VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

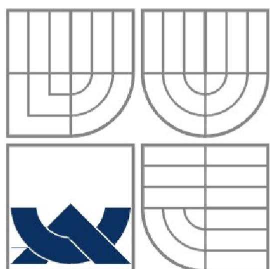
VYUŽITÍ DATABÁZE FIREBIRD V JBOSS APLIKAČNÍM SERVERU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

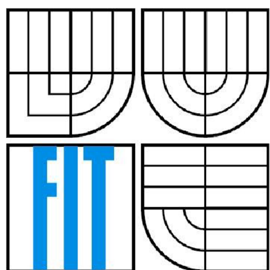
AUTOR PRÁCE
AUTHOR

Lukáš Maruniak

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INTELLIGENT TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYUŽITÍ DATABÁZE FIREBIRD V JBOSS APLIKAČNÍM SERVERU

USE FIREBIRD DATABASE IN JBOSS APPLICATION SERVER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Lukáš Maruniak
AUTHOR

VEDOUCÍ PRÁCE — Ing. Zdeněk Letko
SUPERVISOR

BRNO 2010

Abstrakt

V práci je uvedené řešení pro připojení databáze Firebird k JBoss aplikačnímu serveru.Práce se zabýva konfigurací zdrojových kódu JBoss aplikačního serveru, tak aby pracoval s databází Firebird včetně postupu na přípravu prostředí a implementací jednoduché enterprise aplikace.V závěru práce jsou analyzovány chyby konfigurace a uveden postup pro jejich napravení.

Abstract

The task was to propose a solution for the Firebird database connectivity to JBoss aplication server.The work deals with configuring source code of JBoss application server so it could work with Firebird database, including procedure for the preparation of environment and implementation enterprise aplication.In conclusion of the work there are analyzed and shown misconfigurations.

Klíčová slova

Firebird, JBoss, aplikační server, databáze, java, java enterprise edition, j2ee

Keywords

Firebird, JBoss, aplication server, database, java, java enterprise edition, j2ee

Citace

Lukáš Maruniak: Využití Databáze firebird v jboss aplikačním serveru, bakalářská práce, Brno, FIT VUT v Brně, 2010

Využití databáze Firebird v JBoss aplikačním serveru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zdeňka Letka
Další informace mi poskytli Ing. Jíří Pechanec.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Lukáš Maruniak
19.5.2010

Poděkování

Tímto bych se chtěl poděkovat mému vedoucímu Ing. Zdeňkovi Letkovi a technickému vedoucímu Jířimu Pechancovi za odborné vedení, cenné rady, konzultace, připomínky, které mi přispěli k dokončení bakalářské práce.

© Lukáš Maruniak 2009
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

Obsah....1

1 Úvod....3

1.1 Cieľ a motivácia práce....3

1.2 Obsah práce....3

2 Java a Java Enterprise Edition....4

2.1 Java....4

2.2 Úvod do J2EE....5

2.3 Architektúra J2EE....5

2.3.1 Java EE klientská vrstva....7

2.3.2 Java EE webová vrstva....8

2.3.3 Java EE bussines vrstva....9

2.3.4 Java EE Databázová vrstva....12

2.3.5 Aplikačné Servery....17

3 Databáza....18

3.1 Databázy všeobecne....18

3.2 Databáza Firebird....18

3.2.1 Classic Server....19

3.2.2 Super Server....20

3.2.3 SuperClassic....21

3.2.4 Použitie databázy Firebird....22

3.2.5 JDBC ovládače databáze Firebird – Jaybird....23

3.3 Databáza Hypersonic....24

3.3.1 Architektúra databázového systému Hypersonic....24

3.3.2 Použitie Hypersonic databázy....25

3.3.3 JDBC ovládač databáze Hypersonic....26

4 JBoss Aplikačný server....27

4.1 Aplikačný server....27

4.2 Adresárová štruktúra JBoss....27

4.3 Konfigurácia servru....28

5 Konfigurácia....30

5.1 Príprava testovacieho prostredia....30

5.2 Tvorba testovacej aplikácie....32

5.2.1 Návrh testovacej aplikácie – Spolužiaci....32

5.2.2 Implementácia testovacej aplikácie....32

5.3 Konfigurácia JBoss konfiguračných súborov a databázových zdrojov 36
5.3.1 Databázové zdroje..36
5.3.2 Java zdrojové súbory 38
5.4 Testovanie...39
5.4.1 Nastavenie testovacej sady..39
5.4.2 Štatistické výsledky..39
5.4.3 Analýza chýb 39
6 Záver...41
Literatura..42
Zoznam príloh...43

1 Úvod

1.1 Cieľ a motivácia práce

Cieľom práce bolo nakonfigurovať zdrojové kódy aplikačného serveru JBoss tak, aby po jeho zostavení pracoval výlučne s databázou Firebird. Následne je potrebné výsledky konfigurácie zdokumentovať a navrhnúť poprípade implementovať opravy konfigurácie. Testovanie konfigurácie by malo prebiehať s testovacou sadou JBoss JUnit.

1.2 Obsah práce

V úvode je vysvetlená architekúra a funkcionalita platformy Java Enterprise Edition, pretože je na nej postavený JBoss aplikačný server (JBoss AS). Ďalej práca svojím obsahom nadväzuje na architektúru databázového serveru Firebird, pričom v tejto časti je dôraz na pochopenie odlišností u jednotlivých verzií tejto databázy.V texte ďalej naväzujem na architekúru JBoss aplikačného serveru, kde popisujem na adresárovej štruktúre význam jednotlivých častí z ktorých sa JBoss AS skladá.
V praktickej časti sa jemným úvodom snažím vysvetliť postup na získanie a správne nastavenie aplikačného serveru a databázového serveru, pričom následne pokračujem vysvetlením implementácie jednoduchej aplikácie z dôvodu, aby bolo demonštrovaná funkčnosť aplikačného a databázového serveru.
Následne pristupujem k samotnej konfigurácii zdrojových súborov JBoss AS. Snažím sa nadviazať na konfiguráciu z predošlej časti a vysvetliť postup pri jej vykonávaní. Následne sa snažím dosiahnuté výsledky zanalyzovať, pričom sa venujem zásadným chybám, ktoré priamo súvisia s nastavením databázových zdrojov.
Ku teoretickej časti som sa snažil pristupovať s dôrazom na pochopenie jednotlivých technológií a zameranie sa na architekúru platformy J2EE, aplikačného serveru JBoss a databázy Firebird.
V praktickej časti som sa v úvode zameral na vysvetlenie konfigurácie na praktickom príklade pričom som nadviazal na časť, ktorá sa venuje konfigurácii zdrojových súborov, z dorazom na vysvetlenie postupu konfigurácie a analýzu chybných testov.
V závere sú výsledky práce zhodnotené.

2 Java a Java Enterprise Edition

V tejto kapitole sa budem venovať platforme J2EE[1], kde popíšem jej architektúru a jednotlivé časti (komponenty) tejto architetkúry.V úvode popíšem Java technológiu na ktorej je J2EE postavená a taktiež popíšem rozdiely oproti ostatným platformám, ktorými sú Java Standard Edition(Java SE) a Java Micro Edition(Java ME). Postupne prejdem k architekúre J2EE a k jej jednotlivým častiam z ktorých sa skladá, popíšem takisto technológie potrebné pre vývoj J2EE. Samozrejme rozsah práce mi nedovoľuje popísať jednotlivé technológie do najmenších detailov, preto u jednotlivých technológií uvediem základný princíp na ktorom sú postavené. Voľne na túto kapitolu nadviažem v ďalšom texte pri konfigurácii databázi a aplikačného serveru JBoss.

2.1 Java

Pod pojmom Java označujeme ako programovací jazyk tak aj platformu. Java je vyšší, objektovo orientovaný programovací jazyk. Java platforma zahŕňa aplikačně programové rozhranie (API) a prostredie v ktorom bežia preložené Java programy.

Existujú tri Java platformy, ktoré sa líšia podľa funkčnosti a požiadavkov, ktoré musia spĺňať.

- Standard Edition (Java SE)[2]
- Enterprise Edition (Java EE)
- Micro Edition (Java ME)[3]

Všetky Java platformy pozostávajú z Java Virtual Machine (JVM) a API. JVM je vlastne program, ktorý interpretuje bytecode, čo je medzikód, ktorý je vytvorený po skompilovaní java programových súborov a ktorý dokáže interpretovať jedine JVM. To znamená, že umožňuje beh vytvorených Java aplikácií. JVM je taktiež zodpovedný za správne uvoľnenie prostriedkov po skončení programu, ako je napríklad zdieľaná pamať. API je kolekcia programových komponentov, ktoré slúžia na vytvorenie Java aplikácii a ktoré može programátor využiť v návrhu svojej aplikácie, pričom sú zdokumentované. Každá Java platforma poskytuje vlastné JVM a API, ktoré je rozdielne od platformy, a umožňuje aplikáciám, aby mohli fungovať na akomkoľvek kompatibilnom systéme so všetkýmy výhodami, ktoré Java poskytuje - nezávislosť operačného systému, stabilitu, bezpečnosť a mnohé ďalšie.

Platforma Java SE

Pojem java je u značnej časti programátorov chápaný ako Java SE API. Java SE API poskytuje základnú funkcionalitu Java programovacieho jazyka. Definuje všetko od základných typov a objektov Java programovacieho jazyka až po triedy slúžiace na vytvorenie a správu sieťového spojenia, prácu s XML, vytvorenie zabezpečených aplikácií, prístup k databázam, či vytvorenie grafického užívateľského rozhrania (GUI). Java SE platforma taktiež pozostáva z JVM a roznych nástrojov, ktoré sú spoločne využívané vo výsledných aplikáciách.

Platforma Java EE

Java EE platforma je postavená nad Java SE platformou – čiže využíva jej vlastnosti a prostriedky. K tomu ale poskytuje rozšírené API a behové prostredie slúžiace na vývoj a správu veľkých, viacvrstvových, spoľahlivých a bezpečných sieťových aplikácií.

---

1 http://java.sun.com/javaee/

2 http://java.sun.com/javase/

3 http://java.sun.com/javame/

Platforma Java ME

Java ME platforma poskytuje API a JVM, ktoré sú určené pre malé prenosné zariadenia ako sú mobilné telefóny, smartphony a podobné prístroje.API obsahuje podmnožinu Java SE API spolu so špeciálnymi triedami použiteľných pri vývoji malých mobilných aplikácií. JVM taktiež musí spĺňať nároky na pamaťovú a výkonovú náročnosť, nakoľko je pochopiteľné, že mobilná aplikácia nemože spĺňať požiadavky, aké sú kladené na štandardnú desktopovú aplikáciu.

2.2 Úvod do J2EE

Aby sme sa mohli dostať k popisovaniu aplikačného servra JBoss[4] musíme najprv porozumieť platforme, na ktorej JBoss pracuje a tou je Java Enterprise Edition[5] (ďalej len J2EE).J2EE označuje skupinu technologií,ktoré sa používajú pri tvorbe tzv. Enterprise aplikácií.Za enterprise aplikácie možme považovať aplikačný softvér, ktorý vykonáva obchodné funkcie, ako je vybavovanie objednávok, zákaziek, plánovanie výroby, manažment zasielania správ zákazníkom ,alebo zamestnancom, účtovníctvo, atď. Tento software je typicky umiestnený na serveroch a poskytuje simultánne služby veľkému počtu užívateľov, obyčajne cez počítačovú sieť. To je v kontraste k aplikáciám, ktoré sa spúšťajú na osobnom počítači užívateľa a slúžia len jednému užívateľovi naraz.

2.3 Architektúra J2EE

Platforma Java EE používa viacvrstvový aplikačný model pre tvorbu výsledných enterprise aplikácií.Vo viacvrstvovej aplikácii je funkcionalita aplikácie rozdelená do izolovaných funkčných oblastí, ktoré nazývame vrstvy.Vačšinou viacvrstvové aplikácie majú klientskú vrstvu, strednú vrstvu a dátovú(databázovú) vrstvu. Klientská vrstva pozostáva z programu u klienta, ktorý posiela požiadavky strednej vrstve (aplikačnému serveru), kde komponenty strednej vrstvy vykonajú potrebné funkcie a odošlú odpoveď naspať klientovi, pričom prípadne pracujú s dátovou vrstvou, ktorá zapisuje, alebo číta potrebné údaje z úložíšť (databáza, XML,CSV súbory).

Výsledná Java EE aplikácia je zložená z viacerých častí - komponentov, a tieto aplikačné komponenty, ktoré tvoria výslednú enterprise aplikáciu, rozdeľujeme, podľa vrstvového modelu do jednotlivých vrstiev podľa ich funkcionality.

Pod pojmom komponenta možeme chápať technológie J2EE ako sú napríklad JSP, JMS, a mnohé ďalšie, ktoré poskytujú rozne služby(JSP – dynamické generovanie HTML stránok,JMS – zasielanie správ) kde význam jednotlivých komponentov priblížim v ďalšom texte.

Obrázok 1 znázorňuje rozdelenie jednotlivých vrstiev u Java EE vrstvového modelu.

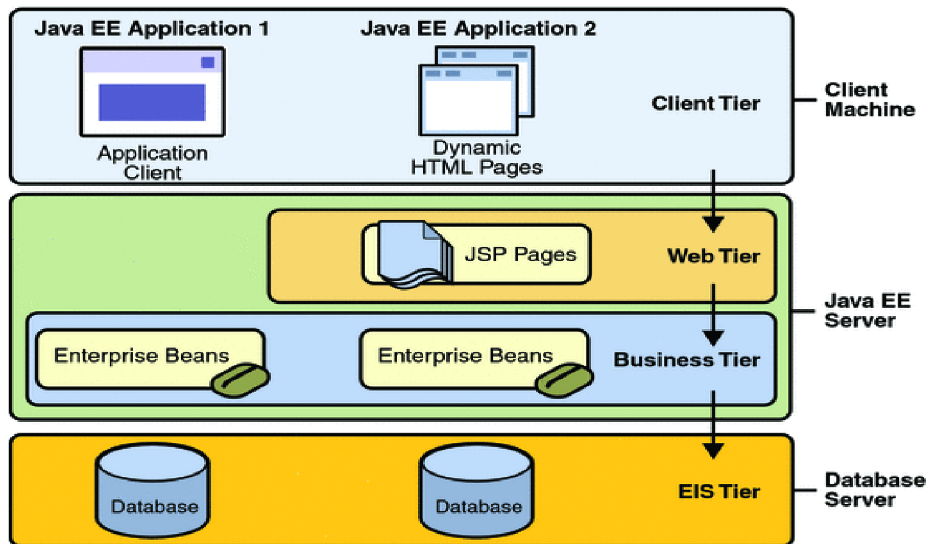
- Klientská vrstva(client tier) – obsahuje programy, ktoré bežia na koncovom zariadení (počítač klienta ,terminál)
- Komponenty webovej vrstvy(web tier) bežia na Java EE aplikačnom serveri(Java EE Server).
- Komponenty bussiness vrstvy(bussines tier) bežia na Java EE aplikačnom serveri(Java EE Server).
- Dáta sú spravované databázovým serverom.

