Univerzita Pardubice
Dopravní fakulta Jana Pernera

Vývoj vícevrstvé aplikace na Platformě Java EE

Bc. Filip Baláš

Diplomová práce
2012

Univerzita Pardubice
Dopravní fakulta Jana Pernera
Akademický rok: **2011/2012**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Filip Baláš**
Osobní číslo:	**D09776**
Studijní program:	**N3708 Dopravní inženýrství a spoje**
Studijní obor:	**Aplikovaná informatika v dopravě**
Název tématu:	**Vývoj vícevrstvé aplikace na platformě Java EE**
Zadávající katedra:	**Katedra informatiky v dopravě**

Zásady pro vypracování:

Cílem práce je zvládnout problematiku vývoje vícevrstvých aplikací na platformě Java EE a na této platformě vytvořit informační rezervační systém autobusové dopravy. Zvláštní pozornost věnovat návrhovým vzorům, objektově relačnímu mapování řešenému pomocí Hibernate framework a problematice běhu systému v prostředí aplikačního serveru.

Práce bude splňovat následující požadavky:

- Bude provedena analýza požadavků, architektury a dostupných technologií.
- Rezervační systém bude modelován s použitím jazyka UML.
- Data budou uložena v relační databázi.
- Uživatel bude přistupovat k aplikaci pomocí webového prohlížeče.
- Aplikace bude pracovat s konkurenčním přístupem mnoha uživatelů.

Systém bude mít následující funkce:

- Autentizace a autorizace uživatelů.
- Rezervace místenek zákazníkem.
- Správa vozového parku a jízd provozovatelem.

Rozsah grafických prací:

Rozsah pracovní zprávy: **50 normostran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **RUBINGER, A. L., BURKE, B.** *Enterprise JavaBeans 3.1*. **6th Edition. United states of America : O'Reilly Media, Inc., 2010. 738 s. ISBN 978-0-596-15802-6.**

2. **Oracle Corporation.** *Java EE 6 Tutorial*. **USA : Oracle Corporation, 2010. 906 s. PartNo: 821-1841-13.**

3. **BAUER, Ch., KING, G.** *Java Persistence with Hibernate : Revised edition of Hibernate in action*. **USA : Manning Publications Co., 2007. 841 s. ISBN 1-932394-88-5.**

4. **MARCHIONI, F.** *JBoss AS 5 Development : Develop, deploy, and secure Java applications on this robust, open source application server*. **Birmingham : Packt Publishing, 2009. 397 s. ISBN 978-1-847196-82-8.**

Vedoucí diplomové práce: **Ing. Ondřej Rejsek**
Katedra informatiky v dopravě

Datum zadání diplomové práce: **23. listopadu 2011**

Termín odevzdání diplomové práce: **23. května 2012**

prof. Ing. Bohumil Culek, CSc.
děkan

L.S.

doc. Ing. Josef Volek, CSc.
vedoucí katedry

V Pardubicích dne 18. listopadu 2011

Prohlašuji:
Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst.1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích 28.11.2012

Bc. Filip Baláš

Anotace

Tato práce si klade za cíl čtenáře seznámit s problematikou návrhu a vývoje vícevrstvých aplikací. Představuje platformu Java EE a ve zkratce popisuje její klíčové součásti. Důraz je kladen na škálovatelnost, objektově-relační mapování a konzistenci dat. Praktická část práce je tvořena aplikací, jejíž zdrojový kód dokumentuje praktické použití popisovaných technologií.

Klíčová slova

Vícevrstvá architektura, Java EE, JSF, EJB, JPA, JBoss.

Annotation

This thesis aims to familiarize the reader with problems of design and development of multi-tier applications. Presents the Java EE platform and in a nutshell describes its key components. The emphasis is on scalability, object-relational mapping and data consistency. The practical part consists of application, whose source code demonstrates practical use of the described technology.

Keywords

Multitier architecture, Java EE, JSF, EJB, JPA, JBoss.

Obsah

1 Vícevrstvá architektura

Vícevrstvá architektura (anglicky multitier architecture, n-tier architecture) je v softwarovém inženýrství speciálním případem klient-server architektury. Základem této architektury je rozdělení související logiky do samostatných celků. Funkčnost aplikace je tak rozdělena do n samostatných vrstev, které spolu vzájemně komunikují přes definovaná rozhraní.

Ve vícevrstvé architektuře se můžeme setkat s termíny „tier“ a „layer“, které jsou často v tomto kontextu chybně považovány za synonyma. Termín „tier“ se používá pro označení fyzické hardware vrstvy, zatímco termín „layer“ označuje logickou software vrstvu.

Pokud se rozhodneme zvýšit výkon aplikace, můžeme tak učinit vylepšením hardware daného tier. Tento přístup se nazývá horizontální. Druhou možností je zařazením dalšího logického celku (layer) na dedikovaný fyzický stroj (tier), což je přístup vertikální. V praxi se můžeme setkat i s kombinací obou výše uvedených přístupů, takový přístup se pak nazývá diagonální.

Podle způsobu komunikace mezi vrstvami rozlišujeme dva obecné typy rozhraní. „Ukecané“ rozhraní (chatty interface) komunikuje často v krátkých intervalech a pracuje s malými objemy dat. Naproti tomu „tlusté“ rozhraní (chunky interface) komunikuje v delších časových intervalech a pracuje s většími objemy dat.

Jednou z největších výhod vícevrstvé architektury je, že kterákoliv vrstva může být vyměněna nebo upravena, aniž by to mělo vliv na chod aplikace jako celku. Toto je umožněno důsledným oddělením jednotlivých vrstev a volnými vazbami mezi nimi. Další výhodou je možnost rozdělit vývoj různých vrstev mezi různé vývojáře nebo týmy. V neposlední řadě může být stejná aplikační, datová nebo prezentační logika použita v jiné aplikaci.

Nejvýznamnější nevýhodou vícevrstvých aplikací je pravděpodobně vyšší režie při správě většího množství serverů (tiers). Další nevýhodou je, že při změně nebo výměně vrstvy může dojít ke změně jejího veřejného rozhraní. Změna veřejného rozhraní vrstvy způsobí nutnost změny i sousedních vrstev. [1][2]

Aby bylo možné maximalizovat výhody, a minimalizovat nevýhody uvádí [2] tyto zásady vývoje vícevrstvých aplikací:

- Každá vrstva by měla komunikovat vždy jen se sousedními vrstvami.
- Při tvorbě vrstev dodržovat volné vazby (low coupling) aby bylo možné kdykoliv vrstvu upravit nebo vyměnit.
- Zavádět další logické a fyzické vrstvy jen pokud je to opravdu nutné.
- Vrstvy by spolu měly komunikovat co nejméně.

Při návrhu vícevrstvé aplikace je prvním krokem zaměřit se na nejvyšší úroveň abstrakce a rozdělit funkcionality do logických vrstev. Dále je nutné každé vrstvě definovat její veřejné rozhraní. Jakmile jsou rozhraní definována, je potřeba se zamyslet, jakým způsobem bude aplikace nasazena. Posledním krokem je volba protokolů pro komunikaci mezi jednotlivými vrstvami. [3] Doporučuje při návrhu vícevrstvé aplikace postupovat v těchto krocích:

- Určit strategii vytváření vrstev (Layering strategy) – rozdělením aplikace do příliš mnoha, nebo naopak příliš málo vrstev znamená zvýšení složitosti a snížení výkonu, znovupoužitelnosti a flexibility. V tomto ohledu je rozhodnutí o granularitě vrstev kritickým.
- Rozhodnout jaké vrstvy vytvořit – existuje několik způsobů jak spojovat funkcionality do vrstev. Nejběžnějším způsobem je rozlišovat funkcionality na:
    - prezentační
    - business
    - datové

  (viz. Třívrstvá architektura)
  Další skupiny, podle kterých lze rozlišovat funkcionality, jsou například:
    - management
    - report
    - infrastruktura

  Vrstva je chybně navržena pokud nepředstavuje logické seskupení funkcionalit, které poskytuje.
- Rozhodnout o distribuci vrstev – jakým způsobem budou logické layers umístěny na fyzické tiers. Vrstva by měla být umístěna na samostatný server, pouze pokud je to nezbytné. Nejběžnější důvody umístění vrstvy na samostatný server jsou:
    - škálovatelnost
    - hardwarová omezení
    - bezpečnostní politika
- Navrhnout pravidla interakce vrstev – hlavní důvody proč specifikovat pravidla pro interakci vrstev jsou minimalizace závislostí a eliminace cyklických referencí. Cyklická reference vznikne například, pokud každá z dvou vrstev obsahuje komponenty, které mají referenci na druhou vrstvu. Běžný postup jak se tomuto problému vyhnout je povolení interakce pouze jedním směrem, použitím jednoho z následujících přístupů:

    - Interakce shora dolů – vrstvy v modelu výše položené mohou pracovat s vrstvami pod nimi. Výše položené vrstvy jsou informovány o změnách ve vrstvách níže pomocí událostí.
    - Přísná interakce – každá vrstva může pracovat pouze s vrstvou, která je v modelu přímo pod ní. Výhodou tohoto přístupu je, že pokud zaneseme do aplikace změnu, která si vynutí změnu veřejného rozhraní vrstvy, musí na tuto změnu reagovat pouze vrstva, která se nachází přímo pod měněnou vrstvou. Z tohoto důvodu je tento postup doporučován tam, kde se předpokládá postupné rozšiřování aplikace o další funkcionality a aplikace bude nasazena na různé fyzické vrstvy.
    - Volná interakce – v modelu výše postavená vrstva může pracovat s kteroukoliv vrstvou, která je položena pod ní. Přeskočením vrstev lze zvýšit výkon aplikace, ale vznikne více závislostí. Tento přístup je doporučován zejména tam, kde nebude aplikace nasazena do několika fyzických vrstev nebo tam, kde změny kódu lze provádět s minimálním úsilím.
- Identifikovat funkcionality napříč aplikací – tzv. „Cross-cutting concerns" jsou funkcionality, které překračují hranice jedné vrstvy. Patří sem například:
    - autentizace a autorizace
    - logování
    - exception management
    - caching
    - validace

  Tyto funkcionality je nutné v návrhu aplikace identifikovat a navrhnout pro ně vlastní komponenty, jejichž kód bude oddělen od kódu ostatních komponent jednotlivých vrstev. Tímto postupem se zvýší znovupoužitelnost a udržovatelnost aplikace. Hlavní přístupy k implementaci funkcionalit napříč aplikací jsou použití společných knihoven a aspektově orientované programování.
- Definovat rozhraní mezi vrstvami – při definici rozhraní vrstvy je nejdůležitější zajistit volné vazby. To znamená, že vrstva nebude odkrývat detaily vnitřní implementace, na které by mohly v jiných vrstvách vzniknout vazby. Místo toho poskytne vrstva veřejné rozhraní, které tyto detaily skryje. Takové skrývání se také nazývá abstrakce, a lze ji dosáhnout těmito přístupy:
    - Abstraktní rozhraní – abstraktní třída, nebo rozhraní, které používají všichni konzumenti vrstvy.
    - Společný návrhový typ – rozhraní, které je společné více vrstvám v aplikaci.

    - Inverze závislostí (Dependency inversion) – abstraktní rozhraní jsou definována nezávisle na vrstvách. Místo toho aby byla jedna vrstva závislá na druhé, jsou obě vrstvy závislé na společném rozhraní. Implementaci inverze závislostí řeší návrhový vzor injekce závislostí (Dependency injection). Kontejner definuje jakým způsobem najít objekty, na kterých může být jiný objekt závislý a v případě potřeby tyto objekty vytvoří a automaticky "injektne". Tento přístup poskytuje vysokou míru flexibility, protože závislosti vznikají konfigurací a ne v kódu.
    - Zasílání zpráv – namísto přímého volání metod komponent jiné vrstvy lze implementovat interakci mezi vrstvami pomocí zasílání zpráv. Platformy pro vývoj vícevrstvých aplikací ve většině případů pro tento účel obsahují vlastní řešení.
- Zvolit způsob nasazení – při volbě způsobu nasazení vícevrstvé aplikace je nutné vzít v úvahu hlavně:
    - možnosti cílového fyzického prostředí
    - architektonické a návrhové omezení plynoucí z cílového prostředí
    - bezpečností a výkonnostní dopady cílového fyzického prostředí

  Strategie nasazení lze rozdělit na dva základní druhy. Nedistribuované nasazení je vhodné například do podnikových intranetů, kde je množina uživatelů omezená. Naopak distribuované nasazení je vhodné u komplexních aplikací, u kterých je důležitá optimalizace škálovatelnosti a udržovatelnosti.
- Zvolit komunikační protokoly – komunikační protokoly používané pro komunikaci mezi vrstvami hrají velkou roli ve výkonu, bezpečnosti a spolehlivosti celé aplikace. Pokud jsou komunikující vrstvy umístěny na stejné fyzické tier, mohou spolu většinou komunikovat přímo. Pokud jsou ovšem umístěny na různých fyzických vrstvách, je nutné vzít v úvahu, jak spolu budou efektivně a spolehlivě komunikovat.

1.1 Třívrstvá architektura

Zřejmě nejběžnějším případem vícevrstvé architektury je architektura třívrstvá, kterou často využívají webové aplikace. Tuto architekturu navrhl John J. Donovan a někdy bývá označena i jako návrhový vzor. Třívrstvá architektura se skládá z prezentační vrstvy, aplikační vrstvy a datové vrstvy.

Na první pohled může být třívrstvá architektura podobná s MVC návrhovým vzorem, nicméně topologicky jsou tyto koncepty odlišné. Základním pravidlem třívrstvé architektury je, že klientská vrstva přímo nikdy nekomunikuje s datovou vrstvou. Veškerou komunikaci těchto vrstev zprostředkovává aplikační vrstva. Z tohoto pohledu je třívrstvá architektura lineární. Naproti tomu MVC je trojúhelníková, protože View posílá zprávy na Controller, Controller posílá zprávy na Model a Model následně přímo zasílá zprávy na View.

Třívrstvá architektura se začala objevovat v 90. letech. Jednotlivé vrstvy běžely na fyzicky oddělených platformách a komunikovaly spolu pomocí počítačové sítě. MVC koncept se objevil o dekádu dříve v samostatných grafických stanicích.

Ve vícevrstvé architektuře je MVC návrhový vzor často implementován tak, že prezentační vrstva obsahuje View a Controller, model se nachází na aplikační vrstvě.

1.1.1 Prezentační vrstva

Prezentační vrstva obsahuje komponenty, které kontrolují zobrazení grafických prvků uživateli a zároveň se stará o zpracování interakce těchto prvků s uživatelem. Grafické komponenty spolu dohromady tvoří Graphical User Interface (GUI). Někdy je tato vrstva rozdělena na dvě samostatné vrstvy.

Klientská vrstva běží výhradně na zařízení uživatele a stará se pouze o zobrazení GUI. Komponenty této vrstvy mají za úkol uživateli prezentovat informace nebo naopak informace od uživatele získat.

Vrstva prezentační logiky však může běžet na vlastním zařízení, které je s klientem spojeno. Prezentační logika reaguje na události vyvolané uživatelem, tyto události v podobě zpráv posílá klientská vrstva. Obsahuje kód, který definuje logickou strukturu a chování uživatelského rozhraní a navíc je nezávislý na konkrétní specifikaci prvků klientské vrstvy.

S implementací prezentační vrstvy souvisí výběr typu klienta aplikace. Některé vícevrstvé aplikace podporují přístup více druhů klientů zároveň. Základní druhy klientů jsou: [4]

- tenký klient (thin client)
- tlustý klient (thick client)
- chytrý klient (smart client)

Tenkým klientem je obvykle prohlížeč webových stránek. Komunikace s prezentační vrstvou probíhá přes HTTP protokol. Klient neobsahuje žádnou prezentační logiku, kromě jednoduché validace uživatelského vstupu. Použití tenkého klienta přináší tyto výhody: [4]

- přenositelnost a nezávislost na platformě
- klient nevyžaduje výkonný hardware
- nulové náklady na instalaci a údržbu aplikace na straně uživatele

Mezi nejvýznamnější nevýhody tenkého klienta patří:

- server vyžaduje výkonný hardware
- náročnější vývoj
- strohé uživatelské rozhraní
- pomalá odezva uživatelského rozhraní

Tlustý klient v sobě může obsahovat jak prezentační, tak aplikační logiku a může se připojovat přímo k databázovému serveru. Obvykle pracuje s větším objemem dat, která po vyhodnocení přenese zpět na server. Mezi výhody tlustého klienta patří: [4]

- server nevyžaduje výkonný hardware
- snadnější vývoj
- bohaté uživatelské rozhraní
- rychlá odezva GUI

Nevýhody tlustého klienta jsou:

- klient může vyžadovat výkonný hardware
- vyšší nároky na šířku pásma počítačové sítě
- náklady s pojené s nasazením a údržbou na straně uživatele

Chytrý klient se pokouší kombinovat výhody a potlačit nevýhody tenkého a tlustého klienta. Přičemž za klíčovou výhodu tenkého klienta považuje nulové náklady na instalaci, údržbu a update a klíčovou výhodu tlustého klienta bohaté a rychlé GUI. Hlavní platformy pro vývoj chytrých klientů jsou: [5]

- Flex
- Java FX
- Silverlight
- DataSnap

Kromě výběru typu klienta a volby technologie GUI je nutné při návrhu prezentační vrstvy vzít v úvahu následující problémy: [4]

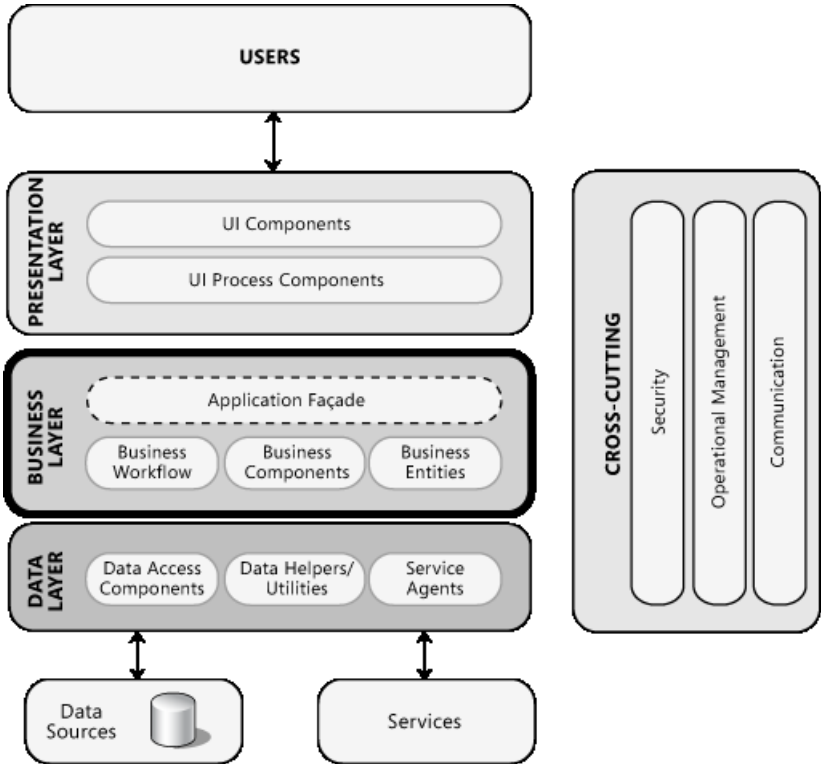
- Caching – jedna z nejúčinějších technik, jak zvýšit výkon aplikace a odezvu GUI. Prezentační vrstva je obohacena o paměť, do které se ukládají výsledky nákladných nebo často se opakujících operací. Pokud se požadovaná data nacházejí v cache, není nutné zasílat požadavek na aplikační vrstvu.
- Komunikace – při použití synchronního vzdáleného volání musí uživatel čekat na dokončení požadavku, během tohoto času GUI nereaguje na jeho akce. Pokud vzdálená volání trvají delší dobu, je vhodné zvážit použití asynchronního volání, kdy uživatel může pokračovat v práci a o dokončení požadavku je vyrozuměn. Dále je vhodné uživatele informovat o průběhu dlouho trvajícího požadavku, případně mu umožnit jeho stornování.
- Kompozice – aby bylo možné kód efektivně udržovat, je nutné minimalizovat závislosti mezi komponenty. Mezi základní techniky jak toho dosáhnout patří použití abstrakce, implementace proti rozhraní, run-time Dependency injection. Při komunikaci mezi komponenty lze pro minimalizaci závislostí použít návrhový vzor Publish/Subscribe.
- Zpracování výjimek – v aplikaci je vhodné navrhnout centralizovaný systém pro zachycení a zpracování výjimek. Je nutné rozlišovat systémové a business výjimky. Pokud je aplikace po vzniku výjimky v nekonzistentním stavu, musí být ukončena. Při zobrazení výjimky uživateli je nutné dbát na citlivost dat. Prezentační vrstva zobrazuje uživateli i výjimky, které vznikly a nebyly zachyceny na jiných vrstvách.
- Navigace – v aplikacích které obsahují velké množství stránek, formulářů, dialogů nebo oken je nutné navrhnout jednotnou strategii navigace, která zvýší uživatelský komfort a pomůže zakrýt komplexitu aplikace. Navigační logiku je vhodné separovat od logiky uživatelského prostředí. Uživatelský komfort lze dále zvýšit použitím menu, nástrojových lišt a průvodců.
- Uživatelský komfort – z hlediska použitelnosti aplikace se jedná o klíčovou vlastnost. Vnímaný výkon aplikace je pro uživatele důležitější než výkon skutečný. Uživateli například nevadí čekat několik sekund na dokončení operace, pokud během této doby dostává informace o průběhu nebo pravděpodobném okamžiku dokončení. Naopak uživateli může vadit čekat třeba i zlomek sekundy pokud nedostane informaci o průběhu operace a uživatelské rozhraní je neresponzivní. Negativní dopad na uživatelský komfort mají také příliš komplexní a přeplněné formuláře. Design formuláře či okna by měl být navržen s důrazem na hlavní scénář implementované use case.
- Uživatelské rozhraní – při návrhu uživatelského rozhraní je třeba dodržovat jednotný vzhled definovaný organizací, která aplikaci vyvíjí a také obecná pravidla návrhu UI. Důležité je také rozhodnutí, jestli bude UI navrhovat

specializovaná skupina designerů, nebo programátoři. Textové řetězce není vhodné vkládat přímo do zdrojového kódu, ale umísťovat je do externích zdrojů, což v budoucnu umožní snadnější změnu lokálních a globálních nastavení.

