VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

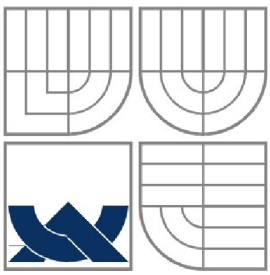
ENTERPRISE NASAZENÍ PLATFORMY OSGI

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

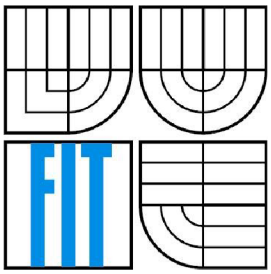
AUTOR PRÁCE
AUTHOR

DAVID PECH

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ENTERPRISE NASAZENÍ PLATFORMY OSGI

ENTERPRISE DEPLOYMENT OF THE OSGI PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID PECH

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2011

Abstrakt

Práce se zabývá problematikou technologie OSGi, která nabízí odlišný přístup k modularitě jazyka Java. Vývoj modularity jazyka je dán do souvislosti s novou technologií. Práce rozebírá praktické aspekty OSGi přístupu od návrhu aplikace, přes její vývoj až po samotné nasazení a následnou údržbu. Dle uvedených doporučení je realizována demonstrační aplikace, nad kterou jsou prováděny zátěžové a další testy s cílem nasimulovat běžný provoz serverové aplikace. Práce v závěru zhodnocuje použití technologie OSGi jako celku při tvorbě rozsáhlých aplikací.

Abstract

This thesis focuses on the OSGi technology that offers distinct approach to modularity of the Java language. The technology is shown in connection to the natural evolution of the language. The thesis discusses practical elements of the OSGi approach from application design through development stages to the server deployment and maintenance phase. An example application is build upon the presented recommendations. The application is stress-tested with several simulations of different realistic traffic types. The thesis sums up the usage of the OSGi technology and evaluates its aspects.

Klíčová slova

OSGi, jazyk Java, Eclipse Virgo, modularita, sdílené knihovny, serverové nasazení, classloader

Keywords

OSGi, Java language, Eclipse Virgo, modularity, shared libraries, server deployment, classloader

Citace

David Pech: Enterprise nasazení platformy OSGi, bakalářská práce, Brno, FIT VUT v Brně, 2011

Enterprise nasazení platformy OSGi

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Burgeta, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

David Pech
5. května 2011

Poděkování

Chtěl bych poděkovat panu Ing. Radku Burgetovi, Ph.D. za cenné rady a připomínky při tvorbě práce.

Obsah

1 Úvod

Předmětem této bakalářské práce je podrobně analyzovat technologii OSGi, která rozšiřuje jazyk Java o prvky dynamické modularity. Cílem je především zhodnotit, za jakých okolností a zda vůbec je tato technologie přínosná pro rozsáhlé podnikové aplikace spadající do kategorie enterprise.

Práce bude uvedena teoretickou analýzou vývoje modulárnosti jazyka Java v průběhu několika historických etap, na kterých lze sledovat vývoj požadavků na samotný jazyk, a zároveň vzrůstající komplexitu aplikací, kterou lze úspěšně řešit nasazením zmiňované technologie.

V jazyce Java existuje mnoho historických konstrukcí či problémů, které jsou v běžném prostředí obtížně řešitelné. Za pomoci nového přístupu lze však nalézt řešení intuitivnější a zároveň daleko jednodušší.

K práci bude přiložena i jednoduchá demonstrační aplikace, jejímž cílem není předvést rozsáhlou implementaci, ale naopak ukázat i na malém projektu rozvrstvení do racionálně definovaných a vzájemně komunikujících modulů. Vzájemná komunikace autonomních modulů aplikace je malým náhledem do dnes běžné používaných OSGi aplikací. Pro přiblížení problémů v enterprise prostředí bude aplikace využívat dnes široce používané knihovny pro různé oblasti své funkcionality.

Jádro práce se zabývá nasazením ukázkové aplikace v testovacím prostředí webového serveru. Práce podrobně analyzuje jednotlivé aspekty od zapojení technologie do vývoje až po testy dlouhotrvajícího běhu v konkrétní serverové instalaci.

Práce by měla čtenáři přiblížit a vysvětlit nejdůležitější aspekty technologie OSGi a umožnit mu odhadnout, zda je pro jeho projekty koncept přínosný či nikoliv.

2 Jazyk Java z pohledu modulárnosti

Jazyk Java již od svých prvních verzí obsahoval velmi jednoduchou podporu pro moduly. Základním stavebním kamenem všech Java aplikací jsou třídy. Třída je nejmenší jednotka programu, kterou jazyk Java dokáže rozlišit. Žádná metoda nebo kód nemůže existovat samostatně, vždy musí být obalen ve třídě.

Třída je uložena v balíčku, který logicky sdružuje části programu. Balíčky lze do sebe zanořovat a vytvářet tak složitější struktury zdrojových kódů. Balíček rovněž udává adresář, ve kterém se třída nalézá.

Běh Javy zajišťuje virtuální stroj (*JVM - Java Virtual Machine*), který dokáže vykonávat byte-kód. Byte-kód obsahuje přeloženou Java třídu, jedná se o obdobu běžného binárního překladu, nicméně byte-kód není závislý na platformě a tedy nelze jej spustit bez interpretu. Každá třída je obsažena v samostatném souboru s příponou *.class*. Vnitřní třídy překladač vyčlení do zvláštních souborů, ačkoliv ve zdrojových souborech byly součásti obalové třídy. [1]

Historicky se v Javě vyvíjela potřeba vytvářet stále rozsáhlejší systémy a s ní rostly i požadavky na samotnou modulárnost jazyka. Manipulace s jednotlivými *.class* soubory byla nepraktická, rozsáhlejší komponenty nebyly vhodně ohraničeny. Přirozeným vývojem byl jazyk Java obohacen o koncept *classpath*.

2.1 Koncept *classpath* a *classloader*

Modulárnost jazyka Java nabídla společnost Sun v konceptu *classpath* a potažmo *classloader*. *Classpath* byl zaváděn v době, kdy podporu pro dynamické linkování nabízeli všichni hlavní konkurenti jazyka Java, jako např. C nebo C++. Java oproti nim přidala pro praxi značnou výhodu - nezávislost na architektuře, která prakticky znamenala velmi snadnou znovupoužitelnost knihoven a tedy výrazné usnadnění přenosu aplikace na jiné platformy.

Virtuální stroj v jazyce Java načítá jednotlivé třídy pomocí soustavy *classloaderů*. *Classloader* je Java třída, která implementuje dvě základní operace [2]:

- Nalezení třídy dle jména - samotné načtení a interpretace *.class* souboru je v režii JVM, *classloader* pouze označí *.class* soubor, který třídu obsahuje. Nalezené třídy většinou využívají systém mezipaměti pro dosažení optimálního výkonu.
- Uvolnění paměti - neboli zahození všech tříd načtených v průběhu životního cyklu *classloaderu*.

Classloadery se uspořádávají do hierarchické struktury (od verze Java 1.2. [3]), což podporuje myšlenku modulárnosti. Zaváděcí *classloader* je kořenem hierarchie a může obsahovat další podřízené uzly typu *classloader*. Vytváření a správa těchto speciálních tříd je plně v režii programátora. Hierarchické uspořádání mj. umožňuje:

- Oddělit kontexty - vzájemnou viditelnost tříd v rámci jedné aplikace
- Zahodit některé načtené třídy (a např. na jejich místo načíst novější verze stejných tříd - neboli inkrementální překlad známý z moderních vývojových prostředí)
- Sdílení některých tříd mezi nezávislými moduly aplikace (bootstraping)

Technologii *classpath* lze chápat jako obdobu UNIX proměnné *PATH*. Tato proměnná uchovává všechna umístění, ve kterých se bude vyhledávat spustitelný program. Podobně *classpath* obsahuje seznam umístění, kde se nalézají přeložené třídy. Do *classpath* lze zadat seznam adresářů s třídami a zároveň adresáře s Java knihovnami [4].

Vývojáři sdružují logicky související třídy do knihoven - *JAR* souborů. Souboru typu *JAR* po technické stránce běžný ZIP soubor s předepsanou strukturou. Obsahuje samotné *.class* soubory včetně jejich balíčku (určeného adresáři, do kterých je vnořen) a několik dalších speciálních souborů, např. *META-INF/MANIFEST.MF* obsahuje důležité informace o knihovně jako celku (název, verze, sestavovací nástroj). Dříve byl tento soubor výhradně informační (jeho obsah virtuální stroj neinterpretuje), dnes nabývá na významu i díky technologii OSGi.

Standardní *classloader* vyhledává ve všech umístěních zadaných *classpath* konkrétní třídu - příslušný *.class* soubor. Pokud jej nalezne, nechá virtuální stroj, aby jej načetl a pak načtenou třídu vrátí zpět do programu. Uvnitř programu se *classloader* vyvolá pomocí operátoru *new*.

Z dnešního pohledu je přístup skrze *Classloader* poměrně omezující. Nelze jakkoliv pracovat s verzemi jednotlivých knihoven, navíc často dochází ke konfliktům, kdy program používá dvě odlišné verze téže knihovny, každá z nich využívá stejný subsystém např. pro logování, nicméně každá v odlišné verzi. Tyto problémy jsou dnes běžné a jejich řešení není triviální.

2.2 Naivní přístup

Nejjednodušší formou modularity je případ, kdy aplikace běží zcela samostatně a pro svůj běh může využívat všech prostředků virtuálního stroje. Touto formou se spouští řada *desktop* aplikací. Virtuální stroj je pouze interpret v nejjednodušší možné formě, aplikace se chováním podobá kterémukoliv jinému procesu v operačním systému. Mnoho programů používá zcela základní hierarchii *classloaderů*, většinou s minimálním větvením.

Aplikace může obsahovat knihovny třetích stran. Knihovny jsou v jazyce Java snadno použitelné a jedná se o velmi oblíbený způsob sdílení zdrojových kódů. V tomto jednoduchém přístupu se knihovny používají globálně - ze všech míst aplikace je k dispozici totožná třída. Pokud je volán operátor *new*, vždy se vrátí objekt totožné definice třídy.

Pro snadnější správu knihoven se v prostředí jazyka Java uchytilo mnoho balíčkovacích systémů, které dokáží knihovnu snadno najít a stáhnout včetně všech jejích závislostí. Většinou jsou odvezeny od dvou základních formátů pro repositáře knihoven - *Maven* a *Apache Ivy*. Balíčkovací systémy jdou i dále, např. *Maven* nebo *Gradle* dokáži v prostředí Javy nahradit tradiční make.

Mnoho knihoven je dnes velmi komplexních, potřebuje desítky závislostí a grafy závislostí pro ně jsou značně členité. Situace je dále komplikována nepovinnými závislostmi - submodul může a nemusí být použit. Pro ilustraci lze uvést počet závislostí *Apache Tika*, jedná se o aplikační rámec pro extrakci textu z dokumentů, obrázků a mnoha dalších zdrojů. Knihovna přímo vyžaduje 20 různých závislostí. Většina z těchto závislostí potřebuje další knihovny pro svůj běh. Pokud tedy vývojář zahrne tuto knihovnu do projektu, pak přímo nebo nepřímo musí využít desítky dalších knihoven. [5]

Graf závislostí je velmi složitý a především je z něj patrný vážný problém - některé knihovny se zahrnují opakovaně a ve více verzích. Java pracuje s nejmenší jednotkou - třídami, proto běžně nastává následující situace:

- Zahrnutí 2 stejných knihoven v Classpath není detekováno virtuálním strojem (některé knihovny však toto umí detekovat)
- Classloader vždy hledá třídu dle názvu, pokud tedy obě verze knihovny obsahují třídu se stejným názvem, *classloader* ji načte pouze jedenkrát (buď starou, nebo novou verzi)
- Operátor *new* vrací pro jednu z knihoven správnou instanci třídy
- Pro jinou vrací nevhodnou verzi, aniž by virtuální stroj detekoval jakoukoliv chybu
- Chyba nastane až v případě, že ve třídě nebude existovat metoda nebo konstruktor s volanou signaturou (a bude se jednat o výjimku za běhu)

Další komplikace se pojí s nepovinnými závislostmi. Mnohdy tímto způsobem řeší rozsáhlé knihovny své potřebné závislosti - pokud např. pro odesílání emailů poskytují několik alternativ, umožní tak programátorovi si vybrat a nevnucují mu jedno řešení. Pokud si uživatel žádnou ze závislostí nezvolí, případně omylem použije jinou, je vyvolána výjimka (*ClassDefNotFound*) za běhu aplikace. O chybu v době komplikace se jedná pouze, pokud programátor používá některé API z nepovinné knihovny přímo, což není běžné (už z toho důvodu, že hlavní knihovna by měla koncového uživatele odprostit od detailů implementace svých závislostí).

Částečným řešením těchto problémů byla např. metoda přebalíčkování staré knihovny pomocí programu *jarjar* [6]. U všech tříd z knihovny se změnil balíček, většinou se prefixoval. Programátor tedy ve své aplikaci mohl používat starou verzi knihovny (např. z důvodu stability) a novější verzi knihovny ponechat v jejích běžných balíčcích.

Výsledkem byl často vznik nové knihovny, která byla totožná s již existující, pouze se lišila v balíčku. Vhodným příkladem je *tomcat-juli*, což je pouze znovu vydaná verze *commons-logging*.

2.3 Enterprise Archive (EAR)

S přibývajícím počtem závislostí je v jazyce Java začal uchycovat novější formát pro nasazení zvaný *EAR* (enterprise archive). Jedná se o několik JAR knihoven spojených do jediného souboru. [7]

EAR nabízí jednodušší manipulaci s rozsáhlými programy, snadnou správu, a zároveň si uchovává vnitřní strukturu jednotlivých modulů.

Strukturou není *EAR* většinou složitější než běžná *JAR* knihovna (může se lišit dle aplikačních kontejnerů) - všechny moduly mají vzájemnou viditelnost a *EAR* se chová celistvě. Širšímu rozšíření zabránilo pravděpodobně uvedení tohoto formátu jako součást jazyka Java v edici enterprise, namísto standardní. Formát tak není dostatečně znám širšímu okruhu programátorů.

Před zavedením formátu *EAR* se používal jednoduchý přístup, kdy se všechny knihovny nahrály do jediného (poměrně objemného) *JAR* souboru. Vznikla tak rozsáhlá knihovna. Často ale docházelo k problémům - např. soubory ve stejných umístěních se přepisovaly (speciální soubory v *JAR* archivu, nikoliv třídy). Z knihoven se musely odebrat elektronické podpisy, což poškozovalo koncept bezpečnosti, případně se některé soubory musely zcela přegenerovat. Oproti tomuto přístupu umožňuje *EAR* zachovat knihovny v původním vydání beze změn.

2.4 Sdílené knihovny v aplikačních serverech

Mnoho aplikací se chová jako kontejner pro menší programy nebo komponenty. V prostředí jazyka Java mají svou dlouhou tradici aplikační servery, které umožňují spustit několik nezávislých aplikací a např. je zpřístupnit z webu (potom se jedná o webový server). Nicméně obecně lze hovořit o aplikačních serverech, protože mohou poskytovat i mnoho jiných služeb - integrační, souborové apod.

Podstatou aplikačních kontejnerů je izolovaně aplikace spravovat a sdílet zdroje (paměť, CPU). Izolace je zde vyšším stupněm modularity než u EAR archivů. Hierarchie classloaderů se často používá propracovaněji, některé části jsou společné pro všechny aplikace uvnitř kontejneru. Vytváří se tak prostor pro sdílení některých tříd mezi aplikacemi uvnitř kontejneru. [8]

Např. u aplikačního serveru *Tomcat* je struktura *classloaderů* členěna na 4 části. Kořenem v hierarchii je *Bootstrap* část, která obsahuje esenciální knihovny virtuálního stroje Java (např. balíčky *java.lang*, *java.util*). Systémová část zahrnuje veškeré knihovny předané při startu aplikace skrze proměnnou *classpath* virtuálnímu stroji. Tyto dvě části jsou pro každou aplikaci v jazyce Java obdobné.

