed-Dependency>!sisu-guice,*;scope=compile|runtime;inline=false</Embed-Dependency>
            <Embed-Directory>lib</Embed-Directory>
            <Embed-StripGroup>true</Embed-StripGroup>
        </instructions>
    </configuration>
</plugin>
```

Obrázek 5: Ukázka užití projektu Maven Bundle Plugin

Projekt **Web Console** představuje uživatelsky přívětivé grafické rozhraní (webovou aplikaci) pro konfiguraci a správu rámce Felix. Felix ve výchozí konfiguraci Web Console neobsahuje, a proto je nutné příslušný modul doinstalovat včetně příslušných svazků, které pro svou funkci požaduje (např. *javax.servlet*, *org.apache.commons.fileupload* atd.). Identicky, jako shell Gogo, umožňuje správu životního cyklu modulů nasazených v běhovém prostředí včetně jejich stažení z url odkazu, instalaci a spuštění. Další užitečnou vlastností je zobrazení registrovaných služeb a v seznamu instalovaých svazků je pro každý modul k dispozici příslušný seznam jeho požadavků a schopností.

4.2.2 Apache Karaf

Apache Karaf Runtime je běhové prostředí (komponentový kontejner) pro modulární aplikace. Základem je rámec OSGi Felix nebo Eclipse Equinox. Distribuce Karaf obsahuje řadu dalších integrovaných rozšíření. [Kar]

- **Rychlé spuštění**: přetažením souboru do nastaveného deploy adresáře, aplikace Apache Karaf rozpozná typ souboru a pokusí se ho nasadit.

- **Konzole**: kompletní příkazová konzole typu Unix, kde je možné plně spravovat kontejner.
- **Dynamická konfigurace**: sada příkazů zaměřených na správu vlastní konfigurace. Všechny konfigurační soubory jsou v *etc* složce centralizovány. Jakákoli změna v konfiguračním souboru je zaznamenána a znovu načtena.
- **Protokolovací systém**: podpora všech oblíbených protokolovacích rámců (slf4j, log4j, atd.). Bez ohledu na druh protokolování Karaf centralizuje konfiguraci v jednom souboru.
- **Přiřazení**: správa adres URL, odkud je možné nainstalovat aplikace (repozitář Maven, HTTP, soubor atd.). Poskytuje také koncept „Karaf Features“. Podrobnosti o tomto konceptu budou uvedeny v dalších odstavcích.
- **Správa**: indikátory řízení a operací prostřednictvím JMX.
- **Vzdálené řízení**: obsahuje server SSHd, který umožnuje vzdáleně používat konzoli. Řídící vrstva je také vzdáleně dostupná.
- **Bezpečnost**: bezpečnostní rámec (založený na JAAS) a poskytuje mechanismus RBAC pro přístup ke konzolím a JMX.
- **Instance**: více instancí Karafu lze spravovat přímo z hlavní instance (root).
- **Rámec OSGi**: Karaf není pevně svázán s jedním rámcem OSGi. Ve výchozím nastavení Apache Karaf běží s Apache Felix Framework, ale je možné snadno přepnout na Equinox (stačí změnit jednu vlastnost v konfiguračním souboru).

Další výhodou Karafu je například podpora Apache Camel a CXF. Apache Camel je otevřený rámec založený na zprávách s mechanismem jejich směrování na základě pravidel. Pro konfiguraci pravidel používá deklarativní jazyk (na bázi XML). Apache CXF otevřený rámec pro usnadnění vývoje webových služeb založených na celé řadě protokolů (*SOAP*, *XML/HTTP*, *RESTful HTTP* nebo *CORBA*). [Ari]

4.2.3 Karaf Features

Svazek OSGi může záviset na dalších svazcích a ty poté na dalších tak, aby uspokojily své vlastní závislosti. Způsob nasazení aplikace po jednotlivých svazcích není tohoto důvodu jednoduchý. Další problém je v nastavení konfigurace. Apache Karaf poskytuje pro nasazování aplikací užitečnou podporu - **features**. [Karc]

Zavedení aplikace znamená instalace všech modulů, načtení konfigurace a tranzitivních závislostí. Feature je charakterizována následujícími atributy:

- jméno,
- verze,
- popis (volitelně),
- seznam svazků,
- konfigurace nebo konfigurační soubory (volitelně),
- závislosti feature (volitelně).

Feature má kompletní životní cyklus: instalace, spuštění, zastavení, aktualizace, odinstalování. Při instalaci feature aplikace Apache Karaf nainstaluje všechny prostředky popsané ve funkci. To znamená, že automaticky rozpozná a nainstaluje všechny svazky, konfigurace a závislosti obsažené v popisu feature. Služba *Resolver* [Osg] řeší požadavky a nainstaluje svazky (pokud nejsou již nasazené), které tyto požadavky splňují. Instalaci features lze provést zadáním příkazu:

```
feature:install [name].
```

Seznam dostupných feature je možné vypsat příkazem:

```
feature:list.
```

Příklad je uveden na obrázku číslo 6.

Obrázek 6: Ukázka příkazu list pro skupinu

Features jsou charakterizovány funkčním XML deskriptorem (popisovač). Popisovač feature je pojmenovaný stejně jako repositář features. Před instalací feature

musí být repositář zaregistrován `feature:repo-add [url]`. Registrované repositáře lze vypsat příkazem: `feature:repo-list`. Ukázka XML popisovače feature je uvedena na obrázku číslo 7.

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.3.0">
  <feature name="feature1" version="1.0.0">
    <bundle>mvn:mysql/mysql-connector-java/8.0.15</bundle>
    <bundle>mvn:org.apache.logging.log4j/log4j-core/2.11.2</bundle>
  </feature>
  <feature name="feature2" version="1.1.0">
    <feature>feature1</feature>
    <bundle>mvn:org.apache.camel/camel-core/2.23.1</bundle>
  </feature>
</features>
```

Obrázek 7: Ukázka syntaxe popisovače feature

Uvedený popisovač obsahuje dvě feature. V tagu s označením je uvedena verze. Feature s názvem *feature1* a verzí 1.0.0 obsahuje dva svazky (případně dostupné z Maven úložiště), feature s názvem *feature2* obsahuje kromě jednoho definovaného svazku také *feature1*. Při instalaci feature je možné uvést požadovanou verzi. Pokud konkrétní verze není uvedena, Apache Karaf vybere pro instalaci nejvyšší verzi dostupnou v repozitáři.

Další důležitou vlastností feature je možnost definovat pořadí spouštění (*start-level*). Tato vlastnost je zásadní pro kontrolu závislosti. Pokud na svazek *projekt-dao* mají závislost další svazky, např. *projekt-service*, je podstatné, aby svazek *projekt-dao* byl instalován, registrován a spuštěn před svazkem *projekt-service*. Toto lze zajistit doplněním parametru v tagu `<bundle>` v descriptoru feature. Na obrázku číslo 8 je uveden zápis popisovače feature toho případu.

```
<feature name = "moje-feature" verze = "1.0.0">
    <bundle start-level = "80"> mvn:cz.zcu.kiv.muj-projekt/projekt-dao</bundle>
    <bundle start-level = "85"> mvn:cz.zcu.kic.muj-projekt/projekt-service</bundle>
</feature>
```

Obrázek 8: Příklad pořadí instalace svazků feature

Skutečnost, že je ve feature možné definovat závislost na jiné feature požadované verze, již byla v předchozím textu uvedena. V případě, že není přímá závislost na konkrétní verzi, ale na rozsahu verzí, je možné toto definovat způsobem, který je uveden na obrázku číslo 9. *feature1* bude nainstalována v nejvyšší verzi dostupné v repositáři z uvedeného rozsahu.

```
<feature name = "feature2">
  <feature version = "[2.5.6,4)"> feature1 </feature>
  ...
</ feature>
```

Obrázek 9: Závislost *feature2* na daném rozsahu verzí *feature1*

Feature Apache Karaf dávají možnost uvést v deskriptoru konfigurační parametry, které budou viditelné v aplikaci. Zápis se provádí systémem klíč/hodnota. Pro odlišení jednotlivých konfigurací se používá označení PID. Výhodou užití konfiguračních parametrů je, že jsou uvedeny přímo v popisu aplikace a není nutné spravovat příslušný konfigurační soubor. Na obrázku číslo 10 je zobrazen příklad zápisu konfiguračního parametru. Jak je zřejmé z přechozího textu, feature představují silný nástroj pro nasazování, údržbu a správu aplikací složených z komponent v OSGi.

```
<config name = "cz.zcu.kiv.crce.attribute">
  datasourceUrl = jdbc:mysql://localhost:3306/crce?characterEncoding=utf8
</config>
```

Obrázek 10: Příklad zápisu konfiguračního parametu

4.2.4 Red Hat Fuse

V předchozí kapitole byl zmíněna implementace specifikace OSGi Apache Karaf zejména s důrazem na způsoby správy a nasazování aplikací sestavených ze samostatných komponent a využití features. V této podkapitole bude uvedena implementace specifikace OSGi, která obsahuje další rozšíření možností nasazování a spravování aplikací. [Fus]

Red Hat Fuse je otevřená integrační platforma s podporou Apache Camel. Poskytuje infrastrukturu a nástroje pro integraci služeb, mikroslužeb a aplikací složených z komponent. Platformu Fuse představila společnost LogicBlaze jako SOA Open Source. Skupina stála za řešením ServiceMix[4] a ActiveMQ[5] (nyní pod Apache Software Foundation). [Log] Zásadní podíl na vývoji Fuse měla společnost jBoss, která verze 6.0.0 až 6.3.0 distribuovala ještě pod názvy jBoss Fuse. Akvizicí firmy jBoss společností RedHat v roce 2006 došlo nejen k přejmenování jBoss Fuse na jBoss Red Hat Fuse, respektive později pouze na Red Hat Fuse, ale zejména od verze 7 k zásadnímu přepracování platformy. Od verze 7 došlo k ukončení podpory možnosti clusterového uspořádání s podporou Fabric8[6] a nyní je kladen důraz na hybridní cloudovou platformu založenou na technologii OpenShift[7]. V roce 2018 byl dokončen odkup společnosti RedHat firmou IBM. Bude tedy obtížné předpokládat další vývoj Red Hat Fuse.

V této práci bude brán zřetel na variantu Red Hat (jBoss) Fuse ve verzi 6.0.0, respektive 6.3.0. Je to zejména z důvodu praktických zkušeností autora práce s touto platformou v provozním prostředí stěžejního systému instituce a také, jak bylo uvedeno v úvodu podkapitoly, na možnosti správy a nasazení aplikací tvořených komponentami.

Fabric8 umožňuje správu jednotlivých uzlů Fuse v clusterovém uspořádání. Využití je pro klíčové podnikové aplikace a systémy, kde je vyžadována vysoká dostupnost. Jednotlivé nódy běží nezávisle, komunikují po síti, mohou být nasazeny na nezávislých operačních systémech a doporučuje se i na oddělených strojích. Administrovat systém je možné při přihlášení na libovolný uzel - systém nemá centrálního arbitra. Replikace změn je záležitostí systému Fabric8.

Z hlediska správy aplikací implementuje Fuse profily a verze a řeší jejich nasazování na kontejnery. Předpokládá se, že každý uzel obsahuje jeden kontejner, ale tato skutečnost nemusí být pravidlem. Profil je pomyslná nadstavba nad feature, jehož popis funkčnosti byl uveden v předchozí kapitole 4.2.2. Záznam profil obsahuje: seznam features a samostatných jednotek komponentové aplikace (moduly) a parametry konfigurace. S ohledem na přehlednost záznamů při velkém možství nasazovaných aplikací je vhodné sdružovat související moduly do samostatných profilů. Pro jejich uspořádání se poté používá stromová struktura (rodič - potomek). Tyto profily jsou přiřazeny kontejneru. Pokud například všechny nódy ob-

[4]ServiceMix - `http://servicemix.apache.org`
[5]ActiveMQ - `http://activemq.apache.org/`
[6]Fabic8 - `http://fabric8.io`
[7]OpenShift - `https://www.openshift.com/`

sahují stejné profily, budou shodně aplikace distribuovány na všechny uzly Fuse.

Fuse podporuje možnost verzování aplikací. Každá verze vytvořená ve Fuse udává aktuální konfiguraci profilů. Povýšením verze provede Fuse kopii aktuálních profilů a je možné provést jejich úpravu. Úprava spočívá například ve změně (povýšení) verzí features nebo jednotlivých modulů, které jsou v profilu zaznamenány. Na kontejnery se tedy nenasazují profily jako takové, ale jejich konkrétní verze. Výhodou takovéhoto uspořádání je bezvýpadkový přechod na novou verzi aplikací.

Pro instalaci svazků je k dispozici výchozí lokální Maven úložiště uživatele, pod kterým je spuštěn proces Fuse. Pokud daný artefakt není v lokálním úložišti dostupný, hledá proces instalace v dalších repositářích, které jsou uvedeny v konfiguračním souboru *org.ops4j.pax.url.mvn.cfg* umístěným ve složce *etc*. Zde je možné definovat vlastní url úložiště komponent. Konfigurační soubor obsahuje záznam adresy zdroje artefaktů a features, poskytované RedHat, pro vlastní rozšíření funkčností Fuse. Před instalací daného artefaktu je nejprve stažen do lokálního Maven úložiště a následně z něj instalován.

Níže jsou uvedeny konkrétní příkazy pro administraci profilů a verzí a nasazování na kontejnery. Cílem není kompletní výčet všech příkazů. K tomuto účelu je k dispozici uživatelská dokumentace. [Fab] Uvedení ukázek si klade za cíl seznámení s příkladem již existující implementace správy komponentových aplikací v prostředí OSGi.

Vytvoření profilu ve Fuse se provede příkazem:

```
fabric:profile-create
```

s volitelnými parametry verze *–version* a nastavení rodičovského profilu: *–parents*. V případě vynechání parametru *–version* se profil založí do aktuální (default) verze. Na uzlu, kerý je zapojen do clusteru Fabric8, je možné úvést příkaz bez klíčového slova *fabric*. Skupina příkazů *fabric* je již ve výchozím stavu nastavena. Úpravy v existujícím profilu lze provést příkazem:

```
fabric:profile-edit [name],
```

kde atribut *name* je jméno profilu nastavené verze. V editoru, který je součástí administračního rozhraní Fuse, je možné provést doplnění či úpravu konfigurace profilu. Při změně konfiguračního parametru není nutné editovat celý profil. Parametr s identifikačním jménem *count* je obsažen v rozlišovací skupině

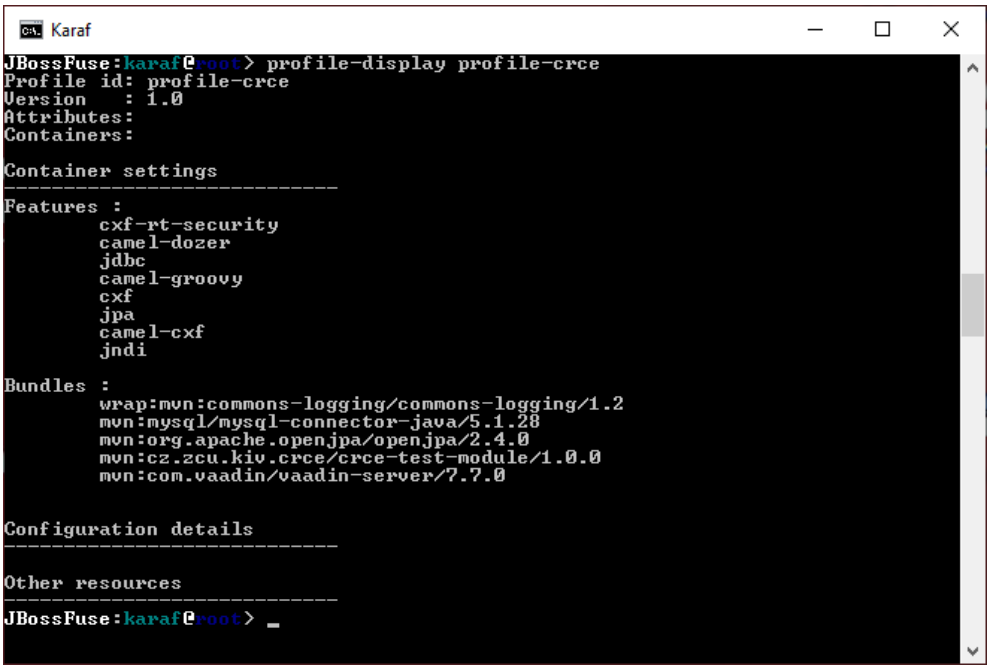
cz.kiv.crce.quartz v profilu *profile-crce*. Příkaz *profile-edit* se syntaxí uvedenou níže nastaví *count* na hodnotu 10:

```
profile-edit
--pid cz.kiv.crce.quartz/count = 10 profile-crce.
```

Výpis obsahu profilu je proveden po zadání příkazu:

```
profile-display [name].
```

Na obrázku číslo 11 je ukázka výpisu obsahu profilu *profile-crce*.