- Validace dat – efektivní strategie validace vstupních dat je důležitou součástí bezpečnosti a funkčnosti aplikace. Validaci je dobré rozdělit na dvě části. Vstupní validaci, kterou provádí prezentační vrstva a business validaci, kterou provádí aplikační vrstva. O neúspěchu validace je potřeba uživatele srozumitelně informovat a například zvýraznit prvek nebo prvky, které obsahují neplatná data.

1.1.2 Aplikační vrstva

Anglicky je tato vrstva označována termínem business, domain nebo application layer. Do aplikační vrstvy se agregují funkcionality týkající se výpočtu a vyhodnocení.



Obrázek 1: *Schéma vícevrstvé aplikace.* Aplikační vrstva je tučně orámovaná. Zdroj:[6].

Aplikační vrstva obvykle obsahuje tyto komponenty: [6]

- Aplikační fasáda – komponenta, která implementuje návrhový vzor fasáda. Poskytuje zjednodušené rozhraní k business logice a skrývá implementační detaily. Typicky se jedná o rozhraní vyšší úrovně k sadě rozhraní business logiky.

- Komponenty business logiky – jako business logiku lze označit veškerou logiku, která se týká:
    - získávání, zpracování, transformování a řízení aplikačních dat
    - aplikace business pravidel a zásad
    - zajištění konzistence a validity dat

  Komponenty aplikační logiky lze dále rozdělit na:
    - Business workflow – hodně business procesů zahrnuje několik kroků, které musí být vykonány ve správném pořadí a vzájemně spolu komunikují. Business workflow komponenty definují a koordinují takové procesy.
    - Business entity – nebo obecněji business objekty zapouzdřují logiku a data nezbytná k reprezentací objektů reálného světa. Zajišťují konzistenci dat, která obsahují a provádí jejich validaci.

Při návrhu aplikační vrstvy je cílem minimalizace složitosti rozdělením souvisejících úloh do skupin. Komponenty v rámci takové skupiny by se měly soustředit pouze na tuto skupinu a neměly by obsahovat kód související s jinými skupinami.

Softwarový architekt, který navrhuje aplikační vrstvu, by se měl v první řadě zamyslet nad tím, jestli takovou vrstvu vůbec potřebuje. Samostatná aplikační vrstva je z hlediska výkonu a udržovatelnosti aplikace téměř vždy výhodou, výjimku tvoří aplikace, které mají málo nebo žádná business pravidla. To jsou aplikace, které pouze načítají nebo ukládají data, případně provádí jejich validaci.

Dále je nutné identifikovat konzumenty a odpovědnosti vrstvy. V tomto kroku je potřeba si uvědomit jaké úlohy bude vrstva plnit. Pokud je konzumentem prezentační vrstva nebo externí aplikace lze aplikační vrstvu prezentovat jako službu. Dobrý návrh aplikační vrstvy by neměl obsahovat žádné komponenty obsahující prezentační logiku nebo logiku přístupu k datům.

Pokud bude aplikační vrstva umístěna na samostatném tier, je potřeba minimalizovat komunikaci s ostatními vrstvami či klienty aby nedocházelo k přetížení sítě. Toho lze dosáhnout například implementací asynchronního volání založeného na zasílání zpráv nebo zařazením vrstvy, která bude transformovat velké množství „fine-grained“ operací na menší množství „coarse-grained“ operací.

Stejně jako při návrhu jakékoliv jiné vrstvy je vhodné minimalizovat závislosti na okolní vrstvy. Techniky jak tohoto dosáhnout byly již popsány v předchozí kapitole. [6]

Problémy typické pro návrh aplikační vrstvy jsou: [6]

- Autentizace – návrh efektivní autentizace uživatelů na aplikační vrstvě je důležitý z hlediska bezpečnosti a spolehlivosti aplikace jako celku. Uživatele je

většinou nutné autentizovat pokud je služba aplikační vrstvy volána vzdáleně. Pokud je však služba aplikační vrstvy volána jinou vrstvou na stejném tier, není uživatele nutné autentizovat, protože to již provedla volající vrstva a zároveň potencionální útočník neměl možnost napadnou komunikaci těchto vrstev. Platformy pro vývoj vícevrstvých aplikací většinou obsahují hotový mechanizmus autentizace.

- Autorizace – stejně jako autentizace je důležitým prvkem bezpečnosti a spolehlivosti aplikace. Autorizace chrání citlivé zdroje ověřováním volajícího na základě jeho identifikace, role, či jiné související informace. Odborná literatura doporučuje minimalizovat granularitu rolí, aby se předcházelo vzniku složitých kombinací požadovaných oprávnění. Business logika by neměla obsahovat autorizační logiku, ta by měla být v samostatných komponentách. Jelikož je možné, že autorizace bude probíhat nejen na aplikační vrstvě, ale napříč celou aplikací, je důležité, aby byla efektivní a nezpůsobovala snížení výkonu aplikace.
- Caching – stejně jako v prezentační nebo jakékoliv jiné vrstvě je dobrý návrh kešovací strategie způsob jak zvýšit výkon a odezvu aplikace. Správným kešováním se lze vyhnout opakovaným vzdáleným voláním nebo opětovnému zpracování stejných dat. Vhodným kandidátem na kešování jsou statická data, naopak data nestála, která se často mění, není vhodné kešovat. Data je vhodné ukládat v takovém formátu, aby bylo možné je bez jakékoliv transformace na aplikační vrstvě použít. Není doporučeno kešovat citlivá data, v takovém případě je pak nutné navrhnout mechanizmus, jak tyto data zabezpečit.
- Závislosti a soudržnost – podobně jako dobrý objektový návrh třídy, měl by i dobrý návrh každé vrstvy minimalizovat závislostí na okolí a maximalizovat svoji soudržnost. Důležité je vyhnout se kruhovým závislostem. Aplikační vrstva by měla být ideálně závislá pouze na vrstvě, která se nachází v hierarchii vrstev přímo pod ní. V třívrstvé architektuře se jedná o datovou vrstvu, ve vícevrstvé architektuře se většinou jedná o integrační vrstvu. Ke snížení závislostí slouží techniky abstrakce, které již byly popsány dříve. Vysoká soudržnost vznikne dobrým návrhem aplikačních komponent, které jsou úzce specializované pro konkrétní činnost. Odborná literatura klade důraz na důsledné oddělení business logiky a logiky přístupu k datům (data access).
- Exception management – stejně jako autentizace nebo autorizace má velký vliv na bezpečnost a spolehlivost aplikace. Špatný návrh systému zpracování výjimek může zapříčinit citlivost aplikace vůči útokům a odkrýt citlivá data. Vyvolávání a zachytávání výjimek je nákladnou operací, proto je nutné

navrhnout systém s ohledem na výkon aplikace. Při návrhu systému výjimek uvádí [6] tyto zásady:

- zachytávat pouze výjimky, které lze v daném místě ošetřit, případně přidat informace a výjimku poslat dále
- nepoužívat výjimky k řízení chodu business procesů
- ujistit se, že nejsou zachytávány výjimky, které by měly být zachyceny jinde, např v globálním error handleru
- po zachycení a ošetření výjimky je nutné uvolnit zdroje a zanechat proces v konzistentním stavu
- kritické chyby by měly být s dostatečným množstvím informací zalogovány a uloženy

Systém zpracování výjimek je jedním z Cross-cutting concerns. To znamená, že se netýká pouze aplikační vrstvy a bude použit napříč celou aplikací.

- Logování – aplikační vrstva by měla obsahovat centralizovaný systém logování. Soubory s logy mohou být cenné při hledání chyby v aplikaci nebo můžou sloužit jako podklady pro právní kroky v případě vážného pochybení uživatele. Všechny požadavky na služby aplikační vrstvy by měly být zaznamenány v logu. V logu by se však neměla vyskytovat citlivá business data. Chyba vzniklá při logování nesmí ovlivnit normální funkcionalitu aplikační vrstvy.
- Validace – validace na úrovni aplikační vrstvy je důležitá zejména pro konzistenci dat a dodržení business pravidel. Zároveň je validace na aplikační vrstvě důležitým prvkem bezpečnosti, který chrání aplikaci před útoky. Veškeré vstupy aplikační vrstvy je nutné validovat, a to i pokud již byly validovány (např. na prezentační vrstvě). Validaci je vhodné centralizovat, což vede k větší použitelnosti a udržovatelnosti. Každý uživatelský vstup by měl být považován za škodlivý, dokud není validován.
- Nasazení – při návrhu aplikační vrstvy je nutné vzít v úvahu možnosti cílového produkčního prostředí, zejména v oblasti výkonu a bezpečnosti.

Při návrhu nové aplikační vrstvy doporučuje [6] postupovat v následujících krocích:

- Identifikovat konzumenty vrstvy, to mohou být jiné vrstvy, služby nebo externí aplikace. Na základě těchto informací navrhnout rozhraní.
- Určit požadavky na bezpečnost a validaci, navrhnout strategii validace, autentizace a autorizace.
- Navrhnout business komponenty.
- Navrhnout business entity komponenty.
- Navrhnout business workflow komponenty.

1.1.3 Datová vrstva

V souvislosti s datovou vrstvou se můžeme setkat také s termíny enterprise a persistence tier. Jedná se o nejspodnější vrstvu vícevrstvého aplikačního modelu. Obsahuje data uložená v relační nebo objektové databázi, případně souboru.

Datová vrstva musí požadavky plnit efektivně a spolehlivě. Musí být snadno udržovatelná a rozšiřitelná aby bylo možné snadno a rychle reagovat na změnu business pravidel.

V tomto ohledu je důležitým rozhodnutím volba integrační technologie, kterým se bude věnovat následující kapitola. Dalším důležitým prvkem dobrého návrhu datové vrstvy je zapouzdření. Datová vrstva by měla před konzument skrývat logiku týkající se například generování dotazů, udržování a řízení databázového spojení a mapování dat z business entit do databázových datových struktur.

Velmi důležité je také rozhodnutí, jakým způsobem bude právě jmenované mapování business entit a databázových struktur probíhat. Běžným přístupem je implementace návrhového vzoru Domain Model nebo Table Module. Další možností je využití ORM (Object Relation Mapping) frameworku, který poskytuje tuto funkčnost již hotovou, efektivní a vyzkoušenou. Výběr tohoto frameworku je závislý na výběru vývojové platformy.

Další otázkou, kterou je potřeba při navrhování datové vrstvy řešit je v jaké datové struktuře budou data přenášena mezi vrstvami. Řešením může být například implementace návrhového vzoru DTO (Data Transfer Object), který pomáhá organizovat data do unifikovaných datových struktur a zároveň vynucuje „coarse-grained“ operace při vzdáleném volání vrstev.[7]

Datová vrstva by měla vytvářet a řídit všechna připojení ke všem datovým zdrojům, které aplikace vyžaduje. V tomto ohledu je potřeba navrhnout způsob jakým budou připojení uchovávána a chráněna proti zneužití. Příkladem takové ochrany je zašifrování konfiguračních souborů těchto připojení. Dále je potřeba proti krádeži a poškození chránit i samotná data. Zásada je nepoužívat uživatelský vstup v dynamických SQL dotazech a používat parametrizované dotazy tam, kde je to možné. Bezpečnostní opatření by měla být implementována jak na straně datové vrstvy, tak na straně datového zdroje. [7]

Pozornost je potřeba věnovat také ošetření výjimek. Datová vrstva by měla ošetřovat výjimky týkající se CRUD (Create, Read, Update, Delete) operací, timeoutů a nekonzistence dat.

Návrhář datové vrstvy by se měl také zamyslet nad tím, jaké budou požadavky na výkon a škálovatelnost. Například, pokud navrhuje datovou vrstvu komerční internetové aplikace, je pravděpodobné, že datová vrstva bude „bottleneck“. Pokud je výkon datové

vrstvy pro výkon aplikace kritický, je vhodné použít profilovací nástroje k zjištění nákladných operací a věnovat pozornost jejich optimalizaci. [7]

Shrnutí pro návrh datové vrstvy specifických problémů: [7]

- Dávkování (Batching) – slučováním podobných databázových příkazů do dávek může zvýšit výkon aplikace. S každým samostatným databázovým příkazem je spojený tzv. „overhead“ (využití zdrojů potřebných k vykonání operace). Dávkováním lze snížit overhead, zvýšit odezvu a propustnost, protože databáze kešuje exekuční plány podobných dotazů.
- Správa připojení – základní funkcionalita datové vrstvy. Vytváření a správa připojení k jednotlivých datovým zdrojů využívá cenné zdroje. Z tohoto důvodu je vhodné řídit se následujícími obecnými pokyny:
    - otevřít nové připojení co nejdéle je to možné
    - zavřít připojení co nejdříve je to možné
    - provádět transakce pomocí jediného připojení kdykoliv je to možné
    - implementovat Object Pool návrhový vzor pro správu připojení
- Objektově relační mapování – mezi objektovým a relačním paradigmatem existuje nesoulad. Tento nesoulad lze řešit implementací návrhových vzorů, například Repository nebo pomocí ORM frameworku. Moderní ORM frameworky mohou velmi rapidně snížit množství potřebného kódu. Dále umožňují generování datového modelu na základě doménového a obráceně.
- Dotazy – jsou primární operací pro manipulaci s daty. Optimalizace dotazů vede k větší propustnosti aplikace. Použitím parametrizovaných dotazů lze omezit riziko SQL Injection útoku. Moderní ORM nástroje umožňují sestavovat dotazy z objektového modelu.
- Databázové procedury – v minulosti představovaly zvýšení výkonu oproti dynamicky generovaným dotazům. Moderní databázové servery tento rys však nevykazují.
- Transakce – skupina operací, která je považována za atomickou s cílem splnit požadavek a zajistit integritu dat. Důležitým krokem je identifikovat model konkurence a navrhnout způsob řízení transakcí. Obecně platí, že transakce by měla být co nejkratší. Také je potřeba rozhodnout o úrovni izolace transakce, ta by měla být co nejnižší, avšak taková aby zaručovala konzistenci dat.
- Validace – doporučuje se validovat veškerá data, která vstupují do datové vrstvy. Pozornost je potřeba věnovat null hodnotám a filtrovat nepovolené znaky.
- XML – bývá základem interoperability aplikací. Pokud je z nějakého důvodu třeba udržovat nějaké objemy dat mimo databázi, XML je vhodný formát.

Pro práci s opravdu velkými objemy dat se doporučuje,z důvodu výkonu, používat schéma, kde jsou data uložena v podobě atributů namísto elementů.

- Výkon – aby bylo možné maximalizovat propustnost datové vrstvy, je doporučováno dodržet následující obecná pravidla:
    - poolovat databázová připojení
    - řídit se výsledky simulací zátěže
    - snížit úroveň izolace ke zvýšení datové propustnosti
    - zvýšit úroveň izolace k zajištění konzistence dat
    - seskupovat dotazy do dávek (batching)
    - optimistický přístup k zamykání dat s nízkou volatilitou
- Bezpečnost – datová vrstva musí chránit databázi před neoprávněným přístupem a v případě oprávněného přístupu poskytovat pouze nezbytně nutnou část datového zdroje. Zároveň musí chránit i mechanizmus k získání připojení k datovému zdroji.
- Nasazení – stejně jako při nasazení jakékoliv jiné vrstvy do cílového prostředí, musí softwarový architekt vzít v úvahu možnosti a omezení plynoucí z tohoto prostředí, zejména v oblasti bezpečnosti a výkonu. Při nasazování datové vrstvy je doporučeno:
    - nasadit datovou vrstvu na stejný tier jako business vrstvu, tam kde to je z hlediska bezpečnosti a škálovatelnosti možné
    - pokud je nutné nasadit datovou vrstvu na vzdálený stroj, využít TCP protokol pro komunikaci
    - pokud je to možné nenasazovat datovou vrstvu na stroj, který obsahuje zdroj dat (databázový server)

Návrhář datové vrstvy by měl při návrhu postupovat v tomto pořadí: [7]

- Navrhnout celkový design datové vrstvy na základě modelu, který je k dispozici. Tím může být buď doménový model, nebo datový model.
- Navrhnout business entity objekty a případně DTO objekty.
- Zvolit integrační technologii.
- Navrhnout komponenty pro přístup k datům.

1.2 Další vrstvy

Tato kapitola obsahuje stručný popis vrstev, o kterých třívrstvá architektura zdánlivě nehovoří. Nicméně funkcionalita těchto vrstev je v třívrstvém modelu obsažena v rámci některé z vrstev.

1.2.1 Klientská vrstva

Jak již bylo řečeno v kapitole věnované prezentační vrstvě, klientská vrstva může být od prezentační oddělena a nasazena samostatně na stroji uživatele. Klientská vrstva komunikuje s vrstvou prezentační, pokud prezentační vrstva implementuje MVC návrhový vzor, pak klientská vrstva obsahuje View a Controller. [1]

1.2.2 Integrační vrstva

V třívrstvém modelu je tato vrstva součástí datové vrstvy, jinak se nachází mezi aplikační a datovou vrstvou. Její hlavní funkcí je odstranění závislosti business vrstvy na datové vrstvě. To umožní například změnit druh databázového serveru bez nutnosti měnit aplikační vrstvu. Využitím některého z ORM frameworku je většinou vytvořena integrační vrstva.

1.2.3 HW a OS vrstva

O fyzické hardwarové vrstvě a vrstvě operačního systému, která je jejím konzument, se literatura příliš nezmiňuje. Bez těchto nejspodnějších vrstev však není model vícevrstvé aplikace kompletní. Navíc právě tyto vrstvy představují omezení, hlavně z hlediska výkonu a bezpečnosti, která je nutné mít na paměti při návrhu a nasazení kterékoliv jiné vrstvy. [1]

1.3 Cross-cutting concerns

Většina vícevrstvých aplikací obsahuje společnou funkcionalitu, která přesahuje hranice jednotlivých logických i fyzických vrstev. Tato funkcionalita typicky zahrnuje logování, autentizaci, autorizaci, systém zpracování výjimek, kešování, validaci atd. Nazývá se cross-cutting concerns, protože ovlivňuje celou aplikaci napříč vrstvami. Snahou návrháře aplikace by mělo být takovou funkcionalitu v kódu centralizovat na jednom místě a neopakovat stejný nebo podobný kód v každé vrstvě.

Prvním krokem při návrhu cross-cutting concerns by měla být identifikace požadavků každé vrstvy na společnou funkcionalitu. Tam, kde jsou požadavky stejné nebo podobné potom tuto funkcionalitu abstrahovat do společných komponent. Je nutné zajistit, aby tyto komponenty byly k dispozici na každém stroji, na kterém běží vrstva požadující společnou funkcionalitu.

Při návrhu společné funkcionality může být výhodné implementovat návrhový vzor Dependency injection. To umožní injektovat konkrétní instance komponent poskytujících společnou funkcionalitu do aplikace na základě informací v konfiguračním souboru, což v důsledku vede k možnosti měnit chování těchto komponent bez nutnosti znovu kompilovat a nasazovat celou aplikaci.

Dále je vhodné zvážit použití software třetích stran. Tento bývá vysoce konfigurovatelný a může rapidně redukovat čas potřebný na vývoj aplikace.

Aspektově orientované programování je technika, kterou lze velmi výhodně zakomponovat společnou funkcionalitu do aplikace. Umožňuje volání společné funkcionality na základě sledování toku aplikace, což redukuje coupling a vede k čistšímu kódu, protože aplikační logika nemusí volat společné funkcionality explicitně. [8]

1.4 Atributy kvality

Tato kapitola uvádí a stručně popisuje atributy kvality aplikace. Míra, kterou aplikace naplňuje kombinaci požadovaných atributů kvality indikuje úspěch návrhu a celkovou kvalitu aplikace. Při návrhu aplikace, jejímž cílem je naplnit určitou kombinaci atributů kvality je nezbytné zvážit dopad těchto atributů na ostatní požadavky aplikace, analyzovat důležitost jednotlivých atributů a zvolit kompromisy mezi nimi. [9]

Atributy kvality lze rozdělit do následujících skupin: [9]

- návrhu
- běhu
- systému
- uživatele

Mezi atributy kvality návrhu patří:

- Integrita koncepce – definuje konzistenci a soudržnost celkového návrhu aplikace. Zahrnuje způsob, jakým jsou jednotlivé komponenty navrženy, kvalitu kódu, pojmenování proměnných, funkcí a tříd.
- Udržovatelnost – schopnost aplikace snadno podstoupit změnu. Změna se může týkat komponent, služeb a rozhraní. Aplikace se mění při opravování chyb, změně business pravidel nebo při přidávání nové funkcionality.
- Znovupoužitelnost – lze definovat jako pravděpodobnost, že komponenta bude použita v jiné komponentě nebo scénáři pro přidání nové funkčnosti s minimálním nebo žádným úsilím. Odstraňuje duplicitu kódu, která je velkým zdrojem chyb a redukuje čas potřebný na vývoj aplikace.

Atributy kvality běhu jsou:

- Dostupnost – časový interval, kdy je aplikace funkční a pracuje. Lze vyjádřit jako procento z celkového měřeného časového intervalu. Je ovlivněna chybami, infrastrukturními problémy, útoky a zatížením.
- Interoperabilita – schopnost sytému komunikovat, pracovat a vyměňovat si data s jiným systémem, často provozovaným třetí stranou.
- Ovladatelnost – definuje jak obtížná je pro administrátory správa aplikace. K dispozici by měly být nástroje pro sledování zátěže, debugging a ladění výkonu.
- Výkon – schopnost vykonávat požadované operace v zadaném časovém intervalu. Sleduje se rychlost odpovědi a propustnost aplikace. Propustnost je definována jako počet operací uskutečněných ve sledovaném časovém intervalu. Výkon aplikace přímo ovlivňuje i její škálovatelnost, platí zde přímá úměrnost.

