VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a Informatiky

DIPLOMOVÁ PRÁCE

2013 Bc. Tomáš Vantuch

VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a Informatiky

Katedra informatiky

Modulární Inteligentní Úložiště Obrázků

Modular Intelligent Image Storage

2013 Bc. Tomáš Vantuch

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Vantuch**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Modulární inteligentní úložiště obrázků
Modular Intelligent Image Storage

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat dále rozšiřitelné úložiště kolekcí obrázků, které bude umožňovat jejich předzpracování, anotaci (metadata), vyhledávání na základě anotace nebo obsahu obrázku nebo jejich doplněných rysů získaných různými metodami a zpřístupnění v požadované kvalitě a formátu.

Mimo vlastní implementaci základního vyhledávání bude definováno i API rozhraní pro zpřístupnění obrázků, jejich předzpracování a vyhledávání v kolekci.

Úložiště bude zpřístupňovat kolekce obrázků více způsoby (souborový systém, webové služby, HTTP, apod.), v základní podobě bude definováno API pro přístup ke kolekci a individuálním obrázkům a implementovány alespoň 2 metody přístupu k úložišti.

1. Prozkoumejte možnosti uložení rozsáhlých kolekcí obrázků pro vyhledávání na základě podobnosti a metadat.
2. Definujte strukturu úložiště, podporované formáty a možné mechanismy pro práci s úložištěm.
3. Definujte API pro zpřístupnění úložiště externím aplikacím (vyhledávání, získání kolekce a uložení dat) a API pro tvorbu dalších modulů (předzpracování dat, vyhledávání v metadatech).
4. Navrhněte a implementujte úložiště a základní metody předzpracování (barvy, rozměry, apod.)
5. Otestujte funkčnost a efektivitu úložiště (overhead přenesených dat, rychlost vyhledávání a zpřístupnění kolekce)

Seznam doporučené odborné literatury:

[1] William K. Pratt - Digital image processing 4th edition, Pixelsoft Inc., Los Altos, California, 2007
[2] Michael S. Lew - Principles of Visual Information Retrieval (Advances in Computer Vision and Pattern Recognition), Springer, 2011
[3] Corinne Jörgensen - Image Retrieval: Theory and Research. Scarecrow Press, 2003.
[4] Andrew L. Rubinger, Bill Burke - Enterprise JavaBeans 3.1. O'Reilly Media; Sixth Edition, 2010.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Pavel Moravec, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Na tomto mieste by som rád poďakoval vedúcemu diplomovej práce Ing. Pavlovi Moravcovi Phd., ktorý mi napomáhal k lepšiemu pochopeniu požiadavkou, čím sa celý projekt jednoduchšie vyprofiloval.

Prohlášení

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostrave dňa 28. apríla 2013 .....

Abstrakt

Zjednodušenie a zjednotenie prístupu k určitému problému v informatike je cieľom akéhokoľvek rámcového riešenia, inak povedané frameworku. V rámci diplomovej práce som sa snažil využitím dostupných metód a spôsobov nasadenia, pokryť požiadavky zadania za účelom zisku čo najjednoduchšie rozšíriteľného riešenia. Uvedomenie si pojmu digitálneho spracovania obrazu a postavenie zadania na tomto požiadavku, tvorilo profilovanie konkrétneho využitia celého projektu. Diplomová práca postupne prechádza jednotlivými etapami vývoja od objasnenia problematiky oblasti, cez výber technológií vhodných pre riešenie, ďalej popisuje jednotlivé naimplementované celky a ich dôležité detaily. Súčasťou je takisto návod na nasadenie a jednoduché testovanie s popísanými výsledkami.

Kľúčové slová

Digitálne spracovanie obrazu, modularita, Java, framework, rozšírenie

Abstract

Simplifying and unifying approach to some problems in computer science is the goal of any solution framework. In the thesis I tried using available methods and ways of usage to cover the requirements of the assignment to profit for as easy as possible extensible solutions. Realising of the concept of digital image processing and state of assignment on this requirement formed a specific usage of the entire project. Diploma thesis is gradually going through various stages of development from clarification of the issues through selection of appropriate technologies to the solution, further describes the implemented units and the important details. It also includes instructions for simple deployment and testing of with described results.

Keywords

Digital image processing, modularity, Java, framework, extension

Zoznam skratiek a pojmov

DIP : digital image processing, označenie pre digitálne spracovanie obrazu

OSGi : Open Services Gateway initiative, súbor špecifikácií pre OSGi od OSGi Alliance

Java EE: Java 2 Enterprise

Hibernate : Java framework pre prácu s databázov

Maven : nástroj na riadenie procesu zostavovania projektov

Vaadin : webový Java framework

XML : všeobecný značkovací jazyk od W3C

JMS : Java Message Service, správami orientované Java API

API : rozhranie pre programovanie aplikácií

BLOB : binary large object, typ pre binárne dáta v databázy

Obsah

Úvod

Diplomová práca, modulární inteligentní úložište obrázku, by mala slúžiť na nájdenie spôsobu ako riešiť jednoducho opísateľný problém, zjednotenie prístupu k tvorbe algoritmov pre digitálne spracovanie obrazu. Riešením by mal byť modulárne rozšíriteľný systém, práve o dané metódy s webovým rozhraním a prístupom pomocou webových servýs pre ďalšie aplikácie.

V prvej kapitole sú popísané úvodné očakávania od výsledného systému. Takisto uvádza typy budúcich užívateľov, čím sa snaží popísať reálne potreby na systém.

Druhá kapitola uvádza v krátkosti problematiku digitálneho spracovania obrazu a pojednáva o metódach ako diskrétna kosínusová transformácia, prevod farebného obrazu do odtieňov šedi, čím uvádza teóriu k implementovaným prídavným modulom.

Tretia Kapitola detailne rozoberá hlavné a odvodené požiadavky na systém, vychádzajúce ako zo zadania, tak aj s konzultácií s vedúcim diplomovej práce.

Vo štvrtej kapitole je obsiahnutá jemná analýza možností ukladania dát, modulárneho prístupu a riešenia v rámci profesionálnych systémoch a popis architektúry výsledného systému.

V rámci piatej kapitoly je zameranie na popis vybraných technológií a frameworkov, ktoré boli využité na implementovanie systému. Popísané sú hlavne také kľúčové prvky, ktoré systém reálne využíva.

Šiesta kapitola obsahuje implementačný popis jednotlivých modulov. Postupne rozoberá moduly oboch behových prostredí, popisuje ich hlavné prvky a funkcionalitu.

V siedmej kapitole je pojednanie o implementačných detailoch, ktoré sú pre tento systém kľúčové. Obsahuje podrobnosti o tvorbe zásuvných modulov, o procese ich nasadenia a komunikácie s nimi a o možnostiach API pre dané moduly.

Ôsma kapitola je zameraná na nasadenenie a štart systému na aplikačnom servery. Od základných nastavení vnesených do modulu a server až po postupné nasadenie jednotlivých modulov popisuje návod ako celý systém spustiť.

Deviata kapitola je spisom testovania nad súborom dát. Vybrané kolekcie obrázkov použité na otestovanie základnej a pridanej funkcionality systému s podrobnejším popisom výsledkov.

1 Úvodné očakávania

V úvodnej časti by som rád rozobral očakávania tejto práce. Cieľom bude navrhnúť a implementovať platformu splňujúcu kritériá, ktoré vyplývajú zo zadania diplomovej práce, alebo boli ďalej špecifikované počas osobných konzultácií s vedúcim tejto práce.

Hlavný dôvod je návrh riešenia a vytvorenie akéhosi základu pre systém, s ktorým bude možné čo najviac zjednotiť vývoj a testovanie metód pre spracovanie digitálneho obrazu. Z terajšieho pohľadu si môžeme predstaviť množstvo malých, samostatne pracujúcich programov postavených na rôznych platformách, s rôznymi typmi vstupov a výstupov a s rôznymi možnosťami užívateľského prístupu, tudíž značná nejednotnosť komplikujúca celkový prístup k veci. A na druhej strane vidina jednoduchého rozhrania, ktoré bude umožňovať vlastné rozšírenia o konkrétne metódy a prístup viacerými spôsobmi pre užívateľa, čo by malo viesť k celkovej jednotnosti a jednoduchosti v rámci danej práce.

Pre lepšiu predstavu budem definovať budúceho užívateľa systému. Bude sa jednať o technicky zdatnejšieho človeka, ktorého záujmom bude testovanie metód na digitálne spracovanie obrazu, ďalej to bude takisto vývojár, ktorý dané metódy bude implementovať ale nakoniec by mohol byť prítomný aj človek poskytujúci odbornejší dohľad, ktorý by pomohol kvalifikovanejšie hodnotiť ako výkonnosť metód, tak aj výsledky testov. Pre prvý prípad užívateľa, ktorý si chce vyskúšať najrôznejšie metódy spracovania obrazu, je prvoradý jednoduchý prístup k systému a užívateľsky jednoduchá dostupnosť samotných metód pre spustenie. Pre tohto užívateľa musí byť jasné ako sa dostane k dátam, ako obrázkom, kolekciám a pridaným dátam, ako spustí jednotlivé metódy nad týmito dátami a samozrejme ako získa výsledky z daných metód. Pre ďalší typ užívateľa, vývojára samotných metód, bude podstatné veľmi jasne definované API, ktoré mu pre jeho naimplementované metódy zabezpečí prístup k dátam a spustenie v behovom prostredí. Nakoniec typ užívateľa ako vedúci práce bude potrebovať jednotlivé testy pre overenie zopakovať, nahliadnuť kód metódy a takisto minimálne ručne overiť výsledky. Takže možnosť perzistentých výsledkov a verejná dostupnosť metód budú preňho kľúčové.

V rámci toho, že nie je nijak definovaný stávajúci prístup ani súbor aplikácií riešiacich podobné scenáre, budem pristupovať k vývoju ako k samostatnému systému bez závislosti na akomkoľvek predošlom riešením. Riešením by mal byť samostatne bežiaci informačný systém s vlastnou databázou, ktorý kompletne pokryje požiadavky zadania.

Technické požiadavky na daný systém sú celkom jasné. Mal by bežať na aplikačnom servery, užívateľské prostredie by malo byť prístupné cez webový prehliadač, s tým súvisí podpora všetkých dostupných prehliadačov, takisto by malo byť dostupné verejné API a jednoduchá možnosť rozšírenia a pre samotný vývoj prídavných modulov by mala byť umožnená podpora v najbežnejších vývojárskych nástrojoch v rámci zvoleného jazyka.

2 Spracovanie obrazu

V rámci tohto projektu nie je cieľom rozobrať tému digitálneho obrazu a jeho spracovania, ale vytvoriť aspoň miernu predstavu problému, na ktorý by mal byť v cieli projekt pripravený.
Obrazom v digitálnej forme sa samozrejme myslí dátová reprezentácia pomocou obrazových bodov uložených v matici alebo schopnosť ich generovania pomocou spojitej funkcie. Jednotlivé obrazové body sú následne reprezentované, v rámci monochromatického obrazu hodnotou a v prípade farebného obrazu vektorom hodnôt, kde jednotlivé hodnoty určujú množstvo farebnej zložky zobrazovaného spektra.
Digitálne spracovanie obrazu v informatike sa potom označuje proces, v ktorom pre akýkoľvek obraz v digitálnej forme na vstupe získame v závislosti na užitej metóde určitý výstup. V rámci rozsiahlosti témy je k dispozícii nespočetné množstvo typov algoritmov, ktoré na svoju realizáciu vyžadujú rôzne vstupy, majú rozdielnu výpočtovú náročnosť a samozrejme rôzne rozsiahle množstvo výstupov. Pre ilustráciu sa jedná napríklad o metódy predspracovania za účelom odstránenia nežiaducich prvkov (šum, redundancie, ...) alebo naopak zviditeľnenia rôznych vlastností, ďalej to môžu byť rôzne analytické algoritmy na detekciu hrán, tvarov, farieb, rôzne kompresné algoritmy, atď.
V najširšej abstrakcii by systém mal poskytovať API na vytvorenie, beh a uloženie výsledkov akéhokoľvek pridaného modulu alebo súboru modulov, ktoré budú spolupracovať za účelom vykonávania algoritmu digitálneho spracovania obrazu.
Pre príklad bude nutné implementovať nejaké z metód ako zásuvné moduly, preto v ďalších podkapitolách uvediem trocha teórie k nim.

2.1. DCT

Informácie pre túto kapitolu som získal zo zdoja [6].
Diskrétna kosínusová transformácia (DCT) je typ stratovej kompresie, ktorá sa skladá z viacerých krokov:

Transformácia farieb z RGB (prípadne CMYK) do YC_bC_r.

Jedná sa o bezstratový typ transformácie, ktorého podstatou je oddeliť farebné zložky od zložky vyjadrujúcu jas pixelu. Samotný prevod z RGB do Y (zložka jasu) a C_b,C_r čo sú rozdielové hodnoty farieb pixelov (oproti Y) je nasledovný:

$Y = 0{,}299\ R + 0{,}587\ G + 0{,}114\ B$
$Cb = -\ 0{,}1687\ R - 0{,}3313\ G + 0{,}5\ B + 128$
$Cr = 0{,}5\ R - 0{,}4187\ G - 0{,}0813\ B + 128$

Pre spätný prechod:

$R = Y + 1.402\ (Cr-128)$
$G = Y - 0.34414\ (Cb-128) - 0.71414\ (Cr-128)$
$B = Y + 1.772\ (Cb-128)$

Redukcia (podvzorkovanie) farebných zložiek.

Ľudské oko je vybavené väčším počtom receptorov, ktoré vnímajú svetlosť ako samotnú farbu. Preto zmena farebných zložiek nemusí byť tak razantne vnímaná ako zmena jasu. Redukcia farebných zložiek spočíva v spracovaní $C_{b,}$ a C_r zložiek a to jednoduchým spriemerovaním, buďto susedných dvoch alebo štyroch pixelov. Pri podvzorkovaní dvoch pixelov dostaneme namiesto 6 bytov len 4, čo je úspora až 66% a pri podvzorkovaní štyroch pixelov (tvoriacich štvorec) získame namiesto 12 bytov len 6, čo je úspora až 50%. Je treba si ale uvedomiť, že sa jedná o stratový prevod, pretože ďalej nemáme informácie o farbe jednotlivých pixelov.

Diskrétna kosínusová transformácia

Tento samotný krok je robený nad zložkou Y každého pixelu a to prepočtom pomocou kvantizačnej funkcie, čím získame koreláciu medzi susednými (prípadne vzdialenejšími) pixelmi – tzv. medzipixelová redundancia.

DCT je rozlišované v jednorozmernej úrovni pre určité signálny v systéme jednorozmerného poľa ako 1D DCT :

$$t(k) = c(k) + \sum_{n=0}^{N-1} \mathrm{s(n)} \cos \frac{\pi(2n+1)k}{2N}$$

Z tohto vzťahu bázová funkcia pre 1D DCT:

$$\cos \frac{\pi(2n+1)k}{2N}$$

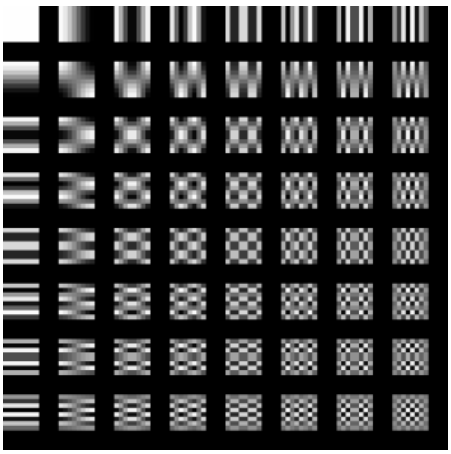
Pre dvojrozmerný raster (obrázky) hovoríme o dvojrozmernej diskrétnej kosínovej transformácii:

$$t(i,j) = c(i,j) \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \mathrm{s(m,n)} \cos \frac{\pi(2m+1)i}{2N} \cos \frac{\pi(2n+1)j}{2N}$$

Bázová funkcia pre 2D DCT má tvar:

$$\cos \frac{\pi(2m+1)i}{2N} \cos \frac{\pi(2n+1)j}{2N}$$

Typický priebeh 2D DCT pre N=8



Obrázok 1: priebehy dvojrozmerného DCT pre N=8

Kvantizácia DCT koeficientov

Kvantizačná tabuľka je matica koeficientov nastavená (zväčša) podľa štruktúry obrázku a použitej kvality kompresie. Pre vyzdvihnutie vysokých frekvencií (hrany, šum) sa v kvantizačnej tabuľke vyskytujú koeficienty s vysokými hodnotami (v konkrétnom, úzkom zastúpení), a pre nižšie frekvencie (priestor s vyrovnanými farebnými zložkami) sú kvantizačné koeficienty nižšie. Pomocou vhodne zvolenej kvantizačnej tabuľky môžeme presnejšie vygenerovať vodorovné alebo zvislé hrany, či iné ostré prekrytia farieb. Pri vyššej stratovosti (nevhodne zvolená kvantizačná tabuľka, pre zmenšenie veľkosti výsledného súboru) získame kvantizáciou namiesto hrán šum. Takže výsledné hodnoty po DCT podelíme hodnotami v kvantizačnej tabuľke (vždy jeden s daným, ktorý mu prislúcha), v tomto prípade sa jedná o celočíselné delenie (nastáva ďalší stratový jav).

Po prevedení kvantizácie ďalej spracujeme DC zložky, keď od DC zložky v každom bloku odčítame túto zložku z predošlého bloku. Keďže sa jedná o odčítanie stejnosmernej zložky, výsledkom bude rad nízkych čísel (núl).

V ďalšom kroku, v ktorom preskladáme maticu 8x8 pomocou cik-cak sekvencie na lineárny tvar, dostaneme väčšinu núl za seba takže ich môžeme veľmi efektívnym spôsobom zakódovať.

Huffmannovo kódovanie

Je špeciálny typ kódovania, ktorý nijak s DCT nesúvisí ale pre jeho pomerne vysokú efektivitu a jednoduchú implementáciu sa častou využíva.

Má princíp zápisu dát pomocou kódových slov systémom, v ktorom majú najčastejšie sa vyskytujúce slová najkratšiu bitovú dĺžku. Pre jednotlivé zložky obrazu JPEG (DC a AC) sú použité kódovacie tabuľky, pomocou ktorých sa s využitím vlastností Huffmannova kódovania šetrí ďalšia kapacita.

Vo vkladanom čísle rozlíšime o akú zložku ide (DC/AC), následne podľa jeho veľkosti určíme diferenčnú kategóriu a nakoniec podľa toho či sa jedná o luminanciu (Y) alebo chrominanciu (Cb - Cr) vyberieme kódové slovo.
AC zložky pre vysoké frekvencie majú po kvantizácii väčšinou nulové hodnoty a cik-cak zápisom sa nachádzajú na konci bloku 8x8, preto ich vyjadrovať nemusíme (nie je pre ne ani diferenčná kategória) ale blok ukončíme značkou EOB.

2.2. Grayscale

Následujúca kapitola čerpá z [11].
Jedná sa o jednoduchý prevod celého farebného obrázku na čiernobiely. Každý obrazový bod reprezentovaný troma farebnými zložkami je prevedený na jedinú hodnotu reprezentujúcu odtieň šedi. Pre príklad uvediem tri algoritmy tohto prevodu:

- Prvá metóda ráta priemer najvyššej a najnižšej hodnoty farebných zložiek: (max(R, G, B) + min(R, G, B)) / 2.
- Druhá ráta jednoducho priemer všetkých farebných zložiek pre pixel: (R + G + B) / 3.
- A posledná uvedená metóda je sofistikovanejšia, pretože vychádza zo znalosti, že ľudské oko je citlivejšie na zelenú zložku spektra viac ako na zostávajúce dve. Nepočíta jednoduchý priemer ale zrátava podiely jednotlivých zložiek, ktoré vnikli vážením o koeficienty na základe spomínanej citlivosti. Výsledný pixel získa hodnotu zo vzťahu: 0.21 R + 0.71 G + 0.07 B.

2.3. MatrixMarket

Informácie som prebral zo zdroja [8]
Tento formát je len uložením matice do súboru a to takým spôsobom, že v každom riadku je vždy na začiatku riadkový index, za ním stĺpcový index a následne hodnota pixelu, ktorá mu odpovedá. V princípe pokiaľ reprezentujeme kolekciu obrázkov jednou maticou, musí práve jeden riadok reprezentovať obrázok kolekcie, ktorý je doň preukladaný pixel po pixeli postupne. Na začiatku má len pridané ako komentáre, ktorý riadok odpovedá ktorému obrázku.

3 Hlavné požiadavky

Keďže v zadaní sú aj nepriamo spomenuté požiadavky vzťahujúce sa na architektúru, mal by som ich brať rovnako na vedomie ako ostatné. Jedná sa o zmienku o modularite, rozšíriteľnosť a pomocou definovaného API prístup aplikáciám tretích strán. Prispôsobiť architektúru tejto požiadavke je celkom jasné a v ďalších kapitolách spomeniem jeho výhody a postupne jeho realizáciu.
Jeden z najhlavnejších bodov analýzy je definovanie hlavných funkčných aspektov:

- **Ukladanie obrázkov a kolekcií:** bude prístupný spôsob odoslania kolekcie v zbalenom formáte na server, následne aplikácia daný súbor rozbalí, obrázok po obrázku uloží

a pridané informácie (názov, autor, kolekcia,...) vloží do databáze. Tento scenár by mal byť prístupný cez viacero rozhraní (web, webová služba, ...)