Aj keď Java EE aplikácia može pozostávať z troch, alebo štyroch vrstiev, tak aplikácie sú väčšinou rozdelené do troch vrstiev, pretože sú distribuované medzi troma vzdialenými miestami : klientskou stanicou, Java EE aplikačným serverom a databázovým serverom. Trojvrstvové aplikácie, ktoré fungujú na tomto princípe rozširujú štandardné dvojvrstvové aplikácie typu klient – server o

4 Domovská stránka JBoss AS – http://www.jboss.org/jbossas

5 Domovská stránka J2EE - http://java.sun.com/j2ee/overview.html

viacvláknový aplikačný server medzi klientskou aplikáciou a databázovým serverom na ktorom sú fyzicky uložené dáta.



1. Architektúra J2EE prevzaté z [3]

Java EE komponenty

Každá Java EE aplikácia pozostáva z komponentov. **Java EE komponenta** je samostaná funkčná sowtfarová jednotka ktorá je zahrnutá v Java EE aplikácii,je zodpovedná za svoju časť funkcionality a komunikuje s ostatnými komponentami. Je to vlastne implementácia časti J2EE API, ktorá sa može nasadiť pri vývoji aplikácie.
Napríklad Enterprise Java Beans definujú priamo v Java kóde akým spôsobom sa majú mapovať objekty na tabuľky v databázi.Toto mapovanie samozrejme možme využiť u viacerých aplikácií.
Dovod na zavedenie takýchto komponentov je v tom, že sú štandardizované, čiže sú nezávislé od operačného systému, alebo aplikačného serveru.Takže pri vývoji určitej enterprise aplikácie je uľahčená migrácia na iný operačný systém, či aplikačný server a taktiež je uľahčená jej správa.

Java EE špecifikácia definuje nasledujúce Java EE komponenty:

- Aplikační klienti a applety sú komponenty, ktoré bežia na strane klienta (klientská vrstva).
- Java Servlety, JavaServer Faces (JSF), JavaServer Pages (JSP) komponenty sú komponenty webovej vrstvy, ktoré bežia na strane servera.
- Enterprise JavaBeans (EJB) komponenty (enterprise beans) sú komponenty bussines vrstvy, ktoré bežia na strane servera.

Java EE komponenty webovej a bussines vrstvy sú spravované aplikačným serverom.Bližšie si k týmto komponentom povieme v nasledujúcich kapitolách.

2.3.1 Java EE klientská vrstva

Klientská vrstva pozostáva z aplikačných klientov (programov), ktoré pristupujú k aplikačnému serveru a sú umiestnené na iných zariadeniach ako aplikačné servery.Programy tejto vrstvy posielajú požiadavky na aplikačný server. Server požiadavky spracuje a pošle klientovi odpoveď. Existuje viacero typov aplikácií, ktoré možu byť považované za aplikačných klientov a nemusia byť napríklad ani Java aplikácie. Sú to napríklad aplikácie bežiace vo webových prehliadačoch, samostatné Java aplikácie, alebo iné servery, ktoré bežia na inom počítači ako aplikačný server.
Za komponenty tejto vrstvy, alebo lepšie povedané typy aplikácií, ktoré možu patriť do tejto vrstvy považujeme webových (aplikačných) klientov, applety a samostatné java aplikácie.

Weboví aplikační klienti

Weboví aplikační klienti pozostávajú z dvoch častí : z dynamicky vygenerovaných stránok (HTML,XML), ktoré sú generované komponentami bežiacich na webovej vrstve, a z webových prehliadačov, ktoré zobrazujú stránky získané zo servera.Webový aplikačný klient sa niekedy nazýva **tenký klient.** Tenký klient vačšinou nevykonáva operácie nad databázou, ani nevykonáva aplikačnú(bussines) logiku.

Applety

Webová stránka prijatá od komponentov webovej vrstvy aplikačného serveru može obsahovať vstavaný applet. **Applet** je malá aplikácia napísaná v jazyku Java ,ktorá je interpretovaná v JVM inštalovanom v internetovom prehliadači ako zásuvný modul a využíva na svoje zobrazenie AWT, alebo SWING knižnicu

Samostatné Java aplikácie

Java aplikácia beží na klientskej stanici a poskytuje užívateľovi možnosť riešiť úlohy, ktoré si vyžadujú bohatšie užívateľské rozhranie, než možu poskytnúť značkovacie jazyky a internetové prehliadače.Štandardne obsahujú grafické užívateľské rozhranie (GUI) vytvorené pomocou SWING, alebo AWT API, ale ako GUI može byť použité aj iné technológie, napríklad aj príkazový riadok.
Java aplikácia može priamo pristupovať k tzv. Enterprise beans, ktoré budú vysvetlené v nasledujúcich kapitolách, bežiacich na bussines vrstve, ktoré sú zodpovedné za aplikačnú logiku. Ak to aplikácia vyžaduje, može aplikačný klient taktiež nadviazať komunikáciu pomocou protokolu http so servletom, ktorý beží na webovej vrstve.

JavaBeans komponenty

Klientská vrstva moze taktiež obsahovať komponenty, ktoré sú postavené na architektúre JavaBeans, aby mohli riadiť tok dát medzi aplikačným klientom a komponentami bežiacimi na Java EE aplikačnom serveri, alebo medzi serverovými komponentami a databázou.
Java Beans sú triedy programovacieho jazyka Java, ku ktorým sa pristupuje pomocou dopredu nadefinovaných referencií (JNDI[6]).Musia pritom spĺňať určité špecifikácie firmy Sun, aby mohli byť označené ako Java Bean.
Medzi tieto špecifikácie patrí to, že k premenným triedy Java Beanu sa pristupuje LEN prostredníctvom setterov a getterov.
Napríklad ak je premenná pomenovaná prop, tak metódy umožňujúce prístup k nej sú getProp() a setProp(nova_hodnota), poprípade pokiaľ by sa jednalo o premennú typu boolean, isProp().

6 http://java.sun.com/products/jndi

Výhodou može byť, že pokiaľ je Java Bean správne implementovaná, možete ju využívať pomocou horeuvedených metód, napríklad Java Bean pre odosielanie mailov atď.Taktiež može JavaBens trieda dočasne ukladať informácie o užívateľovi, ktorý aplikáciu využiva – napríklad počet odoslaných mailov atď.

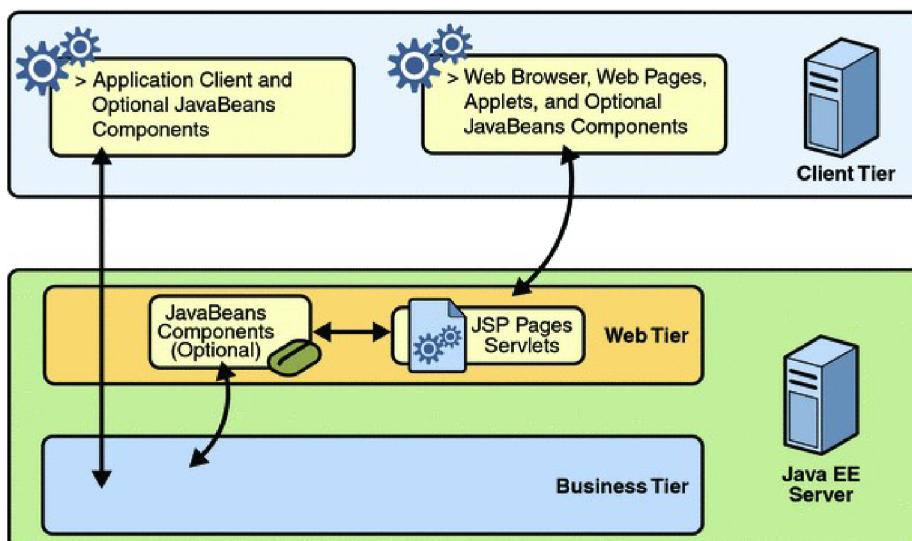
2.3.2 Java EE webová vrstva

Webová vrstva pozostáva z komponentov, ktoré riadia komunikáciu a prenos dát medzi klientskou vrstvou a bussines vrstvou. Jej primárnou úlohou je :

- Dynamicky generovať obsah v roznych formátoch pre klientskú vrstvu
- Spracovávať vstup od užívateľov z klientskej vrstvy,preposielať a vracať opdovedajúci výsledok od komponentov z bussines vrstvy.
- Udržiavať stav užívateľských dát pre každého prihláseného užívateľa
- Poskytovať základnú funkcionalitu a udržiavať určité dáta dočasne v JavaBeans komponentoch

Java EE komponenty webovej vrstvy sú buď servlety ,alebo stránky vytvorené technológiou JSP a JSF. Servlety sú triedy jazyka Java ,ktoré dynamicky spracúvavajú požiadavky a vytvárajú odpovede pre klientov.JSP stránky sú textové dokumenty ,ktoré sú vykonávané ako servlety,ale umožňujú jednoduchší a prirodzenejší prístup pre generovanie dynamických stránok.Java Server Faces je technológia ktorá používa technológie servletov a JSP a slúži na vytváranie užívateľského rozhrania pre webové aplikácie.

Ako je ukázané na obrázku 2 ,webová vrstva ,tak ako klientská vrstva možu obsahovať JavaBeans komponenty určené na kontrolu užívateľských vstupov a posielanie týchto vstupov enterprise beans komponentom bežiacim na bussiness vrstve na spracovanie.Enterprise Beans komponenty zdieľajú vlastnosti JavaBeans komponentov pričom majú pridanú funkcionalitu – ktorú priblížim v nasledujúcich kapitolách.



2. Webová vrstva a jej komunikácia [3]

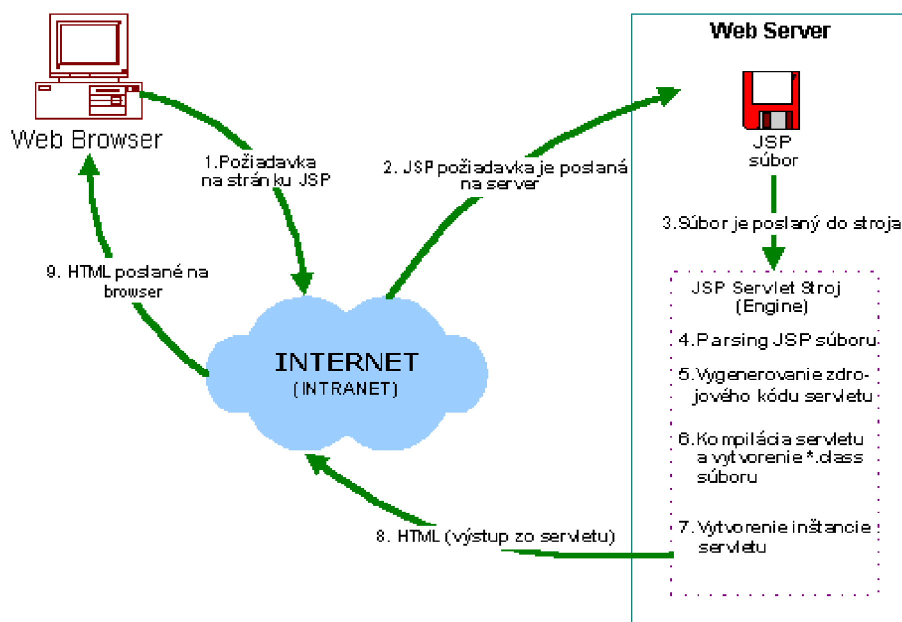
Servlety

Servlety slúžia na dynamické generovanie HTML stránok.Princíp je ten, že se vytvoria Java programy, ktoré budú fungovať podobne ako php skripty– dostanú na vstup HTTP požiadavok a vrátia kód stránky. Servlet je teda trieda ,ktorá má určité metódy ako doGet a doPost ,ktoré sa pre každý požiadavok volajú.V parametroch dostanú informácie o požiadavku a referenciu na objekt odpovedi.Podrobnejšie budú servlety vysvetlené na príklade generovania stránky u testovacej aplikácie.Pod pojmom sedenie (anglicky sessions) možeme chápať, že sú to dáta uložené na serveri, ktoré sú jedinečné pre každého užívateľa a pomocou nich možme trebárs ukladať nastavenie aplikácie u každého klienta, ako napríklad farbu pozadia aplikácie apod. Napríklad, ak užívateľ používa webovú aplikáciu a má na výber z viacerých možných grafických podôb, tak pri znovuzobrazení stránky sa mu načíta zo serveru posledne nastavený grafický profil.

Java Server Pages

Java Server Pages(JSP) je technológia, ktorá určitým sposobom nahradzuje servlety.JSP súbory obsahujú statický HTML kód a java zdrojový kód umiestnený medzi tagy <%= %>. Tento kód potom vytvára dynamický obsah stránky. Princíp fungovania JSP je zobrazený na obrázku 3.
Klient pošle požiadavok na webovú vrstvu aplikačnému servru, ten požiadavok spracuje a danú JSP stránku preloží do servletu(pokiaľ je volaná viackrát, je vytvorená iba ďalšia inštancia servletu) a odpoveď" v podobe HTML kódu pošle naspať klientovi.

Použiť JSP je vhodné hlavne vtedy, keď väčšiu časť stránky tvorí statický obsah. V opačnom prípade je vhodné použiť technológiu **servletov**.[5]



3. Architektúra JSP[5]

2.3.3 Java EE bussines vrstva

V bussines vrstve sú komponenty, ktoré majú za úlohu riešiť logiku enterprise aplikácií.Je sem naprogramované jadro celej enterprise aplikácie.Znova možeme povedať, že na vývoj sa používajú štandardizované technológie z dovodu prenositeľnosti a spravovania.Tieto komponenty označujeme jednotným názvom Enterprise Beans (enterprise JavaBeans) .

Enterprise Beans

Enterprise bean je serverová komponenta, ktorá vykonáva logiku aplikácie.Logika aplikácie je určitý kód, ktorý spĺňa účel použitia.

Enterprise beans zjednodušujú vývoj veľkých, prenositeľných aplikácií.Po prvé preto, že EJB „kontajner“ kontroluje vykonávanie enterprise beans, zabezpečuje správne chovanie pri chybách, takže sa vývojár može plne sústrediť na riešenie funkcionality aplikácie.Takže nie vývojár, ale EJB „kontajner“ je zodpovedný za prístup k systémovým službám, akými sú napríklad transakčné spracovanie, alebo bezpečnostná autorizácia.