- Spolehlivost – schopnost pracovat s požadováným chováním ve sledovaném časovém intervalu. Je měřena jako pravděpodobnost, že systém v daném časovém intervalu neselže a udrží si požadovanou funkčnost.
- Škálovatelnost – může být definována jako schopnost systému zvládnout nárůst zátěže bez dopadu na výkon systému nebo jako připravenost systému navýšit výkon vertikálním nebo horizontálním způsobem.
- Bezpečnost – schopnost redukovat šanci škodlivých a náhodných akcí, které jsou za hranicí navrhované funkčnosti, číst nebo poškodit informace. Mezi bezpečností a spolehlivostí existuje přímá úměra.

Atributy kvality systému jsou:

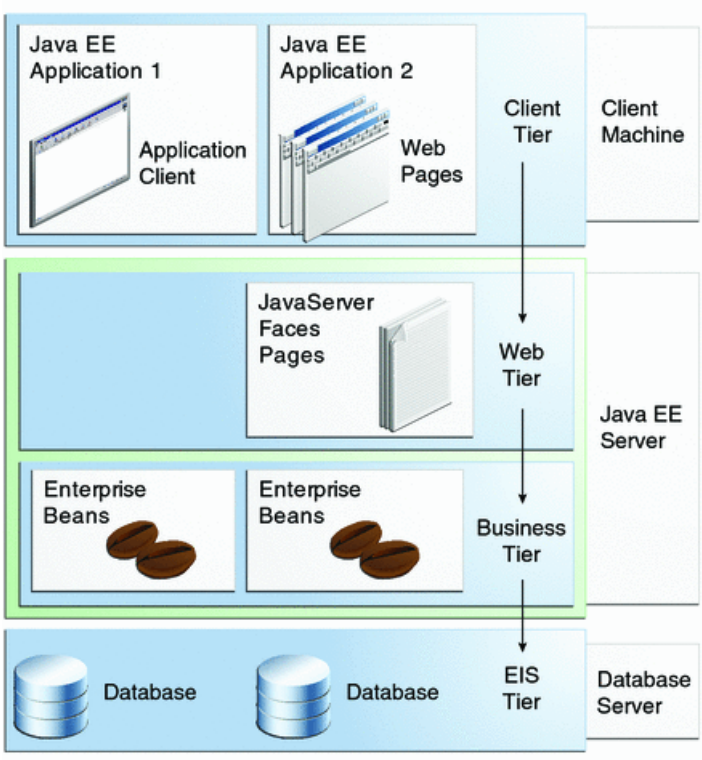
- Supportability – schopnost systému poskytovat informace, které vedou k identifikace a vyřešení problému, pokud systém z nějakého důvodu selže.
- Testovatelnost – udává jak obtížné je systému nebo testované komponentě sestavit kritéria a provádět testy zda byla tato kritéria splněna. Umožňuje efektivně odhalit a izolovat chyby v systému.

Jediným atributem kvality uživatele, který [9] uvádí je uživatelský komfort. Uživatelská rozhraní aplikace musí být navržena s ohledem na uživatele. Musí být intuitivní, lokalizované a globalizované. Návrhář by měl zvážit možnost přidání uživatelského rozhraní pro postižené.

2 Java EE

Java EE (Enterprise Edition) je výpočetní platforma společnosti Oracle. Poskytuje APIs (Application Programming Interface) a běhové prostředí pro vývoj a běh vícevrstvých, škálovatelných, spolehlivých a bezpečných síťových aplikací. Rozšiřuje platformu Java SE (Standard Edition) o odolnost proti chybám, objektově relační mapování, distribuovanou a vícevrstvou architekturu a webové služby. Její návrh je z velké části založen na modulárních komponentách běžících na aplikačním serveru. Software vyvíjený na platformě Java EE je primárně psaný v jazyce Java a konfiguruje se pomocí XML.

Tato platforma je vyvíjená skrz JCP (Java Community Process), který odpovídá za všechny Java technologie. Expertní skupiny složené ze zájmových skupin vytváří JSRs (Java Specification Requests), který definuje různé Java EE technologie. Vývoj v rámci JCP programu pomáhá zajistit standard v oblasti stability a kompatibility. Cílem platformy je poskytnout vývojářům širokou škálu APIs za účelem redukovat složitost a čas potřebný na vývoj aplikací. [10]



Obrázek 2: *Schéma vícevrstvé aplikace vyvinuté na platformě Java EE.* Zdroj: [10].

Aplikační model platformy se zakládá na použití jazyka Java a běhu v prostředí virtuálního stroje Java. Tato kombinace poskytuje prověřenou produktivitu vývojářů, bezpečnost a nezávislost na platformě cílového stroje. Platforma je navržena

pro podporu a vývoj podnikového software se kterým pracují zákazníci, zaměstnanci, dodavatelé atd. Takové aplikace bývají často složité a přistupují k datům z různých zdrojů. Aby bylo možné chod takových aplikací řídit a kontrolovat, jsou funkcionality uživatelů prováděny v aplikační vrstvě, kde nad nimi má informační oddělení větší kontrolu. Aplikační model platformy rozděluje činnost nutnou k implementování vícevrstvého modelu na dvě část:

- business a prezentační logika, kterou implementuje vývojář
- standardní systémové služby, které poskytuje platforma

Ačkoliv může mít aplikace vyvinutá na platformě Java EE libovolný počet vrstev, je aplikační model platformy považován za třívrstvý, protože komponenty jsou rozmístěny do tří typů fyzických lokací:

- komponenty klientské vrstvy běží na zařízení klienta
- webové komponenty běží Java EE serveru
- business komponenty běžící na Java EE serveru
- EIS (Enterprise Information System) běží na EIS serveru

Tato třívrstvá architektura rozšiřuje klasický client-server model přidáním vícevláknového aplikačního serveru mezi klientskou aplikaci a úložiště dat.

Java EE definuje komponentu jako samostatnou softwarovou jednotku vloženou do aplikace, společně se souvisejícími třídami a soubory, která komunikuje s ostatními komponentami. Specifikace definuje následující druhy komponent:

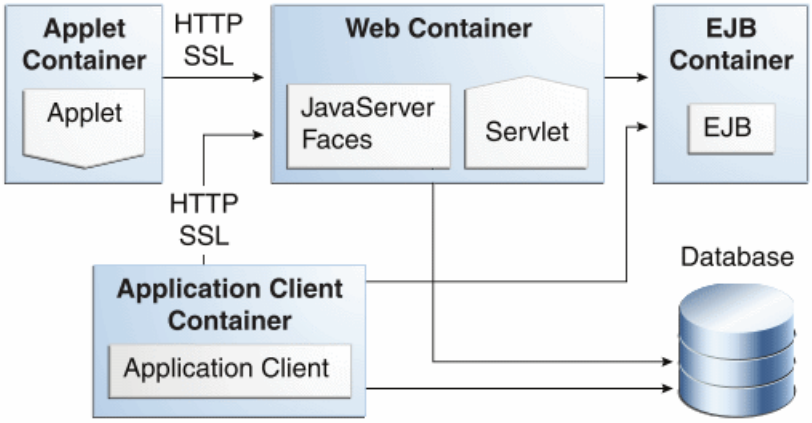
- aplikační klienti a applety
- servlety, a komponenty technologií JSF, JSP
- EJB (Enterprise Java Beans)

Rozdíl mezi komponentou a standardní Java třídou je, že komponenta je dobře navržená, ověřená a v souladu se specifikací, její běh je řízen Java EE aplikačním serverem. [10]

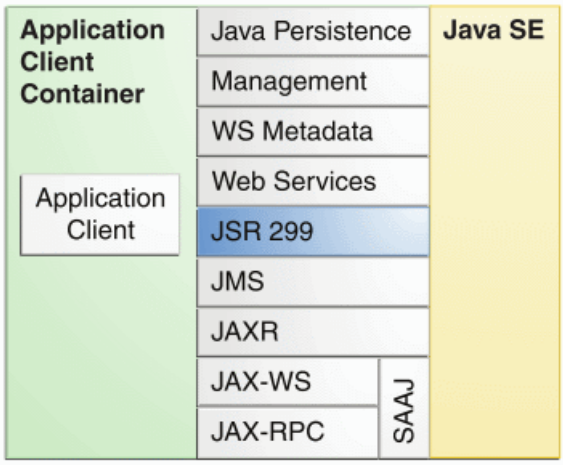
2.1 APIs

Termínem API se označuje rozhraní pro programování aplikací. Jedná se o sbírku procedur, funkcí a tříd, které má programátor k dispozici. API dále určuje jakým způsobem jsou tyto funkce volány ze zdrojového kódu programu.

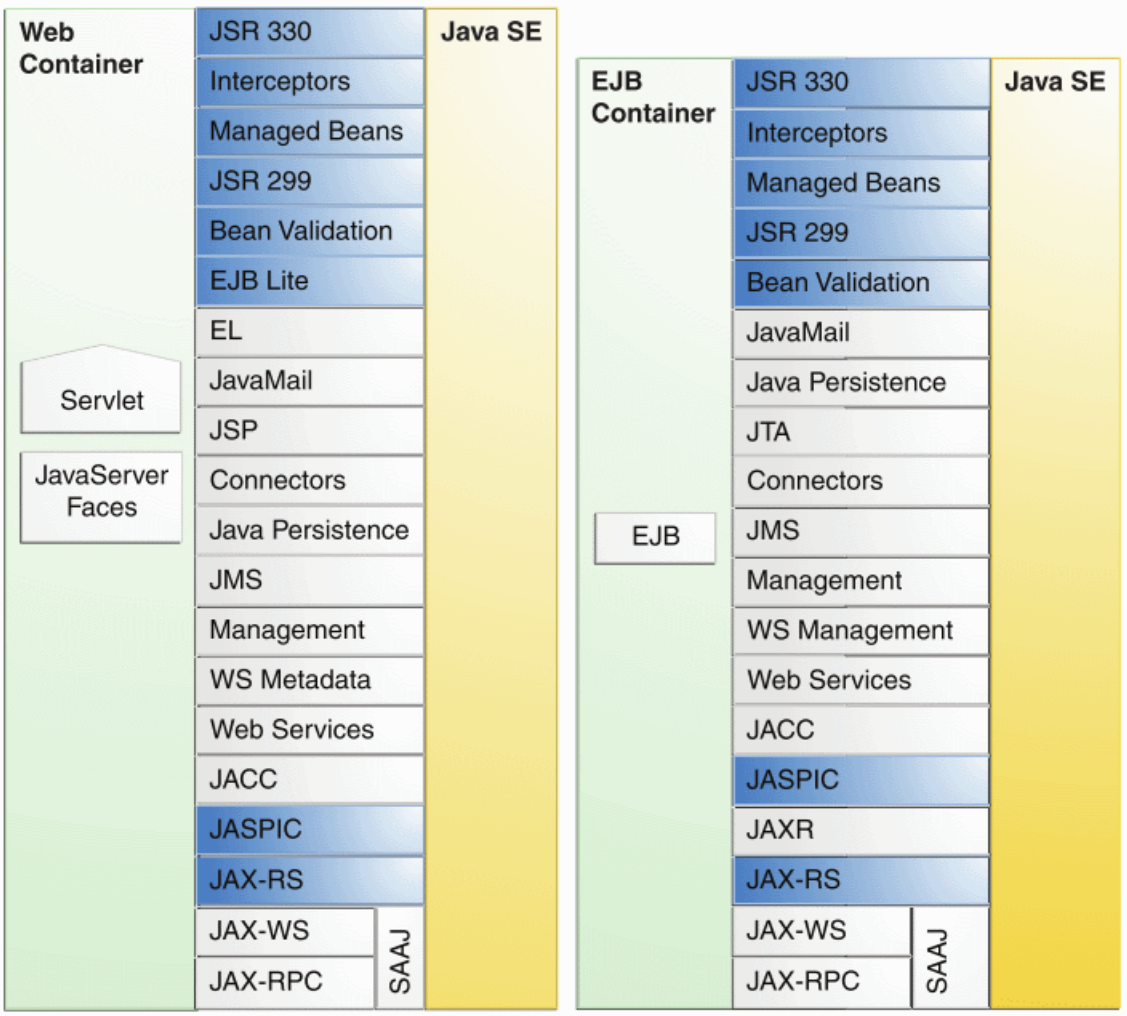
Aplikační model Java EE dále uvádí termín kontejner. business logika je v aplikačním modelu Java EE zapouzdřena do znovupoužitelných komponent. Java EE server těmto komponentám poskytuje služby formou kontejneru. Mezi služby poskytované kontejnerem patří například řízení transakcí, souběžnost, poolování zdrojů a management stavu. Kontejner je v podstatě rozhraní mezi business komponentou a funkcionalitou poskytovanou Java EE serverem. Předtím než bude komponenta použita, musí být přidána do Java EE modulu a nasazena do kontejneru Java EE serveru.



Obrázek 3: *Java EE kontejnery a jejich komunikace.* Zdroj: [10].



Obrázek 4*: APIs dostupné kontejneru klienta aplikace.* Zdroj: [10].



Obrázek 5: *APIs dostupné ve webovém a aplikačním (EJB) kontejneru.* Zdroj: [10].

Následující text obsahuje stručný popis APIs, které jsou součástí Java EE platformy, v pořadí, v jakém je uvádí [10].

- Enterprise JavaBeans (EJB) – technologie jejíž komponenty obsahují implementaci business logiky. EJB komponenty jsou samostatné celky, které samostatně nebo společně s ostatními EJB, vykonávají business logiku v kontejneru Java EE serveru.
- Java Servlet – rozšiřuje možnosti serverů hostujících aplikace, ke kterým se přistupuje na základě request-respond modelu. Ačkoliv servlety mohou odpovídat na jakýkoliv typ žádosti, jsou nejčastěji využívány k rozšíření webových aplikací.
- JavaServer Faces (JSF)– GUI framework pro vytváření webových aplikací.

- JavaServer Pages (JSP) – technologie, která umožňuje vložit části servletového kódu do textového dokumentu. JSP stránka je textový dokument, který obsahuje statická data vyjádřena např. v podobě HTML a JSP elementy, které určují jakým způsobem stránka vytvoří dynamický obsah.
- JavaServer Pages Standard Tag Library (JSTL) – zapouzdřuje základní funkcionalitu společnou pro většinu JSP aplikací. Poskytuje standardizovanou sadu JSP tagů. Tato standardizace umožňuje nasadit aplikaci na libovolný JSTL kontejner.
- Java Persistence API (JPA) – ORM specifikace založená na standardech Java.
- Java Transaction API (JTA) – standardizovaný interface pro demarkaci transakcí.
- Java API for RESTful Web Services (JAX-RS) – API pro vývoj webových aplikací založených na Representational State Transfer (REST) architektuře.
- Managed Beans – jednoduché kontejnerem řízené POJO (Plain Old Java Object) objekty s minimálními požadavky, které poskytují základní služby jako např. resource injection, life cycle callbacks a interceptory.
- Contexts and Dependency Injection for the Java EE Platform (JSR299, CDI) – sada kontextových služeb poskytovaných Java EE kontejnerem. Usnadňuje vývojářům práci s EJB a JSF technologií, poskytuje flexibilitu, umožňuje volné vazby a zaručuje typovou bezpečnost.
- Dependency injection for Java (JSR 330) – poskytuje standardizovanou sadu anotací, které lze využít na injektovatelné objekty.
- Bean Validation – poskytuje metadata model a API pro validaci v JavaBeans komponentách. Validace nemusí být distribuována mezi vrstvami, namísto toho se deleguje na business vrstvu.
- Java Message Service (JMS) – standard, který dovoluje Java EE komponentám vytvářet, číst a zasílat zprávy. Poskytuje spolehlivou asynchronní komunikaci s volnými vazbami.
- Java EE Connector Architecture – používá se k vytvoření adaptérů pro přístup Java EE aplikací k EIS. Typicky existuje jeden druh adaptéru pro jeden druh RDBMS. Poskytuje výkonově orientovaný, bezpečný, škálovatelný a transakční způsob integrace Java EE webových služeb do existujícího EIS.
- Java Mail API – umožňuje Java EE aplikacím zasílání emailů.
- Java Authorization Contract for Containers (JACC) – definuje kontrakt mezi Java EE aplikačním serverem a poskytovatelem autorizační politiky.
- Java Authentication Service Provider Interface for Containers (JASPIC) – definuje rozhraní poskytování služeb, pomocí kterého poskytovatel autentizace,

který implementuje autentizaci pomocí zasílání zpráv, může být integrován do metod zpracovávajících zprávy na klientovi nebo na serveru.

Následující APIs jsou součástí platformy Java SE 6.0, tím pádem jsou k dispozici i na platformě Java EE 6.0. [10]

- Java Database Connectivity API (JDBC) – umožňuje provádění SQL dotazů uvnitř metod. V prostředí Java EE lze využít k přístupu do databáze přímo ze servletu nebo JSF stránky, bez nutnosti průchodu Enterprise bean.
- Java Naming and Directory Interface API (JNDI) – umožňuje aplikaci přistupovat k službám jako např. LDAP, DNS, NDS a NIS. Dále poskytuje aplikacím metody pro provádění standardních adresářových operací, jako např. spojování atributů s objekty nebo vyhledávání objektu podle jejich jména. S použitím JNDI může aplikace uložit a načíst jakýkoliv Java objekt, což umožňuje aplikaci fungovat společně s jinou aplikací nebo systémem. Java EE naming services poskytují klientům, Enterprise beans a webovým komponentám přístup do JNDI naming environment. Naming environment umožňuje upravit komponentu bez nutnosti změny jejího kódu. Kontejner implementuje environment komponenty a poskytne jej jako tzv. naming context. Komponenta může vyhledat naming context pomocí JNDI rozhraní.
- JavaBeans Activation Framework (JAF) – poskytuje služby pro práci s libovolnou částí dat. Mezi tyto služby patří identifikace, zapouzdření přístupu, zjištění dostupných operací a vytvoření JavaBeans komponenty pro vyvolání těchto operací. Využívá jej Java Mail API.
- Java API for XML Processing (JAXP) – podporuje zpracování XML dokumentů prostřednictvím Document Object Model (DOM), Simple API for XML (SAX) a Extensible Stylesheet Language Transformation (XSLT). JAXP umožňuje překládat a transformovat XML dokumenty nezávisle na konkrétní implementaci zpracovávání XML.
- Java Architecture fo XML Binding (JAXB) – poskytuje komfortní způsob jak svázat XML schéma s reprezentací v jazyce Java. Všechny typy Java EE klientů a kontejnerů podporují JAXB API.
- Soap with Attachments for Java (SAAJ) – umožňuje vytvářet a konzumovat zprávy odpovídající SOAP formátu. Vývojáři většinou nepracují se SAAJ, ale s JAX-WS, který je vyšší úrovni a SAAJ využívá.
- Java API for XML Web Services – poskytuje podporu pro webové služby, které využívají JAXB pro svázaní XML s objekty v jazyce Java.
- Java Authentication and Authorization Service (JAAS) – poskytuje Java EE aplikacím způsob jakým autentizovat a autorizovat uživatele.

2.2 Web Tier

2.2.1 Webové aplikace

Webové aplikace jsou dynamickým rozšířením webového nebo aplikačního serveru. Můžeme je rozdělit na následující dva typy: [10]

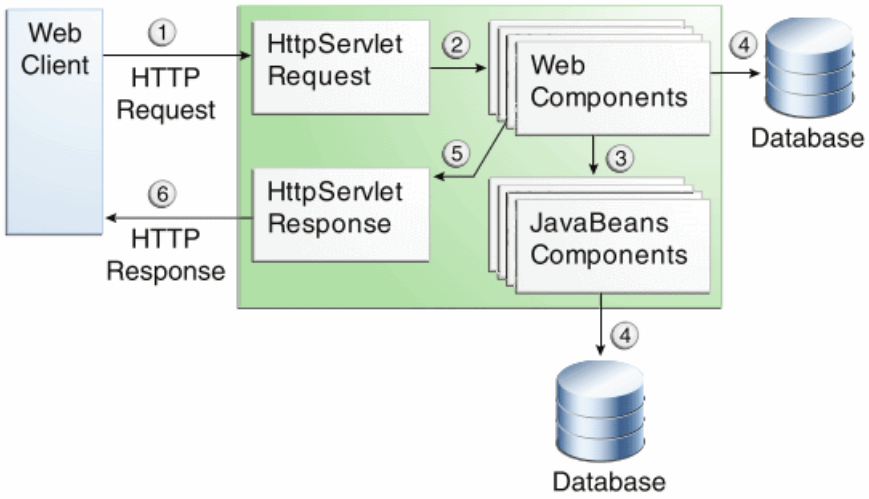
- Presentation-oriented
- Service-oriented

Presentation-oriented aplikace generují interaktivní webové stránky obsahující různé druhy značkovacího jazyka, které zobrazují dynamický obsah na základě požadavku uživatele.

Service-oriented aplikace implementují koncový bod webové služby. Presentation-oriented aplikace často bývají klienty service-oriented aplikací.

Webové komponenty na platformě Java EE poskytují možnosti dynamického rozšíření webového serveru. Tyto komponenty jsou: [10]

- webové stránky implementované JSF technologií
- koncové body webových služeb
- JSP stránky



Obrázek 6: *Zpracování požadavku webové aplikace na platformě Java EE.* Zdroj: [10]

Obrázek 6 ilustruje interakci mezi klientem a webovou aplikací. Klient pošle HTTP request na webový server. Webový server, který implementuje JavaServlet nebo JSP konvertuje požadavek na objekt typu `HttpServletRequest`. Tento objekt je následně doručen webové komponentě, která interakcí s JavaBeans komponentou nebo s databází vytvoří dynamický obsah. Webová komponenta nakonec vygeneruje

`HttpServletResponse` objekt, který webový server konvertuje na HTTP response a odešle klientovi.