Server *Tomcat* a jeho závislosti se ukládají v classloaderu *Common*. Zjednodušeně řečeno se v této fázi zpřístupní obsah všech *JAR* souborů z podadresáře *lib*. Tyto třídy jsou následně viditelné všem aplikacím uvnitř kontejneru. Pro každou aplikaci se vytváří samostatný *classloader*, který je v hierarchii zařazen pod *Common classloader*. Zpřístupňuje definice tříd ze souboru typu *WAR*, tedy ze dvou umístění:

- */WEB-INF/class* - přeložené soubory *.class*
- */WEB-INF/lib* - knihovny specifické pro webovou aplikaci ve formátu *JAR*

Konkrétně aplikační server *Tomcat* obsahuje invertovaný classloader. Jeho chování se odlišuje od běžných classloaderů - pokusí se třídu vyhledat nejdříve v rámci konkrétní webové aplikace (*WAR* souboru) a při neúspěchu použije tradiční cestu - dotazování se od kořene stromu classloaderů (*Bootstrap => System => Common*). Existuje několik výjimek (jaderné třídy z *JRE*), kdy se tato delegace neuplatňuje. Cílem tohoto schématu je umožnit webové aplikaci přepsat definici některé z tříd nabízených v rámci aplikačního serveru. Důvodem může být např. aktualizace konkrétní knihovny z hlediska bezpečnosti nebo požadavek na odlišnou implementaci ospravedlněný požadavky na výkon.

Schéma zavedené aplikačními servery je výrazným krokem kupředu v zavádění modularity do jazyka Java, nespornou výhodou se stává reálná možnost sdílení tříd v rámci knihovny. Sdílení má pozitivní vliv na velikost aplikace v paměti, optimalizace (pomocí technologie *HotSpot* aplikované v *JVM*), nicméně přináší další míru komplexity při sestavování programů.

Programátor musí být dobře seznámen s konfigurací serveru, aby věděl, které knihovny musí zahrnout přímo do WAR souboru své aplikace a které nikoliv. Server *Tomcat* v tomto ohledu nabízí inovaci v podobě invertovaného objektu *classloader*, v mnoha konkurenčních a především enterprise aplikačních serverech uživatel možnost přepsat třídy ze sdílené knihovny vůbec nemá. I zde je programátor většinou nucen do značné míry distribuovat *WAR* soubor na míru konkrétní konfiguraci serveru.

Jednou z dlouholetých tradic v jazyce Java je vydávání specifikace pro konkrétní moduly v rámci *JSR* (*Java Specification Request*) jako balíček rozhraní bez implementace (např. technologie

SAX nebo *JPA*). Rozhraní jsou obecná, implementaci může poskytnou více výrobců (a většinou poskytuje). Pokud tedy používáme v rámci aplikace tato rozhraní, musíme nezbytně přiložit do *WAR* souboru implementaci. Zároveň ale může jinou implementaci sdíleně poskytovat aplikační server, vzniká tedy problém, která z nich se při běhu aplikace použije. Ačkoliv by se mohlo zdát, že implementace jsou díky použití rozhraní zcela zaměnitelné, realita taková mnohdy není. Buď některá z implementací poskytuje vyšší výkon nebo neobsahuje některé chyby apod.

Dalším z důvodů, proč musí programátor na míru sestavovat WAR soubory, je neexistence mechanizmu, jak specifikovat, které knihovny aplikace potřebuje k běhu. Webová aplikace jednoduše očekává, že definice tříd k dispozici jsou, pokud nikoliv, vyhodí se za běhu výjimka *ClassDefNotFoundException*.

V neposlední řadě je nutno připomenout, že některé starší knihovny (které jsou ale z důvodu kompatibility v enterprise prostředí používány) používají statické metody tříd, skrze které nabízejí implementaci svých služeb. V této implementaci nerozlišují, ze které webové aplikace jsou jejich metody volány. Tento přístup vede k nemožnosti takovou knihovnu sdílet, protože jakmile si uloží libovolná data (např. jako mezipaměť), vzniká prostor pro bezpečnostní díru. Vývojář si toho nemusí být vědom a vystaví svou aplikaci bezpečnostní chybě, většinou však těžko odhalitelné a predikovatelné.

V praxi se mnoho společností uchyluje k nasazování aplikačních serverů, které obsahují minimum sdílených knihoven, všechny definice tříd přikládají do distribučního *WAR* souboru webové aplikace. Důvodem je možnost distribuce jediného *WAR* souboru napříč celým spektrem zákazníků namísto složitého ladění podporovaných konfigurací knihoven.

2.5 Technologie OSGi - modulární přístup

Narůstající potřeby modulárnosti jazyka Java se rozhodla řešit *OSGi alliance*. Projekt vznikl již v roce 1999 [9], nicméně mezi širší programátorské publikum se dostal až v posledních letech. Klíčem k jeho rozšíření je především úspěšné nasazení v rozsáhlých projektech (např. integrované vývojové prostředí *Eclipse*).

OSGi si klade za cíl umožnit v jazyce Java vytvářet dynamické knihovny, které si explicitně definují veškeré třídní závislosti. OSGi není pouze koncept tříd, ale jde ještě dále: umožňuje definovat služby, které daný modul (neboli *bundle*) nabízí a které naopak sám vyžaduje (návrhový vzor *producer - consumer*). OSGi si lze představit jako kontejner, ve kterém autonomně běží jednotlivé moduly, mezi kterými lze zasílat zprávy.

OSGi alliance vydává specifikace (aktuálně verze 4.2), které popisují chování a detailní API na úrovní rozhraní. Dle této specifikace se utvářejí konkrétní implementace OSGi kontejnerů, mezi nejznámější patří *Eclipse Equinox*, *Apache Felix* nebo *Knopflerfish*. Každý implementuje specifikaci v různé šíři, dnes je za nejrozsáhlejší považován *Equinox*. [10]

2.5.1 Důvody pro zavedení OSGi

Modularita nikdy nebyla silná stránka jazyka Java, *OSGi alliance* nemá za cíl sestavit pouze specifikaci, ale koordinovat široké hnutí, které se snaží situaci napravit [9]. V jazyce Java dnes běží velmi komplexní programy (řádově miliony řádků kódu), které jsou rozsáhlé, složité a těžko

udržovatelné. U mnohých je vyžadována vysoká dostupnost, *SLA* (*Service Level Agreement*) či další požadavky na kvalitu.

V běžném aplikačním serveru nelze za běhu manipulovat s již načtenými aplikacemi. Většinou je možnost je jako celek restartovat. Pomocí zahození části hierarchie *classloaderů* lze teoreticky vyměnit pouze část běžící aplikace, ale tento postup je značně složitý, náchylný k chybám a náročný na vývoj. S technologii OSGi lze modul restartovat nebo dokonce aktualizovat na novější verzi za běhu s minimálním výpadkem ve službách.

Jakmile začne být vývojář závislý na knihovnách třetích stran, téměř okamžitě se dostane do problémů s verzováním knihoven popsaných dříve. Tyto problémy nemají snadné řešení a vyžadují časově náročné přebalíčkování knihoven, případně dokonce přepsání některých částí kódu, což vyžaduje zaučit programátora v cílové problematice. S technologii OSGi lze transparentně používat více implementací stejné knihovny nebo služby. Koncový modul si zvolí, kterou z těchto implementací potřebuje, případně v jaké verzi.

OSGi nabízí i více prostoru při vývoji samotného projektu. Vhodným použitím OSGi konceptů lze těžit z rozdělení do modulů. Vývoj lze separovat na menší jednotky, které lze snadno testovat a samotné manažerské vedení projektu pak může být jednodušší.

2.5.2 Hlavní funkce OSGi

OSGi technologie definuje kontejner jako prostředí pro spouštění a správu modulů. Modul (*bundle*) je oddělená knihovna sestavená z definic tříd, která plně slouží určitému úzce vymezenému cíli. Samotný kontejner poskytuje modulům některé běžné služby jako:

- logování
- webový server
- diagnostika chyb
- administrátorská konzole
- validace XSD schémat

Jednotlivé moduly mezi sebou mohou vzájemně komunikovat pomocí rozhraní služeb. Dodavatel (*producer*) definuje služby, které nabízí a odběratel (*consumer*) se přihlásí k odběru dané služby. Spojení probíhá na základě společného rozhraní.

Moduly mezi sebou mohou sdílet stejné definice tříd, konkrétní modul si vybere pouze potřebné balíčky, o které má zájem, a případně specifikuje i jejich verzi. [11]

Přirozeným způsobem se tak vytváří hierarchie mezi balíčky, která vychází ze systémových balíčků, pokračuje napříč sdílenými knihovnami a končí u balíčků konkretních webových aplikací. V rámci webové aplikace je nutné využívat třídy z jiných modulů a mnohdy vhodné využívat i jejich služby.

Pro správnou funkčnost musí každý modul v *MANIFEST.MF* souboru definovat následující údaje:

- Symbolický název modulu (většinou název hlavního balíčku)
- Verze modulu
- Balíčky, které modul exportuje a ostatní moduly je mohou využívat (v rámci konceptu *classpath*)

- Balíčky, které modul importuje (potřebuje pro svou funkci), některé balíčky lze definovat jako volitelné (vhodné u rozsáhlejších knihoven, protože ne vždy jsou všechny části využity)
- Volitelně aktivátor neboli zavádějící třídu modulu

Aktivátor je třída implementující specifické rozhraní, které obsluhuje životní cyklus modulu. Aktivátor registruje služby nabízené modulem a provádí další doprovodné akce (logování, auditing apod.). Při načítání modulu do kontejnerů se prochází především několika stavy (zjednodušeno) [12]:

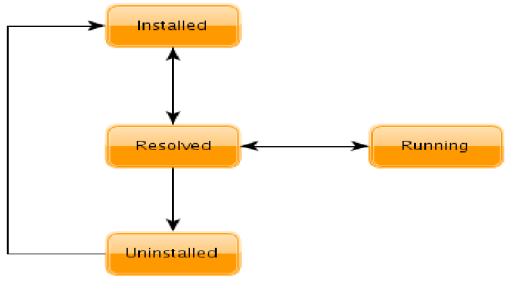


Schéma 2.1: Životní cyklus modulu

- *Uninstalled* - modul je vypnuty, není nasazen
- *Installed* - modul je správně načten, jeho závislosti nejsou vyhodnoceny
- *Resolved* - veškeré importy jsou k dispozici a modul může být spuštěn
- *Running* - modul je spuštěn a v provozu, pouze v tomto stavu lze využívat jeho služby

OSGi specifikace pro nejběžnější služby nabízí jednoduché implementace, které jsou sice minimální, snadno implementovatelné, ale často nedodržují běžné standardy v jazyce Java. Vhodným příkladem je rozhraní webového serveru, které umožňuje modulu zaregistrovat si URL adresu a na ni zviditelnit obsah pomocí vlastní implementace třídy *Servlet*. Rozhraní je jednoduché, ale ne zcela implementuje *Servlet API*, které se dnes považuje v této oblasti za standard a kterému většina výrobců své produkty přizpůsobuje. [13]

Oproti tomu jiné služby jsou implementovány v dostatečném rozsahu pro praktické užití (např. logování) a použití OSGi kontejneru zjednodušuje programátorovi práci.

Mezi moduly lze definovat služby, děje se tak pomocí rozhraní jazyka Java, které je dostupné zároveň dodavateli služby i odběrateli. Rozhraní se obalí pomocí dynamické proxy a při každém volání odběratele se volání deleguje do kontextu dodavatele, kde je požadavek zpracován. [14]

Při startování modulu se čeká na všechny služby po určitou dobu, mnohdy je tento interval poměrně dlouhý. Například dostačuje na aktualizaci modulu dodávajícího služby.

2.5.3 Chování *classloaderů* v OSGi

Koncept OSGi především nabízí především dynamické spojování modulů. Ve skutečnosti je hlavní funkcí samotného OSGi kontejneru při hledání třídy pomocí technologie classpath nalézt vhodnou třídu. Každý modul zveřejňuje svou verzi a zároveň seznam balíčků, které nabízí okolí k importování. Verzi lze zadat i u konkrétního exportovaného balíčku.

Při žádosti o vytvoření nové instance třídy se implementace konceptu classloader pokusí za běhu vyhodnotit viditelnost z daného modulu. Projde všechny importované balíčky a pokouší se nalézt konkrétní třídu v odpovídající verzi.

Importování balíčků je značně pracné (někdy moduly exportují i stovky balíčků), proto vznikají nadstandardní direktivy zjednodušující definici importů. Lze tedy zadat import celého modulu, který za běhu kontejner přeloží na import všech balíčků, které importovaný modul exportuje. Případně lze importovat celé knihovny (seznam modulů). Tato rozšíření však vznikají až v poslední době a jen minimum z nich je zahrnuto do samotné specifikace OSGi. [15]

U každého importu (modulu nebo balíčku) se doporučuje uvádět i rozmezí verzí, které jsou kompatibilní s konkrétním modulem. V praxi se využívá kvalifikovaného odhadu, protože lze očekávat, že rozhraní importovaných knihoven se do další hlavní verze nebude měnit.

Jakmile je instance třídy již vytvořena, s instancí lze nakládat libovolně, mohou ji zpracovávat i moduly, které na definici její třídy přímo nevidí a použít např. mechanizmus reflexe.

3 Enterprise aplikace

Aplikace třídy enterprise mají mnoho charakteristik, kterými se odlišují od menších aplikací, především:

- Široký rozsah funkcionality
- Rozsah zdrojových kódů - řádově stovky tisíc až desítky milionů řádků
- Týmovou spolupráci
- Požadavky na zpětnou kompatibilitu - téměř vždy je nutné v návrhu zohledňovat historické spojitosti - ať z pohledu podpory starších typů souborů, aplikačních serverů apod.
- Vysoká stabilita a požadavky na vysokou dostupnost - na systému již závisí mnoho klientů, dodavatelů a nelze jej snadno vypnout nebo odstavit
- Dlouhověkost - software většinou vzniká několik let, kvalita v průběhu vývoje kolísá
- Striktní procesní řízení - firmy kontrolují kvalitu zdrojových kódů

Vzhledem k takto širokým požadavkům enterprise aplikace vyžadují určitá minima, bez kterých by jejich vývoj byl velmi problematický, často však zcela nemožný.

V rámci enterprise aplikací vždy rozlišujeme dvě základní vrstvy:

- Aplikační rámec (*framework*) - obecná funkcionalita, kterou lze mezi projekty sdílet
- Obchodní logika (*business logic)* - konkrétní implementace systému používaného konkrétním zákazníkem

V obchodní logice se pracuje s doménovými objekty (*entitami*), které reprezentují objekty ze zákazníkova oboru.

3.1 Minimální požadavky

Z pohledu managementu platí především důraz na stabilitu ve vývojovém procesu. Každý výrobní článek musí být relativně snadno nahraditelný. Nelze tedy pracovat s experimentálními knihovnami, kterým rozumí pouze jediný člen týmu. Není čas vyvíjet podpůrné knihovny - naopak se volí časem prověřená a mnohdy i robustní řešení subdodavatelů.

Enterprise aplikace jsou ze své podstaty složité a rozsáhlé. Veškerá funkcionalita musí být viditelně oddělena, každý modul musí být vhodně separován a jasně ohraničen.

Rozdělení prací na projektu a mnohdy i kvalita práce musí být neustále měřena. Lze použít nástroje ze skupiny Continuous Integration (Jenkins) či metodiku programování řízené testy (TDD). Vhodnější je pak oba přístupy spojit. [16]

3.2 Perzistentní vrstva

Perzistentní vrstva již ze svého názvu zajišťuje bezeztrátové uložení dat. Jako úložiště se dnes používají databáze (nejčastěji relační, objektové). Běžným úkolem je zajistit mapování mezi doménovými objekty (objektový přístup) a tabulkami relační databáze. Využívá se zde technologie

ORM, která zajišťuje abstrakci nad konkrétním typem databázového systému a umožňuje uživateli v případě potřeby snadno zaměnit konkrétní implementaci databázového systému.

V prostředí jazyka Java má dlouhou tradici specifikace *JPA* (*Java Persistence API*), která definuje způsob práce a přístupu k *ORM* systému. Za celou řadu implementací jmenujme knihovny *Hibernate* a *EclipseLink* od nadace *Eclipse*.