Po druhé preto, že entepríse beans a nie klient obsajujú logiku aplikácie, takže sa može vývojár komponent klientskej vrstvy sústrediť na vývoj designu a celkového vzhľadu aplikácie a nemusí sa starať, o získavanie dát z databázy, alebo iných súborov.Vďaka tomuto prístupu možu komponenty klientskej vrstvy, dajme tomu j2me aplikácie, bežať aj na relatívne malých zariadeniach, pretože nevyžadujú veľké systémové nároky.

Po tretie, pretože enterprise beans sú prenositeľné komponenty,takže aplikácia može byť relatívne ľahko znovu postavená na inom aplikačnom servery.Rozlišujeme tri typy enterprise beans:

- Session Beans – Vykonávajú úlohy pre komponenty klientskej vrstvy.
- Entity Beans – Reprezentujú bussines objekty, ktoré sa mapujú a existujú v databázi.
- Message-Driven Beans – Chovajú sa ako java listenery pre službu Java Message Service, ktorý poskytuje správu, posielanie a odosielanie správ v rámci aplikácií.

Session Beans

Jedná sa o klasický Java objekt, ktorý obsahuje aplikačnú logiku business procesu. Inštanciu tejto komponenty je možné použiť pre viac klientov. Vždy však maximálne jednu inštanciu pre jedného klienta súčasne. Session bean može implementovať akcie ako zadávanie objednávky, kompresia dát, bankové tranzakcie atd.

Na žiadosť od klienta musí aplikačný server vytvoriť inštanciu tejto komponenty, zavolať na ňu požadovanú metódu a može ju zahodiť. Z pohľadu klienta začína session bean existovať potom, ako na ňu klient získa odkaz, a končí po uzavrení sedenia.

Session beans nie sú stále, to znamená, že nie sú ukladané do dátového úložišťa.Všetky enterprise beans vedú komunikáciu s komponentami klientskej vrstvy. Konverzáciu možme chápať ako interakciu medzi enterprise bean a klientskou komponentou a pozostáva z niekoľkých volaní metód komponenty. Konverzácia zaisťuje vykonanie určitého procesu nad komponentou, ako napríklad nákup tovaru na internete. Preto nastávajú situácie, kedy je žiadúce medzi volaniami metód udržiavať určitý stav konverzácie na strane komponenty (napríklad zadané meno a priezvisko nového zákazníka z predchádzajúcej stránky formulára). Práve z tohoto dovodu rozdeľujeme session beans na dva základné typy

- bezstavové (stateless) session beans
- stavové (stateful) session beans

Príkladom bezstavovej sesion bean može byť trieda, ktorá komprimuje video,čiže počíta pomocou zložitých algoritmov kompresiu videa a nie je potrebné uchovávať stav objektu.V kóde je uvedená anotáciou *stateless*.

Príkladom stavovej session bean može byť trieda v rámci aplikácie elektronického obchodu.Komponenta sa stará o pridanie tovaru do košíku aj o vytvorenie objednávky a pri vytváraní objednávky si musí pamatať obsah košíku – čiže musí uchovávať stav konverzácie.V kóde je uvedená anotáciou *statefull*.

Príklad bezstavovej session bean :

```
import javax.ejb.Stateless;

@Stateless
public class HelloWorldBean implements HelloWorld {
    public String getHello() {
        return "Hello World !";
    }
}
```

Na uvedenom príklade možeme vidieť bezstavovú session bean a sposob akým je anotovaná v kóde.

Entity Beans

Entity Beans riešia otázku prenositeľnosti jednotlivých aplikácií v kontraste s rozličnými databázovými systémami. Jedná sa o to, že trieda ,ktorú označujeme ako entity bean funguje na princípe objektovo-relačný mapovania.

Pod pojmom objektovo-relačné mapovanie (ORM) rozumieme automatické a transparentné ukladanie Java objektov do tabuliek v relačnej databáze s využitím metadat, ktoré toto mapovanie popisujú.

ORM pracuje v podstate tak, že transformuje data z jednej reprezentácie(Java triedy) do druhej (databázové tabuľky – MySQL,Oracle,Firebird).

ORM taktiež šetrí prácu vývojárom aplikácií, pretože správa metadat pre mapovanie je omnoho jednodušsia a prehľednejšia, než ručne spravovať množstvo kódu pre prácu s databázou.

Konkrétnu implementáciu ORM přinášajú práve entity beans. Otázka vhodného uloženia metadat pre mapovanie je vyriešená pomocou anotacií priamo v zdrojových kódoch objektov. Tieto anotácie sú štandardizované, čo umožňuje nezávislost na použitém aplikačním serveru.

Príklad Entity Bean:

```
@Entity
@Table(name="EMPLOYEE")
@SecondaryTables({
   @SecondaryTable(name="EMP_DETAIL", pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPL_ID")),
   @SecondaryTable(name="EMP_HIST", pkJoinColumns=@PrimaryKeyJoinColumn(name="EMPLOYEE_ID"))
})
public class Employee { ... }
```

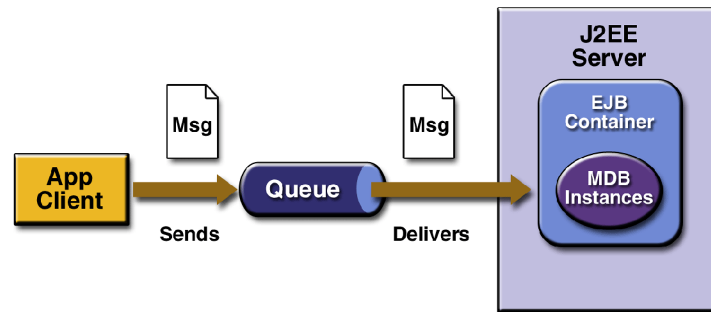
Na uvedenom príklade, je znázornená entity bean, ktorá mapuje danú triedu do tabuľky employee.Je tu aj znázornené, akým sposobom je vyriešená referenčná integrita.

Message-Driven Beans

Message-driven bean je enterprise bean,ktorá poskytuje J2EE aplikáciám možnosť spracovávať správy, ktoré posielajú J2EE komponenty (nie užívateľ).Slúži teda na komunikáciu medzi aplikáciami(nie nutne J2EE) a J2EE komponentami.

Inštancie tejto triedy reagujú na prijímanie a odosielanie správ.Správy možu byť posielané ktoroukoľvek J2EE komponentou – aplikačným klientom, inou enterprise bean-ou, webovým komponentom, alebo systémom, ktorý nie je postavený na J2EE architektúre.

Keď príde správa od komponenty, alebo iného zdroja na aplikačný server, tak sa zavolá metóda onMessage, ktorej sa predá obsah správy a onMessage metóda potom vykoná potrebný kód.



4. Architektúra aplikácie pracujúcej s Message-driven bean komponentou[3]

Na obrázku 4 je znázornená architektúra aplikácie, ktorá pracuje s message-driven bean komponentou.
Aplikačný klient (app client) pošle správu(Msg) do fronty správ(Queue).Poskytovateľ služieb – v našom prípade JBoss Aplikačný server prepošle správu inštancii message-driven bean-y, ktorá správu spracuje(pomocou metódy onMessage()).
Môžeme sem vidieť výhodu takéhoto spracovania a to v tom, že klient sa nemusí starať o to, či je správa doručená, pretože o to sa postará aplikačný server.

2.3.4 Java EE Databázová vrstva

Databázová vrstva (v zahraničnej literatúre taktiež označovaná ako enterprise information system EIS) pozostáva z komponentov, ktoré sú väčšinou uložené na inom stroji, ako aplikačný server a sú kontaktované pomocou komponentov bussines vrstvy.

Ako už názov naznačuje táto vrstva obsahuje komponenty(databázový server, súborový systém), ktoré sú zodpovedné za uloženie dát.Ako formát sa može použiť XML súbor ,textové súbory ,alebo databázový systém. Tým može byť MySQL ,Oracle ,Firebird ,MSSQL, alebo iný databázový systém.

Na prístup k databázovej vrstve sa využívajú nasledujúce technológie

- Java Database Connectivity API (JDBC)
- Java Persistence API
- Java EE Connector Architektúra
- Java Transaction API (JTA)

Java Database Connectivity API[4]

JDBC API poskytuje základné rozhranie pre prístup k databázám. Základom konceptu JDBC je využitie funkčnosti poskytovanej JDBC ovládačom, ktorý dané metódy následne prekladá do nativních volaní danej databázi. Vďaka tomu je programátor zbavený špecifického API databázi a može se naučiť jednotné rozhranie JDBC, ktoré potom použije pre prístup do ľubovoľnej databázi.

JDBC nie je určené iba pre prístup k relačným databázam, ale k ľubovoľnému formátu dát, uloženého v stĺpcovom formáte,čo možu byť i textové soubory (napr. CSV) atď.JDBC špecifikácia rozpoznává štyri typy JDBC ovládačov.

Typ 1
Tento typ ovládača využíva lokálny ODBC[7] ovládač.Aplikácia používajúca tento typ ovládača vyžaduje nainštalovanie a nastavenie lokálneho ODBC ovládača pre danú databázu.

ODBC ovládače sú špecifické pre každého výrobcu databázi a preto je práca s nimi zložitá, je nutné nainštalovať lokálne knižnice, ktoré musia byť synchronizované s ohľadom na aktuálne použitú databázu, ich administrácia je časovo náročná z dovodu nejednotnosti rozhrania, použitia a nastavenia.Z tohoto dovodu je použitie v čisto klientsky orientovaných aplikáciách komplikované práve kvoli náročnosti na konfiguráciu.

Typ 2
Ovládač typu 2 prekladá požiadavky z JDBC do určitého špecifického ovládača, ktorý je nainštalovaný na počítači a ktorý je určený práve pre jeden typ databázy.

Typ 3
Tento typ už nepoužíva žiadny natívný kód pre ovládač, ale je čisto založený na Jave a JDBC, ktoré konvertuje svoju komunikáciu do sieťového protokolu, ktorý sa spojí so vzdialeným databázovým servrom, ktorý poskytuje pripojenie k databázi. Tento server konvertuje sieťový protokol, ktorým komunikuje s klientami do databázového špecifického protokolu, ktorému databáza rozumie.Takýto model je vysoko efektívny.

Typ 4
Ovládač typu 4 je napísaný celý v Jave s plnou podporou pre optimalizáciu vzhľadom k danej databázi.Výhodou tohoto ovládača je, že klient nemusí byť akokoľvek nakonfigurovaný a nie sú nutné žiadne lokálne klientské inštalácie ovládačov.

Použitie JDBC

Ako už bolo uvedené, základom JDBC je ovládač. Tento ovládač je špecifická trieda, obvykle poskytovaná výrobcom databázi. Predtým, než budú spravené akékoľvek operácie nad databázou, je nutné ovládač nahrať a registrovať. K tomu slúži príkaz

Class.forName("trieda_JDBC_ovládača");

Registrácia JDBC ovládača je viazaná na DriverManager, čo je trieda s metódami pre správu a prácu s JDBC. V podstate ide o vrstvu medzi aplikačným kódom a JDBC ovládačom, pričom v tejto vrstve je vykonaná registrácia JDBC ovládača vo chvíli, kedy je zavolaná vyššie popísaná metóda forName(). Prakticky by mal tento ovladač zavolat pri nahraní metódu registerDriver() triedy DriverManager. Ten potom ovládač eviduje a je poskytnutý pre prácu s databázou.

Spojenie s databázou

Akonáhle je JDBC ovládač nahraný a správne zaregistrovaný DriverManagerom, je možné naviazať spojenie s databázou. Spojenie s databázou sa identifikuje ako „databázové URL“, ktoré sa skladá z týchto súčastí:

- URL musí byť uvedené „jdbc.“, čo je určené špecifikáciou.
- Potom može byť uvedený podprotokol, čo je záležitosť závislá od daného poskytovateľa databázi. Napríklad pre Firebird je nutné uviesť „firebirdsql.“.
- Posledná časť je DSN (Data Source Name), ktorý identifikuje názov databázi

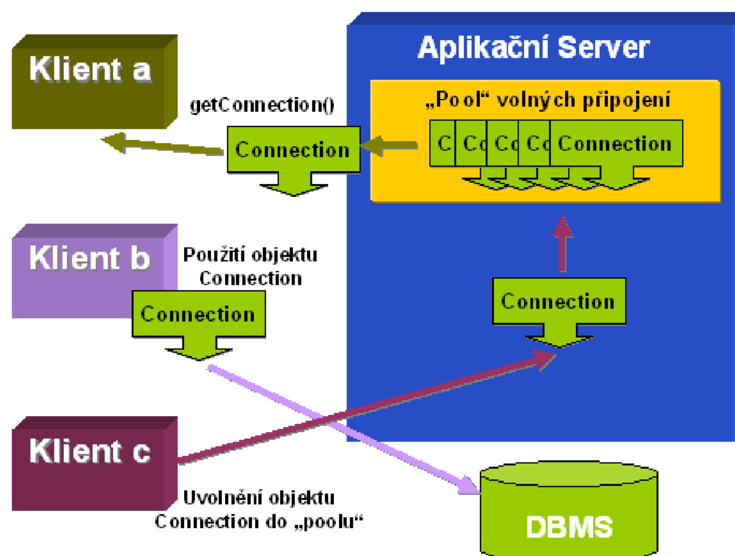
7 http://searchoracle.techtarget.com/definition/Open-Database-Connectivity

Výsledkom je potom reťazec ako napríklad : *jdbc.firebirdsql:localhost/3050*

Pooling

Pretože vytvorenie samotného pripojenia je časovo a systémovo veľmi náročná operácia, ktorá by při neustálom vykonávaní pri každom príkaze od klienta znamenala výrazné spomalenie aplikácie, tak sa tieto objekty predvytvárajú a ukladajú sa po použití spať do „poolu“, odkiaľ sa znova berú podľa potreby klienta.

Na obrázku 5 je znázornená funkcionalita „Poolingu“.



5. Pooling [4]

Klient požiada aplikačný server o pridelenie spojenia na databázu. Aplikačný server zistí, či je dostupný objekt Connection vo fronte voľných pripojení („poole“) a ak áno pridelí klientovi tento objekt, čím sa vytvorí spojenie medzi databázou a klientom. Po skončení práce klient pripojenie znova uvoľní a vráti objekt spať do fronty voľných pripojení.

Java Persistence API

Java Persistence API (JPA) je framework[8] programovacieho jazyka Java, ktorý umožňuje objektovo-relačné mapovanie (ORM). To uľahčuje prácu s ukladaním objektov do databázi a naopak.

