VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

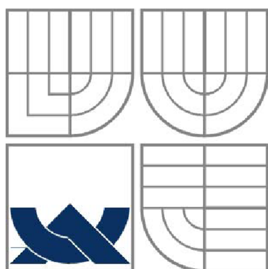
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKAČNÍ RÁMCE PRO VÝVOJ WEBOVÝCH APLIKACÍ V JAVĚ

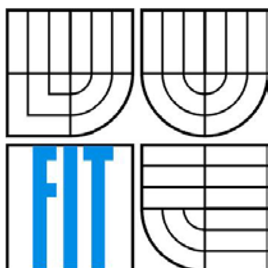
DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE Bc. Tomáš Tulka
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKAČNÍ RÁMCE PRO VÝVOJ WEBOVÝCH APLIKACÍ V JAVĚ

JAVA WEB APPLICATION FRAMEWORKS

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE Bc. Tomáš Tulka
AUTHOR

VEDOUCÍ PRÁCE Doc. Ing. Jaroslav Zendulka, CSc.
SUPERVISOR

BRNO 2009

Abstrakt

Aplikační vývojové rámce jsou nástroje v mnohém usnadňující vývoj aplikací a zvyšující jejich kvalitu. Aplikačních vývojových rámců pro webové aplikace v jazyce Java je obrovské množství, a není snadné zvolit ten pravý rámec relevantní pro konkrétní aplikaci. Tato práce rozebírá vývojové rámce z obecného pohledu, popisuje jejich typické funkce i funkce, kterými se jednotlivé rámce liší, a usnadňuje tak uživateli výběr vhodného rámce pro jeho aplikaci. Dále názorně rozbírá zvolené typické představitele rámců pro jistý okruh použití, demonstruje jejich funkčnost pomocí jednotné typizované ukázkové aplikace a srovnává je z praktického hlediska.

Klíčová slova

Vývojové rámce, Java EE, EJB, MVC architektura, informační systém, webová aplikace

Abstract

Application frameworks are tools that enable easier development of web applications and enhance their quality. Because there are so may Java web application frameworks in existence it is difficult at times to choose the correct one to suit a particular application. This thesis examines frameworks from a general point of view. Descriptions of the characteristic functions of frameworks are also covered, along with functions, and the ways in which they differ from each other; and thus it enables the most appropriate framework to be chosen. The thesis also analyses typical representatives of frameworks for certain uses, it demonstrates their activity by means of standardized applications and compares them in terms of practical use.

Keywords

Application frameworks, Java EE, EJB, MVC architecture, information system, web applications

Citace

Tulka Tomáš: Aplikační rámce pro vývoj webových aplikací v Javě. Brno, 2009, diplomová práce, FIT VUT v Brně.

Aplikační rámce pro vývoj webových aplikací v Javě

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Ing. Jaroslav Zendulky, CSc.
Další informace mi poskytl Mgr. Marek Rychlý.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Tomáš Tulka
7. 5. 2009

Poděkování

Velice děkuji vedoucímu mé práce Doc. Ing. Jaroslav Zendulkovi, CSc. za neocenitelnou odbornou pomoc, dobré rady a ochotu, kterou mi poskytl.

© Tomáš Tulka, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah ... 1
1 Úvod ... 4
2 Technologie Java EE ... 5
2.1 Součásti Java EE ... 5
2.2 Java servlety ... 6
2.2.1 Java Servlet API ... 6
2.3 JavaServer Pages (JSP) ... 7
2.4 JavaServer Faces (JSF) ... 8
2.5 Enterprise JavaBeans (EJB) ... 8
2.5.1 Session Beany ... 9
2.5.2 Entity Beany ... 9
2.5.3 Message-Driven Beany ... 10
2.6 Správa stavů ... 10
2.7 Architektury Java EE ... 10
2.7.1 Vrstva persistence dat ... 11
2.7.2 Aplikační vrstva (střední vrstva) ... 11
2.7.3 Prezentační vrstva ... 11
2.7.4 Ne-distribuované architektury ... 12
2.7.5 Distribuované architektury ... 13
2.8 Testování Java EE aplikace ... 14
3 Webové aplikační rámce ... 16
3.1 Výhody používání frameworků ... 16
3.2 Nástroje a služby poskytované frameworky ... 18
3.3 Srovnávání frameworků ... 20
3.3.1 Aplikační logika ... 20
3.3.2 Srovnávání pomocí matice ... 21
3.4 Katalog frameworků ... 22
3.4.1 Komplexní aplikační frameworky ... 22
3.4.2 Frameworky zaměřené na prezentaci ... 24
3.4.3 Aplikačně specifické frameworky ... 24
4 JSF framework ... 26
4.1 Technologie JavaServer Faces (JSF) ... 26
4.2 Výhody JSF technologie ... 27
4.3 Aplikace JSF ... 27

4.4 MVC architektura ve frameworku JSF 28
4.5 Role ve frameworku JSF 28
4.6 Vytváření aplikací v JSF 29
4.6.1 Kroky ve vývojovém procesu 29
4.6.2 Mapování instance FacesServlet 29
4.6.3 Vytváření stránek 30
4.6.4 Definice navigace stránek 31
4.6.5 Vytváření backing beanů 32
4.6.6 Vložení řídících beanů (managed bean) do konfiguračního souboru 33
4.7 Životní cyklus stránky JSF 33
4.8 Internacionalizace dat 35
4.8.1 Používání zdrojových svazků 35
4.8.2 Internacionalizace statických dat 36
4.8.3 Internacionalizace chybových zpráv 36
4.9 Standardní validátory 36
4.10 Standardní konvertery 37
4.11 Naslouchače událostí (event listeners) 38
4.12 JavaServer Pages Standard Tag Library (JSTL) 38
4.12.1 Tag spolupráce 38
4.12.2 Tag podpory proměnných 39
4.12.3 Tagy řízení toku programu 39
4.13 Vytváření vlastních komponent UI 40
4.13.1 Kroky pro vytvoření vlastní komponenty 40
4.14 Konfigurační soubor JSF aplikace 41
5 Struts framework 43
5.1 Deklarativní přístup 43
5.1.1 Deklarace založené na XML 43
5.1.2 Deklarace založené na Java anotacích 44
5.2 MVC komponentový model rámce Struts 45
5.2.1 Komponenty řadiče 46
5.2.2 Komponenty modelu 47
5.2.3 Komponenty pohledu 51
5.3 Správa výjimek 55
5.3.1 Výjimky v Javě 56
5.3.2 Správa výjimek uvnitř frameworku Struts 56
5.4 Internacionalizace 57
5.4.1 Podpora internacionalizace v Javě 58

5.4.2 Internacionalizace ve Struts frameworku 59
5.5 Validace vstupních dat 60
5.5.1 Vytváření validátorů 60
5.5.2 Registrace validátorů 62
6 Ukázková aplikace 64
6.1 Určení požadavků 64
6.1.1 Neformální specifikace 64
6.1.2 Analýza požadavků 65
6.1.3 Business model požadavků 65
6.2 Specifikace požadavků 68
6.2.1 Diagram případů užití 68
6.2.2 Strukturovaný zápis některých případů použití 69
6.3 Implementace 71
6.3.1 Implementace datového modelu v architektuře MVC 71
6.3.2 Implementace v JSF frameworku 75
6.3.3 Implementace ve Struts 2 frameworku 82
7 Srovnání a zhodnocení 89
7.1 MVC přístup 89
7.2 Aplikační logika 89
7.3 Prezentační vrstva 90
7.4 Srovnání pomocí matice 90
8 Závěr 93
Literatura 94
Seznam příloh 95

1 Úvod

Tato diplomová práce se zabývá aplikačními vývojovými rámci (frameworky) pro webové aplikace v jazyce Java z hlediska jejich srovnání a výběru pro řešení konkrétních specifických projektů.

