Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Task Server

Pavel Šulc

Bakalářská práce

2009

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury. Byl jsem seznámen s tím, že se na moji práci vztahují práva povinnosti vyplývající ze zákona č. 12/200 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním mé práce v Univerzitní knihovně Univerzity Pardubice.

V Pardubicích dne 20.05.2009

Pavel Šulc

Poděkování

Chtěl bych poděkovat firmě Trask solutions a především Ing. Zdeňku Martincovi za vedení mé diplomové práce.

Sourhn

Cílem práce představit programovací jazyk Java EE, technologie Web Service a EJB. Dále vytvořit a zdokumentovat aplikaci běžící v kontextu aplikačního serveru, která bude zpracovávat časově a výpočtově náročné úlohy v podobě libovolné metody třídy napsané v programovacím jazyce Java. Tyto úlohy musí být možné řadit do prioritní fronty.

Klíčová slova

Java EE, web service, EJB, SOAP, JAX-WS

Title

TaskServer

Abstract

The aim of this work is to present the programing language Java EE, Web Service technology and EJB. Furthermore to create an application running in context of the application server, which will be processing tasks in form of any method from Java written class. These tasks must be possible to arrang in the priority queue.

Keywords

Java EE, web service, EJB, SOAP, JAX-WS

Obsah

Seznam obrázků

Seznam zkratek

Při čtení tohoto dokumentu se předpokládá jistá odborná znalost. Vypsány jsou tudíž pouze nejčastěji používané termíny – zkratky použité v tomto dokumentu.

EJB	Enterprise Java Bean
GUID	Globally Unique Identifier
HTTP	HyperText Transfer Protococol
Java EE	Java Platform, Enterprise Edition
JAX-WS	Java API for XML Web Services
JMS	Java Message Service
JVM	Java Virtual Machine
MDM	Master data managment
SOAP	Simple Object Access Proto
XML	Extensible Markup Language
WS	Web Service
WSDL	Web Services Description Language

Kapitola 1

1 Úvod

Úvodem bych rád představil sebe a společnost, ve které pracuji na pozici java developer a která mi zároveň zaštítila tuto bakalářskou práci, za což jsem jí vděčný.

Jsem zaměstnancem firmy Trask solutions, konzultační a technologické společnosti, která pomáhá svým zákazníkům rychleji a efektivněji inovovat. Trask solutions je respektovaným partnerem mnoha významných nadnárodních společností.

Pracují v týmu, jenž vyvíjí aplikaci pro správu master dat. Tato aplikace je psaná v jazyce java a slouží jako back-end obsahující bussines logiku zahrnující uchovávání, generalizaci, historizaci, validaci a další operace s master daty klientů. Tento projekt jsme koncem loňského roku rozšířili o webový front-end, postavený na frameworku struts 2.

1.1 Cíl práce

Cílem této práce je navrhnout, implementovat, zdokumentovat a otestovat aplikaci umožňující spouštění úloh napsaných v jazyce Java na vzdálením serveru. Tato implementace by měla být schopná přenést výpočtově náročnou logiku z klienta na server. Tento server může být zároveň na stejném fyzickém stroji jako databáze, nad kterou jsou prováděny úlohy. Jednotlivé úlohy musí být řazené do fronty a zpracovány na základě dané priority.

Primárním záměrem výsledné aplikace je zpracovávání logiky obsahující specifické operace nad databází přímo na databázovém serveru. Přenesení zpracování z klienta na server by mělo vést k výraznému snížení síťového provozu a v závislosti na výkonu serveru také k výraznému zrychlení prováděných úloh.

Důraz musí být kladen na vzájemné ovlivňování úloh. Úlohy mohou obsahovat

transakční operace nad databází, ve které jsou vzájemné interakce nežádoucí. Z tohoto důvodu by úlohy měly být zpracovávány jedna po druhé, v pořadí daném umístěním v prioritní frontě.

V případě vzájemně se neovlivňujících úloh (tj. úloh pracujících s různými daty) by mělo být možné přeskočit prioritní frontu a přímo úlohu spustit.

Vzdálené volání serveru má probíhat pomocí technologií nezávislých na platformě ani na implementačním jazyku. Díky univerzálnosti tohoto řešení je možná jednoduchá integrace nezávislých aplikací. Mělo by tak být teoreticky možné volat z klienta napsaného v programovacím jazyce Net aplikaci napsanou v jazyce Java.

1.2 Přehled současného stavu

Komunikace klient-server probíhá prostřednictvím Web Service implementované v EJB 3.0. Pro tvorbu web service bylo použito Java Api for XML Web Services (JAX-WS), který je standardní součástí Java EE API. Serverová aplikace naslouchá pouze na jednom jediném TCP portu a požadavky a výsledky jsou mezi klientem a serverem přenášeny v podobě XML souborů.

Současná verze je napsána v jazyce Java EE. Pro vývoj jednotlivých serverových komponent byly zvoleny Enterprise Java Beans (EJB) v3.0.

Prioritní fronta je implementována pomocí perzistentní Message Queue, se kterou aplikace komunikuje prostřednictvím Java Message Service (JMS). Z klienta lze vypnout či povolit zpracovávaní úloh pro jednotlivé fronty. Message Queue nám umožňuje asynchronní komunikaci pomocí zpráv v XML podobě, tyto zprávy jsou ve frontě umístěny tak dlouho, dokud si je někdo nevybere.

Aplikace je optimalizována pro aplikační server Glassfish community release v2, který obsahuje i vlastní implementace open MQ od Sun Microsystems.

Kapitola 2

2 Teoretická východiska

Kapitola se věnuje popisu vlastností použitých technologií a možností implementace specifických částí aplikace. Smyslem této kapitoly není poskytnout vyčerpávající popis jednotlivých technologií a postupů, ale spíše jejich přiblížení.

2.1 Prioritní fronta

O tom, že Task server byl primárně určen pro transakční operace nad databází jsem se již zmínil v kapitole 1. Vzhledem k těmto operacím bylo potřeba přijít s řešením, které by tyto vzájemné interakce vylučovalo. Jako nejvhodnější se ukázalo použití prioritní fronty a zpracování maximálně jedné úlohy naráz.

Způsobů implementace prioritní fronty se nabízelo hned několik. Jednou z možností implementace prioritní fronty byla fronta implementovaná databázově za použití Java Persistence Api (JPA). Jako další, z technologického hlediska atraktivní řešení, se jevilo použití libovolné implementace JMS API.

Po delší úvaze bylo rozhodnuto, že fronta bude implementována prostřednictvím Sun OpenMQ. Hlavní výhodou tohoto řešení je použití pouze standardního Java API. Serverová aplikace se tímto stala také databázově nezávislá a odpadlo tím řešení databázově specifických operací.

2.2 Webová služba

Komunikace mezi klientem a serverem je zprostředkovávána pomocí webové služby. Webová služba (Web Service) je řešení, jak spolu aplikace mohou komunikovat pomocí sítě, jsou často spojovány s XML a SOAP.. Požadavky jsou zpravidla zasílány pomocí POST metody HTTP protokolu. Pro porozumění webovým službám je nejdříve třeba upřesnit si několik základních pojmů se kterými se můžeme při

kontaktu s danou problematikou setkat.

