VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

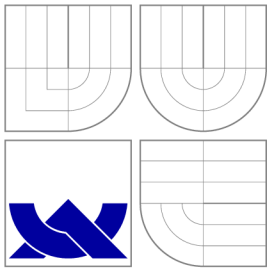
VYSOCE DOSTUPNÝ ŠKÁLOVATELNÝ CMS V PROSTŘEDÍ JAVA EE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

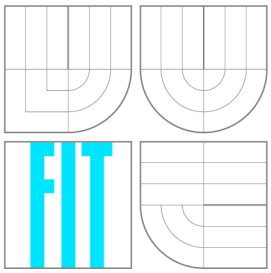
AUTOR PRÁCE
AUTHOR

Bc. SAMUEL ŠRAMKO

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYSOCE DOSTUPNÝ ŠKÁLOVATELNÝ CMS V PROSTŘEDÍ JAVA EE

HIGHLY AVAILABLE SCALABLE CMS IN THE JAVA EE ENVIRONMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. SAMUEL ŠRAMKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2013

Abstrakt

Tato práce se věnuje studiu předpokladů pro návrh dostupného, škálovatelného a modulárního CMS systému na platformě Java EE s využitím OSGi frameworku a implementaci navrženého systému. Popisuje návrh a řešení rozdělení aplikace do jednotlivých modulů, jejich vzájemnou komunikaci a provázání. Na závěr prezentuje výsledky testování aplikace a předkládá možná rozšíření aplikace.

Abstract

This thesis deals with the background of the design of a highly available, scalable and modular content management system based on the Java EE platform and the OSGi framework and with the implementation of the designed system. It describes the design and implementation of the application decomposition to modules, their communication and bindings. Finally, it presents the results of the application testing and proposes available extensions of the application.

Klíčová slova

Java EE, OSGi, MVC, CMS, MongoDB, SOA, bundle, Spring, Virgo Tomcat Server, Equinox, load balancing

Keywords

Java EE, OSGi, MVC, CMS, MongoDB, SOA, bundle, Spring, Virgo Tomcat Server, Equinox, load balancing

Citace

Samuel Šramko: Vysoce dostupný škálovatelný CMS v prostředí Java EE, diplomová práce, Brno, FIT VUT v Brně, 2013

Vysoce dostupný škálovatelný CMS v prostředí Java EE

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D.

.....
Samuel Šramko
16. května 2013

© Samuel Šramko, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Cieľom diplomovej práce je navrhnúť a vytvoriť systém pre správu vzdialeného obsahu. Na navrhovaný systém sú kladené 3 základné požiadavky:

- vysoká dostupnosť – systém by mal byť neustále dostupný pre svojich klientov
- škálovateľnosť – systém by malo byť jednoducho možné nasadiť na niekoľko serverov pre zvýšenie a optimalizáciu výkonu
- modularita – systém by sa mal skladať z niekoľkých menších, ľahko zameniteľných častí

V nasledujúcich kapitolách postupne vysvetlím základné pojmy, s ktorými budem pracovať. Okrem pojmov predstavím súčasné riešenie Ki-Wi Server firmy Ki-Wi Digital, s.r.o., ktorý v tejto práci navrhujem nanovo, oboznámim čitateľa s možnosťou využitia OSGi frameworku pri návrhu tohto systému a zhrniem výhody, ktoré má nový návrh oproti súčasnému riešeniu. V ďalšej kapitole predstavím návrh tohto systému vrámci OSGi frameworku, detailne popíšem jednotlivé bundles a funkcionalitu, ktorú musí navrhované riešenie spĺňať. V časti implementácia bude predstavená implementačná časť aplikácie, postupy práce, prípadne problémy, ktoré pri vývoji nastali. Na záver zhrniem dosiahnuté výsledky a naznačím ďalšie pokračovanie projektu v budúcnosti.

Kapitola 2

Prehľad základných pojmov a použitých technológií

V tejto kapitole zoznámim čitateľa so základnými teoretickými predpokladmi pre návrh a implementáciu požadovaného CMS systému. Vysvetlím jednotlivé pojmy, porovnám možné aplikačné frameworky, aplikačné servre a komunikačné systémy použité pre komunikáciu CMS so svojimi klientmi.

2.1 Systém pre správu obsahu

Systém pre správu obsahu, v angličtine *Content Management System*, je obecne systém, ktorý sa používa na správu rôzneho druhu digitálneho obsahu, ako sú napríklad obrázky, videá, dokumenty atď. Nad každým z rôznych typov obsahu je poskytovaná základná množina operácii označovaná ako CRUD, ktorá reprezentuje vytvorenie – z anglického *create*, čítanie – *read*, zmenu – *update* a odstránenie – *delete*. Okrem tejto minimálnej požadovanej množiny operácií sú dostupné ešte operácie špecifické pre jednotlivé typy dát.

Cieľom systému pre správu obsahu je poskytnúť svojím užívateľom prístup k obsahu z ktoréhokoľvek miesta a súčasnú prácu viacerých užívateľov so spravovaným obsahom. CMS je preto vhodné použiť napríklad ako systém pre správu/archiváciu podnikových dokumentov.

2.1.1 Ki-Wi Server ako systém pre správu obsahu

V tejto časti zoznámim čitateľa s Ki-Wi Serverom, jeho jednotlivými časťami, obsahom a štruktúrou systému.

Ki-Wi Server je proprietárna aplikácia firmy Ki-Wi Digital, s.r.o., ktorá má za úlohu spravovať digitálny obsah svojich klientov a rozposielať ho na vzdialené digitálne panely a interaktívne zariadenia v reálnom čase. Digitálne panely a všetky ostatné zariadenia pripojené a komunikujúce so serverom nazvem súhrnne klientmi. Každý klient sa pomocou rôznych komunikačných protokolov pýta Ki-Wi Servera na svoj aktuálny obsah. Obsah pre každého z klientov nahrávajú do systému užívatelia.

Obsah

Každý jeden digitálny objekt sa obecne nazýva médium. Jedno médium môže byť súčasne nastavené ako obsah pre rôzny počet klientov. Médium môže byť:

- obrázok – jpg, gif, png, bmp
- video – avi, mkv, mpeg, wmv
- audio – mp3, wma
- externé médium – odkaz na webovú stránku
- pdf dokument
- prezentácia vo formáte ppt
- komponenta – zip archív so špecifickou štruktúrou
- flash video – flv, swf

Médiá sa môžu organizovať do playlistov, prípadne šablón. Playlist môžeme chápať rovnako ako v hudobných aplikáciách, teda zoznam médií, ktoré sa prehrávajú jedno po druhom. Šablónou rozumieme rozloženie jednotlivých médií na výslednú zobrazovaciu plochu a ich paralelné prehrávanie. Šablóna môže okrem médií obsahovať iné playlisty, nemôže však obsahovať inú šablónu. Playlist môže obsahovať médiá, šablóny a iné playlisty, nemôže však obsahovať sám seba, pretože by vznikla kruhová závislosť.

Klienti

Ki-Wi Server pod pojmom klient rozumie zariadenie, ktoré komunikuje pomocou komunikačného kanálu so serverom a formou požiadavka - odpoveď si zisťuje informácie potrebné k svojmu fungovaniu. Ki-Wi Server podporuje viacero typov zariadení, ktoré s ním môžu komunikovať, a to:

- Player – digitálny panel, ktorý prehráva statický obsah, nie je možná interakcia so zákazníkom
- Kiosk – vonkajšia obrazovka, ktorá zobrazuje požadované informácie, interakcia so zákazníkom je možná
- Interactive – vnútorná obdoba Kiosku

Každý z týchto typov zariadení sa líši vo funkcionalite podľa zakúpenej licencie. Pri zariadení typu *Player* je ešte delenie na *Flow player* a *.NET player*. Tieto dva typy sú podobné vzhľadom na funkcionalitu, ale *Flow player* je staršia verzia napísaná v JavaScripte a je nutné dávať pozor na spätnú kompatibilitu pri novom vývoji.

Organizácie a užívatelia

Užívatelia oprávnení pristupovať k aplikácii, vytvárať nové médiá, playlisty či šablóny a nastavovať obsah jednotlivým klientom, sú združovaní v organizáciách. Organizácia reprezentuje jednu pobočku, predajňu alebo obecne zákazníka. Organizácie sa môžu združovať v spoločnosti. Spoločnosť môžeme chápať ako materskú organizáciu a organizácie ako jej dcéry. Napríklad majme spoločnosť A, ktorá má svoje predajne v rôznych mestách. V každom meste teda existuje organizácia, ktorá spadá pod spoločnosť A. Užívatelia sú združení vždy pod práve jednou organizáciou. Každý užívateľ môže mať definované iné prístupové práva, a teda aj funkcie, ktoré môže vykonávať. Napríklad užívateľ s právami *user* nemôže

nastavovať typ klienta, ale môže mu nastaviť obsah. Práva sú usporiadané hierarchicky, teda užívateľ s vyšším oprávnením môže vykonávať všetko, čo užívateľ s nižším oprávnením, plus ďalšia funkcionalita.

Značky a adresáre

Médiám, playlistom aj šablónam môže užívateľ priradiť značku. Značka je textová informácia, ktorá je zviazaná s organizáciou. Užívateľ si môže pridaním značky bližšie špecifikovať zvolený objekt. Napríklad majme značku *auto* a 5 obrázkov, ktoré obsahujú autá. Užívateľ môže daným obrázkom priradiť značku *auto* a tým bližšie špecifikovať danú skupinu obrázkov. Následne môže napríklad chcieť vytvoriť playlist, ktorý má obsahovat len médiá, ktorých obsahom je auto, a môže podľa značky jednoducho dohľadať požadované médiá. Podobné príklady si vieme predstaviť aj pri playlistoch alebo šablónach.

Médiá a playlisty môžu byť okrem značiek združované v adresároch. Adresáre môžeme chápať rovnako ako v operačnom systéme. Adresáre pri médiách priamo určujú fyzické uloženie daného média na úložisku, pri playlistoch je to len logické usporiadanie playlistov pre uľahčenie práce užívateľovi.

Plánovanie obsahu a riadiace úlohy

Prehrávanie samotného média, prípadne prehrávanie médií v playliste za sebou je pre užívateľa veľmi obmedzujúce. Preto je v systéme prítomné plánovanie obsahu v niekoľkých podobách:

- statické plánovanie obsahu – nastavenie obsahu, ktorý sa má nastaviť na klienta v daný časový okamih
- dynamické plánovanie obsahu playlistu – jednotlivým zložkám playlistu sa nastaví počet opakovaní za minútu a priorita pri plánovaní
- plánovanie médií – vytvoria sa takzvané médiaplány, ktoré určujú, v ktorý deň a v ktorú hodinu sa dané médium môže, resp. nemôže prehrávať

Veľmi často zákazník požaduje, aby boli prehrávače aktívne len v určitých dňoch a hodinách a po zbytok času boli vypnuté a nespotrebovávali zbytočne elektrickú energiu. Pre tento prípad su v Ki-Wi Serveri prítomné riadiace úlohy, ktorými sa dá nastaviť vypnutie daného zariadenia v určitú časovú hodinu a jeho opätovné zapnutie v nastavený čas. Zapnutie vypnutého počítača v daný okamih sa vykonáva pomocou *Wake-on-LAN* štandardu[1], ktorý pošle tzv. *magic packet* po sieti a prebudí počítač s požadovanou MAC adresou.

2.2 Databázová infraštruktúra

V tejto sekcii predstavím dokumentovú databázu *MongoDB*, ktorú budem v novom návrhu využívať, porovnám ju s ďalšou rozšírenou dokumentovou databázou *CouchDB* a zhrniem, prečo je pre návrh vhodnejšia práve prvá spomenutá. Na záver sekcie popíšem súčasné riešenie a zhrniem, prečo je lepšie použiť dokumentovú databázu ako relačnú.

[1] http://en.wikipedia.org/wiki/Wake-on-LAN

2.2.1 Dokumentová databáza

Dokumentová databáza je typ databázy určený k ukladaniu, získavaniu a správe dokumentových, semi-štrukturovaných informácii. Je to jeden z najvýznamnejších predstaviteľov tzv. NoSQL databázových systémov. Na rozdiel od štandardných relačných databáz a základného relačného objektu tabuľka je dokumentová databáza postavená na abstraktnom objekte *Dokument*. Implementácia dokumentu sa v každej dokumentovo-orientovanej databáze líši, ale základom je obalenie uchovávaných dát v nejakom štandardnom formáte a kódovaní. Štandardne používané formáty sú XML, YAML, JSON, BSON, prípadne binárne formáty typu PDF a tiež dokumenty MS Word, Excel a iné. Dokumenty si môžeme predstaviť ako riadky v relačnej databáze, no v porovnaní s nimi dokumenty nemusia spĺňať presne definovanú štruktúru, tak isto ako nemusia mať rovnaké stĺpce, časti alebo kľúče. Napríklad si môžeme predstaviť dokument, ktorý obsahuje informácie o študentovi – jeho meno, priezvisko a vek. Iný dokument môže okrem týchto informácii obsahovať aj údaje o jeho adrese. Tieto dva dokumenty obsahujú niektoré položky rovnaké, niektoré rozdielne. Narozdiel od relačných databáz nemusia byť v jednom z dokumentov prázdne atribúty, nemusia tam byť dokonca vôbec prítomné. Dokumenty sú v rámci systému adresované unikátnym kľúčom, ktorým je väčšinou obyčajný string. V niektorých prípadoch môže byť tento string v URI formáte.

2.2.2 NoSQL databáza MongoDB

MongoDB[4, 12] je dokumentová databáza, ktorá dáta ukladá vo formáte podobnom JSON s dynamickými schémami (MongoDB nazýva tento formát BSON). MongoDB umožňuje tzv. ad-hoc queries. Pod týmto pojmom si môžme predstaviť vyhľadávanie v dokumentoch podľa atribútu, rozsahu hodnôt, regulárnych výrazov. Požiadavky môžu vrátiť rôzne časti dokumentov, prípadne môžu zahŕňať užívateľom definované JavaScript funkcie. Indexovanie je riešené podobne ako v relačných databázach. Index môže byť priradený ľubovoľnej položke v dokumente. MongoDB umožňuje replikáciu dát v režime *master-slave*. Master môže vykonávať zápis a čítanie dát. Slave si kopíruje dáta od mastera a môže byť použitý len na čítanie dát, prípadne zálohovanie. Silnou stránkou MongoDB je *load balancing*. MongoDB využíva horizontálne škálovanie pomocou sharding [12] technológie. Vývojár si zvolí shard key, ktorý určuje, ako budú dáta distribuované. Následne sú dáta rozdelené do sekcií podľa shard key a tieto sekcie sú distribuované na rôzne shards. Shard je master s jedným alebo viacerými slave. MongoDB môže bežať na viacerých serveroch a dynamicky upravovať záťaž a/alebo duplikovať dáta, aby systém stále bežal aj v prípade hardwarovej poruchy. MongoDB využíva techniku predstavenú spoločnosťou Google zvanú *MapReduce*, slúžiacu na spracovanie veľkých množín dát, na batch processing dát a výpočet agregačných funkcií. Ďalšou výhodou MongoDB je nepochybne možnosť využitia server-side JavaScriptu v požiadavkách, agregačných funkciách, ktoré sú priamo posielané databáze na vykonanie.

2.2.3 NoSQL databáza CouchDB

CouchDB[13] je dokumentová databáza od firmy Apache, ktorá je napísaná v programovacom jazyku Erlang. Databázový systém je navrhnutý pre miestnu replikáciu a jednoduchú horizontálnu škálovateľnosť. Databáza bola navrhovaná a vyvíjaná pre webové služby. CouchDB, podobne ako MongoDB, ukladá dokumenty vo formáte JSON a tieto sú zhromažďované v kolekciách. Základným prístupom k tejto aplikácii je REST HTTP API, teda požiadavky posielané v HTTP requestoch. Dokumenty sú v CouchDB ukladané ako dvojice

pole/hodnota vo formáte JSON. Hodnoty môžu byť jednoduché typy ako číslo či reťazec, ale aj zložitejšie elementy ako asociatívne mapy alebo listy. Dôležitou vlastnosťou CouchDB je pracovanie s ACID transakciami, vďaka čomu je zaručené konkurentné čítanie a zapisovanie.

2.2.4 Porovnanie MongoDB a CouchDB

MongoDB, na rozdiel od CouchDB, využíva mechanizmus *update-in-place*. Tento mechanizmus sa snaží nahradiť starý objekt novým, a to najskôr, ako je to možné. MongoDB perfektne zvláda problémy, kde je požadovaná vysoká frekvencia aktualizácie objektov. Naopak CouchDB využíva MVCC architektúru, ktorá zabezpečuje transakčný prístup. Príkladom MVCC správania je databáza, ktorá pri aktualizácii objektu namiesto jeho vymazania vytvorí novú verziu daného objektu, ale starú verziu si stále necháva v pamäti. Tento prístup je vhodný v prípade potreby verzovania ukladaných dát, prípadne neskoršou synchronizáciou databáz. Nevýhodou tohto prístupu je, že v prípade chyby v transakcii sa musí táto chyba odstrániť manuálne vývojárom. CouchDB využíva indexovací systém na vytvorenie indexov, ktoré podporujú určité požiadavky. Nevýhodou tohto prístupu je preddeklarácia jednotlivých štruktúr pre každú požidavku, ktorá sa má vykonať. MongoDB narozdiel od toho využíva štandardné dynamické požiadavky podobné požiadavkám z MySQL, pri ktorých nie je nutnosťou mať vytvorený index nad dátami, prípadne je index prítomný, ale využíva sa len čiastočne. Ako som už spomenul vyššie, CouchDB využíva REST ako základný interface k databáze. MongoDB sa zameriava viac na výkon, preto využíva databázový driver špecifický pre každý jazyk. Vo všeobecnosti sa MongoDB snaží byť orientované na výkon, najmä vďaka:

- napísaný v C++
- špecifický driver pre programovací jazyk; nevyužíva sa REST
- collection-oriented storage – kolekcie, ktoré spolu súvisia, sa ukladajú za sebou
- update-in-place, narozdiel od MVCC

Z vyššie spomenutých dôvodov som sa rozhodol používať MongoDB, pretože potrebujem najvyšší možný výkon, nepotrebujem verzovanie ukladaných dát a v porovnaní s REST API je prístup špecifických driverov pre programovací jazyk výhodnejší.

2.2.5 Databázová infraštruktúra a Ki-Wi Server

Súčasná verzia Ki-Wi Servera využíva relačnú databázu MySQL na oddelenom databázovom serveri. Používaním sa zistilo, že MySQL a obecne relačná databáza začína výkonnostne zaostávať za požiadavkami, ktoré sú na Ki-Wi Server kladené. Aplikácia využíva na komunikáciu s databázou a mapovanie databázových objektov na Java objekty framework Hibernate. Tento framework je veľmi robustný, no svojou robustnosťou stráca na výkone. Výhodou použitia MongoDB je možnosť nepracovania s Hibernate. Okrem toho sú dáta uložené v JSON formáte, s ktorým sa pracuje aj v iných častiach systému, čím sa opäť dokáže ušetriť čas potrebný na spracovanie dát.

2.3 Komunikačné protokoly

V tejto sekcii predstavím dva komunikačné protokoly, ktoré sú používané v Ki-Wi Serveri na komunikáciu medzi serverom a obsluhovanými klientmi.

2.3.1 AMQP a RabbitMQ

AMQP[1] je skratka pre *Advanced Message Queuing Protocol*. Pod týmto pojmom si môžeme predstaviť komunikačný protokol, ktorý sa využíva pri message-oriented middleware, v skratke MOM. MOM je softwarová, prípadne hardwarová infraštruktúra podporujúca prijímanie a odosielanie správ medzi distribuovanými systémami. MOM dovoľuje aplikáciám byť distribuované na rôzne platformy a redukuje komplexnosť návrhu aplikácie, ktorá funguje na viacerých operačných systémoch a sieťových protokoloch. AMQP sa využíva pri posielaní správ medzi rôznymi aplikáciami. AMQP je binárny protokol aplikačnej vrstvy, ktorý je orientovaný na správy a umožňuje kontrolu toku komunikácie, takisto ako autentikáciu a šifrovanie založené na SASL alebo TLS. AMQP predpokladá spoľahlivý transportný protokol, akým je napríklad TCP.