Servlety jsou třídy napsané v jazyce Java, které dynamicky zpracují požadavek a vytvoří odpověď. Technologie JSF a Facelets jsou využívány pro vytvoření interaktivních webových aplikací. Servlety, JSF a Facelets lze využít k dosažení stejných výsledků, nicméně každá z těchto technologií přináší výhody pro konkrétní použití. Servlety se nejlépe hodí pro service-oriented aplikace nebo do kontrolních funkcí presentation-oriented aplikací. JSF a Facelets se nejlépe hodí pro generování značkovacího jazyka a jsou obecně využívány v presentation-oriented aplikacích. [10]

Webové komponenty jsou podporovány službami poskytovanými běhovou platformou, která se nazývá web container. Mezi tyto služby patří:

- přijímání a odesílání požadavků
- bezpečnost
- souběžnost
- řízení životního cyklu
- přístup k APIs jako např. JNDI, JTA, JavaMail

Některé aspekty webových aplikací mohou být konfigurovány při jejich nasazení do webového kontejneru. Konfigurační informace mohou být specifikovány pomocí Java EE anotací přímo v kódu webových komponent nebo v textovém XML souboru, který se nazývá Deployment descriptor. Deployment descriptor musí odpovídat schéma popsanému ve specifikaci JavaServlet API.

Webová aplikace se skládá z:

- statických zdrojů, jako jsou např. obrázky
- helper tříd a knihoven
- webových komponent

Webový kontejner poskytuje služby, které dále rozšíří funkčnost webových komponent a zjednoduší jejich vývoj. Nicméně kvůli těmto službám je proces vytvoření a nasazení webové aplikace odlišný od procesu vytvoření a spuštění tradičních Java tříd. Proces vytvoření, nasazení a spuštění webové aplikace může být sumarizován v těchto krocích: [10]

- vývoj kódu webových komponent
- vytvoření Deployment descriptoru
- kompilace kódu webových komponent a helper tříd, které jsou těmito komponentami referencovány
- zabalení aplikace do deployable unit

- nasazení aplikace do webového kontejneru
- přístup do aplikace zadáním příslušného URL

2.2.2 Technologie JavaServer Faces

JSF je framework, který využívá serverové komponenty pro vytváření webových aplikací založených na Java technologiích. Tento framework se skládá z následujících částí: [10]

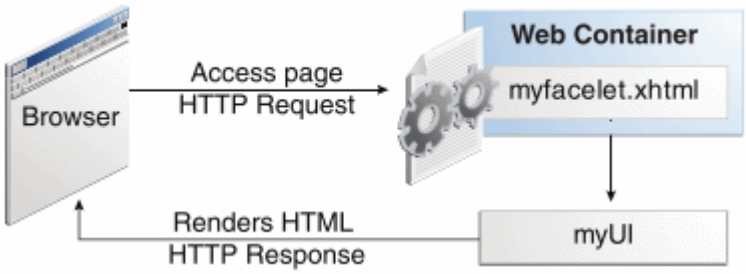
- API pro reprezentaci komponent, management jejich stavu, ošetření událostí, serverovou validaci, konverzi dat, definici navigace stránky, podporu internacionalizace a přístupnosti.
- Knihovna tagů pro přidávání komponent na webové stránky a připojení těchto komponent na jejich serverové protějšky.

JSF poskytuje dobře definovaný programový model, který značně usnadňuje vývoj a údržbu webových aplikací s uživatelskými rozhraními na straně serveru. Následující seznam obsahuje úkony, u kterých [10] použitím JSF deklaruje zjednodušení:

- vytvoření webové stránky
- přidání komponent na webovou stránku (přidáním příslušných tagů)
- svázaní komponent s daty na serverové straně
- svázaní událostí generovaných komponentami s kódem na straně serveru
- uložení a obnovení stavu aplikace za hranicí životního cyklu HTTP požadavku
- možnost znovupoužití a rozšíření komponent

Typická JSF aplikace obsahuje:

- sadu webových stránek v kterých jsou rozmístěny komponenty
- sadu tagů pro přidání komponent do webových stránek
- sadu Managed beans
- web Deployment descriptor
- volitelné konfigurační soubory pro definici navigace, konfiguraci Managed bean a konfiguraci uživatelských komponent
- sadu uživatelských objektů jako např. validátory, listenery, konvertery, atd.
- sadu uživatelských tagů reprezentujících uživatelské komponenty



Obrázek 7: *Vytvoření HTTP response JSF aplikací.*
Zdroj: [10]

Managed beans jsou jednoduché kontejnerem řízené POJOs. Obsahují podporu pro malou skupinu základních služeb, např. resource injection nebo interceptory.

Webová stránka `myfacelet.xhtml` na obrázku 7 je vytvořena pomocí JSF tagů komponent. Pomocí tagů jsou přidány komponenty do view `myUI`, což je serverová reprezentace stránky. View se renderuje jako HTTP response na HTTP request. Rendering je proces, kde kontejner na základě serverového view vytvoří výstup v podobě HTML nebo XHTML, který je pak čten prohlížečem klienta.

Největší výhodou JSF technologie, že webovým aplikacím nabízí čisté oddělení prezentační logiky a prezentace. JSF dále umožňuje mapovat HTTP request na event handler specifické komponenty a řídit komponenty jako stavové objekty na straně serveru.[10]

2.2.3 Facelets

Facelets je jednoduchý jazyk založený na XML, který se používá k vytvoření prezentační části JSF aplikací. Byl vyvinut jako součást technologie JSF 2.0. Umožňuje vytvářet šablony webových stránek a hierarchie komponent.

Facelets jsou od vydání specifikace JavaServer Faces 2.0 preferovanou technologií pro vytváření prezentační části webových aplikací. Původně tuto úlohu zastávaly JSP, ale ty nepodporují všechny nové vlastnosti JSF 2.0, a proto jsou pro vytváření prezentace JSF aplikací považovány za zastaralé. [10]

Následující body shrnují vlastnosti Facelets:

- vytváření webových stránek pomocí XHTML
- podpora knihoven tagů:
  - Facelets
  - JSF
  - JSTL
- podpora Expression language
- vytváření šablon webových stránek a komponent

Výhody Facelets pro vývoj rozsáhlejších a enterprise aplikací jsou následující:

- znovupoužitelnost kódu využitím šablon a kompozice komponent
- rozšíření funkcionality komponent a jiných objektů na straně serveru
- rychlejší kompilace
- validace Expression language výrazů při kompilaci
- výkonný rendering

Využití Facelets, vede ke zkrácení času potřebnému na vývoj, nasazení a údržbu webových aplikací. [10]

Facelets referencují vlastnosti a metody Managed beans pomocí Expression language, který je podporován JSF technologií. Expression language se, v tomto kontextu, využívá pro svázání komponent a objektů prezentace s metodami a atributy Managed beans.

Obecně je nutné při vývoji JSF aplikace, jejíž prezentační část využívá Facelets podniknout následující kroky: [10]

- vývoj Managed beans
- vývoj webových stránek pomocí tagů komponent
- definování navigace mezi webovými stránkami
- mapování instance `FacesServlet` na URL v Deployment descriptoru

2.2.4 Expression language

Expression language (EL) představuje důležitý mechanizmus, který umožňuje prezentaci (webové stránce) komunikovat s prezentační logikou (Managed bean). EL využívá jak technologie JSF, tak JSP.

Konkrétně JSF využívá EL v následujících případech:

- odložené nebo okamžité vyhodnocené výrazu
- čtení a zápis dat
- volání metod

Okamžité vyhodnocení výrazu znamená, že výsledek je vrácen jakmile je stránka renderována. Odložené vyhodnocení využívá vlastní mechanizmus, který vybere vhodný moment vyhodnocení výrazu, někdy během životního cyklu stránky. JSF využívá především odložené (deffered) vyhodnocení výrazu, které je z hlediska výkonu výhodnější. [10]

Příklad okamžitého vyhodnocení: `${sessionScope.cart.numberOfItems}`. Vrátí hodnotu vlastnosti numberOfItems atributu cart.

Příklad odloženého vyhodnocení: `#{customer.calcTotal}`. Vrátí výsledek metody calcTotal atributu customer. [10]

2.2.5 Technologie Java Servlet

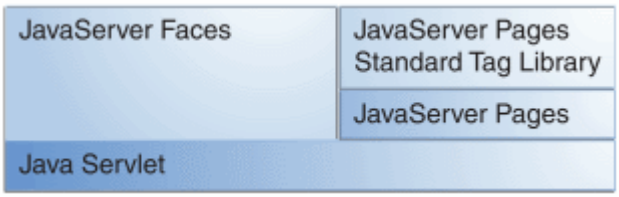
Servlet je třída programovacího jazyka Java, která se používá k rozšíření možností serveru, který hostuje aplikace s kterými se pracuje podle request & response aplikačního modelu. Servlet nejčastěji rozšiřuje možnosti webového serveru. Technologie Java Servlet k tomuto účelu definuje specifické HTTP servletové třídy.

Balíček `javax.servlet.http` poskytuje rozhraní a třídy pro vytváření servletů. Všechny servlety musí implementovat `Servlet` rozhraní, které definuje metody životního cyklu. Třída `HttpServlet` definuje metody jako např. `doGet` a `doPost`, pro práci s HTTP specifickými službami.

Životní cyklus servletu řídí kontejner do kterého byl nasazen. Jakmile je tomuto servletu mapován požadavek, kontejner vykoná následující kroky:

- pokud neexistuje instance servletu
  - načte třídu servletu
  - vytvoří instanci třídy servletu
  - inicializuje instanci servletu voláním metody `init()`
- zavolá metodu `service()` servletu, objekty response a request předá jako parametry

Technologie JSF využívá vlastní typ servletu – `FacesServlet`. Obrázek 8 znázorňuje, jak je JSF API postaveno přímo nad Servlet API. Technologie JSF tím zjednodušuje vývoj webových aplikací a částečně odstiňuje vývojáře od problematiky servletů.



Obrázek 8: *Java technologie pro vývoj webových aplikací.* Zdroj: [10]

Servlety mohou získávat informace z JavaBeans komponent, databáze, nebo je načítat z kontextových scope objektů. [10]

2.3 Web services

Web services (webové služby) jsou klientské a serverové aplikace, které komunikují pomocí HTTP. Jsou charakteristické vysokou mírou interoperability, které lze dosáhnout díky použití jazyka XML. Na konceptuální úrovni je service (služba) softwarová komponenta, přístupná přes koncový bod sítě. Konzument a producent služby spolu komunikují zasíláním zpráv. Na technické úrovni platforma Java EE implementuje webové služby dvěma způsoby:

- „Big“Web services – poskytováno skrz JAX-WS API
- RESTful web services – poskytováno skrz JAX-RS API

2.3.1 JAX-WS

„Big“Web services zasílá XML zprávy, které odpovídají formátu Simple Object Access Protocol (SOAP). SOAP je XML standard, který definuje architekturu zprávy a její formát. „Big“Web services často obsahují strojově čitelný popis operací, které jejich služby nabízejí. Tento popis je psaný ve Web Services Description Language (WSDL). WSDL je XML, které syntakticky definuje rozhraní. [10]

Návrh na základě SOAP musí obsahovat následující elementy:

- Formální kontrakt, který popisuje rozhraní, které webová služba poskytuje. WSDL může být použito k popsání detailů tohoto kontraktu. Nicméně SOAP zprávy lze v JAX-WS zpracovávat i bez publikování WSDL.
- Řešení nefunkcionálních požadavků. Jedná se např. o transakce, bezpečnost, důvěryhodnost, adresování, koordinace, atd.
- Asynchronní volání a zpracování. To je poskytováno standardem Web Service Reliable Messaging (WSRM) nebo přímo JAX-WS API.

JAX-WS API skrývá před vývojářem aplikace složitost SOAP zpráv. Vývojář na straně serveru definuje rozhraní a metody v programovacím jazyce Java. Na straně klienta se vytvoří proxy objekt, reprezentující službu, na kterém klient volá metody služby. Vývojář se nestará a vytváření a překládání SOAP zpráv, tuto funkcionalitu poskytuje JAX-WS API za běhu.

Obrázek 9: *Komunikace JAX-WS služby a klienta.* Zdroj:[10]

2.3.2 JAX-RS

Representational State Transfer (REST) web service jsou navrženy pro práci na WWW. REST architektura specifikuje omezení, např. jednotné rozhraní, které, pokud je aplikováno na webovou službu, vede na požadované vlastnosti, např. škálovatelnost, výkon, modifikovatelnost, atd. V REST architektuře jsou data a funkcionality považovány za zdroje, ke kterým se přistupuje pomocí Unified Resource Identifiers (URIs). Typicky se jedná o webové odkazy. REST architektura omezuje architekturu webové služby, která ji implementuje na client-server architekturu, kde komunikační protokol je bezstavový, typicky HTTP. [10]

RESTful služby jsou jednoduché, nenáročné a výkonné díky následujícím vlastnostem:

- Identifikace zdroje skrz URI – RESTful aplikace jednoznačně identifikují cíle interakce jejich klientů pomocí URI. URI poskytuje globální adresní a jmenný prostor.
- Jednotné rozhraní – se zdroji se manipuluje pomocí sady čtyř pevně stanovených CRUD operací. Konkrétně se jedná o tyto metody:
  - `PUT` – vytvoření nového zdroje
  - `Delete` – odstranění zdroje
  - `GET` – vrátí reprezentaci zdroje v aktuálním stavu
  - `POST` – převede zdroj do nového stavu
- Self-descriptive zprávy – zdroje jsou odděleny od jejich reprezentace, takže k jejich obsahu může být přistupováno v různých formátech, např. PDF, JPEG, XML, HTML, atd.
- Stavová interakce skrze hypertextové odkazy – každá interakce se zdrojem je bezstavová. Stavové interakce jsou založeny na konceptu explicitního přenosu stavu. K přenosu stavu slouží několik technik, jako např. cookies, skrytá formulářová pole nebo „URI rewriting“.

JAX-RS API je navrženo tak, aby co nejvíce zjednodušovalo vývoj aplikací využívajících REST architekturu. Vývojáři dekorují třídy napsané v programovacím jazyce Java JAX-RS anotacemi, které definují zdroje a akce, které nad těmito zdroji mohou být vykonány. Anotace jsou čteny za běhu pomocí reflexe. [10]

2.4 Enterprise Beans

Enterprise beans jsou Java EE komponenty, které implementují technologii Enterprise JavaBeans (EJB). Enterprise beans běží v prostředí EJB kontejneru. EJB kontejner je běhové prostředí poskytované aplikačním serverem. Kontejner poskytuje EJBs, které v něm běží služby na úrovni systému, např. bezpečnost.

EJB je serverová komponenta, která zapouzdřuje business logiku aplikace a je napsaná v programovacím jazyce Java. Cílem EJB je zjednodušit vývoj rozsáhlých distribuovaných aplikací. Tohoto cíle technologie EJB dosahuje zejména díky následujícím vlastnostem:

- EJB kontejner poskytuje system-level služby, vývojář aplikace se může soustředit na business funkcionalitu.
- Aplikační logika je obsažena v EJB. Vývojář klientské části aplikace se soustředí pouze na prezentaci a prezentační logiku. Díky tomu jsou klienti jednodušší a „tenčí“.
- EJB jsou přenosné komponenty. Díky standardním APIs může vývojář aplikace nasadit hotové EJB komponenty na jakýkoliv Java EE kompatibilní server.

Vývojář by podle [10] měl zvážit použití technologie EJB pokud na aplikaci, kterou navrhuje, existuje nějaký z následujících požadavků:

- Aplikace musí být škálovatelná. K obsloužení narůstajícího počtu uživatelů může být zapotřebí distribuovat komponenty aplikace na více strojů.
- K zajištění integrity dat musí být použity transakce. EJB technologie podporuje transakční mechanismus, který umožňuje konkurenční přístup ke sdíleným zdrojům.
- K aplikace bude přistupovat několik typů klientů.

EJB můžeme rozdělit na dva základní druhy:

- Session beans
- Message driven beans

2.4.1 Session beans

Session beans zapouzdřují business logiku, kterou klient programově volá skrz lokální, vzdálené nebo webové rozhraní. Klient přistupuje k aplikaci nasazené na serveru voláním metod Session beans. Session bean vykoná klientem požadovanou operaci na serveru, v běhovém prostředí EJB kontejneru, tím skrývá složitost před klientem. Session beans existují následující druhy:

- Stateful Session beans

- Stateless Session beans
- Singleton Session beans

Stateful Session bean reprezentuje právě jednu session (sezení, konverzaci) klienta a aplikace. Její stav je definován proměnnými její instance. Klient po dobu session komunikuje se svojí stateful Session bean, z tohoto důvodu se o stavu Stateful session bean někdy hovoří jako o conversational state (stavu konverzace). Stateful Session bean nemůže být sdílena, může mít pouze jednoho klienta. Jakmile klient ukončí session, instance Stateful Session bean je zrušena. [10]

Stateless Session bean neudržuje stav konverzace s klientem. Když klient zavolá metodu Stateless Session bean, instance Session bean obsahuje atributy, které reflektují specifický stav konverzace s klientem, ale tento stav je garantován pouze v rámci provádění volané metody. Po návratu z volané metody se už nelze na tento stav odvolávat. EJB kontejner udržuje instance Stateless Session bean v poolu. Všechny tyto instance považuje kontejner za rovnocenné. Při požadavku klienta na vykonání metody Stateless Session bean vybere EJB kontejner libovolnou instanci z poolu. Pokud v poolu není žádná volná instance, kontejner v případě, že nebylo dosaženo maximálního počtu instancí, vytvoří novou. Po návratu z metody tuto instanci opět vrátí do poolu. Z tohoto důvodu jsou instance Stateless Session bean dlouhodobé.

Stateless Session beans poskytují větší míru škálovatelnosti, protože jedna instance může obsloužit požadavky několika klientů. Získání instancí z poolu a jejich návrat jsou méně nákladné, než jejich vytváření a rušení. Aplikace ke kterým přistupuje mnoho klientů typicky využívají Stateless Session beans, protože menší počet instancí dokáže obsloužit stejný počet klientů.

Webová služba může být implementována pomocí Stateless Session bean.

Singleton Session bean je vytvořena pouze jednou a její životní cyklus je stejný jako životní cyklus celé aplikace. Je navržena pro případy, kdy je jediná instance Enterprise bean společná pro všechny uživatele aplikace a ti k ní konkurenčně přistupují. Funkčností se podobá Stateless Session bean a stejně tak může implementovat koncový bod webové služby. Singleton session beans udržují svůj stav mezi jednotlivými voláními, ale není vyžadováno aby jejich stav byl udržován mezi odstávkami nebo pády serveru. [10]

Použití Stateful Session beans doporučuje [10], pokud platí některá z následujících podmínek:

- stav bean reprezentuje stav interakce se specifickým klientem
- bean potřebuje udržovat data klienta i mezi jednotlivými voláními metod
- bean představuje prostředníka mezi klientem a ostatními částmi aplikace
- bean řídí workflow několika dalších Enterprise beans

Protože Stateless session bean poskytuje lepší výkon, doporučuje [10] její použití, pokud platí některá z následujících podmínek:

- stav bean nereprezentuje pro specifického klienta žádná data
- jednotlivé metody bean provádí obecný úkol, stejným způsobem pro všechny její klienty
- bean implementuje webovou službu

Vhodné případy kdy použít Singleton bean jsou podle [10] následující:

- stav bean je potřeba sdílet napříč celou aplikací
- k instanci bean je potřeba jednotlivými vlákny přistupovat konkurenčně
- aplikace potřebuje Enterprise bean, která vykoná určité operace při startu a ukončení celé aplikace
- bean implementuje webovou službu

2.4.2 Message driven beans

Message driven bean je Enterprise bean, která umožňuje Java EE aplikaci asynchronní zpracování zpráv. Jedná se o listener, který nereaguje na události, ale na zprávy JMS API. Odesílatelem takové zprávy může být klient, jiná Enterprise bean, webová komponenta nebo jiná aplikace používající JMS. [10]

Nejvíce patrným rozdílem mezi Session bean a Message driven bean je, že klienti do message driven bean nepřistupují pomocí rozhraní.

Message driven beans se nejvíce podobají Stateless Session beans, zejména v následujících ohledech: [10]

- instance message driven beans neudržují data a stav konverzace specifického klienta
- všechny instance message driven beans jsou ekvivalentní, což umožňuje kontejneru přidělit libovolnou právě nečinnou instanci jakékoliv příchozí zprávě
- jedna message driven bean, může zpracovávat zprávy několika klientů

Instance message driven bean mohou obsahovat proměnné, které jejich stav definují za hranicemi zpracování zpráv klienta. Jedná se například o připojení k JMS API, databázové připojení nebo reference na jinou Enterprise bean.

Komponenty na straně klienta nezískávají instanci Message driven bean, aby na ní volali metody. Místo toho, klientská komponenta odešle JMS zprávu na definovanou destinaci. Pokud je Message driven bean `MessageListener` této destinace, zprávu obdrží a zpracuje.