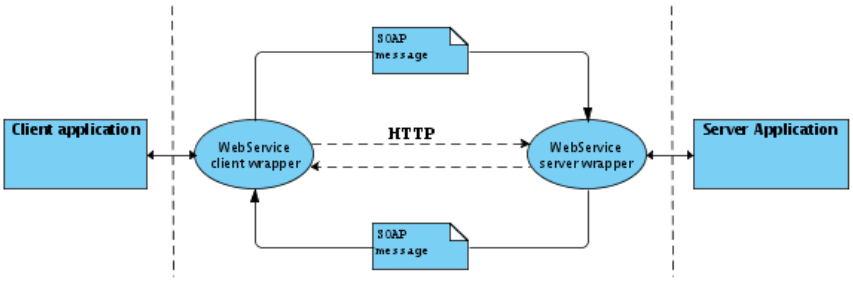
XML (eXtensible Markup Language) *je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C. Umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé účely a široké spektrum různých typů dat. Jazyk je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů. Jazyk umožňuje popsat strukturu dokumentu z hlediska věcného obsahu jednotlivých částí, nezabývá se sám o sobě vzhledem dokumentu nebo jeho částí. (3)*

SOAP (Simple Object Access Protocol) *je protokolem pro výměnu zpráv založených na XML přes síť, hlavně pomocí HTTP. Formát SOAP tvoří základní vrstvu komunikace mezi webovými službami a poskytuje prostředí pro tvorbu složitější komunikace.(1)*

WSDL (Web Services Description Language) *popisuje, co nabízí webová služba za metody a způsob, jak se jí na to zeptat. Zapisuje se jako XML. Zpravidla tedy popisuje SOAP komunikaci. Toto znamená, že WSDL popisuje veřejné rozhraní: webová služba. WSDL je často používáno v kombinaci se SOAP a XML, aby poskytovalo webové služby po internetu. Klientský program připojující se k webové službě umí číst WSDL, aby zjistil, jaké funkce jsou dostupné na serveru. Jakékoli použité speciální datové typy jsou uložené ve WSDL souboru ve formátu XML. Program může používat SOAP pro zavolání funkcí napsaných ve WSDL.(2)*

Java API for XML Web Services (JAX-WS) je Api programovacího jazyka Java usnadňující vývoj webových služeb a klientů komunikujících prostřednictvým XML. Toto Api je standardní součástí Java EE platformy. Stejně jako jiné Java EE Api i JAX-WS používá anotace, představené prvně v Java SE 5, usnadňující vývoj a deployment webových služeb a jejich klientů. Referenční implementace JAX-WS, je vyvíjena jako open source a je součástí aplikačního serveru GlassFish.

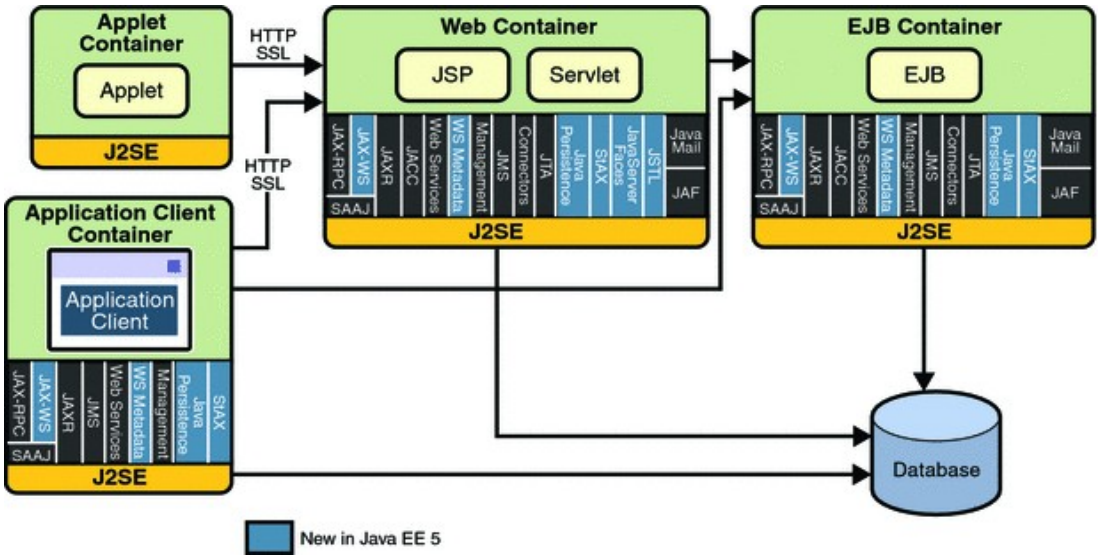
Následující obrázek zobrazuje průběh komunikace mezi klientskou a serverovou aplikací pomocí webové služby.



Obrázek 1: Web Service - komunikace klient-server

2.3 Java EE

Specifikace vznikla na základě potřeby sjednotit prostředí pro vývoj rozsáhlejších podnikových systémů v Javě a umožnit integraci s ostatními technologiemi mimo Javu. Jedná se o soupis specifikací, které by měly podporovat všechny aplikační servery a umožnit tak přenositelnost aplikací mezi různými servery. Rozdělení java EE specifikací je zobrazeno na následujícím obrázku.



Obrázek 2: Rozdělení Java EE specifikace (4)

2.3.1 Architektura

Dle Martina Kleimana (5) se jedná o komponentovou architekturu, kde každá komponenta systému má svou specifickou úlohu. Obecně platí, že má definované

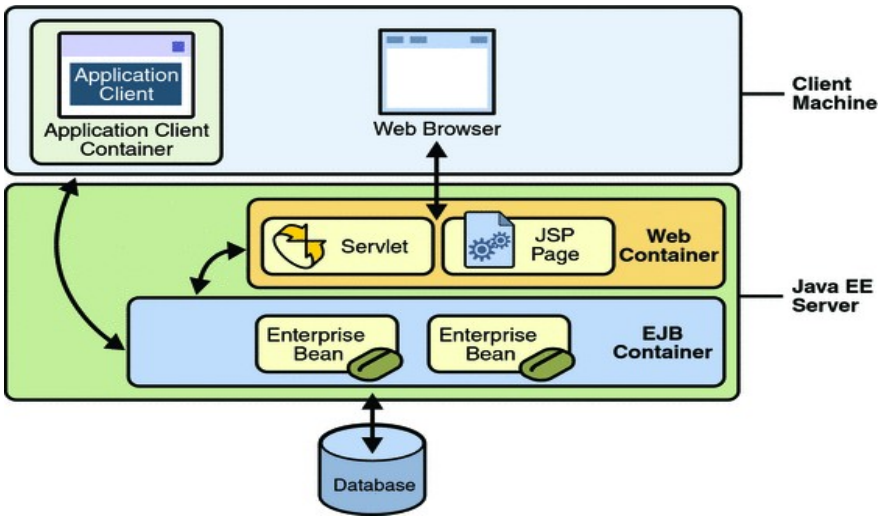
rozhraní pro integraci s dalšími komponentami.

Jak uvádí Martina Kleiman (5) rozdělení komponent dle typu implementace je následující:

- **Applety** jsou aplikace psané v jazyce Java, které jsou spuštěny v prostředí webového prohlížeče na straně klienta.
- **Webové komponenty**. Tyto komponenty reagují na HTTP požadavky a generují příslušnou odezvu. Jedná se zejména o servlety a JSP stránky.
- **Aplikační klient**. Jedná se o klasickou Java SE aplikaci která je typicky umístěna na klientském počítači. Bývá často spojovánas pojmem SWING.
- **EJB (Enterprise Java Bean)**. Komponenty, určené zejména k implementaci aplikační logiky a ke komunikaci s persistentním úložištěm nebo jinými systémy.

Následující rozdělení kontejnerů vychází z publikace (4). Prostředí aplikačního serveru, do kterého jsou nasazovány komponenty, se nazývá kontejner. Kontejnery rozlišujeme dva:

- **Webový kontejner**. Prostředí pro Web komponenty a také uložiště HTML stránek. Přístup ke komponentám je výhradně pomocí protokolu HTTP.
- **EJB kontejner**. Umožňuje přístup k EJB. Přístup je možný pomocí více protokolů.



Obrázek 3: Java EE kontejnery (4)

Výše zmiňované komponenty lze dále členit z pohledu třívrstvé architektury.