RabbitMQ

RabbitMQ[7] je sprostredkovateľ správ, ktorý implementuje AMQP špecifikáciu. Základná myšlienka je veľmi jednoduchá: RabbitMQ prijíma správy a preposiela ich ďalej. V jednoduchosti je možné si pod pojmom RabbitMQ predstaviť schránku, do ktorej sa odosielajú správy a RabbitMQ ich doručí danému adresátovi. V RabbitMQ terminológii sa môžeme stetnúť s nasledujúcimi označeniami:

- producent – aplikácia, ktorá odosiela správy
- fronta správ – buffer, do ktorého producenti odosielajú správy a konzumenti odtiaľ správy prijímajú
- konzument – program, ktorý prijíma správy

2.3.2 Bayeux protokol

Základným cieľom Bayeux protokolu[6] je podpora obojsmernej interakcie medzi webovými klientmi a webovým serverom, napríklad pomocou AJAXu. Bayeux je protokol na transport asynchrónnych správ, zväčša pomocou HTTP, s nízkou latenciou medzi serverom a klientom. Správy sú smerované cez pomenované kanály a môžu byť doručené:

- od servera klientovi
- od klienta serveru
- od klienta klientovi cez server

Doručovanie asynchrónnych správ od servera klientom bez toho, aby si klient nejaké správy vyžiadal, sa často nazýva *server push*. Kombinácia server push techník a AJAX webovej aplikácie sa nazvalo *Comet*. Comet využíva dlhotrvajúce HTTP spojenie pre komunikáciu medzi klientom a serverom. Existujú dve základné techniky pre implementáciu Comet: streaming a odosielanie požiadaviek. Pri streamingu si aplikácia otvorí jeden dlhotrvajúci kanál na server pre všetky udalosti. Udalosti sú postupne spracovávané a interpretované

na strane klienta vždy, keď server nejakú udalosť vytvorí a odošle. Odosielanie požiadaviek je technika, pri ktorej klienti odosielajú požiadavky na server pre udalosť. Klient vytvorí AJAX požiadavku na server, ktorá je otvorená, až kým server nemá nové dáta, ktoré by mohol odoslať klientovi. Klient inicializuje novú požiadavku na získanie nasledujúcich udalostí. Pri implementácii tejto techniky sa využíva XMLHttpRequestObject ako pri štandardnej XHR komunikácii. Klient vytvorí asynchrónnu požiadavku na server a server odpovie vtedy, keď má k dispozícii dáta, ktoré zabalí do odpovede napr. ako JSON. Klient následne vytvorí novú požiadavku na získanie ďalšej udalosti. Ki-Wi Server využíva implementáciu Cometu, *CometD* od Dojo Foundation, ktorá podporuje implementáciu Bayeux protokolu vo viacerých programovacích jazykoch a využíva techniku AJAX odosielania požiadaviek na server od klientov.

2.4 MVC architektúra

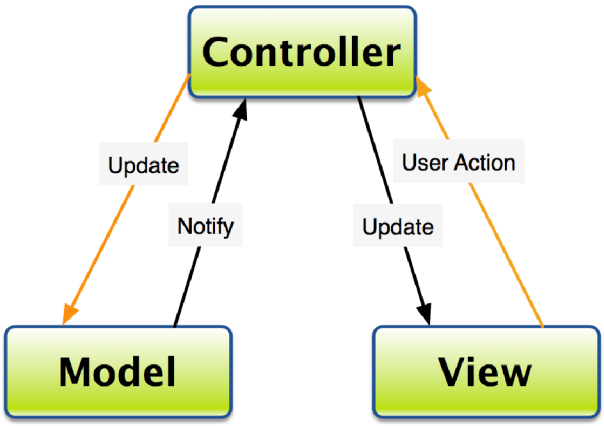
MVC predstavuje skratku *Model-View-Controller*. Pod týmto pojmom si môžeme predstaviť architektúru, ktorá oddeľuje prezentačnú vrstvu od vrstvy dátovej. Medzi tieto vrstvy zavádza takzvaný controller, ktorý sprostredkováva dáta prezentačnej vrstve. MVC architekrúra pozostáva z troch častí:

- **Model** sú dáta, s ktorými aplikácia pracuje, je to teda dátový základ celej aplikácie. Model poskytuje prostriedky pre prístup k dátam, ktoré sú často uložené v databáze.
- **View** je prezentačná vrstva, teda vrstva, ktorá zobrazuje informácie a dáta z dátovej vrstvy.
- **Controller** reaguje na udalosti, ktoré väčšinou prichádzajú od užívateľa a adekvátne na ne odpovedá. Pod pojmom odpovedá si môžeme predstaviť volanie funkcií, zmenu stavu modelu, prípadne zmenu View, a iné.

Jednotlivé vrstvy architektúry medzi sebou komunikujú nasledovne:

- **Model** informuje pripojené views a controller, že došlo k zmene jeho stavu. Tieto notifikácie dovoľujú views obnoviť výstup pre užívateľa a controlleru vykonať množinu príkazov, ktoré sú spojené s danou zmenou stavu.
- **View** požaduje informácie od modelu potrebné na prezentáciu pre užívateľa.
- **Controller** môže odosielať príkazy pre views, aby zmenili prezentáciu modelu, prípadne môže odoslať príkazy modelu, aby zmenil svoj stav.

Na obrázku 2.1 sú znázornené väzby medzi jednotlivými časťami MVC architektúry.



Obrázek 2.1: Komponenty MVC architektúry

Výhodou pri použití MVC prístupu je:

- jednoduché použitie; pri pridaní nového klienta netreba meniť model, niekedy ani controller
- minimalizácia duplicitného kódu, najmä vytváranie databázových spojení
- jednoduché rozdelenie vývojárskych rolí
- oddelenie dátovej a prezentačnej vrstvy
- znovupoužiteľnosť kódu
- komplexný návrh, aplikácia je znovupoužiteľná do budúcnosti

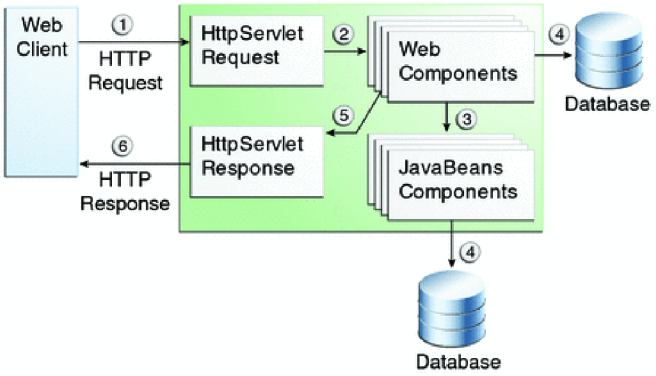
2.5 Webová aplikácia v prostredí Java EE

V tejto časti oboznámim čitateľa s tým, ako funguje webová aplikácia v prostredí Java EE[10]. Webová aplikácia v Java EE je dynamickým rozšírením webového, prípadne aplikačného servera. Navrhovať budem aplikáciu zameranú na prezentáciu dát, ktorá generuje dynamické webové stránky obsahujúce rôzne typy značkovacieho jazyka, ako je napríklad HTML, XHTML, XML, atď., a iný dynamický obsah ako odpoveď na požiadavku.

V Java EE platforme sú prítomné webové komponenty, ktoré poskytujú dynamické rozšírenie pre webový server. Webovým komponentom môže byť Java servlet, webové stránky implementované JavaServer Faces technológiou, webová služba, prípadne JSP stránky. Obrázok 2.2 zobrazuje väzby medzi webovým klientom a aplikáciou využívajúcou servlet. Užívateľ odošle HTTP požiadavku na server. Webový server, ktorý implementuje Java Servlet a JavaServer Pages konvertuje požiadavku na HttpServletRequest a odošle ho webovému komponentu, ktorý ho spracuje, vygeneruje dynamický obsah, zaobalí ho do HttpServletResponse objektu, ktorý server transformuje na HTTP response a odošle ju späť klientovi.

Servlety sú triedy v Jave, ktoré dokážu dynamicky prijímať požiadavky a generovať odpovede. JavaServer Faces a Facelets sú technológie, ktoré umožňujú generovať interaktívne webové stránky. Servlety sú najčastejšie používané v service-oriented aplikáciách a ako kontrolné funkcie v prezentačných aplikáciách, ako napríklad doručovanie požiadaviek.

JavaServer Faces a Facelets sú vhodnejšie na generovanie textovo založených stránok, ako napríklad HTML, a sú používané pre prezentačné webové aplikácie.



Obrázek 2.2: Spracovanie požiadavky webovou aplikáciou, prevzaté z [3]

2.6 Aplikačný framework Spring

Aplikačný framework je softwarový framework, ktorý je navrhnutý tak, aby vývojár nemusel vykonávať všetky bežné rutinné práce, ale framework ich vykoná za neho. Väčšina frameworkov je postavená na návrhovom vzore MVC, ktorý som už predstavil.

Aplikačný framework Spring[9] je open source projekt, ktorý bol predstavený v roku 2002 a prvý release bol uvedený v roku 2003. Spring poskytuje komplexný programový a konfiguračný model pre moderné JavaEE aplikácie. Spring je navrhnutý pre zefektívnenie riešenia niektorých bežných problémov, s ktorými sa vývojári stretávajú a pre celkové zjednodušenie tvorby webových aplikácii. Spring uľahčuje prácu z nasledujúcich dôvodov:

1. odstránenie tesných programových väzieb jednotlivých POJO objektov a vrstiev pomocou Inversion of Control (IoC) a Dependency Injection
2. riešenie rôznych aplikačných domén bez nutnosti použitia EJB
3. podpora pre implementáciu komponentov, ktoré pristupujú k dátam, napríklad JDBC alebo ORM
4. možnosť voľby bussiness vrstvy pre aplikačnú architektúru, nie naopak, napríklad EJB a POJO
5. odstránenie konfigurácii a ich presunutie na jednotné miesto

2.6.1 Inversion of Control a Dependency Injection

Spring je postavený na návrhovom vzore Inversion of Control (IoC). Princípom tohto vzoru je prenesenie riadenia programu z kódu, ktorý navrhne programátor, na aplikačný framework. Programátor teda len vytvára obsluhu udalostí, ktoré prídu z kontajneru, v ktorom je daný komponent zaregistrovaný.

IoC sa doplňuje so vzorom Dependency Injection (DI). DI rieši konštrukciu závislostí medzi jednotlivými triedami, ktorých býva veľké množstvo pri rozsiahlych aplikáciách. Závislosti vznikajú vtedy, ked trieda A obsahuje odkaz na triedu B. DI pri štarte kontajneru,

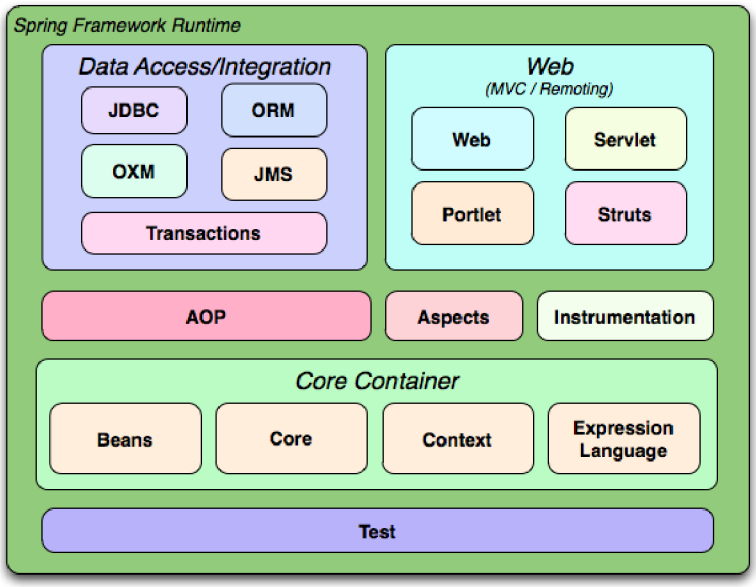
v ktorom sú tieto objekty zaregistrované, obidva vytvorí a inicializuje. Spring postavený na IoC a DI zaisťuje:

- spájanie častí aplikácie a knižníc do fukčného celku
- neinvazívnosť – programátor nemusí použiť knižnicu z frameworku, ale môže použiť vlastnú
- pri IoC sa nemusia používať singletony, aplikácia je napísaná čistejšie a IoC vedie k používaniu rozhraní a SOA

IoC nám pomáha vytvárať aplikáciu ako skladačku, DI pomáha k jej dobrej dekompozícii.

2.6.2 Architektúra Springu

Spring pozostáva asi z 20 modulov, ktoré sú rozdelené do niekoľkých hlavných častí, viď obrázok 2.3. Tieto časti sú Core Container, Data Access-Integration, Web, AOP, Instrumentation a Test.



Obrázek 2.3: Rozdelenie modulov, prevzaté z [5]

Core Container

Core Container pozostáva z modulov Core, Beans, Context a Expression Language. **Core a Beans** moduly sú fundamentálne časti frameworku zahŕňajúce IoC a DI. *BeanFactory* je implementácia návrhového vzoru továreň, ktorá odstraňuje potrebu vytvárania singletonov a povoľuje oddeliť konfiguráciu a špecifikáciu závislostí z programovej časti. **Context** modul stavia na moduloch Core a Beans. Context sa využíva na prístup k objektom spôsobom navrhnutým vo frameworku, ktorý je podobný JNDI registrom. **Expression Language** poskytuje výkonný expression language použitý pre manipuláciu s objektami v čase behu programu. Tento jazyk je rozšírením unifikovaného expression language špecifikovaného v JSP 2.1 špecifikácii.

Data Access/Integration

Táto časť frameworku sa stará o prístup k dátam. Jednotlivé moduly (JDBC, ORM, OXM, ...) sa starajú o špecifický prístup k dátam a uľahčenie práce vývojára pri práci s dátami (odstránenie opakujúceho sa kódu, získanie spojenia, ...).

Web

Web vrstva pozostáva z modulov Web, Web-Servlet, Web-Portlet a Web-Struts. **Web** modul pozostáva zo základných webových funkcií, ako napríklad nahrávanie súborov po častiach alebo inicializíacia IoC kontajneru pomocou listenerov naviazaných na servlet a webového aplikačného kontextu. **Web-Servlet** modul obsahuje MVC implementáciu pre webové aplikácie. MVC v Springu poskytuje separáciu medzi kódom doménového modelu a webovými formulármi. **Web-Struts** obsahuje podporu pre klasickú Struts architektúru a jej previazanie so Spring aplikáciou. **Web-Portlet** poskytuje MVC implementáciu pre portlet prostredie a odzrkadľuje funkcionalitu Web-Servlet modulu.

AOP

AOP je skratkou pre aspect-oriented programming. Tento modul poskytuje implementáciu AOP štandardu, nie však v plnom rozsahu.

Instrumentation

Táto vrstva poskytuje podporu class instrumentation a implementácie classloader pre rôzne aplikačné servery.

Test

Tento modul slúži na testovanie aplikácie pomocou jUnit alebo TestNG.

2.7 Aplikačný server

Aplikačný server je server, ktorý poskytuje aplikáciám služby ako sú bezpečnosť, dátové služby, podporu transakcií, load balancing. Tento termín sa väčšinou používa pre webové servery, ktoré podporujú JavaEE. JavaEE definuje core API a funkcie Java Application Server, ktoré vzišli z definície Java Community Process. Každý server, ktorý sa chce prehlásiť za JavaEE server, musí spĺňať presnú a prísnu špecifikáciu a musí byť otestovaný, aby mohol byť certifikovaný. Existuje veľké množstvo dostupných open source riešení, medzi najznámejšie patrí GlassFish od Oracle, JBoss AS od JBoss, Geronimo a Tomcat od Apache.

2.7.1 Webový server Apache Tomcat

Apache Tomcat[2] je Java HTTP server a Java Servlet kontajner, postavený na Jave. Je to open source riešenie z dielne Apache Software Foundation. Tomcat podporuje Java Servlet API 3.0, JSP 2.2 a Expression Language 2.2. Tomcat je rozdelený na niekoľko komponentov, z ktorých pre mňa najdôležitejší je Catalina, čo je servlet kontajner, a Coyote, čo je HTTP

[2] http://tomcat.apache.org/tomcat-7.0-doc/RELEASE-NOTES.txt

konektor podporujúci HTTP 1.1. Tomcat od verzie 7 podporuje clustering aj high availability. Clustering je možné využiť na zníženie záťaže na server za použitia Load balancing. Práve z týchto dôvodov som sa rozhodol zvoliť si Tomcat za webový server, na ktorom bude nasadená aplikácia.

2.7.2 Virgo Tomcat Server

Virgo Tomcat Server[3] je aplikačný server, ktorý je postavený na moduloch. Virgo je vyvíjaný komunitou Eclipse a je nasledovníkom Spring DM Server. Virgo je navrhnutý na beh aplikácií postavených na Springu a OSGi. Virgo beží na dvoch rôznych webových serveroch – Jetty a Tomcat. Pre moju aplikáciu som zvolil verziu s Apache Tomcat, z dôvodov spomenutých vyššie. Virgo je dodávané a beží na Spring frameworku a aktívne ho využíva pre svoju činnosť. Vďaka tomu sa vývojár nemusí starať o previazanie Springu a OSGi a môže sa sústrediť primárne na vývoj. Virgo využíva tiež Gemini Blueprint špecifikáciu, ktorá uľahčuje zverejňovanie a vyhľadávanie služieb vrámci OSGi frameworku. Výhodou Virga je tiež jeho funkcia hot deployment, ktorá automaticky detekuje zmenu v balíku, ktorý spravuje, a podľa potreby odinštaluje starý modul a nainštaluje a spustí nový.

2.8 Load balancing

Load balancing je postup, ktorý sa využíva pri škálovaní a zaručení vysokej dostupnosti aplikácie. Load balancing rozdeľuje záťaž na jednu aplikáciu medzi viaceré servery, ktoré sú vrámci load balancingu nakonfigurované. Tým sa šetrí výkon každého z nakonfigurovaných serverov. Ideálny stav pri load balancingu je ten, keď sa za t časových jednotiek pripojí N klientov a všetci sú obslúžení po t časových jednotkách. Pri nenastavenom load balancingu a sekvenčnom spracovávaní klientov by sa dané požiadavky vykonali za $N*t$ časových jednotiek.

2.8.1 Integrated Load Balancer

Integrated Load Balancer[2] je load balancer distribuovaný v operačnom systéme Solaris. ILB prijíma požiadavky od klientov a na základe nastavených pravidiel preposiela tieto požiadavky na jednotlivé pripojené servery. ILB podporuje dva typy load balancingu:

- Direct Server Return – tento mód preposiela prišlé požiadavky od klientov na pripojené servery, ale odpovede už nepreposiela klientom on, ale preposielajú ich priamo servery
- NAT mód - v tomto móde ILB prepisuje hlavičky prišlých aj odišlých správ. Tento mód poskytuje vyššiu bezpečnosť a je odporúčaný pre HTTP alebo SSL prevádzku

ILB podporuje taktiež *sticky session*. V tomto režime ILB preposiela komunikáciu vrámci jednej session stále na ten istý server. Nevýhodou tohto režimu je, že prípadný výpadok servera znamená stratu session pre užívateľa. Výhodou je lepšie využívanie lokálnej cache.

[3] http://eclipse.org/virgo/documentation/virgo-documentation-3.6.1.RELEASE/docs/virgo-user-guide/html/index.html

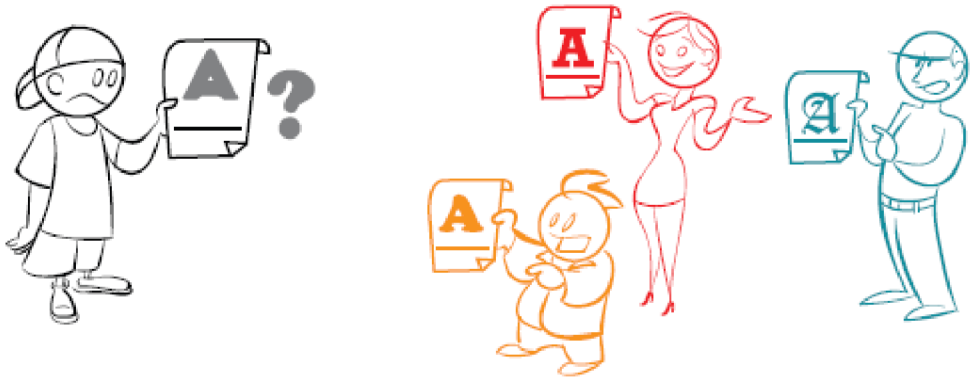
2.9 Service-oriented architecture

Service-oriented architecture[15], v skratke SOA, je množina odporúčaní a metodológií určených pre návrh softwaru vo forme kooperujúcich služieb. SOA je členom event-driven metodológii.