- **Prístup k obrázkom a kolekciám:** jednotlivé obrázky a kolekcie budú môcť byť vybrané k náhľadu a k stiahnutiu, v prípade kolekcie sa znova stiahne zbalený súbor. Takisto v rámci tohto požiadavku sa počíta s prístupom viacerými spôsobmi.
- **Ručné anotovanie a doplnenie informácií k obrázkom a kolekciám:** jednotlivo k obrázkom ale aj k celým kolekciám by malo byť možné pridať niekoľko popisných anotácií (tagov), ktoré by boli následne uložené do databáze pre ich ďalšie využitie
- **Vyhľadávanie na základe pridaných informácií:** na základe pridaných anotácií, ale aj názvov či autora by malo byť možné nájsť obrázok, či kolekciu v jednoduchom vyhľadávači, ktorý bude súčasťou užívateľského prostredia aplikácie
- **Predspracovanie jednotlivých kolekcií:** jednotlivé kolekcie by mali byť prístupné k rôznym typom predspracovania a jeden z týchto typov by mal byť ako názorný príklad implementovaný
- **Vonkajší prístup k predspracovaným dátam:** takéto dáta či už uložené v databáze alebo v súborovom systéme ako dátové súbory by mali byť dostupné k exportovaniu a voľnému náhľadu
- **Integrácia nových modulov:** jednotlivé metódy, o ktoré by mal byť systém ďalej rozšíriteľný, by mali byť implementované formou zásuvných modulov a čo najjednoduchšie integrovateľné do celého systému.
- **Vnútorný prístup k predspracovaným dátam:** jednotlivé záznamy databáze alebo súbory na disku by mali byť dostupné aj pre zásuvné moduly jednotným zdieľaným prístupom a objektovo relačným mapovaním nad databázou.
- **Časové hodnotenie behu procedúr:** jednotlivé prídavné moduly búdu mať záznamy o ich aktuálnom stave v databáze a takisto sa bude zaznamenávať ich priemerný čas (napríklad dĺžka operácie s jedným obrázkom)

3.1. Odvodené požiadavky

Z nasledujúcich hlavných funkčných požiadavkou môžem odvodiť vedľajšie funkčné požiadavky:

- **Zabezpečenie a prihlasovanie pre užívateľov:** jednotlivý užívatelia budú musieť pre prihlásenie do systému uviesť svoj login a heslo a to nielen v rámci webovej aplikácie ale aj pre používanie webových služieb. Následne im bude vygenerované sessionId (kľúč) na základe, ktorého im bude vytvorený súbor pre logové záznamy.
- **Možnosť definovania pridaných vstupov pre zásuvné moduly:** pretože nie som schopný definovať všetky potrebné vstupy pre najrôznejšie metódy, ktorými bude systém rozšírený, mal by byť dostupný spôsob akým vniesť pridaný parameter do spustenia zásuvného modulu.

- **Možnosť pridania entít do databáze pre zásuvné moduly:** modul pre ukladanie výsledkov svojho behu bude môcť si automatizovane vytvoriť databázovú tabuľku, ktorá bude k dispozícii pre celý systém.
- **Možnosť overenia prítomnosti modulu počas behu iného modulu:** v rámci znovu použiteľnosti kódu bude vhodné aby jednotlivé zásuvné moduly mohli k svojmu behu využívať existujúceho kódu iného zásuvného modulu. V takom prípade bude potrebná možnosť overenia existencie modulu v behovom prostredí.
- **Prístup k výsledkom po ukončení behu modulu:** zásuvný modul pokiaľ uloží výsledky ako súbor alebo zložku súborov, mal by mať k dispozícii API, pomocou ktorého daný objekt zaregistruje do databáze. Následne daný súbor bude dostupný pre iné moduly a na základe vygenerovaného URL bude možné pre užívateľa dané súbory stiahnuť.
- **Možnosť nasadenia knižníc tretích strán pre zásuvné moduly**: popri možnosti nasadenia zásuvného modulu, mala by byť dostupná aj možnosť pridania knižníc, ktoré modul na svoj beh využíva. Pomocou pridaného postupu by malo byť možné z akejkoľvek knižnice v danom jazyku urobiť ďalší zásuvný modul pre tento systém.

Cielená architektúra softwaru by mala využívať čo najväčšie množstvo dostupných návrhových vzorov, aby výsledný kód bol čo najprehľadnejší a čo najjednoduchšie rozšíriteľný a v rámci aplikácie ako celku by malo ísť o modulárny návrh s rôzne potrebnými prípadne nahraditeľnými modulmi.

Na projekt nie je stanovená konkrétna požiadavka čo sa týka programovacieho jazyka, či frameworkov. Táto voľba bola plne prenechaná mne. Zvolil som programovací jazyk Java a s ním niekoľko kľúčových riešení, ktoré vždy pokryjú určitú časť zadania a tým odpadne možnosť riešiť úlohy, ktoré už dávno vyriešené sú.

4 Analýza a návrh

4.1. Modulárny prístup

V kapitole uvádzam informácie zo zdroja [10].
Krátkym úvodom do modularity uvediem členenie a hlavné dôvody tohto prístupu, čím ozrejmím orientáciu vývoja tohto projektu. Modulárny prístup vo vývoji robustnejších aplikácií sa v dnešnej dobe ukazuje ako kľúčový a to bez závislosti na platforme na akej produkt beží, jazyku, v ktorom je písaný či odvetví, pre ktoré je aplikácia určená. Modularita vo všeobecnosti zabezpečuje granulárny pohľad na zložité, ťažko ovládateľné celky. Tým získavame množstvo menších, jednoduchších a myšlienkovo ucelených jednotiek (modulov).
V rámci softwarového vývoja sa modulárny prístup delí na dva základné pohľady.
Logický pohľad, v ktorom kód aplikácie je štrukturálne rozdelený. Začína to samotnou triedou, predstavujúcu určitú časť procesu, pokračuje skupinou tried a končí u balíka (v Jave package, v C# namespace), čo je už určitým spôsobom logicky ucelená jednotka. V rámci daného balíka je nutné mať rozhranie alebo API, cez ktoré sme schopný pre všetky prístupy jednotne poskytovať funkcionalitu daného balíka.
Ďalším typom pohľadu je fyzický pohľad. Ten sa skladá z dvoch dôležitých prvkov a to správne zbalený súbor, ktorý obsahuje skompilovaný kód modulu na spustenie a pridané popisné súbory s meta dátami pre jeho definíciu. Pomocou metadat musíme byť schopný definovať závislosti modulu, rozhranie komunikácie s prostredím a množstvo ďalších popisných vecí pre jeho identifikáciu. Potom ďalším dôležitým prvkom bude logicky samotné behové prostredie schopné daný súbor prijať a spravovať na základe pridaných meta informácií.
Samotné moduly spolu v rámci aplikácie musia mať možnosť komunikovať, či už priamym volaním alebo správami a zdieľať dáta. Na základe týchto parametrov je definovaných niekoľko typov previazaní modulov:

- **Content coupling:** modul pristupuje do vnútra iného modulu, mení jeho atribúty a využíva jeho funkcie
- **Common coupling:** moduly zdieľajú len spoločné globálne dáta
- **External coupling:** moduly využívajú definovaný dátový formát, protokol komunikácie alebo vonkajšie rozhranie
- **Control coupling:** jeden modul priamo ovláda druhý modul
- **Stamp coupling:** moduly zdieľajú určitú dátovú štruktúru a každý z nich pracuje s nejakou časťou
- **Data coupling:** moduly zdieľajú len atomické dáta a neexistuje nič ako štruktúrované dáta alebo formát
- **Message coupling:** previazanie modulov pomocou správ s presne definovaným formátom
- **No coupling:** moduly nie sú previazané, spolu nekomunikujú

Popri previazaní modulov sa definujú aj typy súdržnosti medzi modulmi :

- **Náhodná (Coincidental cohesion):** jednotlivé časti spolu nemusia úplne súvisieť, len sú v kope (napríklad súbor tried Utils)

- **Logická:** moduly majú podobné funkcie, odlišujú sa len v kontexte využitia (napríklad skupiny ovládačov prídavných zariadení)
- **Časová (Temporal cohesion):** komponenty vytvárané v určitú dobu alebo periódu vývoja
- **Procedurálna:** moduly, ktorých funkcie sú volané po sebe a tvoria jeden ucelený proces
- **Komunikačné:** komponenty pracujúce na rovnakých dátach
- **Sekvenčná:** výstupy z jednej časti bývajú vstupmi do druhej časti
- **Funkčná:** jednotlivé časti spolu kooperujú v rámci komplexnejšieho procesu (parsovanie XML)

Motiváciou k modulárnemu vývoju software by mala byť samotný pohľad na jednotlivé moduly, ktorý vnáša do vývoja jednoduchosť, nastaviteľnosť, testovateľnosť ale takisto znovu použiteľnosť a variabilitu pri tvorbe rozsiahleho softwarového diela.
Zo serverovo orientovaných modulárnych aplikácií mám osobnú skúsenosť so CMS systémom Magento, ktorý je napísaný v jazyku PHP, podporuje možnosť aplikovania modulu do bežiaceho systému. Na druhú stranu vývoj každého modulu je sprevádzaný rozsiahlymi XML a databázovými konfiguráciami pre samotnú členitosť systému.

Ďalším modulárnym systémom, s ktorým mám osobnú skúsenosť je ERP systém Openbravo, ktorý beží na platforme Java. Jeho jednoduchosť je znovu priamo úmerná členitosti a rozsiahlosti oblasti, ktorú rieši, avšak tento systém nie je schopný použiť rozšírenie v rámci bežiacej aplikácie. Pre túto potrebu je nutné server vypnúť, modul nakopírovať alebo stiahnuť z repozitára, celý systém zostaviť ANT skriptom, v rámci ktorého prebehne registrovanie a validácia modulu a až potom je možné celý systém nasadiť na server. Je to sled operácií, počas ktorých žiaden ďalší užívateľ nemá do systému prístup.

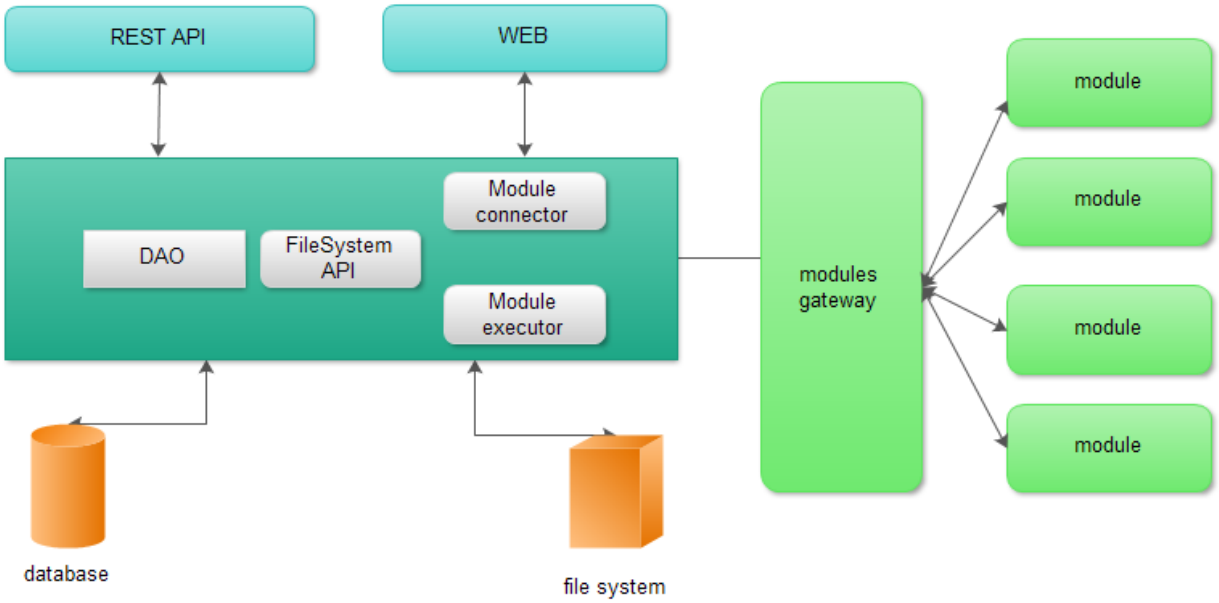
Modulárne systémy, vo väčšine prípadov síce dodržujú minimálne previazanie komponent a maximálne znovu použitie kódu, avšak buďto nie úplne poskytujú spomínanú jednoduchosť alebo rozšírenie v reálnom čase.

V rámci samotného aplikovania modulu je nutné objasniť potreby, vychádzajúce zo zadania. Systém, v ktorom bude bežať algoritmus na digitálne spracovanie obrazu nad väčšou kolekciou, by mal byť stále plne dostupný v rámci všetkých operácií pre určitý počet užívateľov. Takisto pokiaľ bude potrebné rozšíriť systém o zásuvný modul, nebude vhodné systém kvôli tejto operácii reštartovať z dôvodu behu práve spomínaného procesu. Mal by byť rozšíriteľný za behu a v reálnom čase.

Ďalšou vecou vychádzajúcou zo zadania je potreba jednoduchého vytvorenia modulu, bez nadbytočných nastavení, keďže sa v rámci digitálneho spracovania obrazu bude vo väčšine prípadov jednať z hľadiska vstupov a výstupov o generický podobné operácie.

4.2. Návrh architektúry

V rámci návrhu architektúry by som systém vnútorne rozdelil na základe požiadavkou do určitých pomyseľných častí, vždy v závislosti čo daná časť bude mať za úlohu riešiť a ako sa k nej bude pristupovať.



Obrázok 2: návrh architektúry MIIS

- Tmavozelená zóna reprezentuje časti tvoriace akési jadro systému, ktoré bude zabezpečovať základný beh systému, prístup k dátam a prácu so zásuvnými modulmi.
- Oranžové prvky ako databáza a súborový systém budú pochopiteľne zabezpečovať perzistenciu dát pre systém.
- Svetlozelené prvky budú samotné zásuvné moduly a ich jednotné rozhranie pomocou, ktorého bude k ním systém pristupovať.
- Svetlomodré prvky nakoniec ako WEB a REST servisy budú plniť funkciu rozhrania ši už pre užívateľa alebo pre aplikácie tretích strán.

Prídavný modul by mal byť klasický zostavený projekt obsahujúci triedy so spustiteľným kódom, ktoré budú implementovať systémom definované rozhrania. Práve pomocou týchto rozhraní bude systém schopný takýto modul nájsť, poslať doň vstupné dáta a spustiť ho. Takisto bude vhodné pokiaľ určitú časť systémovej funkcionality poskytne danému modulu pomocou svojho API.

4.3. Spôsob uloženia obrázkov

Pokiaľ sa rieši akým spôsobom budú ukladané dátové súbory, v tomto prípade obrázky systémom väčších úložísk, je potrebné zaoberať sa viacerými otázkami. Nakoľko bezpečné je riešenie, akú má výkonnosť, stabilitu, možnosť obnovenia, prípadne ďalšieho rozširovania.

Serverové prostredie pre enterprise aplikácie poskytuje niekoľko možností ako ukladať obrázky, s tým že každé z riešení má rôzne výhody či nevýhody oproti iným.

Jednou z možností je obrázky ukladať priamo do databázy vo formáte BLOB. Tento dátový typ je označením pre bližšie nešpecifikované binárne dáta. Pre databázu nie je možné ich nejakým spôsobom interpretovať pre možnosť porovnávania, formátovania do iných tvarov či konvertovania do inej kódovej stránky. Celý BLOB je interpretovaný až samotnou aplikáciou, ktorá má informáciu o tom ako s dátami pracovať. Takéto databázové riešenie má výhodu v možnosti vytvorenia clusteru za účelom replikácie, pre vyššiu výkonnosť, či back-up. Jednou ale z nevýhod je zvyšujúce sa nároky na prenos, pri vysokom počte súborov dopytovaných z databáze v rovnakom čase.
Výhody tohto riešenia:

+ Je to veľmi jednoduché riešenie a funguje výborne.
+ Všetko na jednom mieste. Dáta ako obsah súborov, či informácie o ňom idú z jedného bezpečného zdroja.
+ Kvalitná správa back-upu či portovania na iný systém.
+ Vysoká flexibilita, zmena štruktúry kolekcie nemusí presúvať dáta, jedná sa len o zmenu referencií.

Nevýhody:

− Môže sa jednať o finančne nákladnejšie riešenie.
− Pre nemožnosť interpretácie BLOB-u môže byť ukladanie ne-relačných dát do relačnej štruktúry považované za „nečisté“ riešenie.
− Upravovať časti súborov v databázy môže byť o mnoho náročnejšie ako len pracovať so súbormi na disku.

Tu sa naskytá riešenie druhé, kde databázu využijeme na uchovanie meta-informácií (názov, adresa v súborovom systéme, autor, veľkosť, tagy, ...) o obrázkoch a samotné súbory uložíme do adresárovej štruktúry. Databáza potom aplikácii pošle len informácie o obrázku a k samotnému súboru sa aplikácia dostane jednoduchým prehľadávaním súborového systému servera.

Výhody súborového systému:

+ Jednoduchšia škálovateľnosť.
+ Jednoduchá manipulácia a práca s obsahom súborov.
+ Nižšia cena.

Nevýhody súborového systému:

− Nutnosť udržiavania informácii o súboroch v ďalšom úložisku (najčastejšie databáza).
− Pri zmenách názvu či umiestnenia je potrebné aktualizovať aj meta-informácie.
− Pre tento projekt som ako primárnu zvolil druhú variantu s tým, že prvá možnosť ale neostala úplne zavrhnutá. Samotnú databázu tohto projektu je možné rozšíriť pomocou

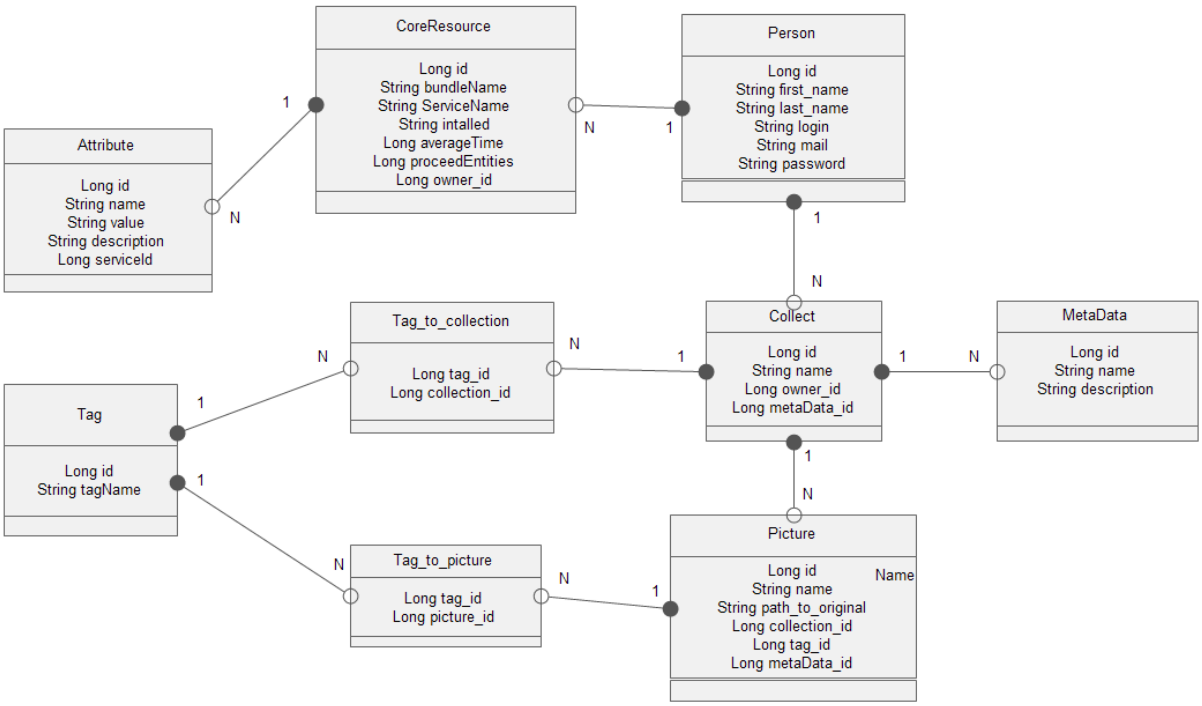
inštalačných skriptov pridaných modulov o nové tabuľky, čo užívateľovi dáva možnosť výberu, ako jeho modul bude získavať dáta.