- **Presenční vrstva (Presentation layer)**. Jedná se zejména podporu komunikace s uživatelem – webové komponenty a komunikace s klientskými aplikacemi.
- **Aplikační vrstva (Business layer)**. Vlastní aplikační logika v JEE zastoupena primárně EJB.
- **Persistenční vrstva (Persistent layer)**. Vlastní persistenci dat poskytují systémy nepodléhající JEE specifikaci. JEE specifikuje rozhraní pro podporu komunikace s databázemi (JPA).

2.3.2 Adresářová služba JNDI

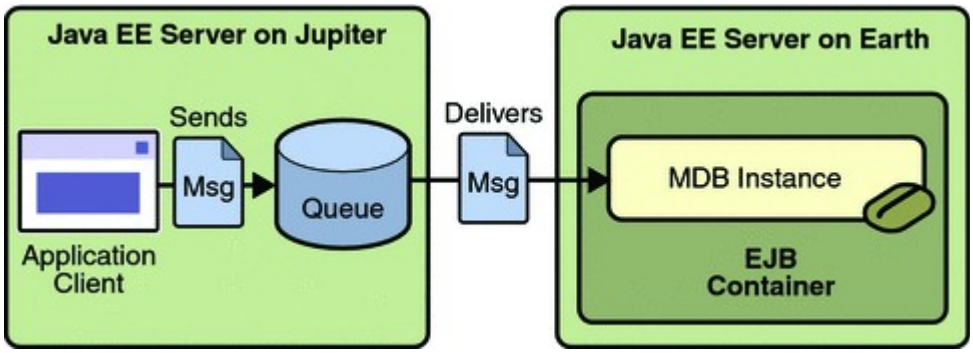
*Jmenná a adresářová služba **(Java Naming and Directory Interface)** aplikačního serveru je základním mechanismem pro vyhledání referencí na prostředky (resources), které aplikační server poskytuje. Jedná se zejména o přístup k persistentní vrstvě (databázové spojení, entitní manažer), přístup k transakčnímu manažeru nebo k beanám, které můžou být umístěny v jiné aplikaci. Jmenná služba přiřazuje názvy ve formě řetězců (JNDI jména) konkrétním objektům. (4)*

2.3.3 Fronty zpráv JMS

Java Message Service (JMS) je specifikací pro práci s asynchronní komunikací prostřednictvím zpráv. Rozdělení režimů ve kterých může komunikace probíhat ve je blíže popsáno v publikaci Martina Kleimana (5) :

- **Point to point komunikace** má jasně stanovený zdroj a cíl, kam je zpráva odeslána. Cíl je určen názvem fronty a lokací fronty (URL).
- **Publish/Subscribe komunikace** oproti tomu odešle zprávu do společného uzlu, kam se jednotliví klienti přihlásí, a pokud mají o zprávu zájem, je jim přeposlána.

Klient má k dispozici zdrojovou a cílovou frontu, kam se řadí zprávy, které se mají poslat a které byly doručeny. Vložení a vybrání z front muže být transakčně zpracováno, tj. pokud se transakce nepovede je proveden rollback a zpráva se do fronty nezařadí (nebo v případě příjmu nevybere).



Obrázek 4: Zobrazení JMS fronty (4)

2.4 Enterprise Java Beans

EJB jsou jednou z hlavních java EE specifikací. Jsou zejména vhodné pro hlavní aplikační logiku serveru stejně jako komunikaci s persistentním uložištěm. Současná verze EJB 3.0 byla vypuštěna v roce 2006 a oproti předchozí verzi je třeba vyzvednout jednoduchost tvorby EJB. Mezi největší novinky v EJB 3.0 patří například používání annotací, odstranění potřeby používat home a remote rozhraní stejně jako odstranění ejb-jar.xml.

2.4.1 EJB kontejner

Jedná se o součást JEE aplikačního serveru, který spravuje EJB. Dle Martina Kleimana (5) se stará zejména o:

- načtení konfigurace bean
- řízení životního cyklu bean včetně vytvoření instance beany
- umožnění přístupu k beanám včetně registrace do JNDI adresáře
- integrace podpory transakcí, persistence, bezpečnosti
- vícenásobné volání bean
- cachování instancí bean
- dependency injection – zajišťuje vkládání závislostí
- distribuované zpracování

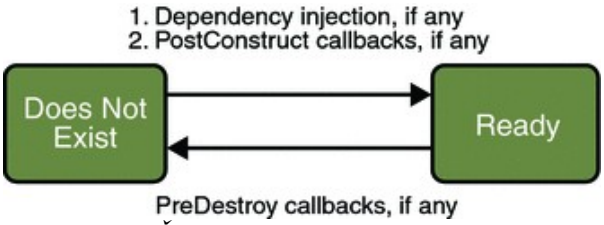
2.4.2 Správa instancí bean

Vytváření instancí a jejich správu má na starosti EJB kontejner řídící i jejich životní cyklus. To má význam i z hlediska vyřizování požadavku (tj. volání metod) na session beany. Je potřeba vyřešit problém synchronizace, kdy více klientů požaduje stejnou beanu. Aby nedocházelo ke kolizím, dostává každý klient svou vlastní instanci, která je spuštěna v separátním vláknu EJB kontejneru.

Vzhledem k tomu, že vytváření instance a kontextu beany muže být časově náročné, již vytvořené instance se ukládají do cache.. Svou cache mají ze stejného důvodu zamezení časové režie při vytváření, i vlákna, ve kterých jsou beany spuštěny. Pokud má klient (myšlen aplikační klient, webová komponenta nebo jiná beana) požadavek na beanu (volání metody), kontejner se pokusí klientovi předat odkaz na již existující beanu v cache. Pokud žádná taková neexistuje (nebo ji využívá jiný klient), vytvoří novou instanci.

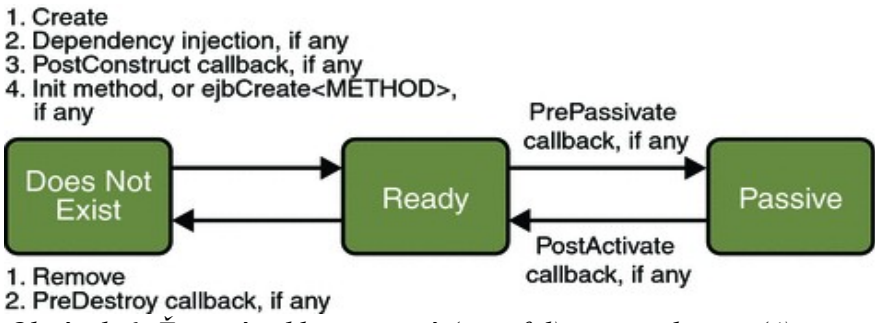
Dle publikace (5) jsou beany rozděleny do dvou základních typů :

- **Bezestavové** - volaná metoda beany není ovlivněna pořadím, v jakém jsou na beany volány. Beana nemá žádné vnitřní stavové proměnné, nebo jejich hodnota je před každým voláním metody stejná pro všechny instance. Z toho vyplývá, že kontejner klientovi může poskytnout libovolnou volnou instanci beany.



Obrázek 5: Životní cyklus bezestavové (stateless) session beany (4)

- **Stavové** - metoda závisí na předchozích voláních metod beany. Každý klient musí dostat právě tu instanci, kterou získal při minulém přihlášení, a to až do jejího odstranění z klientské strany. To přináší problém s vyvažováním zátěže.