Služba je fyzicky nezávislý program, ktorý má za úlohu nejakú bližšie špecifikovanú funkčnosť. Služba musí spĺňať nasledujúce kritériá:

- viacnásobné použitie
- nezávislé na kontexte
- skladateľné s inými komponentami
- zapúzdrená, t.z. neobjaviteľná cez svoje rozhranie

Príkladom služby môže byť vyplnenie formulára, podanie objednávky v obchode, ... Službu môžeme zjednodušene chápať ako činnosť, ktorú za nás vykonal niekto iný. Túto definíciu môžeme aplikovať na volanie metódy medzi dvomi objektami, pretože volaný vykonáva činnosť namiesto volajúceho. Rozdiel medzi službou a volaním metódy je ten, že služba predpokladá dohodu medzi poskytovateľom služby a zákazníkom. Zákazníka väčšinou nezaujíma presná implementácia služby, niekedy ani samotný poskytovateľ, pokiaľ všetko spĺňa vopred stanovené pravidlá. Použitie služieb v sebe tiež zahŕňa istú formu vyjednávania alebo objavovania, pretože každá služba má nejaké špecifické črty, ktoré ju jednoznačne identifikujú a určujú. Príklad je na obrázku 2.4.



Obrázek 2.4: Služby dodržiavajú kontrakt a zahŕňajú istú formu objavovania, prevzaté z [14]

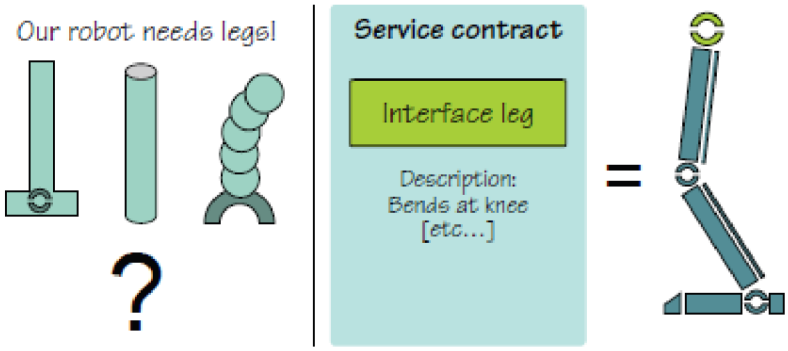
Služby sú mnohými často zamieňané s komponentmi, avšak service-oriented návrh a component-oriented návrh sú dva úplne odlišné prístupy. V service-oriented prístupe, narozdiel od component-oriented prístupu, sa architekt zameriava na pohľad zákazníka, nie na pohľad poskytovateľa. V component-oriented prístupe sa architekt zameriava na zaistenie toho, že komponent, ktorý poskytuje, je zabalený v takom štýle, že to zjednodušuje prácu s ním. Naopak, pri service-oriented prístupe sa architekt zameriava na dodanie funkcionality zákazníkovi, ktorý typicky nemá záujem o to, ako je daná služba interne navrhnutá, ale zaujíma ho, ako sa daná služba bude správať.

Základným mottom, prečo sa využíva service-oriented prístup, je myšlienka vykonávania nejakej činnosti niekým iným namiesto pokusu vykonať všetko sám. Výhody SOA sú:

- menej previazania medzi poskytovateľom služby a konzumentom, čím sa zjednodušuje možnosť znovupoužitia komponentov
- väčší dôraz na rozhraniach (abstraktné) ako na supertriedach (konkrétne)
- jasne špecifikované závislosti, vďaka čomu je hneď jasné, čo k čomu patrí
- podpora viacnásobnej implementácie tých istých vecí, takže je jednoduché vymeniť časti medzi sebou

Inými slovami, SOA podporuje plug-and-play prístup k tvorbe software, čo znamená oveľa väčšiu flexibilitu počas vývoja, testovania, nasadzovania a údržby.

Jednou z najdôležitejších vecí pri práci so službami je kontrakt. Každá služba potrebuje nejaký druh kontraktu, inak by sa konzument nevedel rozhodnúť, ktorá služba spĺňa jeho požiadavky, viď obrázok 2.5. Kontrakt by mal obsahovať všetko, čo by mal konzument vedieť o ponúkanej službe, avšak nič viac. Pridaním informácií, ktoré nie sú potrebné pre konzumenta, sa vytvára tesnejšie prepojenie medzi producentom a konzumentom a obmedzuje sa možnosť zamenenia implementácie.



Obrázek 2.5: Kontrakt, prevzaté z [14]

Najočividnejšie miesto, kde je vhodné použiť služby, je medzi hlavnými komponentmi, špeciálne ak chceme nahradiť, prípadne zmeniť dané komponenty v čase bez nutnosti prepisovať iné časti aplikácie.

2.10 OSGi Service Platform

V tejto časti práce oboznámim čitateľa so základmi a princípmi fungovania OSGi. Predstavím, na čo je vhodné použiť OSGi, aké sú výhody, prípadne nevýhody a prečo ho budem používať.

2.10.1 Čo to je OSGi

OSGi Service Platform[8, 11, 14] je podnikový štandard, ktorý definuje podporu modularity v Jave. OSGi poskytuje lepšiu kontrolu nad štruktúrou kódu, dáva možnosť dynamicky riadiť životný cyklus kódu a povoľuje využiť prístup menšej previazanosti kódu. OSGi je štandard navrhnutý z nasledujúcich dôvodov:

- Java nemá podporu pokročilej modularizácie kódu

- pri vývoji je nemožné presne a explicitne rozdeliť vývoj do nezávislých častí
- pri nasadení je nemožné jednoducho analyzovať, pochopiť a vyriešiť požiadavky obsiahnuté v nezávisle vyvíjaných častiach, ktoré vytvárajú kompletný systém
- pri behu je nemožné meniť bežiace časti systému

V OSGi sa často spomína pojem modularita. Modularitu v OSGi chápeme ako štrukturovanie častí kódu aplikácie do logických častí. Keď je aplikácia modulárna, je jednoduchšie ju vyvíjať a udržiavať.

V Jave existuje možnosť vytvárať modulárny kód. Jednou z nich sú operátory prístupu (private, protected, public, package private). Tieto operátory sú však používané na nízkoúrovňovú modularizáciu a enkapsuláciu, nie na rozdelenie systému do logických blokov. Na to sú v Jave použité balíčky, anglicky *package*. Niekedy si však logická štruktúra aplikácie vyžaduje, aby bol kód vo viacerých balíčkoch. V tom prípade musí byť kód prístupný ako public, čím je možné k nemu pristúpiť z viacerých balíčkov, avšak kód sa stáva prístupným aj pre všetkých ostatných. Toto môže samozrejme spôsobiť odhalenie implementačných detailov, na ktoré sa môže spoliehať implementácia tretej strany, čo môže spôsobiť problémy v budúcnosti pri zmene API.

OSGi rieši väčšinu z problémov, s ktorými sa stretne architekt, keď sa pokúša navrhnúť modulárnu aplikáciu. OSGi môže byť nápomocné v nasledujúcich prípadoch:

- *ClassNotFoundException* pri štarte aplikácie, keď je class path neplatná. OSGi zaručí vyriešenie všetkých závislostí pred samotným vykonaním kódu.
- Chyby pri behu programu spôsobené zlou verziu knižnice. OSGi overí, že všetky závislosti sú konzistentné podľa verzie a iných zadaných požiadaviek.
- Zabalenie aplikácie ako logicky nezávislé JAR archívy a nasadenie len tých balíkov, ktoré sú potrebné. Toto v podstate popisuje dôvod vzniku celého OSGi.
- Zabalenie aplikácie ako logicky nezávislé JAR archívy definujúc, ktorý kód je prístupný z ktorého archívu. OSGi vytvára nový spôsob viditeľnosti kódu na úrovni JAR archívu.

OSGi Service Platform sa delí na dve časti: OSGi framework a OSGi standard services. Framework je prostredie, ktoré implementuje a poskytuje OSGi funkcionalitu. Standard Services definujú znovupoužiteľné API pre bežné úlohy, akými sú napríklad logovanie alebo nastavenia.

2.10.2 OSGi framework

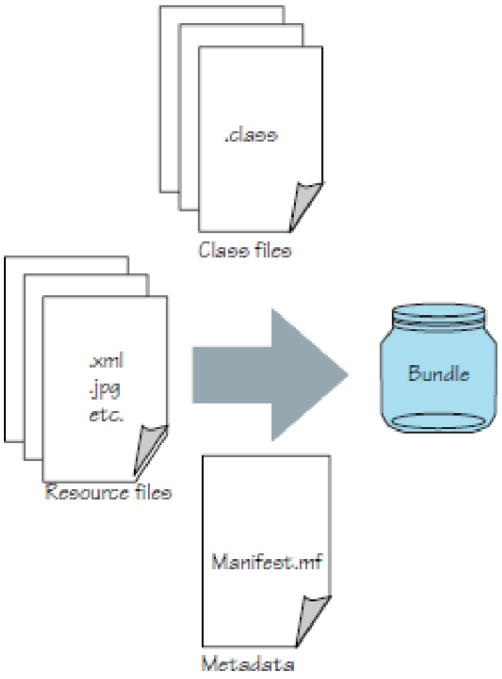
OSGi framework hrá hlavnú rolu pri tvorbe OSGi aplikácii, pretože je to prostredie, v ktorom aplikácia beží. OSGi špecifikácia definuje správne správanie sa frameworku, ktorý poskytuje API pre programy a takisto umožňuje rôznu implementáciu frameworku, vďaka čomu vzniklo niekoľko voľne dostupných implementácii. Najvýznamnejšími z nich sú Apache Felix, Eclipse Equinox a Knoplerfish. OSGi framework sa delí na 3 časti:

- Module layer – zaoberá sa balíkmi a zdieľaním kódu
- Lifecycle layer – zaoberá sa správou modulov počas behu aplikácie a prístupu k OSGi frameworku
- Service layer – zaoberá sa interakciou a komunikáciou medzi modulmi

2.10.3 Module layer

Táto vrstva definuje koncept OSGi modulu, zvaný *bundle*. Bundle je JAR archív, ktorý obsahuje extra metadáta. Bundle obsahuje class files a resources k nim pripojené, viď obrázok 2.6.

Bundle väčšinou nie je celá aplikácia zabalená do jedného JAR archívu, ale sú to logické moduly, ktorých kombináciou aplikácia vznikne. Bundles sú efektívnejšie a silnejšie ako klasické JAR archívy, pretože architekt si v nich môže definovať, ktoré balíčky budú viditeľné (*exportované balíčky*). Ďalšou silnou stránkou bundle je možnosť definovať, na ktorých balíčkoch je dané bundle závislé (*importované balíčky*). Výhodou tohto prístupu je možnosť automatickej verifikácie konzistentnosti OSGi frameworkom – tento proces sa nazýva *bundle resolution*.



Obrázek 2.6: Štruktúra bundle, prevzaté z [14]

Štruktúra metadát

Metadáta sa používajú na presnú špecifikáciu a popis modularity daného bundle pre potreby OSGi frameworku. Vďaka nim dokáže framework rozpoznať závislosti daného bundle. Metadáta obsahujú nasledujúce informácie o bundle:

- *Human-readable informácie* – nepovinné informácie určené pre pomoc ľuďom, ktorí dané bundle využívajú
- *Bundle identifikácia* – vyžadované informácie určené k identifikácii daného bundle
- *Code visibility* – povinné informácie slúžiace na definovanie exportovaných a importovaných balíčkov

Syntax manifestu JAR archívu je postavená zo skupiny atribútov meno-hodnota. Formát atribútu je *meno: hodnota*. Meno nie je case-sensitive a môže obsahovať alfanumerické znaky, podtržítka a zátvorky. Hodnota môže obsahovať ľubovoľné znaky okrem CR a LF. Meno a hodnota musia byť oddelené dvojbodkou a medzerou. Hodnota môže pozostávať z viacerých viet, ktoré sú oddelené čiarkou. Každá veta sa môže deliť na target a zoznam dvojíc meno-hodnota oddelených bodkočiarkou.

Human-readable informácie sú pomocné informácie pre človeka a nie sú spracovávané OSGi frameworkom, avšak špecifikácia definuje niekoľko informácií, ktoré môžu byť použité pre tento účel.

```
Bundle-Name: Sample Name
Bundle-Description: Some Description
```

Prvý atribút definuje názov bundle, druhý je krátky popis bundle.

Bundle identifikácia slúži na presnú a jedinečnú identifikáciu daného bundle vrámci OSGi frameworku. Pre tento účel sa využíva atribút `Bundle-SymbolicName`. Hodnota tohto atribútu má presné pravidlá, ktoré sú podobné názvom balíčkov v Jave: je to séria bodkou oddelených reťazcov. Spolu s názvom bundle sa využíva aj jeho verzia, ktorá sa špecifikuje atribútom `Bundle-Version`. Existuje ešte atribút `Bundle-ManifestVersion`, ktorý je povinný od špecifikácie R4, avšak nadobúdať môže len jednu hodnotu. Nasledujúca trojica teda jednoznačne identifikuje bundle:

```
Bundle-SymbolicName: org.foo.sample
Bundle-Version: 2.0.1
Bundle-ManifestVersion: 2
```

Viditeľnosť kódu slúži na špecifikáciu, ktoré balíčky budú viditeľné z bundle, ktoré balíčky potrebuje bundle importovať a ktoré balíčky budú v bundle obsiahnuté. Keďže štandardné JAR archívy implicitne prehľadávajú všetky adresáre od koreňového pri hľadaní interných tried ako keby to boli názvy balíčkov, OSGi používa špecifickejší prístup zvaný *bundle class path*. Tak isto ako v Jave, bundle class path je zoznam adresárov, v ktorých sa má prehľadávať. Rozdiel je v tom, že bundle class path odkazuje na umiestnenie vo vnútri bundle JAR archívu. Pre definovanie bundle class path sa využíva atribút `Bundle-ClassPath`. V prípade, že bundle je závislé na iných bundle, potrebujeme špecifikovať, ktoré balíčky v akej verzii potrebuje dané bundle importovať. Toto sa dosiahne použitím atribútu `Import-Package`. V nasledujúcom príklade potrebuje bundle importovať balík org.foo.pckg vo verzii 1.4.6.

```
Import-Package: org.foo.pckg; version="1.4.6"
```

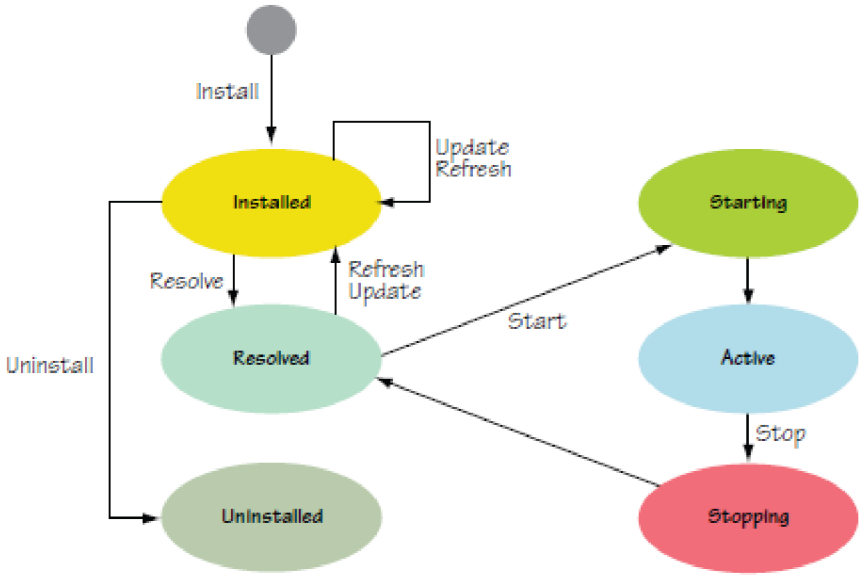
V prípade vystavenia niektorých balíkov pre verejnosť sa používa atribút `Export-Package`, ktorému sa môže určiť tvorca. Verzia exportovaného balíku sa určí z verzie bundle.

```
Export-Package: org.foo.pckg; vendor="FIT"
```

2.10.4 Lifecycle layer

Lifecycle layer definuje, ako sú bundles dynamicky inštalované a spravované vrámci OSGi frameworku. Keby sme stavali dom, module layer by definovala základy a steny a lifecycle layer by bolo elektrické vedenie.

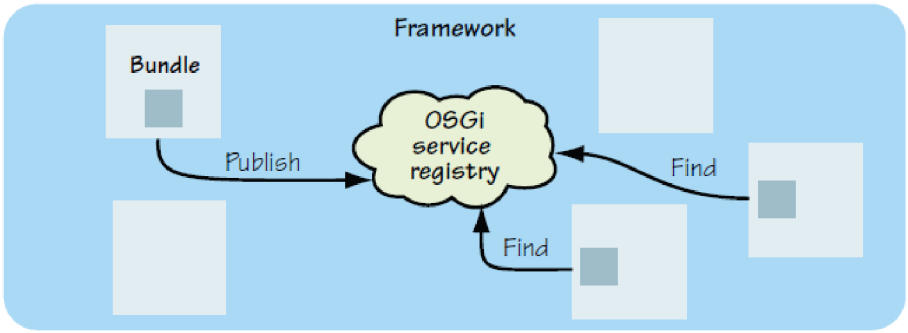
Lifecycle layer má dve hlavné úlohy. Ako prvé, definuje základné operácie každého bundle, ktorými sú inštalácia, update, štart, stop a odinštalovanie. Tieto operácie povoľujú dynamicky spravovať a vyvíjať aplikáciu vopred definovaným spôsobom. To znamená, že bundles môžu byť pridávané a odoberané bez toho, aby sa aplikácia musela reštartovať. Druhou úlohou tejto vrstvy je definovanie, akým spôsobom získajú bundles prístup ku kontextu, v ktorom sú spustené, vďaka ktorému sú schopné interakcie s OSGi frameworkom a možnosťami, ktoré poskytuje za behu aplikácie. Tento prístup je silný, pretože dovoľuje vytvárať externe riadené aplikácie, prípadne samoovládajúce sa aplikácie.



Obrázek 2.7: Životný cyklus bundle, prevzaté z [14]

2.10.5 Service layer

Service layer predstavuje flexibilný programovací model postavený na princípe SOA. Poskytovateľ služby zverejní službu do registru služieb, kým konzument prehľadáva register pre nájdenie služby, ktorú potrebuje, ako je zobrazené na obrázku 2.8. OSGi service layer je intuitívna, pretože podporuje prístup postavený na rozhraniach, ktorý je považovaný ako good-practice najmä tým, že oddeľuje rozhranie a implementáciu. OSGi service a Java interface poskytujú kontrakt medzi poskytovateľom služby a konzumentom, čo spôsobuje, že je service layer odľahčená, pretože poskytovateľ poskytne len Java objekty prístupné cez priame volanie metód.
OSGi služby sú dynamické, to znamená, že služba môže kedykoľvek zaniknúť. Môže sa to stať napíklad vtedy, keď bundle, ktoré ju poskytovalo, bolo odstránené, prípadne nastal výpadok hardware alebo čokoľvek iné. Architekt by si mal byť vedomí tejto funkcionality a mal by sa s ňou vedieť vysporiadať.
Predtým, ako sa služba zverejní, tak ju potrebujem identifikovať, aby mohla byť zákazníkom nájdená. Inými slovami, potrebujem zobrať detaily z implementovaného kontraktu a nahrať ich do registru služieb. Aby som mohol službu nahrať do registru, potrebujem určiť meno rozhrania, implementáciu danej služby a nepovinný slovník metadát, ktorý bližšie špecifikuje daný kontrakt.



Obrázek 2.8: OSGi register služieb, prevzaté z [14]

Po tom, ako je služba úspešne pridaná do registru služieb, konzument ju potrebuje nájsť a naviazať sa na ňu. Pre vyhľadanie služby sú potrebné tie isté detaily, ktoré boli potrebné pre jej zverejnenie. Po úspešnom nájdení služby je vrátený nepriamy odkaz na požadovanú službu. Register musí vracať odkaz, pretože služba a jej implementácia boli oddelené, čím sa zachovala podpora dynamických služieb, tak isto ako možnosť sledovať a kontrolovať prístup k službe, podpora lazy načítania a možnosť oznámenia konzumentovi, že daná služba bola odstránená z registru.

2.11 OSGi a Ki-Wi Server

OSGi som sa rozhodol použiť v Ki-Wi Serveri z nasledujúcich dôvodov:

- modulárne delenie aplikácie
- možnosť správy jednotlivých modulov za behu aplikácie
- možnosť súčasného behu viacerých verzií modulov a prepínanie medzi nimi
- v prípade potreby zvýšenia výkonu je aplikácia ľahko distribuovateľná
- jednoduchší tímový vývoj a správa jednotlivých modulov
- moderný SOA prístup
- výborná dostupnosť dokumentácie
- aktívny vývoj
- najrozšírenejší a najobľúbenejší modulárny framework pre Javu

Kapitola 3

Návrh novej architektúry aplikácie Ki-Wi Server

V tejto kapitole prestavím návrh aplikácie Ki-Wi Server s využitím OSGi frameworku a zmeny, s ktorými sa oproti súčasnému stavu počíta. Následne predstavím rozdelenie aplikácie na jednotlivé bundles a návrh riešenia najdôležitejších častí aplikácie.