Obrázek 11: Výpis obsahu profilu *profile-crce*

Pro smazání profilu je určen příkaz:

```
profile-delete [name].
```

Profil s sebou nese informaci o své verzi. Tato informace je také uvedena v hlavičce profilu. Profily jsou nasazené na kontejnery ve verzi, na kterou je nastaven kontejner. Není striktním pravidlem, že všechny kontejnery musí být ve shodné verzi, ale s ohledem na clusterový režim aplikací se tento stav předpokládá.

Nasazení profilu na kontejner je provedeno při zadání příkazu:

```
container-add-profile [name-container] [name-profile].
```

Systém profilů společně s verzováním tvoří užitečný nástroj pro správu aplikací. Následující text bude věnován správě verzí. Každý kontejner má nastavenou verzi profilů. Vytvoření nové verze se provede příkazem:

```
version-create [number]
```

s volitelným parametrem *–default*, který způsobí, že se nová verze nastaví na výchozí (pro editaci profilů). Povinný parametr *number* udává číslo nové verze. Používá se systém číslování verzí: *verze.mikroverze*. Při vytvoření nové verze se do této zkopíruje obsah profilů z poslední výchozí verze. Po editaci profilů v nové verzi (povýšení svazků, změna konfiguračních parametrů) je možné přistoupit k nasazení této nové verze na kontajnery. Příkazem:

```
version-list
```

je možné zobrazit seznam dostupných verzí profilů, kde textem *true* je označena aktuální verze, kterou je možné měnit. Uvedení čísla u verze je značen počet kontejnerů, kde je daná verze nasazená. Nasazení konkrétní verze na kontejner, respektive kontejnery, se provede příkazem:

```
container-upgrade [number_of_version].
```

Výše uvedený příkaz povýší verzi pouze na aktuálním kontejneru (na přihlášeném uzlu Fuse). Při doplnění nepovinného parametru *–all* dojde k povýšení na všech uzlech v clusteru. V situaci, kdy z jakéhokoliv důvodu vykazuje nová verze po nasazení problémy, je možnost operativně, rychle a bez jakéhokoliv výpadku funkčnosti vrátit původní verzi příkazem:

```
container-rollback [number_of_version].
```

Uvedením nepovinného parametru *–all* způsobí ponížení verze na všech kontejnerech v clusteru.

OSGi specifikace RedHat jBoss Fuse, zejména nasazování množin komponent jako celistvých aplikací a správa jejich verzí, je inspirací pro implementaci vybraných funkčností v rámci projektu úložiště komponent s podporou kompatibility CRCE.

4.3 Apache Maven

Prioritním účelem nástroje Apache Maven je pomoc při sestavování složitých, na komponentách založených projektů, reportování a vytváření dokumentace. I v případě jednoduché aplikace v programovacím jazyku Java se nelze plně vyhnout závislosti (dependency) na jiných projektech. Tato situace je logická. Je snahou, pokud je to možné, využívat již hotové a ověřené funkčnosti. Tyto závislosti lze uspokojit zahrnutím externích souborů (knihoven) v distribuci projektu. Tato distribuce pak může obsahovat několik desítek až stovek komponent potřebných pro chod aplikace.

Sestavovací nástroj Maven toto řeší mnohem efektivněji díky centrálnímu úložišti artefaktů[8]. V případě nedostupnosti hledaného artefaktu v centrálním úložišti je možné definovat a použít vlastní Maven úložiště. Projekt aplikace vytvořený v programovacím jazyku Java před sestavením nástrojem Maven obsahuje pouze vlastní balíky, třídy, rozhraní atd. a konfigurační soubor *pom.xml*. Je zřejmé, že distribuce takovéhoto projektu je co do velikosti mnohonásobně menší než kompletně sestavený projekt.

V konfiguračním souboru *pom.xml* v sekci označené tagem `<dependencies>` resp. `</dependencies>` jsou uvedeny závislosti na externích aretfaktech. Na obrázku číslo 12 je pro názornost uvedena ukázka zápisu. Každý artefakt v Maven úložišti je jednoznačně definován kombinací následujících popisných identifikátorů: skupina (*groupId*), název artefaktu (*artefactId*), verze (*version*) a přípony (*package*).

Aplikací, které implementují vlastní Maven repository, je několik. Jako příklad je možno uvést Nexus Repository[9] od společnosti Sonatype[10] [Nex]. Pro popis sestavení projektu pomocí nástroje Maven se používá konfigurační soubor POM (*pom.xml*). Pro proces sestavení projektu je nutné mít instalovaný nástroj Maven[11]. Proces Maven sestavení projektu podporují také vývojové nástroje (IDE) pro jazyk Java. Apache Maven není jediný zástupce sestavovacích nástrojů projektů v programovacím jazyku Java. Alternativou mohou být nástroje: Apache Ant, Apache Ivy, Gradle Build Tool a další.

[8] Maven Repository - `https://repo.maven.apache.org/maven2/`
[9] Nexus - `https://www.sonatype.com/nexus-repository-sonatype`
[10] Sonatype - `https://www.sonatype.com/`
[11] Stažení Maven 3.6.0 - `https://maven.apache.org/download.cgi`

```
<dependencies>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongo-java-driver</artifactId>
        <version>3.9.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.9.0</version>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>
</dependencies>
```

Obrázek 12: Zápis závislostí v *pom.xml* souboru

Až při samotném sestavení projektu pomocí Maven nástroje dojde ke stažení závislých artefaktů z centrálního (nebo vlastního) Maven úložiště. Toto je ovšem jen jedna z předností nástroje Maven. Dále je možné definovat název a verzi projektu, hiearchii projektu, vlastnosti (properties), například zdrojový kompilátor, životní fáze cyklu projektu, spouštění zásuvných modulů pro rozšíření funkčnosti (pluginů). Priroritním zaměřením s ohledem na zaměření práce je samotné Maven úložiště artefaktů a nástroje pro jejich identifikaci a získávání.

Základní přístup do centrálního úložiště artefaktů je url odkazem. Pro artefakt *java-mongo-driver* uvedený na obrázku číslo 12 má url odkaz tvar:

`https://repo.maven.apache.org/maven2/org/mongodb/mongo-java-driver/3.9.1/mongo-java-driver-3.9.1.jar`.

Přístup k Maven úložišti artefaktů poskytuje sofistikovanějším způsobem knihovna Aether. Podrobnější údaje budou uvedeny v kapitole 6.1.

5 Kompatibilita komponent

V předchozích kapitolách byl přijat předpoklad, že jednotlivé části, které tvoří celistvou aplikaci, jsou vzájemně slučitelné (kompatibilní). Rozlišují se funkční (struktury rozhraní) a mimofunkční (doba odezvy, výpočetní náročnost) požadavky na kompatibilitu. Jak bylo vysvětleno na příkladu features u implementace specifikace OSGi Apache Karaf nebo u verzí profilů RedHat jBoss Fuse (podkapitola 4.2.3 a 4.2.4) softwarový architekt definoval popis jejího sestavení a příslušné kompatibilní svazky (moduly). Za ověření zavislostí a případně zajištění dostupnosti požadovaných svazků je odpovědná služba *resolver* příslušné implementace specifikace OSGi [Osg]. Je zřejmé, že při velkém počtu komponent tvořící aplikaci, je velmi obtížné ověřovat jejich vzájemnou kompatibilitu. Situace je složitější při velkém počtu úprav a při změně verzí jednotlivých součástí.

5.1 Ověřování kompatibility

Ověřovací proces je časově a výpočetně náročný. [BJ15] Na provozních systémech s velkým výkonem je situace ještě poměrně únosná. Při distribuci nové verze aplikace je možné clusterové řešení systému (viz Fabric RedHat Fuse popsaný v kapitole 4.2.4). Nasazování a ověřování vazeb probíhá postupně po jednotlivých uzlech, bez ovlinění ostatních a bez výpadku funkčnosti. Pokud ověření vazeb selže na kterémkoliv uzlu, není provedeno povýšení verze a původní zůstává v provozu. Další příkladem je distribuce změn využívaná při aktualizaci systémů. Funkčnosti se stále vyvíjejí, rozšiřují a je nutné tyto změny odeslat koncovým aplikacím. Aplikace posílá informaci o své verzi, respektive verzi komponent, ze které se skládá, případně ještě informace o svém běhovém prostředí. Aktualizační server následně odešle na koncové zařízení potřebné změny. Tento způsob má výhodu, že ověření kompatibility vazeb a funkčností nezatěžuje koncové zařízení. V případě častečných změn také není zatěžována komunikační síť, jako v případě komplexních distribucí.

V současné době se zvyšuje výkon malých zařízení (chytré telefony, hodinky, IoT - Internet věcí). Z tohoto důvodu i na těchto zařízeních jsou s výhodou používány komponentové aplikace. Ovšem jejich výpočetní výkon je zapotřebí pro funkčnost aplikace a nikoliv pro kontrolu kompatibility jejích modulů. Je zapotřebí, aby vyřešení vzájemné kompatibility zajistilo úložiště komponent.

Skládání aplikací z již existujících komponent nebo služeb předpokládá následující aspekty:

- úložiště nebo registr obsahující artefakty nebo odkazy na jiná úložiště artefaktů,
- popis jejich vlastností na různých úrovních kontraktů (popis funcionality, specifikace chování, extrafunkční vlastnosti, sémantika),
- na základě výběru vhodných kriterií poskytnutí komponent v rámci kompozice.

Úložiště, podporující vzájemnou kompatibilitu obsažených komponent, spravuje artefakty (distribuční balíčky). Umožňuje jejich vkládání, odebírání, aktualizaci a samozřejmě je zpřístupňuje navenek. Dále poskytuje popisná data (metadata) k daným artefaktům. Může se jednat o základní syntaktickou analýzu vzájemných vazeb (rozhraní) komponent, ale v případě pokročilých modelů, které poskytují několikaúrovňové vazby rozhraní, je nezbytné použít sémantické či iterační specifikace. V případě mimofunkčních charakteristik je nutné provést výkonnostní nebo bezpečnostní testy. Metadata obsahují prohlášení o kompatibilitě pro každou verzi komponenty a popis vazeb, které jsou dány požadovanými a poskytovanými vlastnostmi. Dále obsahují podrobné údaje vysvětlující označenou kompatibilitu (výsledky testů rekurzivního výběru vhodného typu, protokoly z testů extra-funkčních vlastností).

Ověření kompatibility může být provedeno dvěma způsoby. Koncovým zařízením na základě úložištěm poskytnutých metadat nebo klient požádá úložiště o poskytnutí poslední verze hledané komponety, která je s danou kompozicí kompatibilní.

5.2 Úložiště CRCE

Uvedené poznatky byly použity při návrhu konkrétní implementace úložiště podporující kontrolu kompatibility (CRCE). Koncepce CRCE byla navržena a realizována v rámci diplomové práce [Kuc] a v dalších pracích a projektech výzkumné skupiny ReliSA na katedře informatiky a výpočetní techniky je dále rozšiřována jeho funkcionalita. Aplikace CRCE je vytvořena na principu modulárního systému. Její komponentová architektura umožňuje aktualizaci a rozšiřitelnost. Základními částmi jsou:

- **Component Store** - správa úložiště artefaktů,
- **Metadata Store** - popisná data uložených artefaktů,

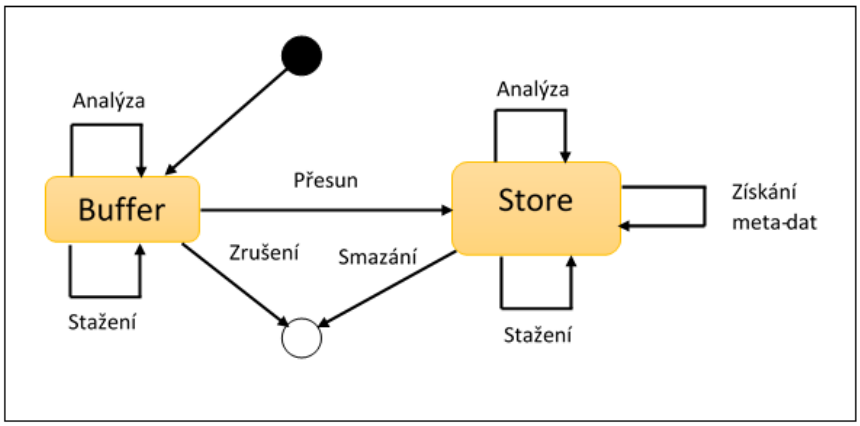
- **Result Store** - výsledky testů kompatibility.

Zmíněné prvky mají k dispozici přístupové programátorské rozhraní API.

Pro podporu kompatibility vyžívá CRCE následující princip: po načtení nové verze artefaktu ověřovací proces provede kontrolu součinosti se všemi komponentami, ke kterým může být potenciálně připojen. [BJ15] Sestava komponent (model) a specifikace vazeb mezi jednotlivými částmi prochází sadou testů, kterou zajišťuje architektura pluginů. Jsou používány vhodné ověřovací metody k vyhodnocení příslušné části obsahu komponenty. Následně dochází k aktualizaci souvisejících částí metadat. Indexování a ukládání artefaktů je integrováno do úložiště pomocí API modulů *Metadata* a *Results*

Výsledky indexování komponent a ověření kompatibility jsou uloženy jako metadatové položky ověřených prvků, které mohou být jednotlivou funkcí na rozhraní, jejich množin nebo i celé komponenty. Tato data (generovaná pluginy CRCE) jsou klíčovým prvkem systému, který umožňuje klientům úložiště CRCE ověřit konzistenci komponetové sestavy.

Vytvořeny jsou dvě vzájemně oddělené části úložiště: dočasné *Buffer* a trvalé *Store*. Správa uložených artefaktu se řídí životním cyklem, jak je zřejmé z obrázku číslo 13.



Obrázek 13: Životní cyklus artefaktu v úložišti CRCE

V rámci oborového projektu v předmětu KIV/OPSWI byl autorem této práce modernizován a nově vyvinut modul grafického uživatelského rozhraní s využitím

frameworku Vaadin[12]. Důvodem změny byla zejména obtížná údržba a rozšiřování funkčností stávající webové aplikace založené na technologii servletů a JSP stránek. Další důvodem byla nekompatibilita stávajícího modulu *crce-webui* s běhovým prostředím JVM verze 8 a výše.

Nová aplikace grafického uživatelského rozhraní umožňuje správu životního cyklu artefaktu v úložišti CRCE. Její struktura je v rámci této práce připravena na rozšíření o nové funkce (možnost získávání artefaktů z externích zdrojů a správu množin artefaktů). Aplikace úložiště CRCE umožňuje získávat artefakty dvěma způsoby: z lokálního souborového systému a z přímého url odkazu. Zaměřením předkládané práce je analyzovat další možnosti získávání objektů z externích úložišť, navrhnout a implementovat rozšíření. Též se předpokládá aktualizace modulu grafického uživatelského rohraní (vytvořeného v rámci projektu KIV/OPSWI) o podporu nových funkčností úložiště CRCE.

Podpora základních funkčností, která musela být zachována i v novém modulu grafického uživatelského prostředí, je exaktně definována životním cyklem artefaktu v úložišti, viz obrázek číslo 12. Jedná se o tyto funkčnosti:

- získání artefaktu ze souborového systému nebo url odkazu,
- uložení artefaktu do dočasného úložiště (Buffer),
- zobrazení obsahu dočasného úložiště (Buffer),
- zobrazení metadat artefaktů uložených v dočasném úložišti (Buffer),
- odebrání artefakt uloženého v dočasném úložišti (Buffer),
- přesun artefaktů z dočasného úložiště (Buffer) do trvalého úložiště (Strore),
- zobrazení metadat artefaktů uložených v trvalém úložišti (Store),
- odebrání artefaktu uloženého v trvalém úložišti (Store).

I když jsou si funkčnosti dočasného úložiště (Buffer) a trvalého úložiště artefaktů (Store) podobné, zásadním rozdílem je, že dočasné úložiště je vázano na vytvořené spojení (seesion) přihlášeného uživatele. Jinými slovy: každý přihlášený uživatel má vlastní instanci dočasného úložiště a po odhlášení uživatele je tato instance odstraněna. Trvalé úložiště je společné pro všechny uživatele CRCE.

Důležité API úložiště CRCE, které byly využity v novém modulu uživatelského

[12]Vaadin Framework - `https://vaadin.com/`

grafického prostředí *crce-webui -v2* (projekt OPSWI) a jejichž využití je předpokládáno v dalším rozšíření funkčnosti v rámci této práce jsou dostupné v modulech:

- *crce-repository-api*,
- *crce-metadata-api*,
- *crce-metadata-service-api*.

Activator, zaváděcí třída při inicializaci v prostředí OSGi, modulu *crce-webui-v2* byla s částečnými úpravami inspirována původní verzí *crce-webui*. Jedná se zejména o vytvoření instancí tříd *Buffer*, *Store*, *MetadataService*.

Klíčové metody pro přístup k dočasnému i trvalému úložišti a popisných metadat uložených artefaktů (instance třídy *Resource*) pro modul *crce-webui-v2* jsou implementovány ve třídě *ResourceService* v balíku *repository.services*.