Obrázek 6: Životní cyklus stavové (stateful) session beany (4)

2.4.3 Přístup k jiné beaně

Klient nevolá metody na cílové session beane přímo. Nejprve musí získat referenci, která se odkazuje na jeho cílovou beanu. Je to z toho důvodu, že tímto způsobem kontejner muže zprostředkovat služby jako transakce, bezpečnost, poolování apod. Pokud by se instance beany vytvořila přímo pomocí operátoru new, kontejner by neměl jak ovlivňovat volání metod a navíc by beana měla být vykonávána v prostředí klienta. Klientem se v této kapitole myslí všechny JEE komponenty, které mají přístup k EJB kontejneru, takže třeba i jiná beana apod.

Referenci poskytuje kontejner a pro jeho implementaci platí pouze to, že musí implementovat aplikační rozhraní cílové beany.

2.4.4 Vzdálené volání

Používá se zejména pokud cílová beana není dostupná ve stejné JVM. Pak je nutno použít JNDI vyhledávání, pro získání reference. Komunikace pak probíhá nad protokolem IIOP a všechny přenášené objekty (parametry metody, návratové typy) se musí serializovat (předávání hodnotou). Toto volání je časově náročnější. Rozhraní označené jako remote kontejner v EJB 3.0 se automaticky registruje do globálního JNDI adresáře. Každý, kdo má přístup do JNDI adresáře aplikačního serveru, může získat referenci na tuto beanu. Některé aplikační servery mohou mít nadstandardní vlastnosti (například IBM WebSphere 6.1), kdy při volání poznají, že volaná i volající beana je ve stejném JVM a předají parametry odkazem. Nejedná se ale o součást specifikace. (4)

2.4.5 Vkládání závislostí v EJB

Výhodou oproti předchozím verzím (EJB 2.1 a nižší) je mimo jiné vkládání závislostí (Dependency injection). S tímto principem původně přišel Spring, v EJB 3.0 je tento princip poněkud zjednodušen a nelze ho použít obecně. Jedná se o princip, kdy hodnotu nějakého atributu objektu neinicializuji pomocí své aplikace, ale inicializuje ji EJB kontejner. Atributem může být kupříkladu připojení do databáze, nebo třeba odkaz na jinou třídu. Motivací tohoto přístupu je tzv. volná vazba, kdy vazba mezi komponentami umožňuje snadnou výměnu jednotlivých komponent, aniž by se musel upravovat samotný kód aplikace. K jednotlivým referencím na EJB, ke kontextu, k databázovým zdrojům apod. lze přistupovat pomocí JNDI vyhledávání. (5)

2.5 Class Loader

Základem Task serveru je zpracovávání úloh. Jak již bylo řečeno výše, úloha je ve své podstatě specifická metoda Java třídy. Volané java třídy nejsou přímou součástí releasu task serveru, nýbrž jsou umístěny na class-path aplikačního serveru. Tento class loader se pokusí nahrát java třídu a spustit její metodu.

Podobně jako u prioritní fronty i tady lze implementovat svůj vlastní class loader. Stačí rozšířit rozhraní a upravit konfigurační konstantu tak, aby ukazovala na novou implementaci. Takto by šlo použít například JAR class loader a docílit tím dynamického nahrávání tříd. Při této implementaci by mělo být teoreticky možné za běhu bez zastavení aplikačního serveru nahrazovat knihovny volaných úloh za nové verze.

2.6 Aplikační server

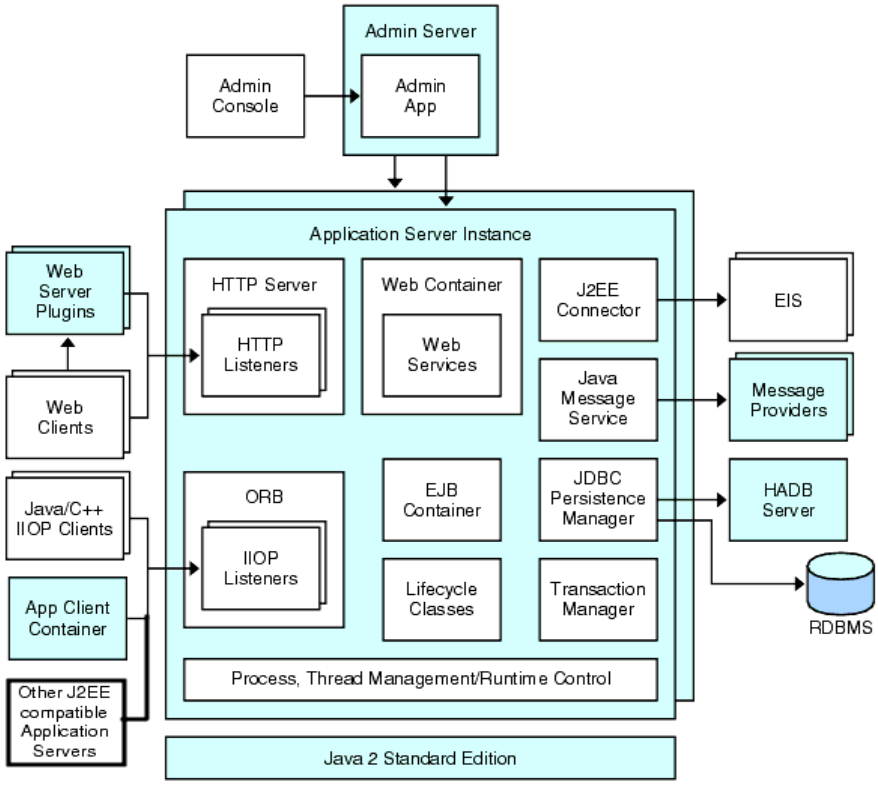
Pro běh této serverové aplikace je potřeba mít nainstalovaný aplikační server splňující následující podmínky:

- Java EE 5 kompatibilní
- EJB 3.0 kompatibilní
- JSP 2.1 a Servlet 2.5 kompatibilní

Mezi nejznámější aplikační servery splňující předešlé podmínky patří: Glassfish 2.1, Jboss 4.x, WebSphere 2.0, WebLogic 10.

Pro nasazení Task serveru byl zvolen aplikační server Glassfish. Na rozdíl od jiných WebSphere 2.0 a WebLogic 10 se jedná o free software. Další výhodou oproti Jboss je příjemné webové rozhraní a vlastní integrovaná implementace JMS.

GlassFish je open source aplikační server z dílny Sun Microsystems cílený na Java EE platformu. Komerční verze se nazývá Sun GlassFish Enterprise Server. Použitý GlassFish však je free software a je licensován pod dvojí licencí: Common Development and Distribution License (CDDL) a GNU General Public License (GPL).



Obrázek 7: Glassfish enterprise server

Kapitola 3

3 Implementace

Výsledná implementace zamýšlené aplikace je rozdělena do tří projektů. První hlavní projekt je parentem pro dva své moduly: klient a server. Sestavování výsledných archivů probíhá standardní cestou pomocí apache maven.

3.1 Klient

Součástí tohoto projektu je i jednoduchá demonstrační aplikace sloužící jako klient. Vzhledem k web service rozhraní lze klientskou aplikaci napsat v libovolném programovacím jazyce. I přes tuto skutečnost budeme nadále uvažovat pouze klientské aplikace psané v jazyce Java.

Dle publikace (6) může klient přistupovat ke vzdálené webové službě dvěma základními způsoby:

- **Port** – v tomto případě se ze strany klienta volá webová služba prostřednictvím dynamické proxy. Proxy pro webovou službu je vytvořena z generované služby a koncového rozhraní. Jakmile je jednou proxy vytvořena, klient může volat metody stejně jako při standardní implementaci tohoto rozhraní. Tento způsob je dále rozveden níže při popisu způsobů generování klientské aplikace.
- **Dispatch API** – je používáno pro pokročilé vývojáře XML, kteří preferují použití XML na úrovni java.lang.transform.Source, nebo javax.xml.soap.SOAPMessage. Vzhledem k technické náročnosti se tímto způsobem dále v textu nebudeme zabývat.