Druhá kapitola je úvodem do problematiky webových aplikací v jazyce Java, popisuje stručně technologii Java EE, její specifika, jednotlivé vrstvy a metody testování, dále technologie Java servletů, JSP stránek a Enterprise JavaBeans (EJB). Znalosti z této kapitoly jsou vyžadovány pro správné pochopení zbytku práce.

Třetí kapitola se zaměřuje obecně na webové frameworky, zabývá se jejich výhodami a nevýhodami, možnostmi, specifiky, chováním a poskytovanými funkcemi, dále postupy srovnávání frameworků a konečně katalogovým popisem některých vybraných frameworků v Javě.

Následující dvě kapitoly se již věnují zvoleným dvěma frameworkům podrobně. Jedná se o JavaServer Faces framework (JSF) a Struts framework. U druhého jmenovaného se tato práce zaměřuje na současně nejnovější dostupnou verzi, tedy Struts 2, v úvahu však bere i dřívější verze Struts 1.x, porovnává obě verze z popisovaných hledisek a komentuje změny a zlepšení v nové verzi. Tyto informace vychází z praktické zkušenosti s oběma verzemi Struts získané při vývoji ukázkových aplikací. Většina komentovaných příkladů a ukázek zdrojových kódů při popisu funkcionality zvolených frameworků čerpá právě z těchto ukázkových aplikací.

JSF framework byl zvolen proto, že se jedná o standard, z něhož vychází množství dalších ryze prezentačních frameworků, a Struts byl vybrán na základě jeho historického vlivu na webové frameworky v Javě a také proto, že je v současnosti prakticky nejpoužívanější.

V šesté kapitole je nastíněn postup při vývoji ukázkových aplikací při uplatnění správných postupů softwarového inženýrství, jsou popsány důležité aspekty při vývoji v jednotlivých frameworcích a vývojové milníky dosažené během fáze implementace.

Zhodnocením dosažených výsledků se zabývá sedmá kapitola obsahující srovnání zvolených frameworků.

2 Technologie Java EE

Java Platform, Enterprise Edition (dále jen **Java EE**, dříve označovaná jako Java 2 Enterprise Edition nebo J2EE) je součást platformy Java určená pro vývoj a provoz podnikových aplikací a informačních systémů. Základem pro platformu Java EE je platforma Java SE, nad ní jsou definovány součásti tvořící Java EE [1].

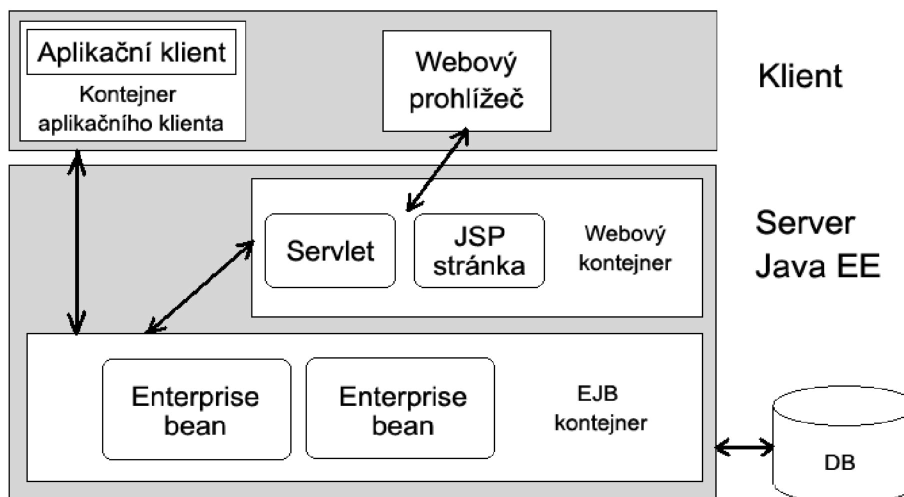
Aktuální verze Java EE je Java Platform, Enterprise Edition 5 - Java EE.

Java EE aplikace bývá pro usnadnění vývoje často implementována pomocí nějakého aplikačního rámce – frameworku. Webové aplikační rámce jsou hlavním předmětem zájmu této práce.

2.1 Součásti Java EE

Součástí platformy Java EE jsou především specifikace pro:

- **vývoj webových aplikací** - Servlety, Java Server Pages (JSP), JavaServer Faces (JSF)
- **vývoj sdílené business logiky** - Enterprise Java Beans (EJB), Spring framework
- **přístup k legacy systémům** - Java Connector Architecture (JCA), Hibernate
- **přístup ke zprávovému middleware** - Java Messaging Services (JMS)
- **podpora technologií Web Services**



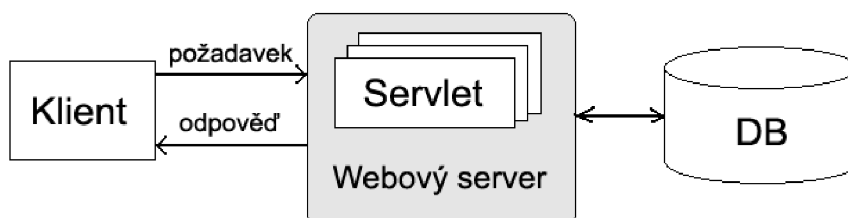
Obrázek 2-1: Architektura aplikace v Java EE

Obrázek 2-1 znázorňuje architekturu server-klient typickou pro aplikace Java EE. Klientem může být aplikace Java SE (tzv. tlustý klient) nesoucí v sobě veškerou aplikační logiku a komunikující přímo s datovým modelem (zde implementován pomocí technologie Enterprise

JavaBean – EJB), nebo jím může být webový prohlížeč komunikující s webovým kontejnerem na straně serveru (implementovaná servletem nebo stránkou JSP).

2.2 Java servlety

Java servlety (dále jen **servlety**) jsou aplikace běžící na straně serveru pod platformou Java jako program JVM (Java Virtual Machine). Servlety jsou objekty, které implementují princip požadavek/odpověď. Na základě obdržení žádosti jako svůj výstup dynamicky generují HTML, XML a podobně [2].



Obrázek 2-2: princip požadavek/odpověď zpracovávaný servlety webového serveru

Servlety mohou být vytvářeny automaticky technologií JavaServer Pages (viz podkapitola *JavaServer Pages*).