Úložiště CRCE pro fyzické ukládání artefaktů využívá souborový systém a pro uchovávání jejich metadat databázi MongoDB[13]. Každý uložený artefakt v CRCE, v dočasném úložisti (Buffer) nebo trvalém úložišti (Store), je identifikován záznamem v kolekci *resources*. Jednoznačným identifikátorem je atribut *id*. Dalšími atributy záznamu *resource* jsou *id* úložiště (Buffer nebo Store), *uri* adresa fyzického uložení artefaktu a další popisná data. Jedná se o pole typů *Object* s vlastními atributy: name, type, value. Dočasné úložiště (Buffer) má pro každého přihlášeného uživatele vlastní url odkaz pro fyzicky uložené artefakty odlišený identifikátorem spojení. Kolekce *capabilities* a *requirements* obsahují záznamy požadavků a schopností artefaktů, respektive příslušných záznamů *resource*.

V novém modulu grafického uživatelského prostředí *crce-webui-v2* byla předpokládána podpora pro výsledky testů kompatibility. K tomuto účelu by mohla sloužit implementace rozhraní *crce-compatibility-api* v komponentě *crce-compatibility-impl* v projektu *crce-jacc*. Jedná se o metody ve třídě *CompatibilityServiceImpl* v balíku *cz.zcu.kiv.crce.compatibility.internal.service*. Například metoda *findHighestUpgrade()* přebírá jako svůj formální parametr objekt *Resource* a najde nejvyšší kompatibilní verzi stejného názvu objektu *Resource*, který je k dispozici pro upgrade (má vyšší hodnotu).Vzhledem k funkčním problémům bylo od podpory testů kompatibility v rámci projektu nového grafického prostředí ustoupeno. Po jejich vyřešení na úrovni implementace úložiště je, vzhledem k modulárnímu systému frameworku Vaadin, který byl použit pro implementaci grafického prostředí, možné doplnit příslušné formuláře (implementace třídy *FormLayout*) o podporu spouštění testů kompatibility. Další využití projektu *crce-*

[13]Databázový systém Mongo DB - `https://www.mongodb.com/`

jacc je zejména pro sestavování aplikací ze vzájemně kompatibilních verzí daných komponent. Více informací o frameworku Vaadin bude uvedeno v kapitole číslo 6.3.

Stávající způsob získávání artefaktů z lokálního souborového systému nebo z url odkazu zůstal v nové implementaci webového grafického prostředí zachován. Respektive byl přepracován s využitím podpory frameworku Vaadin. V rámci této práce je plánováno rozšíření o získávání a stahování artefaktů z externích úložišť. V následujících kapitole číslo 6 budou analyzovány možné technologie, u nichž je předpokládáno jejich využití při tomto rozšíření funkčnosti.

6 Technologie pro implementaci

V této kapitole budou uvedeny implementační technologie pro vyhledávání a získávání komponent z externích zdrojů. Dále bude charakterizován rámec pro vytvoření webové aplikace pro uživatelsky přívětivou obsluhu nových funkcí.

6.1 Aether

Knihovna umožňuje vyhledávat a stahovat artefakty z externích úložišť s kompatibilním rozhraním.

6.1.1 Charkateristika

Knihovnu Aether vyvinula společnost Sonatype jako aplikační rozhraní pro přístup k repositáři artefaktů, zejména pak k vlastní implementaci Nexus. Její další vývoj byl uvolněn v rámci komunity Eclipse Foundation. [Aet] Jak již bylo uvedeno v kapitole číslo 4.3: distribuce aplikace s uvedením popisu závislostí a fyzickém stahování souborů ze vzdálenéných úložišť až při jejím sestavování a nasazování do běhového prostředí, je velmi efektivní přístup.

Každý artefakt v externím Maven úložišti je jednoznačně identifikován kombinací identifikátorů: skupina, název artefaktu, verze a přípony. Základním předpokladem je, že je tato konkrétní kombinace identifikátorů známa. Knihovna Aether umožňuje připojení vzdálených repositářů, tzn. že pokud není hledaný artefakt nalezen v jednom z externích úložišť, prohledává se další. Aby se při každém sestavování aplikace nemusel příslušný artefakt po síti opakovaně stahovat, knihovna Aether vytvoří tzv. lokální Maven úložiště, do nějž jsou ukládány všechny doposud potřebné artefakty. Lokální Maven úložiště je při ověřování existence artefaktu výchozí. Jeho umístění v souborovém systému lze definovat parametrem.

V předchozím odstavci bylo uvedeno, že pro výběr artefaktu je nezbytné uvést jeho konkrétní identifikátory. Toto platí s jednou vyjímkou. Knihovna Aether umožňuje definovat rozsah verzí výběru příslušné komponenty.

6.1.2 Příklad použití

V úložišti GitHub projektu *aether-demo* [Aetd] jsou dostupné komplexnější ukázky zdrojového kódu implementace. Pro účely objasnění principu a zejména z důvodu možného využití v praktické části práce budou dále uvedeny pouze základní části. Na obrázku číslo 14 jsou uvedeny závislosti při užití knihovny Aether.

```
<dependency>
        <groupId>org.eclipse.aether</groupId>
        <artifactId>aether-impl</artifactId>
        <version>1.1.0</version>
</dependency>
<dependency>
        <groupId>org.eclipse.aether</groupId>
        <artifactId>aether-connector-basic</artifactId>
        <version>1.1.0</version>
</dependency>
<dependency>
        <groupId>org.eclipse.aether</groupId>
        <artifactId>aether-transport-file</artifactId>
        <version>1.1.0</version>
</dependency>
<dependency>
        <groupId>org.eclipse.aether</groupId>
        <artifactId>aether-transport-http</artifactId>
        <version>1.1.0</version>
</dependency>
<dependency>
        <groupId>org.apache.maven</groupId>
        <artifactId>maven-aether-provider</artifactId>
        <version>3.3.9</version>
</dependency>
```

Obrázek 14: Závislosti při využití knihovny Aether

Základní částí je implementace rozhraní *RepositorySystem*. Jedná se hlavní vstupní bod úložitě a jeho funkcionality. Ukázka kódu je uvedena na obrázku číslo 15.

```
private 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);
}
```

Obrázek 15: Metoda vytváří vstupní bod úložiště

Třída *DefaultServiceLocator* představuje jednoduchý vyhledávač služeb. Pro získání kompletního repozitářového systému musí klienti přidat konektory úložiště pro vzdálené přenosy. Třída utilit *MavenRepositorySystemUtils* založí systém úložišť typu Maven. Metoda *newServiceLocator()* vytvoří nový lokátor služby, který již ví o všech implementacích služby zahrnutých v této knihovně.

Na obrázku číslo 16 je uvedena ukázka kódu metod pro vytvoření objektu třídy *RepositorySystemSession* a *RemoteRepository*.

```
private static RepositorySystemSession newSession(RepositorySystem system, String localAetherUrl) {
        DefaultRepositorySystemSession session = MavenRepositorySystemUtils.newSession();
        LocalRepository localRepo = new LocalRepository(localAetherUrl);
        session.setLocalRepositoryManager(system.newLocalRepositoryManager(session, localRepo));
        return session;
}

public static RemoteRepository newCentralRepository(String centralAetherUrl) {
        return new RemoteRepository.Builder("central", "default", centralAetherUrl).build();
}
```

Obrázek 16: Metody pro vytvoření relace se vzdáleným úložištěm

Rozhraní třídy *RepositorySystemSession* definuje nastavení a komponenty, které řídí systém úložiště. Jakmile je inicializován, samotný objekt relace má být neměnný, a proto může být bezpečně sdílen přes celou aplikaci a všechny souběžné podprocesy, které ji čtou. Komponenty, které si přejí vyladit některé aspekty existující relace, by měly použít konstruktor třídy *DefaultRepositorySystemSession* a jeho mutátory k odvození vlastní relace. Parametry metody *newSession()* jsou: objekt *RepositorySystem*, viz metoda uvedena na obrázku číslo 15, a řetězec udávající cestu pro vytvoření lokálního Maven úložiště.

Třída *RemoteRepository.Builder* vytvoří novou instanci úložiště, kde použije zadaný vzdálený repozitář jako prototyp nového. Jako parametr přebírá metoda *newCentralRepository()* url odkaz na vzdálené úložiště.

Na obrázku číslo 17 je zobrazen kód pro vytvoření spojení s centrálním Maven úložištěm (objekt *externalAetherUrl* obsahuje textový řetězec adresy) a vyhledáním artefaktu s identifikátory: groupId (*mysql*), artefactId (*mysql-connector-java*) a verzí (*[8.1.15]*). Pokud daný artefakt není nalezen v lokálním repository (objekt *localAetherUrl* udává textový řetězec adresy), je prohledáno vzdálené úložiště. Pokud je ve vzdáleném úložišti objekt nalezen, je stažený do lokálního Maven repostirory.

Při znalosti kompletních identifikátorů artefaktu v centálním Maven úložišti je možné sestavit přímý url odkaz. V tomto případě je možné se obejít bez podpory Aether. V situaci, kdy ale není známá verze komponenty, knihovna poskytuje možnost vypsání seznamu verzí, které jsou v repository dostupné. *RepositorySystem*.

```
private String localAetherUrl = "aether-local-repo";
private String externalAetherUrl = "http://repo.maven.apache.org/maven2/";

RepositorySystem repoSystem = newRepositorySystem();
RepositorySystemSession session = newSession(repoSystem, localAetherUrl);

ArtifactRequest artifactRequest = new ArtifactRequest();
//Artifact artifact = new DefaultArtifact("groupId:artifactId:(0,]");
Artifact artifact = new DefaultArtifact("mysql:mysql-connector-java:[8.1.15]");

artifactRequest.setArtifact(artifact);

artifactRequest.addRepository(newCentralRepository(externalAetherUrl));
ArtifactResult artifactResult;
try {
        artifactResult = repoSystem.resolveArtifact(session, artifactRequest);
        artifactResult.getArtifact();
} catch (ArtifactResolutionException e) {
        return false;
}
return true;
```

Obrázek 17: Ukázka kódu pro vyhledání artefaktu v centrálním Maven úložišti

Na obrázku číslo 18 je uveden úsek kódu pro výpis seznamu verzí hledaného artefaktu, které jsou dostupné v úložišti. V ukázce kódu je důležitý zejména objekt třídy *DefaultArtifact* implementující rozhraní *Artifact*. Objekt této třídy reprezentuje nalezený artefakt a je možné přistupovat k jeho atributům. Pro vyřešení požadavku na výběr dostupných verzí je k dispozici třída *VersionRangeResult*. Objekt této třídy vrací metoda *resolveVersionRange(RepositorySystemSession session, VersionRangeRequest request)* třídy *RepositorySystem*.

Na obrázku číslo 18 je uveden řetězec $''(,)''$, přiřazený proměnné *versionText*. Tento výběrový předpis udává, že budou do výsledku vyhledávání zahrnuty všechny dostupné verze artefaktu. Při požadavku na omezený rozsah verzí lze zadat například následující syntaxi: *(5.0.2,6.1.0)*. Tento zápis značí, že jsou požadovány všechny verze od *5.0.2* do *6.1.0*, přičemž kulaté závorky $($ $)$ znamenají otevřený interval zdola i zhora (udané verze nebudou zahrnuty do výběru). Nahrazení kulatých závorek hranatými $[$ $]$ značí uzavřený interval zdola i zhora (dané verze, pokud v úložišti existují, budou zahrnuty do výběru).

Z výše uvedené syntaxe je zřejmé, že pokud není uvedena hodnota dolní, respektive horní meze intervalu verzí, jedná se o ekvivalentní zápis $<$ resp. $>$. V případě užití uzavřeného intervalu (hranaté závorky) se jedná o zápis $<=$ resp. $>=$.

```
private String externalAetherUrl = "http://repo.maven.apache.org/maven2/";

RemoteRepository central = new RemoteRepository.Builder("central", "default", externalAetherUrl)
                .build();
RepositorySystem repoSystem = newRepositorySystem();
RepositorySystemSession session = newSession(repoSystem, settings.getLocalAetherUrl());
String versionText = "( ,)";
Artifact artifact = new DefaultArtifact("mysql" + ":" + "mysql-connector-java" + ":" + versionText);
VersionRangeRequest request = new VersionRangeRequest(artifact, Arrays.asList(central), null);
try {
        VersionRangeResult versionResult = repoSystem.resolveVersionRange(session, request);
        if (!versionResult.getVersions().isEmpty()) {
                if (versionResult.getVersions().size() > 1) {
                        // Výpis výsledků
                        System.out.println("Skupina" + artifact.getGroupId());
                        System.out.println("Artefakt id" + artifact.getArtifactId());
                        System.out.println("Dostupné verze:");
                        for (Version v : versionResult.getVersions()) {
                                System.out.println(v.toString());
                        }
                }
        }

} catch (VersionRangeResolutionException e) {
        e.printStackTrace();
}
```

Obrázek 18: Ukázka kódu pro výpis dostupných verzí nalezeného artefaktu

Pokud by byl požadavek na vrácení nevyšší, respektive nejnižší verze daného artefaktu, která je v repository dostupná, lze použít metodu *getHighestVersion()*, respektive *getLowestVersion()* objektu třídy *VersionRangeResult*.

Na obrázcích číslo 17 a 18 je jako externí úložiště použita adresa centrálního Maven repository (proměnná *externalAetherUrl*). Lze použít jako externí Maven úložiště vlastní implementaci Nexus uvedenou v kapitole 4.3. Pokud bylo požadováno připojení zabezpečeného úložiště Nexus (ověření přístupu), do proměnné *externalAetherUrl* by se zadal například následující řetězec:

```
String externalAetherUrl = "http://login:heslo@nexus:
8081/nexus/content/repositories/releases/";,
```

kde *login* udává přístupové jméno, *heslo* znamená přístupové heslo. Text *nexus* představuje ip adresu nebo DNS záznam úložiště a číslo *8081* port, na kterém je repository dostupné. Textový řetězec za portem značí kontext.

6.2 Maven indexer

V předchozí kapitole byla uvedena technologie k získávání artefaktů z externích Maven úložiť. Bylo uvedeno, že pro vyhledání artafaktů je nutné přesně znát jeho identifikátory: *groupId*, *artafactId* a *package* (typ souboru). V případě, že tyto identifikátory nejsou známé, lze použít textové vyhledávání s využitím indexace popisu artefaktů.

6.2.1 Lucene index

Základní princip fulltextového vyhledávání v Maven uložištích je založen na technologii Lucene [Luc]. Knihovnu navrhli v roce 2000 Doug Cutting[14] a Mike Cafarella[15]. V současné době je podporována Apache Software Foundation a je vydávána pod Apache Software licencí. Společnost Sonatype použila principy technologie vyhledávání Lucene pro svou implementaci úložiště Nexus.

Ve stručném popisu by se dal algoritmus vyjádřit následujícím způsobem: nejprve je nutné text převést do formátu, který umožní rychlé vyhledávání, čímž se eliminuje proces pomalého sekvenčního skenování. Tento proces převodu se nazývá indexování a jeho výstup se nazývá index. Index lze považovat za datovou strukturu, která umožňuje rychlý náhodný přístup ke slovům v něm uloženým. Koncept je analogický s indexem na konci knihy a umožňuje rychle vyhledat stránky, které diskutují o určitých tématech. V případě Lucene je index speciálně navrženou datovou strukturou, která je obvykle uložena v systému jako sada indexových souborů. [GH05]

Proces vytváření indexu tvoří následující fáze:

- získávání obsahu,
- tvorba dokumentu,
- analýza dokumentu,
- indexace dokumentu.

[14] Douglass Read Cutting - softwarový návrhář a tvůrce open-source vyhledávací technologie.

[15] Mike Cafarella - počítačový expert specializující se na systémy správy databází . Je vědeckým pracovníkem University v Michiganu.

Získávání obsahu je první fází analýzy. Jedná se o proces zahrnující použití prohledávacího algoritmu, který shromažďuje a rozšiřuje obsah, jež je třeba indexovat. To může být triviální, například pokud je indexována sada souborů XML, která je umístěna v určitém adresáři v systému souborů, tj. pokud je veškerý obsah umístěn v organizovaném prostředí. Alternativně může být složitý a chaotický, tj. obsah je roztroušen na nejrůznějších místech. Při použití oprávnění, což znamená, že pouze ověření uživatelé mohou vidět určité dokumenty, může tato skutečnost zkomplikovat proces získávání obsahu. Lucene, jako hlavní vyhledávací knihovna, neposkytuje žádnou funkcionalitu pro podporu získávání obsahu. Tato funkčnost musí být implementována v samostatném softwaru.

Ve fázi **tvorba dokumentu** dochází k třídění hrubého obsahu, který je třeba indexovat, do jednotek používaných vyhledávačem. Tyto jednotky se obvykle nazývají dokumenty. Dokument se obvykle skládá z několika odděleně pojmenovaných polí s hodnotami, jako je například název, tělo, abstrakt, autor a adresa URL. Je nutné pečlivě navrhnout, jak rozdělit nezpracovaný obsah do dokumentů a polí a jak vypočítat hodnotu pro každé z těchto polí. Další součástí tvorby dokumentu je raiting oblastí, které jsou považovány ze více nebo méně důležité

Analýza dokumentu rozděluje dokument na jednotlivé atomické prvky, které se nazývají tokeny. Každý token odpovídá zhruba slovu v daném jazyce a tento krok určuje, jak jsou textová pole v dokumentu rozdělena do řady tokenů.