Způsobů jak vytvořit proxy je několik. Od verze Java EE 5 se stalo majoritním JAX-WS API na kterém jsou postaveny i všechny zde uváděné způsoby. Podrobně se

v této podkapitole budeme věnovat následujícím technikám:

- použití JAXWS-API
- použití jaxws-maven-plugin
- použití standardního TaskServer API

Pro následující ukázky kódu je nutné předpokládat tyto definované konstanty:

Konstanty definované

```
    String BEAN_WS_GATEWAY    = "TSGatewayBean";
    String WSDL_GW_SERVICE    = BEAN_WS_GATEWAY + "Service";
    String WSDL_GW_PORT       = BEAN_WS_GATEWAY + "Port";
    String WSDL_NAMESPACE     = "http://ejb.core.ts.trask.cz/";
    String WSDL_HOSTNAME      = "localhost";
    String WSDL_HOSTPORT      = "8080";
    String WSDL_GW_LOCATION   = "http://" + WSDL_HOSTNAME + ":" +
                                WSDL_HOSTPORT + "/" +
WSDL_GW_SERVICE + "/" +
                                BEAN_WS_GATEWAY + "?wsdl";
```

Použití JAXWS-API

Java API for XML Web Services (JAX-WS) 2.0 je stěžejní technologií pro vývoj webových služeb postavených na SOAP komunikaci. Je klíčovou součástí projektu Metro a webových služeb uvnitř Glassfish enterprise serveru. Úvod k tomuto API je nezbytný i k dalším příkladům, protože všechny z tohoto API vycházejí. Mimo jiné JAX-WS nabízí wsimport a wsgen nástroje pro vytváření webových služeb a jejich klientů.

Nástroj wsimport přijímá jako parametr umístění souboru WSDL a generuje portovatelné artefakty jako je koncové Web Service rozhraní (service endpoint interface – SEI). Wsgen oproti tomu přijímá třídu, která slouží jako web service endpoint rozhraní a generuje všechny potřebné artefakty pro deployment a volání webové služby.

Příklad použití JAX-WS API

```
//vytvoření service factory
ServiceFactory serviceFactory = ServiceFactory.newInstance();
```

```
// nahrání implementační třídy web service
URL url = new URL( cz.trask.ts.api.remote.TSGatewayBeanService.c-
lass.getResource("."), JNDIConstants.WSDL_GW_LOCATION;
TSGatewayBeanService service = (TSGatewayService) serviceFacto-
ry.createService( url , TSGateway.class);
TSGateway port = service.getPort();    // získání portu
Result result = port.startProcessor(); // volání metody spouštějící
procesor
```

Použití jaxws-maven-plugin

Jak již bylo zmíněno v předcházejícím bodě, JAX-WS poskytuje dva nástroje pro vývoj webových služeb: wsimport a wsgen. JAX-WS Maven plugin poskytuje plugin pro tyto nástroje. Plugin nabízí funkcionalitu těchto nástrojů v podobě dvou goalů: jaxws:wsimport a jaxws:wsgen.

Pro jaxws:import goal, plugin přečte WSDL souboru a vygeneruje Java třídu potřebnou pro vytvoření, deployment a volání webové služby. Goalem jaxws:wsgen se v části určené klientské aplikaci zabývat nebudeme.

Příklad definice wsimport goalu pro jaxws-maven-plugin

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>jaxws-maven-plugin</artifactId>
   <executions>
      <execution>
   <goals>
      <goal>wsimport</goal>
   </goals>
   <configuration>
      <wsdlUrls>
   <wsdlUrl>

   http://10.2.44.1:8080/TSGatewayBeanService/TSGatewayBean?wsdl
   </wsdlUrl>
      </wsdlUrls>
      <packageName>cz.trask.ts.wcf</packageName>     </configu-
ration>
```

```
      </execution>
   </executions>
</plugin>
```

Použití standardního TaskServer API

Pro demonstrační účely byla vytvořena jednoduchá klientská aplikace napsaná v programovacím jazyce. Pro navázání spojení se serverem klient používá standardní TaskServer API , které obsahuje třídu TSGatewayBeanService. Tato třída je stejná jako hlavní třída vygenerovaná pomocí jaxws-maven-pluginu, jediný rozdíl je v použití konstant pro jednotlivé parametry služby.

Ukázka TaskServer Service třídy

```
@WebServiceClient(name = JNDIConstants.WSDL_GW_SERVICE,
        targetNamespace = JNDIConstants.WSDL_NAMESPACE,
        wsdlLocation = JNDIConstants.WSDL_GW_LOCATION)
public class TSGatewayBeanService extends Service {
    ...
    public TSGatewayBeanService() {
        super(TSGATEWAYBEANSERVICE_WSDL_LOCATION,
              new Qname(JNDIConstants.WSDL_NAMESPACE,
              JNDIConstants.WSDL_GW_SERVICE));
    }

    @WebEndpoint(name = JNDIConstants.WSDL_GW_PORT)
    public TSGateway getTSGatewayBeanPort() {
        return super.getPort(new Qname(JNDIConstants.WSDL_NAMESPA-
CE,
                              JNDIConstants.WSDL_GW_PORT), TSGa-
teway.class);
    }
    ...
}
```

Tato demonstrační aplikace slouží zároveň pro testování serverové aplikace. Obsahuje sadu Unit testů simulující nejrůznější situace nastávající při komunikaci se serverem.

V následující ukázce je vidět plná konstrukce objektu reprezentující úlohu. V konstruktoru je patrné, že bude spuštěna metoda valueOf třídy java.lang.Integer s prioritou 3. Dále je vidět nastavení počtu a typy parametrů této metody pomocí metody setParameterTypes, metoda setParameters již přijímá hodnoty těchto parametrů v podobě pole typu Object. Tato konstrukce se může zdát komplikovaná, ale pokrývá libovolný počet parametrů složených z primitivních typů. Bohužel pomocí tohoto způsobu není možné předat pole určitého typu jako parametr, to lze provést nastavením metodou setSimpleParameters – tou lze nastavit pole String jako parametr metody.

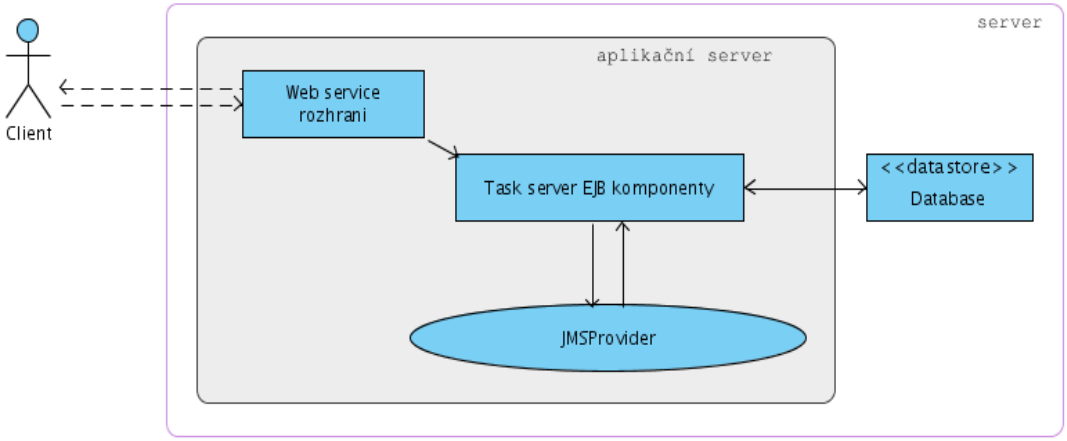
Ukázka použití TaskServer API Service

```
    @Test
    public void testConnection() throws Exception {
        TSGatewayBeanService service = new TSGatewayBeanService();
        TSGateway port = service.getTSGatewayBeanPort();
        Result r = null;