Servlety jsou přenositelné, obsahují prostředky pro správu sezení, komunikaci mezi klientem a webovým serverem, a jsou tak komplexním nástrojem pro tvorbu webových aplikací.

Servlet v aplikacích Java EE musí implementovat rozhraní nacházejí se v balíčcích označovaných jako Java Servlet API.

2.2.1 Java Servlet API

Java Servlet API je množina Java tříd a rozhraní definující rozhraní mezi klientem a serverem.

Java Servlet API se skládá z dvou balíčků:

- `javax.servlet`
- `javax.servlet.http`

Základný balíček tříd `javax.servlet` obsahuje abstraktní třídy a rozhraní pro tvorbu všeobecných servletů pro různé protokoly, záleží na konkrétní implementaci servletu. Balíček `javax.servlet.http` rozšiřuje funkcionalitu základního balíčku a přidává podporu protokolu HTTP.

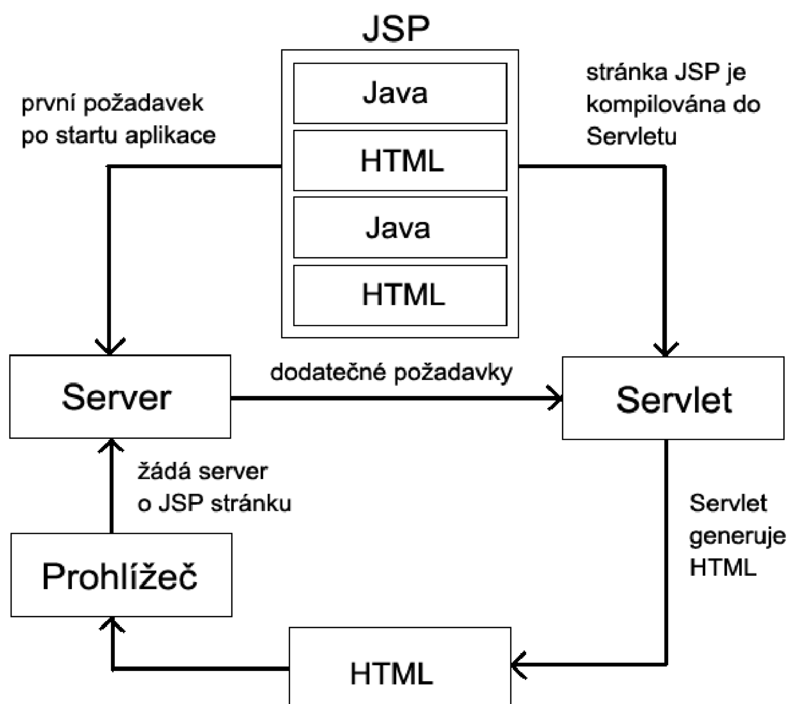
2.3 JavaServer Pages (JSP)

JavaServer Pages (dále jen **JSP**), stejně jako servlety, jsou soustředěny na dynamické generování obsahu webových stránek (obvykle v HTML) a, stejně jako servlety, poskytují komplexní prostředky pro tvorbu webových aplikací pod platformou Java.

Motivací pro vznik technologie JSP byl fakt, že většina dynamicky generovaných stránek pomocí servletů obsahovala ve většině případů více vypisovaného HTML než vlastního kódu servletu, což vedlo k nepřehlednosti a znesnadnění práce vývojářům. JSP pak tuto vadu odstranily způsobem psaní Java kódu pomocí procedurálních značek přímo do kódu HTML, XML, apod. JSP stránky jsou tedy dokumenty, založené na HTML obsahující bloky Java kódu, zvané *skriptlety*.

JSP jsou následně kompilovány do servletů pomocí JSP kompilátoru.

Nevýhodou může být větší množství kódu ve vygenerovaném servletu, což může mít za následek nižší výkonnost.



Obrázek 2-3: nasazení technologie JSP pro webové aplikace

Obrázek 2-3 znázorňuje vzájemnou kooperací jednotlivých prvků v aplikaci Java EE. Stránka JSP (složená z Java a HTML kódu) je při prvním požadavku na server po spuštění aplikace zkompilována do Objektu Java servletu, který posléze generuje HTML kód pro webový prohlížeč žádající JSP stránku.

2.4 JavaServer Faces (JSF)

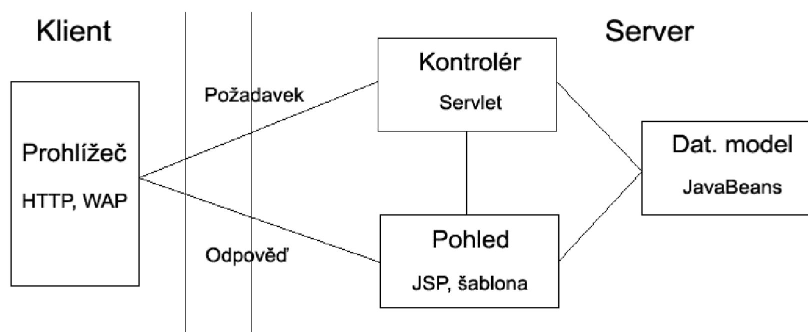
Relativně nová technologie JavaServer Faces (dále jen **JSF**) je navržena pro jednoduchou tvorbu webových aplikací z hlediska uživatelských rozhraní pomocí komponent a jejich propojení na business objekty. Automatizuje také proces navigace (akce napříč stránkami) a validace vstupu (formát vstupu, minimální/maximální délka, apod.). U komponent je možné reagovat na události pomocí naslouchačů událostí (např.: `valueChangeListener` – provádí určenou akci při události změny obsahu komponenty) nebo vypisovat chybové či informační zprávy.

JSF je doplňková technologie, která má využití pouze ve spojení s dalšími Java EE technologiemi (JSP, Servlety, EJB), případně frameworky (Struts, Seam, ...).

JSF, podobně jako většina webových frameworků, využívá návrhový vzor Model-View-Controller (dále jen **MVC**). Model může být tvořen jednoduchými Java objekty (plain old Java objects – POJOs), EJB, apod.; pohled (View) tvořící uživatelské rozhraní bývá stránka JSP; řadič (Controller) je většinou implementovaný jako servlet.

Výhodou je, že každá z částí modelu MVC může být vyvíjena a spravována zvlášť, nezávisle na ostatních (např. pokud pohled tvořený stránkou JSP obsahuje chyby, tato skutečnost se nijak neprojeví v řadiči nebo datovém modelu). K dalším výhodám MVC přístupu patří možnost odděleného testování v rámci každé vrstvy MVC.

JSF se tato práce věnuje podrobněji v kapitole *Webové aplikační rámce*.