V prostředí *JPA* je chápání perzistentních objektů odlišné od pojetí v mnoha jiných jazycích. Doménové objekty jsou totiž často s databází neustále spjaty. Při změně hodnoty se vzápětí tato změna automaticky ukládá do databáze. Tento postup zjednodušuje správu a práci s objekty, nicméně je velmi nebezpečný, pokud by došlo k „nepozornému“ přepsání některé hodnoty objektu: např. programátor výstupní HTML šablony by změnil ID objektu. Z toho důvodu se provádí přepis perzistentních doménových objektů (připojených k databázi) na transportní objekty.

Transportní objekty jsou po obsahové stránce totožné jako doménové, ale nejsou spojeny s databází, změny v nich se dále nepropagují. S tímto přepisem objektů je vhodné spojit rovněž transakční hranici aplikace. Tato hranice transparentně definuje, kdy se otevírá databázová transakce a kdy bude uzavřena (voláním *commit* nebo *rollback*). Přepis objektů na transportní je vhodné provádět těsně před ukončením transakce a následně transakci uzavřít - aby systém garantoval, že doménové objekty již nebudou změněny.

3.3 *n-tier* architektura

Pro co nejpřehlednější oddělení funkcionality aplikačního rámce od konkrétní obchodní logiky vznikl v jazyce Java princip separace povinností do několika vrstev (*tiers*). Dnes se běžně používaná moderní n-vrstvá architektura [17]:

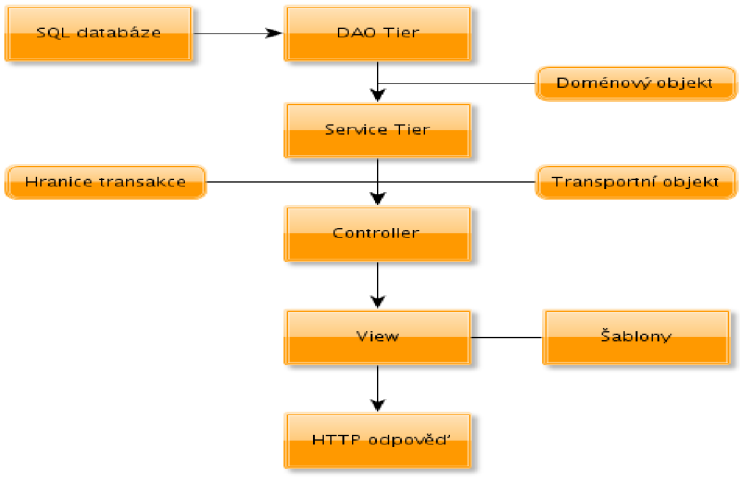


Schéma 3.1: Struktura n-tier aplikace

- Vrstva *DAO* - zajišťuje přístup k perzistentním objektům (např. přístup do *SQL* databáze), tato vrstva by měla být velmi tenká, v moderních systémech se používá relativně jednoduchá implementace konkrétní *ORM* knihovny.
- Servisní vrstva - zajišťuje přepis doménových objektů na transportní objekty. Většinou je právě zde implementována většina obchodní logiky.

- Hranice transakce - při průchodu touto vrstvou se otevírá, respektive ukončuje aktuální transakce.
- Prezentační logika - pracuje výhradně s transportními objekty.

Tento přístup umožňuje programátorovi vhodně oddělit jednotlivé úlohy při práci s perzistentnímy daty a poskytuje mu velkou kontrolu nad způsobem zpracování dat.

Pod prezenční logikou se může skrývat poměrně rozsáhlý aparát, např. část návrhového vzoru *MVC* (*Model-View-Controller*) nebo *MVP* (*Model-View-Presenter*).

3.4 IoC kontejner

Společnost SUN reagovala na potřeby enterprise segmentu aplikací speciální edicí jazyka Java nazvanou Java EE (enterprise edition). Jednalo se v podstatě o rozšíření standardní edice o velkou řadu specifikací a implementací ať už z oblasti integrací, persistence dat nebo systémů zpráv (*JMS*).

Jedním z inovativních nápadů bylo použití jednoduchých Java objektů (*JavaBean*) k řízení celého systému a předávání informací v něm. Jedná se prakticky o triviální objekty, které nabízí několik metod typu *getter* a *setter* pro přístup ke svým privátním proměnným. Řešení bylo pro mnohé vývojáře revoluční, protože po této změně nebyli odkázání pouze na třídy či rozhraní definované výrobcem, ale mohli si sami sestavovat objekty, které aplikaci řídily.

Tento koncept se dále rozvíjí v návrhovém vzoru obrácená kontrola (*Inversion of Control*), kdy programátor definuje pouze třídu a její závislosti. O víc se nestará, protože vytvoření instance, připojení všech závislostí (ve formě instancí) zajišťuje kontejner návrhového vzoru *IoC*.

Programátor tak nemusí řešit celou řadu detailů, pouze velmi přehledně definuje, jaké třídy chce používat a o propojení se postará aplikační rámec.

Návrhový vzor celou aplikaci výrazně zjednodušuje, podporuje správné programátorské návyky (hlavně odklon od statických metod a modulárního programování) a zároveň zrychluje výrobu samotné aplikace. [18]

Prakticky standardem v této oblasti se dnes stal *Spring Framework* od společnosti *SpringSource*.

3.5 *Failover*, rozložení zátěže, vysoká dostupnost

Tradičním požadavkem na enterprise aplikace je vysoká dostupnost (*high-availability*), kdy je klientům (často smluvně) garantována určitá dostupnost aplikace. Dostupnosti se mohou značně lišit, ale vždy je nezbytné v architektuře aplikace tento fakt zohlednit a dokázat se vypořádat s výpadky elektrického proudu, nefunkčního síťového připojení, pádu databázového systému a mnoha dalšími.

Řešení těchto problémů mohou být značně komplexní. Především se využívá techniky horizontálního rozložení (*scaling*), kdy několik serverů zastává totožnou funkci a v případě výpadku jednoho jej mohou ostatní zastoupit, aniž by se systém jako celek restartovat a koncoví uživatelé výpadek zaznamenali.

Používají se dvě základní techniky:

- Rozložení zátěže (*load balancing*) - všechny servery jsou dostupné a všechny zároveň vyřizují požadavky klientů. V případě výpadku jednoho z nich se jednoduše přestane používat, ostatní jej plně zastoupí
- *Failover* - jeden uzel vyřizuje všechny požadavky a další jsou v záloze. Při výpadku prvního uzlu se požadavky směřuji výhradně na jeden ze záložních uzlů. Toto řešení je technicky výrazně méně náročné, ale neposkytuje snadnou možnost navyšování celkového výkonu aplikace.

Na úrovni aplikace je nezbytné provést různé změny - např. u webové aplikace sdílení sezení s informacemi o aktuálních uživatelích. Volba OSGi jako prostředí pro běh aplikace může pomoci i v tomto ohledu. Zapínat a vypínat jednotlivé moduly je vždy výrazně rychlejší než start celé aplikace. Moduly lze poměrně snadno přenášet mezi jednotlivými servery díky jejich zřetelné separaci a aplikaci tak lze již vyvíjet s ohledem na rozdělení mezi jednotlivé servery a redundanci za běhu.

3.6 Řešení typu *cloud*

Moderní způsob prodeje aplikací staví na c*loud* modelu. Jedná se v zásadě o pronájem velkého množství totožných instancí webové aplikace zákazníkům. Každá aplikace má vlastní data, ale obchodní logika je stejná ve všech instancích. Objem pronajímaných kusů je řádově ve stovkách až desítkách tisících instancí. [19]

Tento přístup zvyšuje tlak především na kvalitu aplikace a rovněž na vysokou dostupnost, protože odstávky jsou v produkčním prostředí možné pouze minimálně.

V tomto odvětví lze s úspěchem využít modularitu jazyka Java, protože aplikace mají většinou společný základ (společné definice tříd) a využívají i stejné služby. Instance jednotlivých aplikací je nezbytné nasazovat bez restartu, stejně tak instance odebírat či přesouvat mezi jednotlivými servery.

Technologie pro *Cloud* programování mohou těžit z centralizace některých služeb - není nezbytné, aby každá instance měla vlastní implementaci, když mohou jednu vzájemně sdílet a šetřit tak serverové zdroje.

OSGi technologie jde konceptu *Cloud* vstříc. Centrálním prvkem může být sdílený *bundle*, který distribuuje služby napříč velkým množstvím *micro-projektů*, konkrétních instancí aplikace, které si klient pronajímá.

3.7 Kvalita a programování řízené testy

Již od středně rozsáhlých aplikací nastává problém s kvalitou aplikace jako celku. Není časově únosné aplikaci pokaždé spouštět před vydáním nové verze a znovu prověřovat každou možnou akci, kterou může uživatel provést. Z hlediska kontroly kvality kódu lze zavést testy, které automatizovaně ověřují funkcionalitu aplikace.

Testy se spouští pravidelně, existují v odlišných úrovních, např:

- Jednotkové testy - ověřují implementaci konkrétní třídy, testují jednotlivé metody, chování třídy apod.

- Integrační testy - simulují průchod aplikací z pohledu zákazníka, interagují s aplikací pomocí několika málo kanálů

Technika programování řízené testy dokonce psaní testů povyšuje nad samotné programování, nejdříve jsou návrhu aplikace sestaveny podrobné testy na chování jednotlivých tříd a následně vývojáři doplňují implementaci pouze do míry, kdy splňuje testy. Tato technika softwarového inženýrství se v praxi výrazně osvědčila a umožňuje udržet vysoký standard kvality i u velmi rozsáhlých produktů, kde to lidským testováním (vzhledem k rozsahu) nebylo možné. [20]

Především programátoři přecházející z menších projektů často nedokáží ocenit přínos testů. Pokud jsou totiž zvyklí na malý projekt, kde opravdu mohou po každé změně znovu a znovu vše otestovat z pohledu uživatele, pak samozřejmě testy nemusí mít výrazný přínos (pokud pomineme kvalitu). U rozsáhlejších projektů je však běžný vývoj po modulech, kdy jednotlivý tým zpracovává pouze část zadání a dle předem smluvených rozhraní komunikuje s okolím. Testy jsou pak nezbytné, protože v průběhu vývoje neexistuje možnost si aplikaci vyzkoušet z pohledu zákazníka.

Analogie rozdělení projektu do menších celků a přesné definování jejich vzájemných interakcí lze opět chápat jako analogii k OSGi modulům a jejich komunikaci skrze služby.

3.8 Životnost a údržba aplikaci

S vysokou kvalitou kódu dále souvisí volba samotného vývojového cyklu produktu. Kromě zmíněného testování je nezbytné do něj zařadit i refaktoring, tedy zlepšování zdrojového kódu při zachování stejné funkcionality. Manažeři tento postup často opomíjejí, vhodným argumentem je většinou poukázání na aktuální stav - aplikace již dosahuje požadované úrovně funkcionality, proč ji nelze dále plynule navyšovat? [21]

Pro programátora je typické, že kvalita jeho práce kolísá a k již napsanému kódu je často nezbytné se vrátit a přepsat jej vhodněji a přehledněji, aby jeho rozšiřování bylo snadnější (nebo dokonce vůbec možné).

Údržba aplikaci je prakticky bez rozdílů nejdražší položkou na ceně aplikace jako celku, je tedy nezbytné snažit se ji kvalitním návrhem a programováním minimalizovat.

Technologii OSGi samu o sobě nelze v tomto směru hodnotit jako výraznější posun ke zlepšení. Údržba je vždy definována kvalitou aplikace, tedy především kvalitou návrhu a zpracování samotných zdrojových kódů. Nasazení technologie negarantuje snazší údržbu, menší chybovost nebo transparentnější refaktoring. Nicméně lze předpokládat, že motivace k vhodnému rozdělení do projektů vyústí v menší, specializovanější projekty, které budou pro vývojáře snáze pochopitelné a pravděpodobně i jejich návrh bude méně složitý než u monolitické aplikace. Při dodržení tohoto pravidla lze předpokládat, že kód bude méně náchylný k chybám.

3.9 Modernizace a zapojení jiných jazyků

Za svůj dlouhý vývoj musí rozsáhlé aplikace čelit mnoha vlivům okolí, v programování se objevují nové trendy, do vývojového týmu přicházejí noví programátoři s novými návyky. Nemálo existujících aplikací je extrémně konzervativní vůči novým vlivům a prakticky je odmítá - modernizace je potlačena. Pro mnoho aplikací je vzhledem k jejich zavedené infrastruktuře objektivně obtížné zavést

např. nový programovací jazyk, když infrastruktura je připravena pouze pro jazyk Java. V takovém případě je nutné začínat od změny vývojových nástrojů, znovu vyhodnotit proces nasazení aplikace a zvážit mnoho dalších faktorů.

Nicméně moderní přístup a trendy mnohdy mají značné výhody - navýšení produktivity. Lze využívat moderní trendy jako jazyky specifické pro konkrétní doménu (*DSL*), které velmi čitelně umožňují popsat danou problematiku. Lze zapojit do určitého modulu funkcionální jazyk, protože daným problém je v něm elegantněji řešitelný. Rovněž se nabízí otázka, zda neprogramovat testy pomocí jiného nástroje, protože jazyk Java je v tomto směru poměrně neohrabaný.

Pro jazyk Java existuje mnoho jazyků, které lze spustit přímo pod *JVM*, např.:

- Closure
- Groovy
- Scala
- Jruby (Ruby)
- Jython (Python)

Každý z těchto jazyků má své silné stránky, které lze při vývoji vhodně zužitkovat. Je vhodné řešení rozdělit - pokud je patrné, že např. obchodní logika se vždy píše na míru a nebude se měnit, pak se nabízí použití skriptovacích jazyků, které jsou přesně k tomuto určeny.

OSGi nabízí prostředí, které dokáže jednotlivé komponenty spojit. Mostem mezi moduly jsou rozhraní jazyka Java, které každá implementace jazyka spustitelného uvnitř *JVM* umožňuje v určité podobě implementovat. Při nasazení pak není podstatné, zda uvnitř modulů běží kód v jazyce Scala nebo Java, protože jediná změna nastává při zavádění modulu. Bez OSGi by byla podobná funkcionalita jen velmi obtížně dosažitelná.

4 Běžné problémy v jazyce Java a jejich řešení v kontextu OSGi

Dynamická modularita řešena dle specifikace OSGi nabízí zjednodušení či nápravu problémů v jazyce Java. Jedná se o obecné přístupy, které lze považovat za benefity užívaní modulů. Jazyk Java je oproti svým přímým konkurentům výrazně jednodušší a velmi konzervativní.

4.1 Načítání tříd

Typický manuál demonstrující připojení k databázovému systému pomocí jazyka Java obsahuje dvě části:

- načtení ovladače ke konkrétní databázi
- použití balíčku *javax.sql* k ovládání databáze

Při vytváření spojení se používá konstrukce "*Class.forName(nazevOvladace)*", která načte pomocí technologie C*lassloader* definici třídy s ovladačem. [22] Tento úsek kódu lze považovat za archetyp, většina programátorů si jej spojí právě s vytvářením spojení do databáze.

Daný úsek má však mnohem hlubší význam - kromě načtení samotné třídy zajišťuje registraci databázového ovladače, aby jej bylo možné následně použít při vytváření spojení s databází (metoda *DriverManger.getConnection()*).

Pokud volání *forName* vývojář nepoužije, vystavuje se riziku, že ovladač nebude registrován, metoda *getConnection* selže a spojení s databází nenaváže.

Volání *forName* je ekvivalentní načtení třídy pomocí technologie *classloader*, jedná se o velmi nízkoúrovňovou sémantiku, se kterou by běžný programátor vůbec neměl přicházet do styku. Hrozí totiž řada problémů a situací:

- Překlep v názvu třídy - hledání neexistující třídy - chyba až za běhu aplikace
- Knihovna s třídou není v rámci *Classloader* hierarchie k dispozici - volání selže s výjimkou *ClassDefNotFoundException*
- Registrace pro následné volání *getConnection* není jakkoliv explicitní - uživatel musí důvěřovat tvůrcům knihovny, že pomocí vhodného mechanizmu registraci provedou
- Volání má stejný efekt jako: *TridaSNazvemOvladace.getClass()*, nicméně toto se nedoporučuje používat, protože se pak váže implementace ke konkrétnímu ovladači.
- Nelze používat dvě knihovny pro stejnou databázi (toto omezení lze však částečně obejít vhodnou hierarchií technologie *Classloader*)

Dynamické spojování modulů zde nabízí značkou výhodu. Modul si může pomocí systému importu balíčku nadefinovat, se kterými ovladači dokáže pracovat. Import lze provést jako volitelný - lze tedy přinést podporu pro velké množství databází a ovladačů a umožnit programátorovi zvolit a nasadit pouze ten nejvhodnější.