Indexace dokumentu je poslední fází analýzy. Během indexování je dokument zařazen do příslušné položky indexu.

Jednotlivými fázemi prochází také proces vyhledávání. Jedná se o:

- vyhledávací uživatelské rozhraní,
- sestavení dotazu,
- spuštění vyhledávacího dotazu,
- výpis výsledků.

Lucene algoritmus tvoří jádro funkčnosti full-textového vyhledávacího procesu pro Maven úložitě. Z tohoto důvodu byl ve stručnosti vysvětlen základní princip Lucene indexu. Ovšem v Maven úložištích není prováděna indexace textových dokumentů, ale řetězců označujících název artefaktu a příslušné skupiny.

6.2.2 Příklad použití

Na obrázku číslo 19 jsou uvedeny potřebné závislosti na externích knihovnách při použití indexace a fultextového prohlédávání v centrálním Maven úložišti.

```
<dependencies>
        <dependency>
                <groupId>org.apache.maven.indexer</groupId>
                <artifactId>indexer-core</artifactId>
                <version>5.1.2-4333789</version>
        </dependency>
        <dependency>
                <groupId>org.apache.maven.wagon</groupId>
                <artifactId>wagon-http-lightweight</artifactId>
                <version>2.3</version>
        </dependency>
        <dependency>
                <groupId>org.eclipse.sisu</groupId>
                <artifactId>org.eclipse.sisu.plexus</artifactId>
                <version>0.3.3</version>
        </dependency>
        <dependency>
                <groupId>org.eclipse.aether</groupId>
                <artifactId>aether-util</artifactId>
                <version>1.1.0</version>
        </dependency>
        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-simple</artifactId>
                <version>1.7.22</version>
        </dependency>
        <dependency>
                <groupId>org.sonatype.sisu</groupId>
                <artifactId>sisu-guice</artifactId>
                <version>3.2.6</version>
        </dependency>
</dependencies>
```

Obrázek 19: Požadované závislosti při užití Maven indexeru

Na obrázcích 20, 21, 22 a 23 je uveden kód pro vytvoření indexu. [Ind] Z důvodu přehlednosti jsou jednotlivé části kódu naznačeny samostatně. Na obrázku číslo 20 je zobrazen kód třídy *MavenIndex*. Obsahuje řetězce udávající cesty umístění indexu v souborovém systému, instanci třídy *MavenIndex* a volání její metody

perform(). Dále obsahuje deklaraci objektů tříd, které jsou užity při vytváření indexu úložiště.

```
public class MavenIndex {
        private String centralCachePath = "repository-index/central-cache";
        private String centralIndexPath = "repository-index/central-index";

        public String checkIndex() throws Exception {
                final MavenIndex indexCentralRepository = new MavenIndex();
                return indexCentralRepository.perform(settings);
        }

        private final PlexusContainer plexusContainer;
        private final Indexer indexer;
        private final IndexUpdater indexUpdater;
        private final Wagon httpWagon;
        private IndexingContext centralContext;
}
```

Obrázek 20: Výchozí zdrojový kód třídy *MavenIndex*

Na obrázku číslo 21 je zobrazen kód konstruktoru třídy *MavenIndex*. Voláním konstruktoru jsou postupně vytvořeny objekty tříd, jejichž deklarace je uvedena na obrázku číslo 20. Za pozornost stojí zejména instance třídy *DefaultPlexusContainer*. Jedná se o implementaci rozhraní *PlexusContainer* respektive IoC[16] kontejneru. Metoda objektu *lookup()* vyhledá a vrátí objekt komponenty odpovídající jejímu popisovači.

```
public MavenIndex() throws PlexusContainerException, ComponentLookupException {
        this.plexusContainer = new DefaultPlexusContainer();
        this.indexer = plexusContainer.lookup(Indexer.class);
        this.indexUpdater = plexusContainer.lookup(IndexUpdater.class);
        this.httpWagon = plexusContainer.lookup(Wagon.class, "http");
}
```

Obrázek 21: Konstruktor třídy *MavenIndex*

Na obrázku číslo 22 je zobrazena první část kódu metody *perform()*. Metoda obsahuje seznam (*List*) objektů třídy *IndexCreator*. Třída *IndexCreator* je odpovědná za ukládání a čtení dat v Lucene indexu. Spuštění vytváření indexu provede metoda *createIndexingContext()* třídy *Indexer*.

[16]IoC - Inversion of Control

```
private String perform()
                throws IOException, ComponentLookupException {
        File centralLocalCache = new File(centralCachePath);
        File centralIndexDir = new File(centralIndexPath);

        List<IndexCreator> indexers = new ArrayList<IndexCreator>();

        indexers.add(plexusContainer.lookup(IndexCreator.class, "min"));
        indexers.add(plexusContainer.lookup(IndexCreator.class, "jarContent"));
        indexers.add(plexusContainer.lookup(IndexCreator.class, "maven-plugin"));

        String centralMavenUrl = "http://repo.maven.apache.org/maven2/";
        centralContext = indexer.createIndexingContext("central-context", "central",
                centralLocalCache, centralIndexDir, centralMavenUrl, null, true,
                true, indexers);

        System.out.println("Aktualizace indexu Maven...");
```

Obrázek 22: Metoda *perform()* třídy *MavenIndex* - 1. část

Na obrázku číslo 23 je zobrazena druhá část kódu metody *perform()*. Konstruktor třídy *WagonFetcher* nadtřídy *WagonHelper* vytvoří objekt implementace rozhraní *ResourceFetcher*. Konstruktoru třídy *WagonFetcher* je předán parametr (handler) instance *AbstractTransferListener*. Jedná se o implementaci rozhraní *TransferListener*. Dále je pak použit pouze objekt implementace rozhraní *ResourceFetcher*, který je předán jako parametr konstruktoru třídy *IndexUpdateRequest*. Voláním konstruktoru vznikne nový objekt třídy *IndexUpdateRequest*.

Celý proces vytváření indexu vzdáleného úložiště, jehož programový kód byl popsán v předchozích odstavcích, může mít, v případě úspěšného dokončení, trojí výsledek. Pokud v daném umístění index ještě nebyl vytvořen, zakládá se kompletně znova - viz návratová hodnota *true* metody *isFullUpdate()* objektu třídy *UpdateResult*. V případě, že v zadaném umístění je již Maven index vytvořen, umožňuje program inkrementální indexaci, která může, v případě úspěšného dokončení, skončit dvojím způsobem. Porovná se rozdíl mezi staženým indexem a změnou ve vzdáleném úložišti. Pokud není změna, vypíše se informace, že je index aktuální. V opačném případě se provede pouze přírůstková indexace. Je zřejmé, že tento přístup urychluje indexaci úložiště.

Pokud je indexace vzdáleného úložiště dokončena, ověřena aktuálnost či doplněn index přírůstku, je možné provádět full-textové dotazy na existenci artefaktů. Jinými slovy musí být vytvořen objekt *IndexingContext* voláním metoda *createIndexingContext* objektu třídy *Indexer*.

```
        TransferListener listener = new AbstractTransferListener() {
                public void transferStarted(TransferEvent transferEvent) {
                        System.out.print(" Stahuji " + transferEvent.getResource().getName());
                }

                public void transferProgress(TransferEvent transferEvent, byte[] buffer, int length) {

                }

                public void transferCompleted(TransferEvent transferEvent) {
                        System.out.println(" - Dokončeno");
                }
        };

        ResourceFetcher resourceFetcher = new WagonHelper.WagonFetcher(httpWagon, listener, null, null);
        Date centralContextCurrentTimestamp = centralContext.getTimestamp();

        IndexUpdateRequest updateRequest = new IndexUpdateRequest(centralContext, resourceFetcher);
        IndexUpdateResult updateResult = indexUpdater.fetchAndUpdateIndex(updateRequest);

        indexer.closeIndexingContext(centralContext, false);

        if (updateResult.isFullUpdate()) {
                return "Plná aktualizave dokončena.";
        } else if (updateResult.getTimestamp().equals(centralContextCurrentTimestamp)) {
                return "Není nutná žádná aktualizace, index je aktuální.";
        } else {
                return "Došlo k postupné aktualizaci "
                                + centralContextCurrentTimestamp + " - "
                                + updateResult.getTimestamp() + " časové období,"
                                + " na které se vztahuje změna.";
        }
}
```

Obrázek 23: Metoda *perform()* třídy *MavenIndex* - 2. část

Obrázek 24 znázorňuje zdrojový kód metody *getArtefact()*. Jedná se o demonstrační ukázku vyhledání artefaktu, který je definován identifikátory: skupinou, názvem, verzí a typem. Tyto identifikátory tvoří textové řetězce a jsou zároveň parametry metody *getArtefact()*, tj. *group*, *artifact*, *version*, *packaging*. Metoda vrací seznam (*List*) nalezených, respektive zadanému dotazu v repository odpovídajích artefaktů jako objektů třídy *ArtifactInfo*. Jak již bylo uvedeno v kapitole 6.2.1, Lucene index nezajišťuje získávání obsahu. To znamená, že fyzické stažení artefaktu musí zajistit aplikace. V případě, že jsou známy identifikátory artefaktu, což jsou atributy objektu třídy *ArtifactInfo*, spolu s adresou úložiště, je možné sestavit příslušný url odkaz.

Na obrázku číslo 24 je sestavení příslušného vyhledávacího dotazu zajištěno objektem třídy *BooleanQuery*. Jedná se o seznam jednotlivých částí dotazu respektive do tohoto seznamu vložených objektů třídy *Query*. Parametrem předání, kro-

mě hledaného textového řetězce, je i výraz výskytu. Jde o výčet *Occur*, který může nabývat hodnot: *MUST*, *MUST_NOT* nebo *SHOULD*. Konstruktor třídy *IteratorSearchRequest* přijímá jako parametr instanci *BooleanQuery*, seznam objektů třídy *IndexContext* a také implementaci rozhraní *ArtifactInfoFilter*.

```
private List<ArtifactInfo> getArtefact(String group, String artifact, String version, String packaging,
     String range) throws IOException, ComponentLookupException, InvalidVersionSpecificationException {

     final GenericVersionScheme versionScheme = new GenericVersionScheme();
     Version versionIndex = versionScheme.parseVersion(version);
     final Query groupIdQ = indexer.constructQuery(MAVEN.GROUP_ID, new SourcedSearchExpression(group));
     final Query artifactIdQ = indexer.constructQuery(MAVEN.ARTIFACT_ID,
                    new SourcedSearchExpression(artifact));
     final BooleanQuery query = new BooleanQuery();
     query.add(groupIdQ, Occur.MUST);
     query.add(artifactIdQ, Occur.MUST);
     query.add(indexer.constructQuery(MAVEN.PACKAGING, new SourcedSearchExpression(packaging)),
          Occur.MUST);
     query.add(indexer.constructQuery(MAVEN.CLASSIFIER, new SourcedSearchExpression(Field.NOT_PRESENT)),
                    Occur.MUST_NOT);

     // Zde musí být vložena implementace rozhraní ArtifactInfoFilter

     final IteratorSearchRequest request = new IteratorSearchRequest(query,
                    Collections.singletonList(centralContext), versionFilter);
     final IteratorSearchResponse response = indexer.searchIterator(request);
     List<ArtifactInfo> artifactInfoList = new ArrayList<ArtifactInfo>();
     for (ArtifactInfo ai : response) {
             artifactInfoList.add(ai);
     }
     indexer.closeIndexingContext(centralContext, false);

     return artifactInfoList;
 }
```

Obrázek 24: Příklad využití vytvořeného indexu - metoda *getArtefact()*

Příklad implementace rozhraní *ArtifactInfoFilter* je uveden na obrázku číslo 25. V tomto konkrétním případě se jedná o filtr verzí výbraných záznamů. Request je předán jako parametr metodě `searchIterator()` indexeru, objektu třídy *Indexer*. Vrácená hodnota představuje seznam dotazu vyhovujících objektů třídy *ArtifactInfo*.

Následující informace vychází z principu Lucene indexace: je možné při vyhledávání používat neúplné řetězce, přičemž vynechaná část je doplněná znakem $*$. Vynechání textu nelze použít na začátku hledaného výrazu, ale jinak v rámci textu bez omezení. Nemožnost použít samotný znak $*$ je dána skutečností, že v opačném případě by index položek dosahoval enormní velikosti.

Z praktických testování funkčnosti výše uvedených kódů vyplynul tento poznatek: požadované komponenty, uvedené jako závislosti (viz obrázek číslo 19), je nezbytné v daných verzích dodržet. Klíčová závislost je zejména *org.sonatype.sisu-/sisu-guice/3.2.6*, poskytující výše uvedený IoC kontejner. Podobných nebo do-

```
final ArtifactInfoFilter versionFilter = new ArtifactInfoFilter() {
        public boolean accepts(final IndexingContext ctx, final ArtifactInfo ai) {
                try {
                        final Version aiV = versionScheme.parseVersion(ai.version);
                        if(range.equals("<=")){
                                return aiV.compareTo(versionIndex) <= 0;
                        }
                        else if(range.equals(">=")){
                                return aiV.compareTo(versionIndex) >= 0;
                        }
                        else{
                                return aiV.compareTo(versionIndex) == 0;
                        }

                } catch (InvalidVersionSpecificationException e) {
                        return true;
                }
        }
};
```

Obrázek 25: Implementace rozhraní *ArtifactInfoFilter*

konce identických implementací IoC kontejneru je více. Zejména v prostředí OSGi může nastat situace, kdy požadované závislosti poskytne jako své vlastní schopnosti jiný modul. Typickým příkladem je knihovna Google Guice Core (*com.google.inject/guice*). V tomto případě může vzniknout nekompatibilita a indexace nebude funkční.

6.3 Vaadin

Vaadin je softwarový framework pro tvorbu webových aplikací. [Vaa] Jeho výhoda tkví v odstínění od problematiky serverové a klientské části, kterou musí řešit klasické webové aplikace (založené např. na technologiii JSP a servletů). Vývojáři mohou přistupovat k webovým aplikacím, jako by se jednalo o desktopové. Vaadin je vytvořený v programovacím jazyku Java. Pomocí nástroje Google Web Toolkit (GWT) je překládán do Java Scriptu a následně interpretován prohlížečem. Framework využíval technologii GWT až do verze 8.

Od verze číslo 10 (verze 9 byla vynechána) došlo k zásadní změně. Jedná se o opuštění principu výchozí komponenty (UI) a přechod na odlišný způsob směrování mezi jednotlivými formuláři využitím anotace - Vaadin Flow[17]. Pro uživatelské prvky je nově využívána knihovna Polymer[18]. Výhodou frameworku Vaa-

[17] Vaadin Flow - `https://vaadin.com/flow`
[18] Polymer - `https://www.polymer-project.org/`

din je přístup tvorby webových formulářů založený na hotových komponentách uživatelských prvků. Tato skutečnost je výhodná pro jednoduchý a rychlý vývoj aplikace.

V roce 2018 v rámci projektu KIV/OPSWI bylo navrhnuto a vytvořeno nové grafické uživatelské prostředí pro úložitě CRCE, respektive správu životního cyklu artefaktů v úložišti na platformě frameworku Vaadin ve stabilní verzi 7. Při každé změně makroverze frameworku je nezbytné provést změny. Úprava aplikace na kompatibilní verzi s Vaadin 8 by sice nebyla obtížná, ale nepřinesla by žádné nové přidané funkčnosti. V případě povýšení aplikace pro kompatibilitu s frameworkem verze 10 LTS[19] nebo na aktuální verzi 13 (non-LTS) by znamenalo provést změny na úrovni směrování stránek, ale přechod na Vaadin Flow by také nebyl složitý.

Zásadní problém je v ukončení podpory důležitých komponent (např. *Tree*, *MenuBar*, *MenuItem*) ve verzi 10 a výše, zejména z důvodu, že tyto komponenty jsou úzce vázány s desktopovou platformou a jejich zobrazování na mobilních zařízeních je ve stávající implementaci komplikované. Do budoucna byla přislíbena aktualizace uživatelských prvků s podporou knihovny Polymer, nicméně tyto v aplikaci klíčové komponenty stále ještě nejsou nahrazeny (duben 2019).

Bezplatná odpora Vaadin ve verzi 7 byla ukončena v lednu 2019. Rozšířená (placená) podpora je garantována až do roku 2029. Uvolnění verze 14 (LTS) se plánuje 5. června 2019. Předpokládá se, že v případě uvolnění aktualizovaných komponent ve verzi 14 bude možné provést úpravu aplikace, aby byla kompatibilní s frameworkem Vaadin 14 (LTS).