Obrázky ukladám do súborového systému priamo na servery, spolu s ich variáciami ako napríklad komprimované verzie, verzie po stratovej kompresii, verzie v stupňoch šedi či iné predspracované dáta. Nakoniec o každom obrázku existuje záznam v databázy, ktorý obsahuje jeho identifikátor, názov, autora a adresu súboru a pridané metadáta ako anotácie či príslušnosť ku rovnako anotovanej kolekcii. Vyhľadávanie obrázkov na základe meta informácii je plne ponechané na databázu, a naopak vyhľadávanie podľa vnútorného obsahu budú riešiť pridané moduly systému.

4.4. Popis dátovej vrstvy

V tejto časti by som vykreslil základný dátový model aplikácie. Základný preto, pretože v rámci rozširovania o zásuvné moduly bude vhodné, aby jednotlivými zásuvnými modulmi bolo možné ho rozšíriť podľa potrieb užívateľa. V čistej inštalácii systém poskytuje správu nasledujúcich entít:

- **Person:** správa užívateľov, obsahuje meno, priezvisko, pre prihlasovanie login a heslo, mail a identifikátor (ID)
- **Picture**: základné informácie o obrázku identifikátor (ID), názov, cesta k originálu, cesta k náhľadu (využívané modulom resize_image budle), identifikátor kolekcie ako cudzí kľúč a takisto referencia na tabuľku metadata.
- **Collect:** tabuľka pre ukladanie informácii o kolekciách. V základe obsahuje indentifikátor, názov a referencie na tabuľku Person (označuje autora kolekcie) a Metadata, kde sú uložené pridružené informácie ku kolekcii.
- **Tag:** jednoduchá tabuľka pre ukladanie tagov jednotlivých obrázkov. Obsahuje len identifikátor a TagName, názov.
- **TagToCollection**, **TagToPicture:** toto sú väzobné tabuľky zjavne medzi entitami Picture a Tag a medzi Collect a Tag, z dôvodu možnosti priradenia viacerých tagov na kolekciu alebo obrázok.
- **MetaData:** spomínaná tabuľka s pridruženými informáciami pre obrázok či kolekciu obsahuje v základe len identifikátor, názov, meno vlastníka popisovaného prvku a popis.
- **CoreResource:** je entita, ktorá drží záznami o všetkých moduloch, s ktorými daná instancia systému prišla do styku. Jednotlivé moduly sa počas inštalácie registrujú do tejto tabuľky a vypĺňajú záznamy o názve bundlu, názve servisy, referencia na autora (Person). Ďalej obsahuje záznam, ktorý indikuje či modul je v systéme aktívny, po odinštalácii sa samozrejme prepne na false a takisto obsahuje údaj o priemernom behu modulu (averageTime) na jeden obrázok a takisto počet spracovaných obrázkov, ktorý sa používa na prepočítavanie váženého priemeru.
- **Attribute:** jednou z posledných entít v základnej inštalácii je entita pre pridané atribúty prídavným modulom. Jednotlivé moduly počas svojej inštalácie registrujú tieto atribúty, kde parametrami sú identifikátor, popis, názov, servisa, ktorá ho využíva a referencia na pridaný bundle.



Obrázok 3: triedny UML diagram doménovej vrstvy

Popri entitnom modeli na ukladanie dát bezpochyby slúži aj súborový systém. Jednotlivé adresy na súbory sú uložené v databázy s tým, že na samotný súborový systém servera sa siaha pomocou pripraveného API alebo vlastnou cestou z prídavných modulov. Jednotlivá štruktúra adresárov je vetvená do stromovej štruktúry podľa kolekcií, takže je snaha aby jednotlivé obrázky neboli vedené v jednom adresári, čím sa výši rýchlosť nájdenia súborov.

5 Výber technológií

5.1. Možné technológie

V rámci možnosti učiniť výber jazyka a technológii podľa svojich preferencií, venoval som nejaký čas samotnej skladbe projektu a to najmä za účelom, čo najširšieho pokrytia požiadavkou už hotovými riešeniami, či už pomocou celých frameworkov alebo knižníc tretích strán.

- **Objektovo: relačného mapovania pre riešenie perzistentnej vrstvy:** v tejto oblasti sa naskýta široké pole riešení. V Jave je dostupný Hibernate, JDBC, JPA, GORM, iBATIS, ďalej v rámci platformy .NET je k dispozícii Entity framework a LINQ a takisto PHP poskytuje napríklad Cake alebo Zend. Na nejaké porovnávania či výkonnostné testy nie je dôvod, pretože každé z riešení pri vhodnom nastavení vie poskytnúť dostatočné benefity. Hlavne bude záležať aby riešenie sedelo do celého kontextu a aby zbytočne nekomplikovalo celkovú prácu.
- **Webové frameworky na tvorbu webovej aplikácie:** znova kladiem dôraz na jednoduchosť a produktivitu a pokiaľ je to možné, čo najjednoduchšiu integráciu do aplikácie ako celku. Opäť začnem čo je na výber z platformy Java: ide o Spring Web MVC, Wicket, Vaadin, JSF, Grails alebo užitie klasických servletov. V rámci PHP sa situácia veľmi nemení oproti predchádzajúcemu bodu a na platforme .NET sa naskýta využiť technológiu ASP.NET.
- **Riešenie pre tvorbu webových služieb:** v tomto bode ešte nezáleží, či ide o webové služby typu SOAP alebo REST. Vo všeobecnosti Java dáva možnosť tvorby tejto veci v Spring REST, Java EE, JAX-RS a ďalšie. V .NET ide o známy prístup pomocou WCF a napríklad Zend framework v jazyku PHP takisto poskytuje možnosť písať SOAP, či REST služby.

Keď to však zhrniem, základom bude vybrať si, ktorý jazyk bude jednoduchší, dostatočne výkonný, bude aspoň nejakým spôsobom riešiť modulárny prístup a bude značne škálovateľný v rámci výkonnosti.

V nasledujúcom kroku som začal zvažovať tvorbu modulárnej aplikácie so základom vo forme Enterprise aplikácie, ktorá by zabezpečila jednoduchosť v rámci samotného vytvorenia ale takisto správy na aplikačnom server. V tomto bode som si nebol úplne istý možnosťami zo strany .NETu, preto som sa ďalej venoval už len výberu frameworkov na jazyku Java.

Poslednou voľbou bolo rozhodnúť sa medzi Spring ako enterprise platformou, Spring DM (dynamic modules) pre komunikáciu s dynamickým prostredím OSGi a na druhej strane čisté OSGi a Java EE. V tomto bode už záležalo minimálne či to vo finále bude Spring alebo Java EE, vo výsledku oba pokrývajú značnú časť požiadavkou, oba sú dostupné zdarma pre vývoj a takisto poskytujú značnú podporu formou návodov a rád a ich výkonnostné testy v tom čase boli takmer totožné. Takže v úvode som si vyskúšal obe varianty na vytvorenie štruktúr hlavných modulov. Posledným faktorom, ktorý rozhodol bola podpora zo strany vývojárskych nástrojov, pretože Eclipse STS (Springsource Toll Suite) proti Netbeans neposkytovali úplnú jednoduchosť a bezchybnosť pri tvorbe a správe projektov.

5.2. Použité technológie

5.2.1. Server GlassFish

V tejto kapitole popisujem informácie zo zdroja [4].
Projekt ako celok beží na aplikačnom server GlassFish. Tento server je vyvíjaný momentálne spoločnosťou Oracle pre platformu Java EE. GlassFish sa radí medzi open-source projekty spadajúce pod licenciu GPL a CDDL. Používaná verzia a typ servera je len referenčná implementácia, to znamená, že nie je priamo určená pre prevádzku aplikácií, ale slúži skôr ako ukážka implementácia nových prvkov v aktuálnej špecifikácii platformy Java EE. Takisto existuje komerčná verzia, ktorá má označenie Oracle GlassFish Server. Tieto servery sa ale funkčne takmer nelíšia, hlavný rozdiel je však v dostupnej podpore a automatickom sťahovaní aktualizácií.
Architektúra aplikačného serveru je založená na modulárnom jadre vychádzajúce z OSGi (Open Service Gateway initiative) frameworku. GlassFish väčšinou využíva implementácie Apache Felix, ale môže bežať aj na implementáciách Equinox, či Knoperfish OSGi. OSGi framework umožňuje, že aplikácie aj komponenty je možné vzdialene inštalovať, štartovať, ukončovať, aktualizovať či odinštalovať bez potreby reštartu servera. Tento projekt je však napísaný pre distribúciu Apache Felix, takže odporúčam pred štartom skontrolovať v adresári \glassfish\osgi, či sa tam práve táto nachádza.
Ďalšie podstatné adresáre v rámci tohto serveru:

- **Adresár files:** miestom v rámci doménového adresára \domains\<názov domény>\, kde sú ukladané všetky súbory posielané alebo generované serverom (kolekcie, obrázky, predspracované dáta,…).
- **Adresár bundles:** priestor pre osgi moduly, nachádza sa v \domains\<názov domény>\autodeploy\. Pokiaľ modul, ktorý inštalujeme na server nie je možné z nejakého dôvodu poslať, je možnosť ho ručne uložiť práve sem, systém by mal inštaláciu automaticky spustiť.
- **Adresár osgi-cache:** už podľa názvu zrejme sem osgi ukladá informácie o bežiacich moduloch z dôvodu zrýchlenia ich využitia. Jedná sa chash pamäť, ktorú v problémových situáciách je vhodné mazať.
- **Adresár logs:** na toto miesto sú ukladané log súbory generované aplikáciou, každý nesie názov odpovedajúci aktuálnemu užívateľovmu sessionId. Jednotlivo je k nim prístup aj cez REST rozhranie.
- **Adresár lib:** miesto pre uloženie knižníc využívaných Java EE kontajnerom. Pokiaľ chýba nejaká závislosť je vhodné práve sem uložiť požadovanú knižnicu. Napriek tomu závislosti v rámci OSGi kontajneru sú ukladané do adresára bundles, s tým že musí ísť o osgi-fikované jary (popísané v neskorších kapitolách).

V rámci verzie servera 3.1.2_b23 je nutné upozorniť na chybu, ktorá spôsobuje, že nie je možné využívať upload cez RIA frameworky ako Vaadin alebo RichFaces. Problém nastával pri získavaní viacdielnych dát z formuláru na servlet neanotovaný ako @MultipartConfig alebo nedefinovaný

vo web.xml na danú operáciu. V rámci Servlet 3.0 špecifikácie to nebolo jasne definované, tak niektoré RIA frameworky to nesprávne implementovali. V danom čase nestačil update ale dodatočne bol upravený jeden z modulov servera web-core.jar, ktorý prikladám v prílohe. V prípade problémov je ho nutné nahradiť za stávajúci v adresári glassfish/modules.

5.2.2. JavaDB

Údaje o JavaDB sú získané zo zdroja [2].

Apache Derby alebo inak Java DB je súbor knižníc, ktoré môžu byť nasadené do akéhokoľvek Java programu v rámci potreby databáze, či už pre desktopovú aplikáciu, webovú službu alebo webovú aplikáciu. Môže takisto byť zbalená ako súčasť vašej aplikácie, bez nutnosti dodatočnej inštalácie či nastavovania. V rámci tohto riešenia ide o embedded móde, kde všetky databázové súbory ukladá priamo do JAR súborov. V rámci tohto malého riešenia, ktoré zaberá asi len dva megabyty priestoru, poskytuje okrem riešenia väčšiny SQL štandardov aj množstvo databázových funkcií ako aplikovanie triggrov, riešenie súbehu či zálohy.

Ako samostatne bežiaci server, Derby poskytuje online zálohy, replikáciu medzi databázami, XA transakcie, LDAP autentikáciu a SSL/TLS kryptované pripojenie. Štandardná distibúcia poskytuje webové rozhranie vo forme Java servletov, pre nainštalovanie Derby servís na daný aplikačný server. Takisto samotnú instanciu Derby databáze môžeme kontrolovať pomocou jej API priamo z našej aplikácie alebo využiť JMX (Java Management Extension) pre vzdialený monitoring a správu.

Java DB je jednoducho značka pre verziu Apache Derby, ktorá je tvorená a podporovaná Sun Microsystems (dnes Oracle). Zhodne takisto existujú pod licenciou Apache Software License a ako open source sú voľné k akémukoľvek využitiu. Najčastejšie sa však využíva pri prototypových aplikáciách práve pre jej jednoduché nasadenie a skoro nulovú réžiu.

Preto tento typ databáze využívam aj ja v úvode tohto projektu. So správnym ovládačom pre Hibernate nie je jediný problém pri jej využívaní a vďaka prítomnosti Hibernate bude v budúcnosti veľmi jednoduché tento typ databáze zmeniť za sofistikovanejšie riešenie.

5.2.3. OSGi

Modularita v Jave

Java ako jazyk samotný neposkytuje veľmi pohodlný prístup k riešeniu modularity. V skutočnosti návrh modulárnych aplikácií v čistej Jave nesie so sebou určité problémy.

V rámci logického pohľadu na modularitu ide o enkapsuláciu, nízko levelová kontrola viditeľnosti kódu. Pretože public prvky sú volateľné odkiaľkoľvek rovnako ako protected v prípade použitia dedičnosti, programátor musí prispôsobiť logickú štruktúru balíkov tak, aby nezverejnil zo svojho API príliš, ale zároveň aby ho mohol efektne rozšíriť. Logická štruktúra projektu vo výsledku nemusí presne odpovedať myšlienke, pretože býva často zabalená rôznymi bezpečnostnými prvkami.

V ďalšom rade zdrojom problémom v rámci fyzického pohľadu je samotná classpath. Tento prvok sa totiž vôbec samostatne nestará o správu verzií a defaultne vracia prvú nájdenú verziu

požadovanej triedy. V inom prípade nekontrolované mixovanie závislostí na balíkoch vyvoláva „jar hell“, kedy v niektorých častiach projektu sa zbytočne nachádzajú obsahovo duplicitné balíky, prípadne ešte horšie, v rozličných verziách. A nakoniec ani samotný deploy Java aplikácie nemusí byť úplne jednoduchý. Práve skutočná práca nastáva v momente, keď máme aplikáciu úspešne nasadenú a potrebujeme ďalej upravovať a znovu nasadzovať niektoré komponenty aplikácie. Máme možnosť neustále vykonávať re-deploy celej aplikácie, prípadne využívať pluginové rozširovanie, pokiaľ naša aplikácia využíva služieb určitého class loaderu, čo môže byť četným zdrojom chýb. Možnosti, ktoré poskytuje Java EE, vytváranie EJB balíkov, EAR komponent, WAR webových balíkov alebo servisné archívy SAR(JBoss Service Archive) môže pripadať ako celkovo zložité a nejednotné riešenie.

Následné informácie sú získané zo zdroja [9].
OSGi je framework zabezpečujúci modularitu v Jave. OSGi (pôvodne ako Open Services Gateway initiative) predstavujúci štandard pod licenciou OSGi Alliance, je v dnešnej dobe považovaný za jeden z najvyspelejších frameworkov riešiacich túto problematiku v Jave. Na implementáciách OSGi ako napríklad Apache Felix, Eqiunox či Knopflerfish sú dnes postavené tie najmodernejšie aplikačné servery (JBoss, GlassFish, Websphere,...) či vývojárske nástroje (Eclipse). V širšom ponímaní sa jedná o robustné riešenie, ktoré poskytuje celkovú správu modulárnej architektúry projektu od vytvárania jednotlivých modulov, cez ich nasadenie, správu životného cyklu modulu ale aj ich vzájomné interakcie, previazania, a vonkajší prístup ale takisto spoluprácu s inými frameworkami.
V ponímaní tohto riešenia sa stáva modularita skutočne architektúrou, kde jednotlivé moduly môžu na sebe nezávisle pracovať, dynamicky sa meniť a celkové previazanie modulov je skutočne variabilné a ovládateľné.
Samotné behové prostredie, ktoré tento framework poskytuje pre nasadenie modulov OSGi Alliance nepopisuje ako štandardný kontajner, podobný aplikačným serverom. Nazýva ho priamo ako collaborative environment, v preklade kolaboratívne prostredie. Zdôvodňuje to práve nadštandardnou správou modulov, možnosťou zdieľania kódu medzi modulmi a ďalších mnohých možností. Pre jednoduchosť ale budem toto prostredie uvádzať ďalej ako kontajner.
Základná architektúra tohto riešenia sa skladá z viacerých prvkov:

- **Bundle:** zbalený jar súbor s pridaným popisným súborom MANIFEST, pripravený na deploy do OSGi kontajnera.
- **Servisa:** základný prvok publish-find-bind modelu, kde servisy sú exportované/importované inými modulmi.
- **Registrátor servís:** API umožňujúce siahať do OSGi kontextu a manipulovať so servisami.
- **Module vrstva:** definícia ako pridaný bundle importuje/ exportuje kód.
- **Security:** Vrstva zabezpečujúca bezpečnostné aspekty.
- **Execution Environment:** Určuje aké triedy a metódy sú v danom čase dostupné na špecifickej platforme.

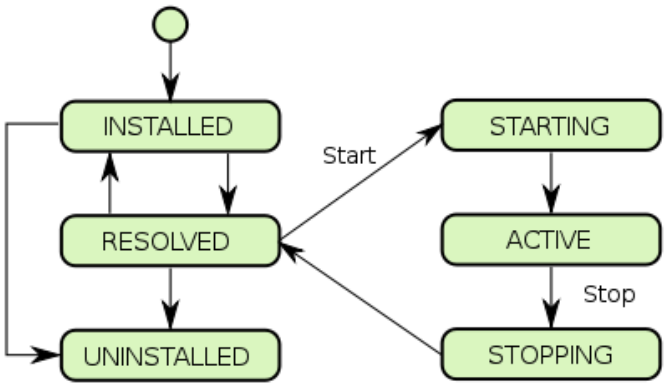
Základným stavebným kameňom OSGi modularity je Bundle. V klasickom pohľade je to jednoduchý jar súbor, obsahujúce triedy s určitou logikou, ktoré chceme registrovať do kontajnera ako servisy, alebo máme len set interfacov na export do prostredia. V oboch prípadoch náš jar súbor musí obsahovať MANIFEST.MF súbor kde je detailný popis celého bundlu (jar s popisným súborom môžeme už považovať za bundle).
Pre vytvorenie popisného súboru dnes existujú pluginy pre maven, ktoré dokážu počas zostavovania jaru, presný súbor vygenerovať. Pre správnosť je vhodné to však skontrolovať, prípadne dodatočne upraviť. Súbor by mal obsahovať:

- **Bundle-Name:** Názov bundlu v ľudskom tvare.
- **Bundle-SymbolicName:** The only required header, this entry specifies a unique identifier for a bundle, based on the reverse domain name convention (used also by the java packages).
- **Bundle-Description:** Popis funkcionality bundlu.
- **Bundle-ManifestVersion:** Indikuje OSGi špecifikáciu užívanú na čítanie bundlu.
- **Bundle-Version:** Číslo verzie samotného bundlu.
- **Bundle-Activator:** Názov triedy, ktorá je užitá ako Activator.
- **Export-Package:** Set exportovaných balíkov. V základe obsahuje len názov, no môže obsahovať aj špecifickú verziu a cielené bundly, pre ktoré je balík určený.
- **Import-Package:** Popisuje, ktoré balíky a akých verzií budú potrebné z vonkajšieho prostredia na beh bundlu.

V tomto zozname bola spomenutá špeciálna trieda Activator. Jedná sa o jednoduchú triedy, ktorá implementuje rozhnranie BundleActivator a je registrovaná v MANIFEST súbore. Takáto trieda je spustená práve raz počas nasadenia bundlu do prostredia a takisto počas jeho odinštalovania. V jednoduchých metódach tejto triedy máme možnosť zasahovať do OSGi kontextu a inštalovať/odinštalovať nami vytvorené servisy, prípadne registrovať poslucháčov na udalosti evokované v kontajnery (zmeny stavov bundlov, servís...).
Proces registrovania triedy za OSGi servisu je podmienený, tým že daná trieda musí implementovať rozhranie, ktoré bude alebo už je importované v prostredí (v balíku, ktorý je vedený v export-package sekcii). Samotná registrácia si potom vyžaduje ako vstup samotnú instanciu danej triedy, rozhranie, ktoré implementuje a ďalšie voliteľne parametre. Pod daným rozhraním a pridanými parametrami bude kontajnerom vyhľadávaná a vracaná na volania.
V neposlednom rade je veľmi dôležitý samotný proces inštalovania modulov, jedná sa o jednoduchý životný cyklus, ktorý obsahuje jednotlivo sledovateľné fázy- stavy bundlu:

- **INSTALLED:** bundle bol úspešne nainštalovaný.
- **RESOLVED:** Všetky bundlom požadované triedy sú v prostredí dostupne. Tento stav indikuje, že bundle je pripravený na štart alebo stop.
- **STARTING:** Bundle je v procese štartu. Metóda BundleActivator.start bola zavolaná a očakáva sa jej ukončenie spojené s registrovaním servís, či poslucháčov. Daný stav zostáva pre bundle platný v závislosti na štartovacom nastavení modulu.