OSGi kontejner poskytuje programátorovi informace o konkrétním modulu včetně podrobností o právě importovaných balíčcích.

4.2 Konflikty verzí a závislostí knihoven

V rozsáhlejších projektech běžně dochází k nesourodosti mezi importovanými knihovnami. Typickým příkladem je současný import dvou různých knihoven, z nichž každá dále importuje jinou verzi např. logovacího systému. Tento problém nese název peklo JAR knihoven (*JAR hell*) [23]

Jazyk Java bohužel neposkytuje žádnou podporu pro řešení tohoto obvyklého problému. V rámci konkrétního kontextu se načte jediná definice třídy. Záleží na pořadí uvedení knihoven v rámci cesty *classpath,* která definice to bude.

Při běhu programu pak bude jedna knihovna fungovat správně, druhá může vykazovat problémy. Volání metody v jazyce Java probíhá podle signatury metody - jejího názvu a typů jednotlivých parametrů. Pokud virtuální stroj za běhu u třídy tuto konkrétní metody nenajde, vyhodí se výjimka.

Při vývoji by k podobné situaci nemělo vůbec dojít, protože na rozpor by měl upozornit kompilátor. V této konkrétní situaci nastane chyba až za běhu, navíc až ve chvíli volání poškozené metody. Problémy tohoto typu jsou velmi těžko odstranitelné, v praxi je nezbytné jednu z importujících knihoven přepsat, aby používala totožný logovací subsystém jako druhá.

Specifikace OSGi umožňuje importovat konkrétní knihovnu v konkrétní verzi. Elegantně tedy pro konkrétní modul využívající zastaralý logovací aparát nabídne pouze třídy ze starého aparátu, zatímco celý zbytek aplikace používá verzi novou.

4.3 Úniky paměti

Proces obnovování paměti (*garbage collection*) v jazyce Java je vždy předmětem vášnivých diskuzí, protože se jedná o jeden z hlavních rozdílů oproti srovnatelným konkurenčním programovacím jazykům. Běžné obnovení paměti využívá algoritmu *mark-and-sweep*, kterým označuje a následně uvolňuje nepoužívané objekty na haldě.

Pokud na objekt existuje alespoň jedna silná reference, nelze jej do recyklačního cyklu zařadit a zůstává v paměti. Tato situace běžně nastává u špatně udržovaného kódu, kde je problém reference spravovat. Dále existují až nechvalně proslulé knihovny, které pomocí nevhodného přístupu způsobují úniky paměti, např. knihovna *commons-logging* od nadace *Apache*:

- Některá z logovacích tříd si do pole ukládá všechny instance tříd, jejichž akce loguje
- Existuje vždy spojení z konkrétní instance (v podobě lokální proměnné) na logovací třídu a zároveň podobná reference opačným směrem.
- Problém nastává v případě, pokud je aplikace s instancemi odstraněna z paměti a logovací aparát zůstává běžet jako sdílená knihovna
- Na instance zůstane silný odkaz z logovacího aparátu.
- Instance tedy nelze uvolnit z paměti a vzniká únik paměti

Ladění těchto problémů je vždy časově velmi náročné, protože se projeví za běhu až v konkrétní situaci, kterou někdy nelze vhodně navodit. Reference na objekt mohou být skryty přes několik úrovní podpůrných tříd, pro vývojáře je nesnadné ukázat na konkrétní zdroj problémů.

V praxi programátor musí hlouběji zkoumat vnitřní fungování jednotlivých knihoven a odhadovat, proč si knihovny referenci ponechávají. Mnohdy je nejvhodnějším řešením vyměnit implementaci pro danou službu.

OSGi bohužel nedokáže tento problém spolehlivě vyřešit. Jedná se o princip jazyka Java a jeho uvolňování paměti, do kterého nelze přirozenou cestou zasahovat. Nicméně výhoda OSGi přístupu spočívá opět v možnosti definování importovaných závislostí z prostředí.

Výměna implementace je velmi snadná, specifikace v jazyce Java většinou definují rozhraní, stačí tedy za běhu zaměnit modul za jiný. Zároveň modul nevidí na žádné jiné třídy, než které si v importech definuje, při hledání úniků může vývojář začít hledat v importovaných balíčcích, což je v praxi výrazné usnadnění.

4.4 Aktualizace aplikace za běhu

Virtuální stroj jazyka Java poskytuje výhradně jediné místo, kdy je možné aktualizovat implementaci konkrétní třídy - při načítání její definice. Jakmile jsou vytvořeny instance této třídy, není už možné do nich zasahovat.

Tradiční postup, jak část aplikace aktualizovat se vztahuje k zahození části hierarchie *classloaderů*. Po zahození se struktura částečně obnoví, třídy se načtou znovu. Při vytváření nové instance třídy se použije již její nová, upravená definice.

Tento postup je poměrně obtížný na provedení, vyžaduje od programátora hluboké znalosti a samozřejmě je obtížné jej odladit.

V běžném provozu je tento problém řešen uvnitř aplikačních serverů, které umožňují aplikaci odebrat právě odebráním určité části načtených tříd z hierarchie. Tento postup však často naráží na komplexní problémy úniku paměti (často díky nevhodné implementaci pomocí statických metod tříd). Pokud se serveru nepodaří aplikaci odebrat, může se stát, že kontextovou cestu drženou starou aplikací již neuvolní a tím neumožní aktualizaci aplikace. Řešením je používat rozdílnou kontextovou cestu, což je často vázáno na použití webové proxy a dále komplikuje situaci. Tradičním řešením je pak restart celého aplikačního serveru se všemi nedostatky, které postup přináší.

4.5 Použití integračních technologií

Počítačové systémy jsou ze své podstaty modulární. Dle tradičních UNIX principů [24] by každý program měl být jednoduchý a řešit pouze úzce vymezený problém. Specializace umožňuje věnovat problému náležitý prostor a vyřešit jej obecně a precizně.

Mnoho systému je však monolitických, modularitu postrádají již ze své podstaty nebo jen z nedostatku kvality návrhů svých tvůrců. K prostředí jazyka Java EE již dlouho patří rozsáhlé integrační systémy. Jejich rozsah je značně široký, protože si kladou ambiciozní cíl - umožnit snadno propojit heterogenní systémy. Mezi nejznámější se řadí *Apache Camel* nebo *Spring Integration*.

Obsahují velké množství koncových bodů (např. pro čtení souborů z FTP, JMS, AMQP, práce s IMAP atd.). Část jejich funkcionality však pokrývá integraci komponent psaných v jazyce Java. Pomocí složitého aparátu umožňují např. spojit externí účetnictví s informačním systémem. Spojení má mnoho fázi, v zásadě je např. na jedné straně HTTP server, který data serializuje do formátu XML a zasílá je příjemci, který musí data deserializovat a následně interpretovat.

Tento složitý proces však v mnoha případech není nutný, pokud jsou projekty vhodně rozděleny do znovupoužitelných modulů. Tyto moduly vzájemně sdílejí doménové objekty (odpadá potřeba serializace a deserializace) a zároveň komunikují skrze velmi jednoduchý a stabilní kanál (volání jiného modulu, tedy prosté volání metody v jazyce Java). Celý proces se velmi zjednoduší a zrychlí, ovšem pouze za předpokladu, že obě partnerské aplikace využívají principu OSGi kontejneru a je možné je spustit vedle sebe.

5 Ukázková enterprise aplikace

Hlavním cílem této práce je podrobně analyzovat platformu OSGi, aplikace je již od začátku navržena jako velmi jednoduchá ukázka využití konceptu modularizace a vzájemného spojení komponent za běhu. Aplikace je přiložena pouze jako ukázka rozvrstvení funkcionality, nikoliv funkcionality jako takové.

Pro aplikaci byl zvolen název BlogCity, jejím cílem je umožnit spouštět jednoduché blogovací aplikace na společném základu. Cílem je co nejvíce zjednodušit implementaci nového blogu za využití stávající infrastruktury. Zároveň v rámci demonstrace možnosti spojení modulů připojíme monitorovací projekt nazvaný Portál, který bude informovat o počtu spuštěných blogů a lákat uživatele objednat si vlastní instanci blogu.

Každý blog bude mít jednoduchou administrační část chráněnou heslem, přes kterou lze příspěvky přidávat. Zároveň si může každý bloger upravovat několik polo-statických ploch (např. panel O mně v postranním panelu).

Konkrétní blog by měl reprezentován velmi tenkou vrstvou aplikace, aby se redukovalo množství kódu, které je nezbytné sestavit pro spuštění další instance blogu. Vzhledem ke snazší správě bude potřebná centralizace aplikační logiky a uložiště dat. V aplikaci bude tento koncept reprezentován sdílenou logikou umístěnou do společných modulů a zároveň sdílením jediné instance databáze.

5.1 Zadání a případy užití aplikace

Z technického hlediska musí být aplikace spustitelná v OSGi kontejneru i mimo něj, zároveň musí být optimalizována na přidávání nových instancí blogu.

Z pohledu návštěvníka blogu:

- Prohlédnout si nejčerstvější příspěvky na blogu (Homepage)
- Procházet archiv příspěvků členěný dle měsíce
- Shlédnout detail konkrétního článku
- Přečíst si další informace - o autorovi, kontakt

Z pohledu přispěvatele na blog (vychází z návštěvníka):

- Přihlásit se do administrace
- Spravovat příspěvky na blogu
- Editovat polo-statické texty

Běžný uživatel portálu:

- Zjistit více informací o projektu
- Zjistit počet běžících blogů a odkazy na ně
- Přečíst si, o které články se ostatní návštěvníci blogů právě zajímají

5.2 Návrh

Podpora znovupoužitelných a snadno srozumitelných modulů je základním kamenem návrhu. U každého modulu je nezbytné definovat jasné komunikační kanály pomocí rozhraní jazyka Java. Skrze tyto kanály budou nabízeny služby konkrétním webovým aplikacím, kterých může být potenciálně velký počet.

5.2.1 Návrh aplikace - rozdělení do projektu, verzování

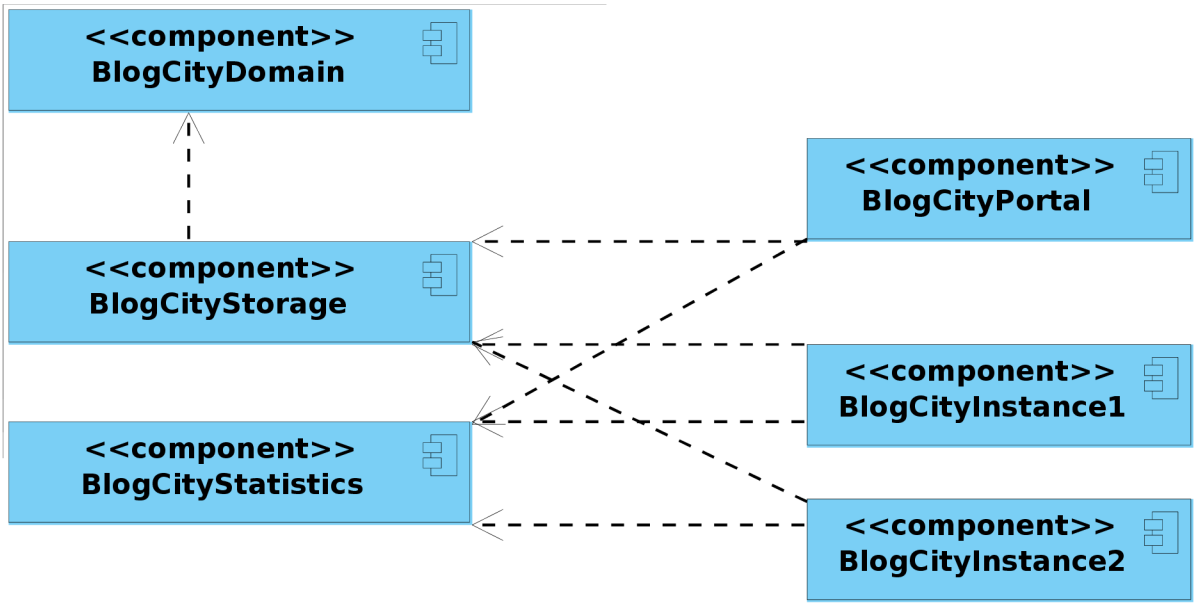


Schéma 5.1: Rozdělení projektů

Rozdělení a závislosti mezi projekty jsou znázorněny na schématu. Klíčový je projekt *storage*, který bude zajišťovat perzistenci pro doménové objekty.

Napříč aplikaci jsou ustaveny čtyři komunikační kanály, jinak jsou moduly samostatné. Ve všech kromě *domain* poběží samostatný *Spring* kontext. Webové aplikace (*portal* a jednotlivé *instance*) budou komunikovat nižšími vrstvami:

- *storage-blog* - servisní vrstva pro příspěvky na blogu
- *storage-staticContent* - servisní vrstva pro polo-statické texty
- *statistics-runningInstances* - registrace běžících instancí blogu (zobrazí se seznam všech běžících instancí)
- *statistics-lastVisited* - při každém přístupu na detail článku se zapíše do seznamu „Návštěvnici si právě prohlížejí tyto články“ a bude později zobrazen na portálu

Verzování je řešeno vždy v souvislosti se zveřejněním verze zákazníkovi nebo širšímu publiku. Při vývoji se budeme prakticky výhradně držet totožné verze (*SNAPSHOT*), verzování se navýší při vydávání oprav aplikace.

5.2.2 Návrh struktury tříd - ukázka z modulu pro perzistenci dat

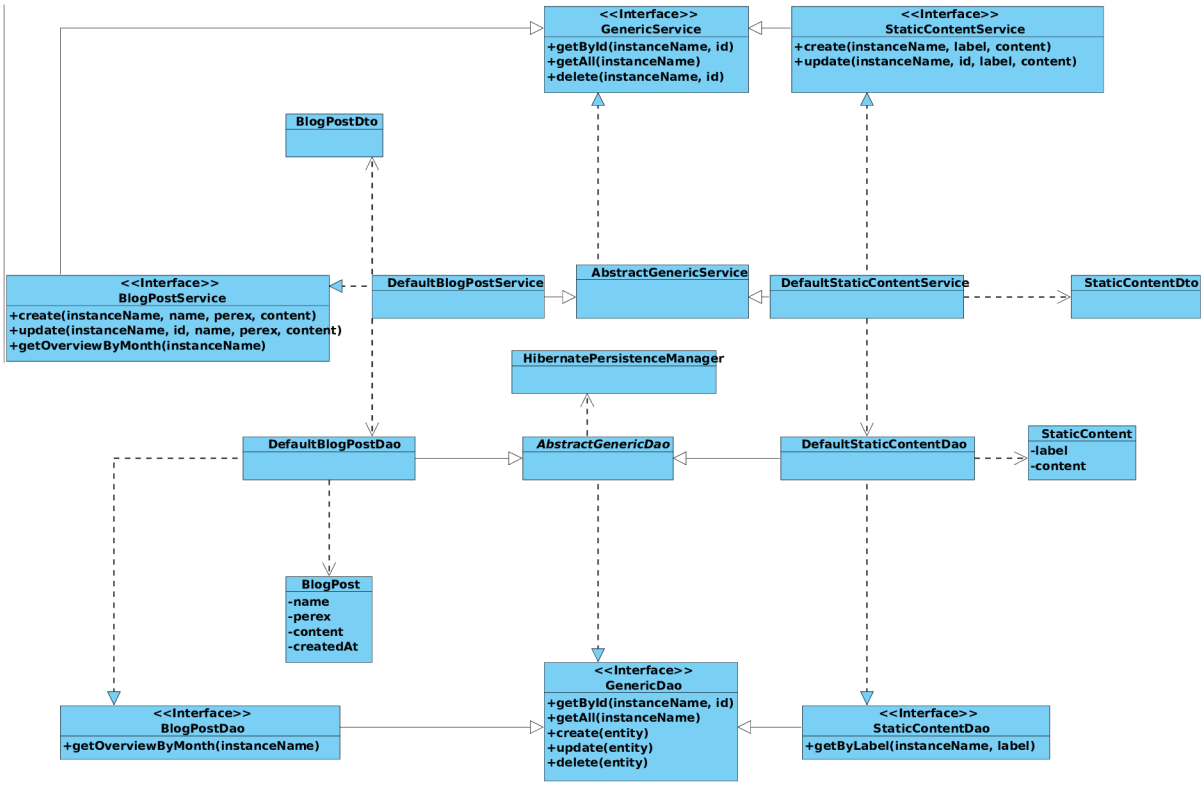


Schéma 5.2: Třídní diagram modulu pro perzistenci dat

Na grafu je viditelné schéma projektu *storage* znázorněné pomocí třídního diagramu. V jeho jádru stojí rozhraní:

- *GenericDao* - obecný přístup k objektům, základní metody pro perzistenci, pracuje s doménovými objekty připojenými k perzistentní vrstvě
- *GenericService* - přístup k transportním objektům, implementace obsahuje hranici transakce.