[19]LTS - Long-term support (pětiletá podpota tj. do roku 2023)

7 Návrh řešení

V předchozích kapitolách jsou analyzovány implementační technologie, které jsou vhodné pro rozšíření funkčnosti úložiště komponent CRCE. Stávající verze CRCE umožňuje uživatelsky nahrávat artefakty z lokálního souborového systému nebo pokud je známá a platná adresa zdroje (url). V kapitole 7.1 je uvedena charakteristika návrhu rozšíření těchto vlastností o možnost vyhledávání a získávání objektů z Maven úložiť.

Aby bylo možné udržovat a spravovat systém verzí jednotlivých komponentových aplikací, je nezbytné vést evidenci jednotlivých částí, které jsou v relaci k verzím aplikace. Je vzata v úvahu literatura [RBB11], [Con98]. V kapitole 7.2 je uvedena charakteristika návrhu systému evidence a správy množin artefaktů, které jsou dostupné v úložišti CRCE. Inspirace je převzatá z feature implementace OSGi Apache Karaf (viz kapitola 4.2.3) a verzovacího systému Red Hat (jBoss) Fuse uvedeným v kapitole 4.2.4.

7.1 Návrh rozšíření úložiště CRCE o externí zdroje dat

Pro rozšíření funkčnosti CRCE o možnost získávání artefaktů z externích úložišť bude použita knihovna Aether. Pro fultextové vyhledávání artefaktů v centrálním Maven úložišti je navrženo využití Maven indexeru.

Rozšíření obsahuje:

- správu lokálního Maven úložiště knihovny Aether,
- vyhledávání artefaktů v definovaném úložišti knihovnou Aether,
- výpis obsahu skupin v centrálním Maven úložišti,
- fultextové vyhledávání artefaktů v centrálním Maven úložišti.

Lokální Maven úložiště slouží pro urychlení stažení artefaktů z externího repository. Pokud je v lokálním úložišti objekt dostupný, je poskytnut a není nutné jej přenášet znovu po síti. Rozšíření bude obsahovat prohlížení seznamu artefaktů v lokálním Maven úložišti, vytvořeném knihovnou Aether, a nahrání označeného objektu do dočasného úložiště CRCE (Buffer). V určitých případech nemusí být žádoucí, aby artefakt zaujímal místo v souborovém systému lokálního Maven úložiště knihovny Aether a zároveň byl obsažen v úložišti CRCE. Ve výchozím nastavení není dovoleno provádět úpravy v lokálním Maven repository, ale po uživatelské změně nastavení bude aplikace umožňovat odebrání artefaktu či celké skupiny.

Definovaným Maven úložištěm se rozumí uživatelsky zadaný url odkaz na vzdálené Maven repository, jehož rozhraní podporuje knihovna Aether. Na základě zadaných parametrů: *názvem skupiny*, *názvu artefaktu*, *verze (nepovinné)* a *typu souboru* lze ověřit dostupnost artefaktu ve vzdáleném repository, provést jeho stažení do lokálního Maven úložiště a nahrání do CRCE (Buffer). V případě vynechání nepovinného parametru *verze* bude vypsán kompletní seznam dostupných verzí.

Centrální Maven úložiště je dostupné na url:

`http://repo.maven.apache.org/maven2/`.

Jedná se o globální úložiště artefaktů pro sestavování projektů pomocí nástroje Maven. Uvedený odkaz je možné použít jako definované úložitě, viz předchozí odstavec, a využít podporu knihovny Aether.

Dále je možné vytvořit přímý url odkaz uloženého objektu při znalosti *názvu skupiny*, *názvu artefaktu*, *verze* a *typu souboru*. Tohoto faktu bude využito pro přímé ověření dostupnosti a následném stažení objektu z url odkazu a také při fultextovém vyhledávání s využitím indexu. Aplikace nepovolí full-textové vyhledávání v centrálním Maven repository, pokud pod daným přihlášeným uživatelem není vytvořen index či ověřena jeho aktuálnost. Full-textové vyhledávání bude k dispozici pro skupinu a název artefaktu, přičemž bude možnost zadat neúplný řetězec znaků viz kapitola 6.2.2. Ale tomto případě není povoleno použít znak $*$ na začátku řetězce - omezení velikosti Lucene indexu. Bude možné definovat rozsah požadovaných verzí hledaných artefaktů. Vzhledem ke skutečnosti, že proces indexace (zejména kompletní) je časové náročný a poměrně značnou dobu by blokoval uživatelskou aplikaci, je navržen jako asynchronní, tzn. bude spouštěn v samostatném vláknu.

Jednotlivé webové stránky, dostupné přes výše uvedený odkaz, obsahují seznam skupin, podskupin a artefaktů jako odkazy na příslušné vnořené podskupiny či soubory. Metodou parsování webové stránky bude možné zobrazit kompletní stromovou strukturu podskupin a souborů zadané skupiny. Uvedená metoda je ovšem časově náročná, proto nebude ve výchozím nastavení povolená. Bude doporučeno, při uživatelském povolení této volby, aplikovat výpis podskupin pouze na skupiny s nízkým obsahem položek.

7.2 Návrh správy množin (kolekcí) artefaktů

Navržené rozšíření obsahuje:

- vytvoření nové kolekce artefaktů,
- vytvoření nové verze kolekce artefaktů z již existující kolekce,
- výpis seznamu uložených kolekcí,
- zobrazení informací o obsahu kolekce,
- stažení obsahu kolekce (artefaktů a příslušných metadat),
- editaci nebo odstranění uložené kolekce.

Artefakty, které budou obsahovat kolekce, budou dostupné z trvalého úložiště CRCE (Store). Vzhledem k chrakteru dočasného úložiště CRCE (Buffer), jehož instance je dostupná pouze přihlášenému uživateli a po odhlášení je odstraněna, nebylo o vytváření kolekcí artefaktů uložených v Bufferu uvažováno.

Fyzické umístění artefaktů, uložených ve Store, se nachází v souborovém systému. Pro jejich popisná data (metadata) je použit databázový systém MongoDB. Je navrženo využít stávající databázový systém a pro evidenci záznamů kolekcí rozšířit databázi *crce* o novou složku (collection). Záznamy v nové kolekci budou obsahovat tyto atributy:

- datový typ: *String*, název: *ObjectId* - identifikátor záznamu,
- datový typ: *String*, název: *name* - název kolekce,
- datový typ: *String*, název: *version* - verze kolekce,
- datový typ: *Array*, datový typ položky: *String*, název: *specificArtifact* - seznam podmožin nebo artefaktů,
- datový typ: *Array*, datový typ položky: *String*, název: *parameters* - seznam parametrů kolekce,
- datový typ: *Array*, datový typ položky: *String*, název: *rangeArtifact* - seznam artefaktů definovaných názvem a rozsahem verzí.

Kolekce artefaktů je jednoznačně definována identifikátorem záznamu v *ObjectId* v kolekci *collection*. Každý artefakt, který je uložen ve Store, má přiřazen unikátní identifikátor *_id* v kolekci *resources*. Je navrženo (inspirace feture a profily viz kapitoly 4.2.2 a 4.2.3): množiny artefaktů mohou obsahovat stávající kolekce

(podmožiny). Identifikátorům záznamů množin a resource odpovídají položky pole atributu *specificArtifact*.

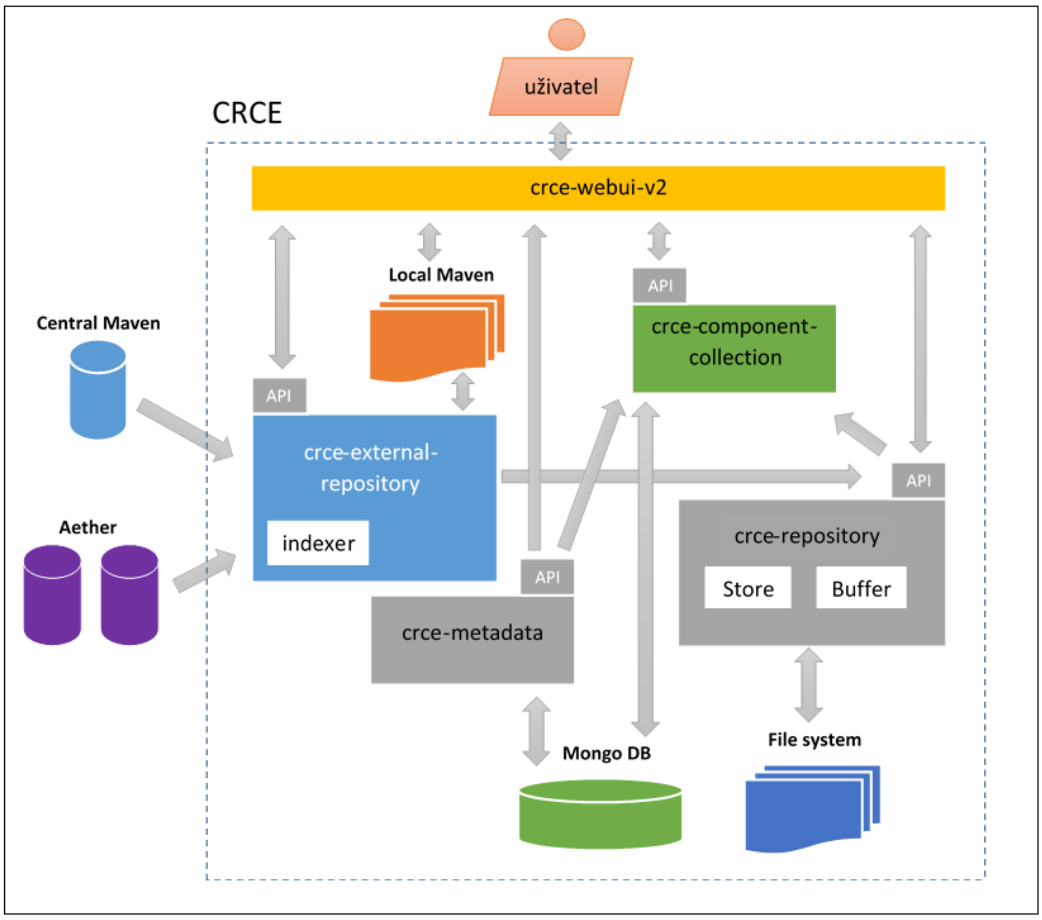
Popisnými atributy záznamu kolekce jsou její název a verze. V kapitole 4.2.3 bylo uvedeno, že každý profil dané verze může obsahovat seznam parametrů. Není nutné evidovat a nastavovat na globální úrovni centrální parametry pro každou aplikaci, a tedy každá aplikace sestavená z komponent má vlastní položky parametrů klíč/hodnota. Typickým příkladem může být url řetězec připojení databáze, přístupové údaje, nastavení časovače apod. Navženo: parametr se uvádí jako klíč/hodnota, kde hodnota je od klíče v řetězci oddělena znakem = (viz kapitola 4.2.3).

Kromě artefaktů ze Store a uložených kolekcí může množina obsahovat také objekty, u nichž není striktně vyžadovaná verze. Příklad je uveden na obrázku číslo 9 v podkapitole 4.2.2. Dle nastavených politik je možné vybrat příslušný artefakt v nejvyšší/nejnižší verzi, která je v úložišti dostupná. V Maven úložištích je situace zjednodušená tím, že každý objekt je jednoznačně definován *skupinou*, *názvem* a *typem souboru*. Ke každému artefaktu lze vypsat seznam příslušných verzí. Tento přístup ale CRCE nepodporuje. Je navržen zápis ve tvaru klíč/hodnota, kde klíč představuje atribut *symbolicName* objektu *resource* a hodnotu udává předpis rozsahu verzí uzavřený v hranatých závorkách. Rozsah verzí je oddělen čárkou, tj. znakem ,. Řetězec klíče a hodnoty jsou v záznamu odděleny znakem =.

Aplikace umožní export kolekce do souboru archivu `zip`. V archivu budou adresáře odpovídat příslušným kolekcím či podkolekcím (název, verze), v nich budou obsaženy soubory artefaktů s popisnými daty (seznam jejich schopností a požadavků) ve formátu xml a se syntaxí dle služby CRCE pro export metadat. Identickým způsobem budou exportovány také artefakty definované *symobolicName* a rozsahem verzí, viz odstavec výše. Návrhem je v nastavení uživatelsky definovat, zda bude exportována nejvyšší či nejnižší verze z uveného rozsahu. Rovněž bude možné uživatelsky definovat, zda bude exportován kompletní výpis požadavků a schopností či zjednodušená verze.

7.3 Integrace nových součástí do úložiště CRCE

Na obrázku číslo 26 je schématicky znázorněna součinnost nových komponent *crce-external-repository*, *crce-component-collection* a *crce-webui-v2* se stávajícími částmi úložiště CRCE.



Obrázek 26: Schéma integrace nových součástí do CRCE

8 Implementace

S ohledem na stávající modulární architekturu úložiště CRCE je rozšíření rovněž implementováno v samostaných komponentách. Nová webová aplikace (grafické uživatelské prostředí): modul *crce-webui-v2* pro správu životního cyklu artefaktů v úložišti CRCE, která byla navržena a vytvořena autorem této práce v rámci projektu KIV/OPSWI, je rozšířena o nové formuláře, které zajišťují uživatelský komfort pro nové funkcionality: externí zdroje dat a správa kolekcí, které poskytují dva samostatné moduly: *crce-external-repository* a *crce-component-collection*.

8.1 Modul crce-external-repository

Modul obsahuje v balíku *cz.zcu.kiv.crce.crce_external_repository.api* následující rozhraní:

- *CentralMavenApi*,
- *DefinedMavenApi*,

a implementační třídy v balíku *cz.zcu.kiv.crce.crce_external_repository.api.impl*

- *ArtifactTree*,
- *CentralMaven*,
- *DefinedMaven*,
- *MavenIndex*,
- *ResultSearchArtifactTree*,
- *ArtifactTree*,
- *SettingsUrl*.

Interní (neexportovatelný) balík *cz.zcu.kiv.crce.crce_external_repository.internal* obsahuje třídu:

- *Activator*.

CentralMavenApi - rozhraní předepisuje hlavičku metody pro získání artefaktu z centrálního úložiště Maven.

DefinedMavenApi - rozhraní určuje formu metody vyhledávání artefaktů v externím úložišti s podporou knihovny Aether.

ArtifactTree - instance této třídy představují objektově artefakt z externího úložiště s atributy: skupina, id, typ souboru, seznam dostupných verzí, url úložiště. Třída neobsahuje žádnou výkonnou metodu, pouze konstruktor a metody *get()* a *set()* pro čtení či změnu atributů objektů třídy.

ResultSearchArtifactTree - objekt této třídy je výsledek hledání v externím úložišti. Logická proměnná *status* udává, zda bylo hledání úspěšné. V případě, že ano, seznam je naplněn objekty třídy *ArtifactTree*.

SettingsUrl - atributy třídy udávají url lokátory externích úložišť. Dále pak umístění lokálního Maven úložiště a uživatelské volby (povolení/zakázání editace lokálního Maven úložiště a výpisu obsahu celé skupiny). Každé spojení (session) přihlášeného uživatele ve webové aplikaci (*crce-webui-v2*) má nastaven jako atribut objekt třídy *SettingsUrl*.

MavenIndex - metody této třídy zajišťují vytvoření či ověření aktuálnosti indexu a vyhledání artefaktu s využitím indexu v centrálním Maven úložišti. Podrobně jsou jednotlivé metody popsány v návrhu implementace viz podkapitola 6.2.2 (Příklad použití Maven indexeru).

CentralMaven - třída implementuje rozhraní *CentralMavenApi*. Obsahuje jednu veřejnou a čtyři soukromé metody. Veřejná metoda *getArtifactTree()* vrací objekt třídy *ResultSearchArtifactTree*. Jedná se o výsledek hledání artefaktů dle zadaných parametrů metody v centrálním Maven úložišti. Kromě skupiny, názvu artefaktu, verze a typu souboru přebírá metoda ještě textový řetězec s informací, z nejž rozliší, zda se jedná o přímý url lokátor na základě kompletní znalosti identifikátorů nebo o podporu indexového vyhledávání. Volány jsou podle těchto kritérií soukromé metody *directSearch()*, nebo *indexSearch()*. Pokud je uživatelsky povolen výpis celé skupiny a je dán požadavek na tento výpis, je volána soukromá metoda *onlyGroupSearch()*.

DefinedMaven - třída implementuje rozhraní *DefinedlMaveApi*. Je odpovědná za vyhledávání artefaktů z externích úložišť s podporou knihovny Aether. Klíčové metody jsou popsány v kapitole 6.1.2 v ukázce použití knihovny Aether. Třída *DefinedMaven* obsahuje dvě veřejné metody: *getArtifact()* vrací jako návratovou hodnotu objekt třídy *ArtifactTree* a *resolveArtifact()* ověří dostupnost hledaného objektu v externím úložišti. Návratová hodnota je *true* v případě úspěšného nalezení, respektive *false* v případě neúspěchu. V kapitole 6.1.2 je vysvětlena vlastnost knihovny Aether, kdy nejdříve hledá artefakt v lokálním úložišti a v případě neúspěchu v externích úložištích. Metoda *resolveArtifact()* ověří přítomnost arte-

faktu v lokálním Maven úložišti. Pokud není přítomen a je jeho existence úspěšně ověřena v externím repository, je tento objekt stažen automaticky do lokálního úložiště.