3.1 Zmeny oproti súčasnému stavu

V tejto sekcii zhrniem hlavné rozdiely v novonavrhovanej aplikácii oproti súčasnej verzii Ki-Wi Serveru a dôvody, pre ktoré tieto zmeny robím.

3.1.1 Zariadenie

Najvýraznejšie zmeny oproti súčasnému stavu sú v prístupe k zariadeniu. Ako už bolo spomenuté vyššie, zariadenie bolo doteraz chápané ako jeden konkrétny prehrávač. Táto reprezentácia narazila na svoje limity, keďže stále pribúdajú nové typy klientov, ako napríklad kiosk, katalóg a iné, ktoré sa nedajú konfigurovať cez súčasnú verziu. Po novom bude zariadenie predstavovať obrazovku, prípadne skupinu obrazoviek, na ktorej sa prehráva zvolený obsah. Zariadeniu sa oproti starej verzii bude priradzovať obsah, ktorý nebude pozostávať priamo zo zvolených médií, playlistov médií alebo šablón, ale jednotlivé moduly, prípadne playlisty modulov alebo šablóny modulov. Vznikne tak úplne nová vrstva obsahu na zariadení, ktorá doteraz neexistovala.

3.1.2 Obsah

Oproti aktuálnej verzii Ki-Wi Serveru sa práca s obsahom mení hlavne na úrovni zariadenia pridaním jednej vrstvy obsahu navyše. Zariadeniu nie je priradený priamo obsah, ktorý v sebe zahŕňa médiá, playlisty médií alebo šablóny, ale modul, väčšinou prehrávač, prípadne playlisty modulov alebo šablóny modulov. Vzniká tým potreba zjednotiť prácu s obsahom na úrovni modulu prehrávač a na úrovni zariadenie. Jediný rozdiel v playlistoch a šablónach pre zariadenie a modul je typ obsahu, ktorý sa im priraďuje.

3.1.3 Práca s databázou

Vďaka prechodu z relačnej databázy MySQL na dokumentovú databázu MongoDB sa zmení aj mapovanie jednotlivých tabuliek. V súčasnej verzii sa využíval ORM framework Hibernate, ktorý mapoval jednotlivé tabuľky podľa svojich vnútorných pravidiel a tým sa niekedy stávala databázová štruktúra neprehľadná. MongoDB nie je vôbec také striktné ako Hibernate a mapovanie objektov do kolekcií prebieha čisto v programátorovej réžii. Okrem Hibernate sa ešte využíval nástroj EHCache, ktorý si uchovával databázu v pamäti a urýchľoval tak prácu s ňou. Tento nástroj sa v novej verzii využívať nemusí, lebo MongoDB pracuje v režime in-memory automaticky.
Vďaka využitiu dokumentovej databázy bolo možné zjednodušiť prácu s niektorými objektami. Jedná sa hlavne o možnosť ukladať si do dokumentovej databázy zložitejšie konštrukcie ako napríklad List alebo Set, ktoré museli byť v MySQL databáze modelované inak.

3.1.4 Komunikácia

V novej verzii Ki-Wi Servera sa nepočíta s podporou starých typov prehrávačov, ktorých hlavným komunikačným kanálom bol Bayeux protokol. Z tohto dôvodu sa podpora pre tento typ komunikácie neimplementuje a hlavným a jediným komunikačným prostriedkom bude RabbitMQ. Vďaka novej práci so zariadením a obsahom je nutné prekopať aj formát konfiguračného súboru, ktorý je odosielaný na klientov a ktorý popisuje obsah, ktorý sa má na danom zariadení prehrávať.

3.2 Rozdelenie aplikácie na OSGi bundles

V aplikácii Ki-Wi Server som pre správne logické rozdelenie do OSGi bundles musel identifikovať kľúčové časti. Z analýzy systému som prišiel na to, že v Ki-Wi Serveri môžem nájsť štyri logické celky:

- databázový komponent
- komponent zodpovedný za komunikáciu
- komponent starajúci sa o webovú prezentáciu aplikácie
- ovládací komponent

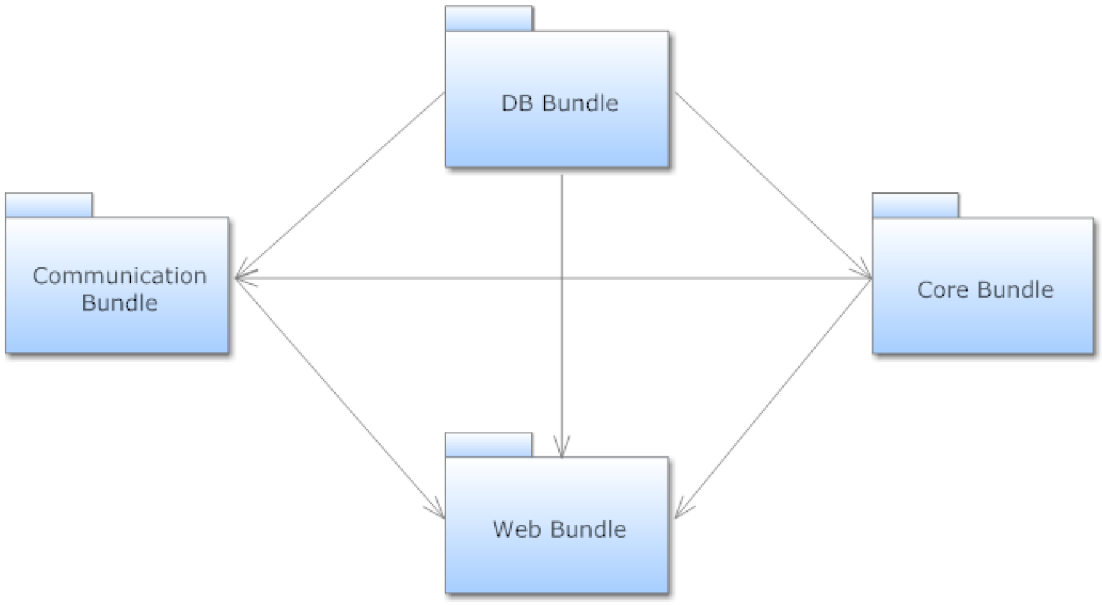
Táto identifikácia presne zapadá do návrhového vzoru MVC. Model je databázová časť a s ňou spojené API, view je webová prezentačná časť a controller je komunikačná a ovladácia komponenta. Rozdelenie je zobrazené na obrázku 3.1.

3.2.1 DB bundle

Databázový komponent bude exportovať rozhrania pre mapovanie databázových objektov na Java objekty. Okrem toho bude z bundle exportované API pre prácu s databázou a jej objektami vo forme rozhraní a SOA princípov.

3.2.2 Web bundle

Prezentačný komponent neexportuje žiadne balíčky. Jeho hlavnou úlohou je prezentácia obsahu zákazníkovi prostredníctvom webového rozhrania. K prezentácii obsahu sú potrebné



Obrázek 3.1: Návrh OSGi bundles

dáta a kontrolné funkcie, preto toto bundle musí potrebné balíčky importovať z DB bundle a Core bundle. Okrem prezentačnej časti, ktorá bude obsahovať balíčky s jednotlivými prezentačnými Beans, je potrebné mať v tomto balíku prítomné aj všetky konfiguračné súbory pre Spring Framework vo forme XML, triedy slúžiace na autentizáciu, plánovanie, prípadne tu budú prítomné utility ako threadpool, enkóder médii a iné. V rámci tohto komponentu budú implementované engines, ktoré uľahčujú prácu pri vykonávaní rovnakých funkcií nad rôznymi objektami. Typickým príkladom je list engine, ktorý slúži na zobrazenie zoznamu objektov v systéme. Túto funkcionalitu užívateľ vyžaduje od každého objektu, preto je vhodné vytvoriť generický engine, ktorý sa o toto stará.

3.2.3 Communication bundle

Komponent zodpovedný za komunikáciu medzi Ki-Wi Serverom a spravovanými klientami potrebuje pre svoju prácu balíčky z databázového komponentu. Databázové balíčky sú potrebné pri odosielaní obsahu na jednotlivé zariadenia, respektíve všeobecne pri práci so zariadeniami ako takými, keďže objekty reprezentujúce zariadenia sa nachádzajú práve v DB bundle. Komunikácia prebieha pomocou protokolu RabbitMQ pre nové prehrávače vytvorené v C#. Exportovaným balíčkom musí byť práve ten, ktorý v sebe zahŕňa rozhranie pre komunikáciu.

3.2.4 Core bundle

Kontrolný komponent obsahuje zoznam balíkov a služieb, ktoré sú dostupné v systéme. Väčšinou sa jedná o prepojenie modelu a prezentačnej vrstvy, preto musí mať tento komponent dostupné balíčky z DB bundle. Opäť sú exportované iba rozhrania a implementácia ostáva skrytá, aby som sa držal SOA metodológie a aplikácia bola správne navrhnutá a jednoducho spravovaná. Keďže navrhujem systém určený pre prácu s rôznymi objektami ale rovnakou množinou funkcií, ideálnym riešením sa javí možnosť použitia generických tried.

3.3 Detailné návrhy riešenia

V tejto sekcii predstavím návrhy riešenia pre najdôležitejšie časti aplikácie Ki-Wi Server. Jedná sa o návrh komunikácie, návrh štruktúry zariadenia a ukážem návrh jedného z engines, ktoré budú v aplikácii dostupné. Všetky návrhy popíšem na úrovni návrhu tried, jednotlivé metódy a parametre zanedbám.

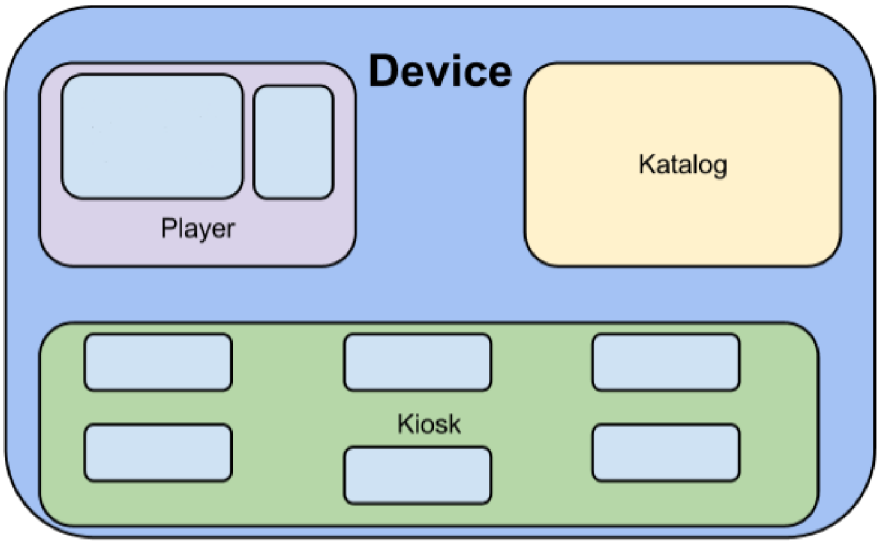
3.3.1 Návrh komunikácie

Ako som už spomenul v predchádzajúcich častiach, komunikácia medzi systémom a zariadením v súčastnosti pozostáva z dvoch komunikačných systémov: použitie Bayeux protokolu a využitie AMQP systému RabbitMQ. V rámci nového návrhu sa už nepočíta s využitím Bayeux protokolu, ale len s RabbitMQ protokolom. Komunikačný systém však musím navrhnúť tak, že AMQP komunikácia nemusí byť jediným komunikačným systémom aj v budúcnosti. Preto pri návrhu musím počítať s vytvorením jednotného komunikačného rozhrania, ktoré bude využívané a exportované ako OSGi služba. Preto navrhnem rozhranie, ktoré bude reprezentovať komunikačnú službu a každá implementácia komunikačného protokolu musí toto rozhranie realizovať. Na obrázku 3.2 je zobrazený diagram popisujúci tento návrh.

Obrázek 3.2: Diagram reprezentujúci komunikačnú službu

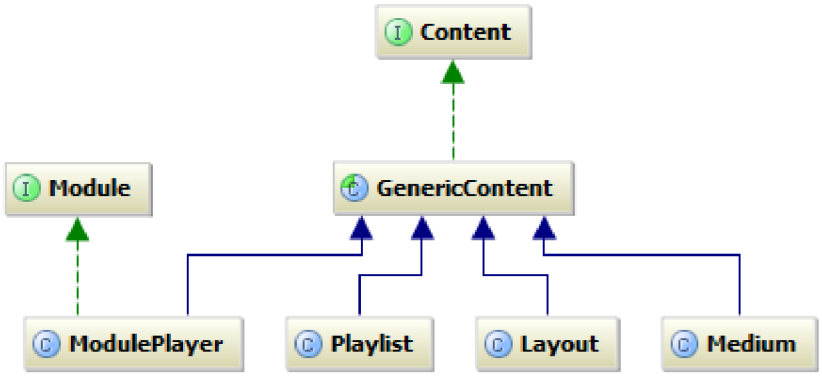
3.3.2 Návrh zariadenia

Oproti súčasnej aplikácii Ki-Wi Server, kde sa pod pojmom zariadenie rozumie jeden konkrétny softwarový typ klienta, teda prehrávač, kiosk, . . . , sú požiadavky na pojem zariadenie zmenené. Po novom je zariadenie definované ako zobrazovacia plocha, na ktorej môžu byť súčasne zobrazené rôzne druhy klientov, prípadne sa rôzni klienti v čase striedajú. Dôvod tejto požiadavky je napríklad interaktívny prehrávač, ktorý detekuje prítomnosť ľudí pomocou kamery. V prípade, že nie sú prítomní ľudia vo vzdialenosti dostatočnej na ovládanie interaktívneho prehrávača, sa tento prehrávač prepne na klasický neinteraktívny prehrávač, ktorý si môžeme predstaviť ako šetrič. Z tohto dôvodu môžem zariadenie modelovať ako obrazovku, ktorá môže mať ľubovoľne veľa modulov, viď obrázok 3.3. Túto požiadavku môžem modelovať ako objekt *Device*, ktorý reprezentuje zobrazovaciu plochu. Tento objekt obsahuje v sebe atribút typu *Content*, čo je interface slúžiaci na reprezentáciu objektov, ktoré môžu reprezentovať obsah. Interface *Module* slúži na definovanie objektu, ktorý môže vystupovať v roli modulu. *Content* implementujú rôzne objekty, ako napríklad *Layout*,



Obrázek 3.3: Zariadenie skladajúce sa z viacerých modulov

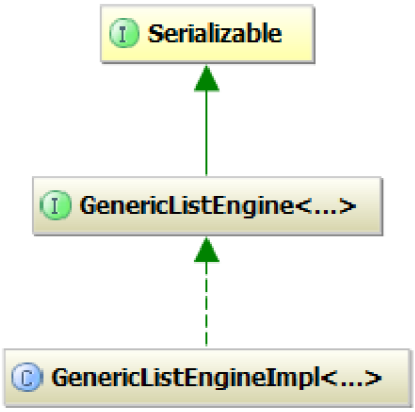
Playlist, *Medium*, ale aj *ModulePlayer*, ktorý okrem implementácie *Content* implementuje aj *Module*. Na obrázku 3.4 je zobrazený návrh obsahu, ktorý môže zariadenie nadobúdať.



Obrázek 3.4: Diagram reprezentujúci nový návrh obsahu zariadenia

3.3.3 Generický engine slúžiaci na zobrazenie zoznamu objektov

Engine slúžiaci na zobrazenie zoznamu objektov môžem chápať ako samostatný komponent zodpovedný za zobrazenie objektov a filtrovanie v nich podľa zadaného kritéra. Možností, kde použiť tento komponent, je v Ki-Wi Serveri veľké množstvo, či už sú to zoznamy zariadení, médií, playlistov alebo organizácií. Vzhľadom na to, že daný engine chcem využiť viacnásobne a každé použitie spĺňa tie isté požiadavky, môžem pri návrhu použiť generické typy. Vďaka nim môžem pracovať s ľubovoľným objektom namiesto presne definovaného. Engine opäť modelujem ako rozhranie, viď obrázok 3.5.



Obrázek 3.5: Diagram reprezentujúci generický engine

3.4 Návrh serverovej infraštruktúry

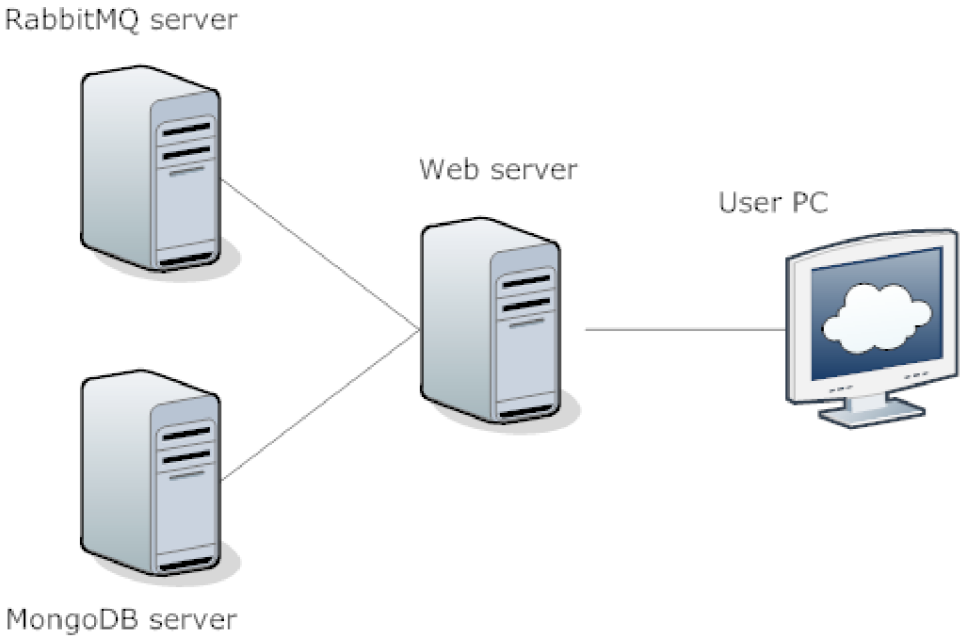
V tejto časti navrhnem infraštruktúru, na ktorej bude systém nasadený, s ohľadom na čo najmenší počet serverov v základnej konfigurácii. Z návrhu a analýzy systému vyplýva, že je možné nasadiť a používať aplikáciu na jedinom serveri, na ktorom bude bežať databázový server aj OSGi framework. Keď však zoberiem ohľad na fakt, že sa na server môže naraz pripojiť niekoľko stoviek až tisícok klientov a každý z nich bude od systému vyžadovať svoj aktuálny obsah, nápor na databázu by sa prudko zvýšil, čím by sa súčasne znížil výkon ostatných programov spustených na danom serveri, ktorým je aj samotný Ki-Wi Server. Prihliadnuc k tomuto faktu som sa rozhodol stanoviť minimálnu serverovú konfiguráciu na dva oddelené stroje, pričom jeden z nich bude slúžiť na beh databázového servera a na druhom bude nasadená naša aplikácia. Táto infraštruktúra je znázornená na obrázku 3.6.

3.5 Návrh vysokej dostupnosti

Na systém sú kladené vysoké požiadavky týkajúce sa dostupnosti a škálovateľnosti. V tejto časti navrhnem riešenia, ktoré požadovaný systém urobia odolnejším proti výpadkom, výkonnejším, stabilnejším a dostupnejším pre užívateľov a klientov.

3.5.1 Databázova dostupnosť

Navrhovaná aplikácia potrebuje pre správnu činnosť neustálu komunikáciu s databázou. Všetky operácie vykonávané užívateľmi cez webové rozhranie alebo klientami komunikujúcimi cez RabbitMQ protokol si vyžadujú prístup k obsahu, ktorý je v databáze uložený. Preto je esenciálne, aby bola databáza dostupná za každých okolností a v prípade výpadku primárneho databázového servera boli aktuálne dáta automaticky dostupné zo servera záložného bez toho, aby užívateľ zaznamenal akékoľvek problémy spojené s nefungujúcou databázovou infraštruktúrou.
MongoDB ponúka niekoľko možností zabezpečenia vysokej dostupnosti a škálovateľnosti databázy, či už za využitia replikácie medzi jednotlivými databázovými uzlami alebo využitia sharding technológie pre distribuovanie dát na rôzne servery. Mojím cieľom bude vytvoriť



Obrázek 3.6: Minimálna serverová infraštruktúra

replikáciu primárneho databázového servera na sekundárny server z dôvodu zabezpečenia neustálej dostupnosti databázy pre aplikáciu. V prípade výpadku master servera replikácia zabezpečí, aby sa zo sekundárneho servera stal primárny s kompletnou kópiou dát.