Z těchto rozhraní se větví další návrh, především abstraktní implementace a následně konkrétní implementace pro jednotlivé entity obchodní logiky.

Využívání rozhraní odděleného od implementace umožňuje snadno konkrétní implementaci zaměnit (například za testovací). Zároveň je tento postup vhodný pro použití v *IoC* kontejneru *Spring*.

5.3 Implementace

Při implementaci bylo využito následujících knihoven, záměrně byly zvoleny netriviální komponenty, které jsou v dnešní době na trhu běžně používané:

- *Git* - verzovací systém zdrojových kódů

- *Gradle* - překladový systém, náhrada za *Apache Ant*
- *Virgo Web Server* - OSGi kontejner s integrovaným webovým serverem *Apache Tomcat*
- *Hibernate* - *ORM*, implementace *JPA2*
- *Spring* - *IoC* kontejner
- *Spring MVC* - *MVC* aplikační rámec
- *Spring Dynamic Modules* - integrační knihovna mezi aplikačním rámcem Spring a technologií OSGi
- *Apache Velocity* - šablonovací aplikační rámec
- *Jetty* - testovací webový server
- *jUnit* - jednotkové testy

Vývoj probíhal do značné míry dle metody programování řízené testy, což jej značně urychlilo a zjednodušilo i počáteční nasazování.

Pro simulaci reálného prostředí, se v rámci vývoje využije jednoduší webový server *Jetty* a zároveň bude existovat možnost nasadit OSGi verzi na *Eclipse Virgo*. Vše vyřeší sestavovací systém *Gradle*, který nám umožní pro každou verzi generovat obě varianty distribučních archivů.

5.3.1 Zajištění OSGi vlastností aplikace

OSGi specifikaci je nezbytné brát v potaž již při samotné fázi návrhu. U aplikace BlogCity lze využít služby webového serveru a logovací služby z kontejneru OSGi.

Zároveň je nezbytné sestavit vhodně soubor *MANIFEST.MF* popisující veškeré využívané balíčky modulu. Lze použít automatizované nástroje jako *BND tool* nebo *Spring Bundlor*, které dokáží sestavit seznam všech importovaných balíčků aplikace automaticky. Nástroj se nastaví pomocí jednoduché šablony *template.mf* a pak automaticky projde veškeré zdrojové a konfigurační soubory, pokusí se najít veškeré odkazované třídy a sestaví výsledný soubor *MANIFEST.MF*, který je přiložen do distribučního archivu.

Automatizovaný nástroj bohužel nedokáže vždy odhalit všechny importované balíčky, je potřeba zasáhnout ručně a některé třídy předdefinovat. V zásadě lze však říci, že *Bundlor* svou práci plní dobře a výrazně vývojáři pomáhá.

5.3.2 Knihovny podporující standard OSGi

Všechny importované balíčky aplikace musí být v OSGi kontejneru k dispozici, jinak nastává chyba a modul není vůbec spuštěn. Bohužel specifikace OSGi stále není mezi mnoha výrobci přijímána.

V praxi převod běžné knihovny na její variantu schopnou běžet v OSGi režimu neznamená více něž doplnit soubor *MANIFEST.MF*. Ač se tento úkol může na první pohled zdát jednoduchý, ve většině případů je komplikovaný a netriviální.

Mnoho projektů je, ať už z historických důvodů nebo jen vlivem nevhodného návrhu, rozděleno do modulů způsobem, kdy moduly na sobě závisejí vzájemně a vytváří cyklický graf. Při běžném použití technologií c*lasspath* toto není problém, protože obě knihovny jsou načteny ve stejný okamžik a obě tedy naplní všechny své importované závislosti.

V OSGi kontejneru je však nutné závislosti uspokojit již při jednotlivém nahrávání knihovny do kontextu. Řešením je spojení těchto souvisejícím modulů do jediné (potenciálně rozsáhlé)

knihovny. Tato knihovna bude zahrnovat závislosti všech spojených modulů dohromady, což ji učiní náročnou na správu, ale vyřeší se problém načítání knihoven bez hlubších úprav samotné knihovny.

Jakákoliv manipulace s knihovnami navíc klade vysoké nároky na programátora, který jí musí do hloubky porozumět, aby s ní dokázal pracovat.

Nemálo výrobců však již začalo na standard OSGi dbát a jejich knihovny jsou vydávány včetně správných *MANIFEST.MF* souborů. Mezi lídry v této oblasti se řadí společnost *SpringSource*, která dokonce vydává nové verze cizích knihoven obohacené o OSGi definice.

5.3.3 Převod knihovny do kontextu OSGi

Knihovna pro perzistenci dat a objektově relační mapování *Hibernate* byla do projektu začleněna ze dvou důvodů:

- standardní řešení pro oblast persistence dat, široce podporované hlavními výrobci
- zcela chybějící infrastruktura z pohledu specifikace OSGi

Na převodu této knihovny do kontextu OSGi lze velmi názorně předvést celý proces. Knihovna trpí mnoha nedostatky, které proces dále komplikují:

- projekt je rozdělen do tří menších knihoven, které na sobě cyklicky závisí
- projekt jako celek má celou řadu závislostí (generování byte-kódu, zpracování jazyka atd.)
- zároveň s projektem je distribuována speciální Hibernate verze standardu JPA2 velmi nestandardně zabalená
- knihovna Hibernate v sobě obsahuje desítky submodulů od konfigurace skrze jazyk XML, konfigurace řízenou anotacemi až po obohacování doménových objektů za běhu a změny jejich byte-kódu (obdoba technologie *LTW - load time weaving*)
- některá rozšíření (napojení na externí cache) jsou volitelná - musí být rozpoznány a odlišeny
- součástí distribučního archivu jsou i testy, které vytvářejí další závislosti mj. na knihovnu *jUnit*

Při analýze všech importů knihovny pravděpodobně dojde k situaci, kdy nedokážeme pojmout všechny importované balíčky, vytvoříme fungující OSGi modul pro naše potřeby, ale pokud by jej chtěl používat někdo s jinými závislostmi, může narazit na problém.

Nejvhodnější je začít sloučením všech pod-knihoven v jednu celistvou. Odpadají tak problémy s cyklickými závislostmi a lze se soustředit na importované a exportované balíčky.

Pro urychlení práce použijeme automatizovaný nástroj *Spring Bundlor*. Jeho vstupem budou:

- Sloučená knihovna *Hibernate*
- Profil OSGi kontejneru - distribuovaný v rámci webového serveru *Virgo*
- Manuálně sestavená šablona *template.mf* - slouží k ovládání nástroje *Bundlor*

V šabloně je nezbytné vyplnit symbolické jméno modulu a jeho verzi, v tomto případě lze převzít z implementace.

Do šablony zapíšeme hvězdičku pro pole exportované balíčky, neboli modul bude exportovat veškeré balíčky, které obsahuje. Tento postup není však doporučovaný, protože mnoho balíčků slouží knihovně interně a přestože třídy jsou označeny veřejnou viditelností, neměly by být k dispozici pro

import z jiných modulů. Nicméně v našem případě by dokázal vybrat interní balíčky pouze programátor samotného projektu *Hibernate*, proto ponecháme exportovat všechny.

Hlavní těžiště práce spočívá v definici vhodných importů knihovny. Automatizovaný nástroj nám při generování závislostí vypíše, u kterých knihoven jsme zapomenuli zadat rozsah importované verze. Nejběžnější strategií pro import balíčků je předpokládat, že rozhraní knihovny se do další hlavní verze nemění, jako minimální verzi lze použít aktuálně nainstalovanou verzi balíčku. Tato strategie předpokládá, že některé knihovny se budou na serveru aktualizovat, jiné zůstanou v původní (stávající) verzi.

Následuje období testování, ve kterém se metodou pokus omyl snažíme spustit aplikaci uvnitř OSGi kontejnerů a dolaďujeme poslední chybějící importované balíčky. V případě knihovny Hibernate nejsou pro naši konkretní instalaci žádné další balíčky potřebné.

Moduly Hibernate jsou věhlasné i svými problémy se zpracováním doménových objektů, u kterých vyžadují, aby se spouštěly ve stejném kontextu jako knihovna samotná. Toto lze ošetřit v aplikaci BlogCity speciální direktivou import-scope při importování modulu.

Převod knihovny je značně netriviální záležitost, která by měla být prováděna výhradně odborníkem na danou oblast, v ideálním případě přímo zahrnuta jako součást vývojového procesu. Nicméně i bez dodané OSGi specifikace knihovny lze převod provést, ač se mnohdy pracuje metodou pokus-omyl.

V nadcházející hlavní verzi 4 knihovny Hibernate je do vývoje jako jedná z nejdůležitějších vlastností zahrnuta právě nativní podpora technologie OSGi.

5.4 Nasazení desíti instancí aplikace

Po dokončeném vývoji testovací verze aplikace a uvedení všech knihoven do stavu podporujícího běh v kontejneru OSGi lze přistoupit k dalším úpravám - simulaci *cloud* nasazení. Vyhotovíme celkem 10 mírně odlišných instancí blogu, se kterými budeme pracovat dále a na nichž budeme zkoumat rozdíly nasazení oproti běžným způsobům v jazyce Java.

Instance budou pro naše užití velmi podobné, pro snadnou identifikaci se budou lišit logem v pozadí.

Všechny instance se pokusíme zprovoznit pomocí tří fundamentálně odlišných přístupů, které následně porovnáme a zanalyzujeme z různých výkonnostních, nárokových nebo vývojových kritérií.

5.4.1 Varianta A - více aplikací

Tato varianta reprezentuje nejstarší přístup k modularitě v jazyce Java - spuštění každé instance aplikace v samostatném servletovém kontejneru. Aby porovnání s ostatními variantami vůbec možné, použijeme stejný webový server, který je zahrnut v OSGi kontejneru *Elipse Virgo* - *Apache Tomcat*. Na referenčním testovacím serveru připravíme 10 vedle sebe běžících instancí *Apache Tomcat*, každou spustíme na rozdílném portu.

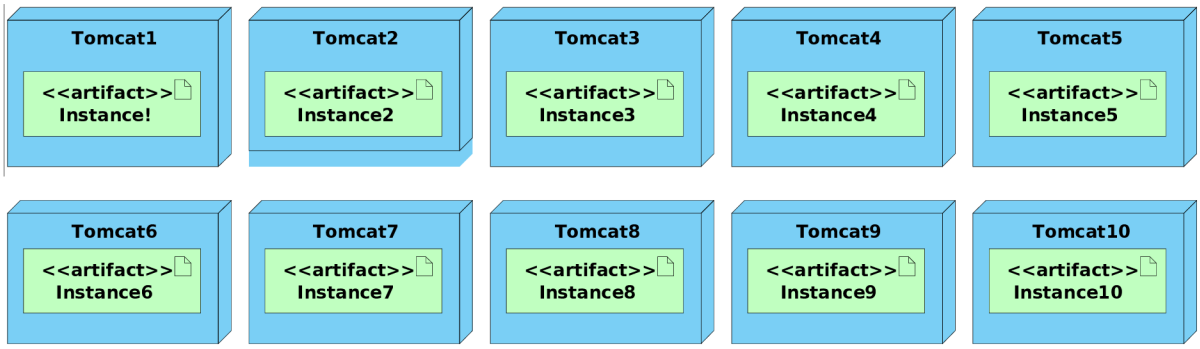


Schéma 5.3: Varianta samostatného serveru pro každou instanci

Je tedy nezbytné pro každou z 10 instancí konkrétního blogu vytvořit distribuční archiv, do kterého zahrneme veškeré používané knihovny jako závislosti.

Webovou aplikaci Portál nebudeme nasazovat vůbec, protože neexistuje jednoduchý způsob, jak by s jednotlivými instancemi blogu mohla komunikovat. Nepřejeme si totiž do této jednoduché ukázkové aplikace zavádět integrační technologie nebo systém pro zasílání a příjem zpráv (JMS).

U této varianty lze předpokládat, že bude nejnáročnější na běh a velmi obtížná na správu, protože ke každému serveru se musí přistupovat zvlášť.

5.4.2 Varianta B - sdílené knihovny

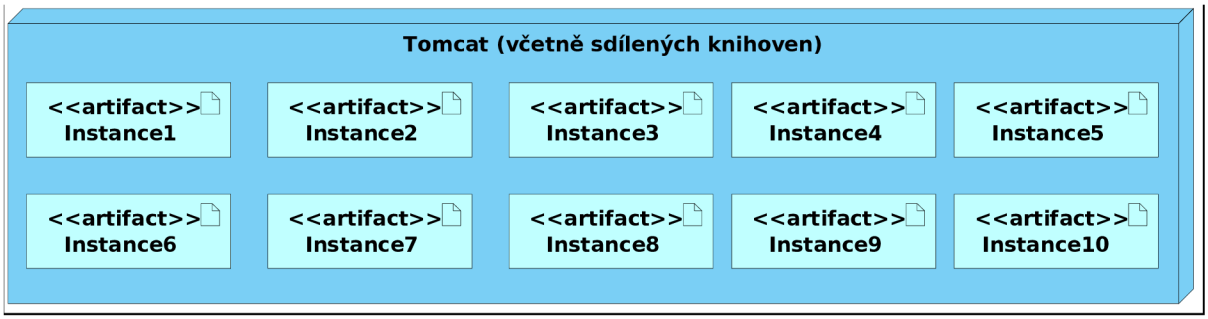


Schéma 5.4: Varianta jednotného prostředí pro běh a sdílení knihoven

Oproti předchozí variantě se bude lišit v počtu spuštěných instancí aplikačního serveru *Tomcat*. Namísto pro každou instanci samostatný spustíme jediný webový server, do kterého sdružíme všech 10 instancí aplikace.

Nasazením do společného kontejneru získáme výhodu sdílet některé části definic tříd a můžeme tedy knihovny, dříve přiložené ke každé instanci, přiložit jedenkrát přímo k webovému serveru a sdílet je napříč projekty.

Podstatné je zdůraznit, že budou sdíleny definice tříd, nikoliv instance na jakékoliv úrovni. Např. knihovna *Hibernate* bude stále pro každou instanci znovu vytvářet továrnu sezení (instanci třídy *SessionFactory*), což je její centrální prvek, který načítá velmi rozsáhlé struktury knihoven.

Nasazení lze provést na stejném portu, aplikace budou odlišeny pomocí kontextové cesty, což zjednodušuje následnou správu.

Tato varianta nasazení je dnes standardní, bude použita jako referenční vůči následující variantě.