        // construction of task: java.lang.Integer.valueOf(3);
        PTask newTask1 = null;
        newTask1 = new PTask("java.lang.Integer", "valueOf", 3);
        newTask1.setParameters(new Object[] { new String("3") });
        newTask1.setParameterTypes(new Class<?>[] { String.c-
lass });
        newTask1.setQueue(TEST_QUEUE);

        // CREDENTIALS
        r = port.setCredentials(new Credentials("user",
"password"));
        assertEquals(Result.OK, r.getCode());

        // ADD TASK
        LOGGER.debug("-- task:" + newTask1.toString());
        r = port.addTask(newTask1);
        assertEquals(Result.OK, r.getCode());

        // starting processor
        r = port.startProcessor();
        …
```

```
    }
}
```

3.2 Server

Pro úspěšný běh serverové aplikace je nezbytné mít nainstalovaný aplikační server, libovolnou implementaci JMS fronty a úspěšně nadeployovaný TaskServer. Vývoj aplikace probíhal na Glassfish aplikačním serveru, který má integrovanou OpenMQ implementaci JMS od Sun Microsystems. Příjemné webové uživatelské rozhraní umožňuje deployment EJB modulů z lokálního počítače na vzdálený server.



Obrázek 8: Struktura serveru

3.2.1 EJB komponenty

Jak již bylo zmíněno výše hlavní logika serverové části aplikace je obsažena v několika komponentách postavených na EJB 3.0. Tyto komponenty jsou podporovány standardním TaskServer API pro logování a konfiguraci serveru pomocí properties souborů.

TSGateway komponenta

TSGatewayBean komponenta slouží jako hlavní vstupní brána z jiných systémů. Jedná se o stateless session beanu implementující WebService rozhraní. Jelikož jde o jediné vstupní místo, musí implementovat veškeré možné operace s Task serverem. Mezi tyto operace patří například:

- nastavení přihlašovacích údajů
- přidávání a odebírání úloh
- přímé spuštění úlohy
- spuštění jedné a více úloh dle priority dané frontou
- spuštění procesoru úloh pro určité fronty

Vzhledem k poměrně komplikované konstrukci úloh na straně klienta se nabízí možnost customizace webové služby a přenos logiky konstrukce úlohy z klienta na server..

Customizace webové služby:

```
public Result executeRDSTask(String user, String role, String
password, String component, String definition) {
      LOGGER.info(">> WS:CALL - executeRDSTask <<");
      Result result = null;
      PTask rdsTask = new Ptask(RDSSpec.COMMAND_RUNNER_CLASS,
                                RDSSpec.COMMAND_RUNNER_MEHTOD, 0);
      String[] params = new String[] { "ConnectionManager=" +
      RDSSpec.DEFAULT_CONNECTION, "user=" + user,
                                "role=" + role, "pass=" + password,
component,
                                 definition };
      rdsTask.setSimpleParams(params);
      rdsTask.setId(UUID.randomUUID().toString());
      ...
      result = executeTask(rdsTask);

      LOGGER.info(">> WS:END - executeRDSTask <<");
      return result;
}
```

TaskProcessor komponenta

Uvnitř této komponenty se, jak již název napovídá, zpracovávají jednotlivé úlohy. Komponenta sama kontroluje JMS frontu na výskyt nezpracovaných úloh. Tato kontrola mohla být implementována buď message driven beanou, nebo vlastní imple-

mentací pomocí beany používající timer service. Zvolena byla vlastní implementace algoritmu z důvodu větší kontroly nad intervaly kontroly fronty. Tyto kontroly se dají navíc pro jednotlivé fronty povolit, případně zakázat.

EJB Timer service je služba poskytovaná EJB kontejnerem aplikačního serveru. Tato implementace nám umožňuje v určitém časovém intervalu kontrolovat frontu úloh zda-li nečeká nějaká úloha na zpracování. Rozsah časového intervalu je daný buď konstantou nebo vstupem z klientské aplikace.

Implementace metody spouštěné prostřednictvím timer service:

```
@Timeout
public void handleTimeout(final Timer timer) {
      Result r = null;
      if (taskExecutor.getStatus() == StatusTypes.PROCESSOR_EMPTY) {
            PTask t = jmsWrapper.getTaskOnTop(activeQueues[0]);
            r = processTask(t);
      }
      ...
}
```

JMSWrapper komponenta

JMSWrapper komponenta usnadňuje práci s JMS frontou. Přebírá zodpovědnost za konstrukci JMS Message a používání QueueSender potažmo QueueReceiver. Způsobů konstrukce JMS message je několik. Vzhledem k tomu, že objekt reprezentující úlohu musí být serializovatelný, aby mohl být posílán přes webovou službu zabalený v XML, používá JMSWrapper výhradně ObjectMessage.

Ukázka konstrukce ObjectMessage:

```
private void setMessageToQueue(String queueName, Object object,
String id, int priority) throws JMSException {
      QueueConnection conn = ctx.getQueueConnFacotry().createQueue-
Connection();
      QueueSession session = conn.createQueueSession(true, 0);
      QueueSender queueSender = session.createSen-
der(ctx.getQueue(queueName));

      ObjectMessage message = session.createObjectMessage();
```

```
        message.setJMSPriority(priority);
        message.setJMSMessageID(id);

        message.setObject((Serializable) object);
        queueSender.send(message);

        conn.close();
}
```

3.2.2 Task Executor

V tomto případě se nejedná o standardní komponentu, nýbrž o rozhraní. Je součástí standardního TaskServer API a může být rozšířeno v rámci customizace aplikace. Toto rozhraní definuje sadu metod a pravidel používaných samotným TaskServerem ke spouštění úloh. Nahrazením tohoto algoritmu lze docílit dynamického nahrávání spouštěných knihoven na server bez restartu aplikačního serveru.

TaskExecutor rozhraní:

```
public interface TaskExecutor {
    void loadClass(String className);
    void loadDependencies();
    StatusTypes getStatus();
    Result getResult();
    Result executeMethod(String className, String method,
                         Class<?>[] parTypes, Object[] pars);
    Result processTask(PTask task);
}
```

Současná integrovaná implementace TaskExecutoru nahrává spouštěnou třídu z class path aplikačního serveru prostřednictvím Class.forName(…) metody. Tento jednoduchý způsob byl zvolen především pro problémy s nahráváním závislostí spouštěných úloh, které se objevily při testování této složitější implementace. Metoda je dále volána pomocí objektu java.lang.reflect.Method a jeho metody invoke.

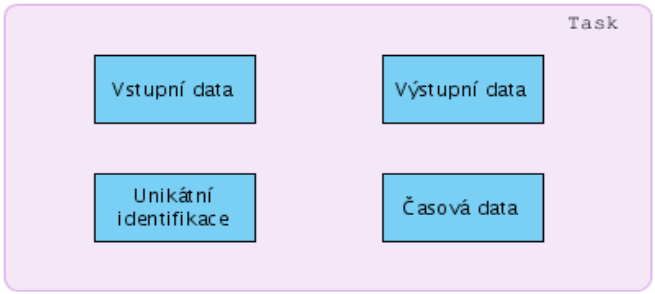
Z následující ukázky je patrné nahrání třídy, přesměrování výstupu, uložení výsledku. Zachycování veškerých výjimek bylo v příkladu z důvodů úspory místa odstraněno.

Ukázka volání metody na nahrané třídě:

```
    private static Result reflectionCall(final Object aninstance,
       final String classname, final String amethodname,
       final Class<?>[] parameterTypes, final Object[] parameters)
       throws ProcessException {

        Object res = null;
        String stdOutput = null;
        try {
            Class cTask = null;
            SimpleClassLoader cLoader = new SimpleClassLoader();
            cTask = cLoader.loadClass(classname);
            final Method amethod =
cTask.getDeclaredMethod(amethodname, parameterTypes);
            LOGGER.trace(".. checking access");
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    amethod.setAccessible(true);
                    return null; // nothing to return
                }
            });