- **ACTIVE:** Bundle bol úspešne nainštalovaný aj volaná metóda BundleActivator.start vykonala všetky definované operácie úspešne.
- **STOPPING:** Bundle je v procese zastavovania. BundleActivator.stop methóda bola zavolaná, aby odinštalovala všetky referencie servís z kontajnera a zrušila takisto všetkých svojich poslucháčov, ale ešte nedala žiadnu návratovú hodnotu.
- **UNINSTALLED:** Bundle bol úspešne odinštalovaný a je pripravený na prechod do iného stavu.



Obrázok 4: životný cyklus servís

OSGi Alliance takisto definuje set štandardných servís, ktoré značne rozširujú samotné API tohto modulárneho frameworku. Jedná sa o servisy zabezpečujúce logovanie, zasahovanie do nastavení bundlov, prístup pridaných zariadení, správu udalostí, IO prístupov, správu udalostí, a mnohé ďalšie.

5.2.4. Enterprise Java Beans

Následujúca kapitola čerpá z [7].
Enterprise Java Beans je špecifická architektúra na manažment modulárnych serverovo orientovaných enterprise aplikácií. Dané aplikačné servery musia obsahovať špeciálne behové prostredie, na ktoré sme schopný nasadiť naše beany. Jedná sa o EJB kontajner. Podľa Java EE špecifikácie daný kontajner obsahuje široké API, pomocou ktorého máme k dispozícii veľké množstvo služieb:

- **Transakčné spracovanie:** deklaráciou môžeme do kontajnera uviesť nedeliteľnú operáciu pre server ukončiteľnou jedine operáciami ako commit alebo rollback.
- **Prepojenie s databázou:** JPA ako súčasť JAVA EE nám dáva možnosť vytvorenia objektovo relačného mapovania nad našou databázou
- **Riadenie stavu:** kontajner je schopný držať stav určitého typu bean (hodnoty jeho atribútov)
- **Zabezpečenie:** deklarovanie prístupových práv na úrovni samotných bean, či dokonca jejích jednotlivých metód.

- **Správa interceptorov:** komponenty, ktoré sú schopné reagovať na udalosť pred alebo po volaní na nami definované metódy.
- **Asynchronné volania:** pomocou posielania správ a vhodných reakcií sme schopný využívať paralérne spracovanie na špecifické úlohy.
- **Plánovač procesov:** vhodnou deklaráciou máme možnosť zadať pravidelné opakovanie jednotlivých metód.
- **Riadenie životného cyklu:** v jednotlivých fázach nasadenia sme shopný zasahovať a upravovať parametre či priamo volať vlastné operácie.
- **Vzdialené volania (remoting, RMI, webové služby):** komunikácia zabezpečená serverom, možnosť z klientskej aplikácie vzdialene volať procedúry cez spomínané rozhrania.
- **Pooling:** správa množstva nasadených instancií bean v behovom prostredí.

Ako bolo viac krát spomenuté, základným kameňom enterprise aplikácií bude beana. Samotná beana musí byť vhodne prezentovaná, kedže sa v zásade rovnako ako u OSGi jedná o POJO (plain old java object). Pre popis objektov sa do verzie EJB3.0 používali XML deplyment descriptor, ktorý obsahoval údaje ako interface, názov beany, security údaje, atď. Od verzie 3.0 sa však toto popisovanie značne zjednodušilo. K označeniu bean pre kontajner sa využívajú anotácie. Takže každú potrebnú triedu či interface stačí vhodne anotovať a tým viažeme na ňu funkcionalitu, náležiacu z Java EE kontajneru. V ďalšom rade, každá beana musím implementovať pre ňu popisné rozhranie. Dané rozhranie je anotované pre kontajner buď ako @remote, alebo ako @local. V závislosti na užitej anotácii je k danej beane umožnený prístup z vnútra kontajnera alebo z klienta, ktorý sa dotazuje na server, na ktorom daný kontajner beží. Vo všeobecnosti proces spúšťania začína nasadením beany do EJB kontajneru, ktorý obvykle beží vnútri aplikačného serveru. Nasadzovaný jar by mal obsahovať javax.ejb balík, ktorý zabezpečuje sadu rozhraní pre prístup k Java EE API a samotnú implementáciu zabezpečuje už kontajner. Takisto samotný klient nepristupuje k beanam priamym instanciovaním ale získava referenciu z kontajnera pomocou JNI alebo modernejšie CDI. Daná referencia však nie je odkazom na samotnú implementáciu beany ale iba na proxy triedu, ktorá dynamicky poskytuje implementáciu požadovanej časti. Pre úplnosť Java EE API podporuje viacero typov bean.

- **Stateless Session Beana:** tento typ beany je instanciovaný vo väčšom počte vpoole, ktorý je určený podľa nastavenia administrátora. Pre každé klientské volanie je pridelená nejaká konkrétna instancia a počas vykonávania požiadavku (beh metódy) sa jej pridelenie nemení. Neudržuje stav vo svojich atribútoch, pretože pre ďalšie volanie môže byť pridelená inému klientovy. Na deklarovanie takéhoto objektu sa používa anotácia @Stateless.
- **Statefull Session Beana:** je na rozdiel od predošlej objekt, ktorý si medzi klientskymi požiadavkami udržiava svoj stav v serverovej, danému užívateľovi pridelenej sessione. Pre každé volanie klienta je mu vrátená rovnaká inštancia a v určitom prípade je možné jej stav dokonca uložiť do databáze – proces pasivácie beany. Takýto objekt deklarujeme anotáciou @Statefull.

- **Singleton session beana:** je objekt, s globálne zdieľaným stavom. Počet inštancií tohto objektu je vymedzený na jedinú inštanciu na celý kontajner. Na tento objekt je možné riešiť synchronizovaný prístup buď riadený kontajnerom, alebo priamo nastavením zamykania (anotáciou @Lock) na samotné procedúry objektu. Takúto beanu deklarujeme anotáciou @Singleton.
- **Message driven beana:** tento objekt predstavuje prístup k asynchrónnemu spracovaniu požiadavkou. Na jednej strane sú v pooli instanciované MDB's. Ďalším hlavným prvkom je správa, ktorá je vytvorená pomocou nejakého procesu a pridaná do zásobníka alebo fronty správ na servery. Táto udalosť pridania je odchytená kontajnerom a všetky voľné MDB's sú schopné na to reagovať prebratím a spracovaním danej správy.

Základným volaním v rámci kontajneru je využitie injekcie (injection), ktoré však v niektorých prípadoch použiť nie je možné. Sú to všetky prípady mimo prostredie kontajneru, webový kontajner iného serveru, klientská aplikácia využívajúca len j2se alebo behové prostredie iného kontajneru (nie Java EE).

V takom prípade získame referenciu na požadované proxy pomocou JNDI a menných konvencií kontajneru. Každá beana so sebou nesie JNDI názvy (globálny, aplikačný a modulový), vždy v závislosti aké rozhrania ju popisujú a tieto názvy sú pre beanu v jedinečné nielen v rámci kontajnera ale v rámci dostupnej siete (napríkla global name je zložené ako java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]). Potom jednoduchým new InitialContext.lookUp(<názov beany>) získame refernciu na požadovanú proxy.

Jednou z posledných vecí, ktoré je vhodné spomenúť, aj pretože to využíva môj modulárny systém, sú možnosti zásahu do životného cyklu beanov. Jedná sa o nami vytvorené metódy, ktoré stačí opäť vhodne anotovať. Pre odchytenie procesu nasadenia užijeme anotáciu @PostContruct a pre proces rušenia beany anotáciu @PreDestroy. V takto označných metódach možeme úpravou atribútov, či spustením ďalších procesov komplexne ovládať tieto procesy.

5.2.5. Hibernate

Údaje pre túto kapitolu sú získané zo zdroja [5].

Na objektovo relačné mapovanie som použil framework Hibernate. Je to robustné API, ktoré v hlavnej miere poskytuje mapovanie objektovo orientovaného doménového modelu na relačnú databázu.

Hibernate poskytuje generické a veľmi široké API, ktoré rieši mnohé technické ťažkosti, vznikajúce v prípade priameho mapovania modelu relačnej databáze na model objektovo-orientovaného programovacieho jazyka.

Najčastejšie otázky pri riešenení objektovo-relačného mapovania:

- Odlišnosti v komplexnejších dátových typoch
- štrukturálne a integritné rozdiely
- rozdiely v manipulácii s objektmi
- odlišnosti v transakčnom spracovaní.

Základnými prvkami Hibernate mapovania sú entity a konfiguračný súbor. Entita je klasický POJO, modelová trieda aplikácie, ktorá má jasne stanovený identifikátor, a na všetky atritbuty vedené samozrejme ako private, get a set metódy. Takýto POJO ďalej musíme vhodne anotovať alebo popísať konfiguračným skriptom.
Súbor najbežnejších anotácií pre príklad:

- **@Entity:** nad deklaráciou triedy, označuje danú triedu ako entity pripravenú na mapovanie
- **@Id:** jasne definovaný identifikátor danej entity, ďalej môže byť rozšírený o spôsob generovania hodnôt, pre každú novú entitu (napr. generate = GeneratorType.AUTO), pokiaľ jeho dátový typ je Long, pre ostatné typi sme schopný si generátor definovať.
- **@Column:** definícia jednotlivých atribútov (názov, veľkosť, možnosť nulových hodnôt,…)
- **@OneToMany prípadne @ManyToOne:** je vyjadrenie väzby 1 ku N, v závislosti na smere. Jedna z tried (trieda A) obsahuje kolekciu referencií tried B (reprezentuje entitu so zastúpením 1) a každá trieda B danej kolekcie má naopak 1 referenciu na triedu A (reprezentuje entitu so zastúpením N). Trieda B nad atribútom referencie na triedu A má anotáciu @ManyToOne a @JoinColumn, ktorým definuje cudzí kľúč na entitu A. V triede B je už len nad kolekciou anotácia @OneToMany.
- **@ManyToMany:** definuje väzbu N ku N. Kde obe triedy majú navzájom na seba referencie pomocou generických kolekcií, anotovaných práve @ManyToMany s pop9sanou JoinTable a JoinColum. Na základe toho bude vygenerovaná väzobná tabuľka s cudzími kľúčmi na dané dve entity.

Veľmi podobné je to v prípade XML konfigurácie. Napríklad:

```
<hibernate-mapping package="net.viralpatel.hibernate">
    <class name="Employee" table="EMPLOYEE">
        <id name="employeeId" column="EMPLOYEE_ID">
            <generator class="native" />
        </id>
        <property name="firstname" />
        <property name="lastname" column="lastname" />
        <property name="cellphone" column="cell_phone" />
        <many-to-one name="department"
         class="net.viralpatel.hibernate.Department"
         fetch="select">
            <column name="department_id" not-null="true" />
        </many-to-one>
    </class>
</hibernate-mapping>
```

Zdrojový kód 1: príklad pre Hibernate mapovanie

Pretože Hibernate od verzie 3.0 je plne nastaviteľný pomocou anotácií, osobne preferujem túto možnosť aj v tomto projekte.
Veľmi jednoducho sa ďalej definuje samotné pripojenie na databázu, kde sa pomocou triedy org.Hibernate.cfg.AnnotationConfiguration definuje url na databázu, užívateľské meno, heslo, ovládač, dialekt, možnosť automatického komitovania, spúšťací mód mapovania, atď...
Po správnom nastavení je Hibernate mapovanie počas nasadenia alipkácie schopné štruktúru databázových tabuliek vygenerovať, či zvalidovať. Následne je pre nás podstatné ako tvoriť dotazy a ako operovať s dátami. Hibernate nám poskytuje znova niekoľko možností. Od tvorenia HQL (Hibernate query language) dotazov, cez využívanie Hibernate- Criteria API prípadne spúšťanie vlastných SQL dotazov.
Hibernate criteria dotaz pre príklad:
vyberám z entity Person, všetky záznamy staršie ako dnešný dátum (parameter created_at menší)-
List<Person> persons= session.createCriteria(Person.class).add(Restriction.lt("created_at", new Date())).list();
Takto som získal generickú kolekciu výsledkov, kde každý z nich má načítané všetky dostupné atribúty a takisto všetky referenčné objekty.
Hlavným aspektom pri samotnom dotazovaní je aj kontrola nad každým dotazom. V rámci criteria API si môžeme pomocou statickej kolekcie Projections.add(<názov atribútu>) nastaviť požadované atribúty v ramci dotazu a pomocou nastavení „lazy loadingu" nad tabuľkovou väzbou, získame, že Hibernate nebude načítať celé kolekcie previazaných objektov ale vráti len to čo naozaj potrebujeme.
Hibernate som zvolil hlavne kvôli jeho jednoduchosti a prenositeľnosti na akúkoľvek inú SQL databázu.

5.2.6. Vaadin

Vaadin popíšem zo zdroja [1].
Webové rozhranie projektu som vytvoril v takisto java frameworku, konkrétne vo Vaadin-e. Jedná sa o jednoduché, open-source, moderné, komponentovo orientované riešenie určené na tvorbu RIA (rich internet applications). Tvorené bolo spočiatku spoločnosťou IT Mills ako IT Mills Toolkit, dnes od Vaadin Ltd. je dostupný pod licenciou Apache License 2.0. Často býva porovnávaný s frameworkami ako Wicket alebo JSF, on sám však razí cestu proti využívaniu viacerých technológií na webe. V skutočnosti programovať webovú aplikáciu s prepracovaným užívateľským rozhraním je možné s Vaadin bez znalosti CSS a javaScriptu.
Po technickej stránke je Vaadin plne Java orientované API, spolupracujúce s GWT (Google web toolkit), čo je webový framework od googlu plne postavený na AJAX so širokou škálou RIA komponent. Vaadin poskytuje API, pomocou ktorého vytvorená webová aplikácia môže byť jeden obyčajný jar balík na servery. Základom je sada komponent na strane Javy, pomocou ktorých vyskladaný web sa zobrazí u klienta. V klientskom prehliadači už ale bežia komponenty, ktorých jadrom je GWT engine a ten cez AJAX volania cez HTTPS protokol komunikuje so serverovou časťou aplikácie. Programátor na strane serveru túto komunikáciu v skutočnosti nijak riešiť

nemusí. API poskytuje celú sadu poslucháčov na najrôznejšie udalosti generované komponentami ako kliknutie, prechod myšou, zmena hodnoty. Celkový pohľad na kód aplikácie výrazne pripomína desktopovú aplikáciu.

Keďže HTML kód stránky je v najväčšej miere generovaný, nejedná sa nijak o optimalizovaný kód pre vyhľadávače. Vaadin sa viac menej nedoporučuje k využívaniu na prezentačné weby alebo e-shopy. Ideálna príležitosť je v rámci intranetových systémov, kde indexovanie vyhľadávačom nie je potrebná.

Benefitom užívania tohto riešenia je nulová potreba tvoriť javaScript alebo HTML, keďže kompletný desing aplikácie vnesieme pomocou Vaadin API z Javy. Jediné čo ostáva v rámci UI riešiť mimo Javu je CSS štýlovanie, s pomocou ktorého si môže vzhľad každej komponenty prispôsobiť vlastným predstavám, či už pomocou úpravy CSS štýlov alebo kombinovaním tematických šablón pre komponenty.

Písanie čisto Java kódu poskytuje takisto možnosť využívania návrhových vzorov známych z desktopových aplikácií ako napríklad composite pattern pre skladanie UI prvkov s podobnými vlastnosťami, state pattern pre definovanie ich stavov prípadne listener pattern pre definovanie reakcií na udalosti, čím kód získa prirodzenú štruktúru a čitateľnosť.

Základom Vaadin aplikácie je servlet. Ten je definovaný vo web.xml s referenciou na triedu, ktorá rozširuje abstraktnú triedu vaadin.com.Application. V rámci poskytovanej metódy si jednotlivo pridáme komponenty, kaskádovo ukladáme, ako vo Swing frameworku čím získavame komplexnejší celok.

Vaadin pre ilustáciu má obrovské množstvo komponent deliacich sa do viacero kategórií:

- **Základné UI komponenty:** tlačidlá, linky, texty, klávesové skratky, ikony,...
- **Komponenty pre vstupné dáta:** kalendáre, upload komponenty, select boxy, slidery,...
- **Dátové modely:** zložitejšie formuláre s vlastnou logikou, prípadne upraveným UI (prihlasovanie)
- **Mriežky a stromy:** tabuľky (textové, s ikonami, s triedením, s postupným načitaním, ...), stromové zoznamy (ako kontextové menu, viacnásobný výber, možnosť ikon, klávesová navigácia,...)
- **Drag'n'drop komponenty:** možnosť presúvania obsahu medzi komponentami ako tabuľky, stromy, formuláre,...
- **Layouty a komponentové kontajnery:** panely, horizontálne či vertikálne layouty (možnosť rozšírenia o CSS rovnako ako u iných komponent)
- **Okná, notifikácie a navigácia:** od základných okien, modálnych okien až po varovania či potvrdzovacie formuláre
- **Témy:** každá komponenta má na výber viacero základných variácií tvarov a farieb.

Ďalej sú k dispozícii priamo na stránke vaadin.com rôzne addony, rozšírenia, čo sú open source komponenty tvorené zväčša komunitou. Od jednoduchých vecí ako komponenta vykresľujúca google mapu, po rôzne galérie tu môžeme nájsť aj riešenia integrácie s inými webovými frameworkami ako grails alebo spring.

Problémom u komoponentovo orientovaných frameworkov je aj jejich podpora webových prehliadačov. Pretože ale GWT funguje už veľmi dlho na všetkých najpoužívanejších prehliadačov stabilne a tvorcom GWT enginu je samotný Google, nie je s Vaadin v tomto smere žiaden problém.
Možnosť vývoja v najpoužívanejších IDE ako Netbeans, Eclipse či InteliJ s interaktívnym UI-builder nástrojom, kvalitná dokumentácia a množstvo obsiahlych návodov a príkladov použitia robí z tohto frameworku skutočne jednoduchý a dostupný nástroj pre vývoj.

5.2.7. Maven

Údaje pre túto kapitolu som získal zo zdroja [3].
Maven je nástroj pre správu, riadenie a automatizáciu zostavenia aplikácií. Napriek tomu, že je možné použiť tento nástroj pre projekty vytvorené v rôznych programovacích jazykoch ako Scala, C#, Groovy, podporovaný je prevažne jazyk Java.
Hlavným motívom vzniku a celá idea nástroja je snaha o štandardizáciu a znovu použiteľnosť zozstavovacích skriptov, ktorá v dovtedy najpoužívanejšom nástroji Apache Ant nebola plne podporovaná. Na rozdiel od spomínaného Ant, Maven je čisto deklaratívny nástroj, slúžiaci pre popis projektu, nastavenie jednotlivý fáz zostavovania, generovania informácií o projekte a hlavne o znovu použiteľnosť daného skriptu. Samotný Ant je totiž nástroj pre unikátneho zostavoacieho skriptu pre každý projekt, volanie jednotlivých projektových častí a vo všeobecnosti sa jedná “len” o zostavovací skript.
Základným princípom fungovania Mavenu je popísanie projektu pomocí Project Object Model (pom) súboru. Tento model popisuje softwarový projekt nie len z pohľadu jeho zdrojového kódu, ale aj závislostí na externých knižniciach, popisu procesu zostavovania a rôznych funkcií s tým spojených (ako je spúšťanie testov, zber informácií o zdrojových kódoch, generovanie špeciálnych balíkov, popisných súborov a tak ďalej).
Maven sám je postavený na modulárnej architektúre a funguje len ako nástroj pre volanie pridaných modulov. Sám spravuje dodanie pluginu o definovanej verzii z určitého repozitára a jeho následné spustenie s možnosťou pridania parametrov. Maven nemá žiadne vlastné grafické užívateľské rozhranie, beží len v príkazovom riadku.
Celý zostavovací proces nástroja sa delí na niekoľko životných cyklov, od validácie, inicializácie, generovanie zdrojov, cez kompilovanie projektu, ďalej kompilovanie a spúšťanie testov balenie (packaging), spúšťanie integračných testov, verifikovanie až po inštaláciu a nasadenie. Pomocou vhodne definovaného skriptu nám dáva Maven počas ktorejkoľvek fázy spustiť pridaný plugin s našimi nastaveniami, čím si samotný proces zostavenia upravíme podľa svojich predstáv. Napríklad pre výpis fáze:

```
<execution>
      <id>id.compile</id>
     <!—definovanie fáze -->
      <phase>compile</phase>
      <goals>
         <goal>run</goal>
      </goals>
      <configuration>
         <tasks>
            <echo>compile phase</echo>
         </tasks>
      </configuration>
</execution>
```

Zdrojový kód 2: definícia fáze v maven

Ako bolo spomenuté väčšina funkcionality Mavenu je v množstve dostupných pluginov pre kompilovanie, generovanie štruktúry, testovanie projektov či spúšťanie serverov alebo vytváranie popisných súborov pre projekt atak ďalej. Podstatné je vždy definovať samotný plugin, nastavenia preň a spustenie (konkrétnu fázu). To je možné znova skriptom alebo z príkazového riadku napríklad ako:
mvn [plugin- name] : [goal- name]
Ďalšou veľmi podstatnou funkcionalitou, ktorú Maven poskytuje je definovanie závislostí (dependencies) projektu, knižníc, ktoré sú potrebné pre jednotlivé fázy alebo pre beh samotného projektu, aj tie ktoré sú normálne definované v class- path projektu.

```
<dependencies>
    <dependency>
      <!—definícia potrebných závislostí -->
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <!—popis užitia- testovanie -->
      <scope>test</scope>
    </dependency>
</dependencies>
```

Zdrojový kód 3: maven definovanie knižníc

V základe Maven má k dispozícii vlastný repozitár, v ktorom hľadá pluginy alebo knižnice na stiahnutie. Pokiaľ sa tam požadovaný objekt nenachádza, Maven vráti výnimku. V takom prípade je treba v popisnom súbore (pom) definovať ďalší repozitáz, v ktorom by sa mali požadované objekty nachádzať.
Príklad pridaných repozitárov:

```
<repositories>
    <repository>
      <id>my-repo2</id>
      <name>your custom repo</name>
      <url>http://jarsm2.dyndns.dk</url>
    </repository>
 </repositories>
```

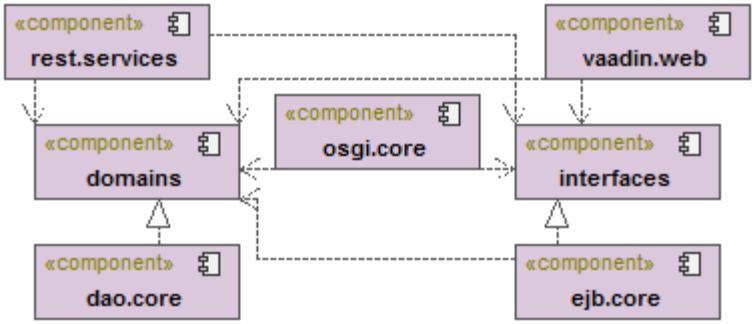
Zdrojový kód 4: pridanie externého repozitára v maven

Práve pre možnosť fixnej a znovu použiteľnej definície projektu je celý projekt Modular inteligent image storage postavený ako Maven projekt a na každý ďalší modul je pred pripravený zostavovací skript, ktorý presne určí štruktúru a zostavenie modulu.