Dá sa povedať, že rozširuje povodnú funkčnosť entity beans.Umožňuje vytvárať parametrizované SQL príkazy štandardizovaným sposobom – pomocou Java Persistance Query Language čo je jazyk podobný SQL a jeho použitie má tú výhodu, že príkazy sú nezávislé na zvolenej technológii databázi (Oracle, Microsoft…). Naviac majú objektové vlastnosti, takže v príkazoch neuvádzame konkrétne názvy tabuliek databází, alebo ich vlastnosti, ale uvádzame priamo názvy tried/atribútov, k porovnávnaniu můžeme využívat i referencie objektov.

Najlepšie bude ukázať si to na konkrétnej implementácii JPA.

Majme majiteľa (*Owner*), ktorý može mať niekoľko áut (*Car*). Ďalej predpokladajme, že každé auto može mať vždy iba jedného majiteľa a že u auta je majiteľ uvedený.

8 **Framework** je softwarová štruktúra, ktorá slúži ako podpora pri programovaní, vývoji a organizaci iných softwarových projektov. Može obsahovať podporné programy, API, návrhové vzory alebo doporučené postupy pri vývoji.Príkladom može byť Hibernate Framework.

Na príklade sú znázornené 2 entitné triedy, ktoré mapujú objekty Owner a Car na databázu – primárny kľúč v triede owner a car je uvedený anotáciou @Id a je u neho aj uvedeny sposob generovania jedinečnej hodnoty.
Taktiež je anótaciami @OneToMany, resp. @ManyToOne uvedené ako sú tabuľky(resp. Objekty) na seba referencované.
O jednotlivé operácie s databázou sa už stará Trieda ExampleEjbBean, ktorá pomocou metód persist (z triedy EntityManager, ktorá implementuje práve JPA) ukladá dáta do databázi, resp. Metódou find vyberá dáta z databázi.
Trieda Owner (entity bean) :

```
@Entity
public class Owner implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    @OneToMany(mappedBy = "owner")
    private List<Car> car;

    // gettery a settery
}
```

Trieda Car (entity bean) :

```
@Entity
public class Car implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String color;
    @ManyToOne
    private Owner owner;

    //gettery a settery

}
```

Trieda ExampleEjbBean (Session Bean) :

```
@Stateless
public class ExampleEjbBean implements ExampleEjbLocal {
    @PersistenceContext
    EntityManager em;

    public void saveOwner(Owner owner) {
        em.persist(owner);
    }

    public Owner getOwner(Long id) {
        return (Owner) em.find(Owner.class, id);
    }
}
```

Java EE Connector Architektúra

J2EE Connector architektúra(JCA) povoľuje J2EE komponentom akým sú napríklad enteprise bean-y spolupracovať z Databázovou vrstvou. JCA zjednodušuje integráciu týchto rozdielnych systémov.Každý takýto systém potom vyžaduje iba jednu implementáciu JCA. Pretože každá takáto implementácia musí podliehať špecifikácii JCA, sú takto vytvorené aplikácie (komponenty) prenositeľné naprieč všetkými J2EE aplikačnými servermi.

Špecifikáciu JCA implementuje pre rozdielny dátový systém komponenta nazývaná *Resource adapter.* Následne aplikácia, ktorá Resource Adapter využíva, komunikuje pomocou neho s týmto systémom (či už je to databázový systém, alebo server na ktorom sú uložené dáta v XML súboroch).

Resource adapter je podobný JDBC ovládaču. Obidve časti poskytujú štandardizované API, ktorými aplikácie možu pristupovať k prostriedkom mimo aplikačného servera.Pre resource adapter komponentu ale platí, že tento prostriedok nemusí byť nutne databázový systém.Zvyčajne sú Resource adaptery a JDBC ovládače vyvíjané skupinami, ktoré sa podieľajú priamo na vývoji buď databázového, alebo iného systému.

Java Transaction API

Java Transaction API podporuje správu a implementáciu mechanismu na spracovanie transakcií.

Transakcia je definovaná ako množina operácií, kde musí byť vykonaná každá operácia v danom poradí, aby sa zachovala integrita dát.

Napríklad, prenos peňazí z jedného účtu na druhý v hodnote 100 € pozostáva z dvoch krokov – najprv sa musí odpočítať položka 100 € z prvého účtu a následne sa musí pripočítať položka 100 € na druhý účet. Takže aby sme zachovali integritu dát a konzistenciu dát musia byť tieto dve operácie vykonané spolu, alebo vobec.Takže dokopy tvoria transakciu prevod peňazí.

Transakcia musí spĺňať nasledujúce vlastnosti(označované spoločne ako ACID)

- Atomicita (**A**tomicity) – buď se vykonajú všetky operácie z transakcie, alebo žiadna z nich
- Konzistencia (**C**onsistency) – beh transakcie zachovává konzistenciu zdrojov (napr. databáza splňuje integritné obmedzenia), v priebehu transakcie može dojsť k nekonzistencii, ale nie však po jej ukončení
- Izolovanost (**I**solation) – aj keď može viac transakcií bežať naraz, nesmia o sebe navzájom vedieť a musia pred ostatnými skrývať dočasné medzivýsledky (miera tohoto skrývania može byť rozna a hovoríme tu o úrovniach transakčnej izolovanosti)
- Trvanlivost (**D**urability) – po úspešnom ukončení transakcie sú všetky zmeny trvale uložené a uložené zostanú aj pri následnom výpadku systému

Pokiaľ má ľubovoľný zdroj, komponenta, alebo služba podporovať štandardné transakcie, mala by implementovať potrebné rozhrania z JTA. JTA dokonca podporuje distribuované transakcie(transakcie naprieč roznymi aplikačnými servermi) a beh transakcie nad roznymi zdrojmi (databázi,JMS,EJB).

Aby sme mohli zistiť informácie o aktuálnej transakcii, budeme potrebovať inštanciu triedy implementujúcu rozhranie Transaction Manager. To je možné buď priamym volaním JNDI, alebo pomocou dependency injection. Ďalšie sposoby vyžadujú znalosť implementačných detailov knižnice pre riadenie transakcií.

2.3.5 Aplikačné Servery

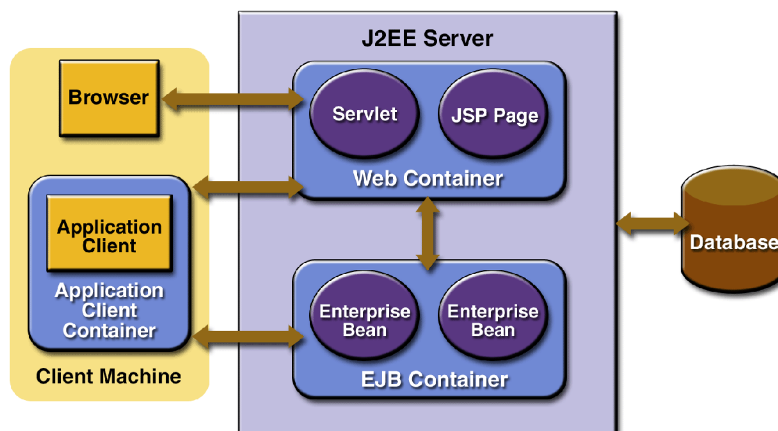
Na to aby vyššie spomenuté služby fungovali a boli pod jednotnou správou za účelom zvýšenia prehľadnosti a bezpečnosti je potrebné mať aplikačný server, ktorý implementuje tieto služby. Správu a implementáciu týchto služieb zabezpečujú tzv. „kontajnery".Vzhľadom k tomu, že sa nemusíte zaoberať implementáciou služieb, ako JPA, alebo JTA máte možnosť sústrediť sa na riešenie aplikačných problémov.

J2EE kontajner

Kontajnery slúžia ako rozhranie medzi J2EE komponentami (entrerprise beans) a funkcionalitou služieb J2EE aplikačného serveru (JNDI,JTA).Predtým než može byť vykonaná funkcia nejakej komponenty musí byť vstavaná do J2EE aplikácie a nasadená (angl.: deploy) do jej špecifického kontajneru.

Proces vstavania zahrňuje nastavenie špecifických parametrov každej komponenty pre daný kontajner, pre ktorý je určená v J2EE aplikácii.Toto nastavenie potom ponúkne možnosť využiť služby daného aplikačného serveru akými sú napríklad bezpečnosť, podpora transakcií, JNDI vyhľadávacia služba a vzdialené pripojenie.

Fakt, že J2EE architektúra poskytuje nastaviteľné služby znamená, že komponenty tvoriace výslednú enteprise aplikáciu sa možu chovať rozdielne, podľa toho na akom aplikačnom serveri sú nasadené.Napríklad, enterprise bean-a može používať zabezpečovaciu službu (JAAS), ktorá jej umožní prístup v jednom aplikačnom serveri, ale na druhom už može byť prístup pre ňu zamietnutý.



6. Architekúra kontajnerov[3]

Na obrázku č. 8 možme vidieť rozdelenie kontajnerov podľa toho, aký typ komponent spravujú.Kontajnery, ktoré riadia beh bussines a web komponentov,čiže bežia na aplikačnom serveri sú:

- **Enterprise JavaBeans (EJB) kontajner**

  Spravuje vykonávanie a životný cyklus enterprise bean-ov, ktoré využívajú(obsahujú) jednotlivé J2EE aplikácie.

- **Web kontajner**

  Spravuje vykonávanie JSP stránok a servletov, ktoré využívajú(obsahujú) jednotlivé J2EE aplikácie.

3 Databáza

V tejto kapitole sa budem venovať databázam všeobecne, kde prejdem naprieč jednotlivými databázovými modelmi a upresním relačný databázový model, ktorý používajú databázy Firebird a Hypersonic.Následne na to priblížim architekúru databázových systémov Firebird a Hypersonic a priblížim možnosti použítia v aplikáciách s príkladmi, ktoré budú rozšírené o konkrétnu konfiguráciu v časti, ktorá sa priamo venuje konfigurácii JBoss AS oproti databáze Firebird.

3.1 Databázy všeobecne

Databázu môžeme definovať ako kolekciu štruktúrovaných dát alebo informácií uložených v počítačovom systéme, takým spôsobom, že počítačový program, alebo človek môže použiť dopytovací jazyk (angl.: query language) na získavanie týchto informácií.

Typicky, v každej databáze existuje popis štruktury dát a typu dát, ktoré sú v databáze: tento popis sa nazýva logická schéma. Táto schéma popisuje objekty, ktoré sú v databáze a vzťahy medzi nimi. Existuje viacero rôznych spôsobov tvorby schém t.j. modelovania databázovej štruktúry: tieto sa nazývajú databázovými modelmi (alebo modelmi dát). V súčasnosti je najviac používaným relačný model. Tento model reprezentuje vzťahy použitím tých istých hodnôt vo viacerých tabuľkách. Iné modely, napríklad hierarchický model alebo sieťový model používajú explicitnejšiu reprezentáciu vzťahov.[6]

Najvyužívanejším modelom databázi je Relačný model, na ktorom je postavená aj databáza Hypersonic a Firebird.Preto sa v ďalšom texte budem venovať tomuto modelu.

Relačná databáza:

Základom relačných databází sú databázové tabuľky. Ich stĺpce sa nazývajú atribúty, alebo polia, riadky tabuľky sú záznamy.Atribúty majú určený svoj konkrétny dátový typ - doménu. Konkrétne tabuľka potom realizuje podmnožinu karteziánskeho súčinu možných dát všetkých stĺpcov - reláciu. Relačná databáza je založená na tabuľkách, ktorých riadky zvyčajne chápeme ako záznamy a eventuálne niektoré stĺpce v nich (tzv. cudzie kľúče) chápeme tak, že uchovávajú informácie o reláciách medzi jednotlivými záznamami.[6]

3.2 Databáza Firebird

Firebird je databáza, ktorá splňuje štandardy SQL. Vychádza z implementácie InterBase databázi ako OpenSource riešenie. Pracuje na architektúre klient – server,kde klient pomocou API s klientskej knižnice, ktorú musí daná aplikácia používať, volá funkcie databázového serveru, pomocou ktorých server priamo pristupuje k databáze a vykonáva dané operácie.
Existuje ešte verzia embedded, ktorá je ekvivalentná k verzii in-memory,alebo mem databázi Hypersonic.Bližšie k inštalácii tejto databázi sa zmienim v kapitole ktorá sa venuje priamo konfigurácii aplikačného servera a databázi.

Klient InterBase / Firebird

Klient Interbase je vlastne akákoľvek aplikácia, ktorá sprostredkováva koncovým užívateľom prístup k funkciám serveru. Tieto aplikácie sú typicky vytvorené v niektorom z jazykov vyššej úrovne napr.: C/C++ , Delphi , Java , PHP a ďalšie.
V architektúre klient – server nemajú klientské aplikácie príamy fyzický prístup databázam, ale komunikujú so serverom pomocou požiadavkou a odpovedí prostredníctvom klientskej knižnice.

Táto knižnica poskytuje aplikácii programové rozrhanie nazývané InterBase API a musí byť nainštalované na všetkých klientských staniciach, ktoré majú prístup k serveru.
Klientské aplikácie sú umiestnené zvyčajne na počítači vzdialenom od servera a k serveru sa pripájajú pomocou komunikačného protokolu (zvyčajne TCP/IP).Existujú však aj riešenia, kde sú klientská aplikácia, klientská knižnica a server fyzicky umiestnené na jednej stanici.

Server Interbase / Firebird :
Server InterBase je samostatný program, ktorý poskytuje služby klientským aplikáciám s ktorými komunikuje prostredníctvom siete.Klienti sa pripojujú k databázam pod správou servera a preto **musia** byť databázi fyzicky umiestnené na rovnakom počítači na ktorom je spustený server(pretože server, ako program musí mať k databázi prístup na úrovni súborového systému).
Server je schopný komunikovať s mnohými uzlami siete súčasne a spracovávať požiadavky od viacerých klientov pracujúcich na jednom sieťovom uzly. Ak je sieťová komunikácia realizovaná pomocou protokolu TCP/IP, tak je veľkosť siete v podstate neobmedzená a klienti možu pristupovať k serveru aj prostredníctvom Internetu.