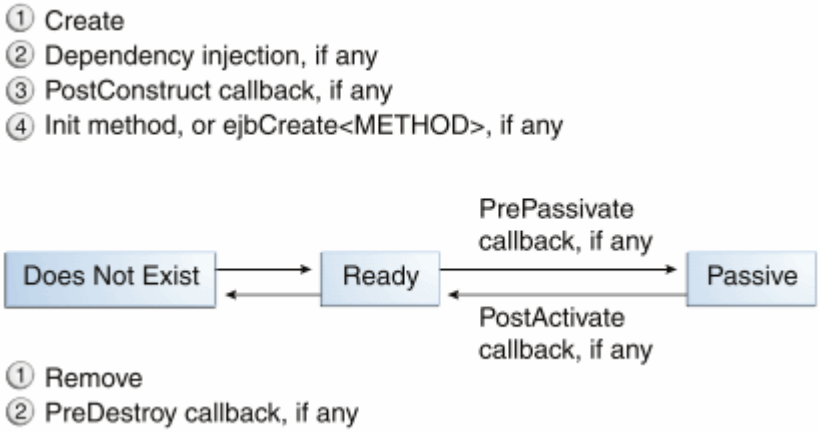
Následující charakteristiky shrnují chování message driven bean: [10]

- vykonávají metody po obdržení zprávy z klienta
- metody se vykonávají asynchronně
- nepoužívají se přímo pro reprezentaci databázových dat, ale přistupují k nim a mohou je aktualizovat
- transaction-aware
- bezstavové

Při doručení zprávy Message driven bean zavolá EJB kontejner metodu `onMessage()` této bean. Metoda `onMessage()` převede přijatou zprávu na jeden z pěti JMS typů zpráv a vykoná operace business logiky. Zpráva může být doručena v rámci transakce, v takovém případě se stane metoda `onMessage()` součástí této transakce.

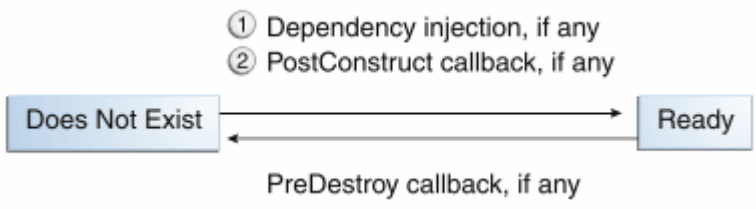
2.4.3 Životní cyklus Enterprise beans

Během životního cyklu prochází Enterprise bean různými stavy. Každý typ instance Enterprise bean má svůj specifický životní cyklus.



Obrázek 10: *Životní cyklus Stateful Session bean.* Zdroj: [10]

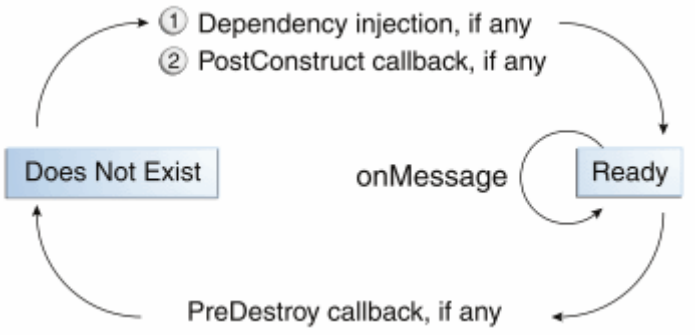
Obrázek 10 ilustruje stavy, kterými prochází Stateful Session bean během svého životního cyklu. Životní cyklus začíná požadavkem klienta na instanci Stateful Session bean. Kontejner vytvoří instanci, v případě potřeby provede dependency injection a zavolá metody s anotací `@PostConstruct`. Bean je nyní ve stavu „Ready“ a může obsluhovat požadavky klienta. Kontejner může instanci Stateful Session bean uložit do externí paměti. Při této operaci přejde tato instance do stavu „Passive“. Analogicky může kontejner instanci aktivovat a převést ji do stavu Ready. Při těchto operacích volá kontejner metody s anotacemi `@PrePassivate` a `@PostActive`. Klient ukončí práci



Obrázek 11*: Životní cyklus Stateless Session bean.*
Zdroj: [10]

s instancí voláním metody s anotací `@Remove`. Kontejner nejdříve zavolá metody s anotací `@PreDestroy` a pak instanci zruší (stav „Does not Exist“). [10]

Obrázek 11 ilustruje životní cyklus Stateless Session bean. Kontejner typicky udržuje instance Stateless Session bean v poolu, kde jsou ve stavu Ready. Ready znamená, že instance je připravena pro volání business metod. Kontejner volá metody s anotací `@PostConstruct` těsně po vytvoření instance a metody s anotací `@PreDestroy` těsně předtím, než kontejner dovolí instanci zrušit. [10]



Obrázek 12: *Životní cyklus Message driven bean.*
Zdroj: [10]

Životní cyklus Singleton Session bean je podobný jako ten Stateless Session bean. Pokud je třída Singleton Session bean opatřena anotací `@Startup`, vytvoří kontejner instanci okamžitě po nasazení aplikace.

Obrázek 12 ilustruje stavy kterými prochází Message driven bean. Životní cyklus Message Driven bean je opět podobný životnímu cyklu Stateless Session bean. Kontejner po vytvoření instance provede dependency injection a zavolá metody s anotací `@PostConstruct`. Před zrušením instance kontejner volá metody s anotací `@PreDestroy`. Instance Message Driven bean kontejner typicky udržuje v poolu. [10]

2.4.4 Použití Enterprise beans

Klient Enterprise beans může získat referenci na tuto bean jedním z následujících způsobů:

- dependency injection
- JNDI lookup

Dependency injection je nejjednodušší způsob jak získat referenci na instanci Enterprise bean. Tento přístup může využít pouze klient, který běží v prostředí Java EE serveru, většinou se jedná o: [10]

- JSF aplikace
- JAX-RS webové služby
- ostatní Enterprise beans

Klienti mimo aplikační server, např. aplikace Java SE, musí pro získání reference na EJB vykonat lookup. Java SE obsahuje JNDI API, které umožňuje provádět lookup na služby pomocí Portable JNDI Syntax. Portable JNDI Syntax používá při lookup na služby následující tři jmenné prostory: [10]

- `java:global`
- `java:module`
- `java:app`

Jmenný prostor `java:global` se používá pro lookup na vzdálenou EJB. Používá následující syntaxi pro specifikaci adresy:

```
java:global [/application name] /module name /enterprise bean name [/interface name]
```

Jmenný prostor `java:module` se používá pro lookup na lokální EJB, která se nachází ve stejném modulu (typicky soubor *.JAR) a má následující syntaxi:

```
java:module /enterprise bean name /[interface name]
```

Jmenný prostor `java:app` se používá pro lookup na lokální EJB, která se nachází ve stejné aplikaci (typicky soubor *.EAR) a má následující syntaxi:

```
java:app [/module name] /enterprise bean name [/interface name]
```
[10]

EJB mohou být zpřístupněny lokálně nebo vzdáleně. V prvním případě může k EJB přistupovat pouze klient, který běží na stejném aplikačním serveru jako EJB. V druhém

případě lze k EJB přistupovat libovolně. Lokální přístup obecně poskytuje lepší výkon, ovšem nelze jej využít v případě vzdálených klientů nebo distribuovaných komponent.

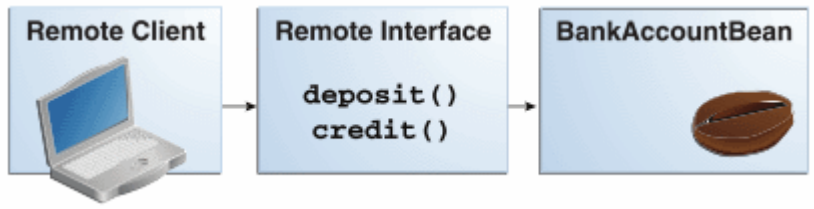
Lokálního klienta EJB charakterizují následující body:

- běží ve stejné aplikaci jako EJB ke které přistupuje
- může se jednat o webovou komponentu nebo jinou EJB
- není mu známé umístění EJB ke které přistupuje
- pro přístup k EJB může, ale nemusí využít business interface

Vzdáleného klienta EJB charakterizují následující body:

- může běžet na jiném stroji (JVM) než EJB ke které přistupuje
- může se jednat o webovou komponentu, klienta aplikace nebo jinou EJB
- je mu známé umístění EJB ke které přistupuje
- pro přístup k EJB musí využít business interface

Business interface pro lokální přístup obsahuje anotaci `@Local`, pro vzdálený pak `@Remote`.



Obrázek 13: *Volání metod EJB přes vzdálené business rozhraní.* Zdroj: [10]

Pro nasazení EJB do kontejneru aplikačního serveru je potřeba zabalit třídu EJB, její business rozhraní a helper třídy do JAR souboru. Tento soubor může být spolu s ostatními moduly Java EE aplikace dále zabalen do EAR nebo WAR souboru. Výsledný archiv se umístí do příslušného adresáře aplikačního serveru.

2.5 Persistence

Java Persistence poskytuje vývojářům možnost objektově-relačního mapování v Java EE aplikacích, které uchovávají data v relační databázi. Java Persistence lze rozdělit na následující okruhy:

- Java Persistence API (JPA)
- Java Persistence Query Language (JPQL)
- metadata objektově-relačního mapování

Cílem JPA je automaticky mapovat Java objekty do relační databáze a naopak relační data z databáze mapovat na Java objekty. JPA definuje Entity bean pro snadnou integraci do EJB technologie. Dále definuje Entity manager objekt, respektive interface, který je centrální autoritou operací persistence s Entity beans. [11]

JPQL je jazyk podobný SQL. Zatímco SQL pracuje s relačním schema, JPQL je navržen pro práci s objektovým modelem.

ORM nástroje potřebují definovat informace týkající se mapování tříd a tabulek, atributů a sloupců, asociací a cizích klíčů, datových typů jazyka Java a SQL, atd. Tyto informace se nazývají metadata objektově-relačního mapování. ORM metadata lze vkládat formou anotací přímo do zdrojového kódu tříd nebo je udržovat odděleně v XML souboru. [12]

2.5.1 Entity bean

Entita je lightweight doménový objekt persistence. Na rozdíl od Session nebo Message driven bean, Entity bean není serverový objekt, ale POJO. Typicky třída entity reprezentuje tabulku v relační databázi, instance této třídy pak reprezentují řádky této tabulky. Vývojáři umísťují kód do třídy entity, ale mohou vytvářet i helper třídy. Persistentní stav entity je definován obsahem jejích persistentních atributů nebo vlastností. Tyto atributy a vlastnosti obsahují ORM anotace, které je mapují do relační databáze. [10]

Třídu entity bean charakterizují následující body: [11]

- třída obsahuje anotaci `javax.persistence.Entity`
- třída musí obsahovat veřejný konstruktor bez parametrů
- persistentní atributy nemohou být deklarovány jako `final`
- pokud bude entita překračovat hranice EJB kontejneru, například instance entity bude návratový objekt metody vzdáleného business rozhraní, musí implementovat rozhraní `Serializable`
- třída entity může dědit od libovolné třídy a libovolná třída může dědit od třídy entity

- atributy entity musí být deklarovány jako `private` nebo `protected`, k těmto atributům může přímo přistupovat pouze třída samotná, ostatní třídy k nim přistupují pomocí metod

Pokud persistentní atribut reprezentuje nějakou kolekci, musí být použito jedno z následujících rozhraní: [10]

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

Každá entita obsahuje primární klíč, který ji jednoznačně identifikuje. Tento klíč může být jednoduchý nebo složený. Atribut jednoduchého primárního klíče obsahuje anotaci `javax.persistence.Id`. Složené primární klíče obsahují anotaci `javax.persistence.EmbeddedId` nebo `javax.persistence.IdClass`. [10]

Třída primárního klíče entity musí splňovat následující požadavky: [10]

- přístupový modifikátor třídy musí být `public`
- přístupové modifikátory vlastností musí být `public` nebo `protected`
- třída obsahuje veřejný bezparametrický konstruktor
- třída musí implementovat metody `hashCode()` a `equals(Object other)`
- třída musí být serializovatelná

Mezi entitami může existovat sedm typů vztahů. Nejprve lze rozlišit následující druhy kardinality: [11]

- one-to-one – jedna k jedné – anotace `@OneToOne`
- one-to-many – jedna k mnoha – anotace `@OneToMany`
- many-to-one – mnoho k jedné – anotace `@ManyToOne`
- many-to-many – mnoho k mnoha – anotace `@ManyToMany`

Každý vztah navíc může být unidirectional (jednosměrný) nebo bidirectional (obousměrný). To ve výsledku dává osm vztahů, ale one-to-many a many-to-one bidirectional jsou stejné. [11]

Bidirectional vztahy musí splňovat následující pravidla: [10]

- protistrana entity, která je vlastníkem vztahu, musí obsahovat v anotaci kardinality vztahu element `mappedBy`, který označuje atribut entity, která je vlastníkem vztahu

- strana many vztahu one-to-many nesmí definovat `mappedBy`, protože strana many je vždy vlastníkem vztahu
- ve vztahu many-to-many bidirectional, může být kterákoliv strana vlastníkem vztahu
- ve vztahu one-to-one bidirectional je vlastníkem vztahu ta strana, která obsahuje cizí klíč

Entity, které jsou spolu ve vztahu často mají závislost na existenci druhé entity. Například pokud je položka vlastněna objednávkou a objednávka je smazána, musí se odstranit i položka. Anotace kardinality umožňuje specifikovat `cascade` element, který nabývá jedné z hodnot výčtového typu `javax.Persistence.CascadeType`: [10]

- `ALL` – aplikuje všechny níže uvedené cascade operace
- `DETACH` – pokud entita, která je vlastníkem vztahu, přejde do stavu detached, přejdou do stavu detached i ostatní entity vztahu
- `MERGE` – pokud je nad entitou, která je vlastníkem vztahu, provedena merge operace, provede se tato operace také nad ostatními entitami vztahu
- `PERSIST` – pokud je nad entitou, která je vlastníkem vztahu, provedena persist operace, provede se tato operace také nad ostatními entitami vztahu
- `REFRESH` – pokud je nad entitou, která je vlastníkem vztahu, provedena refresh operace, provede se tato operace také nad ostatními entitami vztahu
- `REMOVE` – pokud je nad entitou, která je vlastníkem vztahu, provedena remove operace, provede se tato operace také nad ostatními entitami vztahu

2.5.2 Management Entit

Operace persistence prováděné nad Entity beans má na starosti služba `javax.persistence.EntityManager`. V JPA je entity manager centrální autoritou pro veškeré operace persistence. Protože entity jsou POJO, jejich stav není s datovou vrstvou synchronizován dokud programový kód explicitně nezavolá služby Entity managera. Entity manager řídí objektově-relační mapování mezi určitou množinou tříd entit a datovou vrstvou. Poskytuje API, které umožňuje vyhledávat objekty, vytvářet dotazy a synchronizovat objekty s relační databází. Umožňuje kešování a poskytuje interakci mezi entitou a transakčními službami platformy Java EE, jako např. JTA. Ačkoliv je Entity manager dobře integrován do prostředí Java EE, lze jeho služby využívat i jinde, např. v Java SE aplikaci. [11]

Instance entity bean může být ve stavu managed (nebo také attached), nebo unmanaged (detached). Pokud je entita ve stavu managed, Entity manager sleduje změny stavu entity a tyto změny propisuje do databáze. Pokud je entita ve stavu

unmanaged, změny jejího stavu se do databáze nepropíšou, Entity manager nesleduje změny stavu této entity. [11]

Sada entit, která je „managed“ jedním Entity managerem se nazývá Persistence context. Jakmile se Persistence context uzavře, všechny entity, které obsahuje přejdou do stavu unmanaged. Existují následující druhy Persistence kontextů: [11]

- Transaction-scoped
- Extended

Životní cyklus Transaction-scoped Persistence kontextu je svázaný s transakcí, v rámci které byl vytvořen. Jakmile transakce skončí, Transaction-scoped Persistence context se uzavře. Pouze Persistence context řízený kontejnerem může být Transaction-scoped. Jinými slovy, pouze Entity manager injektnutý anotací `@PersistenceContext`, může být Transaction-scoped. [11]

Persistence context, může existovat i déle než jedna transakce. Takový Persistence context se nazývá Extended. Entity připojené k Extended Persistence context během transakce, zůstávají ve stavu managed i po skončení této transakce. Tato vlastnost je důležitá v případech, kdy je potřeba udržovat konverzaci s databází a zároveň neudržovat dlouho trvající transakci, protože transakce zadržuje cenné zdroje. [11]

Entity, které jsou ve stavu detached mohou být serializovány a odeslány na vzdáleného klienta. Klient může provést na těchto entitách změny a odeslat je zpátky na server. Zde je pak lze opět připojit do Persistence kontextu metodou `merge(entita)` Entity managera.

Entity manager mapuje určitou sadu tříd entit do určité databáze. Tato sada se nazývá Persistence unit. Persistence unit je nutné vytvořit, aby bylo možné používat služby Entity managera. Aplikace využívající JPA vyžaduje existenci `persistence.xml` souboru, který je deployment descriptor Persistence unit. Tento souboru může definovat jednu nebo více Persistence unit. `Persistence.xml` je umístěn ve složce `META-INF/` jednoho z následujících umístění: [11]

- EJB JAR soubor
- složka `WEB-INF/classes` WAR souboru
- JAR soubor ve složce `WEB-INF/lib` WAR souboru
- JAR soubor, který je součástí EAR souboru
- JAR soubor klienta aplikace

Každá Persistence unit je svázána pouze s jediným zdrojem dat. Sada entit, která patří do určité Persistence unit nemusí být definována, v takovém případě JPA umožňuje aby Persistence provider prošel všechny třídy uvnitř JAR souboru a do Persistence unit přidal třídy anotované jako `@Entity`. [11]

Jakmile je Persistence unit připravená, je možné získat instanci Entity managera. Jednou z možností je využití třídy `EntityManagerFactory`. V prostředí Java SE lze Entity managera získat pouze pomocí této třídy. V prostředí Java EE lze navíc využít dependency injection. Příslušný atribut nebo setter metoda se opatří anotací `@PersistenceUnit`, kontejner pak automaticky do tohoto atributu vloží příslušnou instanci factory třídy.

Instanci samotného Entity managera pak vývojář vytvoří voláním metody `createEntityManager()` factory třídy. Takto vytvořený Entity manager reprezentuje Extended Persistence context. V prostředí Java EE je opět možné využít dependency injection a příslušný atribut opatřit anotací `@PersistenceContext`, kontejner do atributu vloží instanci Entity managera, která implicitně reprezentuje Transaction-scoped Persistence context. Toto nastavení lze změnit, ovšem Extended Persistence context lze injektovat pouze do Stateful Session bean.

Entity manager poskytuje pro práci s entitami následující metody:

- `persist()` – vytvoří nový záznam v databázi, vývojář alokuje novou instanci entity, nastaví její atributy, případně inicializuje vztahy s ostatními entitami a nakonec zavolá tuto metodu
- `flush()` – okamžitě propíše změny do databáze, entity manager může být nastaven aby propisoval změny do databáze okamžitě automaticky, ale implicitně, pokud je Persistence context součástí transakce, propisují se změny až při commitu této transakce nebo před dotazem
- `find()` – metoda vrací inicializovanou instanci entity, parametry jsou třída hledané entity a objekt primárního klíče této entity, pokud entita není nalezena v jedné z cache pamětí vede tato metoda na databázový select, pokud k zadanému PK entita neexistuje, vrátí null
- `getReference()` – tato metoda má stejnou funkci a parametry jako metoda `find()`, ovšem nevede na databázový select, místo toho vytvoří proxy objekt, tzv. „placeholder“, příslušný select se provede v momentě, kdy bude přistupováno k atributům tohoto objektu, dalším rozdílem je, že tato metoda nevrací v případě nenalezení entity null, ale vyhodí se `EntityNotFoundException`
- `merge()` – metoda umožňuje synchronizovat stav detached entity s databází, původní entita zůstává na dále ve stavu detached, `merge()` vytvoří její kopii, která je ve stavu attached a vrátí na ni odkaz (handle)
- `createQuery()` – vytvoří dynamický dotaz JPQL
- `createNamedQuery()` – vytvoří statický dotaz JPQL

- `createNativeQuery()` – umožňuje vytvořit textový SQL dotaz, který bude odpovídat syntaxi specifického RDBMS
- `remove()` – umožňuje odstranit entitu z databáze, SQL query, který entitu z databáze odstraní lze provést okamžitě voláním metody `flush()`, jinak se tento dotaz vykoná někdy později, typicky při commitu transakce
- `refresh()` – obnoví stav entity z databáze, tato operace způsobí, že lokální změny provedené na entitě, budou ztraceny, pokud daná entita není ve stavu attached, vyhodí se `IllegalArgumentException`
- `contains()` – metoda bere entitu jako argument a na základě toho jestli je tato entita ve stavu attached k příslušné instanci Entity managera vrátí `true`, nebo `false`
- `clear()` – převede všechny entity v Persistence context Entity managera do stavu detached, všechny lokální změny provedené na těchto entitách budou ztraceny, aby k tomu nedošlo, předchází typicky volání metody `flush()`
- `lock()` – umožňuje programové uzamčení entity, zámkem typu `WRITE` nebo `READ`

2.5.3 Java Persistence Query Language

JPQL definuje dotazy na entity a jejich persistentní stav. Umožňuje vytvářet dotazy, které pracují bez ohledu na konkrétního dodavatele RDBMS. Tento jazyk pracuje s abstraktním schéma entit, včetně jejich vztahů, definuje operátory a výrazy, které pracují na základě tohoto schéma. Abstraktní schéma entit definují třídy, které jsou součástí Persistence unit nad kterou konkrétní dotaz pracuje. Jinými slovy, JPAQL umožňuje vyhledat objekty a hodnoty na základě abstraktního, resp. objektového, modelu entit a jejich vztahů. Syntaxe je podobná SQL. [10]

JPQL dotaz lze vytvořit pomocí metod `createQuery()` a `createNamedQuery()` Entity managera. První metoda vytvoří dynamický JPQL dotaz. Dynamický dotaz se vytváří uvnitř business logiky, to umožňuje definovat strukturu dotazu za běhu, podle potřeby. Druhá metoda vytvoří statický dotaz. Statický dotaz je definován v podobě metadat a volá se podle svého jména, které musí být v rámci Persistence unit unikátní. Strukturu statického dotazu nelze měnit. Výhodou statických dotazů je, že jsou parsovány Java EE kontejnerem hned při deploy aplikace tudíž poskytují lepší výkon. Dynamický dotaz je parsován pokaždé před jeho provedením. [10]

JPQL dotazy mohou obsahovat dva druhy parametrů:

- named parameters – parametr se vloží s prefixem `:`
- position parameters – parametr se vloží s prefixem `?`

Parametr se následně nastaví voláním metody `setParameter()` nad dotazem, např:

- `return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName").setParameter("custName", name)` [10]
- `(“SELECT c FROM Customer c WHERE c.name LIKE ? 1”).setParameter(1, name)` [10]

JPQL dotazy, vykonávají následující operace:

- `SELECT`
- `UPDATE`
- `DELETE`

`UPDATE` a `DELETE` dotazy poskytují funkcionalitu tzv. bulk operations. Změna nad každou entitou vede na jeden výsledný SQL dotaz. Ve větším počtu entit se jedná o problém, protože velké množství dotazů zatěžuje komunikaci a konzumuje zdroje. Bulk operace umožňují změnit velké množství entit jedním dotazem.