5.4.3 Varianta C - OSGi nasazení

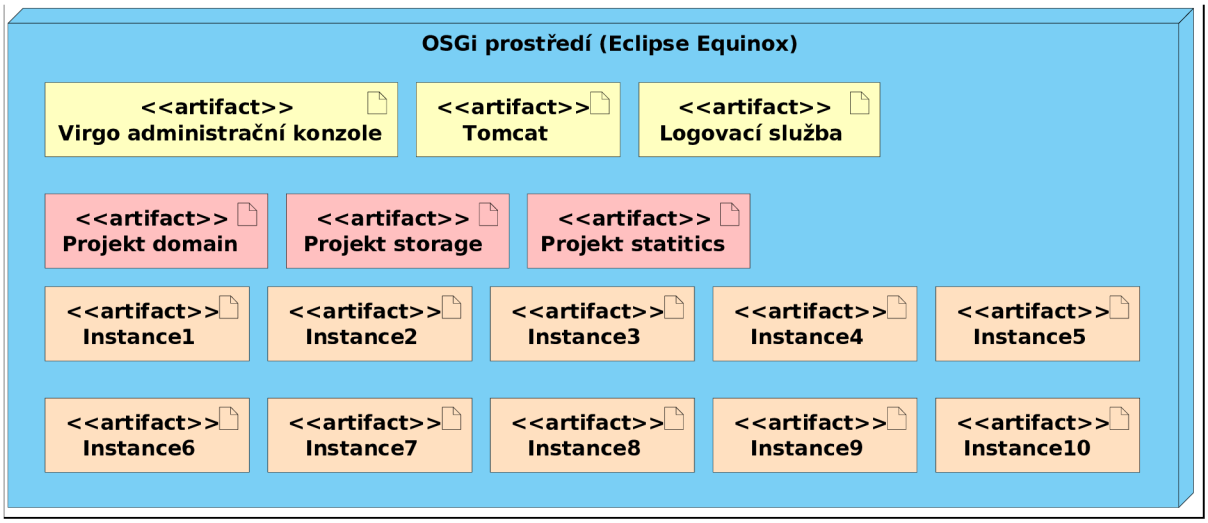


Schéma 5.5: Struktura OSGi kontejneru

Poslední varianta bude nasazení instancí do plnohodnotného OSGi kontejneru *Virgo* od nadace *Eclipse*. Teto webový kontejner vznikl z projektu *Spring dm Server*. Jeho struktura je kompletně modulární, využívající i pro samotný vnitřní chod serveru technologii OSGi a rozvrstvení do modulů. Tento aspekt je velmi důležitý, protože přináší do kontejneru vysokou přidanou hodnotu. OSGi kontejnery ostatních výrobců jsou mnohdy pouze nadstavbami nad jejich tradiční aplikační servery.

Využívat modulárnost jako součást návrhu samotného kontejneru přináší daleko větší kvalitu, protože pro jádro kontejneru (*kernel*) platí totožná pravidla jako pro nasazované moduly.

Odlišení jádra od uživatelským modulů je zajištěno umělou bariérou (*scope*). Tato bariéra je poměrně tenká, používá se především v administračních nástrojích, konzoli, k ovládání serveru, kdy systémové moduly nelze z běžícího serveru odebírat, zatímco uživatelské ano. Nicméně i systémové moduly lze v určité míře aktualizovat, což je použito i v demonstrační aplikaci (knihovna *Spring*).

Při nasazení bude nutné každý modul do aplikačního serveru nahrát jako samostatný distribuční archiv. V administrační konzoli si pak lze přehledně zobrazit, které moduly jsou právě nahrány, jaké mají závislosti a mnoho dalšího.

Kontejner Virgo navíc umožňuje naplno využít návrhový vzor inverze kontroly, protože v rámci každého modulu může být samostatně spuštěno prostředí *Spring*, které je odděleno od veškerého okolí.

Mezi moduly budou sdíleny definice jednotlivých tříd (podobně jako v minulé variantě), nicméně díky komunikaci modulů a vhodného návrhu naší aplikace lze sdílet i velkou část z celkového počtu instancí tříd. Například zmiňovaná *SessionFactory* knihovny Hibernate bude v paměti fyzicky přítomna pouze jednou.

Samostatně se bude nahrávat každá instance blogu, nicméně ty by měly být relativně nenáročné, protože obsahují pouze jeden servlet a několik málo tříd.

V tomto nasazení můžeme i naplno využít monitorovací aplikaci Portál, která bude průběžně zjišťovat informace o spuštěných instancích blogů.

5.5 Ladění aplikací v OSGi kontejneru

Při běžném ladění potřebuje vývojář svůj program spustit v kontrolovaném prostředí, které se co nejvíce podobá prostředí serverovému, a určitá místa svého programu krokovat. Tento přístup výrazně urychluje vývoj a hledaní chyb v aplikaci.

Ladění lze provádět při vývoji proti testům, kde má většina hlavních integrovaných prostředí pro jazyk Java silné zázemí a zde prakticky problém nenastává.

Při běhu aplikací nasazené na serveru je vhodné předávat parametr *-debug*, který umožní externí ladění skrze otevřený port. Tato možnost je rovněž široce používaná v reálném provozu, kdy umožňuje snadné krokování nesnadno odhalitelných chyb vznikajících za těžko simulovatelných okolností běžného provozu.

Pro samotné testování aplikace však není ani jeden z obou přístupů vhodný - pokud vyvíjíme aplikaci interagující s uživatelem, testy nikdy nedokáží pokrýt veškeré myslitelné scénáře a je vyžadováno uživatelské testování. U serverového prostředí je problematická aktualizace modulů - programátor si přeje udělat změnu na několika málo řádcích a okamžitě sledovat, jak se v aplikaci projevila.

V běžném serverovém prostředí je vhodné například spustit integrovanou verzi web serveru přímo z integrovaného prostředí, získáme pak možnost ladění a zároveň inkrementální rekompilace zdrojových kódů, což OSGi kontejner *Virgo* neumožňuje. Nelze ani s úspěchem použít jiné nástroje specializované na testování OSGi modulů, např. *PaxExam*.

Jako vhodný přístup lze vybrat jeden z následujících přístupů:

- Uzpůsobit překladový systém *Gradle*, aby po rekompilaci knihovny aktualizoval na serveru - pro ukázkovou aplikaci byla zvolena tato možnost
- Využít pro spuštění modulů jako kontejner přímo integrované prostředí *Eclipse* - tato možnost je bohužel obtížně přenositelná
- Spustit kontejner *Virgo* pomocí integrovaného prostředí *Eclipse* - řešení by vyžadovalo další do-vývoj
- Rozšíření pro kontejner *Virgo*, které by automaticky nasazovalo moduly při změně zdrojových kódů
- Zprovoznění kontejneru *Virgo* skrze serverový adaptér webové platformy *Eclipse* - velmi snadné na nastavení, bohužel nelze jakkoliv modifikovat na míru a tedy v běžné praxi nepoužitelné
- Využít technologii *JRebel* - umožní za běhu aplikace nahradit libovolnou třídu pomocí přehrání byte-kódu běžící aplikace. Jedná se však o komerční produkt.

Obecně lze říci, že tento aspekt technologie výrazně brání jejímu dalšímu rozvoji. Řešení sice existují, ale jsou složitá, vyžadují hlubší znalost technologie OSGi a začátečníky mohou odradit.

5.6 Zhodnocení vývoje pro platformu OSGi

Vývoj aplikací v OSGi se velmi usnadňuje, pokud je kompatibilita s tímto standardem brána v potaz již v počátečních fázích vývoje. Programátor si může během celého vývoje na prostředí zvykat a získávat nové zkušenosti.

Obecně lze říci, že provést nasazení OSGi standardu do již vydané aplikace nebo těsně před jejím vydáním přinese daleko více problémů než užitku. Technologie má celou řadu úskalí, v dnešní době je to především:

- Obecná neznalost OSGi mezi širším publikem programátorů a dlouhá *křivka učení*
- Relativně málo modulů je s OSGi kompatibilní, u ostatních je nutné, aby programátor do knihoven zasahoval, což je mimo jeho kompetence a často i schopnosti (neznalost cílové domény), vzniká tak celá řada zbytečných chyb
- Špatná podpora pro ladění aplikací

Výhody jsou však zvláště pro větší projekty velmi zajímavé a pro libovolný projekt většího rozsahu by určitě kompatibilita měla do určité míry existovat. I pokud se bude jednat např. pouze o kompatibilitu na úrovni dlouhodobého plánu přechodu na OSGi, jakmile budou kompatibilní všechny knihovny, na kterých aplikace závisí. Tento přístup zahrnuje vhodné dělení do projektů, udržování seznamu importovaných balíčků pro moduly a především osvětu ve vývojovém týmu, který bude na přechod připraven a osvojí si všechny aspekty technologie již během samotného vývojového cyklu.

6 Chování OSGi aplikace v simulovaném provozu

Teoretické uvedení technologie OSGi by nemělo bez praktické ukázky valné hodnoty. Pokud by měl vývojář na této technologii vyvíjet, musí k ní získat důvěru a porozumět jejímu chování v běžném provozu. Často právě běžný provoz odhalí závažné výkonnostní problémy či nestabilitu. Stačí jeden závažnější problém a programátoři jsou nuceni technologie opustit a najít náhradu v některé z konkurenčních.

Technologie OSGi za dobu své existence dokázala velmi výrazně vyspět, jako primární vývojovou platformu ji používají a prosazují přední *open-source* produkty od organizací *Eclipse*, *SpringSource* či *Apache*.

Ze své podstaty technologie zasahuje převážně do soustavy načítání tříd, kdy dokáže jednotlivá prostředí smysluplně separovat a při vytváření instancí objektu vhodně propojit definice tříd z modulů.

Při testech je nutné brát v úvahu, že web server Tomcat běží při provedení "více aplikací" a "sdílené knihovny" samostatně, zatímco u OSGi varianty je obalen nemalým OSGi prostředím *Equinox* a kontejnerem *Virgo*. Navíc v OSGi variantě bude navíc spuštěna portálová webová aplikace, která v předchozích variantách není přítomna.

Pro testování byla sestavena rozsáhlá databáze záznamů z blogů: celkem přes 23 tisíc příspěvku nerovnoměrně rozprostřeno mezi 10 instancí.

6.1 Start aplikace

Předmětem tohoto testu je najednou spustit všech 10 instancí web aplikace a měřit celkový čas spuštění. Pro vyšší přesnost budeme pokus 5x opakovat. Ověření, že jsou weby provozuschopné, vyřešíme zpracováním prvního požadavku skrze příkaz "curl". Při zpracování prvního požadavku na aplikaci se z velké většiny načte celá do paměti.

Stav webových serverů po provedení těchto kroků bude výchozí pro následující testy.

V testu odděljeme pro úplnost u web serveru *Tomcat* 2 varianty:

- Start samotného kontejneru (*Virgo*, *Tomcat*)
- Nahrání samotných webových aplikací

Opakovaný start simuluje restart celého kontejneru (lze využít částečně již zpracované aplikace a moduly). Při testech je proměnná RANDFILE nastavená na /dev/zero, abychom nemarnili čas generováním náhodných čísel. Časy jsou v uvedeny sekundách.

Popis	Varianta A (více aplikací)	Varianta B (knihovny)	Varianta C (Virgo)
„čistý“ start kontejneru	12	2	13
start včetně aplikací (pouze aplikace)	(přes 5 min)	26,7 (24,7)	45,4 (32,4)
opakovaný start (pouze aplikace)	(přes 6 min)	36,4 (34,4)	36,8 (23,8)

Tabulka 6.1: Měření času potřebného pro start kontejneru a aplikace

Variantu A bylo prakticky nemožné spustit naráz, protože přetížila server a proto ve srovnání není uvedena. Instance lze však spouštět postupně (což však nelze srovnat s ostatními), celkem na start 10 instancí potřebuje přes 5 min.

Na variantě B je velmi nestandardní prodlení při opětovném startu - zhruba 10 s oproti „čistému“ startu. Toto pravděpodobně souvisí s nasazením aplikace pomocí *WAR* archivů, které se po nasazení rozbalí. Při opětovném startu se tato rozbalená verze kontroluje (namísto smazání a znovu rozbalení), což může částečně zavinit delší načítání. U serveru *Tomcat* je zvláště důležité zdůraznit, že modul pro generování náhodných čísel byl vypnut, aby se načítání nezpomalilo nesmyslným čekáním na */dev/random*.

U varianty C bylo nezbytné pro čistý start (parametr *-clean*) moduly přejmenovat, aby je server nahrál ve správném pořadí, jinak se nenahrály dobře.

Výsledky varianty A nejsou nikterak překvapivé - aplikace ve více instancích tímto způsobem nelze vhodně spouštět. Varianty B a C jsou relativně srovnatelné, zvláště pokud odečteme start samotného kontejneru - pak se hodnoty vzájemně přibližují. Je však nutno podotknout, že ve variantě B se některé náročné služby (*Hibernate*) inicializují 10x namísto 1x u varianty C, nicméně výsledek je totožný: 10 plně fungujících instancí blogu.

Z výsledků a jejich analýzy lze konstatovat, že nasazení běžným způsobem umožňuje výrazně rychlejší start serveru.

6.2 Nároky spuštěné aplikace

Spuštěné aplikace se mohou výrazně lišit již při svém načtení - vyzkoušíme tedy po spuštění web serveru na každou z instancí vyvolat jeden požadavek (dle předchozího testu). Poté je nutné provést obnovení paměti (*garbage collection*), aby virtuální stroj vymazal všechny nepotřebné objekty a následné měření bylo relevantní. Tento test slouží rovněž jako výchozí stav pro 24 hodinové testování dále.

Velikost alokace paměti lze do určité míry nastavit, zde je údaj součtem paměti samotné aplikace a *JVM*.

Popis	Varianta A (více aplikací)	Varianta B (knihovny)	Varianta C (Virgo)
Paměť RAM (alokace)	2 900 MiB	1 200 MiB	1 100 MiB
Velikost haldy	170 MiB	68 MiB	128 MiB
Počet vláken	220	52	53

Tabulka 6.2: Srovnání náročností variant

Všechny výsledky zde splnily počáteční očekávání, hodnoty pamětí odpovídají hrubě součtu:

- Varianta A - 10 * zavedení serveru + 10 * zavedení aplikace, alokace je vysoká i díky režii *JVM*
- Varianta B - 1 * zaveden server + 10 * zavedení aplikacemi
- Varianta C - 1 * zaveden server + 10 * modul instance + OSGi prostředí

6.3 Zátěžový test

Pomocí příkazů sady *ApacheBench* zjistíme, za jak dlouhou dobu dokáže aplikace zpracovat velké množství požadavků na hlavní stránku v paralelně běžících vláknech. Testování budeme provádět pouze nad jednou instancí aplikace.

OSGi kontejner by měl mít minimální dopad na výkonnost aplikace, výsledky by v tomto testu měly být velmi podobné. V tabulce je uveden počet HTTP dotazů na hlavní stránku (celkem), počet paralelně dotazujících se klientů.

Popis	Varianta A (více aplikací)	Varianta B (knihovny)	Varianta C (Virgo)
10 000, 2 [s]	105	105	103
10 000, 4 [s]	104	103	100
100 000, 8 [s]	896	868	909

Tabulka 6.3: Výsledky zátěžového testu

Výkonnostní charakteristiky rozdílných nasazení jsou velmi podobné, rozdíly jsou minimální. U OSGi varianty je dodatečné zpracování vyhledávání spojení mezi moduly, které se na časech však projevuje minimálně. Varianty A i B by po teoretické stránce měly vykazovat totožné hodnoty, rozdíl lze přičíst fluktuacím ve výkonnosti *JVM*.

Z výsledků lze konstatovat, že technologie OSGi má na výkonnost aplikace minimální dopad.

6.4 Simulace jednodenního provozu

Dlouhodobý běh aplikací je v jazyce Java velmi diskutované téma. Chování aplikace se v čase mnohdy značně nestabilně proměňuje v závislosti na chování virtuálního stroje. Existuje mnoho návodů, které využívají celou řadu přepínačů pro virtuální stroj, snažící se chování zjednodušit a umožnit jej snáze předvídat.

U kontejnerů kromě zmiňovaného problému s paměťovými úniky hraje výraznou roli obnova paměti (*garbage collection*). Pravidla pro uvolňování paměti se v běhu virtuálního stroje často mění. Je známo mnoho případů, kdy aplikace provádějící náročnou paměťovou operaci vykazují nestabilitu:

- krátce po svém spuštění fungují správně, paměť je jim přidělena, po ukončení operace je obnovena
- po delší době nečinnosti (webová aplikace) již spuštění nefunguje správně, virtuální stroj začne aplikaci brzdit radikálním a náročným obnovováním paměti

Simulace jednodenního provozu vychází z nastartovaných instancí aplikace, na které budeme spouštět velké množství dotazů po dobu 24 hodin.