            // redirecting output to string
            StringOutputStream sout = new StringOutputStream();
            PrintStream soutput = new PrintStream(sout);

            System.setOut(soutput);
            System.setErr(soutput);

            res = amethod.invoke(aninstance, parameters);

            // redirecting to standard back again
            System.setOut(System.out);
            System.setErr(System.err);

            // storing standard/err output
            stdOutput = sout.toString();
    } catch (…) {…}
}
```

3.3 Task

Úloha se skládá ze čtyř základních částí: **identifikátor, vstupní data, výsledná data, časové informace**.



Obrázek 9: Grafické zobrazení částí úlohy

Pod pojmem **identifikátor** se skrývá objekt ukládající unikátní identifikaci úlohy. V současné době se jedná o vygenerovaný GUID, který slouží jako ID i v JMS frontě. Ale tato identifikace lze rozšířit o libovolný identifikátor složený z primitivních datových typů.

Do **vstupních dat** patří jméno spouštěné třídy (včetně definice package), jméno volané metody, parametry metody a v neposlední řadě priorita. Parametry metody lze dále rozdělit na dva základní typy: první typ předává pole řetězců (datového typu String) a druhý, pokročilý typ předává neomezený počet parametrů složených z primitivních typů kde každý typ je předán jako java class (např. : Integer.class).

Výsledná data jsou složena z návratového objektu volané metody a dále standardního výstupu volané metody přesměrovaného z konzole do textového řetězce.

Časové informace jsou časová razítka z celého systému. Například datum zařazení do fronty, začátku zpracování, konce zpracování.

3.4 Task Server API

Místy v textu se zmiňuji o TaskServer API, bylo by tedy vhodné tuto část aplikace zmínit. Původní záměr byl takový, že bude vytvořen nový projekt importovaný jako maven dependency do všech ostatních TaskServer projektů. Tento nový projekt by obsahoval:

- například zjednodušené logovací API

- pomocné třídy pro práci s konfiguračními soubory (XML i properties)
- remote rozhraní k web service

V současné době je obdoba tohoto API umístěna jako součást serverového projektu TaskServer-core, který celý importován jako maven dependency do ostatních TaskServer projektů. Tento model přináší jednu nepříjemnou skutečnost, kterou je přenos serverové logiky včetně EJB tříd do klientské aplikace. Klient by přitom měl mít k dispozici pouze nezbytně nutnou serverovou logiku, kterou je remote rozhraní k web service.

Realizace separace logiky společné pro klientskou a serverovou část do samostatného projektu se zdá být jednoduchá, téměř ničím se to neliší od závislosti ostatních projektů na serverové logice jak je tomu teď. Skutečnost je taková, že vývoj takto rozdělených projektů byl velice obtížný. Vývojové prostření čas od času potřebovalo spustit clean pro všechny projekty, aby bralo v úvahu provedené změny v API. Aktuálně je API separováno alespoň do vlastního Java package, aby byla v budoucnu separace do samostatného projektu jednoduší.

TaskServer logging API

Jak již bylo zmíněno výše v textu, součástí TaskServer API je zjednodušené logovací API. Důvodem tohoto přístupu je fakt, že pro EJB a glassfish nelze použít log4j. Log4j samozřejmě není jedinou možností jak správně logovat, ale dle vlastních zkušeností musím říct, že se jedná jednoznačně o nejefektivnější způsob logování.

Jedna možnost jak zachovat log4j bylo použití Apache Commons Logging (JCL) jako wrapper pro log4j. Druhá atraktivnější možnost bylo použití standardního java.util.logging API, což je snadno rozšiřitelné, konfigurovatelné API které je standardní součástí JDK od verze 1.4.

Mezi těmito možnostmi byl zvolen kompromis, a to rozšíření standardního java.util.logging API s vlastní implementací třídy Logger implementující stejné metody známé z Log4j Logger třídy.

Za zmínku stojí třída cz.trask.ts.api.log.Logger. Tato třída implementuje známé metody za použití standardního java.util.logging API.

Ukázka Logger.debug()

```
    public void debug(final Object message, final Throwable t) {
        log(Level.FINE, String.valueOf(message), t);
    }
```

Z následující ukázky je patrné použití standardního java.util.logging API uvnitř log metody používané ve všech log4j-like implementací metod třídy Logger.

Implementace metody log(...) logující pomocí java.util.logging API

```
    private void log(final Level level, final String msg, final
Throwable ex) {
        if (logger.isLoggable(level)) {
            // Hack (?) to get the stack trace.
            Throwable dummyException = new Throwable();
            StackTraceElement[] locations = dummyException.get-
StackTrace();
            // Caller will be the third element
            String cname = "unknown";
            String method = "unknown";
            if (locations != null && locations.length > 2) {
                StackTraceElement caller = locations[2];
                cname = caller.getClassName();
                method = caller.getMethodName();
            }
            if (ex == null) {
                logger.logp(level, cname, method, msg);
            } else {
                logger.logp(level, cname, method, msg, ex);
            }
        }
    }
```

TaskServer configuration API

Konfigurační API se skládá z konfiguračního rozhraní, a implementace tohoto rozhraní používající properties soubory. Rozhraní definuje sadu metod pro načtení, ukládání konfiguračního souboru a získání hodnot v něm uložených. Součástí tohoto roz-

hraní je také tzv. onChangeListener, který zapříčiní monitorování změny souboru a jeho případné nové načtení.

Ukázka rozhraní konfiguračního API

```
public interface ConfigManager {

    void setConfig(String fileName) throws IOException;

    void setConfig(File onputFile) throws IOException;

    void load() throws IOException;

    void clear() throws IOException;

    void reload() throws IOException;

    String get(String property);

    void set(String property, String value);

    void setOnChangeListener(boolean active);

}
```

Jak již bylo zmíněno toto rozhraní implementuje pouze jedna třída pro práci s properties soubory. Třída obsahuje referenci sama na sebe a instance se získává pouze pomocí metody getInstance(), může tudíž existovat pouze jedna instance této třídy.

Úryvek kódu z třídy PropertiesManager

```
public final class PropertiesManager implements ConfigManager {

    private static ConfigManager ref;
    …
    private PropertiesManager() {…}

    public static ConfigManager getInstance() {
        if (ref == null) {
```

```
            ref = new PropertiesManager();
        }
        return ref;
    }

    public void load() throws IOException {
        if (configFile != null) {
            properties = new Properties();
            properties.load(new FileInputStream(configFile));
            lastModificationDate = configFile.lastModified();
        } else {
            properties = null;
        }
    }