Příkaz SELECT v JPQL obsahuje následující klauzule: [10]

- `SELECT` – definuje typ objektů, které jsou výsledkem dotazu
- `FROM` – definuje rozsah dotazu deklarací proměnných, které mohou být referencovány v ostatních klauzulích
- `WHERE` – volitelná klauzule, která filtruje výsledky dotazu podle zadaného klíče
- `GROUP BY` – volitelná klauzule, která rozděluje výsledky do skupin podle zadaných kritérií
- `HAVING` – volitelná klauzule, kterou lze použít s `GROUP BY` k dalšímu filtrování výsledků dotazu
- `ORDER BY` – volitelná klauzule, která řadí výsledky dotazu

Velkým rozdílem JPQL proti SQL je, že výrazy JPQL navigují, resp. traverzují mezi atributy a entitami. Pokud atributem není kolekce, není potřeba specifikovat JOIN operaci. Tento rozdíl je patrný na následujícím příkladě:

- `SELECT * FROM team JOIN player ON player.team_id = team.team_id WHERE player.player_id = 1`
- `SELECT p.team FROM Player p WHERE p.player_id = 1`

Oba dotazy vrací team hráče specifikovaného atributem `player_id`. První dotaz napsaný v SQL vrací řádky tabulky `team`, které odpovídají zadané podmínce. Druhý dotaz napsaný v JPQL vrací entitu `Team`. JPA provider Java EE kontejneru druhý dotaz přeloží na dotaz podobný prvnímu dotazu pomocí ORM metadat definovaných v třídách

těchto dvou entit. Po vykonání takto přeloženého dotazu JPA provider vytvoří instanci hledané entity.

Kolekce hráčů takto vyhledaného týmu je reprezentována JPA proxy objektem. Pokud se odkážeme na nějaký z atributů tohoto proxy objektu kontejner na pozadí provede databázový select, vytvoří instance hráčů a daný proxy objekt nahradí klasickou kolekcí. Tento postup se nazývá lazy inicializace. Opakem lazy inicializace je eager inicializace, která instance hráčů vytvoří hned při vytváření instance týmu. Typ inicializace se nastavuje atributem `FetchType` anotace definující vztah entit. Pokud není specifikován, použije se lazy, protože ve většině případů poskytuje lepší výkon.

Eager inicializaci lze vynutit přímo v JPQL dotazu:

- `SELECT t FROM Team t JOIN FETCH t.plyers WHERE t.team_id = 1`

Tento dotaz vrátí instanci týmu jehož `team_id = 1`. Zároveň jsou vytvořeny instance všech N hráčů tohoto týmu. Nyní, pokud by se v cyklu četli atributy těchto hráčů, nevytvořil by se žádný další query. Kdyby ovšem dotaz neobsahoval klíčové slovo `FETCH` vytvořilo by se N dalších dotazů což by mohlo způsobit problémy s výkonem. Odborná literatura o tomto problému hovoří, jako o N+1.

2.5.4 Konkurenční přístup k datům

V enterprise aplikacích často dochází k situaci, kdy několik uživatelů hodlá v jeden okamžik pracovat s jednou sadou dat. Tato situace se nazývá konkurenční přístup k datům. Data jsou v relační databázi reprezentována řádkem tabulky, v prostředí Java EE kontejneru pak entitou. Může dojít k situaci, kdy uživatel pracuje s daty, která mezi tím jiný uživatel změnil nebo dokonce smazal, což vede ke ztrátě integrity dat. Protože integrita dat je zásadní, je nutné transakce, které s daty pracují, izolovat aby nedocházelo k těmto jevům:

- Dirty reads
- Non-repeatable reads
- Phantom reads

Těžiště řešení problému leží na úrovni datové vrstvy, konkrétně se jedná o to, jakým způsobem RDBMS izoluje transakce. Na úrovni aplikační vrstvy umožňuje JPA izolovat data pomocí mechanizmu zamykání. Obecně existují dva druhy zamykání:

- pesimistické
- optimistické

Pesimistické zamykání zamezuje vzniku chyby. Uživatel uzamkne všechny řádky tabulek, které reprezentují data s kterými hodlá pracovat. Persistence provider vygeneruje nad daty, která se mají zamknout, databázový příkaz `SELECT FOR UPDATE`. Existují dva typy zámků, kterými lze data uzamknout:

- `WRITE` – zamezuje ostatním uživatelům jakoukoliv interakci s uzamčenými daty
- `READ` – umožňuje ostatním uživatelům uzamčená data číst, ale ne je měnit

Nevýhodou pesimistického zamykání je, že drží dlouho trvající zámky v databázi, což má negativní vliv na výkon aplikace. Ostatní uživatelé musí čekat na dokončení transakce uživatele, který drží zámek konkrétních dat. Nicméně [10], [11] doporučují použití pesimistického zamykání v případě, kdy jsou určitá data velmi často modifikována velkým množstvím transakcí. V takovém případě poskytuje pesimistické zamykání lepší výkon (a uživatelský komfort) než optimistické.

Optimistické zamykání předpokládá, že problémy s konkurenčním přístupem k datům budou vznikat ojediněle. Odpovědností vývojáře je tyto problémy (v podobě výjimek) zachytit a ošetřit. Implementace optimistického zamykání je složitější, ale ve většině případů poskytuje lepší výkon. Uživatel obdrží instanci entity na které provádí změny, když jsou tyto změny hotové, je potřeba je uložit. Aplikace se nejdříve pokusí získat `WRITE` zámek nad těmito daty. Následně Persistence provider zkontroluje, že data, která uživatel modifikoval nebyla, v intervalu od jejich načtení do jejich uložení, jinou transakcí změněna Tato kontrola probíhá porovnáním verze dat v entitě s verzí dat v tabulce. Tabulka dat, která mají být optimisticky zamykána musí obsahovat sloupec, který bude držet informaci o verzi dat daného řádku. Persistence provider hodnotu tohoto sloupce automaticky inkrementuje po každé úspěšné změně dat. Atribut entity mapovaný na tento sloupec je nutné označit anotací `@Version`. Pokud Persistence provider při ukládání změn zjistí, že verze v entitě a verze v databázi si nejsou rovny, vyhodí `OptimisticLockException.`

Z těchto dvou obecných principů zamykání JPA odvozuje následující módy zamykání: [10]

- `OPTIMISTIC` – entita s atributem anotovaným `@Version` obdrží optimistický zámek `READ`
- `OPTIMISTIC_FORCE_INCREMENT` – entita s atributem anotovaným `@Version` obdrží optimistický zámek `READ` a zároveň se inkrementuje hodnota verze
- `PESIMITIC_READ` – okamžitě obdrží zámek `READ`, který trvá do ukončení transakce
- `PESIMISTIC_WRITE` – okamžitě uzamkne data zámkem `WRITE`, který drží do konce transakce

- `PESIMISTIC_FORCE_INCREMENT` – okamžitě uzamkne data zámkem `READ` a inkrementuje hodnotu verze
- `READ` – stejné jako `OPTIMISTIC`, [10] doporučuje používat `OPTIMISTIC`
- `WRITE` – stejné jako `OPTIMISTIC_FORCE_INCREMENT`, [10] doporučuje používat `OPTIMISTIC_FORCE_INCREMENT`

Tento mód zamykání lze specifikovat při volání jedné z následujících metod entity managera:

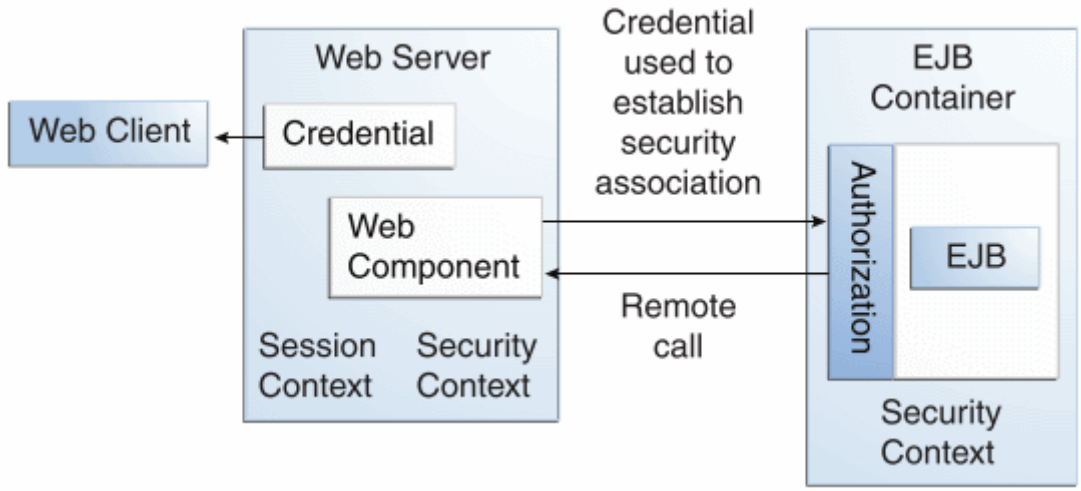
- `lock()`
- `find()`
- `refresh()`

Mód zamykání lze nastavit i objektu `Query` a lze jej specifikovat i v anotaci `@NamedQuery`.

2.6 Bezpečnost

Bezpečnost je komponentám Java EE aplikací poskytována kontejnery v kterých jsou tyto komponenty umístěny. Kontejner poskytuje dva druhy bezpečnosti: [10]

- declarative – požadavky komponenty na bezpečnost jsou vyjádřeny ve formě metadat (v XML deployment descriptoru nebo v aplikačním kódu pomocí anotací)
- programmatic – bezpečnost tvoří samostatný modul, který je vložen do aplikace, typicky se jedná o případy kdy declarative přístup není dostatečný pro splnění požadavků na bezpečnost



Obrázek 14: *Zabezpečení webové Java EE aplikace.* Zdroj: [10]

Na obrázku 14 webový klient požádá o hlavní URL aplikace. Webový server zjistí, že tento klient zatím není autentizován, proto vrátí formulář pro zadání údajů o uživateli (Credential). Po odeslání formuláře provede webový kontejner autentizaci a autorizaci. Autorizace probíhá porovnáním role uživatele a role kterou vyžaduje požadovaná webová komponenta. Webová komponenta volá business metodu EJB, které poskytne Credential uživatele. Na úrovni EJB kontejneru se vytvoří Security context a opět se provede autorizace. Záporný výsledek autorizace vede na exception.

Správně implementovaný mechanizmus zabezpečení by měl mít následující vlastnosti: [10]

- zabraňuje neautorizovanému přístupu k funkcím a datům aplikace
- umožňuje aby uživatel nesl odpovědnost, za operace, které vykonal
- chrání systém před útoky a akcemi, které by narušili poskytovaný QoS
- jednoduchá administrace
- transparentní pro uživatele

- interoperabilní – lze jej snadno rozšířit za hranice aplikace nebo firmy

Platforma Java EE obsahuje následující mechanizmy zabezpečení: [10]

- mechanizmy, které jsou součástí Java SE
  - Java Authentication and Authorization Service (JAAS) – sada API pro kontrolu přístupu a ověřování uživatelů, tvoří základ bezpečnostních mechanizmů Java EE
  - Java Generic Security Services (Java GSS-API) – API používané pro zabezpečenou výměnu zpráv dvou komunikujících aplikací
  - Java Cryptography Extension (JCE) – framework poskytující algoritmy pro šifrování a generování klíčů
  - Java Secure Sockets Extension (JSSE) – framework pro Java implementaci SSL a TSL
  - SASL – protokol specifikující autentizaci a případné zavedení vrstvy bezpečnosti mezi klientem a serverovými aplikacemi
- mechanizmy, které jsou součástí Java EE – jsou poskytovány kontejnerem a mohou být implementovány declarative nebo programmatic způsobem, Java EE dělí zabezpečení enterprise aplikací do následujících logických vrstev:
  - Aplication-Layer Security
  - Transport-Layer Security
  - Message-Layer Security

2.6.1 Zabezpečení na úrovni aplikační vrstvy

Aplikační logika je v aplikačním modelu Java EE zapouzdřena v EJB komponentách. Tyto komponenty vykonávají svůj kód v běhovém prostředí EJB kontejneru. Jednou z několika služeb, které kontejner komponentám poskytuje je bezpečnost. Technika zabezpečení, kterou EJB kontejner využívá, se nazývá role-based security. Uživateli se přidělí role, při volání metody EJB se tato role porovná s rolemi, které volaná metoda vyžaduje.

Když se vzdálený klient přihlásí k odběru nějaké služby EJB, na pozadí proběhne autentizace a vytvoří se identita, která existuje po dobu trvání session. Jakmile klient zavolá nějakou metodu EJB, kontejner automaticky přidá tuto identitu do volání, aby mohla proběhnout autorizace. EJB specifikace nedefinuje jakým způsobem autentizace probíhá, nicméně definuje jakým způsobem má klient informace propagovat na server. Běžný přístup aplikačních serverů je autentizace za pomoci JNDI API. Klient specifikuje přihlašovací údaje do vlastností objektu, který slouží pro získání JNDI spojení s EJB serverem, např.:

- ```
  properties.put(Context.SECURITY_PRINCIPAL, "username");
  properties.put(Context.SECURITY_CREDENTIALS, "password");
  Context ctx = new InitialContext(properties);
  Object ref = ctx.lookup("SecureBean/remote");
  SecureRemoteBusiness remote = (SecureRemoteBusiness)ref;
  ```
  [11]

Jakmile je uživatel autentizován, je potřeba provést kontrolu, že má právo provádět požadovanou operaci. Jinými slovy, je potřeba uživatele pro danou operaci autorizovat. Na rozdíl od autentizace je autorizace přesně definována specifikací EJB. Autorizace se v EJB provádí porovnáním přidělených rolí uživateli a rolí vyžadovaných pro volání business metody. Pro definici těchto rolí slouží anotace `@RolesAllowed`, která může být platná pro celou třídu nebo pro konkrétní metodu. Anotace `@PermitAll`, umožňuje volání metody kterémukoliv uživateli, zároveň se jedná o implicitní nastavení.

Programmatic security slouží v případech, kdy je např. potřeba se v těle business metody dotázat na roli nebo uživatele, který metodu volá. K tomuto účelu slouží třída `EJBContext` nebo její potomek `SessionContext`. Instanci této třídy lze získat pomocí DI od kontejneru anotací `@Resource` příslušného atributu. `SessionContext` poskytuje v souvislosti s bezpečností například tyto metody: [11]

- `getCallerPrincipal()` – vrací objekt uživatele, který volá danou metodu
- `isCallerInRole()` – vrací příznak, zda je uživatel volající metodu v roli předané v parametru