6 Implementácia

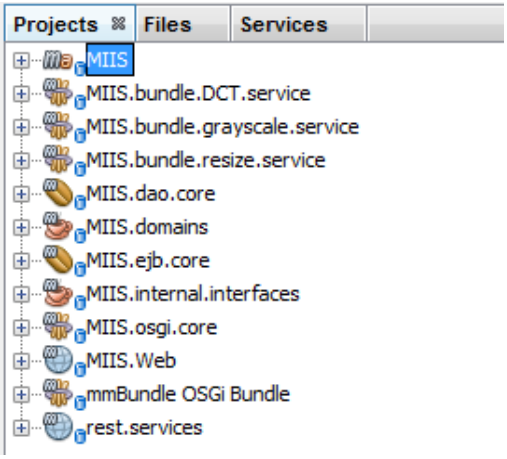
6.1. Popis funkčnosti hlavných modulov

Celý projekt je navrhnutý ako modulárna aplikácia, nasadená na aplikačnom servery. Jednotlivé moduly sú previazané cez komponenty obsahujúce sady rozhraní (interfaces a domains), čím sa zachováva nezávislosť na realizácií týchto komponent. Konkrétne implementácie sú potom poskytované z Java EE initial kontextu, vďaka čomu nie sú potrebné priame väzby na komponenty s implementáciou. Nasledujúci diagram popisuje závislosti a realizácie medzi modulmi.



Obrázok 5: architektúra systému

Je zjavné, že na beh systému nie sú potrebné všetky moduly, niektorý jeden z horných modulov (rest.services alebo vaadin.web) nemusí byť nasadený, pretože oba v skutočnosti poskytujú rozhranie na funkčnosť systému užívateľovi, akurát každý iným spôsobom. Pre ilustráciu ukážem jednotlivé moduly otvorené vývojárskym nástrojom Netbeans.



Obrázok 6: jednotlivé moduly v netbeans

Ostatné moduly sú nevyhnutné na zachovanie funkčnosti, prípadne vôbec na spustenia systému. V ďalších podkapitolách bude nasledovať popis týchto modulov.

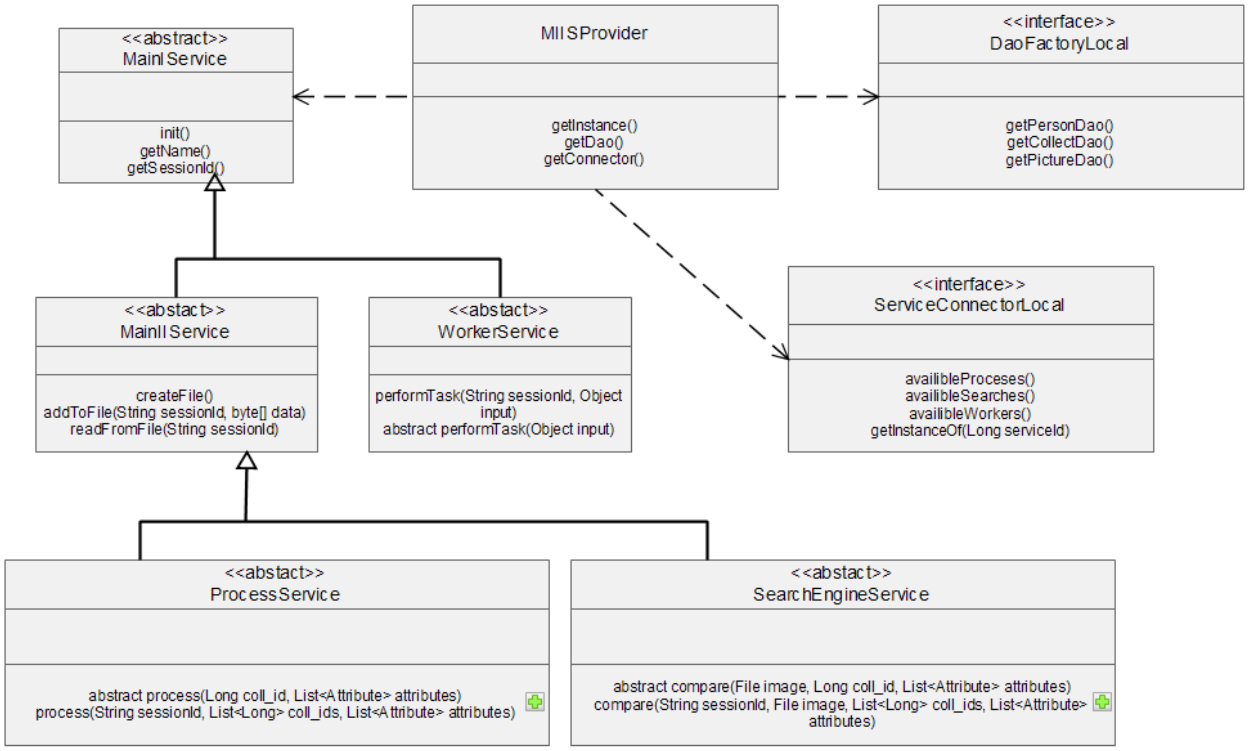
MIIS.interfaces

Balík typu jar, ktorý v hlavnej miere obsahuje rozhrania na väčšinu tried, ktoré v systéme vystupujú či už ako beany alebo OSGi servisy. K jednotlivým beanam obsahuje lokálne (@local) rozhrania, ktoré sú používané v rámci CDI prepojenia poskytovaného Java EE kontajnerom. Pre OSGi-servisy je tu MainIService rozhranie, ktoré je následne čiastočne implementované tromi viacerými abstraktnými triedami. Každá abstraktná trieda odpovedá inému typu servisy:

Typ servisy	Abstraktná trieda
worker	Miis.common.abst.WorkerService.java
search	Miis.common.abst.SearchEngineService.java
process	Miis.common.abst.ProcessingService.java

Tabuľka 1: typy servís v MIIS

MainIService slúži implementáciu metódy init(), kde sa inicializujú základné veci ako logovanie, počiatočný čas a uloženie globálnej sessionId, ďalej je tu metóda na vrátenie samotnej sessionId a deklarácia metódy getName(), pomocou ktorej by mala každá servisa v systéme mať svoj názov.
Jej prvý potomok je WokerService, má implementáciu performTask() metódy, pomocou ktorej je inicializované volanie z aplikácie. Táto metóda má na vstupe sessionId, ktoré pošle do metódy init, nadradenej MainIservice – tým prebehne inicializácia a uloží sa počiatočný systémový čas. Ďalej má na vstupe object input, ktorý pošle ako vstup do ňou deklarovanej metódy performTask, čím spustí nad požadovaným vstupom konkrétny zásuvný modul.
Na rovnakej úrovni je abstraktná trieda MainIIService, ktorá okrem dedičnosti z MainIService obsahuje navyše jednoduché API pre prácu so súbormi, ktoré je pre asynchrónny kontext týmto oddelené. Dôvod tohto rozdelenia je popísaný v kapitole 7.5.5..
Na najnižšej úrovni dedičnosti sa nachádzajú abstraktné triedy ProcessService a SearchEngineService, ktoré podobne ako WorkerService obsahujú naimplementovanú metódu svojho typu, ktorú volá systém, následne v nej prebieha inicializovanie a až potom spustenie modulu nad vstupnými dátami. Pre obe je vstupom kolekcia identifikátorov kolekcií obrázkov, ktoré sú iterované a po jednej spúšťané. SearchEngineService má okrem kolekcie identifikátorov na vstupe aj súbor obrázku, ktorý má za úlohu porovnať s obrázkami danej kolekcie a nájsť zhodu. Ako jediná vracia výsledok do systému a jej výstupným parametrom je kolekcia identifikátorov podobných obrázkov.



Obrázok 7: UML diagram API

Nakoniec sa tu nachádza aj dôležitá API trieda miis.common.MIISProvider.java, ktorá poskytuje komunikáciu s nasadenou enterprise aplikáciou, možnosť volaní jednotlivých modulov či prácu nad databázou. MiisProvider je riešený patternom singleton, nie je možné vytvoriť k nemu ďalšiu instanciu pre jeho privátny konštruktor. Instancia sa získava cez metódu getInstance(). Jeho využitie je podstatné práve pre pridané moduly, ktoré pomocou neho môžu jednoducho volať beany systému.

Na tento modul má závislosť väčšina ostatných modulov systému, keďže som sa pokúšal o čo najvoľnejšie väzby medzi modulmi.

MIIS.domains

Ďalší balík typu jar, ktorý takisto obsahuje len rozhrania exportované do systému. Ideou tohto modulu je definovanie prístupu k práci nad databázou a jej objektom. Obsahuje dve sady rozhraní orm a orm.idomain.

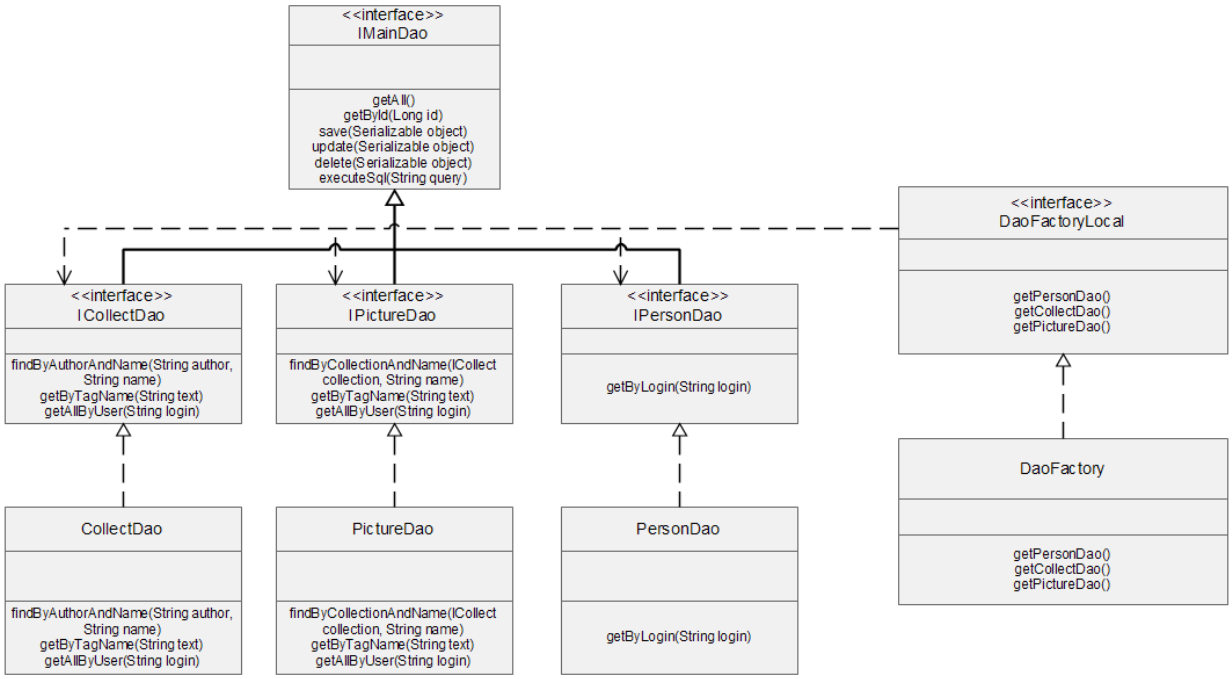
V rámci orm sa nachádza rozhranie DaoFactoryLocal pre popis beany DaoFactory a všetky jej generované prvky ako Attribuo, CollectDao, PersonDao a tak ďalej, akurát jednotlivé rozhrania majú pred názvom príponu I (IAttribuo, ICollectDao,...). Je jasné že implementáciou všetkých rozhraní a správnym referencovaním v DaoFactory, získame návrhový vzor továreň pre prístup k jednotlivým objektom modelu.

Balík orm.idomain, je súbor kde sa nachádzajú rozhrania pre doménové triedy systému. Každé z nich obsahuje definíciu get a set metód pre všetky atribúty danej doménovej triedy a takisto

rozširuje rozhranie Serializable. Vytváranie rozhraní pre doménové triedy sa môže zdať prehnané, avšak pre modulárnu štruktúru a následnú voľnosť rozšírenia je to skoro nevyhnutné.
V prípade, že potrebujeme pracovať s databázou (vyberať či ukladať záznamy) potrebujeme instancie doménových tried. Tie sú ale uložené v moduly, ktorý pomocou frameworku Hibernate ich popisuje a pracuje s databázou. Priama väzba na daný modul neprichádza do úvahy, pretože v prípade potreby týchto tried by musel byť celý Hibernate framework pridaný do classpath niekoľkých modulov a to úplne zbytočne. Ďalšou výhodou je, že máme možnosť vytvoriť viacero implementácií databázového prístupu a len vhodným výberom továrne v skutočnosti budeme rozhodovať o výbere databáze. Takže jednoducho z initialContextu vyberieme implementáciu DaoFactoryLocal, tá nám cez get metódu vráti API pre potrebnú doménovú triedu, ktorej instanciu získame na základe vstupných parametrov. Tým sme schopný pracovať s dátami z databáze v každom moduli bez priamej závislosti na modul, v ktorom je ORM mapovanie implementované.

MIIS.ejb.core
Hlavný modul pre komunikáciu s OSGi kontajnerom, správu súborov a spúšťanie asynchrónnych úloh rieši modul miis.ejb.core pomocou troch hlavných bean. Tento modul má závislosti na predošle spomenuté moduly a to MIIS.domains pre prácu s doménovými triedami a miis.interfaces, pretože rozhrania sú pre nasadenie týchto bean nevyhnutné.
Prvou beanou je FileTransferBean. Jedná sa o stateless beanu, ktorá cez implementáciu FileTransferBeanLocal poskytuje API pre prácu so súbormi, adresármi, vracia streamy pre upload a download súborov a takisto sa stará o dekomprimovanie prijatých súborov.
Ďalšou je ServiceConnectorBean, ktorá je opäť typu stateless. Jej špecialitou je nasadenie v oboch kontajneroch (Java EE a OSGi) zároveň. Je to docielené pomocou využitia jej životného cyklu nasadenia a OSGi API ako je podrobnejšie popísané v kapitole 7.5.1.. Táto výsada jej dovoľuje byť volaná a vracať výsledky do oboch kontajnerov. Takže komunikácia medzi oboma behovými prostrediami, vracanie instancií a spúšťanie procesov sú jej hlavné úlohy.
Poslednou beanou je ImageProcessEngineMDB, ktorá je typu messageDriven, je ako poslucháč na zásobníku správ, reaguje na prijaté správy a spúšťa procesy podľa obsahu prijatých správ. Neimplementuje žiadne systémové rozhranie, cez ktoré by bola volateľná, takže zo systému k nej nie je priamy prístup. Podrobnejší popis k nej sa nachádza v kapitole 7.5.5..

MIIS.dao.core
Nasledujúci popisovaný modul je takisto typu EJB. Rieši mapovanie doménového modelu aplikácie na databázové entity a zabezpečuje prácu nad nimi. Celá táto funkcionalita je zabezpečená frameworkom Hibernate, na ktorý má tento modul priamu závislosť. Vhodným anotovaním a vnesenými nastaveniami ide v skutočnosti len o implementáciu API popísaného rozhraniami v module miis.domains.



Obrázok 8: UML diagram pre DAO

Modelové triedy aplikácie sú obsiahnuté v jednom balíku, kde každá implementuje pre ňu určené rozhranie z modulu miis.domains. Spolu s Hibernate anotáciami sa z nich stávajú entitné triedy.
V ďalšom balíku sa nachádzajú implementácie prístupových tried k týmto entitám. Napríklad pre prístup k entite Person, slúži API trieda PersonDao, ktorá implementuje rozhranie IPersonDao z MIIS.domains. V týchto implementáciách rozhraní sú metódy pre prístup k dátam na základe určitých vnesených parametroch, jedná sa väčšinou o prácu s Hibernate session.
V poslednom veľmi hlavnom balíku je implementácia rozhrania DaoFactoryLocal, DaoFactory, ktorá je typu singleton s pridanou anotáciou startup, čo zabezpečí okamžité spustenie po nasadení a trieda HibernateUtils, ktorá v statickom kontexte tvorí SessionFactory podľa vnesených nastavení. Jedná sa klasicky o meno a typ databáze, užívateľ a jeho heslo a tak ďalej.
Po platnom nasadení tohto modulu si môžeme byť istý, že databáza cez rozhranie z miis.domains je plne k dispozícii.

MIIS.osgi.core

Toto je v skutočnosti jediný OSGi bundle, ktorý je súčasťou základnej kostry systému. Slúži na správu nasadenia užívateľom pridaných modulov, aby boli správne nainštalované, aby systém získal potrebné informácie ale takisto aby sa prejavili zmeny v databázi požadované užívateľom v pridávanom moduli.
Všetko je zabezpečené jediným poslucháčom na udalosti inštalovania a odinštalovania osgi servís. Počas týchto udalostí upraví v databázi záznam o spravovanom moduli, prípadne podľa pridaného XML spustí operáciu nad databázou. Tento proces bude podrobnejšie popísaný ďalšou kapitolou,

zatiaľ stačí vedieť že spolu s poslucháčom je v moduli aj aktivačná OSGi trieda, ktorá ho registruje a celý modul má závislosti definované na moduly s rozhraniami systému.
Podrobnejší popis registrovania a spúšťania poslucháčov je v kapitole 7.5.2..

MIIS.web
Miis.web sa takisto dá považovať za modul, pretože sa jedná o ucelený balík, nasadený na server s možnosťou interakcie s Java EE kontajnerom. Má závislosti na systémové rozhrania a takisto na framework Vaadin, v ktorom je celá webová aplikácia napísaná. Hlavnou úlohou tohto modulu je poskytnutie webového rozhrania pre užívateľa, s pomocou ktorého je schopný uložiť, či stiahnuť kolekciu obrázkov, prezerať jednotlivé obrázky, tagovať ich, nasadzovať pridané moduly a spúšťať ich nad jednotlivými kolekciami. Podrobnejšie rozpracovanie a poukázanie na užité návrhové vzory sú popísane v kapitole 7.2.

MIIS.rest
Ďalším modulom, s ktorého pomocou je užívateľovi prístupná väčšina funkčnosti systému je MIIS.rest. Je už z názvu jasné, že sa jedná o sadu REST servís, nasadenú ako EJB modul v Java EE kontajnery. Vhodným dotazovaním z klientskej aplikácie má užívateľ rovnaké možnosti na prácu so systémom ako keby pracoval z webovej aplikácie.