Testovací skript provádí rekurzivní stažení celé instance, následuje sekundová pauza. Tyto procesy běží dva paralelně. Simulujeme tedy zároveň zátěž a zároveň průchod do všech podstránek jednotlivých instancí, sledujeme především množství zabrané paměti a vytížení serveru v průběhu testu. Zajímavým parametrem je rovněž počáteční a koncová paměť využitá aplikací (po provedení obnovy paměti), dále pak počet načtených definic tříd, tedy počet *.class* souborů, které jsou ve virtuálním stroji právě načteny.

V tabulce je uveden vždy celkový počáteční a následně celkový koncový stav po 24 hodinách provozu. Před měřením paměti na haldě byla vždy provedena obnova paměti několikrát za sebou. Velikost paměti je uvedena v MiB.

Popis	Varianta A (více aplikací)	Varianta B (knihovny)	Varianta C (Virgo)
Paměť haldy	170 => 400	68 => 500	128 => 250
Alokovaná paměť	2 900 => 3 000	1 200 => 1 050	1 100 => 1 100
Průběh alokace	mírně stoupající	klesající	konstantní
Definice načtených tříd v paměti	58 000 => 59 430	9 240 => 9 527	9 317 => 9 521
Počet vláken aplikace	220 => 378	53 => 61	52 => 69
Zátěž na procesory	cca 195 %	cca 172 %	cca 162 %

Tabulka 6.4: Výsledky simulace provozu

Varianta A v simulaci řádově systém destabilizovala, některé webové servery se zcela zablokovaly a server byl prakticky nepoužitelný. Varianta A za 24 hod dokázala obsloužit řádově 20x méně požadavků než varianty B nebo C.

Varianta B je tradičně velmi stabilní dlouhodobé chování *JVM*. Celkový objem alokované paměti postupně klesá, protože ji server postupně přehodnocuje dle využívání prostoru haldy. V porovnání s C je především podstatné uvědomit si objem využité haldy po obnově paměti. U OSGi varianty je poloviční, což může na první pohled vypadat překvapivě, zvláště pokud si k samotnému jádru webového serveru přidáme rozsáhlý OSGi kontejner. Na druhý pohled je však patrný výrazný rozdíl, který lze vyčíst z výpisu haldy (*heap dump*) - paměťově (a časově) je velmi náročné vytvořit instanci *SessionFactory* z knihovny *Hibernate*. Tato instance zastřešuje veškerou komunikaci s databází a je velmi rozsáhlá. Ve variantě B je v paměti celkem 10x, ve variantě C pouze 1x.

Dále je pak zajímavé celkové nižší vytížení procesu při zhruba stejném výkonu. Toto lze velmi pravděpodobně přičíst na vrub technologii HotSpot a lepšímu využívání mezipaměti (právě v důsledku použití pouze jedné instance *SessionFactory*).

Výsledkem tohoto pokusu je závěr, že OSGi je co do stability blízko tradičnímu řešení pomocí webového serveru *Tomcat* a sdílení knihoven, při vhodném rozvržení modulů může být dokonce úspornější.

6.5 Aktualizace aplikace

Při tomto pokusu simulujeme vydání drobné opravy některého z modulů. Nahrajeme jeho novější verzi, stará zůstane v OSGi kontejneru přítomna. Našim cílem je sledovat, jak lze pracovat s jednotlivými instancemi, jak dlouhou odstávku je třeba přečkat.

Vydání probíhá pomocí vygenerování nového distribučního archivu s navýšenou verzí (z 1.0.0 na 1.0.1). Na server *Virgo* se nahraje archiv s novou verzi, je automaticky načten a ihned spuštěn.

Pomocí administrační konzole můžeme sledovat, že je modul načten a spuštěn. Následně stačí smazat archiv se starou verzí modulu. Kontejner zcela automaticky detekuje odinstalaci modulu, zastaví všechny moduly a pokusí se je znovu spustit. Nalezne novější verzi téže knihovny a provede spuštění. Na testovaném serveru celá aktualizace proběhla za méně než sekundu, tedy pro uživatele byla téměř nepostřehnutelná.

Pokud by závislosti mezi jednotlivými moduly nebyly na úrovni tříd (rozhraní), ale pouze nabízených služeb (společná rozhraní by byla např. v doménovém modulu), pak by aktualizace implementace modulu neznamenala ani nutnost přenačtení modulu, ale pouze by se dynamicky připojila jiná služba nabízející stejné rozhraní. Výpadek by bylo možné ještě dále minimalizovat.

Obsluha aktualizace je velmi jednoduchá v porovnání s ostatními variantami nasazení, kde by by aktualizace byla možná pouze restartem celé aplikace. Tento stav bychom mohli vylepšit, nicméně vždy bude nezbytné přenačíst aplikaci jako celek, zlepšení nabídne pouze možnost restartovat jednotlivé instance postupně a nikoliv společně (u varianty sdílené knihovny). Řešení přímým přepsáním nahraných byte-kódů v paměti (*JRebel*) může být poměrně nebezpečné a nelze se na něj zcela spolehnout.

6.6 Zhodnocení nasazení

Na základě experimentů lze provést zhodnocení nasazení OSGi platformy pro případ ukázkové enterprise aplikace.

6.6.1 Stabilita

OSGi kontejner *Eclipse Virgo* a potažmo OSGi implementace *Equinox* jsou stabilní produkty. Jakmile si vývojář po několika experimentech osvojí dynamiku fungování technologie OSGi, začne do chování kontejneru pronikat a hlouběji mu porozumí. Ve všech případech je nezbytné mít na paměti, že prostředí mezi moduly je globální - aktualizace knihovny ji aktualizuje pro všechny odběratele.

Prostředí si dokáže poradit s běžnými chybovými stavy jako například:

- chybějící knihovna
- chybějící verze knihovny
- poškozený distribuční archiv
- chybějící *MANIFEST.MF* soubor

Mnohdy však, především během vývoje, se server snaží nahrát nesmyslný MANIFEST.MF soubor, případně programátor omylem vytvoří cyklickou referenci a s tou se OSGi kontejner většinou vypořádává jen obtížně. Někdy je soustava modulů poškozena natolik, že vyvíjený modul již nelze aktualizovat (nahrát totožnou verzi), je nezbytné nahrávat verzi novější, případně server restartovat. Jen jednou se při vývoji podařilo vyvolat událost typu deadlock, kterou server však automaticky detekoval a provedl zotavení. Hlavním problémem při těchto chybách byla spíše zapnutá schopnost serveru vygenerovat zprávu o chybě, která obsahovala podrobný výpis haldy v řádu stovek MiB. Generování této zprávy vetšinou server výrazně na delší dobu zpomalilo. Toto chování však lze vypnout nebo omezit.

6.6.2 Výkon

Z výkonnostního hlediska se použití technologie OSGi výrazně projeví na startu aplikace, který je pomalejší. Na druhou stranu po odečtení načítání samotného prostředí je doba, za kterou nastartuje samotná aplikace, plně srovnatelná s běžným nasazením.

Nasazení jednotlivých modulů probíhá ve skutečně oddělených celcích, aplikace jsou tak daleko lépe chráněny a případná chyba nemá potenciál zasáhnout celý systém jako u běžné varianty nasazení. Aktualizace modulů je funkce, která v jazyce Java po velmi dlouhou dobu chyběla a v běžném provozu bude nedocenitelná, protože alternativní řešení by bylo často extrémně komplexní (nutnost použít záložní server (*failover*), což vyžaduje zásahy v aplikaci, architektuře a v mnoha dalších oblastech).

OSGi znovu-používá definice tříd, umožňuje tedy virtuálnímu stroji používat celkově méně paměti a vhodněji využívat systém dočasných pamětí (*cache*) na procesoru, ve virtuálním stroji atd. Další rozdíl je patrný rovněž díky technologii *HotSpot* přítomné v *JVM*, která statisticky sleduje nejčastěji spouštěná místa v kódu a ta optimalizuje přednostně.

Po nastartování aplikace (kromě zmíněných možností zrychlení) se rychlostní charakteristika aplikace téměř nezmění. Největší potenciál zpomalit aplikaci se nachází ve třídě *classloaderu*, při vytváření nové instance třídy. Zde je aplikována mezipaměť, takže výkonnostní rozdíl je zanedbatelný.

6.6.3 Dlouhotrvající běh aplikace

Pokus s 24 hodinovým nasazením byl do značné míry syntetický, stejně jako každý umělý test. Nicméně poskytuje nám dostatek podkladů tvrdit, že OSGi aplikace může fungovat i pod zátěží po delší časové úseky bez problémů. Zajímavá je především její zcela konstantní alokace paměti, kdy JVM ani nepřibírá další paměť, ani se ji jakkoliv nesnaží uvolnit. Při odebírání modulů však kontejner pružně reagoval a paměťové hodnoty se snižovaly.

Tento test rozhodně nepřesvědčí administrátora s několikaletými zkušenostmi, že OSGi bude fungovat stejně bezproblémově jako jeho předchozí technologie. Na tomto místě je vhodné připomenout, že i u daleko jednoduššího prostředí jako je spuštěný webový server Tomcat s jedinou

instancí aplikace, je nutno pro náročné aplikace velmi pečlivě studovat detailní nastavení *JVM*, které mohou prodloužit bezzásahový běh aplikace někdy až o měsíce [25]. Nelze očekávat, že bez těchto doladění může server *Virgo* dosáhnout podobných výsledků.

Pro OSGi technologii bude potřeba stejně bedlivě studovat dokumentaci, testovat v reálném prostředí a „alchymii“ nastavení obnovy paměti ověřit na konkrétní aplikaci. V takovém případě má OSGi server *Virgo* potenciál chovat se stejně stabilně jako mnoho jeho starších konkurentů.

7 Závěr

Tato práce se snažila čtenáři přiblížit platformu pro jazyk Java nazývanou OSGi. Byl představen kontext vývoje modulárnosti jazyka Java, ve které OSGi zaujímá pomyslný další krok oproti stávající situaci. Práce rozebírá jednotlivé teoretické i praktické aspekty přechodu na tuto technologii.

Podrobně bylo představeno možné řešení tradičních problémů jazyka, které nastávají především u rozsáhlých projektů - enterprise aplikací. OSGi přístup je v mnohém inovativní, vyžaduje od vývojářů moderní přístup a nabízí jim jinak velmi obtížně dosažitelné výsledky.

Pro demonstraci celé technologie byla vytvořena jednoduchá instruktážní aplikace, ve které byl kladen důraz především na vhodné rozdělení do modulů a jejich vzájemnou interakci. Pomocí aplikace bylo možné simulovat jednoduchý enterpise projekt s velkým množstvím knihovních závislostí. Ne všechny knihovny jsou s OSGi technologii kompatibilní, proto práce nabízí postup, jak k takovéto knihovně potřebnou kompatibilitu dodat.

Dále se práce zaměřuje na testování OSGi technologie v simulovaném provozu, snaží se navodit běžné situace - od vysoké zátěže po dlouhodobý běh aplikace - kterým bude programátor v reálném prostředí čelit a muset je řešit.

OSGi přístup je v mnohém prospěšný, vyžaduje však nemalé nároky především na straně vývojového týmu, kdy je obtížné jej plně pochopit a přejít na něj. Bohužel ne všechny běžné používané knihovny jsou s OSGi kompatibilní, což se může zdát jako obtížně řešitelný problém.

Koncepty OSGi jsou však široce přijímány, začínají je propagovat lídři v oblasti IT jako nadace Apache nebo Eclipse. Podpora těchto institucí je pro rozšíření do širšího povědomí programátorské veřejnosti nezbytná. Zároveň se objevují snahy přenést myšlenky i na jiné platformy, vzniká např. port OSGi prostředí pro jazyk C nazývaný Apache Celix.

Následující směr, kterým je možné se dále při zkoumaní OSGi vydat, leží především v nové verzi OSGi specifikace přinášející Blueprint kontejner. S ním úzce souvisí vydání nové verze projektu *Eclipse Virgo*, kde bude velký důraz kladen na stabilitu a zrychlování prostředí.

Dále v budoucnosti leží projekt *Jigsaw*, který nabízí alternativní přístup k modulárnosti jazyka Java a dle aktuálních zpráv by mohl být zahrnut do *JDK* verze 8.

Literatura

[1] VENNERS, Bill. Java World [online]. 1996 [cit. 2011-03-20]. The Java class file lifestyle. Dostupné z WWW: <http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html>.

[2] TRAVIS, Greg. IBM developerWorks [online]. 2001 [cit. 2011-03-20]. Understanding the Java ClassLoader. Dostupné z WWW: <http://www.ibm.com/developerworks/java/tutorials/j-classloader/j-classloader-pdf.pdf>.

[3] The JNDI Tutorial [online]. 2002 [cit. 2011-03-20]. Class Loading. Dostupné z WWW: <http://download.oracle.com/javase/jndi/tutorial/beyond/misc/classloader.html>.

[4] JDK Tools and Utilities [online]. 2004 [cit. 2011-03-20]. How Classes are Found. Dostupné z WWW: <http://download.oracle.com/javase/1.5.0/docs/tooldocs/findingclasses.html>.

[5] Maven Repository [online]. 2010 [cit. 2011-03-20]. Apache Tika :: parsers. Dostupné z WWW: <http://mvnrepository.com/artifact/org.apache.tika/tika-parsers/0.9>.

[6] Jarjar [online]. 2010 [cit. 2011-03-20]. GettingStarted. Dostupné z WWW: <http://code.google.com/p/jarjar/wiki/GettingStarted>.

[7] The J2EE 1.4 Tutorial [online]. 2005 [cit. 2011-03-20]. Packaging Applications. Dostupné z WWW: <http://download.oracle.com/javaee/1.4/tutorial/doc/Overview5.html>.

[8] Apache Tomcat 7 Manual [online]. 2011 [cit. 2011-03-20]. Class Loader HOW-TO. Dostupné z WWW: <http://tomcat.apache.org/tomcat-7.0-doc/class-loader-howto.html>.

[9] OSGi Alliance [online]. 2011 [cit. 2011-03-20]. About / Homepage. Dostupné z WWW: <http://www.osgi.org/About/HomePage>.

[10] RUBIO, Daniel. Web Forefront [online]. 2010 [cit. 2011-03-20]. Choosing an OSGi distribution: Equinox, Felix, Gemini or other. Dostupné z WWW: <http://www.webforefront.com/archives/2010/10/choosing_an_osg.html>.

[11] The OSGi Alliance. The OSGi Alliance [online]. 2009 [cit. 2011-03-22]. OSGi Service Platform Core Specification. Dostupné z WWW: <http://www.osgi.org/download/r4v42/r4.core.pdf>.

[12] RUBIO, Daniel. Pro Spring Dynamic Modules for OSGi Service Platforms. Berkely, CA, USA: Apress, 2009. 392 s. ISBN 1430216123.

[13] WALLS, Craig. Modular Java : Creating Flexible Applications with OSGi and Spring. United States of America : Pragmatic Bookshelf, 2009. 260 s. ISBN 1-934356-40-9.

[14] Spring Dynamic Modules [online]. 2009 [cit. 2011-03-20]. Spring Dynamic Modules Reference Guide. Dostupné z WWW: <http://static.springsource.org/osgi/docs/1.2.1/reference/html-single/>.

[15] VMware Inc. Eclipse Virgo [online]. 2009 [cit. 2011-03-20]. Virgo User Guide. Dostupné z WWW:<http://www.eclipse.org/virgo/documentation/virgo-documentation-2.1.0.RELEASE/docs/virgo-user-guide/htmlsingle/virgo-user-guide.html>.

[16] DUVALL, Paul; MATYAS, Steve; GLOVER, Andrew. Continuous Integration : Improving Software Quality and Reducing Risk. Boston : Addison-Wesley Professional, 2007. 336 s. ISBN 0-321-33638-0.

[17] CHAFFEE, Alex. Java World [online]. 2000 [cit. 2011-03-20]. Server-side Java: Counting tiers - one, two, or n?. Dostupné z WWW: <http://www.javaworld.com/javaworld/jw-01-2000/jw-01-ssj-tiers.html>.