Programmatic security může být užitečná například, když je potřeba ověřit, že uživatel je v určité roli a zároveň business metodu volá v předepsaném časovém intervalu.
  ```

2.7 Transakčnost

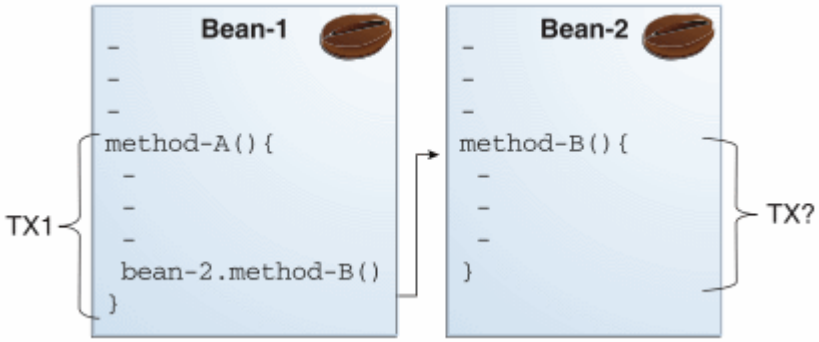
Úkolem transakcí je zajištění integrity dat. Obecně platí, že transakce musí splňovat následující čtyři charakteristiky, které se často souhrnně označují slovem ACID: [11]

- Atomic – nedělitelnost. Transakce proběhne celá, nebo vůbec. Pokud jeden ze sady úkolů transakce selže, transakce se odvolá a data se vrátí do stavu před začátkem transakce. Po úspěšném dokončení všech úkolů se volá commit a změny provedené transakcí se stanou permanentní.
- Consistent – konzistence souvisí s integritou dat v databázových tabulkách. Ta musí být zajištěna z části transakčním systémem a z části vývojářem datové vrstvy definicí validací.
- Isolated – izolace. Průběh transakce nesmí být narušen jinou transakcí nebo procesem. Jinými slovy, data s kterými transakce pracuje nesmí být ovlivněna jinou částí systému dokud transakce neskončí.
- Durable – trvanlivost. Změny provedená nad daty v průběhu transakce, musí být zapisovány do fyzické paměti, dokud transakce neskončí. V případě selhání, lze potom tyto změny dohledat.

Platforma Java EE umožňuje následující způsoby řízení (resp. se jedná o nastavení hranic – transaction demarcation) transakcí: [10]

- Container-managed transactions
- Bean-managed transactions

Při použití Container-managed transactions nastavuje hranice transakcí kontejner. Tento přístup zjednodušuje vývoj, protože kód EJB nemusí obsahovat příkazy, které řídí transakce. Typicky transakce začne před vstupem do volané metody a je ukončena před návratem. Každá metoda je spojená s maximálně jednou transakcí. Jakým způsobem bude metoda spojena s transakcí lze definovat transaction atributem. [10]



Obrázek 15: *Transaction scope.* Zdroj: [10]

Transaction attribute definuje transaction scope – platnost transakce, viz. obrázek 15. Metoda A začne transakci a zavolá metodu B. Před vstupem do metody B může začít nová transakce, nebo lze pokračovat v transakci metody A, nebo například lze metodu B vykonat bez spojení s transakcí. Toto rozhodnutí se řídí transaction atributem metody B. [10]
Transaction attribute může nabývat následujících hodnot: [10]

- `Required` – implicitní nastavení všech business metod. Pokud má volající klient aktivní transakci, volaná metoda se stane součástí této transakce. Pokud volající klient nemá aktivní transakci, vytvoří se před vstupem do metody nová.
- `RequiresNew` – před vstupem do metody se vždy vytvoří nová transakce. Pokud už volající klient má aktivní transakci, tak se tato transakce po dobu vykonávání volané metody pozastaví.
- `Mandatory` – volaná metoda se stane součástí transakce volajícího klienta. Pokud volající klient nemá aktivní transakci, vyhodí se `TransactionRequiredException`.
- `NotSupported` – volaná metoda nebude součástí žádné transakce. Pokud volající klient má aktivní transakci, tak se tato transakce po dobu vykonávání volané metody pozastaví. Protože transakce drží systémové zdroje a jejich řízení představuje overhead, lze tento atribut využít k navýšení výkonu, tam, kde není transakce vyžadována.
- `Supports` – pokud má volající klient aktivní transakci, tak se stane volaná metoda její součástí. Pokud volající klient nemá aktivní transakci, pokračuje volaná metoda bez transakce.
- `Never` – volaná metoda nebude součástí žádné transakce. Pokud volající klient má aktivní transakci, vyhodí se `RemoteException`.

Transaction attribute se metodě nastavuje anotací `@TransactionAttribute`.

Při použití Container-managed transactions lze provést rollback transakce dvěma způsoby. První z nich je vznik `RuntimeException`, v takovém případě kontejner transakci automaticky zavolá rollback. Druhou možností je volání metody `setRollbackOnly()` instance `EJBContext`. Vznik `ApplicationException` nevede nutně na rollback, v takovém případě lze volat tuto metodu. [10]

Při použití bean-managed transaction nastavuje hranice transakcí vývojář, přímo v těle business metod. To může být užitečné například v případech, kdy je o startu nové transakce potřeba rozhodnout na základě nějaké podmínky. Zároveň umožňuje, aby metoda byla součástí více než jedné transakce. [10]

3 Java EE server

Aplikační server poskytuje prostředí a služby pro běh aplikací. Aplikace přistupují ke službám serveru prostřednictvím kontejneru. Obecně lze servery v souvislosti s vývojem na platformě Java EE rozdělit do těchto skupin:

- HTTP servery – klient zasílá na server HTTP request a server zasílá klientovi HTTP response, typicky ve formě HTML dokumentu. Nejpoužívanějším HTTP serverem je Apache, který je vydáván pod licencí podobnou GPL.
- Java aplikační servery – servery zaměřené především na generování dynamického obsahu webových stránek. Poskytují webový kontejner na kterém může běžet servlet a je schopný zpracovávat například JSP stránky. Tyto servery mohou fungovat i jako HTTP servery, ale toto řešení většinou není dostatečně výkonné pro potřeby rozsáhlých webových aplikací, takže se často do aplikace zařazuje i HTTP server. Java aplikační servery poskytují podmnožinu funkcí certifikovaných Java EE aplikačních serverů. Nejznámějším zástupcem této skupiny je Tomcat, který obsahuje referenční implementaci JSP a Java Servlet.
- Java EE certifikované servery – servery, které splňují specifikaci Java EE. Aby mohl být server certifikován musí být podroben sadě testů společnosti Sun. Nejpoužívanějším aplikačním Java EE serverem je JBoss.

Aplikační servery také poskytují nástroje pro vývoj a údržbu aplikací.
Dominantní komerční Java EE aplikační servery:

- WebSphere (IBM)
- WebLogic (Oracle)

Některé open source Java EE aplikační servery:

- JBoss (Red Hat)
- Geronimo (Apache)
- GlassFish (Oracle)
- JonAS (konsorcium ObjectWeb)

3.1 JBoss AS

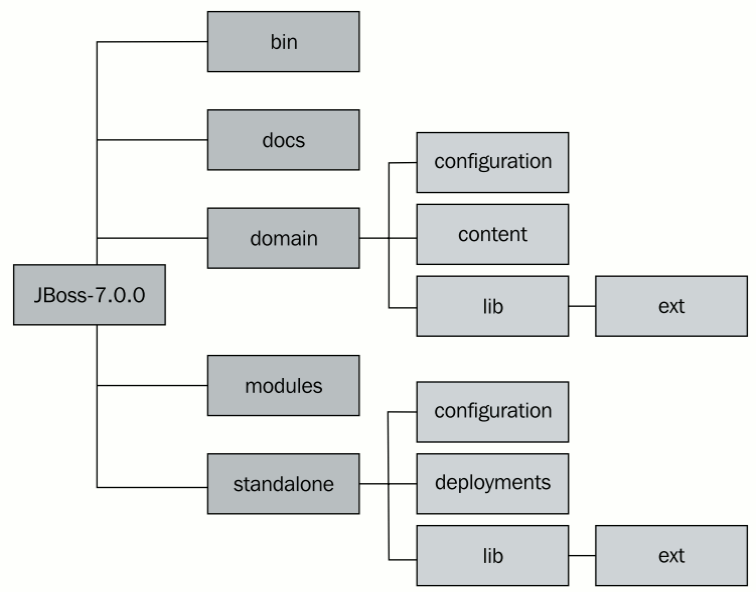
JBoss AS je open-source Java EE certifikovaný aplikační server. Protože server běží na virtuálním stroji Java, lze ho použít nezávisle na operačním systému stroje, stačí aby systém obsahoval JVM. Projekt založil v roce 1999 Marc Fleury. Později se JBoss stal divizí společnosti Red Hat. JBoss AS je vydávaný pod licencí GNU, takže není nutné kupovat licenci, nicméně Red Hat nabízí komerční placenou verzi ke které zákazník získá rychlou technickou podporu.

Aktuální verze JBoss AS je 7.1, vydaná v únoru 2012. Aplikační kód serveru prošel ve verzi 7 kompletní revizí aby splňoval nové požadavky. Mezi tyto požadavky patří zejména bohatá sada funkcionalit a zároveň jednoduchá a flexibilní konfigurace kontejnerů. Nejviditelnější změny oproti předchozí verzi jsou přibližně poloviční velikost na disku a přibližně desetkrát rychlejší startování instance serveru.

JBoss AS není nutné instalovat, distribuuje se ve formě .zip archivu, který je možné rozbalit do libovolného umístění na disku. Po rozbalení je nutné pro správnou funkci nastavit systémové proměnné. Zastavení a start serveru lze provádět z příkazové řádky nebo provedením skriptů přiložených od vydavatele. Tyto operace lze provádět i přímo v IDE, v případě použití Eclipse lze stáhnout plugin JBoss tools, který usnadňuje vývoj na JBoss AS a s ním spojených technologiích.

Jako klíčové vlastnosti AS uvádí Red Hat: [13]

- kompletní integrace do vývojového prostředí Eclipse
- dodržení standardů a interoperability
- certifikace Java EE, ORM řešení pomocí Hibernate framework
- JBoss Seam framework
- caching, clustering, failover, load-balancing
- rozšířená bezpečnost (nad rámec JAAS)
- konzistentní správa modulů



Obrázek 16: *Adresářová struktura JBoss AS 7.* Zdroj: [14]

Popis jednotlivých adresářů znázorněných na obrázku 16: [14]

- bin – obsahuje skripty pro ovládání instancí AS
- docs – obsahuje informace o licenci AS a technologiích na kterých je závislý
- domain – adresářová struktura domény, obsahuje konfigurační XML soubory, soubory s logy a Java EE/SE rozšíření
- modules – knihovny AS
- standalone – podobná struktura jako v adresáři domain, pokud je AS používán v standalone módu obsahuje konfiguraci a soubory s logy. Adresář deployment slouží pro deploy aplikace na server. AS tento adresář skenuje a v případě změny provede deploy nové aplikace. Uživatelům JBoss AS, zejména v produkčním prostředí, doporučuje [14] nespoléhat na systém automatického skenování adresáře a raději pro deploy aplikace využít JBoss management APIs.

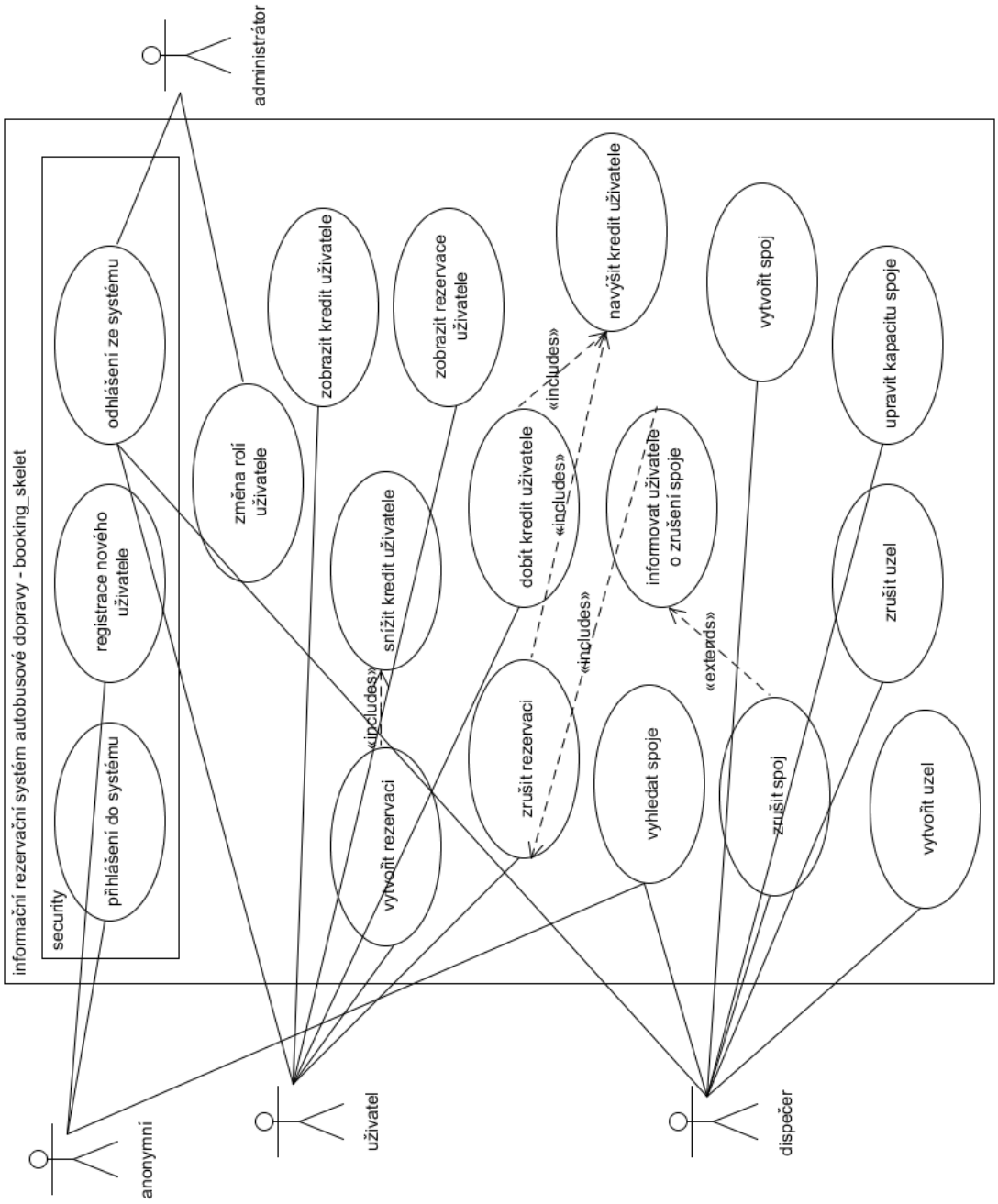
4 Návrh aplikace

4.1 Specifikace požadavků

Požadavky plynoucí ze zadání práce:

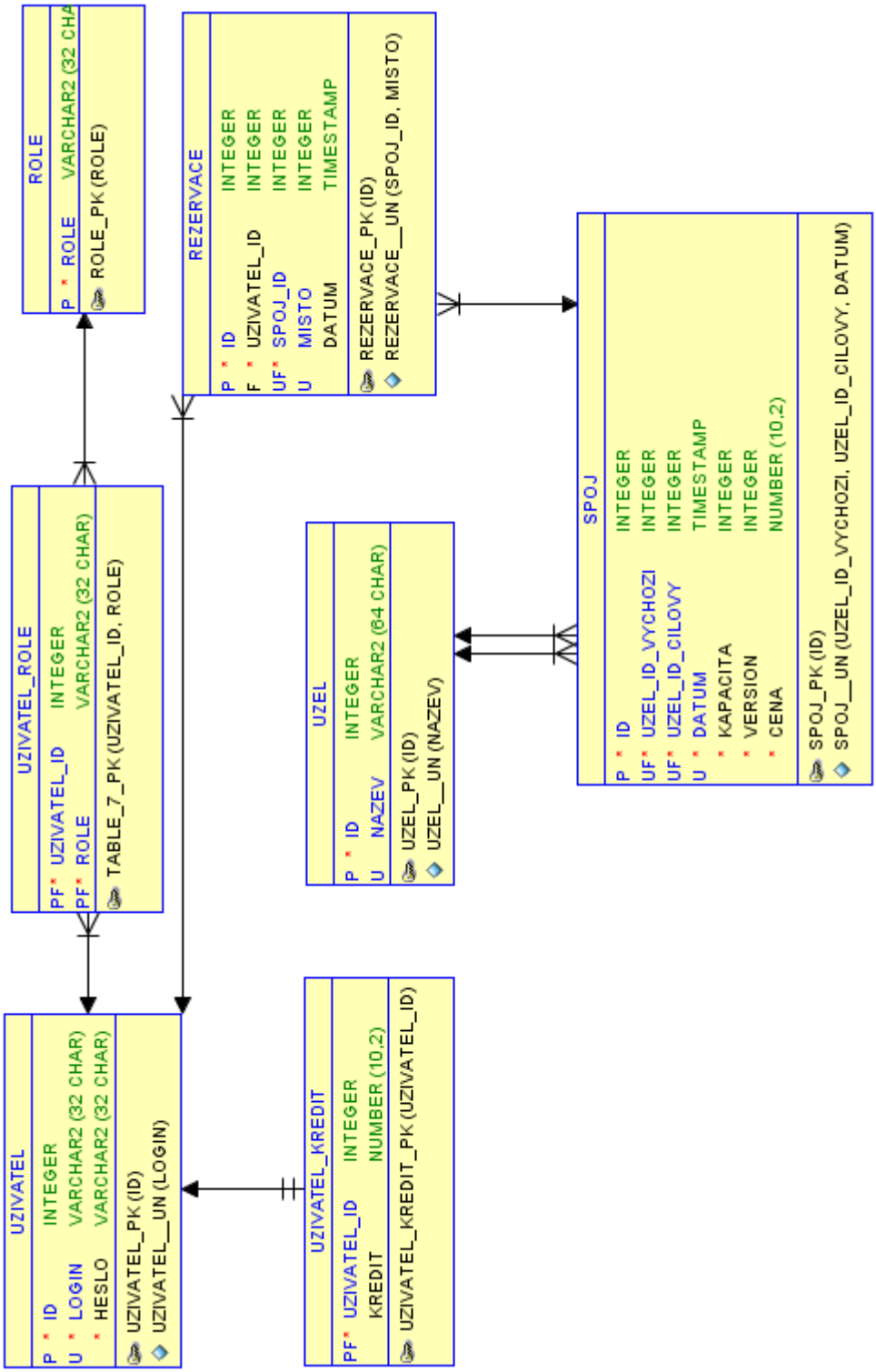
- identifikovat funkční požadavky na informační rezervační systém autobusové dopravy, výsledek prezentovat Use Case diagramem
- identifikovat datové požadavky systému, výsledek prezentovat Data modelem
- zajistit konzistenci dat vhodným mechanizmem řízení transakcí a konkurenčního přístupu k datům
- zajistit bezpečnost dat vhodným mechanizmem autentizace a autorizace klientů
- identifikovat problémy, které je vhodné řešit pomocí návrhových vzorů
- ORM mapování řešit pomocí Hibernate framework, respektive JPA
- implementaci business logiky presentovat diagramem tříd aplikační vrstvy
- pracovat s RDBMS
- pracovat s aplikačním modelem platformy Java EE
- aplikační logiku zpřístupnit pomocí tenkého klienta

4.2 Use Case diagram



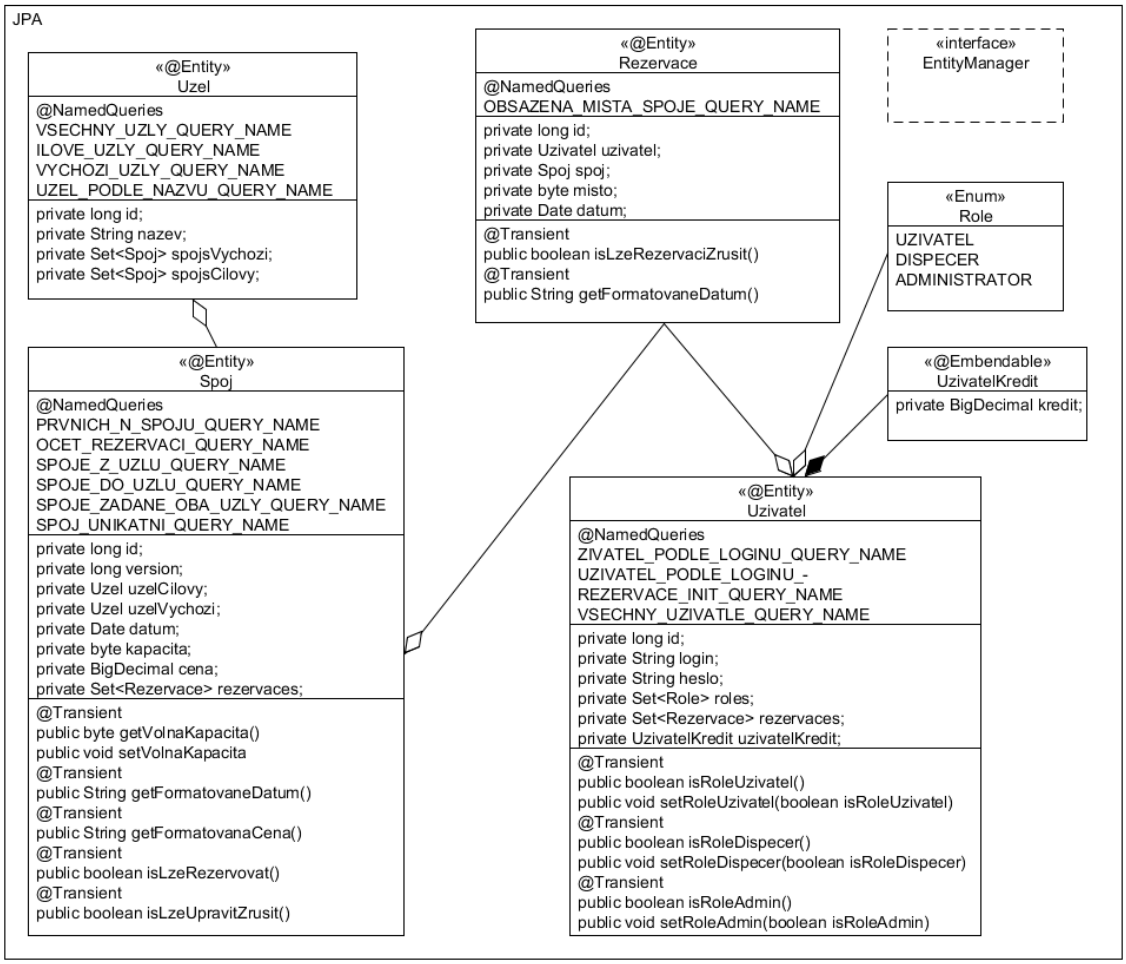
Obrázek 17: *Use case diagram aplikace.*

4.3 Datový model



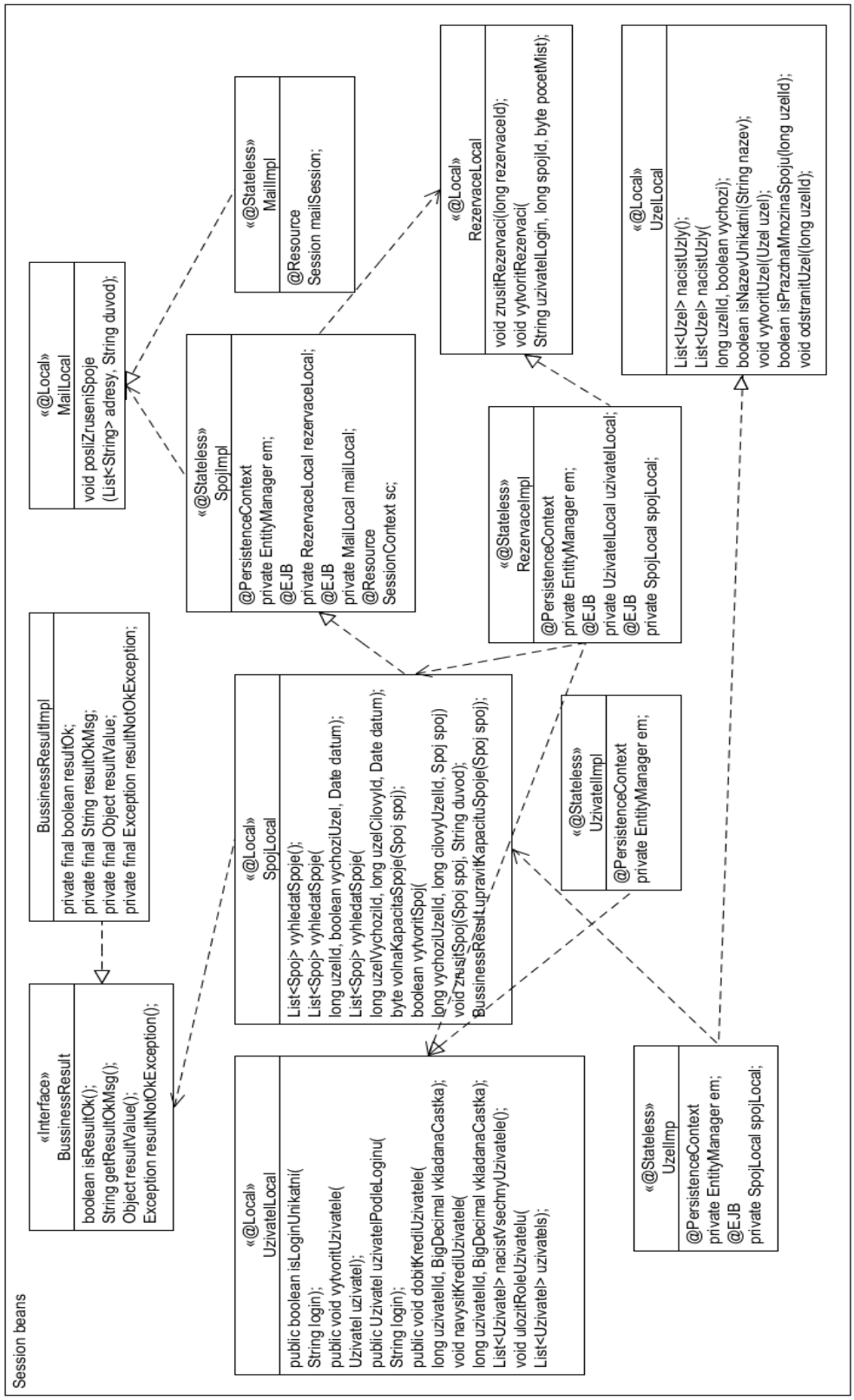
Obrázek 18: *Data model aplikace.*

4.4 Class diagram aplikační vrstvy



Obrázek 19: *Class diagram logické JPA vrstvy, která je součástí EJB kontejneru.*

Vztah mezi systémem na obrázku 19 a systémem na obrázku 20 lze zjednodušeně vyjádřit jako: `Session beans <<uses>> JPA`.

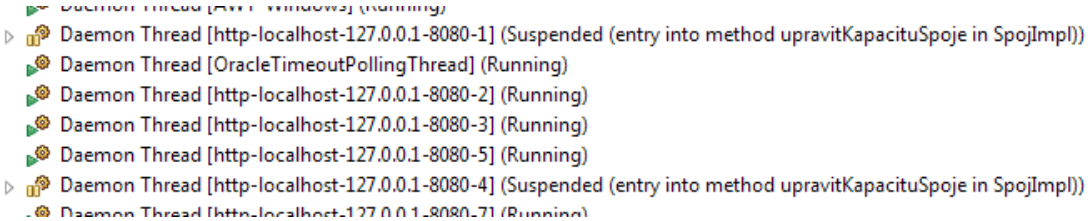


Obrázek 20: *Class diagram logické business session vrstvy, která je součástí EJB kontejneru.*

4.5 Testování aplikace

Vícevrstvou aplikaci lze podrobit několika různým typům testů, typicky se jedná o testy jednotkové, integrační, zátěžové, uživatelské, bezpečnostní, atd. Každý z těchto testů lze opakovat několikrát s jiným cílem. Nejčastějším cílem testů je odhalení logických chyb business logiky, odhalení úzkých míst, problémy s kešováním a pamětí, hledání deadlocks, atd. Reza Rahmann, člen expertní skupiny Java EE 6 a EJB 3.1, ve svých publikacích uvádí, že testování vícevrstvých aplikací je založeno na znalosti technologií a dostupných nástrojů. Moderním nástrojem pro testování např. EJB 3.1 je Arquillian.