3.5.2 Aplikačná dostupnosť

Vzhľadom na to, že navrhujem webovú aplikáciu, je dôležité, aby bol ku nej zaručený čo najbezpečnejší prístup. Taktiež pri vysokom počte aktívne pracujúcich užívateľov musí byť kladený dôraz na čo najplynulejšie pracovanie aplikácie a s tým spojený vysoký výkon aplikácie a infraštruktúry, na ktorej daná aplikácia bude nasadená. Z týchto dvoch dôvodov sa mi javí ideálne použitie istej formy load balancingu, ktorá zabezpečí jednak vysokú dostupnosť aplikácie, pretože pri výpadku jedného svojho uzlu presmeruje požiadavky na iný pripojený uzol, ale aj zabezpečí rovnomerné rozdelenie záťaže medzi jednotlivé pripojené uzly, čím zrýchľuje užívateľský dojem a zefektívňuje prácu s aplikáciou pre užívateľov.

Kapitola 4

Implementácia

V tejto kapitole popíšem implementáciu aplikácie, infraštruktúru, na ktorej bude aplikácia nasadená, technológie použité pre implementáciu rôznych častí a problémy, s ktorými som sa stretol pri implementácii.

4.1 Technológie

V tejto časti predstavím technológie, ktoré budú použité pri vývoji a nasadení aplikácie.

4.1.1 Operačný systém

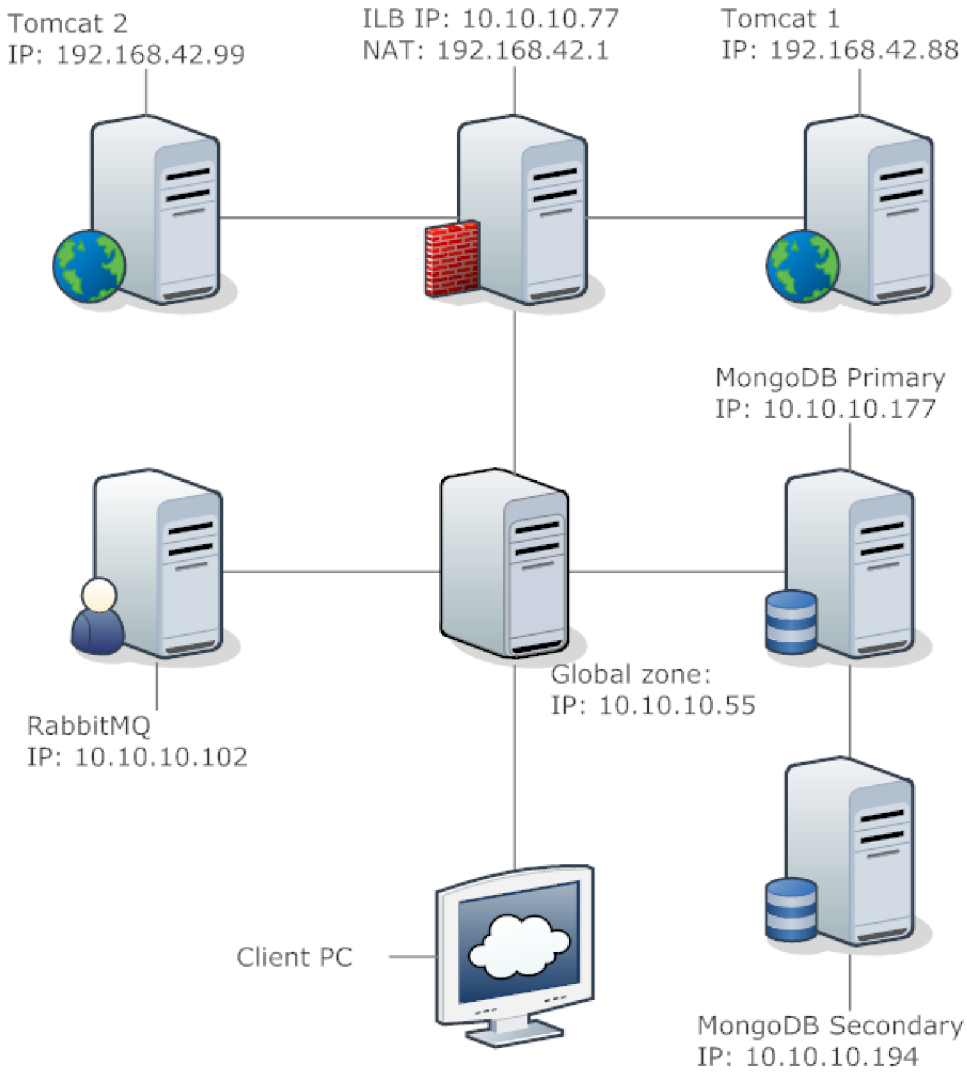
Voľba operačného systému je jednou z najdôležitejších vecí pri návrhu a najmä nasadení aplikácie. Správna voľba systému môže následne uľahčiť ďalšiu prácu respektíve môže poskytnúť technológie, ktoré môže aplikácia využívať.
V rámci nasadenia aplikácie Ki-Wi Server sa počíta s niekoľkými subsystémami, pričom každý z nich potrebuje spĺňať špecifické požiadavky podľa spôsobu využitia. Pre nasadenie a beh aplikácie potrebujem nasledujúce subsystémy:

- subsystém pre beh databázy MongoDB
- subsystém pre beh komunikačného mechanizmu RabbitMQ
- subsystém pre nasadenie Java aplikácie

Toto sú základné subsystémy, z ktorých sa aplikácia skladá. Pre každý z týchto subsystémov bude určený jeden, prípadne viacero virtuálnych strojov. Tu vyvstáva otázka, akú virtualizačnú technológiu používať. Jedným z najmodernejších operačných systémov pre beh v cloude a virtualizáciu je SmartOS[4]. Tento systém je open source verzia postavená na OpenSolaris jadre. Systém je ideálny pre beh cloud infraštruktúry, umožňuje jednoduchú tvorbu virtuálnych strojov pomocou KVM virtualizácie, postavený je na rýchlom ZFS súborovom systéme a taktiež umožňuje jednoduché vytváranie zón, ktoré predstavujú variantu ku štandardnej KVM virtualizácii. Tieto zóny bežia na rovnakom operačnom systéme, teda SmartOS a umožňujú bezpečné a oddelené prostredie pre beh a nasadenie aplikácii. Keďže je SmartOS postavený na OpenSolaris jadre, je k dispozícii Integrated Load Balancer, ktorý môžem využiť.
SmartOS umožňuje vytvorenie SmartMachine, ktorá je špecificky určená pre nasadenie

[4] http://joyent.com/technology/smartos

MongoDB databázy. Pre databázový subsystém teda použijem túto predpripravenú SmartMachine. Vytvorím dve non-global zóny, z ktorých bude každá predstavovať jeden uzol v MongoDB replikácii. Jeden z nich bude nastavený ako primárny server, master, druhý bude záložný, slave.
Pre nasadenie Java aplikácie použijem dve non-global zóny, na ktorom bude nasadený Virgo Tomcat Sever spolu s aplikačným archívom. Na ďalšej non-global zóne bude nasadený load balancer, ktorý bude spravovať záťaž medzi jednotlivými aplikačnými zónami a užívateľ bude pristupovať len ku virtuálnej IP adrese load balancer zóny. Load balancer bude pracovať v režime half-NAT s nastavenou session persistance.
Posledným subsystémom je komunikačný subsystém RabbitMQ. Preň je opäť vytvorená vlastná non-global zóna, na ktorom je nasadený RabbitMQ server.
Na obrázku 4.1 je zobrazená výsledná infraštruktúra, na ktorej bude aplikácia nasadená.



Obrázek 4.1: Výsledná infraštruktúra virtuálnych strojov

4.1.2 Java technológie

Pre vývoj aplikácie a jej beh som použil 64 bitovú verziu Javy 6 a JavaEE 6 štandard. Javu 7 som nepoužil z dôvodu, že pre Joyent SmartOS nebola vydaná oficiálna distribúcia pre balíčkovací systém pkgin. Vrámci Java EE 6 štandardu som na vývoj použil JavaServer Faces 2.1 s rozšírením PrimeFaces 3.4, ktorá poskytuje veľké množstvo ľahko použiteľných komponentov. Za aplikačný framework som si zvolil Spring 3.1.4, ktorý má priamu podporu vo Virgo Tomcat Serveri 3.6. Zvolením Springu za aplikačný framework sa nasledujúca voľba komponentov veľmi uľahčila. Zabezpečenie aplikácie pomocou SSL vykonávam vďaka Spring Security 3.1.3, komunikáciu s databázou a mapovanie databázových objektov na Java objekty vykonáva knižnica Spring Data MongoDB vo verzii 1.1.1. Alternatívou ku Spring Data je ORM framework Morphia, ale zvolil som Spring z dôvodu používania ostatných Spring technológií.
Aplikáciu som vytváral vo vývojovom prostredí IntelliJ IDEA od firmy Jetbrains. Toto vývojové prostredie som zvolil z dôvodu svižnosti, dostupnosti veľkého množstva skratiek a vlastností, ktoré vývojárovi výrazne urýchlia a zjednodušia prácu.

4.1.3 Mechanizmus prekladu

Nástroj pre preklad aplikácie a exportovanie výsledných archívov som zvolil Maven[16]. Maven je štandardný nástroj pre preklad zdrojových kódov, spúšťanie testov, získavania závislostí na externých knižniciach a pod.. Primárne je tento nástroj určený pre Javu, ale podporované sú rôzne jazyky, pre ktoré sú napísané Maven pluginy.
Maven exetrné knižnice hľadá v nakonfigurovaných verejných alebo privátnych repozitároch a sťahuje ich pre potreby prekladu alebo exportovania archívov. Základným súborom, ktorý popisuje chovanie celého procesu prekladu, je tkzv. Project Object Model, ktorý je definovaný ako XML súbor a popisuje projekt nielen na úrovni zdrojových kódov, ale aj na úrovni závislosti na externých knižniciach, postupnosti vykonávania jendotlivých Maven pluginov, spustení testov, ...
Vychádzajúc z návrhu rozdelenia aplikácie na jednotlivé bundles som potreboval rozdeliť aj projekt na zodpovedajúce podprojekty. Každý z podprojektov by mal reprezentovať jeden konkrétny bundle a archív. Všetky podprojekty by mali byť ideálne zaobalené do jedného projektu, cez ktorý sa bude globálne vykonávať preklad celej aplikácie. Z tohto dôvodu som si vytvoril nasledujúcu hierarchiu projektu:

```
- - kiwi_dp.top - typ balíčku: pom
  | - kiwi_dp - typ balíčku: par
  | - kiwi_dp.communication - typ balíčku: jar
  | - kiwi_dp.core - typ balíčku: jar
  | - kiwi_dp.db - typ balíčku: jar
  | - kiwi_dp.parent - typ balíčku: pom
  | - kiwi_dp.web - typ balíčku: war
```

Najvyšší balíček, kiwi_dp.top je typu pom a obsahuje zoznam modulov, ktoré sa majú preložiť. Je to v podstate logická obálka celého projektu. Nasledujú jednotlivé moduly, ktoré sú kiwi_dp.communication, kiwi_dp.core, kiwi_dp.db a kiwi_dp.web. Všetky moduly okrem kiwi_dp.web sú typu jar, kiwi_dp.web je typu war, čo znamená Web Archive. V projekte sa ešte nachádzajú dalšie dva moduly, modul kiwi_dp.parent a kiwi_dp. Modul kiwi_dp.parent je opäť typu pom a sú v ňom definované spoločné repozitáre, externé závislosti a pluginy, ktoré sa majú spúšťať. Posledný modul je kiwi_dp, ktorý je typu par a umožňuje mať v sebe

zahrnutých niekoľko rôznych archívov a nasadiť každý jeden z nich do Virgo Tomcat Servera. Týmto sa výrazne uľahčuje výsledné nasadenie aplikácie, keďže Virgo sa postará o správne vytvorenie všetkých závislostí.
Počas prekladu sú automaticky sťahované všetky externé závislosti. Na záver prekladu pri vytváraní výsledného par archívu sa vytvorí adresár *build*, do ktorého sa uloží výsledný par archív a do podadresára *dependencies* sa nakopírujú všetky externé závislosti.
Počas prekladu sa okrem vytvorenia archívov musia vytvoriť aj potrebné MANIFEST.MF súbory pre každý archív, v ktorých sú informácie o importovaných a exportovaných balíčkoch, závislostiach na iných bundles, obmedzeniach exportovaných balíčkov, ... Pre každý bundle existuje súbor *template.mf*, v ktorom je vytvorená šablóna pre výsledný manifest. Počas prekladu sa z tejto šablóny pomocou maven pluginu *SpringSource Bundlor* vytvorí MANIFEST.MF a umiestni sa do výsledného archívu.

4.1.4 OSGi framework

Ako som už spomenul v predchádzjúcich kapitolách, aplikácia bude rozdelená do niekoľkých modulov, z ktorých každý reprezentuje jeden OSGi bundle. Tieto bundle je potrebné nasadiť do OSGi frameworku spolu so všetkými svojimi externými závislosťami, aby fungovali správne. Prvým krokom je teda zvolenie OSGi frameworku, s ktorým budem pracovať. Na výber som mal Equinox, Apache Felix a Knoplerfish. Vzhľadom na ostatné používané technológie, ako Spring a Virgo, bola voľba jednoznačná. Equinox je dodávaný priamo vo Virgu, takže tým pádom odpadla ďalšia nutnosť konfigurácie a nastavovania samotného frameworku. Virgo okrem všetkých vecí, ktoré Equinox ponúka, má v sebe aj administrátorské webové rozhranie, vďaka ktorému je možné zastavovať, štartovať alebo odinštalovávať jednotlivé nasadené bundles, prípadne si prezerať referencie medzi jednotlivými bundles. Vo verzii Virga, ktoré používam, je posledná verzia Equinoxu, ktorá spĺňa OSGI R4 špecifikáciu.

4.1.5 Nepodporované knižnice

Aby mohol byť jar archív nasadený do OSGi prostredia, potrebuje mať vo svojom manifeste špecifiké informácie potrebné OSGi frameworkom na identifikáciu daného archívu, zistenie závislostí na iných bundles a zistenie exportovaných balíčkov. Nie všetky projekty a zverejňované knižnice však obsahujú OSGi potrebné informácie, tým pádom sa bez úprav nedajú nasadiť do frameworku. Na výber sa ponúkajú tri riešenia:

- zmeniť každé bundle, ktoré potrebuje non-OSGi knižnice na typ war, ktorý podporuje umiestnenie knižníc priamo v sebe v adresári libs
- zmeniť každú nepodporovanú knižnicu na OSGi bundle, teda importovať a exportovať všetky potrebné závislosti a doplniť tieto informácie do súboru MANIFEST.MF
- pokúsiť sa nájsť alternatívu k danej knižnici, ktorú bude možné nasadiť do OSGi frameworku

Prvé zo spomínaných riešení mi neprišlo vhodné, pretože nie každý z nasadzovaných archívov je webová aplikácia a tým pádom mi toto riešenie prišlo nekoncepčné. Tretie riešenie sa ukázalo pri niektorých knižniciach ako dostačujúce, pretože aj keď priamo autori daného projektu nevytvorili z knižnice OSGi bundle, našiel sa niekto iný, ktorý chýbajúce informácie v manifeste doplnil. Väčšinou sa jednalo o knižnice od konzorcia Apache, ktoré väčšinu

jeho vlastných projektov zverejňuje aj ako OSGi bundle z dôvodu, že ich potrebujú pre ich vlastný framework Felix. V niektorých prípadoch som však nedokázal nájsť alternatívne knižnice ku nepodporovaným a bol som nútený vytvoriť si z nich vlastnými silami OSGi bundle. Taktiež som sa stretol s knižnicami, ktoré boli ako OSGi bundle zverejnené, avšak mali buď vadnú špecifikáciu, kedy ich nebolo možné nasadiť popri iných už nasadených knižniciach, alebo nemali správne vyplnené OSGi informácie vo svojom manifeste. V tomto prípade som musel identifikovať, ktoré časti v manifeste nekooperujú s inými technológiami a ručne ich zmeniť, v prípade chybných OSGi informácií tieto chyby nájst a ručne opraviť. V nasledujúcich častiach spomeniem všetky z nepodporovaných knižníc a uvediem riešenie, aké som využil.

PrettyFaces

PrettyFaces[5], verzia 3.3.3, je knižnica, ktorá vytvára tkzv. pekné URL adresy. Jej úlohou je mapovanie aktuálnych URL adries, s ktorými pracuje aplikácia, na pekné adresy, ktoré sú prívetivé pre užívateľa a môže si ich jednoducho zapamätať, prípadne vytvoriť záložku v internetovom prehliadači. Táto knižnica vôbec nepodporuje OSGi, čím som bol nútený vytvoriť všetky potrebné informácie do manifestu sám. Našťastie vývojové prostredie Eclipse, ktoré je celé postavené na OSGi, má podporu vytvorenia projektu, ktorý z existujúceho jar archívu vyexportuje všetky potrebné balíčky a doplní ich do manifestu, čím vznikne validné OSGi bundle. Tento postup však nedoplní všetky externé knižnice, na ktorých je dané bundle závislé. Preto som musel tieto závislosti doplniť ručne. Týka sa to hlavne commons knižníc od Apache, potrebné však boli aj Expression Language a Servlet knižnice. Po týchto úpravách som však dostal validné bundle, ktoré som bol schopný nasadiť do frameworku.

Spring Web Flow

Spring Web Flow[6], verzia 2.3.1, je knižnica vyvíjana konzorciom Spring Foundation a slúži pre rozdelenie prezentačnej časti aplikácie na tkzv. flows, ktoré reprezentujú určitú postupnosť, alebo tok, dát, ktoré užívateľ vidí. Web Flow výrazne uľahčuje prácu s mapovaním jednotlivých Bean a metód z nich na URL adresy. Vďaka Web Flow som nemusel využívať MVC mapovanie samotného Springu na jednotlivé metódy, priraďovať im anotácie a meniť vždy, keď sa zmení štruktúra danej triedy. Problémom tejto knižnice bolo obmedzenie Java Expression Language na verziu 1.0.0 až 2.0.0, lebo najnovšia verzia Expression Language, ktorú používam, je už 2.2.1. Preto som musel ručne zmeniť dané verzie v MANIFEST.MF.

PrimeFaces Extensions

PrimeFaces Extensions[7], verzia 0.6.3, je rozšírenie knižnice PrimeFaces o ďalšie komponenty, z ktorých najdôležitejšia je RemoteCommand. RemoteCommand sa využíva na vyvolanie metódy v Beane na pozadí z JavaScriptu. Táto komponenta rozširuje štandardné chovanie PrimeFaces RemoteCommand o možnosť pridania parametrov do volania funkcie. Toto chovanie sa ukázalo ako esenciálne a veľmi uľahčujúce prácu. Autori PrimeFaces Extensions síce prehlasujú, že archív je zverejnený ako OSGi bundle, aj pridali do súboru MANIFEST.MF informácie pre zaregistrovanie bundle do frameworku, avšak zabudli dodať export všetkých balíčkov a neexportujú žiadne z nich. Vďaka tomu sa dané bundle nenahrá do prostredia

[5] http://ocpsoft.org/prettyfaces/
[6] http://www.springsource.org/spring-web-flow
[7] http://fractalsoft.net/primeext-showcase-mojarra/views/home.jsf

a knižnica je tým pádom nepoužiteľná. Opäť som musel za pomoci Eclipse vyexportovať všetky potrebné balíčky a doplniť importovanie celého bundle PrimeFaces, na ktorom je táto knižnica postavená.

AspectJ Weaver

AspectJ Weaver[8], verzia 1.6.12, je knižnica určená pre podporu aspektovo orientovaného programovania. Aspekty majú široké využitie v aplikáciách, ktoré majú skryté závislosti. Aspektu sa nastaví miesto, v ktorom sa má vyvolať, tkzv. pointcut, a čas, v ktorom sa má vykonať. Čas reprezentujú kľúčové slová Before alebo After, ktoré určia, či sa má aspekt vyvolať pred vykonaním danej metódy, alebo až po ňom. Následne sa v aspekte môže vykonať odstránenie referencií na mazaný objekt, prípadne príprava iných objektov pri vytváraní nejakého nového objektu. V mojej aplikácii je pre aspekty vykonaná príprava, pretože sa s ich využitím počíta v budúcnosti. Z tohto dôvodu som musel integrovať požadovanú knižnicu do systému. V základnom archíve Virga už bola táto verzia knižnice dostupná a je potrebná pre beh samotného Virga. Avšak keď som ho chcel použiť, zistil som, že niektoré exporty spôsobujú nekompatibilitu a dané bundle som nebol schopný nahrať a integrovať. Zistil som, že táto nekompatibilita bola spôsobená exportovaním balíčkov z knižnice, ktoré boli závislé sami na sebe. Po odstránení týchto závislostí už nevznikli žiadne nekompatibility a bundle sa bez problémov nahralo do systému a mohlo byť využívané.