Komponenta *crce-external-repository* tvoří samostatný modul, tzn. nemá požadavky v běhovém prostředí OSGi. Externí závislosti, tj. podpora Aether a Indexeru, jsou staženy jako interní (vestavěné knihovny) při sestavování modulu.

8.2 Modul crce-component-collection

Modul obsahuje v balíku *cz.zcu.kiv.crce.crce_component_collection.api* následující rozhraní:

- *CollectionServiceApi*,
- *ExportCollectionServiceApi*.

Implementace výše uvedených rozhraní je obsažena v balíku *cz.zcu.kiv.crce.crce_component_collection.impl*. Jedná se o tyto třídy:

- *CollectionService*,
- *ExportCollectionService*.

Ve stejném balíku jsou uvedeny pomocné třídy:

- *HelperFileWriter*,
- *HelperResource*,

a výčtový typ:

- *LimitRange*.

Balík *cz.zcu.kiv.crce.crce_component_collection.api.bean* obsahuje komponenty:

- *CollectionBean*,
- *CollectionDetailBean*.

V balíku *cz.zcu.kiv.crce.crce_component_collection.api.settings* je umístěna třída:

- *SettingsLimitRange*.

Interní (neexportovatelný) balík *cz.zcu.kiv.crce.crce_component_collection.internal* obsahuje třídu:

- *Activator*.

CollectionServiceApi - rozhraní obsahuje hlavičky metod pro CRUD[20] operace nad záznamy množin artefaktů v kolekci *collection* databáze MongoDB.

CollectionDetailBean - rozhraní obsahuje hlavičky metod pro export kolekcí, artefaktů a jejich metadat do souborového systému a vytvoření archivu `zip`.

CollectionBean - programová komponenta[21], jejíž atributy (vlastnosti) odpovídají základním popisným položkám množin a artefaktů.

CollectionDetailBean - programová komponenta, jejíž atributy mapují záznamy v kolekci *collection* databáze MongoDB.

SettingsLimitRange - proměnné této třídy uchovávají nastavení uživatelských předvoleb pro správu kolekcí. Každé spojení (session) webové aplikace pro přihlášeného uživatele získá jako atribut instanci této třídy.

CollectionService - třída implementuje rozhraní *CollectionServiceApi*.

ExportCollectionService - třída implementuje rozhraní *ExportCollectionServiceApi*.

HelperResource - metody třídy umožňují zejména přístup v úložišti Store k objektům *Resource* a jejich vlastnostem. Dalšími metodami třídy jsou: *getResourceMaxVersionFromStore()*, respektive *getResourceMinVersionFromStore()*, které implementují algoritmus výběru nejvyšší, respektive nejnižší verze artefaktu dostupného v úložišti Store na základě předepsaného rozsahu požadovaných verzí. Soukromá metoda třídy *compareVersion()* je součástí implementace zmíněného algoritmu. Přebírá jako své parametry maximální, minimální a ověřované číslo verze.V případě, že ověřované číslo je mezi hodnotami minimální a maximální verze, vrací logickou hodnotu *true*. Interval rozsahu verzí je uzavřený z leva a otevřený z prava. V případě nastavení výběru minimální verze artefaktu je tento do výběru zahrnut, pokud je v úložišti přítomen. Při volbě nastavení výběru maximální verze artefaktu není „koncová“ verze artefaktu do výběru zahrnuta, i kdyby byl v úložišti artefakt zmíněné verze přítomen.

[20] CRUD - základní operace nad databází (Create, Read, Update, Delete)

[21] Bean - třída, jejíž proměnné lze měnit pouze metodami *get()* a *set()*

HelperFileWriter - třídní metody vykonávají export parametrů kolekcí a metadat artefaktů do souboru ve formátu xml. V případě exportu metadat se jedná o identický formát, který poskytuje webová služba *metadataDetails()* třídy *MetadataResJersey* modulu *crce-rest-v2* úložiště CRCE. Ve stávající verzi úložiště CRCE není ale tato webová služba funkční, proto třída *HelperFileWriter* obsahuje vlastní imlementaci exportu metadat artefaktů uložených ve Store.

Proces exportu se skládá z těchto fází:

- vytvoření složek odvídající kolekcím a podkolekcím,
- stažení artefaktů z CRCE - záznamy *specificArtifact*,
- stažení artefaktů z CRCE, které jsou definovány symbolickým jménem a rozsahem verzí - záznamy *rangeArtifact*,
- výpis parametrů kolekce do souboru ve formátu xml,
- výpis metadat stažených artefaktů do souborů ve formátu xml,
- vytvoření archivu zip.

Aktivator - jedná se o třídu, jejíž metody jsou volány při spouštění nebo naopak při zastavení svazku v běhovém prostředí OSGi. Dále obsahuje metody, které umožní přístup modulu k objektů *Store* a *MetadataService*.

8.3 Modul crce-webui-v2 (webová aplikace)

Modul webové aplikace grafického rozhraní byl autorem této práce vytvořen v projektu KIV/OPSWI pro uživatelskou správu životního cyklu artefaktů v úložišti CRCE. V rámci podpory nových funkčností je rozšířen o další formuláře, tj. objekty, které rozšiřují (dědí) od třídy *FormLayout* frameworku Vaadin. Formuláře jsou uloženy následujícím způsobem: balík *cz.zcu.kiv.crce.crce_webui_v2.outer* obsahuje formuláře pro podporu vyhledávání artefaktů v externích zdrojích dat:

- *CentralMavenForm*,
- *CheckMavenIndexForm*,
- *DefinedMavenForm*,
- *LocalMavenForm*.

Podbalík *classes* obsahuje třídu:

- *LocalMaven*.

balík *cz.zcu.kiv.crce.crce_webui_v2.collection* obsahuje formuláře pro podporu správy množin artefaktů:

- *CollectionEditForm*,
- *CollectionForm*,
- *CollectionNewForm*,
- *CollectionRangeParamDetailForm*.

A podbalících *classes* a *services* třídy:

- *ArtifactRangeBean*,
- *ParameterBean*,
- *RandomStringGenerator*,
- *FindCollectionService*.

Stávající formulář v balíku *cz.zcu.kiv.crce.crce_webui_v2.webui MenuForm*, byl rozšířen o nové položky menu a výchozí třída *MyUI* byla doplněna o nové metody načítající do svého těla výše uvedené formuláře. Dále byly vytvořeny dialogy, tj. třídy *SettingsUrlForm* a *SettingRangeForm* pro nastavení uživatelských předvoleb.

Každý formulář je složen z prvků grafického uživatelského rozhraní (komponent[22]). Jedná se například o tlačítka, seznamy, textová pole, popisy apod. Tyto komponenty jsou uloženy v horizontálním nebo vertikálním rozvržení *layout*. Pokud příslušný prvek podporuje, může reagovat na uživatele (událost). S výhodou jsou zde užívány anonymní metody (lambda). Podporovány jsou v Java od verze 8.

V následujícím textu jsou ve stručné formě charakterizovány jednotlivé formuláře pro načítání artefaktů z externích úložišť a správu kolekcí. Podrobnosti uživatelského ovládání budou uvedeny v podkapitole 8.4.

[22]Pozn. - Za povšimnutí stojí skutečnost, že tato práce je orientována na komponentovou architekturu aplikací. Framework Vaadin obdobným způsobem podporuje tvorbu uživatelských stránek ze samostaných prvků (komponent).

CentralMavenForm - třída (formulář) pro vyhledávání artefaktu v centrálním Maven úložišti.

CheckMavenIndexForm - dialog obsahuje tlačítko na spuštění vytváření či ověření aktuálnosti indexu centrálního Maven úložiště. Celý proces indexace běží v samostaném vláknu a nezablokuje hlavní uživatelský dialog. Index se vytváří, resp. ověřuje na serverové straně. O výsledku asynchronního procesu informuje serverová strana klientskou část použitím *Vaadin push* notifikace. Po úspěšném vytvoření nebo ověření existence indexu je automaticky nastaven atribut *mavenIndex* uživatelské *session* na *true*. Bez tohoto atributu není povolena volba pro indexové vyhledávání ve formuláři *CentralMavenForm*.

DefinedMavenForm - třída (formulář) pro vyhledávání artefaktu v uživatelsky definovaném Maven úložišti. Na stránce je možné vložit url odkaz vzdáleného úložiště. Jedná se o využití podpory knihovny Aether.

LocalMavenForm - uživatelský náhled na lokální Maven úložiště knihovny Aether. K načtení artefaktů a jejich zobrazení ve stromové struktuře slouží metoda *getTree()*, vracející komponentu *Tree* frameworku Vaadin. Metodu obsahuje třída *LocalMaven*.

CollectionForm - formulář zobrazuje tabulku uložených kolekcí a případně jejich obsah ve stromové struktuře. Dále je možné vytvoření kopie kolekce, volání editačního formuláře *CollectionEditForm*, odstranění uložené kolekce, příprava exportu množiny a její stažení ve formě archivu `zip`.

CollectionEditForm - formulář pro editaci stávající uložené kolekce.

CollectionNewForm - formulář pro vytvoření nové kolekce.

CollectionRangeParamDetailForm - zobrazení podrobností vybrané množiny, tj. přehled uložených parametrů a seznam artefaktů definovaných požadovaným rozsahem verzí.

FindCollectionService - třída obsahuje metodu *getFindCollectionBean()*. Jedná se o full-textový filtr seznamu uložených kolekcí.

ParameterBean a *ArtifactRangeBean* - programové komponenty, jejichž vlastnosti odpovídají hodnotám parametrů kolekcí a artefaktů s definovaným rozsahem verzí.

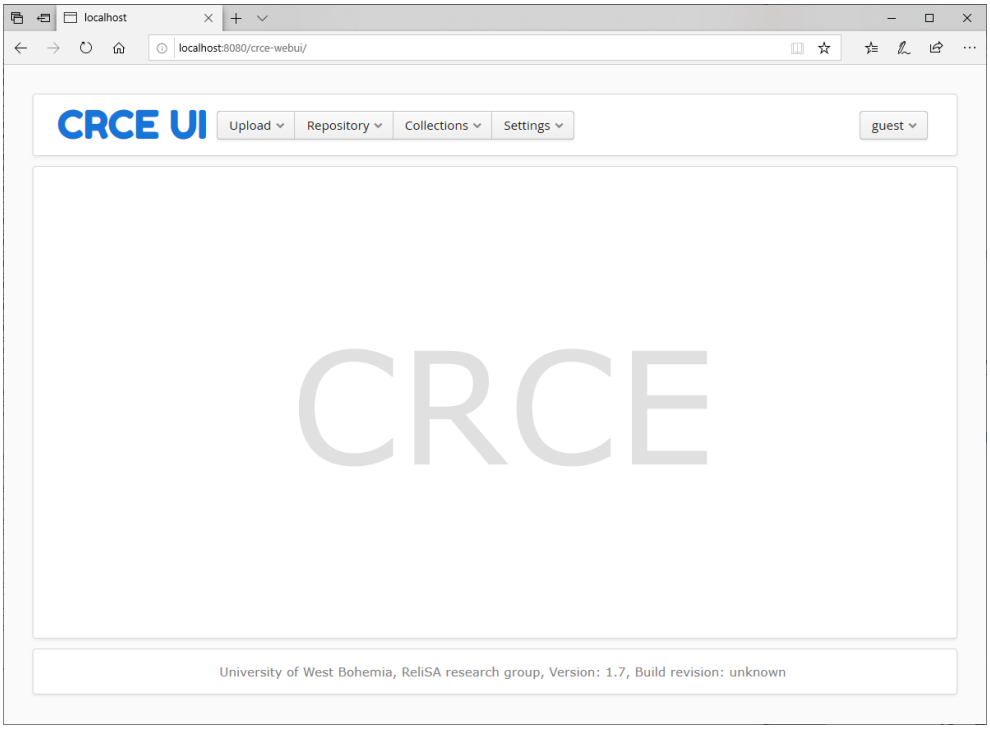
RandomStringGenerator - třída obsahuje metodu *getRandomString()*, která vrací náhodný řetězec 7 znaků. Jedná se o řešení problému komponenty *Tree* frameworku Vaadin. Do tohoto objektu není možné vložit stejné položky. Pokud by různé podkolekce obsahovaly stejné artefakty, byly by zobrazeny pouze jednou. Odlišení je řešeno přidání náhodného řetězce za název položky. Jako popis je ovšem zobrazován správně, tj. bez náhodného řetězce znaků.

8.4 Uživatelská dokumentace

Přepokladem pro úspěšné sestavení projektu je nainstalovaná Java verze 8. Pro svou funkčnost vyžaduje aplikace CRCE nainstalovaný a spuštěný databázový systém MongoDB. Po sestavení projektu a spuštění příkazem `mvn pax:run` je webová aplikace uživatelského webového rozhraní dostupná na url adrese `http://localhost:8080/crce-webui`. Řetězec „localhost“ udává adresu stroje, na kterém je aplikace uložiště CRCE spuštěna.

Po zadání výše uvedené adresy do prohlížeče je nejprve zobrazen přihlašovací dialog. Systém ověřování uživatele není v aplikaci nasazen, nicméně formulář ve třídě *LoginForm* v balíku *cz.zcu.kiv.crce.crce_webui_v2.webui* je připraven pro konkrétní implementaci, např. REST, LDAP apod. Stiskem tlačítka *guest* je možné se přihlásit jako hostující uživatel.

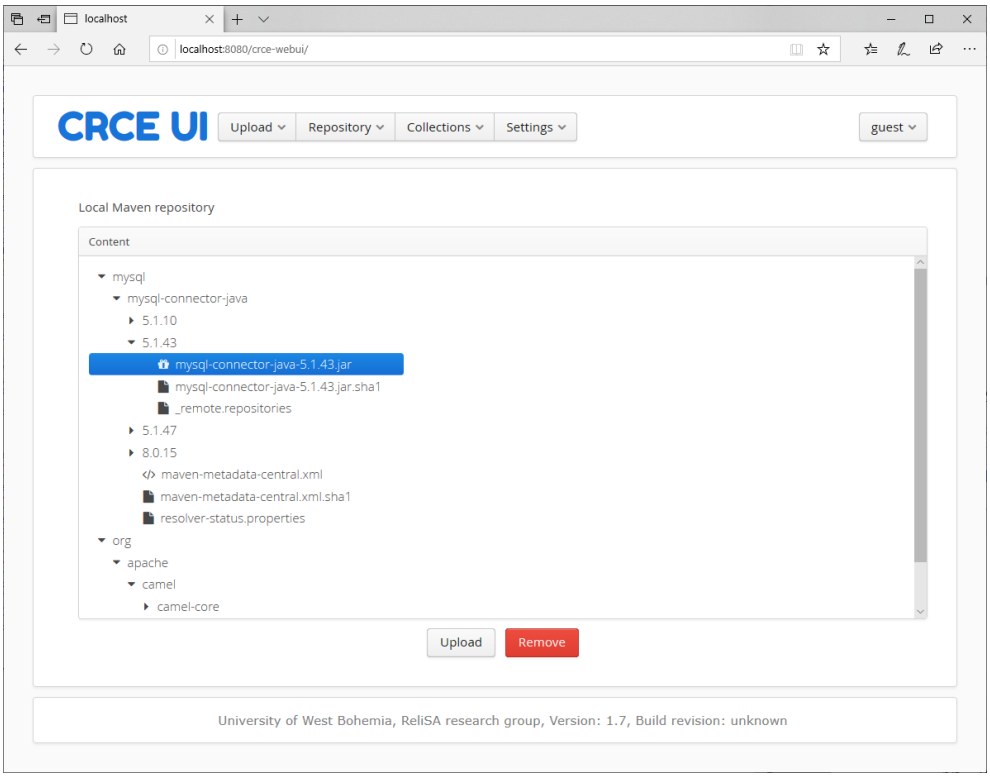
Na obrázku číslo 27 je zobrazena výchozí stránka po úspěšném přihlášení. V menu jsou k dispozici volby *Upload, Repository, Collection a Settings*. V pravém horním rohu je rozevírací menu s textem uživatelského jména přihlášeného uživatele. Zde je možné se z aplikace odhlásit.



Obrázek 27: Výchozí stránka webové aplikace

8.4.1 Externí zdroje dat

Podmenu *Upload* obsahuje tyto položky: *Local*, *Central*, *Defined* a *File/Url*. Na obrázku číslo 28 je ukázka stránky zobrazující lokální Maven úložiště.