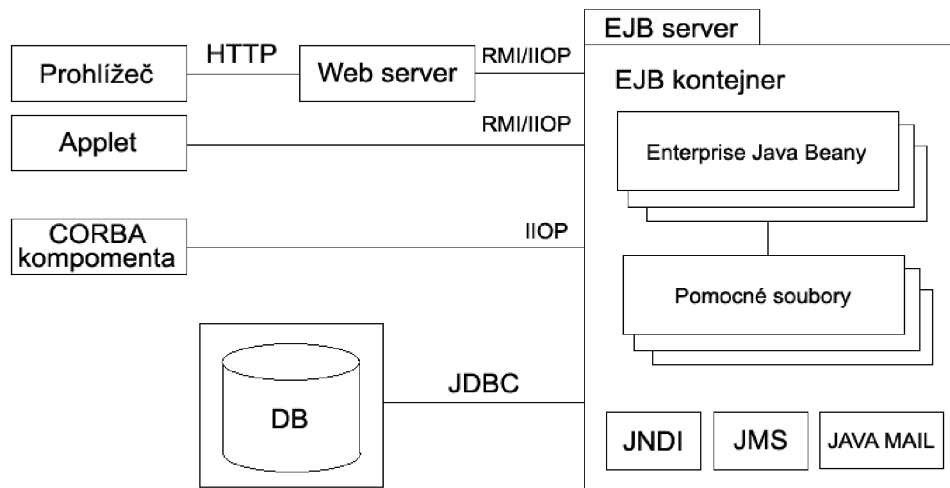


Obrázek 2-4: Model-View-Controller návrhový vzor v JSF

2.5 Enterprise JavaBeans (EJB)

Specifikace Enterprise JavaBeans (dále jen **EJB**) je komponentový model pro vývoj, rozmístění a vzájemné propojení znovupoužitelných komponent na straně serveru.

EJB existuje ve třech druzích: session beany, entity beany a message-driven beany.



Obrázek 2-5: Schéma EJB architektury a umístění v Java EE aplikaci

Obrázek 2-5 ukazuje možné nasazení Java EE aplikace a umístění EJB serveru v této aplikaci. EJB server obsahující EJB kontejner s beanami komunikuje s webovým serverem, appletem (oba obsahující aplikační logiku) nebo komponentou CORBA (middleware) a zprostředkovává tak data z databáze, se kterou komunikuje skrze rozhraní JDBC.

Podkapitola 2.5 čerpá ze zdroje [3].

2.5.1 Session Beany

Session beany jsou vytvářeny klientem a obvykle existují pouze během jednoho sezení komunikace mezi klientem a severem. Session beany vykonávají operace, jako výpočet a přístup k databázi.

Rozlišujeme dva typy session bean, které jsou určeny jejich užitím v interakci s klientem:

- **Bezstavové:** Nedeklarují žádné proměnné na úrovni třídy, jejich metody tedy pracují pouze s lokálními parametry.
- **Stavové:** Tyto beany mohou udržovat klientský stav napříč volání metod díky instancím proměnných deklarovaných na úrovni třídy. Klient nastaví hodnoty těchto proměnných, které potom může používat i v jiných metodách.

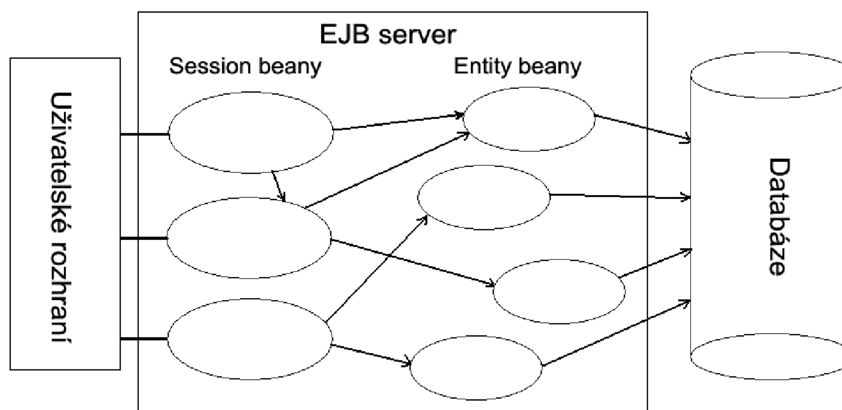
Session beany jsou vhodné pro implementaci business logiky, procesů a řízení toku programu.

2.5.2 Entity Beany

Entity beany jsou objekty navrhované na základě objektově-orientovaných postupů, používané v aplikaci jako objekty. Entity beany obsahují data vytvářející nějaký datový sklad, obvykle relační databázi.

Metody entity bean jsou typicky volány prostřednictvím session bean v případě, že tyto vyžadují přístup k datům.

Entity beany reprezentují perzistentní data v EJB aplikaci.



Obrázek 2-6: Session a entity beany v aplikaci

2.5.3 Message-Driven Beany

Message-driven beany jsou používány pro příjem asynchronních zpráv z jiných systémů do aplikace založené na EJB. Asynchronní zprávy mezi dvěma systémy mohou být převedeny na události vyvolané v uživatelském rozhraní a zachyceny naslouchačem událostí na stejném JVM.

2.6 Správa stavů

Podstatnou součástí architektury Java EE je udržování stavu na straně serveru. Tato vlastnost určuje chování aplikace na základě obsahu klastrů serveru. Je důležité rozhodnout, zda bude aplikace využívat stavů serveru, neboť při běhu na více serverech musí být informace o stavu replikována do klastrů všech serverů, což má vliv na výkon a rozšiřitelnost aplikace – aplikace, které nepracují s udržováním stavů jsou mnohem více rozšiřitelné z pohledu vývojáře.

Java EE poskytuje dva standardy pro udržování stavů ve webových aplikacích: **HTTP sessions** objekty spravované webovým kontejnerem a **stavové EJB sessions** objekty spravované kontejnerem EJB pamatující si stav všech akcí vykonaných daným klientem mezi jednotlivými voláními metod tímto klientem.

2.7 Architektury Java EE

Veškeré architektury Java EE uplatňují princip tří hlavních vrstev, některé navíc uvádějí dodatečné dělení uvnitř prostřední vrstvy.

Základní myšlenka dělení na vrstvy je taková, že veškeré akce uvnitř vrstvy mohou mít vliv na změny pouze v rámci této vrstvy. Například změny v databázi by neměly vyžadovat změnu v uživatelském rozhraní apod.

2.7.1 Vrstva persistence dat

Tato vrstva pracující s informačními zdroji dat, které Java EE aplikace využívá, zahrnuje systém řízení báze dat (Database Management Systém) – dále jen **SŘBD**. Data se ukládají do nějakého perzistentního úložiště, nejčastěji je tímto úložištěm relační databáze. Pro komunikaci se používá obvykle JDBC, často ve spojení s nějakým nástrojem typu ORM (Objektově-relační mapování), například Hibernate nebo Java Persistence API.

2.7.2 Aplikační vrstva (střední vrstva)

Aplikační vrstva zajišťuje vlastní funkcionalitu programu; obsahuje tzv. business logiku aplikace, zprostředkovává přístup ke zdrojům datové vrstvy, je nezávislá na použitém uživatelském rozhraní.