4.2 Práca s databázou

V aplikácii Ki-Wi Server sa využíva dokumentová databáza MongoDB. Výhody využívania tejto databázy boli spomenuté v predchádzajúcich kapitolách, v tejto časti predstavím prepojenie mojej aplikácie na databázovú repliku a popíšem služby, ktoré sú exportované do OSGi prostredia pre ostatné bundle.

4.2.1 Napojenie aplikácie

Pre konfiguráciu prístupu aplikácie ku databáze používam XML konfiguračné súbory, ktoré sú štandardným konfiguračným prostriedkom pre aplikácie postavené na Springu. Namiesto konfiguračných XML som mohol použiť konfiguráciu priamo v konkrétnych triedach pomocou anotácií, ale mne osobne prišlo konfigurovanie vecí prehľadnejšie, keď je oddelené od zdrojového kodu. Rovnako mi príde efektívnejšie písať konfiguráciu do XML vtedy, keď objekty, ktoré potrebujem vytvoriť môžem jednoducho nainicializovať v XML. Vtedy mi príde vytváranie novej triedy neefektívne a nadbytočné, rovnako ako sa stráca prehľadnosť v danom balíčku pri príliš veľkom množstve konfiguračných tried. V konfiguračnom súbore som potreboval vytvoriť objekt, ktorý sa pri štarte daného bundle pripojí na preddefinovanú databázovú repliku a ostane perzistentný po celú dobu behu bundle. Spring Data MongoDB využíva objekt *MongoTemplate*, ktorý dostane ako parametre do konštruktoru repliku, na ktorú sa má pripojiť, databázu a autentifikačný objekt, ktorý uchováva informácie o užívateľovi a hesle. *MongoTemplate* sa dokáže sám vysporiadať s výpadkom primárneho servera v replike, čím je opäť ušetrený čas vývojára. Po úspešnom vytvorení objektu a pripojení sa na repliku je tento objekt dostupný v celom balíku pomocou Spring anotácie *@Autowired*.

[8] http://www.eclipse.org/aspectj/

4.2.2 Mapovanie objektov

Na mapovanie databázových objektov na Java objekty sa v starej verzii Ki-Wi Servera využíval Hibernate. V novej verzii sa s týmto frameworkom nepočíta, využíva sa jedine Spring Data MongoDB. Spring Data podporuje automatické mapovanie pomocou štandardného konvertora, ale vývojár sa v prípade potreby nemusí obmedzovať a môže si vytvoriť vlastný konvertor, ktorý mu poskytuje väčšiu voľnosť. V mojej aplikácii využívam štandardne dodávaný konvertor, pretože využívam štandardné objekty a nepotrebujem ich špeciálne mapovať.

4.2.3 Referencie medzi objektami

Vytváranie prepojenia objektov medzi sebou je v MongoDB databáze možné dvomi spôsobmi. Prvým z nich je využitie vstavaných dokumentov, ktoré premietnu objekt inej triedy ako štandardný Java objekt na JSON, ktorý ale nie je uložený vo vlastnej kolekcii, nemá vygenerované ID a existuje len vrámci iného objektu. Druhým spôsobom je využitie databázových referencií, *DBRef*, ktoré vytvoria odkaz na objekt v inej kolekcii. Tento objekt, alebo dokument, je štandardným databázovým objektom, ktorý ma vygenerované ID a je možné k nemu pristupovať aj mimo objektu, z ktorého bol referencovaný. Druhý spôsob mi prišiel vhodnejší, keďže nepotrebujem vytvárať vždy novú inštanciu toho istého objektu, ale odkazovať sa na jeden konkrétny objekt v celom systéme.

4.2.4 Exportované služby a objekty

Podľa SOA pravidiel by exportovaná služba mala byť pre užívateľa čo najviac oddelená od skutočnej implementácie. Z tohto dôvodu som nemohol exportovať ako databázovú službu priamo *MongoTemplate*, ktorý má všetky potrebné operácie pre manipuláciu s databázou, ale musel som operácie, ktoré používam, nejakým spôsobom zapúzdriť, aby mal užívateľ prístup len ku mojej API, ktorá sa nebude viazať na konkrétnu databázovú implementáciu. Z tohto dôvodu som vytvoril dva manažéri, *DBManager* a *DBQueryManager*, ktoré poskytujú programátorovi kompletné API pre prístup k databáze. Obidva manažéri su implementované a exportované v kiwi_dp.db bundle. Export služieb je implementovaný pomocou Gemini Blueprint špecifikácie, a to pridaním nasledujúcej direktívy do konfigurácie daného bundle:

```
<service
      ref="dbManager"
      interface="cz.kiwi.server.db.service.DBManager"/>
```

DBManager

Tento manažér poskytuje štandardné metódy pre prístup k objektom databázy, mazanie alebo ukladanie. Takisto má programátor k dispozícii metódy pre zistenie počtu objektov, prípadne získanie všetkých objektov, ktoré spĺňajú nejakú podmienku. Každá z týchto metód má ako parameter objekt, ktorý reprezentuje podmienku, podľa ktorej sa budú dané objekty vyhľadávať, mazať, prípadne zisťovať počet objektov, ktoré túto podmienku spĺňajú. Tento objekt je poskytovaný manažérom *DBQueryManager*, ktorý je určený pre dynamické vytváranie podmienok. Okrem podmienky je ďalším atribútom metódy buď trieda, ktorá určuje mapovanie daného objektu do databázovej kolekcie, alebo názov kolekcie, do ktorej má byť daný objekt mapovaný.

DBQueryManager

DBQueryManager je manažár určený k dynamickému vytváraniu podmienok, podľa ktorých sa vyberajú, prípadne mažú objekty v databáze. Na začiatku je nutné manažér vždy inicializovať. Po inicializácii programátor postupne tvorí podmienku metódou *addCriteria(key, value, operation)*, kde argumenty sú postupne referencovaný atribút v objekte, hodnota, ktoré má daný atribút nadobúdať a operácia, ktorá sa bude medzi atribútom a hodnotou vykonávať, napríklad rovnosť, nerovnosť, ... Operácie, ktoré sú podporované, sú uvedené v enumeračnom type *QueryOperation*.

Databázové objekty

Databázové bundle okrem zverejnenia služieb do OSGi frameworku musí exportovať balíčky obsahujúce objekty, s ktorými aplikácia pracuje a ktoré sú ukladané do databázy. Tieto objekty sú logicky rozdelené do štyroch balíčkov podľa typu určenia. V prvom balíčku *common.content* sú objekty, ktoré reprezentujú obsah a sú spoločné pre obsah v zariadení aj v module. V ďalšom balíčku *device* sú objekty spojené so zariadením. V treťom, *module*, sú objekty spojené s modulmi. Tento balík obsahuje podbalík *content*, v ktorom sú umiestnené objekty reprezentujúce obsah na úrovni modulu. Posledný balík je systémový balík *system*, v ktorom sú objekty reprezentujúce organizáciu, užívateľa a úložisko dát.
Každý objekt okrem objektov v balíku *common.content* má anotáciou definované mapovanie do konkrétnej kolekcie. Objekty z balíka *common.content* sú však spoločné pre prácu so zariadením aj modulom, preto je ich mapovanie vykonávané dynamicky podľa typu objektu, ku ktorému prináležia. Napríklad šablóna z médií, ktorá je priradená modulu prehrávač, je mapovaná do kolekcie *player_layout*, avšak šablóna modulov, priradená zariadeniu, je mapovaná do kolekcie *device_layout*.

4.3 Core bundle

Core bundle je jednou z najdôležitejších častí aplikácie, pretože zverejňuje do OSGi frameworku služby, ktoré ponúkajú metódy na manipulovanie s databázovými objektami, výpočty rôznych objektovo špecifických vlastností, ... Pre každý aplikačný objekt existuje manažér, ktorý nad ním vykonáva operácie. Základné operácie spoločné pre všetky objekty sú uloženie, odstránenie, odstánenie všetkých objektov v organizácii, získanie objektu podľa ID alebo mena, získanie všetkých objektov, získanie všetkých objektov v danej organizácii a získanie počtu objektov v organizácii. Keďže všetky tieto služby sú spoločné pre každý objekt, ideálnou implementačnou voľbou sa javí vytvorenie generického manažéra, ktorý implementuje všetky zo spomenutých operácií a každý konkrétny manažér pre daný objekt len rozširuje generický manažér o špecifické metódy. Zvolením tohto postupu som obišiel nutnosť implementovať stále rovnaké metódy pre každý objekt.

4.3.1 Importované a exportované služby

Databázové bundle exportuje dve služby, ktoré sú určené pre nízkoúrovňový prístup ku databáze. Tieto dve služby úplne postačujú pre vykonanie akýchkoľvek operácií nad databázovými objektami, avšak je potrebná ich abstrakcia na vyššiu úroveň pre zjednodušenie práce s objektami.
Každý manažér potrebuje pre manipuláciu s objektami prístup ku databáze a jej operáciám.

Preto je nutné z OSGi frameworku importovať služby, ktoré sú ku tomu určené. Dohľadávanie služieb v OSGi registroch opäť zvládne Gemini Blueprint nasledujúcou direktívou v konfiguračnom súbore:

```
<reference
      id="dbManager"
      interface="cz.kiwi.server.db.service.DBManager"/>
```

4.3.2 Aspekty

V tomto bundle je urobená príprava na prácu s aspektami. Implementovaný je jeden aspekt, ktorý sa využíva pri mazaní organizácie zo systému na odstránenie všetkých objektov, ktoré sú na danú organizáciu naviazané. Pre tento aspekt sa vytvorí Pointcut, ktorý je naviazaný na vykonanie ľubovoľnej metódy umiestnenej v balíku *cz.kiwi.server.core*. Následne je vytvorená metóda, ktorá má anotáciou nastavené spustenie pred vykonaním metódy *remove*.

```
@Pointcut("within(cz.kiwi.server.core..*)")
public void inside() {}

@Before("inside() && args(o) && execution(void remove(..))")
public void deleteDevice(DomainObject o) {
  if (o == null || o.getId() == null)
       return;
  if (o instanceof Organization) {...}
}
```

4.4 RabbitMQ komunikácia

Pre správne zasielanie správ pripojeným klientom je nutné pripojenie aplikácie na RabbitMQ server. Všetky parametre pre pripojenie a nastavenie komunikácie sú umiestnené v súbore *rabbitmq.properties*, ktorý sa automaticky nahrá do systému pri štarte komunikačného bundle. Konfigurácia pre spojenie je okrem toho implementovaná v triede *ServerKiwiRabbitConfiguration*, ktorá má nastavenú anotáciu *@Configuration*.

4.4.1 Importované a exportované služby

Jedinou importovanou službou tohto bundle je manažér pre zariadenie, pretože obsah sa zasiela práve na zariadenie. Tento manažér sa nájde v OSGi registry opäť pomocou Gemini Blueprint.
Služba, ktorá sa zverejňuje do OSGi registru, má v súčasnosti len jednu metódu, ktorou je *sendContent(device)*. Touto metódou sa odošle aktuálne nastavený obsah pre dané zariadenie na zariadenie. Rozhranie, ktoré túto metódu zastrešuje, je *CommunicationManager*, ktoré je implementované triedou *RabbitKiwiManager*. V prípade pridania nového typu komunikácie musí daný komunikačný manažér implementovať spomenuté rozhranie a exportovať s ním spojenú službu do OSGi registru.

4.5 Webové bundle

Poslednou časťou aplikácie je webové bundle, ktoré poskytuje webové rozhranie pre prácu s aplikáciou. V tejto časti je implementovaná kompletná prezentačná vrstva za pomoci PrimeFaces, Spring Web Flow, jQuery a ostatných technológií. Bundle je rozdelené na niekoľko logicky súvisiacich balíčkov. Všetky Beany, ktoré sú využívané pri prezentačnej vrstve, sú umiestnene v balíku *bean*. Tento balík je následne logicky členený, podobne ako databázový balík, do podbalíkov, ktorých obsah spolu súvisí. V balíku *utils* sú umiestnené pomocné utility, ktoré sú využívané naprieč celým systémom. Ďalším dôležitým balíkom je *service*, v ktorom je umiestnený generický engine pre zobrazovanie zoznamu objektov. Tu budú v budúcnosti umiestnené ďalšie engines, ktoré budú postupne vytvárané.

4.5.1 Importované služby

Keďže toto bundle reprezentuje v terminológii MVC view, potrebuje pre svoju prácu controller a model. Model reprezentuje databázové bundle a databázové objekty, ktoré sú exportované. Controller je core bundle, ktoré zverejňuje manažéri pre prácu s objektami ako OSGi služby. Z tohto dôvodu web bundle vo svojom manifeste importuje balíky, ktoré sú zverejnené databázovým bundle a na úrovni zdrojového kódu vyhľadáva potrebné služby v OSGi registroch. V prípade webového bundle nebolo možné použiť na dohľadávanie OSGi služieb Gemini Blueprint špecifikáciu, pretože BundleContext z tejto špecifikácie nebolo možné serializovať, čo je nevyhnutnou podmienkov pre každú triedu, ktorá je nejakým spôsobom zobrazovaná pomocou Spring Web Flow. Vďaka tomu boli síce služby dohľadateľné v OSGi registri, avšak nebolo možné pracovať s Beanou, ktorá tieto služby používala. Z tohto dôvodu som musel nahradiť Gemini Blueprint dohľadávanie v registroch a použiť vlastné vyhľadávanie služieb, ako definuje OSGi štandard.

Dohľadávanie služieb

Pre vyhľadanie služieb som potreboval vytvoriť dve ďalšie triedy, *Activator* a *ServiceTrackUtils*, obidve umiestnené v balíku *utils*. Triedu *Activator* som musel nastaviť v manifeste ako parameter pre `Bundle-Activator`, čím som dal OSGi najavo, že táto trieda sa má brať pri štarte bundle ako aktivátor. Následne som v nej definoval *ServiceTracker* pre každú službu, ktorú som chcel nájsť a spustil som vyhľadávanie pomocou metódy *open*.

```
private ServiceTracker serviceOrganizationManager;
serviceOrganizationManager = new ServiceTracker(bundleContext,
    OrganizationManager.class.getName(), null);
serviceOrganizationManager.open();
```

Pri ukončení behu daného bundle sa vyhľadávanie ukončí pomocou metódy *close*. Každú jednu službu, pre ktorú existoval *ServiceTracker*, som si potreboval uchovať v štruktúre, cez ktorú som následne ku každej službe pristupoval. Táto štruktúra bola mapa v triede *ServiceTrackUtils*, ktorá ako kľúč ku každej službe brala jej názov. Pre každú službu som následne definoval getter, ktorý prehľadával mapu služieb a vracal požadovanú. Následne v kóde, kde som potreboval pristúpiť ku službe, som zavolal príslušný getter pre daný službu a pristúpil som ku nej. Keďže *ServiceTrackUtils* je serializovaná trieda, Spring Web Flow bol schopný danú Bean serializovať a pracovať s ňou.

4.5.2 Prebrané časti

Niektoré časti zo starej verzie sa nemuseli prerábať a ich funkcionalita mohla ostať zachovaná. Jednou z týchto častí je práca s užívateľmi a ich autentizácia, tak isto sa zachoval grafický návrh z pôvodného servera. Časť práce s médiami podstúpila refaktoring, časť z nej však ostala z pôvodnej verzie. Tak isto komunikačná časť musela byť upravená do novej podoby, časť z nej avšak ostala zachovaná.

Kapitola 5

Testovanie

V tejto kapitole popíšem spôsob testovania a výsledky, ktoré rôzne testy priniesli. Zameriam sa na tri hlavné aspekty. Unit testy, ktoré testujú správne chovanie jednotlivých častí aplikácie hneď pri jej tvorbe, záťažové testy určené na overenie záťaže aplikácie a testy vysokej dostupnosti, ktoré overia, či je aplikácia v jej požadovanom nastavení odolná voči výpadku niektorého z potrebných subsystémov.

5.1 Unit testy

Unit test[17] je postup, pri ktorom sa testuje kompaktná časť systému za účelom overenia jej správneho fungovania. Unit testy sú vhodným prostriedkom pri nájdení chýb ešte pred samotným behom programu. Pri unit teste sa definuje požadované chovanie pre danú časť a overuje sa, či vykonanie danej metódy alebo postupnosti metód vedie ku požadovanému výsledku. Unit testy boli vytvorené pre dve bundles, databázové a core. Pre každé z nich sa overovala základná funckionalita, ktorú dané bundle zverejňovalo do OSGi prostredia.

5.1.1 Unit testy v databázovom bundle

Databázové bundle exportuje pre verejnosť dve služby – *DBManager* a *DBQueryManager*. Obidve z týchto služieb sú esenciálne pre správnu činnosť aplikácie, preto musia byť komplexne odtestované. Pre každú zo služieb bol vytvorený samostatný test reprezentovaný triedou umiestnenou v adresáry src/test. Pri testovaní prvej zo služieb je nutné aktívne pripojenie na databázu. Z toho dôvodu sa pred vykonaním každého z testu bolo nutné pripojiť ku MongoDB databáze a po ukončení testu spojenie uzavrieť.

```
@BeforeClass
public static void init() throws IOException {
    mongo = new Mongo(LOCALHOST, MONGO_TEST_PORT);
    mongo.getDB(DB_NAME);
    dbManager = new DBManagerImpl();
    template = new MongoTemplate(mongo, DB_NAME);
    dbManager.setMongoTemplate(template);
}
@AfterClass
public static void shutdown() throws IOException {
    mongo.close();
}
```

Pri testovaní služby *DBManager* bola overená funkčnosť pre 4 základné operácie: save, get, update a remove. Pri overovaní save a remove funkcionality sa overili jediné metódy *save* a *remove*. Pri testovaní get sa okrem získania jedného konkrétneho objektu overila tiež funkčnosť získavania objektov, ktoré spĺňajú nejakú podmienku a zistenie počtu objektov, ktoré spĺňajú danú podmienku. Pri testovaní update sa overovala správnosť aktualizácie jedného alebo viacerých objektov.
Pri testovaní *DBQueryManager* bolo nutné najskôr overiť správnu inicializáciu parametrov daného manažéra, čo sa testovalo v teste *testInit()*. Následne bolo možné testovať vytváranie rôznych obmedzení pomocou metódy *addCriteria* a overovať správne vytvorenie daného obmedzenia. Okrem overenia vytvárania obmedzovacích kritérií som testoval správne vytvorenie radenia objektov podľa podmienky, overenie obmedzenia počtu objektov splňujúcich danú podmienku a na záver správnu konfiguráciu objektu určeného na aktualizáciu vlastností pre daný objekt alebo množinu objektov.

5.1.2 Unit testy v core bundle

Core bundle zverejňuje manažéri pre manipuláciu s rôznymi objektami, ich ukladanie do systému, mazanie, ... Väčšina z funkcií je implementovaná v spoločnom manažéri *GenericOrganizedManager*, z ktorého všetky ostatné manažéri dedia, preto je dostatočné vytvoriť unit test len pre tento manažér a otestovať funkcionalitu, ktorú ponúka. Opäť je nutné mať pri testovaní dostupnú databázu, pretože vrámci testov sa vytvárajú a mažú objekty v nej. Okrem týchto dvoch operácií sa overuje získanie objektu podľa identifikátoru prípadne podľa nejakej inej špecifickej podmienky.
Okrem testovania spoločného manažéra sa testuje správna práca triedy *StorageFactory*, ktorá vracia konkrétnu implementáciu pre požadovaný typ úložiska. V aplikácii sa počíta s podporou štyroch rôznych typov, avšak v súčasnej verzii je implementované len úložisko na Amazon S3 Storage[9].
Unit test je ešte vytvorený pre *StorageManager*, keďže správny prístup k funkcionalite úložiska médií je nevyhnutný pre bezproblémový beh aplikácie. Vrámci testov sa overuje vytvorenie základného úložiska pre organizáciu, ktoré je momentálne reprezentované Amazon S3, získanie úložiska pre organizáciu a získanie všetkých úložísk pre organizáciu. Vytvorenie základného úložiska je dôležitá časť manažéra, keďže vždy po vytvorení novej organizácie je nutné vytvoriť nejaké úložisku a táto funkcia veľmi uľahčuje prácu pre užívateľa, ktorý častokrát nemusí byť oboznámený s rôznymi parametrami pre správne nastavenie Amazon S3.