    public void reload() throws IOException {
        clear();
        load();
    }

    private void reloadOnChange() throws IOException {
        if (reloadOnChange
           && configFile != null
           && configFile.lastModified() != lastModificationDate) {
            reload();
        }
    }
    …
}
```

Kapitola 4

4 Závěr

Cílem této práce bylo vytvořit a zdokumentovat aplikaci určenou konkrétními požadavky. I přes toto konkrétní zadání zbývalo dost prostoru pro kreativní řešení. Výsledná aplikace se ukázala býti dosti časově i technologicky náročná. Před samotným vývojem bylo potřeba nastudovat řadu materiálů a vyřešit nejedno vzniklé úskalí. Výsledkem toho je funkční projekt rozdělený do tří modulů a připravený k použití.

Serverová aplikace splňuje všechny základní požadavky jako je: zpracovávaní úloh přímo nebo procesorem pracujícím nad prioritní frontou, komunikace s klienty přes web service a další. Pro plnohodnotné nasazení aplikace je však stále potřeba implementovat některé plánované funkcionality jako je: logování úloh a jejich výsledků do databáze, webové rozhraní pro administraci a monitoring fronty a podobně.

V současné době probíhá interní nasazení aplikace v Trask solutions. Konkrétně jako zprostředkovatel komunikace mezi Microsoft Office Sharepoint Server (MOSS) a MDM aplikací napsané v jazyce Java. MOOS v tomto případě slouží jako workflow, po jehož dokončení se z .Net aplikace prostřednictvím Web service rozhraní TaskServeru zavolá Java komponenta zpracovávající import dat.

Závěrem musím říct, že takto nabytých znalostí a zkušeností si přes veškerá negativa velice vážím. Dále bych chtěl vyjádřit své díky společnosti Trask Solutions za tuto příležitost a podporu při tvorbě mé práce.

Kapitola 5

5 Seznam použité literatury

(1) Simple Object Access Protocol. [online]. , 5. 3. 2009 [cit. 2009-10-05]. Available from www: <http://cs.wikipedia.org/wiki/SOAP>

(2) Web Services Description Language. [online]. ,22. 2. 2009 [cit. 2009-10-05]. Available from www: <http://cs.wikipedia.org/wiki/WSDL>

(3) Extensible Markup Language. [online]. , 4. 5. 2009 [cit. 2009-10-05]. Available from www: <http://cs.wikipedia.org/wiki/Extensible_Markup_Language>

(4) JENDROCK, E., et al. Extensible Markup Language. [online]. , Listopad. 2008 [cit. 2009-01-05]. Available from www: <http://java.sun.com/javaee/5/docs/tutorial/doc/>

(5) KLEIMAN, M. Porovnání EJB 3.0 a Spring 2.5. Pravha : České vysoké ucení technické v Praze, 2008. 64 p.

(6) PULAVARTHI, R. Using JAX-WS with Maven. [online]. , 18. 1. 2008 [cit. 2009-10-05]. Available from www: <http://java.sun.com/mailers/techtips/enterprise/2008/TechTips_Jan08.html>

Kapitola 6

6 Seznam příloh

A. Návod na instalaci a deployment aplikace

Příloha obsahuje stručný popis zdrojových souborů aplikace, její rozdělení na moduly a návod na sestavení jednotlivých projektů. Dále je také popsán krok po kroku návod na deployment aplikace.

B. CD s aplikačním serverem včetně nadeployované aplikace

Přiložené CD obsahuje nainstalovaný aplikační server obsahující funkční nadeployovanou verzi vyvinuté aplikace.

Příloha A
Návod na instalaci a deployment aplikace

Obsah přiloženého CD

Přiložené CD obsahuje zdrojové soubory aplikace a aplikační server optimalizovaný pro operační systém linux.

Popis zdrojových textů

Zdrojové texty jsou členěny do 3 projektů. Hlavní projekt TaskServer obsahuje dva vnořené projekty: TaskServer-core, TaskServer-client, TaskServer-webgui. Poslední jmenovaný projekt je prázdná struts 2 aplikace přípravena pro implementaci webového rozhraní pro Task Server.

Projekt TaskServer-core obsahuje hlavní zdrojové soubory k serverové aplikaci.

- *src/TaskServer-core/src/main/java/cz/trask/ts/api* obsahuje zdrojové soubory k TaskServer API usnadňující logování, práci s konfiguračními soubory, remote rozhraní k web service a další části
- *src/TaskServer-core/src/main/java/cz/trask/ts/core* obsahuje zdrojové soubory k serverové aplikaci včetně implementací EJB a TaskExecutoru.

Serverová aplikace je vložena do klientské aplikace jako maven dependency. Klientská aplikace neobsahuje žádné zdrojové soubory až na soubory obsahující unit testy.

Příklad sestavení projektu

Pro sestavení projektu je třeba z hlavního projektu spustit příkaz *mvn install*. Pro přeskočení testů lze přidat parametr *-DskipTests*. Parametr install způsobí nainstalování *TaskServer-core-1.0-SNAPSHOT.jar* do lokálního maven repository, toho je zapotřebí k sestavení klienta.

Ukázka sestavení projektu:

```
$ mvn -DskipTests install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   TaskServer
[INFO]   TaskServer-w
[INFO]   TaskServer-client
[INFO]   Struts 2 Starter
…
[INFO] TaskServer.............................. SUCCESS [2.258s]
[INFO] TaskServer-ws ........................... SUCCESS [1.183s]
[INFO] TaskServer-client ....................... SUCCESS [27.368s]
[INFO] Struts 2 Starter ........................ SUCCESS [1.890s]
[INFO] [INFO] BUILD SUCCESSFUL
[INFO] Total time: 33 seconds
```

Příklad deploymentu na server

Deployment aplikace na glassfish aplikační server může probíhat dvěma způsoby:

- ruční deployment z příkazové řádky

- deployment prostřednictvím webového rozhraní aplikačního serveru

V obou výše zmiňovaných případech je nejdříve zapotřebí aplikační server nastartovat, to se provede pomocí příkazu:

```
$ cd glassfish
$ bin/asadmin start-domain domain1
```

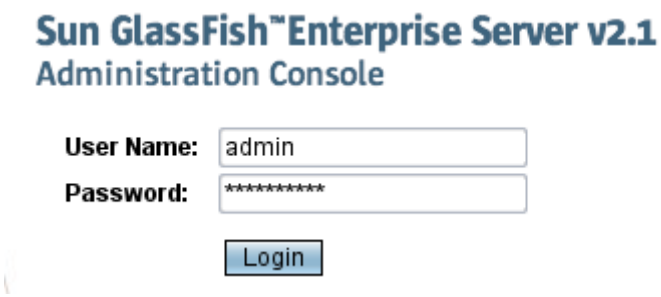
Pro ruční deployment:

```
$ bin/asadmin deploy --user admin src/TaskServer-core/target/Task-
Server-core-1.0-SNAPSHOT.jar
```

Pro deployment prostřednictvím webového rozhraní aplikačního serveru je potřeba:

- v libovolném prohlížeči spustit URL *http://localhost:4848/*

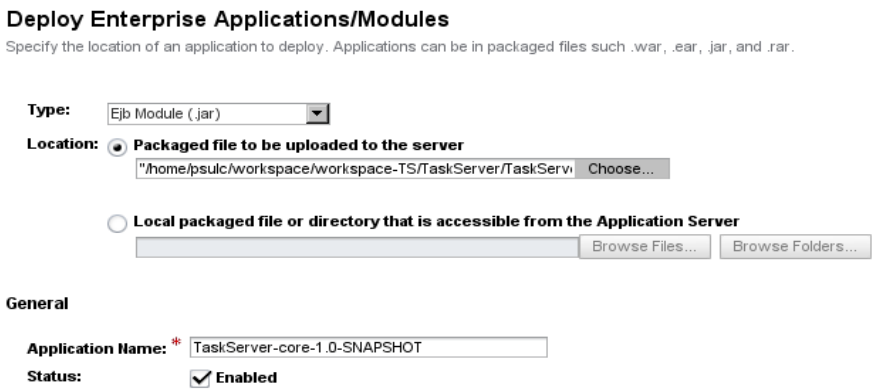
- přihlásit se do aplikace – jméno: *admin*, heslo: *adminadmin*



Obrázek 10: Glassfish - Přihlašovací obrazovka

- v levém navigačním panelu:*Applications-EJB Modules*

- kliknout na deploy – zvolit *TaskServer-core-1.0-SNAPSHOT.jar* – znovu deploy



Obrázek 11: Glassfish - deployment EJB modulu

- pro ověření nahlédněte do levého navigátoru, měl by obsahovat nový EJB module a nový WebServices modul

Pro podrobnější dokumentaci k aplikačnímu serveru Glassfish doporučuji online dokumentaci na adrese: https://glassfish.dev.java.net/docs/project.html.

Příloha B

CD s aplikačním serverem včetně nadeployované aplikace