K realizace je často použit nějaký aplikační rámec, zajišťující základní služby (perzistence, řízení transakcí, řízení přístupu, vzdálený přístup, vkládání závislostí, apod.). Většina aplikačních rámců nabízí tzv. deklarativní přístup (vlastnosti komponent se specifikují deklarativně, vývojář se může soustředit na problémovou doménu a nemusí komplikovat kód konstrukcemi pro zajištění těchto vlastností) příkladem je Spring nebo standard EJB.

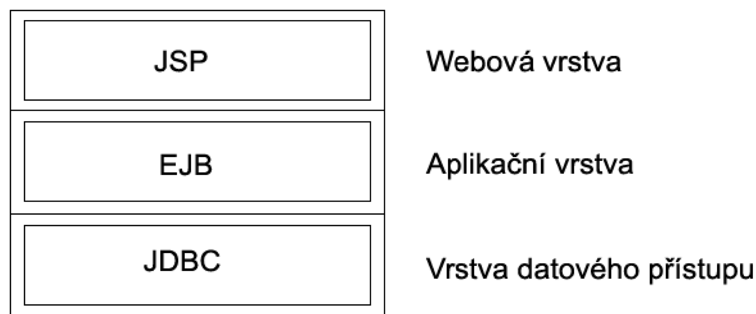
Při použití EJB je aplikační vrstva rozdělena na dvě části: EJB a objekty používající EJB pro podporuji uživatelského rozhraní.

Pro komunikaci s prezentační vrstvou se používá buď přímé volání metod (pokud aplikační i prezentační vrstva běží v rámci jedné JVM), nebo nějaký systém pro vzdálené volání metod (RMI, Web Services, nebo IIOP). Více o komunikaci vrstev viz níže podkapitolu *Distribuované architektury*.

2.7.3 Prezentační vrstva

Tato vrstva obsahuje funkce uživatelského rozhraní. Obvykle existuje několik prezentačních vrstev pro různé druhy zařízení, platformy a prostředí.

Pro realizaci webové prezentační vrstvy slouží technologie JavaServlets a JSP. Často se používají také různé webové aplikační rámce, obvykle založené na architektuře MVC. Tyto rámce mohou být založené na požadavcích (request based; příkladem je Struts, Stripes, Spring MVC), nebo na vizuálních komponentách (component based; příkladem je JSF nebo Tapestry).



Obrázek 2-7: Architektura webové aplikace JSP s EJB

Obrázek 2-7 ukazuje tři vrstvy webové aplikace, kde je prezentační webová vrstva implementována pomocí stránek JSP, střední aplikační vrstva prostřednictvím technologie EJB a vrstva persistence dat je zpřístupněná skrze rozhraní JDBC.

2.7.4 Ne-distribuované architektury

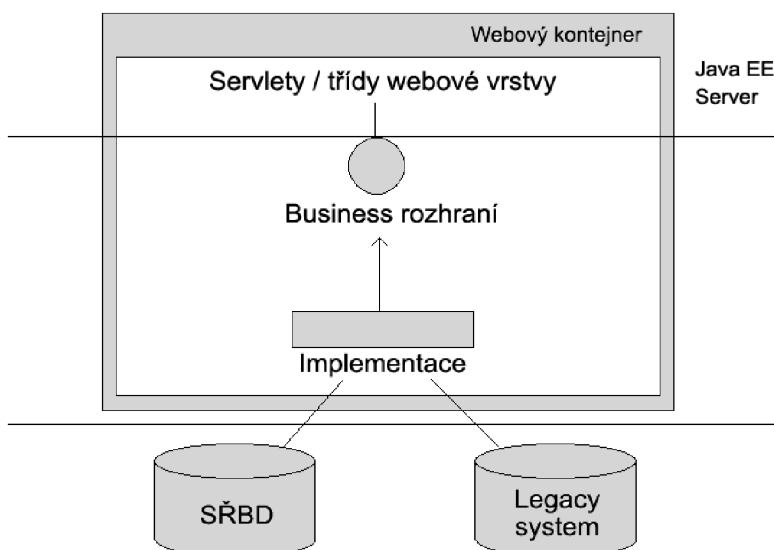
Tyto architektury jsou vhodné pro klasické webové aplikace. Umožňují běh celé aplikace na jediném virtuálním stroji (JVM). Java EE webový kontejner poskytuje komplexní nástroj nutný pro většinu webových aplikací.

Webová i aplikační vrstva zde běží na stejném JVM, nicméně jsou nadále logicky odděleny.

Tento návrh uspokojuje potřeby většiny webových aplikací.

Mezi přednosti těchto architektur patří zejména jednoduchost, s ní spojená rychlost aplikace, snadné testování a rozšiřitelnost.

Mezi nevýhody patří skutečnost, že takto navržená architektura podporuje pouze webové rozhraní, neboť je zde svázána aplikační a prezentační vrstva, které obě běží na stejném JVM.



Obrázek 2-8: Ne-distribuované architektury

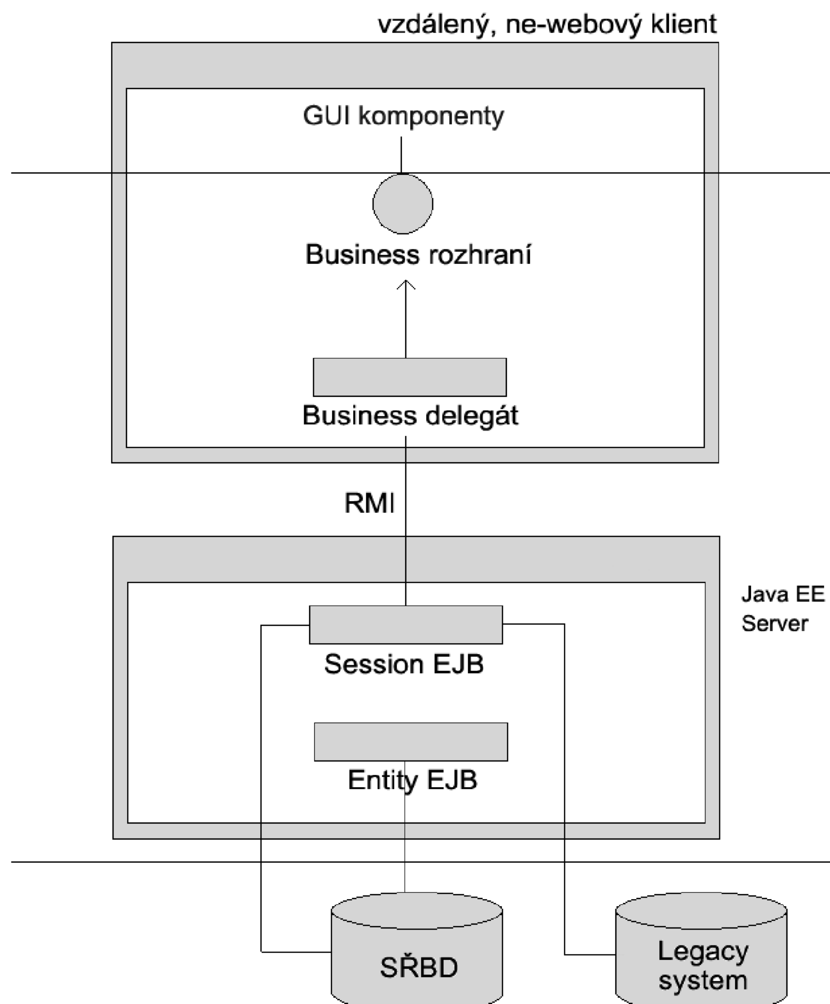
Na obrázku 2-8 je znázorněna ne-distribuovaná architektura typické webové aplikace. Jednotlivé vrstvy jsou odděleny vodorovnými čarami: horní část je prezentační webová vrstva tvořená servlety, střední část je aplikační vrstva tvořená implementací business rozhraní, přes které komunikuje s objekty webové vrstvy, a dolní část reprezentuje databázový systém (SŘDB) nebo Legacy systém (tedy nějaký standardní datový systém).