5.2 Záťažové testy

Aplikácia bola podrobená záťažovým testom. Cieľom testov bolo na jednej strane zistiť, či je aplikácia schopná reagovať v reálnom čase na niekoľko stoviek pripojených klientov požadajúcich obsah, na druhej strane overiť, aká je efektivita load balancera, aké množstvo času ušetrí a koľko klientov je schopný zvládnuť v reálnom čase.

5.2.1 Testovanie doby obsluhy klientov

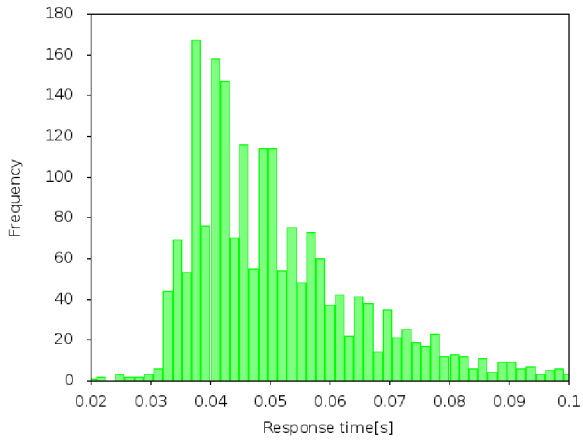
V tejto časti popíšem priebeh testovania dostupnosti aplikácie pri požiadavkach od väčšieho množstva klientov. V reálnej prevádzke sa predpokladá s nasadením niekoľkých stoviek

[9] http://aws.amazon.com/s3/

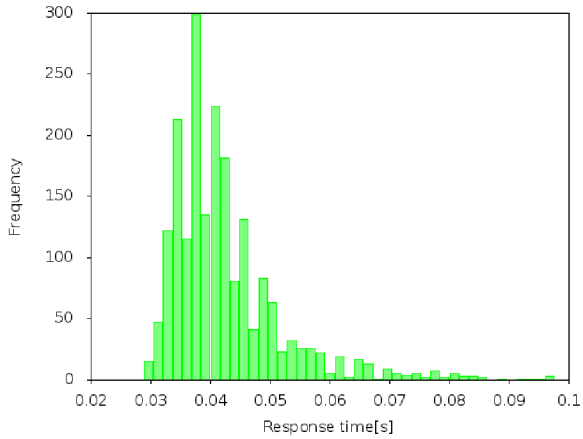
klientov, maximálne však 2000 aktívne spravovaných klientov. Preto bolo toto množstvo klientov testované ako prvé. Vykonávali sa dva druhy testovania dostupnosti:

- testovanie pripojenia z dvoch paralelne pripojených staníc
- testovanie pripojenia z jednej stanice

Na každej stanici bol spustený skript, ktorý sa pripojil ku testovanej aplikácii a žiadal od nej obsah. Dĺžka odoslania obsahu od servera k stanici bola zaznamenaná. Tento postup sa zopakoval x krát pre jednu pripojenú stanicu a $x/2$ krát pre dve paralelne pripojené stanice. Mohol som si dovoliť testovať len dve paralelne pripojené stanice, keďže v load balanceri sú aktívne len dve inštancie aplikácie a teda load balancer bude vždy prepínať medzi dvomi aplikáciami.
Prvý test sa vykonal pre 2000 pripojení z jednej stanice a 2x1000 pripojení z dvoch staníc. Na obrázkoch 5.1 a 5.2 sú zobrazené histogramy, v ktorých je zobrazené rozloženie trvania odpovede od aplikácie.



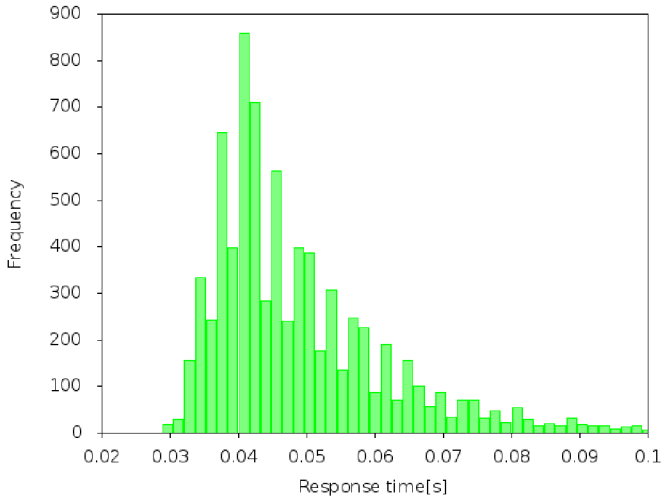
Obrázek 5.1: Histogram rozloženia trvania odpovedí pre test 2x1000 pripojení



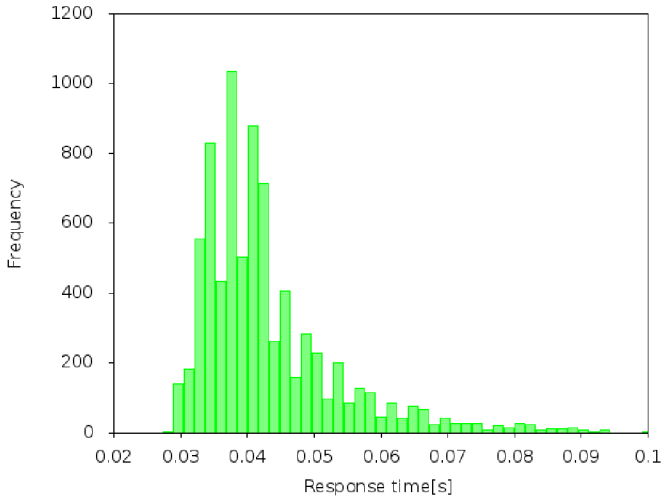
Obrázek 5.2: Histogram rozloženia trvania odpovedí pre test 2000 pripojení

Z histogramov vyplýva, že najčastejšia dĺžka odpovede od aplikácie sa pre dve paralelné

stanice a 1000 požiadaviek pohybovala v intervale $(0,035; 0,06)$ sekúnd. Pre 2000 požiadaviek od jednej stanice sa tento interval zmenil na $(0,03; 0,045)$ sekúnd. Rozdiel v dĺžkach vybavenia jednotlivých požiadaviek je zrejmý. V prípade pripojených dvoch staníc je nutné zapojiť load balancer, ktorý prepína medzi jednotlivými servermi, na ktorých je bežiaca aplikácia. V prípade jednej pripojenej stanice sa vďaka nastaveniu sticky session celá komunikácia z jednej IP adresy presmerováva na jednu inštanciu a tým sa ušetrí práca load balancera, čo sa prejaví aj na rýchlosti odozvy pre každú požiadavku. V ostrom nasadení je reálna situácia s pracujúcim load balancerom, keďže každý klient bude mať inú IP adresu. Časová odozva aplikácie pri aktívnom load balancingu je však stále uspokojujúca pri danej záťaži.
Druhý test bol vykonaný pre 2x4000 spojení. Histogramy sú na obrázkoch 5.3 a 5.4.



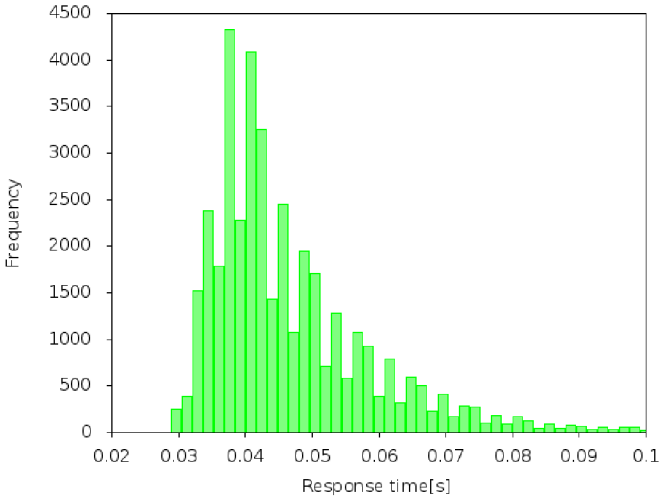
Obrázek 5.3: Histogram rozloženia trvania odpovedí pre test 2x4000 pripojení



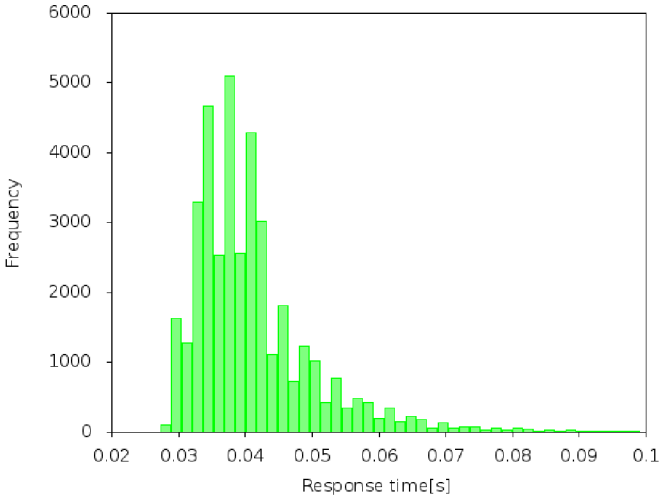
Obrázek 5.4: Histogram rozloženia trvania odpovedí pre test 8000 pripojení

Z histogramov vidno, že najčastejšia doba odozvy pre dve paralelné stanice sa pohybovala v intervale $(0,035; 0,055)$ sekúnd a pre jednu stanicu $(0,03; 0,045)$. Opäť vidno dlhšiu odoz-

vu pre dve stanice, ktoré boli presmerovávané na dve inštancie pomocou load balancera. Doba odozvy sa výrazne nezmenila oproti testu s 2000 pripojeniami.
Pri poslednom teste bolo zvýšené množstvo propojení na 2x20000 a 40000. Týmto testom som chcel overiť správanie sa aplikácie pri dlhodobejšej záťaži. S takýmto množstvom súčasne pripojených klientov sa nepočíta, ale je vhodné overiť aj extrémne prípady. Histogramy sú zobrazené na obrázkoch 5.5 a 5.6.



Obrázek 5.5: Histogram rozloženia trvania odpovedí pre test 2x20000 pripojení



Obrázek 5.6: Histogram rozloženia trvania odpovedí pre test 40000 pripojení

Z histogramov opäť vidno interval najčastejšej doby odozvy, ktorý je pre paralelné pripojenie $(0,032; 0,05)$ a $(0,03; 0,045)$ pre jednu pripojenú stanicu. Histogramy sa oproti predchádzjúcim testom veľmi nezmenili, z čoho vyplýva, že aplikácia pracuje relatívne stabilne pri krátkodobej aj dlhodobej záťaži. Pri bližšom štúdiu histogramov na nich priamo vidno, že histogramy pre paralelné pripojenie sú oproti histogramom reprezentujúcim jednu pripojenú stanicu posunuté doprava, čo vyjadruje dlhšiu priemernú dobu vybavenia jednej požiadavky.

Testovanie doby obsluhy klientov ukázalo, že aplikácia si bez väčších problémov poradí s pripojením plánovaného počtu 2000 klientov v reálnom čase bez väčších ťažkostí. Okrem toho testy ukázali stabilnú dobu odozvy pre veľké množstvo súčasne pripojených klientov a aj pre extrémnu záťaž v podobe niekoľko desiatok tisíc požiadaviek bez zakolísania výkonu.

5.2.2 Testovanie efektivity load balancera

V tejto časti zhrniem výsledky testov, ktoré boli vykonané pre overenie efektivity load balancera a zistenie jeho výkonnosti. Nasledujúce štatistiky budú vychadzať z rovnakých testov, ktoré boli rozobraté v predchádzajúcej sekcii. Pre každý jeden z testov zhrniem v tabuľkách dôležité štatistické informácie.

Test 1 – 2000 pripojených klientov	
Priemerná doba odozvy pre paralelné pripojenie	52, 77 ms
Priemerná doba odozvy pre sériové pripojenie	42, 88 ms
Celkový čas vykonania 2000 požiadaviek pre paralelné pripojenie	53, 4 s
Celkový čas vykonania 2000 požiadaviek pre sériové pripojenie	85, 75 s
Zrýchlenie doby vybavenia nasadením load balancera	1, 606x

Tabulka 5.1: Štatistiky pre prvý test

Test 2 – 8000 pripojených klientov	
Priemerná doba odozvy pre paralelné pripojenie	62, 09 ms
Priemerná doba odozvy pre sériové pripojenie	45, 47 ms
Celkový čas vykonania 8000 požiadaviek pre paralelné pripojenie	258, 97 s
Celkový čas vykonania 8000 požiadaviek pre sériové pripojenie	363, 75 s
Zrýchlenie doby vybavenia nasadením load balancera	1, 404x

Tabulka 5.2: Štatistiky pre druhý test

Test 3 – 40000 pripojených klientov	
Priemerná doba odozvy pre paralelné pripojenie	61, 17 ms
Priemerná doba odozvy pre sériové pripojenie	56, 54 ms
Celkový čas vykonania 40000 požiadaviek pre paralelné pripojenie	1285, 56 s
Celkový čas vykonania 40000 požiadaviek pre sériové pripojenie	2261, 69 s
Zrýchlenie doby vybavenia nasadením load balancera	1, 759x

Tabulka 5.3: Štatistiky pre tretí test

V tabuľkách 5.1, 5.2 a 5.3 sú spísané základné štatistické informácie o behu každého z testov. V každej z nich je uvedená priemerná doba odozvy pre paralelné aj sériové pripojenie, celkový čas vykonania všetkých požiadaviek a poslednou položkou je zrýchlenie riešenia využívajúceho load balancer oproti sériovému prístupu bez load balancera. Z týchto údajov vyplýva očakávaný záver, že load balancer značne urýchľuje odozvu aplikácie na veľké množstvo súčasných požiadaviek, avšak urýchlenie nie je v tomto prípade dvojnásobné,

ako by teoreticky malo byť pri zdvojnásobení počtu aktívnych inštancií, ale len približne 60 %. Spôsobené je to dobou spracovania požiadavky load balancerom, jeho preposlanie na inštanciu aplikácie, získanie odpovede od nej a preposlanie klientovi. Avšak aj tak je to významné urýchlenie, najmä pri väčších množstvách požiadaviek.
Zaujímavá situácia nastala pri teste 3, v ktorom sa priemerná doba odozvy pre jedno pripojenie takmer rovnala pre paralelné aj sériové pripojenie. Pri takto vysokom počte súčasných požiadaviek na jednu aplikáciu už jedna inštancia aplikácie zrejme prestávala stíhať spracovávať všetky požiadavky v takom čase, ako pri menšom počte požiadaviek. Vďaka tomu sa získalo zrýchlenie zavedením load balancera až na hodnotu takmer 76 %, čo je veľmi slušný výsledok.

5.3 Testovanie vysokej dostupnosti

Vychádzajúc z požiadaviek na vysokú dostupnosť aplikácie je nevyhnutné odtestovať, čo sa stane s aplikáciou v prípade výpadku jedného zo serverov, ktoré sú práve používané. Môžu nastať dve situácie, ktoré je potrebné odtestovať.

5.3.1 Výpadok databázového servera

Jednou z krízových situácií je výpadok primárneho databázového servera nakonfigurovaného v replike. Podľa nastavení repliky by mala replika sama zabezpečiť, aby sa pri výpadku primárneho servera nastavil za primárny server jeden zo záložných s plnou kópiou dát. Všetky dáta by mali byť kopírované na repliku real-time, teda užívateľ by počas trvania session nemal pocítiť databázový výpadok. O plynulé prepnutie medzi servermi vrámci repliky sa stará Spring Data MongoDB API. Testovanie výpadku primárneho servera malo dva testovacie prípady: úplný výpadok servera a výpadok primárneho servera. Obidva tieto varianty môžu nastať v reálnej prevádzke a je nutné zistiť, či replikácia dostatočne zabezpečuje užívateľovi neustály prístup ku aplikácii bez straty dát v prípade databázového výpadku.

Úplný výpadok databázového servera

V tomto teste sa uvažuje s úplným výpadkom databázovej inštancie vrámci repliky. Jedná sa teda o hard-crash, kedy replikačný systém nie je schopný prepokladať, že ku výpadku dôjde. Tento výpadok je teda náchylnejší na správne fungovanie replikácie. Testovanie tohto prípadu prebehlo nasledujúcim spôsobom:

1. práca s aplikáciou, pri ktorej je nutné databázová komunikácia – vytváranie nového objektu
2. zaslanie signálu SIGKILL databázovému démonovi na primárnom serveri
3. overenie v databázovej konzole, či je replika aktívna a sekundárny server sa zmenil na primárny
4. uloženie novovytvoreného objektu
5. overenie, či je objekt prítomný v grafickom rozhraní aplikácie a súčasne na aktuálne primárnom serveri

Po vykonaní všetkých spomenutých krokov bol stav databázy konzistentný, čo znamená, že replikácia databázy zabezpečuje udržanie konzistentného stavu dát v databáze, pričom užívateľ si nie je vedomí výpadku.

Výpadok primárneho servera

V tomto teste sa uvažuje s prepnutím primárneho servera na iný zo serverov vrámci repliky. Táto funkcionalita sa dá v MongoDB databáze vykonať príkazom *rs.stepDown()*, ktorý nastaví primárny server ako sekundárny. Toto je vhodné napríklad v prípade, že na danom serveri je nutné vykonať údržbu, ktorá môže spotrebovať veľký výkon, čím by sa zmenšil výkon databázového systému. Nejedná sa o hard-crash výpadok, pretože replika si je vedomá prepnutia primárneho servera a môže podniknúť adekvátne kroky. Testovanie tohto prípadu prebehlo nasledujúcim spôsobom:

1. práca s aplikáciou, pri ktorej je nutné databázová komunikácia – vytváranie nového objektu
2. prepnutie primárneho servera príkazom *rs.stepDown()*
3. uloženie novovytvoreného objektu
4. overenie, či je objekt prítomný v grafickom rozhraní aplikácie a súčasne na aktuálne primárnom serveri

Po vykonaní všetkých spomenutých krokov bol stav databázy taktiež konzistentný, čím sa opäť potvrdilo, že aj pri zmene primárneho servera za behu aplikácie nedôjde k výpadku viditeľnému pre užívateľa a databáza ostane v konzistentnom stave.

5.3.2 Výpadok webového servera

Výpadok webového servera je ďalším z výpadkov, ktoré môžu nastať. Webové servery sú zaradené do load balancera a ten medzi nimi prepína podľa algoritmu Round Robin. Load balancer má taktiež nastavenú session persistance, čo znamená, že prevádzka z jednej IP adresy sa vždy pošle na rovnaký server. Táto funkcionalita na jednej strane zefektívňuje prácu vrámci jednej session, avšak pri výpadku daného webového servera užívateľ stratí všetky informácie o session, s ktorou pracoval. Toto chovanie bolo experimentálne overené nasledujúcim postupom.

1. práca s aplikáciou, prihlásenie a udržiavanie prihlasovacích údajov v session
2. zastavenie práce webového servera, na ktorý sa aplikácia pripojila
3. užívateľ bol odhlásený a presmerovaný na druhý webový server

Toto nastavenie load balancera teda na jednej strane urýchľuje prácu, keďže session nemusí byť distribuovaná na viacero serverov, avšak pri výpadku servera je session stratená. Stále je však zachovaná dostupnosť aplikácie pre užívateľa, čo je hlavným zámerom.

5.4 Porovnanie výkonnosti súčasnej a novej verzie aplikácie

Dôležitou časťou testov je porovnanie výkonnosti novej verzie aplikácie oproti pôvodnej verzii. Z tohto dôvodu som vykonal meranie výkonnosti pôvodnej verzie pri záťaži 2000 a 8000 pripojení. V tabuľke 5.4 sú uvedené výsledky meraní. Posledné dva riadky čerpajú dáta z tabuliek 5.1 a 5.2. Z údajov vyplýva, že nová aplikácia pracuje pri nečinnosti load balancera zhruba rovnako výkonne, ako pôvodná verzia. Avšak pri zapnutí load balancera dochádza k ušetreniu výkonu a zrýchleniu aplikácie. Keďže sa s aktívnym load balancingom počíta, nová verzia je rýchlejšia a efektívnejšia.