Aplikace vytvořená v této práci nebyla žádným testům podrobena. Důvodem je, že vznikla jako praktická dokumentace použití technologií, která tato práce popisuje a proto pravděpodobně nebude na server nikdy nasazena. Počet uživatelů aplikace odhaduji na 3, přičemž s aplikací bude současně pracovat pouze jeden. Nicméně zadání práce požaduje aby aplikace pracovala s konkurenčním přístupem mnoha uživatelů k datům. K ověření správnosti implementace by bylo potřeba aplikaci nasadit na server a pomocí jednoho z nástrojů (např. JMeter) simulovat přístup mnoha uživatelů najednou. Takové testování by bylo časově náročné a pravděpodobně by přesahovalo rámec této práce. Konkurenčním přístup lze z části ověřit v IDE v debug módu pomocí dvou transakcí dvou různých uživatelů.



Obrázek 21: *Ladění konkurenčního přístupu.*

Na obrázku 21 jsou vidět dvě zastavená vlákna, která vykonávají metodu pro upravení kapacity spoje. Obě vlákna (resp. transakce, resp. uživatelé) mění tu samou optimisticky zamykanou entitu, vlákno, které skončí první (zavolá commit) propíše novou hodnotu do databáze, druhé vlákno vyhodí `OptimisticLockException`.

5 Zhodnocení

Celý textový rozsah této práce se věnuje teorii návrhu vícevrstvých aplikací a platformě Java EE. Důvodem je, že se jedná o poměrně komplexní téma, samotný referenční manuál platformy obsahuje cca 1000 stran. Praktická část v podobě aplikace je v práci zastoupena pouze UML diagramy. Samotnou implementaci aplikace dokumentuje její zdrojový kód, opatřený důkladnými komentáři, který čtenář najde v příloze.

5.1 Teoretická část práce

První kapitola práce se věnuje vícevrstvé architektuře. Obsahuje doporučení jak postupovat při návrhu, představuje některé možnosti řešení, shrnuje jejich výhody a nevýhody. Detailně popisuje nejběžnější případ vícevrstvé architektury – třívrstvou architekturu. První kapitola se zmiňuje i o problematice nasazení aplikací do cílového prostředí, což praktická část práce nepokrývá.

Dostupné technologie definuje použití platformy Java EE o které ve zkratce hovoří druhá kapitola. Druhá kapitola je rozdělena do částí přibližně tak, jak je rozděluje referenční manuál Java EE 6 společnosti Oracle. Tento manuál je také primárním zdrojem informací celé druhé kapitoly. Tato kapitola v podstatě poskytuje výtah toho nejdůležitějšího z referenčního manuálu v českém jazyce. Pokud mi nějaké části přišli nesrozumitelné nebo příliš stručné čerpal jsem i z jiných publikací.

Třetí kapitola velmi stručně hovoří o Java EE aplikačních serverech a krátce představuje nejpopulárnější z nich. O službách, které aplikační server komponentám aplikace poskytuje hovoří druhá kapitola.

Zadání dále požaduje věnovat pozornost návrhovým vzorům. Návrhové vzory jsou zmiňovány napříč celou touto prací, ale detailnější popis, či ukázka implementace by překračovaly rozsah této práce. Nutno podotknout, že použití některých návrhových vzorů je přímo vynuceno použitou technologií, např. fasáda, dependency injection, data transfer object, atd.

Teoretická část práce se nezmiňuje o Hibernate frameworku, protože objektově-relační mapování aplikace není tak docela řešeno pomocí Hibernate, jak požaduje zadání, ale pomocí JPA. Důvodem je opět rozsah práce, JPA je značně jednodušší než Hibernate framework. Hibernate framework poskytuje širší množinu funkcí než JPA, ale aplikace využívající JPA jsou přenositelné mezi jednotlivými ORM frameworky, jako je Hibernate, což je podle mého názoru větší výhoda. V implicitní konfiguraci je na aplikačním serveru JBoss JPA specifikace implementována právě Hibernate frameworkem, proto výraz „docela“ v úvodní větě odstavce.

Z tohoto pohledu práce podle mého názoru splňuje analýzu požadavků, architektury a dostupných technologií.

5.2 Praktická část práce

Aplikace je modelována pomocí jazyka UML. Data jsou uložena v relační databázi, použitý typ RDBMS je Oracle11g XE. DDL script je součástí přílohy. Aplikační server je nejnovější JBoss ve verzi 7.1. Vývoj probíhal na nejnovější verzi Javy – JDK7. K aplikaci se přistupuje pomocí tenkého klienta. Autentizace probíhá způsobem specifickým pro JBoss AS 7. Autorizace probíhá na úrovni aplikační vrstvy pomocí JAAS. Technologie prezentační vrstvy je JSF. Technologie použitá pro vytvoření view jsou Facelets. Výstupní XHTML stránky jsou stylovány pomocí Twitter Bootstrap CSS knihovny. Uživatelské vstupy jsou validovány pomocí Bean Validation (JSR-303). Business logika je zapouzdřena v Stateless Session Beans a zpřístupňována pomocí lokálních business rozhraních (fasád). Controller získává referenci na instanci SSB pomocí dependency injection (anotací `@EJB` příslušné fieldu odpovídajícího typu). Business entity jsou pomocí JPA mapovány do databáze.

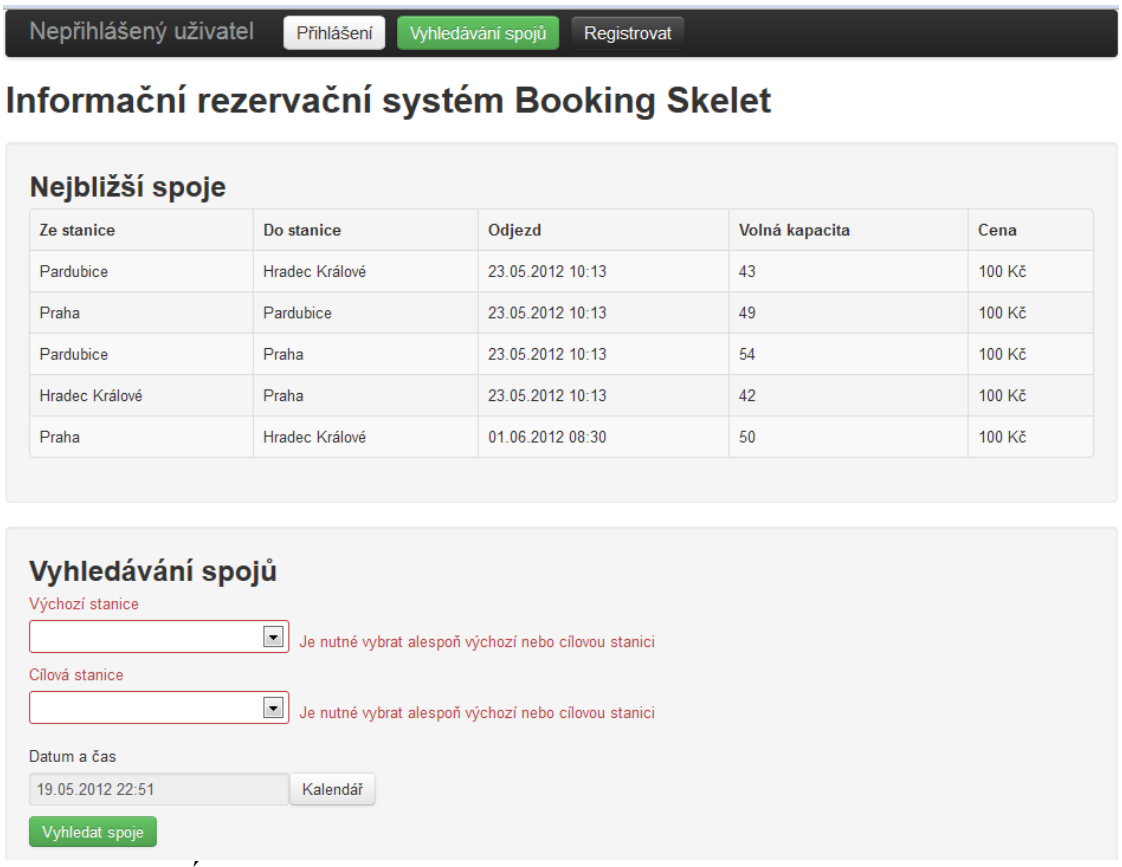
Konkurenční přístup je pro názornost řešen třemi různými způsoby. Editace kapacity spoje je zamykána optimisticky, pomocí atributu `@Version`. Editace rolí uživatele je zamykána pesimisticky. Díky nedostatečné analýze a nevhodnému datovému návrhu je nutné také zamykat vytváření rezervací, aby nedošlo k překročení kapacity spoje a to zamčením celé tabulky pro zápis po dobu trvání transakce. JPA takový režim zamykání nepodporuje, proto je nutné využít native query, který databázi poví, že má tabulku zamknout. Tento postup sice zaručuje konzistenci dat, ale v produkčním prostředí s mnoha uživateli by zcela jistě znamenal problém škálovatelnosti. V aplikaci jsem tento postup ponechal jako varování a jako ukázku použití specifické databázové funkce, čtenář v komentáři najde jednoduchý návrh opravy problému.

Z tohoto pohledu podle mého názoru aplikace splňuje systémové požadavky.

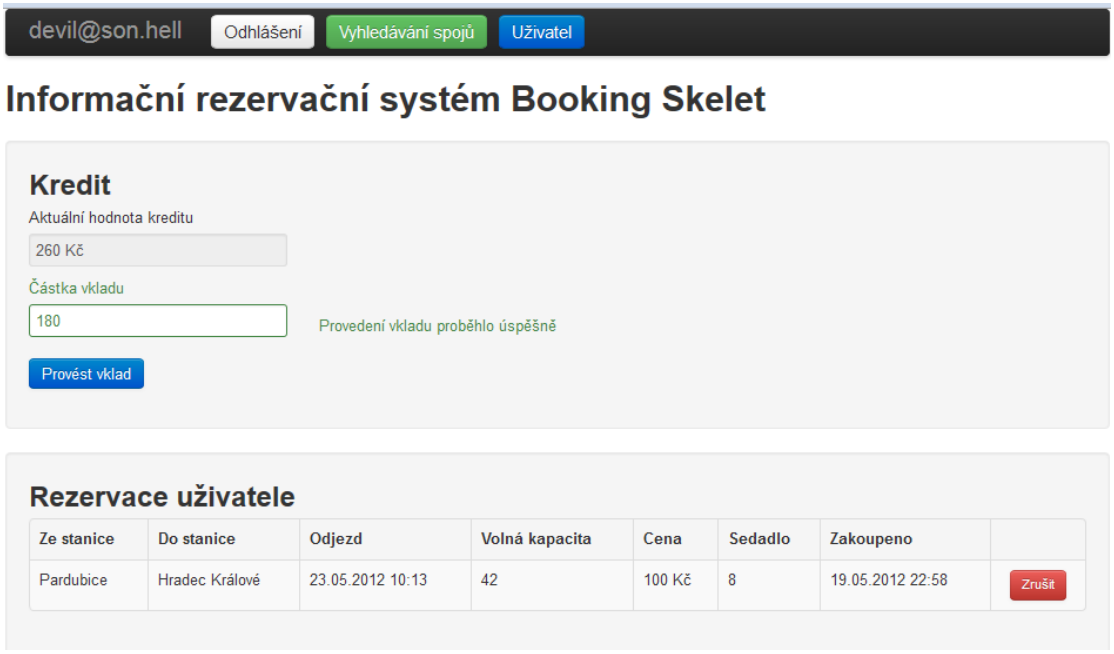
Vzhledem k tomu, že aplikace vznikla zejména pro demonstraci použití platformy Java EE, neexistuje analýza funkčních požadavků. Funkční požadavky reprezentované Use case diagramem vznikly okopírováním a zjednodušením běžících komerčních systémů. Aplikace splňuje požadavky na správu, autentizaci a autorizaci uživatelů. Dispečer vytváří, edituje a ruší spoje mezi uzly. Uživatel rezervuje a ruší rezervace míst na spojích. Aplikace neobsahuje správu vozového parku a v tomto bodě nesplňuje zadání. Důvodem je značná složitost, kterou smysluplná implementace správy vozového parku skrývá a časové důvody. V souvislosti s touto funkcionalitou by bylo nutné řešit např. tyto otázky: (implementací by se z aplikace stal „fleet management system“)

- Je vůz volný? V kterém uzlu?
- Jak dlouho trvá přejezd z uzlu A do uzlu B?
- Nebude v uzlu A vůz později chybět?
- Jak vypadá graf sítě?

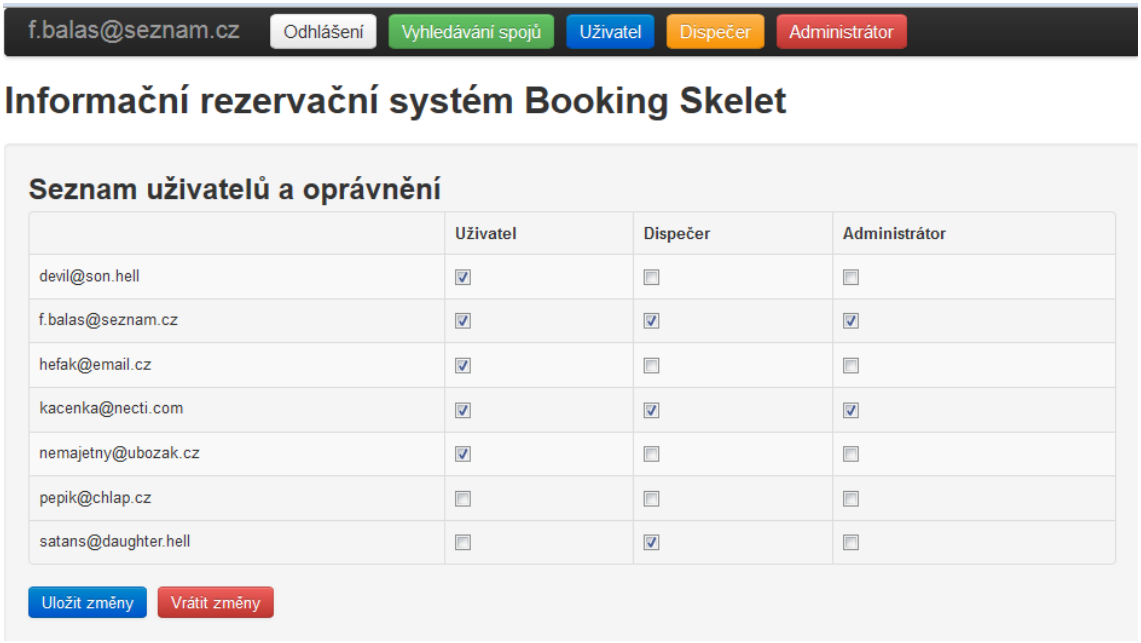
5.2.1 Ukázka aplikace



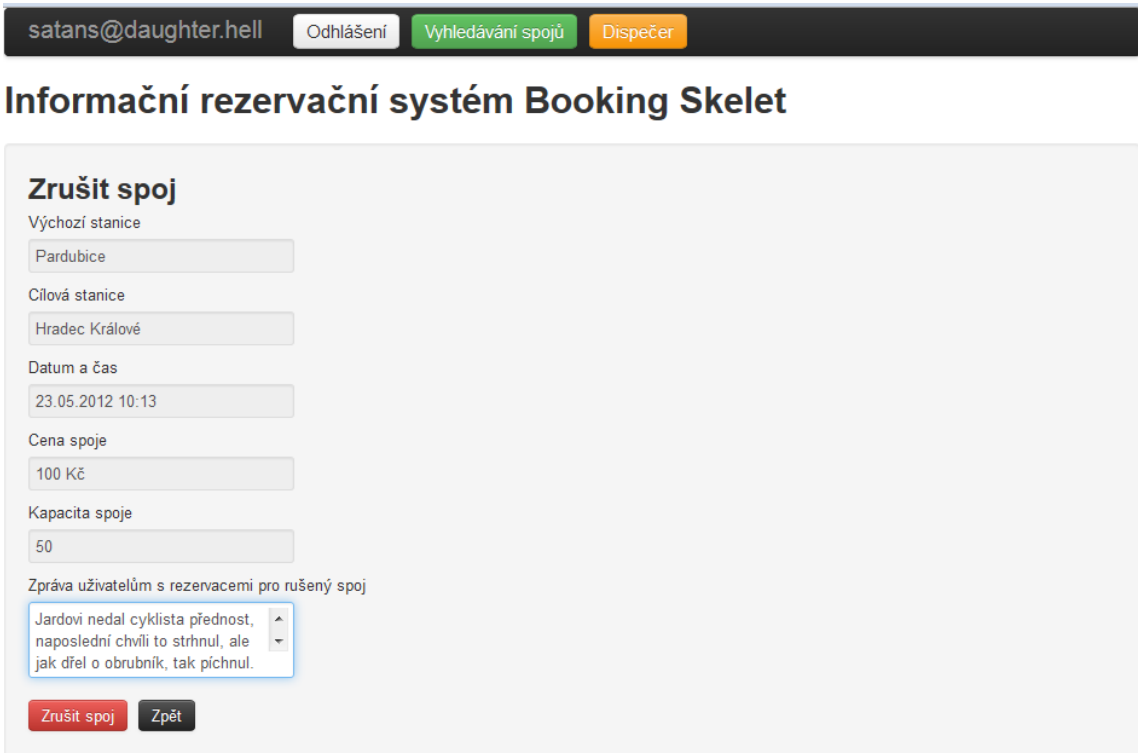
Obrázek 22: *Úvodní stránka aplikace, anonymní uživatel, chybná validace formuláře pro vyhledávání spojů.*



Obrázek 23: *Stránka uživatele, přihlášený uživatel, úspěšná validace.*



Obrázek 24: *Stránka administrátora, uživatel ve všech rolích.*



Obrázek 25*: Zrušení spoje, přihlášený uživatel v roli dispečera.*

Seznam použité literatury

[1] Clever and smart [online]. 2008 [cit. 2011-12-04]. *Vícevrstvá architektura*. *Vícevrstvá architektura: popis vrstev*. Dostupné z WWW: <http://www.cleverandsmart.cz/vicevrstva-architektura-popis-vrstev/>.

[2] Clever and smart [online]. 2008 [cit. 2011-12-04]. *Vícevrstvá architektura*. *Vícevrstvá architektura: výhody a nevýhody*. Dostupné z WWW: <http://www.cleverandsmart.cz/vicevrstva-architektura-vyhody-a-nevyhody/>.

[3] Microsoft [online]. 2012 [cit. 2012-1-8]. *Microsoft Application Architecture Guidelines*. *Chapter 5: Layered Application Guidelines*. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee658109.aspx>.

[4] Clever and smart [online]. 2008 [cit. 2011-12-26]. *Vícevrstvá architektura*. *Vícevrstvá architektura: tenký, tlustý a chytrý klient*. Dostupné z WWW: <http://www.cleverandsmart.cz/vicevrstva-architektura-tenky-tlusty-a-chytry-klient/>.

[5] Microsoft [online]. 2011 [cit. 2011-12-26]. *Microsoft Application Architecture Guidelines*. *Chapter 6: Presentation Layer Guidelines*. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee658081.aspx>.

[6] Microsoft [online]. 2012 [cit. 2012-1-21]. *Microsoft Application Architecture Guidelines*. *Chapter 7: Business Layer Guidelines*. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee658103.aspx>.

[7] Microsoft [online]. 2012 [cit. 2012-1-26]. *Microsoft Application Architecture Guidelines*. *Chapter 8: Data Layer Guidelines*. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee658127.aspx#PatternMap>.

[8] Microsoft [online]. 2012 [cit. 2012-1-30]. *Microsoft Application Architecture Guidelines*. *Chapter 17: Crosscutting Concerns*. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee658105.aspx#StateManagement>.

[9] Microsoft [online]. 2012 [cit. 2012-1-30]. *Microsoft Application Architecture Guidelines*. *Chapter 16: Quality Attributes*. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee658094.aspx>.

[10] *The Java EE 6 Tutorial.* 500 Oracle Parkway, Redwood City, CA 94065, U.S.A :Oracle Corporation. 2011. 906 s. Part No: 821–1841–13.

[11] Andrew Lee Rubinger, Bill Burke. *Enterprise Java Beans 3.1*. 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., 2010. 764 s. ISBN 978-0-596-15802.

[12] Christian Bauer, Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830 : Manning Publications Co., 2007. 876 s. ISBN 1-932394-88-5.

[13] Red Hat [online]. 2012 [cit. 2012-5-1]. *JBoss Enterprise Application Platform features*. . Dostupné z WWW: <http://www.redhat.com/resourcelibrary/articles/jboss-enterprise-application-platform-features>.

[14] Francesco Marchioni. *JBoss AS 7 Configuration, Deployment,and Administration*. Livery Place 35, Livery Street, Birmingham B3 2PB, UK : Packt Publishing, 2011. 380 s. 978-1-84951-678-5.

Seznam obrázků

Příloha č. 1 – DVD-ROM

DVD-ROM obsahuje:

- zdrojový kód aplikace
- Use case diagram
- Data model
- Class diagram
- DDL script
- Eclipse IDE for Java developers s nainstalovanými pluginy
- aplikační server JBoss AS 7.1 s nastavením a moduly
- Apache Maven