2.7.5 Distribuované architektury

Tyto architektury podporují jak webové aplikace (tenký klient), tak aplikace s ne-webovým vzdáleným aplikačním klientem (tlustý klient).

2.7.5.1 Distribuované aplikace se vzdáleným EJB

Tyto architektury jsou obecně známy jako „klasické" architektury. Nabízejí schopnost rozdělení aplikační vrstvy fyzicky a logicky užitím různých JVM pro EJB a pro komponenty EJB využívající (např. webové komponenty).



Obrázek 2-9: Distribuované aplikace se vzdáleným EJB

Na obrázku 2-9 je znázorněna ne-distribuovaná architektura typické aplikace s tlustým klientem. Jednotlivé vrstvy jsou odděleny vodorovnými čarami: horní prezentační vrstva je reprezentována komponentami grafického uživatelského rozhraní klienta, střední aplikační vrstva je fyzicky rozdělena částečně do kódu klienta a serveru (na obrázku je na serverové části využito technologie EBJ, se kterou aplikační klient komunikuje skrze business rozhraní pomocí volání vzdálených metod – RMI) a dolní část reprezentuje databázový systém (SŘDB) nebo Legacy system.

Hlavní výhodou této architektury je podpora všech typů Java EE klientů poskytováním sdílené aplikační vrstvy, dále umožnění distribuce komponent aplikace mezi různé fyzické servery.

K nevýhodám patří přílišná komplexnost architektury, což má mimo jiné i negativní vliv na výkonnost a rychlost aplikace. Distribuované systémy jsou také náročnější na testování a odhalování chyb.

2.7.5.2 Aplikace s rozhraním webových služeb

Architektury orientované na služby (Service-Oriented Architecture, dále jen **SOA)** jsou další možností implementace distribuovaných informačních systémů.

Webová služba implementuje aplikační logiku přístupnou pomocí standardních internetových protokolů jako HTTP, HTTPS nebo SMTP. Pro samotné vyvolání metod a předání dat lze použít protokol SOAP (Simple Object Access Protocol), distribuované aplikace tedy již nejsou svázány s použitím RMI.

Rozhraní webové služby je definováno formou zasílání zpráv, které jsou při využívání služby přijímány a generovány, architektury založené na webových službách jsou tedy nezávislé na platformě a podporují i ne-Java EE klienty.

2.8 Testování Java EE aplikace

Existuje několik konceptů testování obecně použitelných pro jakýkoliv softwarový produkt včetně aplikací Java EE:

- **Jednotkový test** – testování samostatné jednotky funkcionality. Jednotkou většinou rozumíme třídu.
- **Test šíře působení** – udává rozsah testování aplikace, většinou v jednotkách.
- **Černá skříňka** – uvažuje se pouze testování rozhraní aplikace bez znalostí implementace systému.
- **Bílá skříňka** - testují se vnitřní aspekty tříd: data označená jako *private* nebo *protected* a metody tříd.

- **Regresní test** – zjišťuje se, zda provedené změny nemají negativní vliv na správnou funkčnost původních komponent. Tedy zda kód, který fungoval správně, funguje správně i po provedení změny (zavlečené chyby).
- **Test hraničních hodnot** – testuje se činnost kódu v extrémních podmínkách.
- **Test splnění požadavků** – testuje splnění podmínek z pohledu zákazníka: zda aplikace vyhovuje zadaným požadavkům.
- **Zátěžový test** - testuje chování aplikace při vysokém zatížení chodu (například vysoký počet uživatelů). Na základě toho testu se určuje maximální limit zátěže, který je aplikace schopna zvládnout a na jeho základě jsou stanoveny a testovány ne-funkcionální požadavky.
- **Stres test** - testuje chování aplikace při překročení limitů daných zátěžovým testem.

Ačkoliv se jedná o testy obecně použitelné pro jakoukoliv aplikaci, jejich použití zajistí dostatečné prověření aplikace Java EE a odhalení všech potenciálních problémů a slabých míst, které může obsahovat.

3 Webové aplikační rámce

Webové rámce (Web Application Frameworks – dále jen **frameworky**) jsou nástroje pro usnadnění vývoje webových aplikací. Jsou nadstavbou nad základní vrstvou pro obsluhu HTTP protokolu, servlety. Obvykle řeší jednu nebo obě problémové oblasti webových aplikací:

- řízení průchodu aplikací
- generování webových stránek ze šablon

Motivací pro vznik a používání frameworků je fakt, že proces vytváření webových aplikací obsahuje mnoho základních úloh, které se neustále opakují. Tyto úlohy je možné vložit do metod a procedur a kód, který úlohy opakovaně provádí, může být z těchto metod generalizován. Na rozdíl od knihoven řeší frameworky více oblastí funkcionality a často řídí proces vytváření aplikace od začátku až do konce.

Frameworky jsou často využívány různými společnostmi pro řešení specifických interních aplikací. Tyto firmy si vytváří buď zcela vlastní frameworky nebo je odvozují z již existujících.

Frameworky by měly být snadno slučitelné s jinými technologiemi (například EJB, servlety apod.) a použitelné pro obecné řešení úloh tvorby aplikací.

Tato kapitola čerpá zejména ze zdroje [4].

3.1 Výhody používání frameworků

Rychlejší vývoj

Frameworky zrychlují práci díky využívání komponent namísto opakovaného psaní stejného kódu.

Frameworky již definují strukturu a návrh aplikace, tudíž ušetří čas při návrhu sofistikované logiky aplikace. I když aplikace návrh logické struktury vyžaduje, je vždy časově výhodnější pracovat s již hotovými komponentami, které framework nabízí.

Kvalita

Kvalitou aplikace rozumíme její schopnost použití a řešení problému, pro který byla určena.

Použitím frameworků a práce s komponentami zaručuje vytváření kvalitní aplikace již od začátku, což je lepší než zdlouhavé hledání chyb později v testovací fázi vývoje.

Frameworky jsou nástroje poskytující vývojovou filozofii orientovanou na dosažení maximální kvality výsledného produktu.