Pôvodná verzia aplikácie – sériová obsluha klientov	
Priemerná doba odozvy pre 2000 pripojení	45 ms
Priemerná doba odozvy pre 8000 pripojení	40,91 ms
Celkový čas vykonania 2000 požiadaviek	90,01 s
Celkový čas vykonania 8000 požiadaviek	327,27 s
Zrýchlenie doby vybavenia nasadením novej verzie pri 2000 klientoch	1,687x
Zrýchlenie doby vybavenia nasadením novej verzie pri 8000 klientoch	1,264x

Tabulka 5.4: Štatistiky pre pôvodnú verziu aplikácie

5.5 Zhrnutie výsledkov testov

Aplikácia bola podrobená testovaniu z rôznych uhlov. Prvé testovanie nastáva už pri samotnom preklade aplikácie v podobe unit testov, ktoré overia základnú funkcionalitu pre dané triedy. Testovanie dostupnosti aplikácie preukázalo odolnosť aplikácie voči výpadku či už jedného z databázových serverov, alebo výpadku servera nakonfigurovaného vrámci load balanceru. Týmto sa dosiahol stav, ktorý sa dá prehlásiť za uspokojivý, keďže užívateľ je schopný pracovať s aplikáciou aj pri výpadku polovice serverovej infraštruktúry a takisto sú pripojení klienti schopní získavať obsah od aplikácie. Testovanie doby odozvy aplikácie na požiadavky od veľkého množstva klientov naraz preukázalo schopnosť navrhnutej aplikácie spracovať všetky požiadavky v reálnom čase s nízkou latenciou. Doba spracovania sa zásadne nelíšila pre 2000 a 40000 súčasných požiadaviek, preto má aplikácia v tomto smere ešte rezervy a požadovaných 2000 pripojených klientov zvládla bez akýchkoľvek problémov. Pri porovnávaní výkonnosti pôvodnej a novej verzie aplikácie sa potvrdil predpoklad, že nová verzia so zapnutým load balancingom významne urýchli spracovanie veľkého počtu súčasne pripojených klientov.

Kapitola 6

Záver

V tejto práci bol predstavený návrh, implementácia a testovanie aplikácie Ki-Wi Server slúžiacej na vzdialenú správu pripojených klientov. Aplikácia je navrhnutá pre beh v modulárnom OSGi prostredí, ktoré umožňuje dekompozíciu aplikácie na menšie, jednoduchšie spravovateľné časti. V práci sú predstavené teoretické základy pre návrh a implementáciu aplikácie, samotný návrh aplikácie, v ktorom sa berie ohľad na modulárnosť a vysokú dostupnosť aplikácie. V nasledujúcej časti sú popísané implementačné detaily, zoznam použitých technológií, takisto aj riešenia vysokej dostupnosti a škálovateľnosti aplikácie. V časti testovanie sú popísané testy, ktorým bola aplikácia po nasadení vystavená, dôvod vykonávania každého z testov a závery, ktoré z každého z testov plynú.

Oproti pôvodnej aplikácii došlo k zmene hlavne v práci so zariadením, formou obsahu, ktoré môže zariadenie nadobúdať a výraznou zmenou infraštruktúry, na ktorej je aplikácia nasadená. Vďaka týmto zmenám pracuje nová verzia efektívnejšie, rýchlejšie obslúži veľké množstvo klientov a je zabezpečená voči výpadku niektorého, prípadne viacerých zo svojich subsystémov.

Výsledkom mojej práce je modulárna aplikácia, ktorá ma zaručenú databázovú aj webovú dostupnosť pomocou databázovej replikácie a load balancingu. V prípade zvýšenia výkonu, respektíve zmenšenia rizika výpadku jedného zo systémov je možné pridať ďalšie servery do databázovej replikácie, prípadne load balancera.

Do budúcnosti vidím veľké množstvo rozšírení aplikácie. Aplikácia ešte nepodporuje celú funkcionalitu aktuálnej produkčnej verzie, preto je toto zjednotenie prvým krokom. Ďalšou možnosťou optimalizácie vidím zaistenie vysokej dostupnosti aj pre komunikačnú infraštruktúru, keďže v mojej verzii sa so zabezpečením RabbitMQ komunikácie nepočíta.

Literatura

[1] AMQP. http://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf, 2008, [cit. 2013-04-21].
URL http://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

[2] Integrated Load Balancer Overview.
http://docs.oracle.com/cd/E19963-01/html/821-1453/gijjm.html, 2010, [cit. 2013-04-21].
URL http://docs.oracle.com/cd/E19963-01/html/821-1453/gijjm.html

[3] The Java EE 6 Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/, 2012, [cit. 2013-04-21].
URL http://docs.oracle.com/javaee/6/tutorial/doc/

[4] MongoDB Documentation.
http://docs.mongodb.org/master/MongoDB-Manual-master.pdf, 2012, [cit. 2013-04-21].
URL http://docs.mongodb.org/master/MongoDB-Manual-master.pdf

[5] Spring Framework Reference Documentation. http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/, 2012, [cit. 2013-04-21].
URL http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/

[6] Alex Russel, Greg Wilkins, David Davis, Mark Nesbitt: The Bayeux Specification.
http://svn.cometd.com/trunk/bayeux/bayeux.html, 2007, [cit. 2013-04-21].
URL http://svn.cometd.com/trunk/bayeux/bayeux.html

[7] Alvaro Videla, Jason J.W. Williams: *RabbitMQ in Action*. Manning Publications Co., 2012, ISBN 9781935182979.

[8] Craig Walls: *Modular Java: Creating Flexible Applications With OSGi and Spring*. The Pragmatic Bookshelf, 2009, ISBN 9781934356401.

[9] Craig Walls: *Spring in Action, Third Edition*. Manning Publications Co., 2011, ISBN 9781935182351.

[10] Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Devika Gollapudi, Kim Haase, William Markito: *The Java EE 6 Tutorial: Basic Concepts*. Prentice Hall, 2010, ISBN 0137081855.

[11] Holly Cummins, Timothy Ward: *Enterprise OSGi in Action*. Manning Publications Co., 2013, ISBN 9781617290138.

[12] Kyle Banker: *MongoDB in Action*. Manning Publications Co., 2011, ISBN 9781935182870.

[13] Lennon, J.: Exploring CouchDB. http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html, 2009, [cit. 2013-04-21].
URL http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html

[14] Richard S. Hall, Karl Pauls, Stuart McCulloch, David Savage: *OSGi in Action*. Manning Publications Co., 2011, ISBN 1933988916.

[15] Thomas Erl: *SOA: Principles of Service Design*. Prentice Hall, 2008, ISBN 0132344823.

[16] Tim O'Brien, Jason van Zyl, Brian Fox, John Casey, Javen Xu, Thomas Locher, Manfred Moser: *Maven: The Complete Reference*. Sonatype, 2010, ISBN 9780984243341.

[17] Vincent Massol, Ted Husted: *JUnit in Action*. Manning Publications Co., 2003, ISBN 1930110995.

Příloha A

Obsah CD

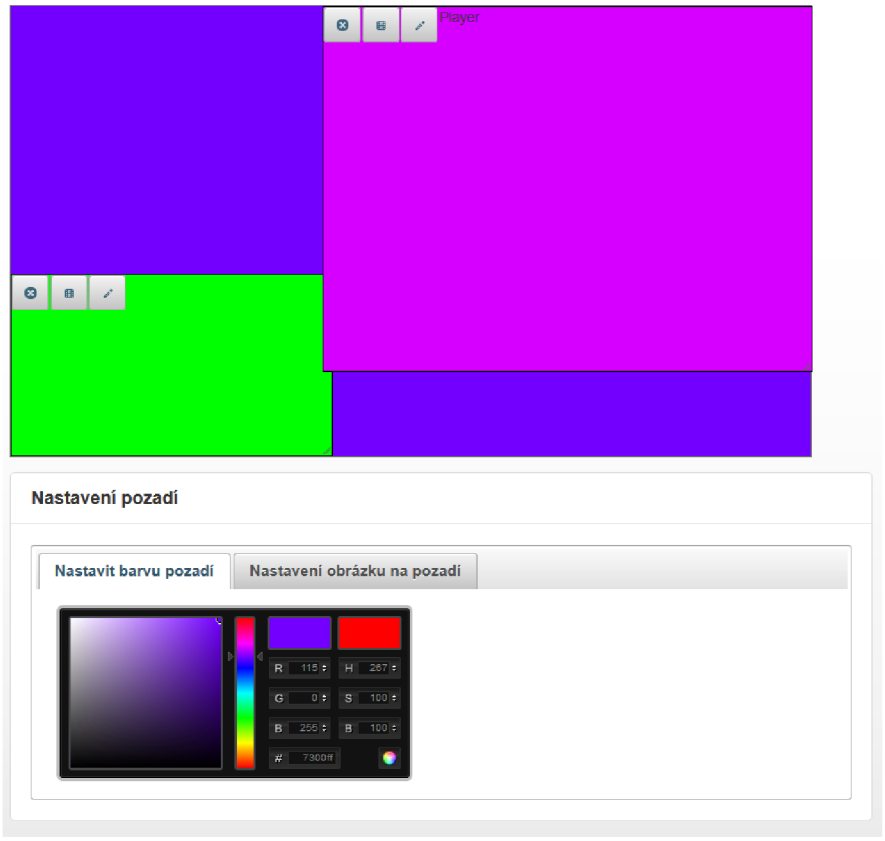
Obsah jednotlivých adresárov:

- **.** – koreňový adresár, nachádzajú sa v ňom ostatné adresáre a Maven POM súbor pre celý projekt
- **build** – adresár, do ktorého sa pri preklade ukladajú JAR archívy, ktoré je nutné nasadiť do OSGi frameworku a výstupný PAR archív aplikácie
- **kiwi_dp** – obsahuje Maven POM súbor pre build PAR archívu
- **kiwi_dp.communication** – obsahuje zdrojové kódy pre komunikačný modul
- **kiwi_dp.core** – obsahuje zdrojové kódy pre core modul
- **kiwi_dp.db** – obsahuje zdrojové kódy pre databázový modul
- **kiwi_dp.parent** – obsahuje spoločnú POM konfiguráciu
- **kiwi_dp.web** – obsahuje zdrojové kódy pre web modul
- **own-bundles** – obsahuje knižnice, ktoré museli byť manuálne upravené, aby ich bolo možné nasadiť do OSGi frameworku

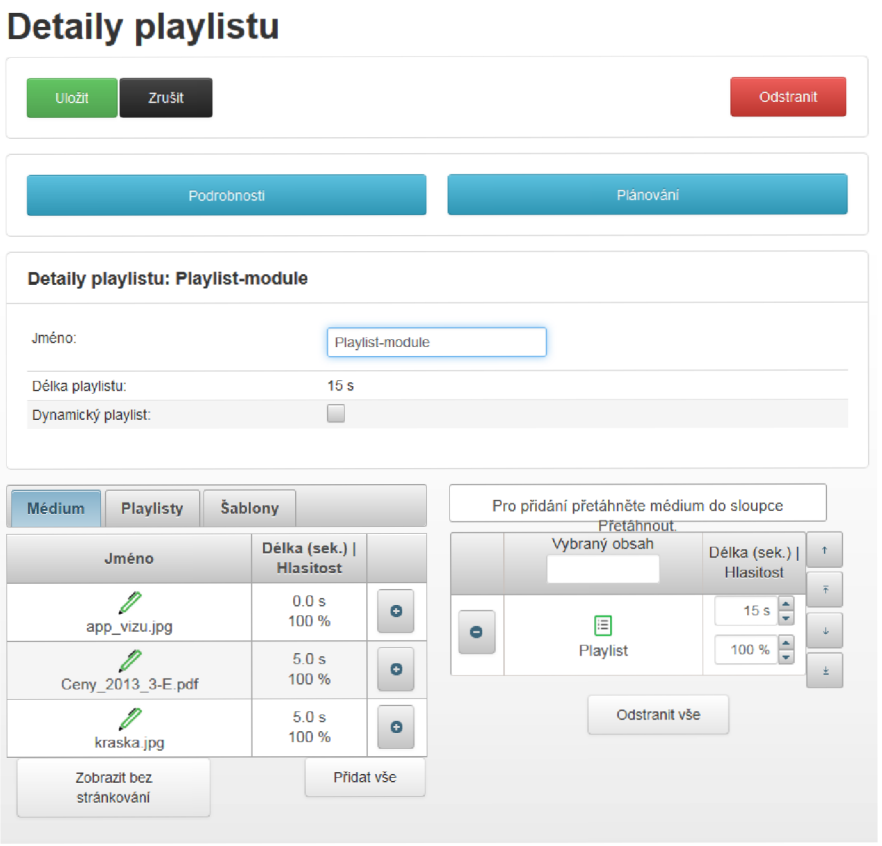
Príloha B

Screenshoty aplikácie

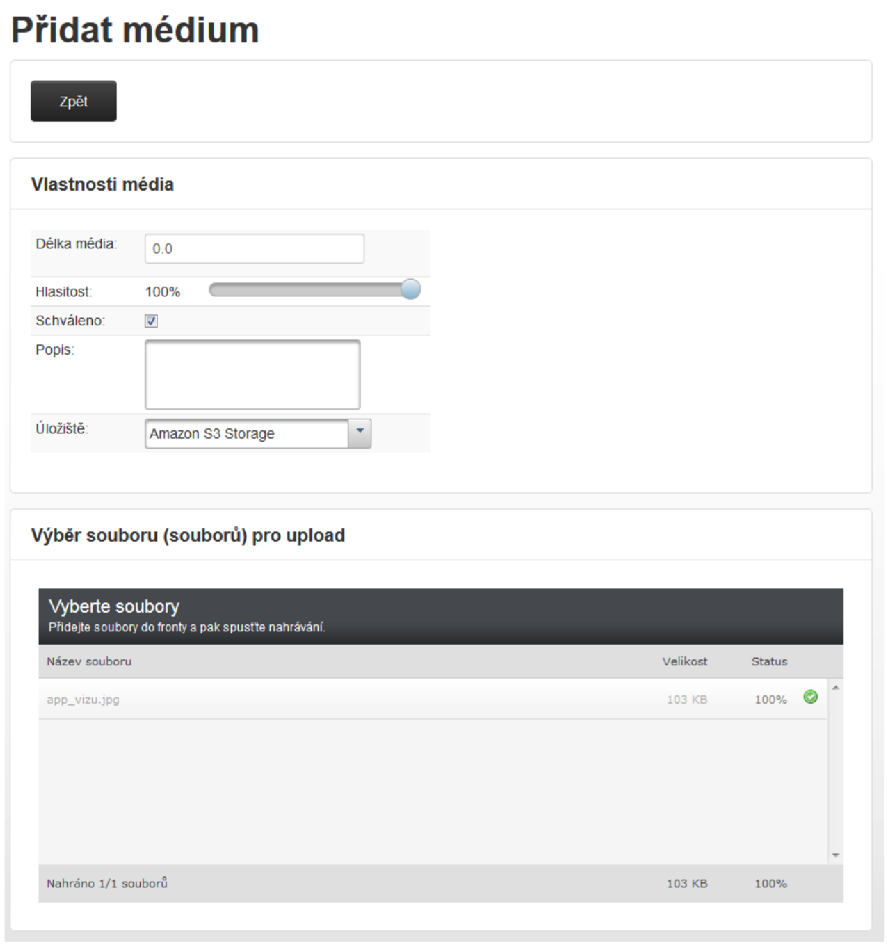
Obrázek B.1: Obrazovka so zoznamom zariadení



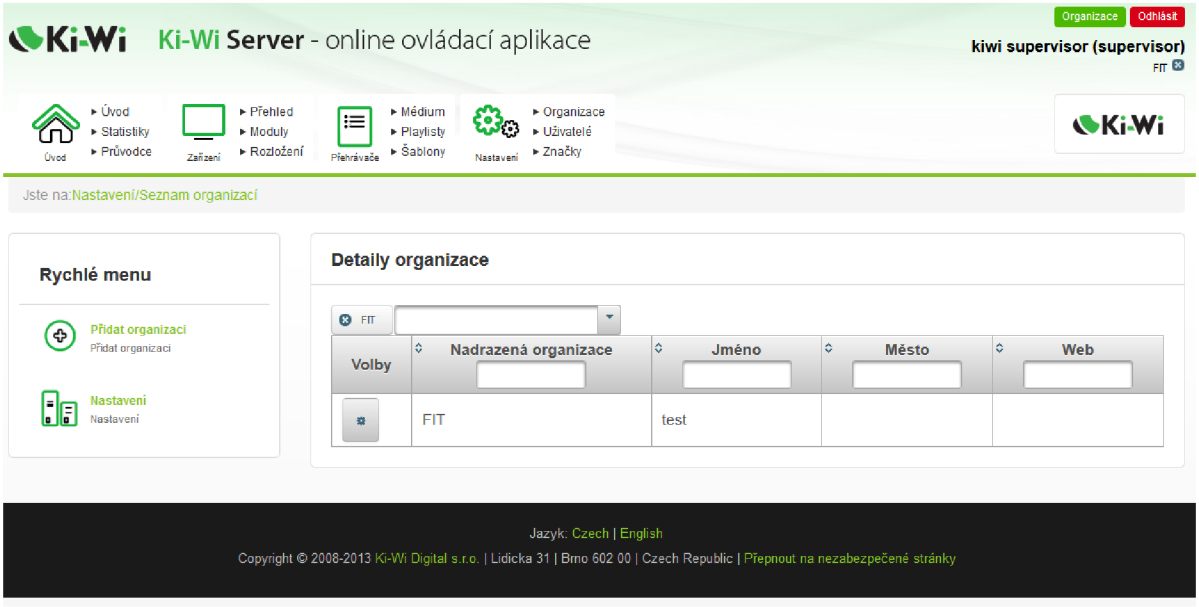
Obrázek B.2: Práca so šablónami



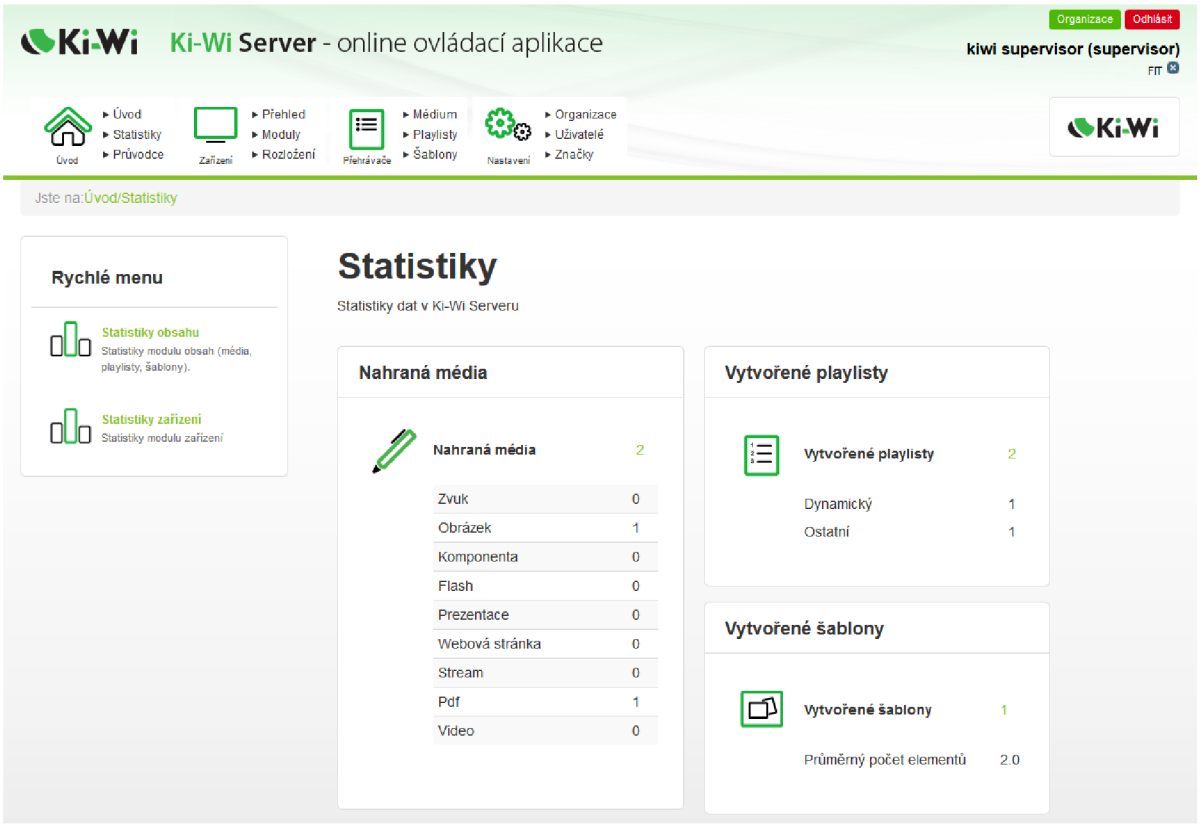
Obrázek B.3: Vytváranie obsahu pre playlist



Obrázek B.4: Nahrávanie média do systému



Obrázek B.5: Prehľad organizácii



Obrázek B.6: Štatistiky systému pre zvolenú organizáciu