Ďalšie moduly, ktoré nie sú zakreslené v úvodnom diagrame, sú modulmi, ktoré tvoria rozšírenie systému o pridanú funkcionalitu a sú tu takisto ako príklad pre ďalšie rozšírenie. Ide o OSGi bundly, so závislosťou na MIIS.domains a MIIS.interfaces, ktoré podľa potreby implementujú určité rozhranie a exportujú vlastnú servisu do OSGi kontextu. Ich funkčnosť, nasadenie popíšem v ďalšej kapitole. Jedná sa o moduly:

- **MIIS.bundle.DCT.service**
- **MIIS.bundle.grayscale**
- **MIIS.bundle.resize**
- **MIIS.bundle.MMService**

6.2. Popis prídavných modulov (OSGi bundly)

Ako bolo spomenuté v predošlej sekcii, projekt obsahuje sadu zásuvných modulov, ktoré slúžia viac menej ako príklady pre tvorbu a užitie API. Každý z nich demonštruje nejakú metódu digitálneho spracovania obrazu a každý z nich využíva API trochu iným spôsobom.

MIIS.bundle.DCT.service
Toto je najrozsiahlejší zásuvný modul tejto kolekcie. Jeho hlavnou funkcionalitou je využitie diskrétnej kosínusovej transformácie a Huffmanovho kódovania popísaného v kapitole 2.1.. Technicky obsahuje dva typy servís. Prvá servisa je procesná, ktorá na základe kolekcie na vstupe vytvorí pre všetky obrázky volania pracovnej servise (worker). Následne je pripravená spomenutá

pracovná servisa, ktorá na základe volaní dané obrázky spracuje (skomprimuje) a uloží do adresáru na servery. Technicky celý proces nad kolekciou prebieha asynchrónne, implementačné detaily preberiem v neskoršej kapitole.

MIIS.bundle.grayscale

Tento modul je podstatne jednoduchší ako predchádzajúci. Teória k prevodu je v rámci kapitoly 2.2.. Znova obsahuje procesnú servisu, ktorá je zavolaná z vonka nad jednou alebo viacerými kolekciami. Následne každú kolekciu preiteruje a jednotlivé obrázky pošle ako volania do API, čím zabezpečí niekoľkonásobné spustenie pracovnej servisi vo svojom vnútri. Táto pracovná servisa už len daný obrázok otvorí, prevedie na bytové pole a jednotlivým prenásobením farebných zložiek každého pixelu prevedie obrázok z plnofarebnej do čiernobielej škály a výsledok znova uloží na server.

MIIS.bundle.resize

Tento zásuvný modul pracuje podobne ako predošlý, akurát s tým rozdielom, že vo finálne pracovná servisa obrázok pomocou BufferedImage a Graphics2D API zmenší na požadovaný formát. Je tu zavedená možnosť vnesenia parametra, ktorý bude udávať veľkosť výsledného obrázku.

MIIS.bundle.MMService

Posledný prídavný modul, ktorý je v základe, je modul zabezpečujúci uloženie celej kolekcie do súboru typu matrix market a následne je schopný nájsť zhodné obrázky k tým, ktoré dostane ako na vstupe. Začína to znova procesnou servisou zavolanou nad niekoľkými alebo len jednou kolekciou, každá z kolekcií je procesnou servisou preukladaná do súboru typu matrix market, ktorý je spomenutý kapitolou 2.3..

Ďalšia prítomná servisa je typu search, určená na vyhľadávanie. Vstupom do nej je obrázok, či už zvolený z aplikácie alebo práve uploadovaný. Táto servisa nevolá žiadne ďalšie asynchrónne procesy a výsledok, kolekciu identifikátorov podobných obrázkov vracia okamžite po ukončení na web alebo ako odpoveď webovou servisov. Celý proces porovnania obrázkov je len porovnanie vektorov a pokiaľ sú dané vektory zhodné, je daný obrázok poslaný na výstup.

7 Popis implementačných detailov

7.1. Zostavovanie modulov

V rámci projektu rozlišujem podľa potreby nasadenia a vnútorného obsahu štyri druhy modulov. Každý je buildený na iný výsledný balík a má iný konfiguračný súbor pre Maven.

Prvým typom je základný JAR balík. Takéhoto typu sú moduly obsahujúce rozhrania a sú nasadené na oba kontajnery, musí ísť o univerzálny balík. Na rozdiel ale od bežného JAR súboru je zostavený pomocou pluginu maven-bundle-plugin, v ktorom sú vnesené nastavenia pre export balíkov, ktoré obsahujú dané rozhrania do popisného súboru. Výsledkom je JAR súbor, ktorý má okrem všetkého aj popisný súbor MANIFEST.MF s kompletným popisom pre OSGi prostredie. Pokiaľ by tak nebolo, nemohol by byť nasadený žiaden z modulov, ktorý má nejakým spôsobom závislosť na dané rozhrania a počas štartu by OSGi kontajner neustále vracal výnimku.

Ďalším typom sú EJB moduly, presne miis.dao.core a miis.ejb.core. Tieto sú nasadené len do Java EE kontajneru, takže nie je potrebné aby obsahovali popisný manifest súbor. Sú zostavené klasickým pluginom maven-ejb-plugin s pridanou úlohou na fázu package, cez ktorú exportujú sebou používané knižnice na server. Tak sa úspešne nasadia, a keď dôjde k nasadeniu samotného modulu, budú pripravené.

Na web a webové služby som klasicky použil Netbeans projekt typu Maven Web-application, takže znova štandardný maven-war-plugin pre zostavenie a pridané závislosti ako v predošlých, vždy podľa potreby.

Posledným typom je projekt typu zásuvný modul, OSGi bundle. Tento má priamo v Netbeans možnosť vytvorenia ako klasický OSGi bundle, ale v prípade, že sa jedná o iné vývojové prostredie, je potrebné mať v Maven konfiguračnom súbore nastavené zostavenie (packaging) na bundle, v závislostiach knižnicu org.osgi.core a na zostavenie využiť maven-bundle-plugin, ktorému správne nadefinujeme Bundle-Activator, triedu ktorá zaregistruje objekty do kontextu. Kompletný konfiguračný súbor pred vytvorením bundlu doporučujem jednoducho skopírovať a použiť. Pokiaľ pridáme akékoľvek závislosti alebo len potrebujeme zostaviť projekt, riadime sa ďalej ako u štandardného Maven projektu.

Projekty majú inak štruktúru klasického Java projektu, a vďaka Maven konfiguračným súborom je možné dané projekty importovať do akéhokoľvek vývojového prostredia, ktoré bude mať pre Maven podporu.

Jediným zásadným problémom môže byť využitie knižníc v zásuvnom moduli, ktoré nie sú zostavené ako bundle balíky. Tieto moduly nie sú bez popisného súboru MANIFEST schopné exportovať vlastné triedy a je nemožné pre servisy systému aby ich používal. Kontajner na ich zavolanie vráti výnimku, že nie je schopný vrátiť instanciu na neexportovanú triedu. Riešením je všetky Java package zapísať ručne do súboru MANIFEST a celý JAR súbor s naším popisným súborom skompilovať. Výsledný JAR bude exportovať triedy, ktoré potrebujeme.

7.2. HTTP a REST

Celý projekt je serverová enterprise aplikácia a v nasledujúcom bode by som rád popísal prístup k nej, možnosť jej ovládania, posielania dát a vracania výsledkov, a keďže v rámci zadania bola požiadavka o naimplementovanie aspoň dvoch spôsobov prístupu, boli vytvorené prístupy pomocou webového rozhrania, REST-ových služieb pre volania beanov z klientskej aplikácie. Oba spôsoby majú spoločnú podmienku aby zariadenie bolo pripojené do lokálnej siete alebo internetu, v závislosti na tom ako je celá aplikácia publikovaná.

U štandardného webového rozhrania je princíp jednoduchý. Ako klientská aplikácia vystačí webový prehliadač s podporou javaScriptu. Podpora javaScriptu je síce dnes skoro samozrejmosť ale v prípade, že by nebola, webový projekt na bázy Vaadin frameworku a GWT by sa v prehliadači nijak nespustil. Po úspešnom spustení stránky je k dispozícii užívateľovi pripravené UI, kde pomocou štandardných komponent dáva príkazy k odoslaniu, prijatiu, náhľadu kolekcie či samotného obrázku, takisto má k dispozícii kontrolu nad zásuvnými modulmi, či už procesnými alebo vyhľadávacími službami a v reálnom čase má možnosť vidieť, ktoré moduly sú nainštalované a pripravené na použitie.

Technologicky bol Vaadin popísaný v predošlej kapitole, jediné čo ostáva objasniť je jeho prepojenie s java enterprise kontajnerom a závere uviesť užívateľa do grafického rozhrania.

Vaadin je servletovo založená technológia, ktorá je primárne určená na beh a spoluprácu so štandardným webovým kontajnerom, no však spôsobov ako komunikovať s aplikačným kontextom je v tomto prípade viac. Riešenie, ktoré využívam je zo všetkých možno najjednoduchšie, využíva Vaadin API nižšieho levelu a nie je v ňom potreba spravovať žiadne náročné XML nastavenia.

Stačí aby trieda, ktorá dedí z com.vaadin.Application a štandardne je registrovaná ako wellcome-servlet vo web.xml popisnom súbore, je ako súšťací bod aplikácie, bola anotovaná ako statefull beana, bude sa držať pre užívateľa jedna instancia v aplikačnom kontajnery a cez ňu budeme mať prístup pomocou štandardného injektovania k ostatným beanam. Tu sa ale naskýta problém. Pokiaľ užívateľ klasicky zadá URL, aplikačný kontajner mu síce vráti aplikáciu ale sám pre užívateľa inicializuje a bude spravovať servlet a nie instanciu beany, takže toto nestačí aby bol prístup do aplikačného kontextu. Zabezpečiť spustenie a správny beh beany na zavolanie užívateľa docielim tak, že k aplikácii budem pristupovať pomocou ďalšieho servletu. Takže som vytvoril štandardnú Java triedu, ktorú som anotoval ako WebServlet s atribútmi udávajúcimi názov a URL cestu, vďaka čomu bude prístupný užívateľovi bez potreby ho pridávať do web.xml, z ktorého som samozrejme odstránil nastavenie pre wellcome-servlet (predošle tam bol štandardný vaadin spúšťacia aplikácia). Ďalej budem potrebovať aby pridaný servlet rozširoval triedu com.vaadin.terminal.gwt.server.AbstractApplicationServlet a následne som vytvoril implementáciu pre metódy getNewApplication, kde siaham do inicialContextu a získavam správnu instanciu beany, ktorú následne vraciam užívateľovi a druhú metódu getApplicationClass kde len musí byť na výstupe spúšťacia trieda, tá, ktorá dedí zo spomínanej Application. V tejto beane nesmiem zabudnúť pridať implementáciu pre metódu close, ktorú anotujem ako @Remove, čo zabezpečí jej zrušenie po skončení relácie.

Ďalej je to štandardná Vaadin aplikácia, sada nastavených komponent a pridaných poslucháčov na ich udalosti.
Ďalšou možnosťou ako spravovať systém je prístup cez webové služby, v tomto prípadne je užívateľovi prístupné REST API. Technologicky samotné služby sú klasické Java triedy bez nutnosti dedenia alebo implementácie akéhokoľvek rozhrania. Anotované sú ako stateless beany s pridanou anotáciou @Path(<cesta>) určujúcu časť URL adresy pre prístup k tejto triede. Následne každá metóda aby bola prístupná na volania musí byť anotovaná v základe aspoň troma tipmi anotácii:

- @POST , @GET, @PUT alebo @DELETE v závislosti aká http metóda má byť použitá na volanie danej metódy
- @Path(<cesta>) udáva konrétnu cestu k metóde, url bude tvorené: http:///<url k aplikácii>/<trieda>/<metóda>
- @Produces(<MediaType>), ktorý určuje typ výstupu pre užívateľa. Môže byť XML, JSON, obyčajný text alebo octet-stream.

Je zjavné, že v tomto prípade fungujú webové služby len ako rozhranie k beanam s výkonným kódom. Vždy na vstupe je http požiadavka s pridelenými atribútmi, či už POST alebo GET, a na výstupe sú XML dáta. Užívateľovi stačí programovo vytvárať volania na server, parsovať XML a bez problémov má prístup ako k dátam tak aj ku kontrole zásuvných modulov.

7.3. Zabezpečenie

Pretože prístup k systému by nemal byť úplne voľný, je aplikácia v rámci rôznych spôsobov prístupu zabezpečená.
Zo strany webovej aplikácie sa jedná o jednoduché prihlásenie pomocou loginu a hesla. Následne po korektnom prihlásení je pre užívateľa vygenerovaný kľúč, ktorý sa drží v session počas jeho prihlásenia. Na základe tohto kľúča je vygenerovaný súbor kam aplikácia ukladá logy tohto užívateľa, s tým že záznam o súbore, užívateľovi a čase pripojenia je takisto uložený v databáze. Po odhlásení sa samozrejme session zmaže ale logový súbor ostáva a záznam v databáze slúži k jeho opetovnému prístupu.
Zo strany webových služieb je prístup podobný. Pokiaľ chce užívateľ volať akúkoľvek metódu musí sa najskôr korektne prihlásiť. To vykoná zavolaním metódy /process/login, kam pošle svoj login a heslo. Pokiaľ sa jedná o korektné prihlásenie, systém vráti kľúč (sessionId), ktorý si užívateľ uloží a následne ho používa v rámci systému na potvrdenie svojej totožnosti. Preto aby si systém mohol overiť, že používaný kľúč patrí správne prihlásenému užívateľovi, sám si musí takisto kľúč uložiť aj s údajmi o užívateľovi a to už počas prihlásenia. V stateless beanach, ktoré tvoria webové služby ale nie je možnosť uloženia záznamu do session, takže som alokoval na servery glassfish custom resource, čo je priestor v pamäti servera dostupný pomocou injektovania celej bežiacej aplikácii. V rámci custom resource mám premennú dátového typu java.util.Properties, kde kľúč je vygenerovaný z prihlásenia užívateľa a dáta sú údaje o užívateľovi. Následne užívateľ volá akúkoľvek dostupnú metódu, ktorú potrebuje a ako prvý parameter na vstupe vloží svoj kľúč, pokiaľ tak neurobí alebo vloží kľúč, ktorý nie je platný,

systém vráti oznámenie o nevhodnom kľúči. Na pozadí sa mu samozrejme rovnako ako pre webovú aplikáciu takisto vytvorí súbor pre logovanie. Nakoniec užívateľ zavolá metódu /process/logout so vstupným parametrom, jeho kľúčom, čím sa odhlási zo systému.

7.4. Ukladanie kolekcií a prístup k ním

Proces vloženia alebo inak povedané poslania kolekcie na server v rámci webovej aplikácie je jednoduchý upload proces, pomocou Vaadin komponenty Uploader. Podmienka je aby posielaný súbor bol spakovaný zip súbor. Ten sa následne po úspešnom prijatí na servery rozpakuje, jednotlivé obrázky sa uložia do adresára s názvom kolekcie a do databáze sa vloží o nich informácia. V rámci webových služieb ide o zaslanie súboru prevedeného do reťazca Base64 ako vstupný parameter do procedúry /process/upload. Tento reťazec sa následne dekóduje na pole bytov, pomocou FileOutputStreamu sa uloží na server a ďalej sa s ním pracuje rovnakým spôsobom ako v prvom prípade.
Pre prístup ku kolekciám alebo obrázkom vo webovej aplikácii slúži jednoduchý StreamResource, čo je trieda vo Vaadin API. Tá si nastaví dáta do svojho parametru typu InputStream a následne celý StreamResource pošlem užívateľovi ako otvorenie Vaadin okna, čo v skutočnosti mu pošle dáta so všetkými potrebnými nastaveniami v header. Pre prístup k samotným obrázkom som pripravil ešte samostatný servlet, do ktorého sa vloží GET parameter id obrázku a on potom vráti obrázok aj neprihlásenému užívateľovi. Je tak možné šíriť obrázky pomocou ich URL. Prístup ku kolekcii alebo obrázku v rámci vebových služieb je pomocou metód /process/get<Image alebo Collection> s id daného elementu, na čo sa užívateľovi vráti Base64 reťazec, ktorý si užívateľ prevedie na súbor.
Anotovanie a vyhľadávanie na základe anotácii a názvov
Anotovať jednotlivé objekty (kolekcie alebo obrázky) môžeme cez web jednoduchým vyplnením formuláru a uložením. Tieto informácie sa pošlú do databáze ako je spomenuté v prvej kapitole pri popise dátového modelu. Pokiaľ chceme anotovať cez webové služby použijeme metódu /process/setTag, kde na vstupe vložíme id elementu, názov elementu (obrázok alebo kolekcia) a samotný tag (textový reťazec).
Vyhľadávanie je už jednoduchý proces, ktorý vyhľadá tagy podobné hľadanému, prejde väzobné tabuľky a vráti všetky dostupné záznamy ako pre kolekcie tak aj pre obrázky. To platí pre webovú službu aj pre aplikáciu.

7.5. Správa nových modulov

V rámci riešenia tejto diplomovej práce bolo potrebné navrhnúť celý proces od vývoja, cez nasadenie, spustenie a celkovú správu zásuvných modulov v systéme. Ako bolo spomenuté v popise prídavných modulov, vytvoril som viacero typov exportovaných servís a takisto viacero spôsobov spustenia.
V rámci aplikácie existuje niekoľko procesov a kľúčových aspektov, ktoré zabezpečujú prácu s modulmi.

7.5.1. Hybridná beana

Toto kľúčový prvok modulárnej stránky systému. Je to objekt, ktorý je nasadený na obe behové prostredia, aby bol schopný svoje funkcie poskytovať celej aplikácií a aby zabezpečil jej obojstranné prepojenie. Jedná sa z počiatku o štandardný objekt, bez žiadnej výnimočnej konfigurácie. Jediná podmienka je aby ním implementované rozhranie, ktoré slúži ako prístup k nemu, bolo nasadené na oba kontajnery a aby bola kontrola nad jeho životnými cyklami v kontajneroch. Najskôr sa označí ako singleton beana s príznakom startup, je vyžiadané vytvorenie instancie ihneď po nasadení.

```
@Singleton
@Startup
public class ServiceConnectorBean implements IServiceConnectorLocal{
//...výkonný kód
```

Zdrojový kód 5: deklarovanie ServiceConnectorBeany

V rámci metódy označenej ako postConstruct (metóda spustená počas jej nasadenia) sa získa jej instancia zo sessionContextu Java EE kontajnera, ktorá je následne pomocou rozhrania registrovaná ako servisa do OSGi behového prostredia.

```
@PostConstruct
public void init(){
      BundleContext bc=FrameworkUtil
      .getBundle(IServiceConnectorLocal.class)
      .getBundleContext();
      bc.registerService(
            IServiceConnectorLocal.class.getName(),
            context.getBusinessObject(IServiceConnectorLocal.class,
            null);
}
```

Zdrojový kód 6: registrovanie serviceConnector beany do OSGi kontextu

Rovnako musí byť táto referencia odobratá počas metódy preDestroy, počas rušenia jej instancie, inak by mohlo vzniknúť volanie na referenciu už neexistujúcej instancie. Volanie v rámci enterprise kontextu funguje jednoducho pomocou dependency injection a v rámci OSGi behového prostredia sa získava referencia z bundleContextu, ktorý je získaný pomocou rozhrania hybridnej

beany. Konkrétne toto volanie je realizované v rámci triedy MIISProvider, ktorý je popísaný v jednom z ďalších bodov.