Kvalitu aplikace můžeme posuzovat z mnoho hledisek a oblastí, v nichž je použití frameworku přínosem:

- **Kvalita prostředí:** Zahrnuje veškeré faktory působící na aplikaci jako jsou další součásti systému, do kterého aplikace náleží, fyzické prostředí, apod.
- **Kvalita návrhu:** Framework je soubor návrhových šablon a definicí vzájemných působení těchto šablon. Vývojáři se díky frameworku zcela nemusí zabývat návrhem podpůrných prvků (jako přístup k databázi, bezpečnost, administrace, apod.) a mohou se plně věnovat funkcionalitě a logickým aspektům aplikace.
- **Prevence chyb:** hotové komponenty mají přesně definované chování, jejich zapojení a předvídání defektů je mnohem snazší a průhlednější.
- **Vyhýbání se chybám v kódu:** Nejlepší způsob, jak udělat co nejméně chyb v kódu, je psát kódu co možná nejméně. Použitím frameworku se počet řádků kódu napsaných vývojářem výrazně redukuje, což velmi usnadňuje testování kódu a odhalování chyb.
- **Standardní postupy kódování:** Dbaním na standardizovaných postupech při psaní kódu redukuje výskyt budoucích problémů a pomáhá odhalovat chyby současné. Standardizovaný kód je také výhodnější pro práci ve vývojářských týmech, neboť je mnohem čitelnější a srozumitelnější. Použití frameworku přináší standardizaci práce automaticky s sebou, a podporuje tak dobré návyky vývojářů.
- **Kvalita dokumentace:** Stejné vlastnosti frameworků, které usnadňují odhalování chyb, umožňují také snazší tvorbu kvalitní projektové dokumentace. Vývojář se může lépe soustředit na popis logiky aplikace a nemusí se zabývat psaním rozsáhlé dokumentace k popsání dílčích prvků (jako přístup k databázi, bezpečnost, administrace, apod.).

Cena

Ve vývoji software je cena určena časem nutným pro vývoj aplikace. Protože frameworky díky použitím hotových komponent namísto vytváření nových šetří čas, jsou výhodnější i z hlediska ceny.

Přizpůsobivost

Komponenty jsou již ze své podstaty mnohem přizpůsobivější nežli nově napsaný kód. Tato vlastnost je dána zejména možností změny chování na základě konfigurace komponenty, tedy bez nutnosti přepisování.

Rozšiřitelnost

Hotové komponenty nabízené frameworky jsou umožňují obecně větší rozšiřitelnost než obyčejný kód. Protože jejich možnosti jsou známy předem, jejich umístěný do aplikace je mnohem snazší, což zajišťuje jejich větší rozšiřitelnost.

Schopnost začlenění

Komponenty jsou navrženy pro použití s ostatními částmi aplikace, a je tedy vyžadována jejich integrace do aplikace. Použitím takovýchto komponent se současně zvyšuje i schopnost integrace celé aplikace.

3.2 Nástroje a služby poskytované frameworky

Většina obecných frameworků poskytuje sady nástrojů pro typické úkoly při tvorbě webových aplikací.

Aplikační logika

Architektura Webu je založena na modelu požadavek/odpověď. Většina frameworků pro webové aplikace se snaží s tímto modelem pracovat.

Přístup k databázi

Sofistikované aplikace v současnosti vždy spolupracují s nějakým typem databáze. Frameworky proto poskytují mechanismy pro snadnější přístup k databázi.

Často tyto mechanismy také poskytují formu pro „*mapování*“ mezi objekty a tabulkami relační databáze. Tento přístup umožňuje pracovat s daty v databázi jako s obyčejnými Java objekty, kde takovýto objekt koresponduje vždy s jedním záznamem v databázové tabulce. Toto skýtá mnohé výhody:

- **Objektový přístup:** Díky objektově/relačnímu mapování je možné uvažovat čistě objektovým přístupem, není třeba kombinovat SQL relační přístup a objektové myšlení v Javě.
- **Nezávislost na typu databáze:** Frameworky často poskytují nezávislost při práci s různými typy databází, což usnadňuje přechod aplikace mezi jednou databází k druhé.
- **Využívání vyrovnávací paměti (caching):** Části frameworků starající se o přístup k datům často používají caching, díky čemuž jsou operace čtení dat mnohonásobně rychlejší než přímý přístup k databázi.

Údržba databáze

Pro snadnou údržbu databáze poskytují frameworky funkce pro tzn. "CRUD", tedy vytvoření záznamu (*create*), čtení záznamu (*read*), aktualizace záznamu (*update*), smazání záznamu (*delete*).

Tato sada funkcí poskytuje základní a nejčastěji využívané operace s databází.

Přihlašování uživatelů

Frameworky často poskytují nástroje pro přihlašování uživatelů na různých úrovních, které lze přidat do aplikace bez větší námahy a s minimem napsaného kódu.

Přihlašování je často implementováno jako komponenta frameworku.

Odchytávání událostí

Většina frameworků obsahuje prostředky pro práci s událostmi. Termín „událost“ popisuje procesy na různých úrovních aplikace. Událostí uživatelského rozhraní může být například kliknutí na tlačítko nebo výběr prvku ve výběrovém menu. Události mohou být také spojeny s interním chováním aplikace nebo s daty. Například může být událost generována při změně záznamu v databázi nebo při vypršení platnosti dočasné paměti. Konkrétní význam termínu „událost“ je dán konkrétním frameworkem.

Události jsou také často používány jako prostředek komunikace mezi distribuovanými komponentami.

Využívání vyrovnávací paměti

Většina frameworků požívá vyrovnávací paměť buď pro databázová a perzistentní data nebo pro všeobecné účely, jako ukládání objektů apod.

Konfigurace

Frameworky poskytují konfigurační služby pro své aplikace a komponenty, často uložené ve formě XML souboru nebo databázové tabulky.

Plánování

Služby plánování poskytované frameworky jsou často spjaty se správou událostí, ale mohou být také nezávislé. Většina umožňuje plánování opakujících se úloh, například úloha vykonávaná každé pondělí ve 12:00 apod.

Zasílání zpráv

Potřeba komunikace, synchronní nebo asynchronní, je součástí požadavků mnoha aplikací. Tyto schopnosti nabízejí mnohé frameworky jako svoji součást.

Odchytávání výjimek a chybových stavů

Frameworky typicky nabízejí standardizované způsoby odchytávání výjimek a chybových stavů aplikace.

Monitorování a testování

Frameworky poskytují nástroje pro různé druhy monitorování a testování:

- **Generování používání:** Vytváření pseudonáhodných testovacích datových sad reprezentujících uživatele ve skutečném světě.

- **Zátěžový test:** Simulace uživatelů pomocí mnohanásobných vláken pro zjištění chování systému při vysoké zátěži (velkém množství uživatelů).
- **Monitorování provozu:** proces periodického testování systému pro zjištění, zda pracuje správně z hlediska funkcionality.
- **Hlášení:** Sledování událostí v systému a automatické zasílání hlášení.

Bezpečnost

Klíčovým prvek mnoha webových aplikací je poskytování patřičného stupně zabezpečení.

Frameworky poskytují prostředky pro zabezpečení zahrnující autentizaci, autorizaci, ochranu dat apod.