[18] JOHNSON, Rod. The Server Side [online]. 2005 [cit. 2011-03-20]. Introduction to the Spring Framework. Dostupné z WWW: <http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework>.

[19] Cloud Computing [online]. 2008 [cit. 2011-03-20]. Dostupné z WWW: <http://www.cloudcomputing.cz/index.html>.

[20] BECK, Kent. Test Driven Development : By Example. Boston, USA : Addison-Wesley Professional, 2002. 240 s. ISBN 9780321146533.

[21] FOWLER, Martin. Refactoring : Improving the Design of Existing Code. Boston, USA : Addison-Wesley Professional, 1999. 464 s. ISBN 0201485672.

[22] ALVIN, Alexander. DevDaily [online]. 2007 [cit. 2011-03-20]. JDBC 101 - Connect to a SQL database with JDBC. Dostupné z WWW: <http://www.devdaily.com/java/edu/pj/pj010024/>.

[23] Java Stories [online]. 2009 [cit. 2011-03-20]. What is JAR hell ?. Dostupné z WWW: <http://tech-read.com/2009/01/13/what-is-jar-hell/>.

[24] The Art of Unix Programming [online]. 2003 [cit. 2011-03-20]. Basics of the Unix Philosophy. Dostupné z WWW: <http://www.faqs.org/docs/artu/ch01s06.html>.

[25] Sun Microsystems, Inc. Oracle Technology Network [online]. 2003 [cit. 2011-03-21]. Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine. Dostupné z WWW: <http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html>.

Seznam příloh

- Příloha A - Obsah přiloženého CD a stručný manuál
- Příloha B - Manuál ke kompilaci aplikace
- Příloha C - Informace o testovacím serveru
- Příloha D - Náhledy obrazovek z aplikace

Příloha A - Obsah přiloženého CD a stručný manuál

Na přiloženém CD jsou k dispozici následující položky:

- *app* - zdrojové kódy aplikace, podrobněji viz příloha B
- *doc* - návrhové diagramy
- virgo-web-server - provozuschopná instalace serveru Virgo
  - Pro nastartování aplikace je nezbytné vytvořit databázi PostgreSQL (viz příloha B)
  - Server se spouští příkazem „bin/startup.sh“ z daného adresáře
  - Po nastartování si lze prohlédnout jeho běh na http://localhost:8080
    - Administrace webového serveru na http://localhost:8080/admin (admin / springsource)
    - Po přihlášení je viditelný seznam aktuálně nahraných modulů (pod Bundles)
    - Jednotlivé moduly si lze rozkliknout a prohlédnout si, které balíčky importují, které služby nabízejí atd.
    - Pod položkou menu *OSGi State* je pak kompletní přehled všech načtených modulů uvnitř serveru (včetně závislostí a systémových)
  - Ukázková instance aplikace je viditelná na http://localhost:8080/instance
  - Druhá instance aplikace je viditelná na http://localhost:8080/instance2
  - Portálový přehled pak na http://localhost:8080/portal
  - Při procházení detailů článků lze sledovat změny na úvodní stránce portálu (zapisuje se seznam naposledy procházených článků)
  - Do administrace konkrétního blogu se lze dostat odkazem v patičce (admin / admin)
- *deployment* - testovací sestava pro kapitolu 6, verze aplikace není nejnovější (po testování byly provedeny dodatečné kosmetické úpravy)
  - Spustil lze vždy z hlavního adresáře serveru pomocí příkazu *bin/startup.sh* (případně pro Windows existuje varianta *.bat*)
  - *standalone* - 10 webových serverů Tomcat - po spuštění k dispozici na adresách od: http://localhost:8011/instance1
  - *shared* - server Tomcat se sdílenými knihovnami - po spuštění k dispozici na adrese: http://localhost:8022/instance1
  - *osgi* - server Virgo s OSGi moduly - po spuštění k dispozici na adrese: http://localhost:8032/portal

Příloha B - Manuál ke kompilaci aplikace

Aplikace se překládá pomocí nástroje *Gradle*, který nahrazuje *Apache Ant*. Součástí balení je jednoduchý skript, který nainstaluje samotný *Gradle* a ten následně stáhne všechny knihovny nutné pro běh aplikace.

Pro správnou funkci je nezbytné mít nainstalováno prostředí *JDK6* a nastavenu proměnou *JAVA_HOME* a j*ava.exe* spustitelný skrze *PATH*.

Zdrojové soubory se nalézají v adresáři *app* na přiloženém CD. Každý projekt obsahuje i *.project* projektový soubor pro integrované prostředí *Eclipse*. Před samotným otevřením uvnitř *IDE* je nezbytné provést alespoň jedenkrát *build*, aby se stáhly potřebné knihovny (viz dále). Dále je nezbytné nastavit si v Eclipse proměnnou prostředí - GRADLE_CACHE - cestu ke knihovnám:

- *Window → Preferences → Java → Build Path → Classpath Variables*
- *New…*
- *Name = 'GRADLE_CACHE'*
- *Path = cesta ke složce cache (na linuxu: ~/.gradle/cache)*

Sestavení samotné aplikace se provede příkazem gradlew v adresáři app:

cd app

(pro Windows je nejdříve nutné provést cleanEclipse eclipse build v adresáři app/cz.vutbr.fit.xpechd00.storage a až poté v app)

./gradlew cleanEclipse eclipse build

vyčištění projektu lze následně provést

./gradlew clean

Nyní je možno projekty importovat v prostředí Eclipse, kódování zdrojových souborů je UTF-8.

Build je inkrementální, kompilují se tedy vždy výhradně upravené zdrojové soubory. Zároveň se stáhnou veškeré závislosti projektů. Při změně zdrojových kódů rovněž *gradle* vygeneruje distribuční archivy do adresáře *build/libs* u projektu. Tuto rovnou kopíruje do adresáře *virgo-web-server/pickup*, stáhnete-li si celou strukturu CD a spustíte Virgo server, *./gradlew build* po každém spuštění provede aktualizaci OSGi modulů právě běžících v kontejneru.

Pro spuštění aplikace je nezbytné mít zprovoznění PostgreSQL databázi verze 9.0, uživatel „blogcity“, heslo: „blogcityPWD“. Pro změnu připojení do databáze je nutné změnit údaje v souborech *META-INF/spring/osgi-service.xml* (v projektu *storage*) a znovu zkompilovat distribuční archiv. Stačí vytvořit databázi *blogcity*, aplikace sama si dotvoří tabulky a nahraje ukázková data.

Příkaz gradlew lze použít i na systému Windows, pak je nutné přidat koncovku *.bat*. V každém projektu je k dispozici *target ../gradlew jar* nebo *../gradlew war*, který zajistí překompilování projektu a nahrání výsledné knihovny do přiloženého *virgo-web-server*. Soubory se zapisují, tzn. tyto příkazy nelze spustit přímo na CD, je nezbytné si projekt překopírovat na disk.

Pokud nastane problém s pořadím nahrávaných knihoven, je nezbytné všechny knihovny z adresáře *virgo-web-server/pickup* přesunout mimo a poté je postupně přesouvat zpět s několika sekundovým odstupem v tomto pořadí (server musí být zapnut):

- cz.vutbr.fit.xpechd00.blogcity.domain-1.0.0-SNAPSHOT.jar
- cz.vutbr.fit.xpechd00.blogcity.storage-1.0.0-SNAPSHOT.jar

- cz.vutbr.fit.xpechd00.blogcity.statistics-1.0.0-SNAPSHOT.jar
- cz.vutbr.fit.xpechd00.blogcity.portal-1.0.0-SNAPSHOT.jar
- Konkrétní instance
- Další soubory

Dokumentaci k jednotlivým projektům lze nalézt v adresáři *doc* (třídní diagramy) a ve formátu *JavaDoc* ji lze vyexportovat u každého projektu pomocí příkazu *../gradlew doc*. Zapíše se do adresáře *build/docs*.

Webový server *Eclipse Virgo* v prostředí Windows nefunguje správně (chyba přímo ve webserveru), ostatní části (projekt v *Eclipse*, *Apache Tomcat*, *Gradle*) fungují správně. Vzhledem k tomuto se doporučuje testovat projekt pod *Linuxem*, kde vše funguje správně.

Příloha C - Informace o testovacím serveru

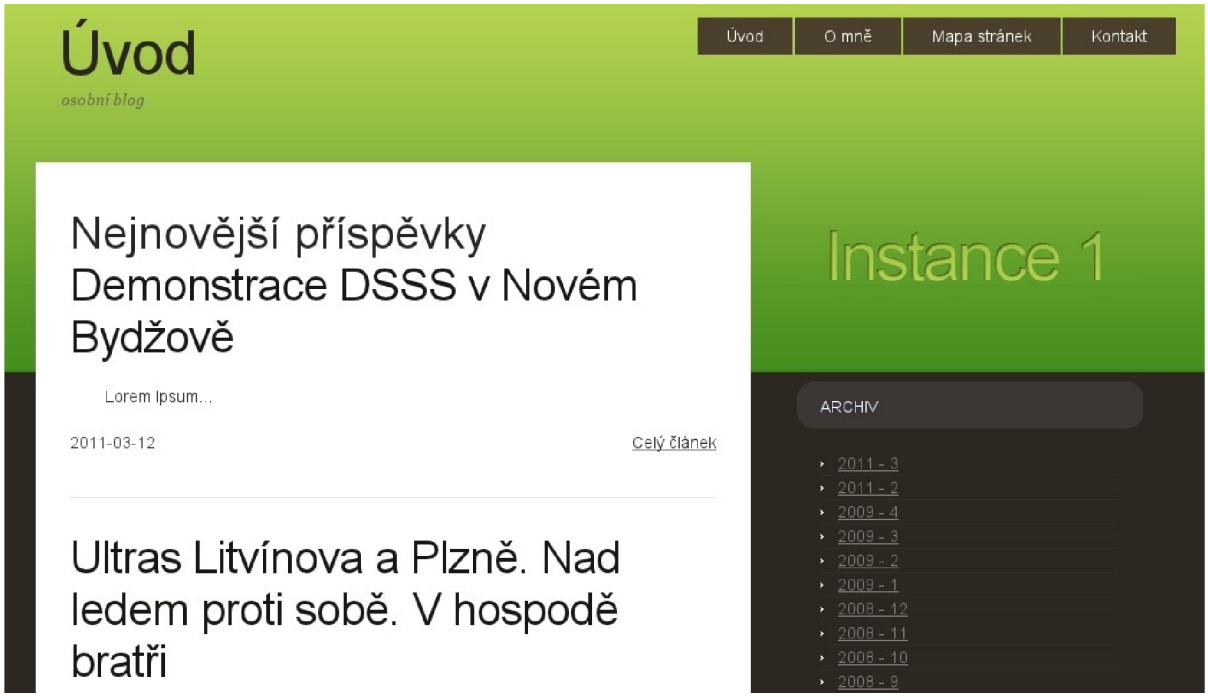
Testy byly vždy provedeny alespoň 5-krát za sobě, výsledky jsou brány jako průměr naměřených hodnot.

Konfigurace serveru je následující:

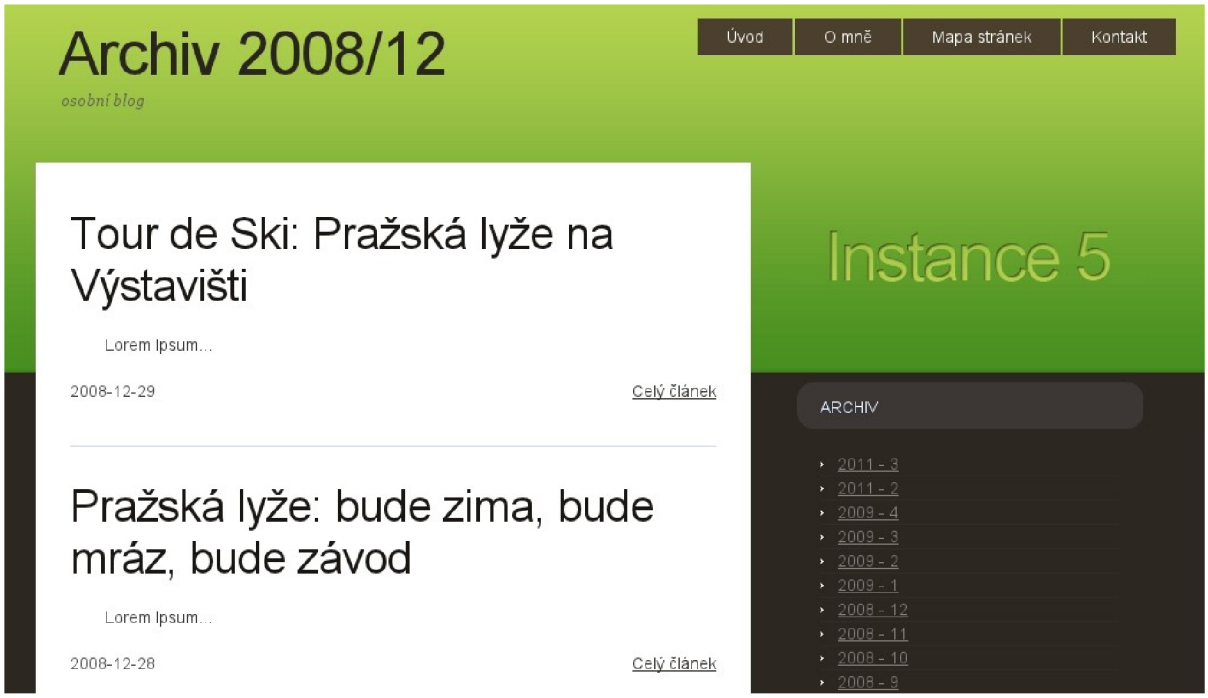
- Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz
- 4 GB RAM
- 80GB, 250GB, 500GB HDD SATAII
- Debian Linux, „Lenny“
- JDK 1.6.0u22
- PostgreSQL 9.0
- Apache Tomcat 6.0.32
- Eclipse Virgo 2.1.0

V průběhu testování na serveru kromě samotné aplikace běželo zcela minimální serverové prostředí, nebyl jakkoliv jinak vytížen (*load* pod 1%).

Příloha D - Náhledy obrazovek z aplikace

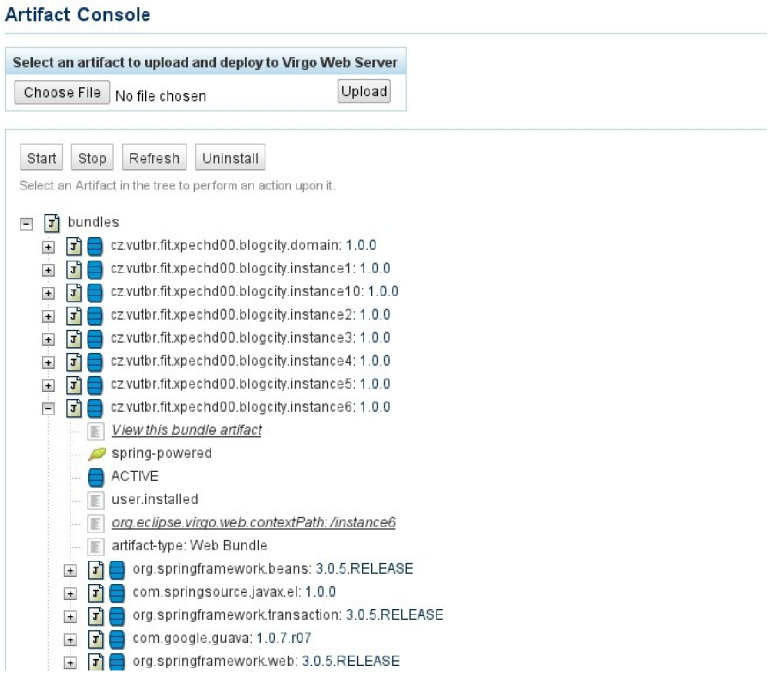


Snímek 1: Hlavní stránka instance blogu



Snímek 2: Archiv instance blogu

Snímek 3: Hlavní stránka portálu



Snímek 4: Ukázka správy modulu v prostředí Eclipse Virgo