7.5.2. Inštalovanie modulov

Dôležitý proces zabezpečujúci informácie o stave modulov a možnosť rozšírenia databáze počas ich inštalácie. Proces je realizovaný pomocou poslucháčov registrovaných cez MIIS.osgi.core aktivátor na udalosť registrovanie servisy (ServiceEvent.REGISTERED), takže pre každú servisu vytvorí záznam v databázy.
V prvom kroku poslucháč musí implementovať potrebné rozhrania a to pre inštalovanie modulov sa jedná o ServiceListener, ktorý dáva možnosť implementovať metódy serviceChanged so vstupným parametrom ServiceEvent.

```
public void serviceChanged(ServiceEvent event) {
      if(event.getType() == ServiceEvent.REGISTERED){
      BundleContext bundleContext = event.getServiceReference()
      .getBundle().getBundleContext();

      if(bundleContext.getService(event.getServiceReference())
                    .getClass().getSuperclass().getSimpleName()
                    .equals(MainIService.class.getSimpleName()))
      {...
```

Zdrojový kód 7: overenie nadradenej triedy servisy po odchytení jej udalosti

Takže po zavolaní tejto metódy si získame o akú konkrétnu udalosť ide, následne si získame bundleContext, z ktorého ďalej získame konkrétnu servisu, ktorá udalosť spustila a len overíme či sa jedná o implementáciu jednej z troch typov API servís. Následne z bundleContextu vytiahnem konfiguračný XML súbor, ktorý ako XML parsujem a vykonám potrebné operácie:

```
URL url = event.getServiceReference()
            .getBundle().getEntry("miis/db_update.xml");
InputStream is = url.openStream();
...
```

Zdrojový kód 8: získanie db_update.xml z inštalovaného bundlu

Pri procese odinštalovania poslucháč reaguje metódou bundleChanged, implementuje rozhranie BundleListener a mení záznam v databázy na udalosť BundleEvent.UNINSTALLED:

```
public void bundleChanged(BundleEvent event) {
        if(event.getType() == BundleEvent.UNINSTALLED)
        { ...
```

Zdrojový kód 9: odchytenie udalosti odinštalovania servisy

Takto pripravený poslucháči sú zaregistrovaný do OSGi kontextu pomocou aktivátora vo vlastnom bundli a po jeho naštartovaní sú schopný reagovať na dané udalosti, rovnako ako sú pridaný do metódy stop, aby boli z daného kontextu zmazaný:

```
public void start(BundleContext context) throws Exception {
        context.addBundleListener(this);
        context.addServiceListener(this);
}
public void stop(BundleContext context) throws Exception {
        context.removeServiceListener(this);
        context.removeBundleListener(this);
 }
```

Zdrojový kód 10: registrovanie poslucháčov pre OSGi kontext

7.5.3. Registrovanie servís

Proces, pri ktorom je instancia servisy počas inštalovania modulu zaregistrovaná do kontajnera a pripravená na použitie. Vykonáva sa v BundleActivator triede, ktorá musí byť obsiahnutá v každom zásuvnom moduli (OSGi bundle). Samotné registrovanie je len vloženie novej instancie servisy, názvu abstraktnej triedy, ktorú rozširuje a prídavné parametre, ktoré zatiaľ systém nevyužíva, takže ostanú ako null.

```
public void start(BundleContext context) throws Exception {
      context.registerService(ProcessingService.class.getName(),
      new DCT_Service(), null);
      context.registerService(WorkerService.class.getName(),
      new DCT_Worker(), null);
}
```

Zdrojový kód 11: registrovanie servís zo zásuvného modulu

Takisto v tejto triede musí byť implementované vymazanie servisy z kontajnera počas odinštalovania modulu. Prevedie sa podobne v metóde stop, ale s tým rozdielom, aby nebolo nutné

držať jej instanciu, je pripravené API, ktoré jej referenciu zmaže. Volanie daného API je potom nasledovné:

```
public void stop(BundleContext context) throws Exception {
    mMService.uninstall(context, new MMService());
    mMService.uninstall(context, new MMSearch());
}
```

Zdrojový kód 12: odinštalovanie servís z OSGi kontextu

7.5.4. Vrátenie instancie servisy

Podľa toho čo aktuálne užívateľ potrebuje, siaha sa do bundleContextu, ktorý sa získa pomocou knižnice FrameworkUtil na základe bundlu exportovaného rozhrania. Pomocou metódy bundleContextu getAllServiceReferences so vstupným parametrom, ktorý je názvom abstraktnej triedy sa získa pole referencií na dané servisy. Získajú sa servisy typu process a searchEngine. Následne ich názvy porovnáme s názvom požadovanej servisy získanej z databáze pomocou vstupného procesId, čo je identifikátor do tabuľky CoreResource. pod ktorou je servisa registrovaná. Pretože ich ale bude takto registrovaných viac, musia sa referencie preiterovať a pomocou metódy getName() sa získa ta správna. Pokiaľ požaduje užívateľ napríklad instanciu servisy DCT_Service, kód bude vyzerať následovne:

```
public MainIService getInstanceOf(Long processId) {
      ICoreResource coreResource = (ICoreResource)
      daoFactoryLocal.getCoreResourceDao().getById(processId);
      ServiceReference[] references = new ServiceReference[0];
      BundleContext bc = FrameworkUtil
      .getBundle(IServiceConnectorLocal.class).getBundleContext();

      MainIService iService = null;
      try {
            references = bc.getAllServiceReferences(
            ProcessingService.class.getName(), null);
            references = (ServiceReference[]) SumArray(references,
            bc.getAllServiceReferences(SearchEngineService.class.getName(
            ), null));
         } catch (InvalidSyntaxException ex) {
            Logger.getLogger(ServiceConnectorBean.class.getName())
            .log(Level.SEVERE, null, ex);
         }
         for(ServiceReference sr : references){
             MainIService service = (MainIService) bc.getService(sr);
             if(service.getName().equals(coreResource.getServiceName())){
                 iService = (MainIService) bc.getService(sr);
                 break;
             }
         }
      return iService.getInstance();
}
```

Zdrojový kód 13: vrátenie instancie konkrétnej servisy

Nakoniec užívateľ získa referenciu požadovanej servisy. Na rovnakej báze fungujú metódy, ktoré vracajú len určitý druh servís, napríklad ak treba na web vrátiť všetky dostupné servisy na vyhľadávanie.

7.5.5. Asynchrónne volanie

V rámci spracovania viacerých obrázkov naraz bez akejkoľvek závislosti medzi sebou, je možné ich spracovávať pomocou paralelných asynchrónnych procesov. Samotný proces bude naimplementovaný v službe typu worker a ich volanie bude generovať ako už bolo spomenuté servisa typu process. Samotné volanie je vykonané užitím MIISProvider API, ktorý dá prístup k ServiceConnectorBean (hybridná beana, kapitola 7.5.1) a zavolaním jej metódy createTask sa

odošlú parametre udávajúce sessionId – pre prístup k logu, serviceName- udáva meno workera, pre ktorého je volanie a serializovaný objekt, ktorý mu bude daný ako vstupný atribút.

```
//prechodom v cykle sa vytvorí volanie pre každý obrázok kolekcie
for(IPicture picture : collect.getPictures()){
      MIISProvider.getInstance().getServiceConnector()
      .createTask(getSessionId(), "DCT_Worker", picture.getId());
}
```

Zdrojový kód 14: vytvorenie volaní na worker servisu

Následne je pomocou beany toto volanie prerobené na správu určenú do JMS kontextu, konkrétne je pripravená fronta (QUEUE) a connectionaFactory, do ktorej je každá vytvorená správa zaslaná s vnesenými atribútmi.
Vnesené atribúty sú:

- **sessionId:** pre prístup do logovania aktuálne prihláseného užívateľa
- **input:** objekt z procesnej servisy zaslaný na worker servisu
- **service –** konkrétna instancia worker servisy podľa vneseného názvu

V rámci zasielaných dát na asynchrónne volania je pre objektový input podmienka aby daný objekt implementoval rozhranie java.io.serialisable, prípadne priamo zasielať objekt ako serializovaný reťazec. Pri zasielaní neserializovaných objektov ako Stream alebo niektoré typy kolekcií, JAVA EE kontajner vyhodí výnimku.

```
connection = connectionFactory.createConnection();
Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

MessageProducer messageProducer = session.createProducer(queue);
messageProducer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

Message message = session.createTextMessage();
message.setObjectProperty("service", service);
message.setObjectProperty("input", input);
message.setStringProperty("sessionId", sessionId);
messageProducer.send(message);
```

Zdrojový kód 15: vytvorenie správy pre JMS kontext

Na druhej strane JMS je pripravená správami riadená (message driven) beana ImageProcessEngineMDB, ktorá musí implementovať rozhranie MessageListener s nastavenou anotáciou aby JMS kontext vedel, nad ktorým zásobníkom alebo frontov počúva:

```
@MessageDriven(mappedName = "jms/MIISQueue", activationConfig = {
@ActivationConfigProperty(propertyName = "acknowledgeMode",
propertyValue = "Auto-acknowledge"),
@ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")
})
public class ImageProcessEngineMDB implements MessageListener {
...
```

Zdrojový kód 16: deklarácia MDB nad konkrétnym JMS Queue

Samotná beana je volaná na každé pridanie správy do fronty s tým, že správu, ktorú získa, z fronty vyberie. Cez jej property získa názov servisy, na ktorý si vytiahne instanciu a nakoniec zavolá jeho metódu performTask, kde zašle sessionId a vstupné dáta.

```
Object input = message.getObjectProperty("input");
String sessionId = message.getStringProperty("sessionId");
WorkerService service = (WorkerService)
message.getObjectProperty("service");
if(service != null)
            service.performTask(sessionId, input);
```

Zdrojový kód 17: spustenie worker servisy nad konkrétnym vstupom

Dané procesy sú absolútne asynchrónne a paralélne s tým, že každý z nich má možnosť logovania svojho postupu a takisto ukladania výsledkov do súborov.

Podstatnou vecou, ktorú treba opomenúť v rámci paralélneho spracovania je riadenie súbehu, transakčné spracovanie. Užívateľ si musí byť vedomý, že nie je možné ako vstup poslať cestu na jeden súbor pre všetky worker servisy aby doň zapísali výsledky, pretože u niektorých implementácií Streamov nie je možné vytvoriť viacero pripojení na súbor a iné Streamy s možnosťou bufferu v čase zápisu (flush) pravdepodobne vrátia výnimku.

Je možné tomuto predísť pomocou implementácie FileHandler v rámci Logger API, ktorý je schopný pomocou implementácie Singleton drží práve jedno pripojenie na daný súbor pre všetky volania, ale znova v rámci súbehu operácií flush a ďalšieho zápisu do Streamu, vráti výnimku.

Odporúčam aby každý worker ak potrebuje si vytvoril vlastný súbor a cestu k nemu uložil napríklad do logu, ktorý v druhom cykle načíta process servisa a tieto vytvorené súbory ďalej spracuje.

7.5.6. MIISprovider API

Trieda prístupná pre všetky zásuvné moduly. Je súčasťou balíka MIIS.internal.interface a jej volanie je implementované pomocou návrhového vzoru singleton, je snaha aby na servery bola maximálne jedna instancia pre všetky volania. Táto trieda vo finále drží instanciu na hybridnú beanu a čím sme schopný z prídaného procesu volať všetky funkcie java EE kontextu (volanie iných procesov, vytváranie správ, prístup do databáze, prístup do súborového systému, ...).

```
// jednoduchým volaním získam instanciu orm mapovania alebo
// hybridnej beany
MIISProvider.getInstance().getDao()...
MIISProvider.getInstance().getServiceConnector()...
```

Zdrojový kód 18: získanie bean pomocou MIISProvider

Pomocou týchto dostupných prvkov by malo byť jednoduché vytvoriť proces pre obrázky s akýmkoľvek vstupom a akýmkoľvek zložitým procesom.

7.5.7. Vytvorenie modulu

Proces vytvorenia zásuvného modulu začína zostavením si java projektu pomocou maven. Je spolu s knižnicami dostupný aj základný maven popisný súbor (pom.xml), ktorý by mal vygenerovať kompletnú štruktúru projektu. Ďalej je potrebné skontrolovať prítomnosť knižníc ako MIIS.domains, MIIS.internal.interface, javaee.api a org.osgi.core. Sú ale dané v popisnom súbore mavenu takže by nemali chýbať. Ďalej podľa potreby a v súlade s filozofiou platformy vytvoríme servisnú triedu, ktorá bude rozširovať jednu z nasledujúcich abstraktných tried:

Miis.common.abst.WorkerService.java pre servisu typu worker s metódou performTask pre implementovanie procesu. Je nutné si uvedomiť, že proces tohto typu je možné volať len ako paralélny asynchrónny proces, pomocou volania inej servisy.

Miis.common.abst.ProcessingService.java je trieda pre štandardnú procesnú servisu, ktorá môže byť priamo volaná užívateľom cez dostupné rozhrania. Vykonávaný process by mal byť implementovaný alebo spúšťaný z metódy process().

Miis.common.abst.SearchEngineService.java je posledný typ servisy, a slúži na rozšírenie možností vyhľadávania medzi obrázkami. Dané vyhľadávanie je naimplementované v metóde compare(). Volaná je takisto priamo užívateľom cez dostupné rozhrania.

Pre všetky typi servís je nutné naimplementovať metódu getName(), ktorá vráti unikátny názov servisy, pod ktorou bude uložená a getInstance, pomocou ktorej systém získa novú instanciu

objektu. Takisto pre servisy platí možnosť prístupu na MIISProvider API spomenutý v predošlom kontexte.
Nakoniec sa samotné servisy musia zaregistrovať v aktivačnej triede ako bolo predošle vysvetlené a proces samotného nasadenia modulu je pomocou webového rozhrania jednoduchý upload rovnako ako pomocou REST API poslanie jar balíku ako base64 reťazec na metódu /process/deploy.
Počas samotnej inštalácie modulu, máme možnosť pridať do databáze vlastnú tabuľku, prípadne zaregistrovať parametre, ktoré budú naplniteľné cez webové rozhranie. Toto sa vykonáva pomocou do modulu pridaného xml súboru, ktorý musí byť umiestnený v src/main/resources/miis/ s názvom db_update a typu .xml. Celkový formát musí byť napríklad nasledovný:

```
<MMService>
      <script>
      create table MatrixMarket_Data (ID integer, Coll_id integer,
      Path VARCHAR(20), Name Varchar(50))
      </script>
      <attribute>file_path</attribute>
</MMService>
```

Zdrojový kód 19: príklad pre db_update.xml

Takýto XML skript získa a spustí práve poslucháč na OSGi kontajnery popísaný v kapitole XY. Tabuľku nechá vytvoriť pomocou Hibernate a daný atribút len uloží do tabuľky Attributes s cudzím kľúčom na daný modul.
Keďže beana s implementáciu ORM mapovania nie je schopná sama namapovať nami pridanú tabuľku, budeme k nej pristupovať sami pomocou dostupného API napríklad pre vloženie záznamu:

```
dao.executeSQL(""
                + "INSERT INTO MATRIXMARKET_DATA "
                + "(coll_id, path, name) "
                + "VALUES
("+collect.getId()+",'"+of.getPath()+"', '"+of.getName()+"')");
```

Zdrojový kód 20: vloženie záznamu do pridanej tabuľky

Alebo pre jeho výber záznamu:

```
ICollect collect = (ICollect)
MIISProvider.getInstance().getDao().getCollectDao().getById(co
llectionId);
String sql = "Select * from matrixmarket_data where coll_id =
" + collect.getId();
HashMap<String, Integer> scalars = new HashMap<String,
Integer>();
scalars.put("coll_id", MIISProvider.TYPE_INTEGER);
scalars.put("id", MIISProvider.TYPE_LONG);
scalars.put("name", MIISProvider.TYPE_STRING);
scalars.put("path", MIISProvider.TYPE_STRING);

List results = dao.executeSql(sql, MatrixMarketData.class,
scalars);
```

Zdrojový kód 21: získanie záznamu pomocou HIBERNATE scalar API

Tabuľka sa vytvrí presne jeden krát, počas inštalácie modulu. Pokiaľ je skript spustený a prebehne korektne, v databáze sa uloží atribút záznamu daného modulu installed na true a pri opätovnej inštalácii modulu skript hľadaný nebude.

8 Nasadenie systému

Samotné nasadenie na server je proces kedy nad prvotnou inštaláciou prevedieme sadu nastavení, vnesieme pridané balíky a prevedieme jednotlivo deploy základných modulov. Celkový čas nasadenia nasadenia nebude viac ako pár minút.

8.1. Server a systém

V rámci serverových nastavení vytvoríme na servery v JMS resources connection factory pod názvom (pool name) jms/MIISConnectionFactory s resource type javax.jms.ConnectionFactory. Nastavenie poolu, počet minimálne a maximálne alokovaných pripojení a ostatné nastavenia ďalej zadáme podľa potreby, tieto nastavenia sa vzťahujú priamo na našu connection factory a je možné ich dodatočne meniť. Ďalej vytvoríme destination resource s názvom jms/MIISQueue a resource typom javax.jms.Queue.

V ďalšom kroku otvoríme projekt MIIS.dao.core a v triede miis.dao.HibernateUtils nastaviť parametre databáze, na ktorú ma byť systém nasadený. Následne modul zbuildime pomocou maven a pokračujeme postupným nasadzovaním všetkých modulov.

Najskôr vnesieme do bežiaceho systému potrebné knižnice a balíky rozhraní. Do adresára /<názov domény>/autodeploy/bundles nakopírujeme :

- internal.interfaces-1.0-SNAPSHOT.jar
- javaee-api-6.0.jar
- MIIS.domains-1.0-SNAPSHOT.jar
- org.apache.felix.framework-4.0.3.jar

Tento proces je ale možné urobiť automatizovane a to pokiaľ v maven konfigurácii (pom.xml) zmeníme cestu na náš server (riadok 64. Tag outputDirectory). Ten v rámci fáze package skopíruje tieto knižnice s rozhraniami na definovaný server sám. Je potrebné aby sa tam tieto knižnice dostali z celého projektu ako prvé, pretože ak nie sú na servery požadované rozhrania a pokúsime sa nasadiť nejakú servisu, server vráti osgi.wiring exception a nasadenie nebude úspešné.

V ďalšom kroku prevedieme nasadenie modulu dao.core, v ktorom znova na riadku 125 zmeníme outputDirectory alebo jednoducho skopírujeme do adresára /<názov domény>/lib balík rozhraní MIIS.domains-1.0-SNAPSHOT.jar. Pred nasadením modulu dao.core musíme mať vytvorenú databázu a nahraný správny ovládač. V skutočnosti nezáleží o akú databázu pôjde, pokiaľ bude na ňu dostupný Hibernate dialect. Do triedy miis.dao.HibernateUtils vnesieme nastavenia pre pripojenie do databáze a celý modul zbuildíme. Následne na to môžeme nasadiť samotný modul dao.core na server. Pokiaľ sa modul správne inicializuje a pripojí tak sa nám na servery vytvorí beana DaoFactory a takisto Hibernate vytvorí databázové tabuľky na základe entitného modelu aplikácie.

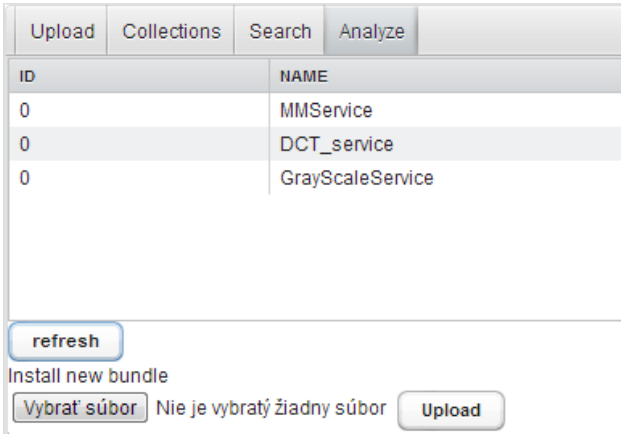
Ďalšou fázou je nasadenie modulu ejb.core. Pokiaľ je všetko správne nasadené, tento modul inicializuje dve beany a pripraví poslucháča pre JMS kontext.

Nakoniec len skopírujeme balík MIIS.osgi.core-1.0-SNAPSHOT .jar do adresáru autodeploy/bundles čím zaregistrujeme poslucháča pre OSGi kontajner.

Tým máme hotový deploy základných modulov. Pre prácu so systémom potrebujeme ale aj rozhrania, takže nasadíme Web-1.0-SNAPSHOT.war pre Vaadin webovú aplikáciu a rest.services-1.0-SNAPSHOT.war pre restové api.
Celý proces nasadenia je mierne komplikovanejší a pri úpravách základných modulov je potrebné proces pre reštart servera opakovať, preto doporučujem využívať automatizované nasadzovanie podporých knižníc pomocou maven procesov, definovaných v príslušných pom.xml.

8.2. Zásuvné moduly

Jednotlivé zásuvné moduly je možné nasadiť tromi spôsobmi. Pretože sa už jedná o proces spadajúci pod funkcionalitu systému, je možné modul nasadiť cez webové rozhranie na karte Analyze po uploade modulu. Následným obnovením obsahu tabuľky pomocou tlačidla refresh, zistíme či sa náš modul nainštaloval správne a servisa je dostupná. Ďalšou možnosťou je modul poslať pomocou nami vytvoreného klienta na REST API do metódy process/deploy, kde modul bude zaslaný ako base64 reťazec a API metódou nasadený. A posledná tretia možnosť ako pripojiť modul na aplikáciu je vytvoriť si FTP pripojenie na adresár autodeploy/bundles na servery a daný modul tam jednoducho nakopírovať. Systém zistí zmenu automaticky a daný modul nainštaluje. Nasadil som moduly pre DCT spracovanie, modul pre prevedenie farebných obrázkov do odtieňov šedi a modul na predstpracovanie kolekcie do matrix- market formátu s následným vyhľadávaním. Celé nasadenie bolo cez webové rozhranie, rýchle, plynulé a na pozadí bola vytvorená databázová tabuľka MM_data, ktorú si vytvoril jeden z modulov pre ukladanie informácii o matrix market súboroch.



Obrázok 9: dostupné servisy

9 Testovanie

9.1. Úvod

Ako sa systém reálne správa a aké má možnosti bude mojím cieľom popísať v nasledujúcej kapitole. Od samotného nasadenia systému na server popíšem základnú funkcionalitu, rozšírenia, použité dáta na testovanie, testované moduly a samozrejme aj výsledky.
Testovaný bude samotný systém nasadený na servery glassfish (popísaný v kapitole XX), na počítači s procesorom Intel Core 2 Duo P8400, s operačnou pamäťou DDR3 o veľkosti 3GB a operačným systémom Windows 7.
Ako testované dáta poslúžia tri náhodné kolekcie obrázkov, ktorých veľkosti a počty obrázkov sú v nasledujúcej tabuľke.