Úlohou serveru je :
- Riadiť pristup k údajom v databázi prostredníctvom transakcií
- Zaistiť konzistentný pohľad na dáta jednotlivých transakcií
- Spracovávať požiadavky na výber ,vkladanie ,mazanie ,alebo modifikáciu dat
- Udržovať štatistiky práce s každou databázou
- Spravovať informácie o štruktúre databázi (metadata) pre potreby transakcií a údržby databázi

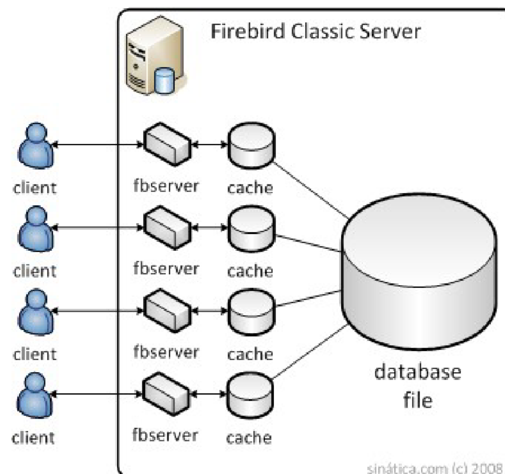
Na žiadosť klientov vykonáva server operácie ako :
- Vytvorenie novej databázi
- Vytvorenie, alebo zmena objektov v databázi
- Kontrola a kompilácia procedúr a triggerov
- Vyhľadávanie dát podľa zadaných kritérií
- Odosielanie požadovaných dát naspať klientom
- Zmena dát
- Spúšťanie procedúr
- Posielanie správ klientom o udalostiach v databázi

Z historického dovodu existujú 3 rozne architektúry(verzie databázového serveru) databázi InterBase / Firebird a sú to **classic server, super server a superClassic.[2]**

3.2.1 Classic Server

Povodná verzia InterBase bola implementovaná v době ,keď operačné systémy neumožňovali používať viac paralelných vlákien v rámci jedného procesu ,ani efektívne zdieľanie pamati medzi bežiacimi programami.Preto povoodná architektúra InterBase – classic – používa pre obsluhu užívateľských pripojení samostatné procesy, ktoré pre vzájomnú synchronizáciu používajú priamo databázové súbory, externých manažérov zámkov a základné prostriedky medziprocesorovej komunikácie ponúkané operačným systémom.
Pre každého pripojeného užívateľa je preto spustená nová kópia procesu serveru, ktorá sa stará výhradne o spracovávanie požiadavkou od tohoto užívateľa.
Nevýhodou tejto implementácie je náročnosť systémových zdrojov a nízka úroveň zdieľania informácií medzi jednotlivými procesmi. Súčasná implementácia tejto architektúry taktiež neumožňuje zdieľať vyrovnávaciu pamať databázi, čo sposoboje vačsie požiadavky na diskové operácie.
Výhodou naopak je jednoduchosť ,výrazne nižšie nároky na zdroje systému a vyššia rýchlosť spracovania požiadavkov pri práci jedného užívateľa.

Server architektúry classic je realizovaný ako zdieľaná knižnica .Pokiaľ klientská aplikácia požaduje lokálny sposob pripojenia k serveru umiestnenom na rovnakom počítači ,zavedie klientská knižnica serverovú časť InterBase priamo do pracovnej oblasti aplikácie. Pretože server v takomto prípade beží ako súčasť klientskej aplikácie ,je možné dáta medzi serverom a aplikáciou prenášať priamo a spracovanie dát je preto veľmi rýchle.
Server musí byť registrovaný ako služba na porte 3050.
Na obrázku 7 vidíme architekúru serveru classic. Proces serveru je označený jako fbserver s vlastnou cache pamaťou.Ako vidíme, pri novom požiadavku od klienta na databázový server, server vytvorí kópiu procesu, ktorý obsluhuje jedného klienta.

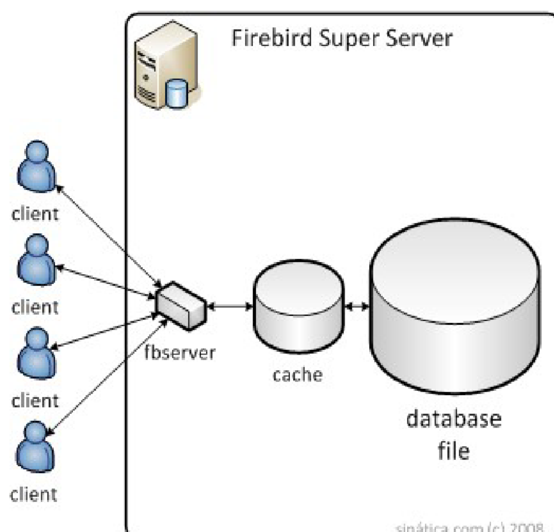


7. Architektúra Classic Server

3.2.2 Super Server

Táto architektúra je schopná v rámci jediného procesu obslúžiť viac klientov súčasne pomocou paralelných vlákien spracovania.Umožňuje nielen zdieľať metadata ,ale aj vyrovnávaciu pamať medzi jednotlivými užívateľmi a výrazne znižuje nároky serveru na každého pripojeného užívateľa.

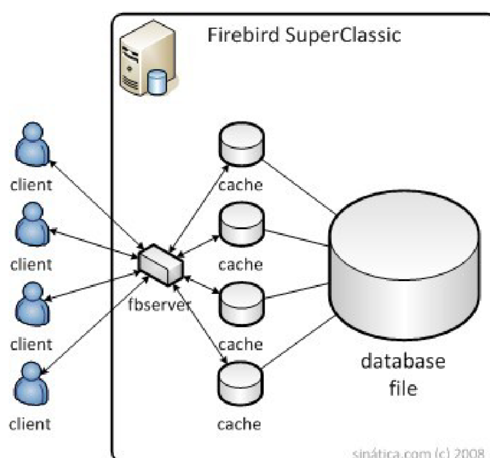
Server architektúry Super Server už nie je realizovaný ako zdieľaná knižnica ,ale aj ako samostatný program. Pre komunikáciu po sieti server sám prijíma všetky prichádzajúce požiadavky na porte 3050. To znamená ,že táto architektúra je použiteľná aj na platformách Windows ,kedže nevytvára kópie procesov pre obsluhu požiadavkov od klientov.
Na obrázku 8 vidíme architektúru super server.Server počúva na porte 3050 a pri novom požiadavku obslúži klienta v rámci jedného procesu. Výhodou je, žč takto umožňuje databázový server obslúžiť viacej klientov v rámci jedného procesu, čím nemá vysoké nároky na systémové zdroje při veľkom počte pripojení narozdiel od verzie classic server.



8. Architektúra Super Server

3.2.3 SuperClassic

SuperClassic verzia databázového serveru Firebird je najnovšou implementáciou, ktorá bola zverejnená len nedávno (apríl 2010) a kombinuje 2 predchádzajúce verzie super server a classic server.
Na obrázku 9 možeme vidieť jej architektúru.Je založená na princípe classic serveru, lenže namiesto nového procesu pri prijímaní požiadavku na spojenie vytvára nové vlákno čím výrazne šetrí systémové prostriedky a uľahčuje tým správu zdieľanej pamati, kedže každé vlákno má pridelenú vlastnú vyrovnávaciu pamať. [9]



9. Architektúra SuperClassic

9 Obrázky sú prevzaté z dokumentu voľne dostupného na adrese www.intitec.com/varios/Firebird-SuperServer-ClassicServer-SuperClassic.pdf

3.2.4 Použitie databázy Firebird

Inštalácia

V nasledujúcej tabuľke sú uvedené hlavné komponenty, ktoré obsahuje inštalačný archív[10] a ktoré sú potrebné pre správny beh Firebird databázového servera.

Samotný postup pri inštalácii je uvedený v kapitole, ktorá sa venuje konfigurácii JBoss AS.

Komponenta	**Meno súboru**	**Umiestnenie**
Dokumentácia	Rozličné názvy	<instal_priecinok>/doc
Firebird server	fbserver (SS) alebo fb_inet_server (CS)	<instal_priecinok>/bin
Nástroje na prácu s databázovým serverom (príkazový riadok)	isql-fb, gbak, gsec, gfix, gstat, atď.	<instal_priecinok>/bin
Knižnice na strane klienta	libfbclient.so.2.m.n (sieťový client) libfbembed.so.2.*m*.n (lokálny klient, s embedded verziou databázového serveru)	/usr/lib

Komponenta *Firebird server* je program, ktorým sa spúšťa databázový server – existujú 2 verzie tohoto programu líšiace sa podľa verzie databázového systému – fbserver – SuperServer verzia, fb_inet_server – Classic Server verzia databázového serveru.

Nástroje pre prácu s databázovým serverom slúžia na spravovanie databázového serveru z príkazového riadku – umožňujú vytvárať databázy, nových užívateľov, prideľovať práva atď.

- isql-fb: nástroj pre správu databáz z príkazového riadku.
- gbak: nástroj na zálohu databáz.
- gsec: nástroj na správu bezpečnosti. dokáže spravovať užívateľov, vytvárať nové účty a prideľovať práva
- gstat:nástroj na tvorbu štatistík a reportov.
- gfix: nástroj na obnovovanie databáz zo zálohy, vykonáva opravy databázového serveru.

Konkrétne použitie je uvedené v kapitole, ktorá sa venuje konfigurácii JBoss AS.

Knižnice na strane klienta slúžia na to aby aplikácie u klienta mohli pristupovať k API Firebird serveru. Taktiež je tu knižnica potrebná pre prevádzku embedded serveru databáze Firebird – viac v kapitole, ktorá sa venuje konfigurácii JBoss AS.

Samozrejme na to aby mohli J2EE aplikácie komunikovať s Firebird databázovým serverom je potrebný JDBC ovládač. Pre Firebird bol preto vyvinutý JAYBIRD.

10 http://www.firebirdsql.org/index.php?op=files&id=engine_213

3.2.5 JDBC ovládače databáze Firebird – Jaybird

JDBC ovládač pre Firebird databázu je JayBird.Inštalačný súbor je možné nájsť na oficiálnych stránkach[11] firmy IBPhoenix, ktorá sa vývojom tohoto ovládača zaoberá.Na to aby sme mohli ovládač v aplikácii využívať, musíme ho zaregistrovať pomocou metódy Class.forName(), tak ako je to uvedené na príklade:

Príklad načítania ovládača v J2EE aplikácii.

```
Class.forName("org.firebirdsql.jdbc.FBDriver");
```

Po zaregistrovaní ovládača možeme vytvoriť spojenie s databázou pomocou triedy DriverManager.

```
import java.sql.*;
 ...
 Connection connection = DriverManager.getConnection(
     "jdbc:firebirdsql:localhost/3050:/home/employee.gdb",
     "SYSDBA",
     "masterkey");
```

Na horeuvedenom príklade sa prípajame k databáze *employee.gdb* umiestnenej na miestnom počítači v priečinku */home* ,ako užívateľ *SYSDBA* s heslom *masterkey*.Samozrejme tieto hodnoty možu byť iné.

JayBird implementuje 2 typy JDBC ovládača – typ 2 a typ 4[12]. Na to aby sme daný typ ovládača použili je potrebné zmeniť reťazec pomocou ktorého sa pripojujeme.

Ovládač typu 4

```
jdbc:firebirdsql:host[/port]:/cesta/k/db.fdb
jdbc:firebirdsql://host[:port]/cesta/k/db.fdb
```

Pripojí sa k databáze pomocou ovládača typu 4. Je to najčastejšie používaná varianta v architektúre klient – server.

Ovládač typu 2 – vzdialené pripojenie

```
jdbc:firebirdsql:native:host[/port]:/cesta/k/db.fdb
jdbc:firebirdsql:native://host[:port]/cesta/k/db.fdb
```

Pripojí sa k databáze pomocou ovládača typu 2 pomocou klientskej knižnice – libfbclient.so.

Ovládač typu 2 – lokálne pripojenie

```
jdbc:firebirdsql:local:/cest/k/db.fdb
```

Použije ten istý typ pripojenia ako v predchádzajúcom prípade s tým rozdielom, že nevyužije komunikáciu pomocou socketov, ale pomocou IPC[13].

11 JDBC ovládač JayBird http://www.ibphoenix.com/main.nfs?page=ibp_download_jaybird

12 Viac v kapitole venujúcej sa databázovej vrstve architektúry J2EE

13 http://en.wikipedia.org/wiki/Inter-process_communication

Ovládač typu 2 – embedded pripojenie

```
jdbc:firebirdsql:embedded:/cesta/k/db.fdb
```

Použije ten istý typ pripojenia ako pri vzdialenom pripojení, ale využije pri volaniach API funkcií serveru embedded knižnicu libfbembed.so na Linux systéme.

3.3 Databáza Hypersonic

HyperSQL (Hyperosonic) databáza je SQL relačná databáza napísaná kompletne v Jave. Obsahuje JDBC ovládač a podporuje takmer všetky verzie SQL jazyka. Ponúka malý, rýchly databázový systém, ktorý ponúka in-memory[14] a klasické diskové verzie databáz.Taktiež obsahuje nástroje, ako konzolový prístup k databázi(command-line SQL tool), alebo grafický nástroj na manipuláciu dát v databázi.
Kedže je databáza Hypersonic primárne dodávaná v JBoss archíve s potrebnými komponentami na jej správu, nebudem sa v tejto časti zaoberať samostatnou inštaláciou.
Pokiaľ by, ale užívateľ chcel nainštalovať Hypersonic, ako samostatný databázový server, bez JBoss AS, tak odporúčam návod z oficiálnych stránok Hypersonic[15].

3.3.1 Architektúra databázového systému Hypersonic

Každá Hypersonic databáza sa nazýva katalóg.Existujú 3 typy katalógov, podľa toho, akým sposobom sú na nich uložené dáta.

Typy katalógov:

- *mem:* databáza je kompletne uložená v operačnej pamati – k strate dát dojde ihneď po ukončení JVM.
- *file:* databáza je uložená v súboroch operačného systému.
- *res:* uložená ako jar,alebo zip archív, vždy iba s prístupom pre čítanie.

Databázy(katalógy) typu ***mem*** možu byť použité na testovanie, tieto databázy neexistujú na pevnom disku.

Databázy typu ***file*** pozostávajú z 2 až 5 súborov,všetky majú rovnaký názov z rozdielnou príponou. Napríklad databáza test pozostáva z nasledujúcich súborov:

- test.properties - obsahuje nastavenie týkajúce sa databázi.
- test.script - obsahuje definíciu tabuliek a iných databázových objektov.
- test.log - obsahuje zmeny, ktoré boli vykonané nad databázou.
- test.data - obsahuje dáta načítané do vyrovnávacej pamati – pre rýchlu odozvu, pokiaľ sa vykonávajú rovnaké SQL príkazy nad databázou.
- test.backup - uchováva v komprimovanej podobe posledný konzistentný stav súboru *test.data.*

14 http://www.linuxjournal.com/article/6133

15 http://www.hsqldb.org/doc/2.0/guide/running-chapt.html

Pokiaľ je otvorené spojenie nad databázou typu file, sú zapisované zmeny do súboru *test.log*. Pri normálnom vypnutí sa súbor zmaže.Pokiaľ ale dojde k chybe, alebo neočakávanému stavu počas vypínania databázy(pád JVM), tak sa pri ďalšom spustení databázi tento súbor načíta, aby sa mohol stav databázy vrátiť do posledného konzistentného stavu.

Databáza typu ***res***: databáza pozostáva z malých databázových súborov, ktoré sú vytvorené iba pre čítanie.Sú uložené v jar, alebo zip archívoch a možu byť prenášané spolu s Java aplikáciou.