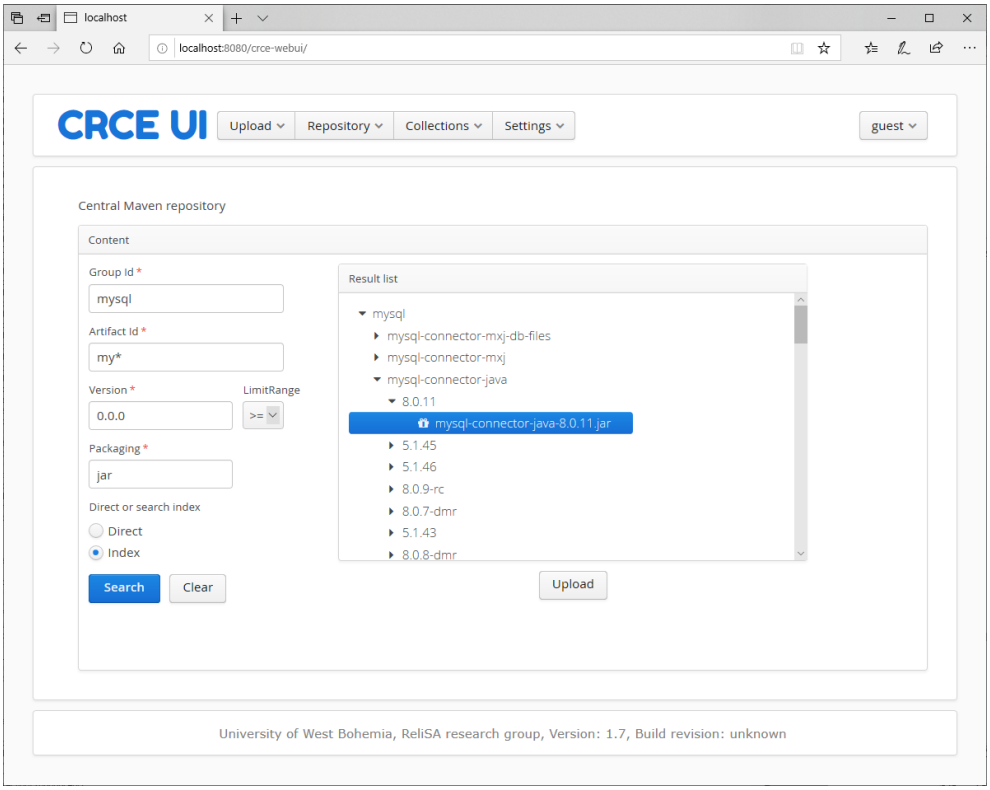
Obrázek 28: Formulář lokálního Maven repository

Ve výchozím nastavení je umožněn pouze režim čtení. Tlačítko *Remove* není dostupné. V případě požadavku na odebrání položek v lokálním Maven úložišti je nutné změnit nastavení na formuláři *Option repository* v podmenu *Settings*. Lokální Maven úložiště slouží pro knihovnu Aether a v určitých situacích nemusí být žádoucí, aby artefakt byl uložen v souborovém systému a zároveň v úložišti CRCE. Odebrání artefaktu nebo celé skupiny se provede vybráním položky a stiskem tlačítka *Remove*.

Ve formuláři zobrazení lokálního Maven repository je k dispozici možnost načtení zvoleného artefaktu do dočasného úložiště *Buffer* CRCE. Nahrání je provedeno po stisku tlačítka *Upload*. Je povoleno pouze načítání artefaktů. V případě, že je zvolena položka skupiny nebo verze a zároveň stisknuto tlačítko *Upload*, načtení není provedeno a uživateli je zobrazena informační hláška, že nebyl zvolen artefakt.

O úpěšném přenesení zvoleného artefaktu do CRCE aplikace rovněž informuje uživatele zobrazením informační hlášky.

Na obrázku číslo 29 je uvedena ukázka stránky formuláře pro získávání artefaktů z centrálního Maven úložiště.



Obrázek 29: Formulář centrálního Maven repository

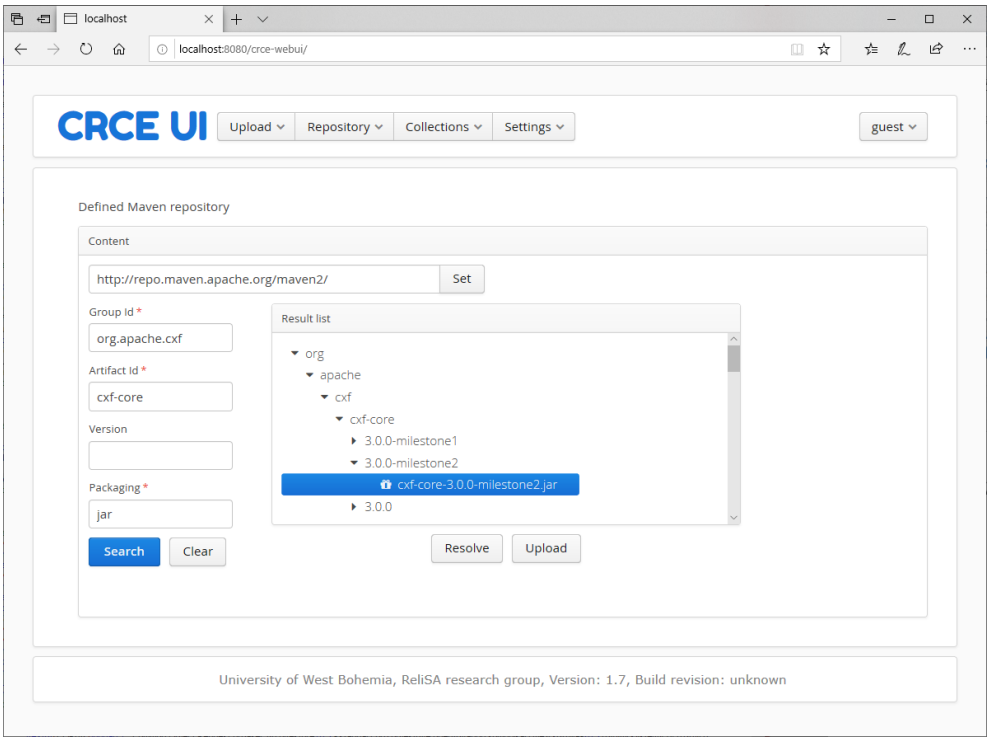
Vyhledání artefaktu je možné provést dvěma způsoby: přímé, s využitím indexu. Výběr způsobu vyhledávání uživatel provede označením příslušné volby v položce *Direct or search index*. Indexové vyhledávání je dostupné pouze v případě vytvořeného indexu nebo po ověření jeho aktuálnosti v menu *Settings* na formuláři *Check Maven index* stiskem tlačítka *Check index*.

Při přímém vyhledávání je nezbytné vyplnit přesné hodnoty identifikátorů artefaktu v centrálním Maven repository do textových polí formuláře. V případě indexového vyhledávání je možné část textu (mimo počátku) v položkách *Group id* a *Artifact id* nahradit znakem $*$. Při indexovém vyhledávání je dále k dispozici volba *LimitRange*, jejímž nastavení může uživatel definovat rozsah verzí požadovaných artefaktů.

Zvolená položka s artefaktem, nikoliv skupina nebo verze, je po stisku tlačítka *Upload* nahrána do dočasného úložiště *Buffer* CRCE. O úspěšném přenesení komponenty aplikace informuje uživatele zobrazením informační hlášky.

V případě povolení volby *Enable only group search*, která je ve výchozím nastavení zakázána, v menu *Settings* na formuláři *Option repository* může uživatel ve formuláři zadat pouze název skupiny a aplikace vypíše její kompletní obsah. Výpis je funkční pouze při nastaveném přepínači *Direct or search index* na volbu *Direct*. Některé skupiny či podskupiny mohou obsahovat velmi mnoho položek. Je proto doporučeno výše uvedenou funkci používat uvážlivě.

Na obrázku číslo 30 je zobrazena ukázka stránky formuláře vyhledávání komponent v Maven úložištích s využitím knihovny Aether.



Obrázek 30: Formulář vyhledávání artefaktů s využitím knihovny Aether

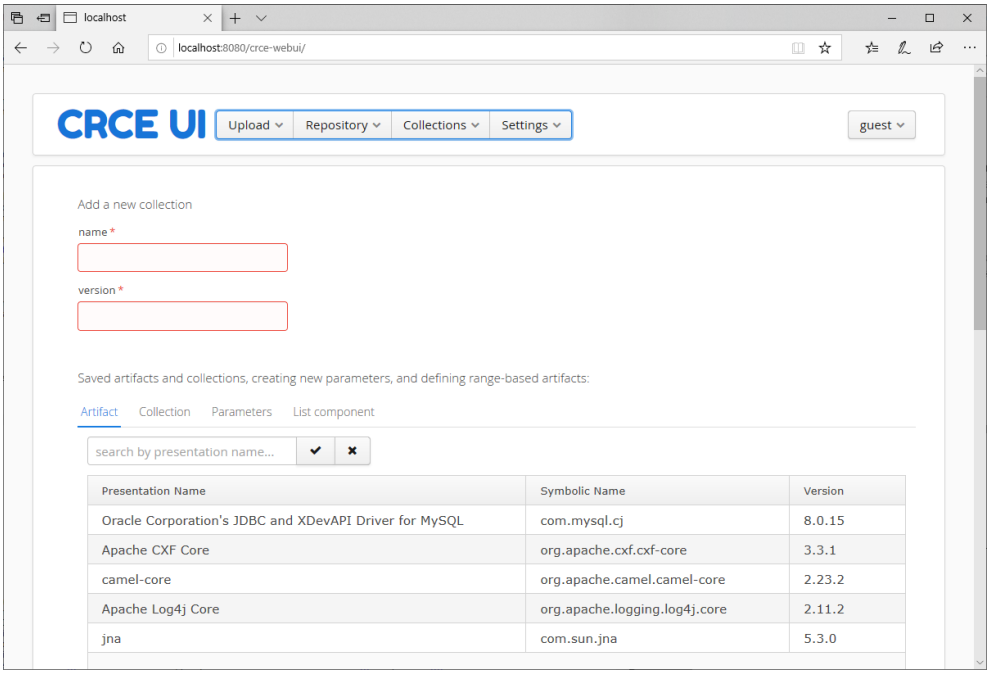
V textovém poli je možné uživatelsky definovat, respektive zadat, url odkaz na požadované Maven úložiště, které poskytuje rozhraní pro knihovnu Aether. Pro vyhledání požadovaného artefaktu je nezbytné zadat přesně identifikátory skupiny, názvu artefaktu v úložišti a typ souboru. Volitelně uživatel může zadat poža-

dovanou verzi. Pokud vynechá textové pole *Version*, aplikace vypíše seznam všech verzí dostupných v úložišti. Jak bylo uvedeno v kapitole 6.1.2, knihovna Aether v první fázi ověřuje, zda se požadovaný artefakt nenachází v lokálním Maven úložišti. Pokud ano, tento artefakt poskytne a není nutné jej přenášet po síti ze vzdáleného zdroje. Tlačítko *Resolve* slouží pouze k přenesení artefaktu ze vzdáleného úložiště do lokálního Maven repository, respektive ověření, že požadovaný artefakt je v lokálním Maven úložišti přítomen. Tlačítko *Upload* provede výše popsané kroky a zároveň načte artefakt do dočasného úložiště *Buffer* CRCE. O úspěšném načtení artefaktu do úložiště CRCE vypíše aplikace informační hlášku.

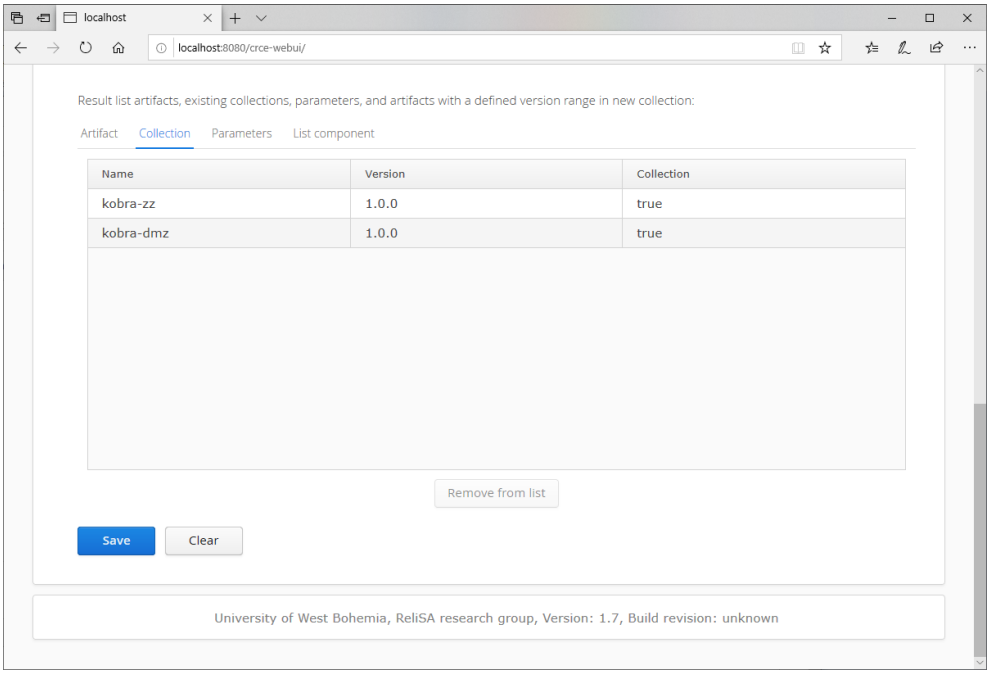
Formulář *File/url* slouží pro načítání artefaktů ze souborového systému na klientské straně aplikace nebo ze zadaného url odkazu. Jedná se o reimplementaci stávajícího způsobu načítání objektů pro úložiště CRCE.

8.4.2 Správa kolekcí artefaktů

V podmenu *Collection* jsou dostupné položky *New collection* a *List collection*. Na obrázcích číslo 31 a 32 je zobrazena stránka formuláře pro vytvoření nové kolekce.



Obrázek 31: Formulář vytváření nové kolekce (1. část)

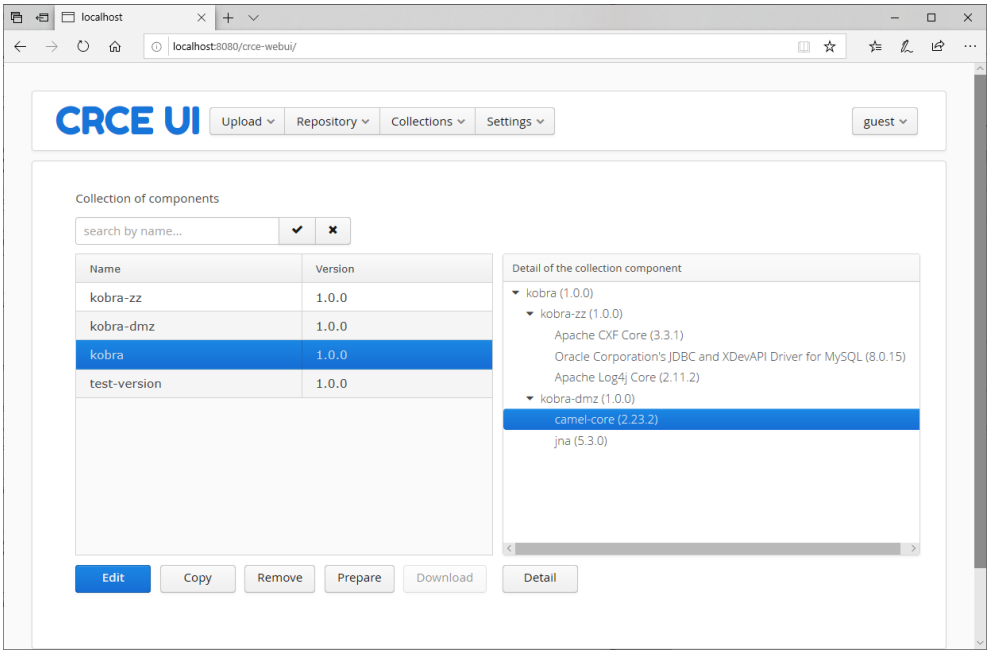


Obrázek 32: Formulář vytváření nové kolekce (2. část)

Množina artefaktů má automaticky přidělen jednoznačný identifikátor v databázi Mongo. Uživatel ve formuláři uvede název a verzi kolekce a dále vybere obsah množiny. Lze vybírat z uložených artefaktů ve *Store* CRCE (ve formuláři záložka *Artifact*), z již uložených kolekcí, (ve formuláři záložka *Collection*). Uživatel dále může volitelně zadat seznam parametrů kolekce ve formátu jméno a hodnota (ve formuláři záložka *Parameters*) a seznam požadovaných komponent s definovaným rozsahem verzí (ve formuláři záložka *List component*). Požadované komponenty se zadávají ve fromátu: symbolický název v úložišti CRCE a požadovaný rozsah verzí (min/max) oddělený čárkou. Příklad požadovaného zadání je uveden ve formuláři u popisu textového pole.

Uživatelem vybrané položky obsahu kolekce jsou vkládány do tabulek v odpovídajících záložkách, viz obrázek číslo 32. Kteroukoliv položku v těchto tabulkách může uživatel po jejím označení a stisknutím tlačítka *Remove from list* odebrat. Celý formulář je možné uvést do výchozího stavu stisknutím tlačítka *Clear*. Klinutím na tlačítko *Save* dojde k uložení nové kolekce a aplikace o úspěšném uložení zobrazí informační hlášku.