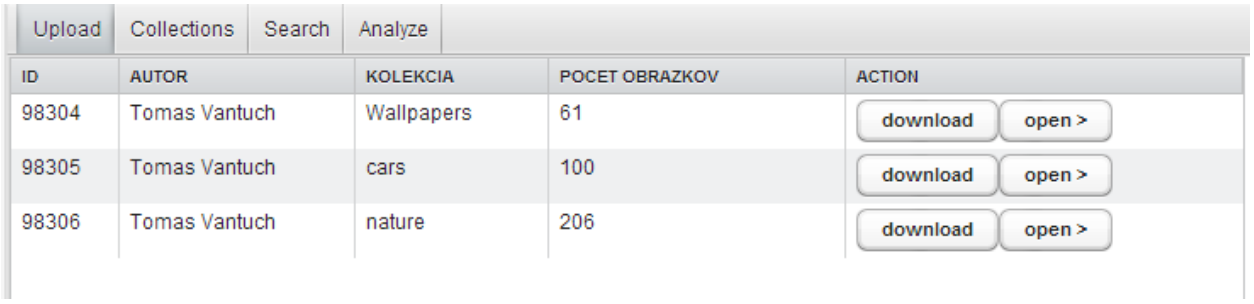
Názov	veľkosť	Počet obrázkov	Rozlíšenie obrázkov
Wallpapers (kolekcia 1)	23,4 MB	60	1280 x 1024
Cars (kolekcia 2)	60 MB	100	1280 x 1024
Nature (kolekcia 3)	46,6 MB	102	1280 x 1024

Tabuľka 2: kolekcie použité na testovanie

Obrázky sú farebné vo formáte JPEG a jednotlivé kolekcie sú zbalené vo formáte ZIP, v ktorom budú posielané do aplikácie na server.

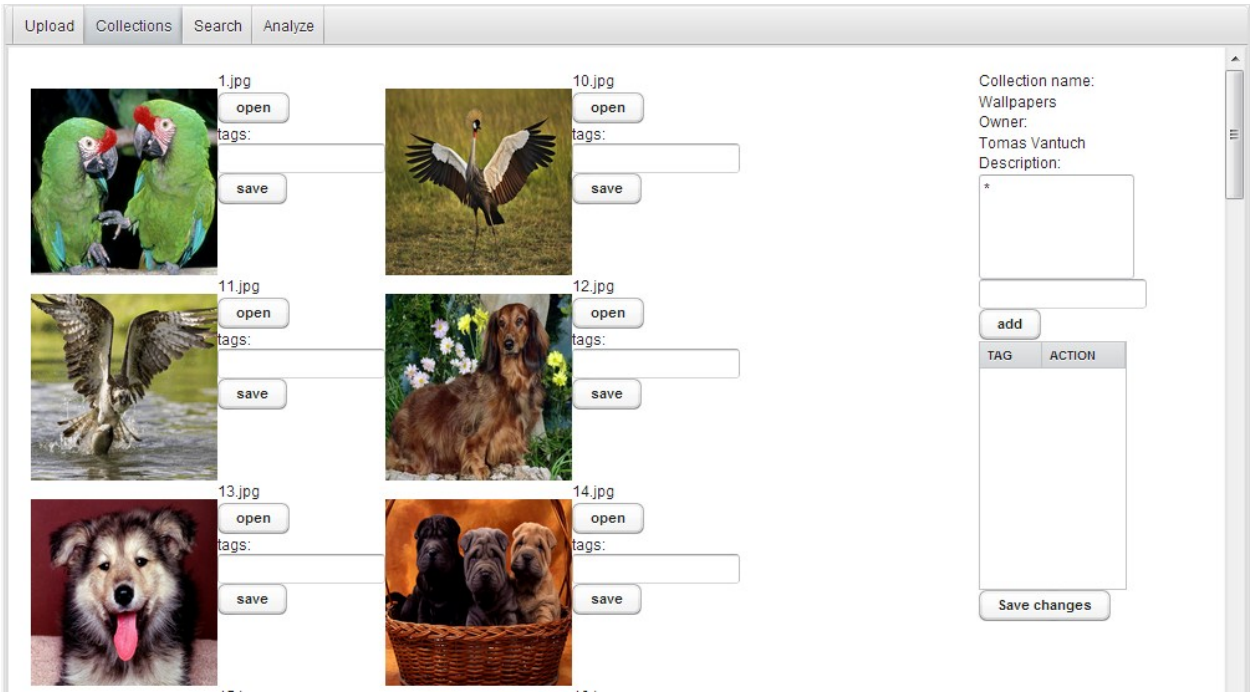
9.2. Základná práca s kolekciami

Následne keď je systém spustený a moduly nasadené, pošleme kolekcie obrázkov na server. Proces zaslania je pomocou karty webu upload, kde posielame ZIP archív kolekcie. Systém ZIP archív prijal a spracoval ako bolo definované v kapitole 7.4 a poskytol ďalšie možnosti ako je samotný náhľad do kolekcie, prechádzanie jednotlivými obrázkami či pridanie popisu kolekcii alebo tagovanie jednotlivých obrázkov.



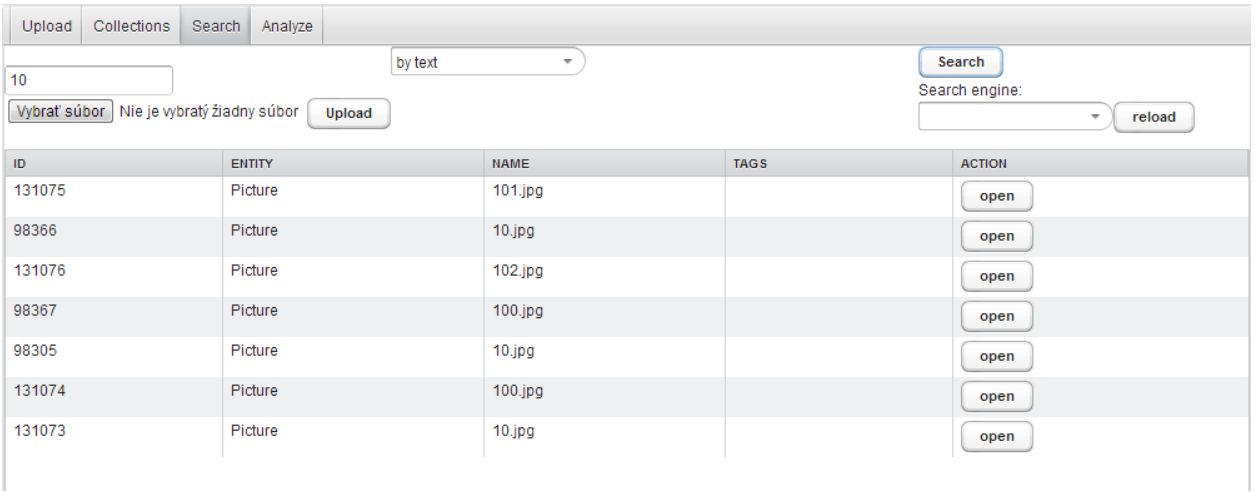
Obrázok 10: zoznam kolekcií na servery s možnosťou stiahnutia alebo náhľadu

Na tejto karte pomocou stiahnutia klient obdrží ZIP archív danej kolekcie. Po stlačení tlačidla open systém prepne kartu na Collections a zobrazí jednotlivé obrázky s ich tagmi či informácie o kolekcii viz. Obrázok 11. Tlačidlom open pri obrázku sa spustí servlet s vneseným od obrázku, takže sa otvorí okno webového prehliadača s obrázkom v jeho rozlíšení.



Obrázok 11: náhľad detailu kolekcie

Karta s vyhľadávaním poskytuje možnosť vyhľadávania pomocou textu alebo pomocou iného obrázku. V rámci vyhľadávania pomocou textu sa prehľadávajú názvy obrázkov a kolekcií, ich tagy a aj popisy kolekcií. V množine výstupov môžu byť ako obrázky tak aj celé kolekcie. Práve stĺpec ENTITY umožňuje rozlíšiť o aký výstup sa jedná. V prípade kolekcie máme možnosť výsledok stiahnuť, v prípade obrázku sme schopný ho znova zobraziť v plnom rozlíšení. V druhom prípade vyhľadávania pomocou obrázku musíme ďalej dodefinovať search engine, metódu vyhľadávania. V takom prípade sa vyhľadáva pomocou pravidiel implementovaných v konkrétne vybranej servise zásuvného modulu.



Obrázok 12: vyhľadávanie, na vstup “10” vráti výsledky obsahujúce v názve tento reťazec

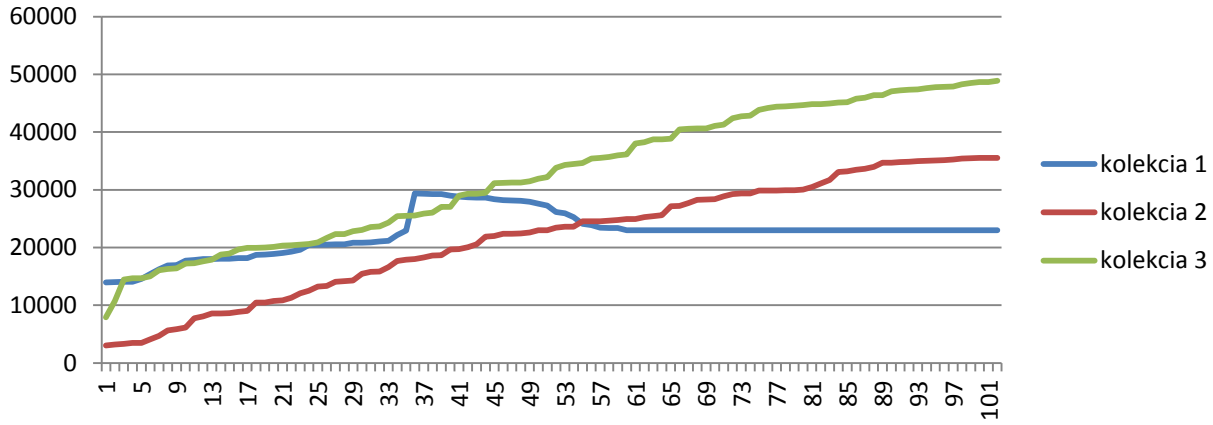
Nakoniec ostáva samotné použitie zásuvných modulov. Najskôr na karte Upload označíme kolekciu, s ktorou chceme pracovať, označenie nás prehodí na kartu Analyze, kde budú v tabuľke dostupné servisy na spracovanie. Označíme požadovanú servisu a tlačidlom Run spustíme proces. Beh procesu si overíme nahliadnutím do logu pomocou tlačidla Open Log, ktoré nám otvorí servletom stránku, kde sa vypíše aktuálny obsah nášho logu.

9.3. Predspracovanie pomocou DCT

Tento proces bol prevedený nad všetkými troma kolekciami. Implementácia modulu je popísaná v kapitole 6.2, takže tu sa skôr zameriam na výsledky.
Do logu boli zapisované časy behu jednotlivých worker servís. Pretože sa jedná o asynchrónne procesy, jednotlivé vracané časy neboli zoradené a tak celkový beh nad celou kolekciou je ten najvyšší vrátený čas. Do týchto časov je zohľadnené aj samotné inicializovanie správ a message driven bean, pretože začiatok času bol zadaný process service a ukončenie má každá worker service samostatne.

Celkový čas behu	
Kolekcia 1	29373 ms
Kolekcia 2	35513 ms
Kolekcia 3	48893 ms
Priemerná dĺžka spracovania obrázku	
Kolekcia 1	489,55 ms
Kolekcia 2	355,13 ms
Kolekcia 3	479,34 ms
Zníženie veľkosti kolekcie pomocou kompresie	
Kolekcia 1	77,2 %
Kolekcia 2	90,5 %
Kolekcia 3	84,2 %

Tabuľka 3: výsledky po DCT



Graf 1: výsledky po DCT

Výsledkom boli kópie obrázkov v rovnakom rozlíšení ale s podstatne nižšou veľkosťou.

9.4. Predspracovanie pomocou MM

Ďalším modulom som otestoval preuloženie kolekcie do súboru typu matrix market ako je popísane v kapitole 6.2. Výsledné súbory boli zapisované na disk a časy boli vypísané do konzole.

Celkový čas behu	
Kolekcia 1	15628 ms
Kolekcia 2	49668 ms
Kolekcia 3	39510 ms
Veľkosť výsledného mm súboru	
Kolekcia 1	355 MB
Kolekcia 2	925 MB
Kolekcia 3	714 MB

Tabuľka 4: výsledky po MM predspracovaní

Následne nad predspracovanými dátami bolo vykonané vyhľadávanie. Boli vybrané dva náhodné obrázky z daných kolekcii a pokúsil som sa nájsť k ním zhodné. V oboch prípadoch sa daný obrázok našiel, jednotlivé časy však boli mierne odlišné. Výsledky vyhľadávania:

Načítanie súboru [ms]	Porovnávaný obrázok 1	Porovnávaný obrázok 2
Kolekcia 1	57300	52613
Kolekcia 2	144847	135535
Kolekcia 3	105405	114262
Vyhľadávanie [ms]		
Kolekcia 1	54	112
Kolekcia 2	65	105
Kolekcia 3	105	61

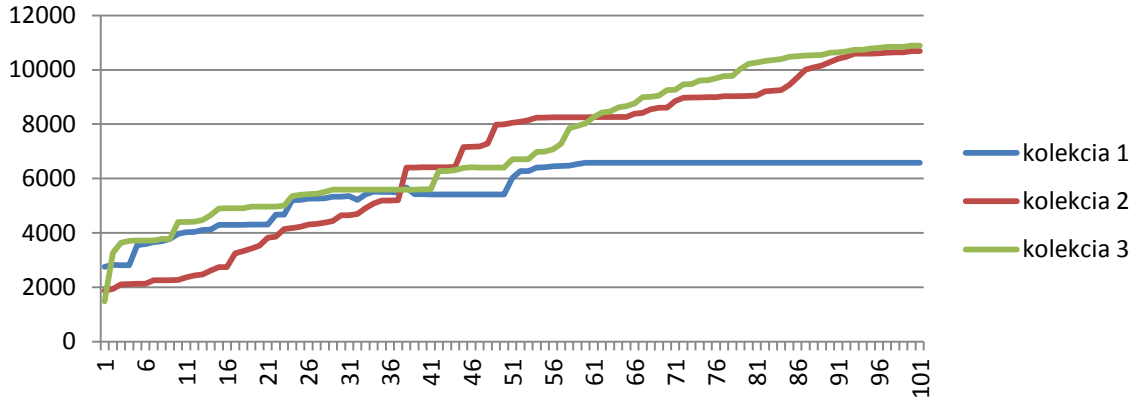
Tabuľka 5: výsledky po MM vyhľadávaní

9.5. Úprava prevodom do grayscale

Posledný testovaný modul mal jednoduchú úlohu. Previesť všetky obrázky do odtieňov šedi a uložiť ich do adresáru kolekcie. Výsledkom boli čiernobiele kópie všetkých obrázkov kolekcií.

Obrázok 13: výsledok po prevode do odtieňov šedi

Pretože sa znova jednalo o asynchrónne procesy, ktoré samostatne obrázok po obrázku spracovávali, aj jednotlivé výsledné časy spracovania sú ako v prvom prípade (DCT) nezoradené:



Graf 2: časové výsledky prechodu do odtieňov šedi

Celkový čas behu	
Kolekcia 1	6577 ms
Kolekcia 2	10683 ms
Kolekcia 3	10948 ms
Priemerná dĺžka spracovania obrázku	
Kolekcia 1	107,81 ms
Kolekcia 2	106,83 ms
Kolekcia 3	107,33 ms

Tabuľka 6: časové výsledky prechodu do odtieňov šedi

9.6. Prístup k výsledkom

V závislosti na tom aké výsledky očakávame máme k nim rôzny prístup. Niektoré výsledky boli uložené na server ako súbory, je k nim prístup pomocou FTP. Iné záznamy boli vypísané do logu, či uložené do databáze pre ďalšie procesy. Výsledky z vyhľadávania boli jednoducho pomocou API vrátené aplikáciou priamo na web.

9.7. Záver testovania

Výsledkom tohto testu bolo overenie funkcionality systému po všetkých základných aj momentálne pridaných funkcionalít. Ťažko hodnotiť jednotlivé časy behu modulov, keďže rýchlosť ich behu bola limitovaná starším typom hardwaru a takisto implementačne sa jednalo o jednoduché ukážky práce s API aplikácie. Pokiaľ by šlo skutočne o to naimplementovať čo najefektívnejšie a najrýchlejšie riešenie, bolo by vhodné v takom prípade napísať ho v natívnom kóde a ten celý zbaliť do modulu. Toto testovanie slúžilo ako demonštrácia funkcionality systému. Aby systém reálne zvládol spracovať veľké kolekcie obrázkov, bude nutné serveru nastaviť viac alokovanej operačnej pamäte a zväčšiť connection pool pre JMS kontext.

Záver

Výsledný projekt je modulárnou aplikáciou bežiacou na aplikačnom servery s viacerými možnosťami prístupu. Projekt je takisto veľmi jednoduché rozšíriť a nasadenie modulu sledovať v reálnom čase. Výsledky jednotlivých testov sú podriadené hardwaru a implementácii jednotlivých ukážkových modulov ako je spomenuté v predošlej kapitole.

V tomto záverečnom zhodnotení by bolo ešte dobré uviesť, že napriek tomu, že hotový projekt spĺňa požiadavky zadania, jedná sa o prototypové riešenie, ktoré do budúcna bude vhodné ďalej rozširovať, prípadne upravovať a to nielen pomocou zásuvných modulov ale aj úpravami či už nastavením servera alebo úpravou samotných hlavných modulov.

Jednou z prvých možností rozšírenia je vytvorenie desktopového klienta komunikujúceho pomocou REST API a slúžiaceho ako testovacie prostredie pre zásuvný modul, predtým ako bude plne funkčný a nasaditeľný na server. V tejto súvislosti by bolo vhodné niektoré metódy REST API doimplementovať, pretože na kompletnú komunikáciu modulu s jadrom systému neboli pre ich momentálnu potrebu implementované.

Ďalej by bolo veľmi prospešné rozšíriť vnútorné API o možnosť pridania poslucháčov na niektoré udalosti ako napríklad upload alebo download kolekcie, čím by sa získalo automatizované predspracovanie pomocou volaných zásuvných modulov.

Samozrejme využitie spojenia použitých technológii poskytuje možnosť vytvoriť projekt prospešný aj v iných oblastiach ako len pre digitálne spracovanie obrazu.

Celkovo tento projekt pre seba hodnotím ako oboznámenie sa s použitím moderných frameworkov s využitím modulárneho prístupu v rámci programovania a oboznámenie sa s problematikou digitálneho spracovania obrazu.

Referencie:

[1] *Matrix Market Exchange Formats*. (2007). Retrieved 2013, from http://math.nist.gov/MatrixMarket/formats.html#MMformat

[2] *Book of Vaadin 7 edition.* (2012).

[3] *GlassFish Project – Documentation Home Page*. (2012). Retrieved Február 15, 2013, from https://glassfish.java.net/docs/project.html

[4] *Hibernate documentation*. (2012). Retrieved Marec 7, 2013, from http://docs.jboss.org/hibernate/orm/4.1/quickstart/en-US/html/

[5] *Apache Derby*. (2013). Retrieved Apríl 23, 2013, from http://db.apache.org/derby/

[6] *Apache Maven*. (2013). Retrieved Apríl 26, 2013, from http://maven.apache.org/guides/getting-started/index.html

[7] Johnson, S. (2006). *Stephen Johnson on Digital Photography.* O'Reilly Media, Incorporated.

[8] Khayam, S. A. (2003). *The Discrete Cosine Transform (DCT): Theory and Application.* Michigan State University.

[9] M.Reese, R. (2011). *EJB 3.1 Cookbook.*

[10] Richard S. Hall, K. P. (2011). *OSGi in Action.* Manning.

[11] Schach, S. R. (n.d.). *Object-Oriented and Classical Software Engineering, Eighth Edition.* Vanderbilt University.

[12] Johnson, S. (2006). *Stephen Johnson on Digital Photography.* O'Reilly Media, Incorporated.

Zoznam obrázkov

Zoznam tabuliek

Zoznam grafov

Zoznam zdrojový kódov

Elektronické prílohy

- Diploma_van431.pdf : text diplomovej práce
- MIIS : adresár projektu
    - core.bundle : projekt pre miis.core.bundle modul
    - fileTransfer.EJB: projekt pre miis.ejb.core modul
    - MIIS.bundle.grayscale.service : projekt pre miis.grayscale.bundle modul
    - MIIS.bundle.resize.serive : projekt pre miis.resize.bundle modul
    - MIIS.dao.hibernate : projekt pre miis.dao.core modul
    - MIIS.domains : projekt pre miis.domains modul
    - MIIS.internal.EJB.interface : projekt pre miis.internal modul
    - MIIS.web.vaadin : projekt pre miis.web modul
    - mmBundle : projekt pre miis.mm.bundle modul
    - rest.services : projekt pre miis.rest modul