3.3.2 Použitie Hypersonic databázy

Jadro systému je obsiahnuté v HSQLDB jar[16] archíve.[17]

Komponenty HSQLDB jar archívu:

- HyperSQL databázový systém (HSQLDB)
- HyperSQL JDBC ovládač
- Databázový manažér (angl.:database manager) (GUI databázový nástroj pre prístup k databázovým objektom, postavený na SWING a AWT)
- SQL nástroj (nástroj pre prístup k databázovým objektom z príkazového riadku)

HyperSQL databázový systém a HyperSQL JDBC ovládač poskytujú základnú funkcionalitu databáze Hypersonic.

Nástroje pre prístup k databázi

Tieto nástroje(databázový manažér, SQL nástroj) sú určené pre interaktívny prístup užívateľov k databázi vrátane vytvárania databáz, vkladania dát a celkovo vykonávania operácií nad objektami v databázi.Všetky tieto nástroje bežia ako normálne Java programy.

V nasledujúcom príklade je spustená GUI verzia databázového manažéra(hsqld.jar súbor je uložený v adresári ../lib)

Spustenie databázového manažéra:

```
java -cp ../lib/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing
```

Hlavné triedy, ktoré implementujú nástroje pre prístup k databázi sú:

- org.hsqldb.util.DatabaseManager
- org.hsqldb.util.DatabaseManagerSwing

16 Java archive – archív vytvorený príkazom jar, ktorý zoskupuje súbory v ňom uložené, viac informácií - http://en.wikipedia.org/wiki/JAR_file_format

17 Tento archív je dodávaný spolu s JBoss AS a po inštalácii JBoss AS sa nastaví na štandardnú konfiguráciu.

3.3.3 JDBC ovládač databáze Hypersonic

Prístup k databázam je riešený cez JDBC API a to konkrétne pomocou triedy DriverManager[18].Pokiaľ je databáza spúšťaná na tom istom stroji, ako aplikačný server stačí volaním metódy getConnection() z triedy DriverManager získať objekt(java.sql.Connection), ktorým získame prístup k databázi.Reťazec predávaný metóde getConnection sa líši podľa typu databázi na ktorú sa pripájame. Uvediem preto príklady u jednotlivých typov databáz. Predpokladajme, že sa prihlasujeme pod užívateľom s prihlasovacím menom SA a bez hesla(štandardné nastavenie hypersonic databázového serveru)

Databáza typu file(databáza testdb):

Connection c = DriverManager.getConnection("jdbc:hsqldb:file:testdb", "SA", "");

Databáza typu mem (databáza mymemdb):

Connection c = DriverManager.getConnection("jdbc:hsqldb:mem:mymemdb", "SA", "");

Databáza typu file (databáza org.my.path.resdb):

Connection c = DriverManager.getConnection("jdbc:hsqldb:res:org.my.path.resdb", "SA", "");

Pokiaľ je databáza fyzicky umiestnená na inom stroji ako aplikačný server, je potrebné naviazať spojenie s databázovým serverom iným sposobom, o ktorom sa zmienim v nasledujúcej časti.

Prístup k databázam zo siete

Na to aby mohli aplikácie pristupovať k hypersonic databázam zo vzdialeného miesta – čiže aplikačný server je fyzicky umiestnený na inom mieste ako databázový server, musím databázu Hypersonic spustiť v tzv. **Server móde**.V takomto prípade databázový systém beží v JVM a načúva na pridelenom porte programom, ktoré sa snažia vytvoriť spojenie s databázou buď pomocou JDBC API, alebo iným sposobom. Samozrejme programy možu byť fyzicky umiestnené na inom ,ale aj na tom istom počítači ako databázový server.Pri akceptovaní spojenia vytvára spojenia tým istým sposobom ako je uvedené v predchádzajúcej časti.

Existujú tri variácej Server módu

- Hypersonic HSQL Server - preferovaný sposob použítia Hypersonic databázového serveru
- Hypersonic HTTP Server – používa sa, keď je prístup k databáze obmedzený na protokol http
- Hypersonic HTTP Servlet – používa sa v tom istom prípade jako Hypersonic http Server, s tým rozdielom, že je využívaný v prípade, že k databáze má prístup iný systém ako Hypersonic databázový server – napríklad iný aplikačný server.

Na nasledujúcom príklade je ukázané, že klient najprv nahrá ovládač JDBC pomocou metódy Class.forName() a následne sa snaží vytvoriť spojenie s databázovým serverom na štandardnom porte(9001), pričom použíje protokol hsql.

Príklad vytvorenia spojenia:

```
try {
        Class.forName("org.hsqldb.jdbc.JDBCDriver" );
    } catch (Exception e) {
        System.err.println("Chyba: nepodarilo sa nahrat JDBC ovladac.");
        e.printStackTrace();
        return;
    }

    Connection c = DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/xdb", "SA", "");
```

18 Viacej v kapitole venujúcej sa architektúre J2EE

4 JBoss Aplikačný server

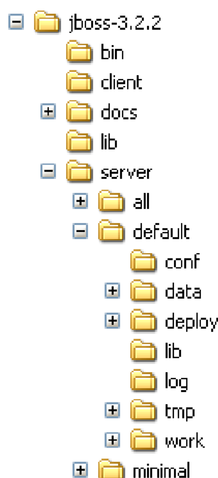
V tejto kapitule upresním, čo sa rozumie pod pojmom aplikačný server a následne popíšem architektúru aplikačného serveru JBoss, z čoho sa skladá a akým sposobom pracuje. Konkrétne nastavenie jednotlivých služieb,dátových zdrojov, či príklad tvorby aplikácie bude uvedený v časti, ktorá sa venuje konfigurácii JBoss AS.

4.1 Aplikačný server

Aplikačný server tvorí vrstvu medzi operačným systémom a aplikáciami. Podobne, ako operačný systém poskytuje základné funkcie programom (napríklad prístup k súborovému systému, alebo k správe procesov), poskytuje aplikačný server často používané funkcie enterprise aplikáciám. Vytvára ďalšiu vrstvu abstrakcie, aby bolo písanie aplikácií jednoduchšie. Príkladom takýchto funkcií môžu byť podpora transakčného spracovania požiadaviek, persistencia objektov do databázy, výmena správ medzi aplikáciami a ďalšie.
Viac o funkcionalite aplikačných serverov je uvedené v kapitole Architektúra J2EE, v sekcii *Aplikačné Servery.*

4.2 Adresárová štruktúra JBoss

JBoss aplikačný server (ďalej len JBoss AS), poskytuje relatívne veľké možnosti nastavenia služieb. Na to aby sme pochopili architektúre JBoss AS, je potrebné poznať jednotlivé časti z ktorých sa skladá.Na obrázku 10 je možné vidieť základnú adresárovú štruktúru JBoss AS.



10. Adresárová štruktúra JBoss AS

- **bin** – obsahuje rozne skripty a súbory s nimi súvisiace. Príkladom može byť run.sh na spustenie serveru
- **client** - Tu sú uložené konfiguračné a jar archívy,ktoré možu byť použité klientskou Java aplikáciou.

- **docs** - Obsahuje XML DTD súbory použité v JBoss AS ako referencie ,alebo ako zdroj dokumentácie JBoss konfigurácie.Obsahuje taktiež príklady JCA konfiguračných súborov pre nastavenie dátových zdrojov (datasources) pre rozne databázy.
- **lib** - Jar súbory ,ktoré sú potrebné pre beh JBoss mikrojádra. Užívateľ by sem nemal pridávať vlastné jar súbory.
- **server** – Každý podadresár tohoto adresáru je vlastne rozdielna serverová konfigurácia. Na načítanie konkrétnej konfigurácie je potrebné při spustení JBoss AS pridať parameter "-c <config name>"k spúšťaciemu skriptu (run.sh).

4.3 Konfigurácia servru

V adresári "server" sú tri povodné konfigurácie : *all , default a minimal* ,kde každá z nich obsahuje rozdielnu množinu služieb.

- **Konfigurácia "default"** sa dá preložiť ako štandardná konfigurácia. Čiže sa použije ,pokiaľ ste k spúšťajúcemu skriptu nepridali žiadne parametre.Obsahuje všetko ,čo potrebujete pre beh samostatného(v odbornej literatúre sa možte stretnúť s pojmom stand-alone) aplikačného servera.
- **minimal**
  Úplné minimum ,ktoré potrebujete pre štart JBoss AS. Obsahuje logovacie služby, JNDI server a URL skener na vyhľadávanie nových verzií J2EE klientských aplikácií.
  Táto konfigurácia sa používa ak užívateľ chce používať JBoss webovú konzolu pre štart vlastných služieb bez využitia J2EE aplikácií.Neobsahuje kontajner pre beh bussines, alebo webových komponent (EJB kontajner,WEB kontajner)
- **all**
  Spustí všetky dostupné služby . Toto taktiež zahŕňa RMI/IIOP[19], klusterovacie služby web služby ,ktoré "default" konfigurácia neobsahuje.

Užívateľ može taktiež pridať vlastnú konfiguráciu. Najefektívnejší sposob je skopírovať povodnú konfiguráciu , trebars konfiguráciu "default" a uložiť ju do adresáru server.
Vlastnú konfiguráciu potom užívateľ može načítať pri spustení JBoss AS príkazom "run -c myconfig" v prípade, že adresár s novou konfiguráciou je nazvaný *myconfig*.
Každá konfigurácia ,alebo konfiguračný adresár obsahuje ďalšie podadresáry ,ktoré sú :

- **conf** - obsahuje súbor jboss-service.xml, ktorý špecifikuje služby jadra.Adresár obsahuje taktiež ostatné konfiguračné súbory potrebné pre beh serveru.
- **data** - toto je miesto, kde Hypersonic(alebo iná databáza) ukladá svoje dáta. Je taktiež používaný JBossMQ (implementácia JMS) na ukladanie správ na disk.
- **deploy** - Užívateľ ukladá svoje aplikácie na server tým ,že ich skopíruje sem. Je tiež používaný pre hot-deployable služby (tie, ktoré môžu byť doplnené alebo odstránené za behu aplikačného serveru).
  Adresár je neustále kontrolovaný a jeho obsah sa prepisuje automaticky pri zmene programových komponentov.
- **lib** - Sem sa ukladajú jar archívy, ktoré sú potrebné pre konkrétnu konfiguráciu serveru.Užívateľ sem taktiež može pridať vlastné súbory(knižnice),například ovládače JDBC.
- **log** - Sem sa ukladajú informácie z aplikačného logu. JBoss AS používa Jakarta log4j knižnice pre logovanie ,ktoré može využiť taktiež užívateľ pri tvorbe vlastných aplikácií.
- **tmp** - Sem je umožnené ukladať dočasné súbory potrebné pre beh konkrétnej J2EE komponenty.

---

19 Implementácia volania vzdialených procedúr (RPC) - http://java.sun.com/products/rmi-iiop/

- **work -** Používa ho Tomcat ,alebo iný webový kontajner pre kompilovanie JSP.

Základ JBossu je postavený na Java Management Extension[20] (JMX) mikrojadre.JMX je framework, ktorý dovoľuje buď užívateľovi, alebo J2EE kompenentom priamo komunikovať so službami serveru za jeho behu. Užívateľ(alebo komponenta) teda može spúšťať, alebo zastavovať služby, meniť parametre služieb a to všetko za behu serveru. Služby, ktoré JMX rozhrania sú enterprise beans, ktoré nazývame managed beans, alebo skrátene MBeans.

20 http://java.sun.com/docs/books/tutorial/jmx/overview/index.html

5 Konfigurácia

V tejto kapitole uvediem postup pre konfiguráciu JBoss AS, tak aby pracoval s databázou Firebird, ku ktorému som dospel na základe testov a analýzy zdrojových súborov. Najprv popíšem postup inštalácie JBoss AS a databázového serveru Firebird a následne uvediem implementáciu jednoduchej testovacej aplikácie na demonštráciu, že správne nakonfigurovaná enterprise aplikácia je schopná pracovať oproti databáze Firebird na aplikačnom serveri JBoss.

V hlavnej časti potom uvediem postup akým som nastavoval jednotlivé dátové zdroje a java zdrojové súbory, tak aby pracovali s databázou Firebird. V tejto časti taktiež zhodnotím výsledky testov a budem sa snažiť analyzovať jednotlivé chyby a navrhnúť možnosti ich riešenia.

5.1 Príprava testovacieho prostredia

V tejto časti uvediem prípravu testovacieho prostredia. Značná časť bude venovaná inštalácii databázového serveru Firebird.

Zdrojové kódy JBoss AS:
Na to aby sme mohli správne nakonfigurovať JBoss AS potrebujeme stiahnuť určitú verziu zdrojových kódov.Ako referenčné prostredie som vybral verziu 5.1.0 aplikačného serveru JBoss.
Zdrojové kódy užívateľ stiahne z SVN repozitára príkazom:

```
svn co http://anonsvn.jboss.org/repos/jbossas/tags/JBoss_5_1_0_GA/
```

Pri inštalácii je potrebné mať správne naištalovanú Java platformu a taktiež správne nastavené systémové premenné. Postup je uvedený na oficiálnych stránkach firmy Sun, alebo stránkach JBoss.[21]

Inštalácia databázového serveru Firebird:
Na to aby sme mohli plne otestovať funkčnosť JBoss AS a databázovy Firebird je potrebné nainštalovať 2 verzie databázového serveru Firebird – super server verziu a embedded verziu.
Inštaláciu verzie super-server vykonáme príkazom:

```
apt-get install firebird2.1-super
```

Tým sa stiahnu potrebné balíčky a nainštalujú sa na cieľovej stanici. Popritom sa v operačnom systéme vytvorí nový užívateľ *"firebird"*, ktorému sú nastavené oprávnenia spúšťať server a spustí sa server.
Po spustení servera je potrebné nastaviť heslo pre nového *„firebird"* užívateľa.
V Linux/Unix systémoch je to možné vykonať príkazom passwd <heslo>.

Následne ako firebird užívateľ je vhodné nainštalovať testovaciu databázu na demonštráciu funkčnosti databázového servera príkazom :

```
apt-get install firebird2.1-examples
```

Archív s databázou employee.fdb bude pri štandardných nastaveniach umiestnený pod adresárom */usr/share/doc/firebird2.1-examples/examples/empbuild/*.

21 http://docs.jboss.org/jbossas/docs/Installation_And_Getting_Started_Guide/5/html/Getting_Started.html

Preto je následne potrebné archív rozbaliť a pripojiť sa pomocou nástroja isql-fb[22] k databáze. To je možné vykonať sériou nasledujúcich príkazov.

```
# cd /usr/share/doc/firebird2.1-examples/examples/empbuild/
sudo gunzip employee.fdb.gz
sudo chown firebird.firebird employee.fdb
sudo mv employee.fdb /home_adresar/

isql-fb

SQL> connect "/home_adresar/employee.fdb " user 'SYSDBA' password 'SYSDBApassword';
```

Následne je možné z príkazového riadku zadať SQL príkaz na otestovanie správnej funkčnosti databázi. Napríklad príkaz *show tables;* pre výpis zoznamu tabuliek v databázi.