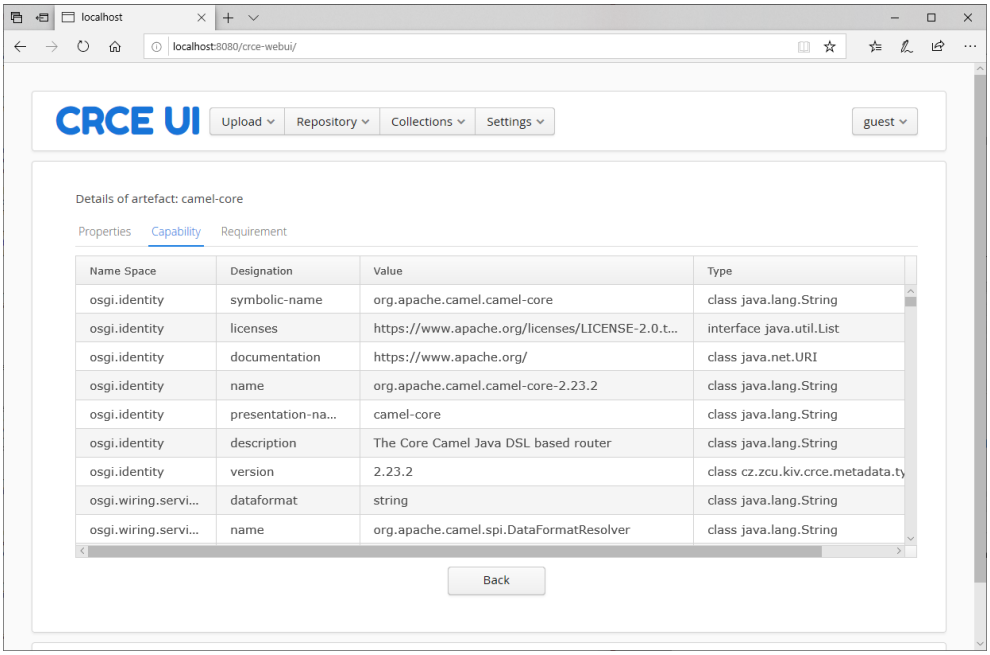
Na obrázku číslo 33 je uvedena stránka formuláře s výpisem seznamu všech uložených kolekcí.



Obrázek 33: Formulář s výpisem seznamu uložených kolekcí

Pokud uživatel vybere položku ze seznamu, na stránce dojde automaticky ke zobrazení struktury kolekce. Po výběru kterékoliv části struktury dochází k povolení tlačítka *Detail*. Kliknutím na toto tlačítko dochází k přesměrování na stránku s výpisem podrobností. V případě vybrané kolekce jsou zobrazeny záložky: *Parameters* a *List component*. Při zvolení artefaktu obsahuje formulář detailu výpis vlasností, požadavků a schopností dané komponenty. Příklad je uveden na obrázku číslo 34. Stiskem tlačítka *Back* je uživatel přesměrován na předchozí stránku, odkud byl na výpis podrobností přesměrován.

Na formuláři *List collection*, viz obrázek číslo 33, dojde stiskem tlačítka *Edit* k otevření stránky pro změnu zvolené kolekce. Struktura editačního formuláře je shodná s formulářem pro založení nové kolekce s tím rozdílem, že jsou vyplněny stávající položky. Volba tlačítka *Copy* způsobí vytvoření kopie zvolené kolekce. Uživatel vyplní číslo nové verze množiny artefaktů v textovém poli zobrazeného modálního okna. Úplné odstranění záznamu o kolekci je provedeno po stisku tlačítka *Remove*. V aplikaci není řešena situace po odebrání kolekce v případě, že tato tvoří podsložku jiné kolekce. Uživatel by si měl být vědom rizika mazání záznamu v evidenci.



Obrázek 34: Formulář s výpisem detailu artefaktu v kolekci

Webové uživatelské rozhraní obsahuje podporu pro stažení kolekcí artefaktů včetně jejich metadat v souboru archivu `zip`. Export je implementovám v modulu *crce-component-collection*. Jednotlivé fáze exportu jsou popsány v kapitole 8.2. Protože nelze provést přípravu `zip` souboru a stažení v jednom kroku, je tento proces rozdělen. Stiskem tlačítka *Prepare* je spuštěn export. Až po jeho dokončení je povoleno tlačítko *Download* a uživatel po jeho stisnutí zahájí proces stažení `zip` do souborového systému na klientském počítači.

Cesta umístění výchozí kořenové složky pro export je uvedena v textovém poli *Export path* ve formuláři *Option collection* v podmenu *Settings*. Aby nedocházelo při exportu k ovlivnění ostatních přihlášených uživatelů, je v této složce založena každému aktuálně přihlášenému uživateli dočasná podsložka s názvem dle identifikátoru spojení (session). Po odhlášení uživatele z aplikace dojde automatickému odstarnění této dočasné podsložky.

Uživatel může ovlivnit ve formuláři *Option collection*, vedle pomocné složky pro export, nastavení ještě dvou předvoleb:

- minimální, respektive maximální verzi artefaktu definováném jejím rozsahem,
- základní, respektive kompletní výpis metadat artefaktů do xml souboru.

9 Zhodnocení

Návrh řešení zadání vycházel zejména z průzkumu aktuálních informačních technologií v oblasti aplikací založených na modulární architektuře a aktuální implementace úložiště komponent podporují kontrolu kompatibility CRCE. Dále pak ze současných možností externích zdrojů artefaktů a způsobů jejich vyhledávání a získávání. Aktuální případy verzování a distribuce aplikací složených z jednotlivých částí (komponent) byla inspirací pro vytvoření podpory správy množin artefaktů uložených CRCE. V takovém případě je možné přistupovat k množině artefaktů jako k celku, tzn. komponentě složené z komponent.

Technologie, které připadaly v úvahu pro implementaci, bylo nezbytné ověřovat pro součinost se stávajícím řešením. Jednotlivé implementační verze byly v pravidelných intervalech distribuovány do verzovacího systému *github* na adresu `https://github.com/ReliSA/CRCE` na větev `rpesek`. Funkčnosti aktuální verze modulů *crce-external-repository-1.0*, *crce-component-collection-1.0*, *crce-webui-v2-1.7* odpovídají návrhu řešení uvedenému v kapitole 7. Funkce rozšíření, které jsou implementovány v modulech výše uvedených verzí, byly testovány úspěšně.

Návrh řešení byl přizpůsoben dalším možnostem rozvoje. Zejména realizace implementace funkčností získávání artefaktů z externích zdrojů a správa kolekcí v samostatných modulech. Toto řešení umožňuje import modulů do jiných svazků aplikace CRCE a využití jejich nabízené funkčnosti. Další možnosti rozšíření jsou:

- úprava služby modulu *crce-rest-v2* pro výpis metadat artefaktů uložených v kolekci,
- doplnění uživatelského rozhraní v modulu *crce-webui-v2* o podporu testů kompatibility,
- doplnění funkčností modulu *crce-component-collection* o podporu vzájemně kompatibilních verzí komponent.

Webová služba *Response metadataDetails();* modulu *crce-rest-v2* přebírá jako svůj parametr jednoznačný identifikátor artefaktu v úložišti *Store* CRCE. Jedná se o *_id* objektů kolekce *resources* v databázi MongoDB. Služba vrací jako svou návratovou hodnotu objekt *Response* s výpisem metadat. Kolekce *collection* v databázi MongoDB obsahují v položce *specificArtifact* pole identifikátorů artefaktů, které množina obsahuje. Úpravou výše uvedené služby je možné docílit kompletní výpis metadat všech objektů, které kolekce obsahuje.

Důležitou vlasností úložiště CRCE je vytváření a správa výsledků testů vzájem-né kompatibility uložených komponent. Návrhem na rozšíření je doplnění formulářů s podporou spouštění testů kompatibility a správa jejich úložiště do modulu grafického uživatelského prostředí *crce-webui-v2* .

Vylepšení modulu *crce-component-collection* spočívá v doplnění funkčností pro podporu vzájemné kompatibility verzí jednotlivých komponent, které kolekce obsahuje. V tomto případě by postačovalo pouze definovat názvy komponent a úložiště by poskytlo nejvyšší dostupné vzájemně kompatibilní verze.

10 Závěr

Přínosem práce je návrh a implementace nových funkčností pro úložiště komponent podporující kontrolu kompatibility CRCE. Jedná se o vyhledávání a získávání objektů z externích zdrojů dat a vytváření a správu verzí množin artefaktů, které úložiště obsahuje. Součástí je dále rozšíření nové webové aplikace grafického uživatelského prostředí na platformě frameworku Vaadin. Uživateli aplikace je v přívětivé a přehledné formě zpřístupněna správa životního cyklu komponent v úložišti CRCE a podpora výše uvedených nových funkcí.

Jsou implementovány funkce indexace centrálního úložiště Maven artefaktů, fulltextové prohledávání obsahu, stahování a ukládání komponent do CRCE. Repository objektů, které poskytují kompatibilní rozhraní pro knihovnu Aether, mohou být v aplikaci CRCE využívány. Díky zmíněné implementaci podpory knihovny Aether je vytvořeno lokální úložiště Maven artefaktů, které je primárně využíváno v situaci, kdy je v něm hledaný objekt dostupný. V tomto případě není nutný síťový přenos souborů ze vzdáleného úložiště. Měřitelný je údaj doby vytváření indexu centrálního Maven úložiště. Při rychlosti síťového spojení 30Mbits byla kompletní indexace vykonána v čase 60 minut. Centrální Maven úložiště obsahovalo 3.69 milionů archivů jar. Velikost indexu dosahuje 3,47 Gb.

Moderní trend vývoje aplikací využívá architekturu založenou na komponentách. Výhodou je znovupoužitelnost těchto částí a jednodušší implementace změn a oprava chyb. Nevýhodou programů složených z komponent je udržování vzájemné kompatibilty součástí při aktualizaci. Dále je nutné v určité formě uchovávat sestavu kompozitních aplikací a jejich správu verzí. Práce pojednává o současných možnostech složení, správy a distribuce množin artefaktů, na které je nahlíženo jako na celek. Z těchto příkladů vychází rozšíření aplikace CRCE o správu množin artefaktů, které jsou v úložišti dostupné. Tyto kompozitní komponety jsou díky rozšíření úložištěm CRCE spravovány identickým způsobem, jako v případě jejich složek, ze kterých jsou sestaveny. Množinu komponent je možné distribuovat včetně popisných dat (metadat) jejich součástí.

Úkol práce byl splněn. Nové funkčnosti byly několikanásobně testovány. Výsledkem je funkční a nasaditelná verze úložistě s rozšířenými funkcionalitami, které budou důležité pro další využívání CRCE v rámci výzkumných aktivit katedry informačních technologií.

Přehled zkratek

CBSE Component-based software engineering
- specifická oblast softwarového inženýrství

CRCE Component Repository supporting Compatibility Evaluation
- úložiště komponent podporující kontroly kompatibility

CORBA Common Object Request Broaker Architekture
- standard pro objektově orientovanou komunikaci mezi softwarovými systémy (v různých programovacích jazycích), operačními systémy a prostředím hardware

IoC Inversion of Control
- obrácení řízení vztahů, poskytování externích závislostí softwarové komponentě.

JAAS Java Authentication and Authorization Service
- ověření identity uživatele a určení, zda ověřený uživatel má práva k přístupu k určité části systému

JaCC Java Compatibility Checker
- nástroj pro ověřování vzájemnné kompatibility vyvinutý v rámci projektu výzkumné skupiny ReliSA

JMX Java Management Extensions
- technologie programovacího jazyka Java pro správu a monitorování aplikací, objektů systému a zařízení

OSGi Specifikace modulárního systému pro programovací jazyk Java

POJO Plain Old Java Object
- Java Bean s odstraněnými omezeními

POM Project Object Model
- (pom.xml) konfigurační soubor pro sestavení projektu využitím nástroje Maven

RBAC Role-based Access Control
- řízení přístupu založené na rolích

ReliSA	Reliable Software Architectures research group - výzkumná skupina Fakulty aplikovaných věd, Západočeské univerzity v Plzni
SSHd	Linux OpenSSH proces - sleduje příchozí požadavky využitím SSH protokolu
.NET	zastřešující název pro soubor technologií společnosti Microsoft
NPM	registr software, komponentová knihovna

Literatura

Tištěné články a knihy:

[Con98] CONRADI, Reidar. *Version Models for Software Configuration Management.* 1998. ISSN 232-282.

[GH05] GOSPODNETIĆ, Otis a Erik HATCHER. *Lucene in action*. Greenwich, CT: Manning, c2005. ISBN 9781932394283.

[Hal11] HALL, Richard S. *OSGi in action: creating modular applications in Java.* Greenwich [Conn.]: Manning, c2011. ISBN 978-193-3988-917.

[Kuc11] KUČERA, Jiří. *Úložiště komponent podporující kontroly kompatibility.* Plzeň, 2011. Diplomová práce. ZČU.

[RBB11] REUSSNER, Ralf, Steffen BECKER, Erik BURGER, et al. *The Palladio Component Model.* Karlsruhe Institute of Technology, Germany, 2011. ISSN 2190-4782.

Elektronické články:

[Aet] *Aether - Eclipsepedia* [online]. Eclipse Foundation, 2014 [cit. 2019-05-10]. Dostupné z: `https://wiki.eclipse.org/Aether`

[Aetd] *GitHub - eclipse/aether-demo: Aether project repository (aether-demo)* [online]. Eclipse Foundation, 2014 [cit. 2019-05-10]. Dostupné z: `https://github.com/eclipse/aether-demo`

[Ari] *Apache Aries - Index* [online]. Apache Software Foundation, 2019 [cit. 2019-04-29]. Dostupné z: `https://aries.apache.org/`

[BJ15] BRADA, Přemek a Kamil JEŽEK. *Repository and meta-data design for efficient component consistency verification*. Science of Computer Programming [online]. 2015, 349-365 [cit. 2018-10-09]. Dostupné z: `https://www.sciencedirect.com/science/article/pii/S0167642314002925?via%3Dihub`

[CSS11] CRNKOVIC, Ivica, Judith STAFFORD a Clemens SZYPERSKI. *Software Components beyond Programming: From Routines to Services*. IEEE Software [online]. 2011, 28(3), 22-26 [cit. 2019-01-21]. DOI: 10.1109/MS.2011.62. ISSN 0740-7459. Dostupné z: `http://ieeexplore.ieee.org/document/5756294/`

[Fab] *Fabric Guide - Red Hat Customer Portal* [online]. Red Hat, 2017 [cit. 2019-05-10]. Dostupné z: https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html/fabric_guide/index

[Fel] *Apache Felix* [online]. The Apache Software Foundation, 2019 [cit. 2019-03-05]. Dostupné z: http://felix.apache.org/

[Fus] *Red Hat Developer|Red Hat Fuse* [online]. Red Hat, 2019 [cit. 2019-03-09]. Dostupné z: https://developers.redhat.com/products/fuse

[Ind] *GitHub - apache/maven-indexer: Apache Maven Indexer* [online]. Apache Software Foundation, 2017 [cit. 2019-05-10]. Dostupné z: https://github.com/apache/maven-indexer

[Kar] *Apache Karaf* [online]. Apache Software Foundation, 2019 [cit. 2019-03-07].Dostupné z:https://karaf.apache.org

[Karc] *Apache Karaf Container 4.x - Documentation* [online]. The Apache Software Foundation, 2019 [cit. 2019-05-10]. Dostupné z: http://karaf.apache.org/manual/latest/

[Log] *LogicBlaze FUSE* [online]. Brenda M. Michelson, 2006 [cit. 2019-05-10]. Dostupné z: http://www.elementallinks.com/2006/04/03/logicblaze-fuse-open-source-soa-platform/

[Luc] *Apache Lucene* [online]. The Apache Software Foundation, 2019 [cit. 2019-04-15]. Dostupné z: http://lucene.apache.org

[Mic] *What are microservice?* [online]. Red Hat, 2019 [cit. 2019-05-09]. Dostupné z: https://www.redhat.com/en/topics/microservices/what-are-microservices

[Nex] *Nexus Repository* [online]. 8161 Maple Lawn Blvd: Sonatype Headquarters, 2008 [cit. 2019-03-11]. Dostupné z: https://www.sonatype.com/nexus-repository-sonatype

[Osg] *The Dynamic Module System for Java* [online]. 2400 Camino Ramon, Suite 375 San Ramon, CA 94583 USA: OSGi Alliance, 2019 [cit. 2019-02-28]. Dostupné z: https://www.osgi.org

[Vaa] *Dokumentace Vaadin 7* [online]. Ruukinkatu 2-4, 20540 Turku, Finland [cit. 2019-04-17]. Dostupné z: https://vaadin.com/docs/v7

[Zip] *Zipping a Unzipping in Java* [online]. Baeldung, 2018 [cit. 2019-04-24]. Dostupné z: https://www.baeldung.com/java-compress-and-uncompress

Seznam obrázků