Ešte je dobré pripomenúť, že užívateľ SYSDBA má administrátorské práva, čiže može zasahovať do indexov a podobných databázových štruktúr. Preto je dobré vytvoriť užívateľa, ktorý bude mať obmedzené práva. Na to slúži nástroj GSEC. Na nasledujúcom príklade je zobrazené vytvorenie užívateľa *maniak* s heslom *maniak_password.*

```
gsec -user sysdba -pass masterkey -add maniak -pw maniak_password
```

Tým je inštalácia verzie super-server databázi firebird dokončená.

Embedded server na platforme Linux[23]:

Architektúra a funkcionalita embedded servera na platforme linux je rovnaká ako na platforme windows, líši sa až sposobom inštalácie.

Linux Classic server obsahuje knižnicu *libfbembed.so*, ktorá je používaná pre lokálne pripojenia.Táto knižnica obsahuje celé jadro Firebird serveru, čo vysvetľuje prečo je o toľko vačšia ako klientská knižnica Super server verzie databázového serveru Firebird. Lokálne pripojenia pomocou tejto knižnice sú realizované pomocou aplikačného procesu užívateľa a nie oddeleného procesu servera.Preto užívateľ tejto knižnice musí mať prístup na úrovni súborového systému ku databázovému súboru.Pri inštalácii je potrebné stiahnuť classic server verziu databázového serveru firebird.

Po rozbalení archívu je potrebné skopírovať doleuvedené súbory z adresáru *opt/firebird* do adresáru, kde bude embedded server uložený, napríklad */home/maniak/firebird_embedded/*, pričom je potrebné zachovať danú adresárovú štruktúru.

Ďalej je potrebné upraviť súbor firebird.conf a to vložením nastavenia

RootDirectory = /home/maniak/firebird_embedded/

Ďalej je potrebné nastaviť dve systémové premenné na zdrojový adresár embedded serveru:

export LD_LIBRARY_PATH=*/home/maniak/firebird_embedded/*
export FIREBIRD=*/home/maniak/firebird_embedded/*

Tým je nastavenie embedded serveru dokončené.

22 Zmienka v kapitole Použitie databázy Firebird

23 Zdroj : http://www.firebirdfaq.org/Firebird-Embedded-Linux-HOWTO.html

5.2 Tvorba testovacej aplikácie

Na to aby som mohol demonštrovať, že správne nakonfigurovaná enterprise aplikácia je schopná pracovať s databázou Firebird na aplikačnom serveri JBoss som vytvoril jednoduchú testovaciu aplikáciu.
Demonštráciu funkčnosti som vykonal z dôvodu, že tým overím základnú funkčnosť JBoss AS a Firebird databázy vrátane funkčného pripojenia pomocou JCA, ktoré bude neskôr potrebné využiť v nastaveniach zdrojových kódov.
Testovacia aplikácia by mala komunikovať s Firebird databázou, pričom by mala byť schopná dáta z databázi zobraziť, aby užívateľovi demonštrovala vzájomnú funkčnosť medzi JBoss AS a Firebird databázovým serverom. Taktiež sa tým preverí funkčnosť JDBC ovládača JayBird.

5.2.1 Návrh testovacej aplikácie – Spolužiaci

Aplikácia bude pracovať s jednou tabuľkou – spoluziaci, uloženú vo firebird databáze.Bude sa jednať o jednoduchú webovú aplikáciu, ktorá bude schopná zobraziť dáta z databázi. Bude postavená na MVC frameworku[24].Jedná sa o webovú aplikáciu, takže na otestovanie funkčnosti postačí internetový prehliadač.Taktiež bude implementovaná v rámci špecifikácie J2EE, takže bude postavená na viacvrstvovom modely.

5.2.2 Implementácia testovacej aplikácie

Nakoľko na potreby demonštrácie funkčnosti Firebird databázy je potrebné implementovať iba komponenty web vrstvy a databázovej vrstvy, vynechal som implementáciu komponent z bussines vrstvy.

Databázová vrstva

Na správnu implementáciu databázovej vrstvy postačí správne nastavenie dátového zdroja a následné nasadenie do JBoss AS.
Dátový zdroj firebird-ds.xml som nastavil následovne :

```
<connection-factories>
   <mbean code="org.firebirdsql.management.FBManager" name="jboss.jca:service=FirebirdManager">
    <attribute name="FileName">${jboss.server.data.dir}/Spoluziaci.fdb</attribute>
    <attribute name="UserName">maniak</attribute>
    <attribute name="Password">lukasmakac</attribute>
    <attribute name="CreateOnStart">false</attribute>
    <attribute name="DropOnStop">false</attribute> </mbean>
 <local-tx-datasource>
   <jndi-name>Spoluziaci</jndi-name>
   <connection-url>jdbc:firebirdsql:localhost/3050:${jboss.server.data.dir}/Spoluziaci.fdb</connection-url>
   <driver-class>org.firebirdsql.jdbc.FBDriver</driver-class>
   <user-name>maniak</user-name>
   <password>lukasmakac</password>
   <transaction-isolation>TRANSACTION_REPEATABLE_READ</transaction-isolation>
   <connection-property name="lc_ctype" type="java.lang.String">UNICODE_FSS</connection-property>
   <connection-property name="maxStatements">10</connection-property>
   <min-pool-size>0</min-pool-size>
   <max-pool-size>200</max-pool-size>
   <blocking-timeout-millis>5000</blocking-timeout-millis>
   <idle-timeout-minutes>15</idle-timeout-minutes>
   <check-valid-connection-sql>SELECT CAST(1 as INTEGER) FROM rdb$database</check-valid-connection-sql>
   <track-statements>false</track-statements>
   <prepared-statement-cache-size>0</prepared-statement-cache-size>
   <metadata>
       <type-mapping>Firebird</type-mapping>
   </metadata>
</local-tx-datasource>
</connection-factories>
```

24 http://zdrojak.root.cz/clanky/uvod-do-architektury-mvc/

- **Mbean** - tento element identifikuje Mbean komponentu, ktorá bude po nasadení súboru na server registrovaná aplikačným serverom. Tým je umožnené spravovať databázu uvedenú atribútom FileName. Príslušné atribúty určujú parametre spojenia s danou databázou. V tomto konkrétnom prípade JBoss AS registruje MBean komponentu org.firebirdsql.management.FBManager ako službu v doméne jboss.jca pod názvom FirebirdManager. MBean FBManager implementuje metódy na kontrolu, či daná firebird databáza existuje.Vďaka tejto službe je umožnené pracovať s Firebird databázou priamo z JMX konzoly JBoss AS(samozrejme je možné využívať iba metódy, ktoré táto komponenta implementuje)
- **local-tx datasource** – definuje nový dátový zdroj,ktorý umožňuje vykonávať lokálne transakcie, ktoré nepoužívajú dvojfázový potvrdzovací protokol.
- **jndi** - definuje meno na ktoré sa tento dátový zdroj zaregistruje pomocou služby JNDI – to znamená, že aplikácie, ktoré budú pristupovať k tomuto zdroju pomocou služby JNDI si nebudú musieť pamatať fyzické umiestnenie databázi.
- **connection-url** - definuje pripojenie na danú databázu, viac o connection-url je uvedené v kapitole Použitie databázy Firebird .
- **driver-class** – definuje triedu JDBC ovládača, ktorá sa použije pri získaní a obsluhe pripojenia. Ovládač musí byť nainštalovaný v lib adresáry serveru, aby ho mohol server pri spustení nahrať.
- **user-name** – meno, ktoré sa použíje pri vytváraní spojenia na databázu.
- **password** – heslo, ktoré sa použíje pri vytváraní spojenia na databázu.
- **type-mapping** – určuje aký typ ORM sa použije.

Zvyšné položky sú nepovinné a bližšie špecifikujú nastavenie dátového zdroja.
Po skopírovaní dátového zdroja do deploy adresára JBoss AS, aplikačný server databázový zdroj a MBean komponentu zaregistruje.

Webová vrstva

Základ tvorí komponenta **controller** – servlet ktorý vykonáva operácie nad **modelom** (objekty, ktoré sú zodpovedné za reprezentáciu dát z databázy) a výsledok potom predá komponente **view**, ktorá dáta zobrazí. Komponenta *view* obsahuje iba JSP súbory potrebné na zobrazenie výsledku v prehliadači. Popri implementácii aplikácie je ešte potrebné nastaviť metadata pre správne nasadenie aplikácie v JBoss AS.

Model

Komponenta model je zodpovedná za priamu interakciu z databázou, čiže načítanie dát a ich objektovú reprezentáciu. Tvoria ju štyri triedy :

- ServiceLocator je trieda, ktorá implementuje získanie databázového zdroja (firebird-ds.xml) podľa jndi referencie.
- StudentDAO je trieda, ktorá implementuje načítavanie dát z databázi.
- StudentDTO je objektová reprezentácia tabuľky Študent.

ServiceLocator:

```
public class ServiceLocator {
        public static DataSource getDataSource(String dsJNDIName) {
                DataSource ds = null;
                try {
                        Context ctx = new InitialContext(); //
                        ds = (DataSource) ctx.lookup(dsJNDIName);
                } catch (NamingException ne) {System.out.println("Chyba - JNDI :" + ne.getMessage());}
                return ds;
        }
}
```

StudentDAO:

```
public class StudentDAO {
        private ArrayList<StudentDTO> studenti;
        ...
        private static final String DataSourceString = "java:comp/env/jdbc/Spoluziaci";
        public StudentDAO() {
                this.studenti = new ArrayList<StudentDTO>();
                ...
        }
        public ArrayList<StudentDTO> getStudenti_from_Pool() {
                ResultSet rs = null;
                Statement stmt = null;
                Connection con = null;
                DataSource ds = null;
                try {
                                ds = ServiceLocator.getDataSource(DataSourceString);
                                con = ds.getConnection();
                                stmt = con.createStatement();
                                rs = stmt.executeQuery("SELECT * FROM Student");

                                while (rs.next()) {
                                        int id = rs.getInt("id");
                                        ...
                                        studenti.add(new StudentDTO(id, meno, priezvisko, mesto,
                                                        ulica, psc, email));
                                }
                                con.close();
                } catch (SQLException se) {
                        System.err.println("Chyba pri vytvarani noveho prikazu"
                                        + se.getMessage());
                } catch (Exception e) {
                        System.err.println("Chyba pri ziskavani ds " + e.getMessage());
                }
                return this.studenti;
        }
}
```

StudentDTO:

```
package model;

public class StudentDTO {
        private int id;
        private String meno,priezvisko,mesto,ulica,psc,email;

        public StudentDTO(int id,String meno,String priezvisko,String mesto,String ulica,String psc,String email){
                ...
        }

        //setters and getters
        ...
}
```

View

Komponenta view obsahuje JSP stránky na zobrazenie výsledku, ktorý mu predá komponenta model(resp. Controller).Obsahuje nasledujúce súbory:

- error.jsp
- index.jsp
- registeredUsers.jsp
- registration.jsp

Súbory zobrazujúce hlavné prvky aplikácie sú registeredUsers.jsp, ktorý slúži na zobrazenie dát a index.jsp, ktorý predstavuje vstupný bod aplikácie – je v ňom uvedený odkaz, ktorým sa zavolá controller a ten pomocou komponenty model načíta dáta a predá ich ďalej komponente view na zobrazenie.

Index.jsp:

```
...
        <a href="controller?action=zobrazStudakovDS">Zobraz registrovanych študentov z poolu</a>
...
```

registeredUsers.jsp:

```
    <c:forEach items='${zoznamRegStudent}' var='uzivatel'>
        <tr>
            <td><c:out value="${uzivatel.id}"></c:out></td>
            ...
        </tr>
    </c:forEach>
```

Controller

Komponentu controller tvorí iba jeden servlet - *ControllerServlet*, ktorý obsahuje v sebe odkaz na model a keď príde požiadavok na výber dát, načíta pomocou komponenty model dáta a prepošle dáta komponente view,ktorá sa postará o ich zobrazenie.

ControllerServlet:

```
    ...
    private static final String ACTION_KEY = "action";

    protected void doGet(HttpServletRequest poziadavok,
                     HttpServletResponse odpoved) throws ServletException, IOException {
        spracujPrikaz(poziadavok, odpoved);
    }

    protected void doPost(HttpServletRequest poziadavok,
                     HttpServletResponse odpoved) throws ServletException, IOException {
        spracujPrikaz(poziadavok, odpoved);
    }

    protected void spracujPrikaz(HttpServletRequest poziadavok,HttpServletResponse odpoved){

            String actionName = poziadavok.getParameter(ACTION_KEY);

            StudentDAO studenti = new StudentDAO();

            poziadavok.setAttribute("zoznamRegStudent", studenti.getStudenti_from_Pool());
            destinationPage = "/registeredUsers.jsp";

            RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(destinationPage);
            dispatcher.forward(poziadavok, odpoved);
    }
    ...
```

Proces celého zobrazenia funguje tak, že po kliknutí na odkaz *controller?action=zobrazStudakovDS* sa zavolá metóda triedy ControllerServlet doGet, ktorej sa predá parameter *action* s hodnotou *zobrazStudakovDS*.Následne načíta zoznam študentov z databázi a výsledok sa v poli *zoznamRegStudent* pošle stránke registeredUsers.jsp, ktorá výsledok zobrazí.
Za zmienku stojí ešte súbor, ktorý je zodpovedný za správne nastavenie komponent vo webovej aplikácii. Je to súbor web.xml a je súčasťou war archívu(web archive).
Obsahuje nasledujúce položky:

- welcome-file - určuje vstupný súbor, ktorý sa má načítať po zadaní adresy definovanej pomocou položky display-name

- servlet – obsahuje definíciu servletu – aby servlet kontajner vedel zaregistrovať daný servlet a spravovať ho.
- Servlet-mapping – obsahuje definíciu mapovania url na názov servletu
- resource-ref – obsahuje nastavenie dátového zdroja.

Web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>Spoluziaci_web</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <description>controller servlet for doing his job</description>
    <display-name>ControllerServlet</display-name>
    <servlet-name>ControllerServlet</servlet-name>
    <servlet-class>controller.ControllerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>/controller/*</url-pattern>
  </servlet-mapping>
  <resource-ref>
    <description>Spoluziaci DataSource</description>
    <res-ref-name>jdbc/Spoluziaci</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <mapped-name>java:/Spoluziaci</mapped-name>
  </resource-ref>
</web-app>
```

Všetky tieto časti web aplikácie sú zabalené do war archívu a skopírované do deploy priečinku JBoss. Po skopírovaní a zadaní adresy *localhost:8080/Spoluziaci_web* je možné zadanú aplikáciu otestovať čím sa dokáže, že Firebird databáza dokáže pracovať s JBoss AS.

5.3 Konfigurácia JBoss konfiguračných súborov a databázových zdrojov

Aby sme mohli správne nakonfigurovať JBoss AS oproti externej databáze, musíme najprv nakonfigurovať server ešte pred samotným zostavením.To znamená, že musíme nájsť databázové zdroje(*-ds.xml súbory) a java súbory, ktoré sú pred zostavením servera primárne nastavené tak, aby pracovali s databázou hypersonic a nastaviť ich tak aby pri zostavovaní pracovali z externou databázou – v našom prípade Firebird.

5.3.1 Databázové zdroje

Výber jednotlivých databázových zdrojov

Na výber potrebných databázových zdrojov som použil jednoduchý skript, ktorý vybere z aktuálneho adresára všetky súbory končiace na '-ds.xml'. Tým sa označujú databázové zdroje v JBoss AS.Slúžia

pri zostavovaní serveru a nastavovaní jednotlivých služieb, pretože jednotlivé služby a komponenty používajú nastavenia definované v týchto súboroch.

Konfigurácia databázových zdrojov:

Pri konfigurácii som vychádzal z oficiálnych stránok JBoss[25], kde je vysvetlený význam jednotlivých elementov u databázových zdrojov, ktoré sú definované v súboroch končiacich na *-ds.xml*.
Na samotné nastavenie databázových zdrojov som vytvoril Java SWING aplikáciu, ktorá na prácu so súbormi používa knižnicu JDOM.
Aplikáciu som konštruoval tak, aby na svoje potreby potrebovala 2 súbory vo formáte xml.
Prvý xml súbor je konfiguračný a obsahuje nastavenia hodnot, ktoré sa majú nastaviť u jednotlivých *-ds.xml* súborov.
Príklad konfiguračného súboru:

```
<root>
    <server_dir>/home/maniak/jboss_source_firebird/JBoss_5_1_0_GA</server_dir>
    <connection-url>
          jdbc:firebirdsql:localhost/3050:test.fdb
    </connection-url>
    <connection-url-embedded>
        jdbc:firebirdsql:embedded:
    </connection-url-embedded>
    <driver-class>org.firebirdsql.jdbc.FBDriver</driver-class>
    <type-mapping>Firebird</type-mapping>
    <username>maniak</username>
    <password>lukasmakac</password>
</root>
```

Nastavenia vychádzajú z nastavení, ktoré sú uvedené v kapitole Tvorba testovacej aplikácie, nakoľko už boli overené, že fungujú. Samozrejme, element *<connection-url>* nie je nastavený na pevnú hodnotu, pretože hodnoty re
Druhý xml súbor potrebný na beh programu obsahuje zoznam súborov, ktoré sa budú prekonfigurovávať(súbory s koncovkou -ds.xml).Na ich získanie som použil jednoduchý skript, ktorý vyhľadáva súbory s názvom končiacim na *-ds.xml.*
Príklad xml súboru so zoznamom súborov potrebných na prekonfigurovanie:

```
<root default_dir="/home/maniak/Plocha/Fit/jboss/jboss_source/jboss_firebird/jboss_firebird">
    <path>./testsuite/src/etc/deployment-test/testMultipleDataSources-ds.xml</path>
    <path>./testsuite/src/resources/bootstrapdependencies/jbas5349/bootstrapdependencies-hsqldb-ds.xml</path>
    <path>./testsuite/src/resources/jcaprops/uppergoodrar/upper-goodrar-ds.xml</path>
</root>
```

Po načítaní súboru je užívateľovi sprístupnená možnosť spustenia konfigurácie. Samotná konfigurácia potom prebieha tak, že aplikácia najprv rozparsuje konfiguračný súbor a názvy jednotlivých súborov a následne rozparsuje XML súbor na DOM objekt a upraví jednotlivé listy podľa načítanej konfigurácie.

U všetkých týchto položiek, vyhľadáva a mení položky podľa hodnot definovaných v konfiguračnom súbore.
V daných súboroch aplikácia mení elementy nasledujúcim sposobom:
Vyhľadáva v koreňovom elemente nasledujúce elementy(elementy, ktoré obsahuje -ds.xml súbor):

25 http://docs.jboss.org/jbossas/docs/Administration_And_Configuration_Guide/5/html_single/index.html#d0e505

- datasources
  - local-tx-datasource
  - mbean
  - xa-datasource
  - no-tx-datasource
- connection-factories
  - mbean
  - xa-connection-factory
  - no-tx-connection-factory

U daných elementov(okrem elementu *mbean*, ktorý má rozdielnu štruktúru) ďalej vyhľadáva a mení nasledujúce elementy:

- connection-url – nastavuje identifikáciu spojenia na databázu
- driver-class – nastavuje triedu ovládača, ktorá sa použije pri vytváraní spojenia na databázu
- user-name – nastaví užívateľské meno
- password – nastaví heslo pod ktorým sa užívateľ identifikuje v databázi
- metadata
  - type-mapping – nastavuje typ SQL príkazov
- depends – nastavuje MBean komponentu, ktorá sa použije pri správe databázového spojenia

Všetky tieto elementy identifikujú spojenie na databázu a sú preto primárne nastavené na databázu Hypersonic. Preto aplikácia dané elementy mení podľa hodnôt definovaných v konfiguračnom súbore. Výnimku však tvorí element *connection-url,* pretože u tohoto elementu sa v zdrojových súboroch vyskytovali rôzne hodnoty a podľa nich bolo potrebné určiť výsledný reťazec.
Pri konfigurácii reťazca som postupoval tým spôsobom, že som najpv rozparsoval reťazec na protokol a podrotokol, ktoré som zmenil na odpovedajúce hodnoty, pričom nastavenie serveru, portu a cesty k databázy zostalo rovnaké.

U elementu *mbean* vyhľadáva atribút *name,* ktorý určuje mapovanie služby na danú triedu a u tohoto atribútu vyhľadáva reťazec „jboss:service=Hypersonic“, čím sa definuje služba, ktorá slúži na spúšťanie a ukončovanie behu databázového serveru. Túto službu som premenoval na „jboss.jca:service=FirebirdManager“, čím sa zaregistruje služba FirebirdManager.
Taktiež vyhľadáva atribút *code* a nastavuje u neho príslušnú triedu, ktorá implementuje rozhranie na prácu s databázou. V tomto prípade som zmenil prímarne používanú triedu „org.jboss.jdbc.HypersonicDatabase“ na triedu „org.firebirdsql.management.FBManager“, ktorá slúži na prácu s databázou.

5.3.2 Java zdrojové súbory

Pri konfigurácii java zdrojových súborov pomocou skriptu je potrebné vyhľadať súbory v ktorých sa nachádzajú reťazce, ktoré priamo nastavujú zdroje na Hypersonic databázu.
Pri rozhodovaní sa, aké súbory budem konfigurovať som sa inšpiroval nastaveniami, ktoré boli uvedené u jednotlivých databázových zdrojov. Teda som vyhľadával súbory, ktoré v sebe obsahovali reťazce, ktoré potenciálne mohli obsahovať nastavenia na databázu Hypersonic.
Tieto súbory som potom manuálne kontroloval a upravoval tak, aby sa pracovali s rovnakými referenčnými JNDI názvamy, ale pritom sa používala databáza Firebird.
Tým som nastavenie pred prvým spustením testovacej sady dokončil.Prípadné chyby, ktoré sa vyskytli v konfigurácii a návrh na ich odstránenie je navrhnutý v časti Analýza chýb .

5.4 Testovanie

5.4.1 Nastavenie testovacej sady

Po nakonfigurovaní súborov je potrebné spustiť testovaciu sadu na overenie správnosti nastavenia aplikačného servera.

Následne ak chceme aby prebehli v poriadku clusterovacie testy, ktoré na testovanie potrebujú 2 bežiace inštancie servera, musíme nastaviť súbor local.properties a v ňom nastaviť položky node0 a node1. Obidve slúžia na to aby mohli byť vykonané clusterovacie testy, pretože na svoju činnosť potrebujú 2 inštancie servera.
Následne spustíme skript potrebný na vygenerovanie daných súborov – príkazom *sh build.sh* z adresáru testsuite.Po vygenerovaní sady testov tieto testy spustíme príkazom *sh build.sh tests.*
Po dokončení testov je potrebné vygenerovať výsledky v podobe reportov na zhodnotenie konfigurácie. Vygenerovanie reportov je možné vykonať pomocou príkazu *sh build.sh tests-reports*.

5.4.2 Štatistické výsledky

Po vykonaní prvej sady testov boli výsledky nasledovné:
Počet testov : 3665
Počet testov, ktoré spadli – došlo k chybe pri vykonávaní testu : 77
Počet testov, ktoré nevyhovovali : 29
Percentuálna úspešnosť: 97.11%

Po vykonaní druhej sady testov boli výsledky nasledovné:

Počet testov : 3713
Počet testov, ktoré spadli – došlo k chybe pri vykonávaní testu : 45
Počet testov, ktoré nevyhovovali : 16.
Percentuálna úspešnosť: 98,36 %

5.4.3 Analýza chýb

U neúspešne vykonaných testov bolo potrebné zistiť typ chýb, ktoré sa opakovali a nájsť súvislosť s nastaveniami java zdrojových súborov a databázových zdrojov. Preto som sa rozhodol najprv zanalyzovať a opraviť chyby, ktoré evidentne súviseli s týmito nastaveniami.
U testov sa vyskytovali tieto druhy chýb:

- Exception: junit.framework.AssertionFailedError
- Exception: org.jboss.deployment.DeploymentException
- Exception: javax.naming.NameNotFoundException
- Exception: java.lang.ClassNotFoundException
  - Message: org.firebirdsql.jdbc.FBDriver
- Exception: javax.ejb.CreateException
- Exception: java.lang.RuntimeException
- Exception: org.jboss.util.NestedSQLException

Na začiatku som sa zameral na analýzu testov u ktorých sa vyskytovala výnimka *ClassNotFoundException.* Preto som analyzoval kód prvého testu u ktorého táto výnimka nastala.
Zistil som, že chyba sa vyskytuje u volaní kódu: *Class.forName("org.firebirdsql.jdbc.FBDriver");*
Vyvolanie tejto výnimky znamenalo, že JDBC ovládač nie je nahratý v classpath.
Podľa JBoss komunitnej stránky[26], určenej k testovacej sade, som *classpath* nastavil v súbore: *tools/etc/modules.ent* , v ktorom sa nastavujú cesty k jednotlivým zdrojom.
Pridal som do súboru nasledovné nastavenie:

<pathelement path="${jboss.test.lib}/jaybird-full-2.1.6.jar"/>

Po zostavení testovacej sady bolo ešte potrebné skopírovať Jaybird JDBC ovládač do adresáru *testsuite/output/lib.*

Po opätovnom spustení testov bol výsledok odlišný,čo dokazujú štatistické údaje, preto bolo potrebné nájsť testy u ktorých došlo k rozdielnym výsledkom a určiť, či nastavenie vykonané po prvej sérii testov bolo správne.
Analýza chybového logu, ktorý je uložený na priloženom CD nosiči v adresári *prílohy/test_results/firebird_second* dokázala, že táto chyba sa v určitých testoch nevyskytovala, čím bolo potvrdené jej správne odstránenie.

Rozdiely medzi jednotlivými konfiguráciami sú uvedené v prílohách na priloženom CD spolu s celkovým výsledkom testov ako u jednej sady testov, tak u druhej.

26 http://community.jboss.org/wiki/TestsuiteChanges

6 Záver

Účelom práce bolo nakofigurovať JBoss aplikačný server tak, aby pracoval výhradne s databázou Firebird.Počas práce som sa zoznámil s viacerými technológiami, pričom som sa snažil postupovať s čo najväčším dôrazom na pochopenie súvislostí. Výsledkom práce je zdokumentovaný postup konfigurácie, zobrazenie štatistík, analýza a návrh riešenia chýb, zoznam konfigurovaných položiek a súbor v ktorom sú uvedené rozdiely medzi nenakonfigurovaným a nakonfigurovaným serverom JBoss a taktiež konfigurovací skript a java aplikácia, ktoré slúžia na automatickú konfiguráciu zdrojových súborov.
V projekte sa vyskytli určité nedostatky, kde analýzou chýb som dospel k riešeniu a testovaním bola overená správnosť riešenia.V rámci rozšírenia považujem za vhodné pokračovať v analýze zvyšných nedostatkov a návrhu na ich odstránenie.

Literatura

[1] Císař P.: InterBase/Firebird Tvorba,programování a správa databází.Computer Press, 2003.ISBN 80-7226-946-1
[2] Tom Marrs,Scott Davis: *JBoss at Work*. O'Reilly, 2006, ISBN 0-596-00734-5
[3] Sun Microsystems. *Java EE 5 tutorial* [online]. 2008 [cit. 2010-05-19]. Dostupné z WWW: <http://java.sun.com/javaee/5/docs/tutorial/doc/>.
[4] ŠEDA, Jan. *Interval.cz : Úvod do JDBC* [online]. 04. 03. 2003 [cit. 2010-05-19]. Dostupné z WWW: <http://interval.cz/clanky/uvod-do-jdbc>.
[5] BRANICKÝ, Marek. *Interval.cz : Java server pages pro vsechny* [online]. 6.8.2002 [cit. 2010-05-19]. Dostupné z WWW: <http://interval.cz/clanky/javaserver-pages-pro-vsechny/>.
[6] *Wikipedia* [online]. 24.4.2010 [cit. 2010-05-19]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Relační_databáze>.
[7] *Root.cz : JBoss aplikacni server* [online]. 18.2.2008 [cit. 2010-05-19]. Dostupné z WWW: <http://www.root.cz/clanky/jboss-aplikacni-server/>.

Zoznam príloh

Príloha 1. súbor diff_server uložený na CD v adresári ./prílohy/diff
Príloha 2. Java aplikácia ktorá konfiguruje JCA databázové zdroje uložená na CD v adresári ./prílohy/JBossConfigJava
Príloha 3. JDBC ovládač JayBird uložený na CD v adresári ./prílohy/JDBC
Príloha 4. testovacia aplikácia Spolužiaci uložená na CD v adresári ./prílohy/Spoluziaci
Príloha 5. súbor obsahujúci zoznam nakonfigurovaných JCA databázových zdrojov ./prílohy/subory
Príloha 6. konfiguračný skript, ktorý mení java súbory uložený na CD v adresári ./prílohy/subory
Príloha 7. chybový log po prvej sérii testov uložený na CD v adresári ./prílohy/firebird_first
Príloha 8. chybový log po druhej sérii testov uložený na CD v adresári ./prílohy/firebird_second