Prezentace a uživatelské rozhraní

Některé frameworky se soustředí na poskytování prvků pro tvorbu vizuální stránky aplikace – uživatelské rozhraní:

- Stránky a formuláře
- Nabídkové a navigační menu
- Chybové hlášky
- Znaková kódování
- Vícejazyčná podpora prezentace
- Odlišné formátování dat (např. číselné formáty, národní formáty měny, data apod.)

K frameworkům specializujících se na uživatelská rozhraní patří JavaServer Faces, kterému se tato práce věnuje v kapitole *JSF framework*.

3.3 Srovnávání frameworků

Každý framework implementuje různé metody řešení problematiky webových aplikací a o žádném z nich nelze říct, že jeho způsob řešení je „správný“ nebo „špatný“. Při srovnávání je třeba porovnávat frameworky z hlediska jejich výhod a nevýhod.

3.3.1 Aplikační logika

Většina frameworků v sobě obsahuje mechanismy pro zapouzdření aplikační logiky, patřičně provázatelné s uživatelským rozhraním.

Porozumění, jakým způsobem je aplikační logika ve frameworku tvořena a jakým způsobem spolupracuje se svým okolím, je důležitým krokem při volbě, zda je tento framework tím pravým smysluplným a efektivním řešením pro konkrétní problém.

3.3.2 Srovnávání pomocí matice

Aspektů pro hodnocení frameworků je celá řada, každý faktor musí být zahrnut, ale důležitost jednotlivých aspektů může být různá v různých situacích. S velkým počtem prvků vstupujících do hodnocení, může být někdy těžké vytvořit opravdu objektivní zhodnocení.

K propojení všech faktorů a jejich relativních důležitostí může být užitečný zápis jednoduchou maticí. Matice je uspořádaná do dvou dimenzí:

- **Seznam kandidátů:** seznam frameworků ke srovnání. Tento seznam by měl nejlépe obsahovat podobné druhy frameworků. Je těžké objektivně srovnat například framework pro tvorbu uživatelského rozhraní s frameworkem pro komplexní užití.
- **Seznam vlastností:** seznam vlastností frameworků vhodné ke srovnání. Tento seznam musí být kompletní, i v případě, že některý framework obsahuje vlastnosti, kterou jiný nemá, je potřeba tuto vlastnosti také zahrnout.
- **Váha vlastností:** Ke každé zahrnuté vlastnosti je potřeba určit její závažnost pro srovnání. Tato skalární hodnota určuje důležitost vlastnosti uvedené v seznamu vlastností

Vlastnost	**Framework A**	**Framework B**	**Framework C**	***Váha***
Vlastnost 1				
Vlastnost 2				
Vlastnost 3				
Vlastnost 4				

Tabulka 3-1: Srovnávací matice

3.3.2.1 Hodnocení frameworků pro každou vlastnost

Hodnota frameworku pro každou vlastnost určuje míru splnění a obsažení této vlastnosti frameworkem. Hodnoty se doporučují skalární, nejčastěji se užívá hodnotícího rozsahu 1 až 10 nebo procentuální vyjádření ve smyslu „tato vlastnost je tímto frameworkem splněna na X procent". Je také možné k určeném rozsahu přiřadit slovní popis splnění

3.3.2.2 Výpočet celkového skóre a srovnání

Dalším krokem je analýza dosažených informací. Výpočet probíhá následovně: Hodnoty vlastností jednotlivých frameworků jsou znásobeny s váhou pro každý řádek. Součet výsledných hodnot z každého sloupce pak udává hodnotu celkového skóre pro jednotlivé frameworky.

Na základě hodnoty celkového skóre můžeme provést konečné srovnání frameworků a zvolit nejvíce vyhovujícího kandidáta pro daný projekt.

3.4 Katalog frameworků

Následující katalog frameworků představuje malou část z velkého počtu frameworků dostupných v současné době. Zde jsou vybrány ty nejznámější a nejpoužívanější, případně ty, které jsou nějakým způsobem zajímavé a jedinečné.

3.4.1 Komplexní aplikační frameworky

Komplexní aplikační frameworky nabízejí nástroje pro všechny typické potřeby při vývoji aplikací. Obsahují minimálně prostředky pro prezentaci, perzistenci dat, konfiguraci, aplikační logiku a přihlašování uživatelů. Většinou obsahují mnohem více a jejich zaměření použitelnosti může být velmi specifické.

V seznamu se nevyskytují frameworky JSF a Struts, kterým se tato práce věnuje podrobněji v samostatných kapitolách.

Avalon

Apache Avalon projekt, známý také jako Java Apache Server Framework, byl jedním z prvních open-source frameworků. Avalon je vysoce sofistikovaný framework, poskytující nástroje pro strukturování aplikace na základě návrhových šablon.

Cocoon

Cocoon patří k frameworkům soustředícím se zejména na prezentaci, ačkoliv poskytuje celou řadu nástrojů pro kompletní vývoj aplikací. Cocoon obsahuje mnoho instrukcí založených na XML přístupu – XML a XSL transformace jsou základní částí jeho operací.

Cocoon je framework vystavěný na jiném frameworku: používá Avalon framework pro přihlašování, konfiguraci a správu obsahu.

Expresso

Expresso framework je komplexní aplikační framework silně orientovaný na databázový přístup. Jádro frameworku obsahuje mohutnou podporu objektově-relačního mapování a mechanismus pro podporu perzistence zvaný DBObjects.

Spring

Spring je vícevrstvý aplikační framework obsahující podporu pro širokou škálu technologií. Spring může být jistou alternativou k technologii EJB, nabízí však mnohé služby a podporu nad rámec EJB (vlastní MVC s podporou integrace různorodých technologií jako JSP, XSLT atd, podporu AOP, podporu jednotkových testů, plánovaného zpracování (job scheduling) a další.

Spring framework je rozdělen do sedmi modulů, z nichž každý je určen pro jiné použití v aplikaci, k nimž mimo jiné patří modul *Web* určený k podpoře integrace s webovými frameworky jako Struts, JSF apod. Dalším zajímavým modulem je *Web MVC* určený pro tvorbu webových aplikací.

Arch4J

Arch4J framework poskytuje kvalitní infrastrukturu pro vývoj webových aplikací. Na rozdíl od většiny ostatních frameworků se nepokouší oddělit „pohled" jako část architektury MVC přímo, ale soustředí se spíše na „model".

Arch4J nabízí koncept business služeb: aplikace modeluje změny stavu. Business služby provádějí prezentaci pouze při přechodu jednoho stavu na druhý.

Arch4J specifikuje strukturu business služeb, jejich konkrétní implementace je však již plně v rukou vývojáře.

ArsDigita ACSJ

ArsDigita Community System framework je komplexní aplikační framework vydaný jako open-source na základě modifikované licence Mozilla opensources.

ACS obsahuje základní jádrový modul (Core module) pro základní podporu aplikací a dále více specializovaný modul pro správu obsahu (Content Management module) poskytující služby pro vytváření, správu a publikování obsahu webových aplikací.

ACS jádro poskytuje správu uživatelů, autorizace a další podpůrné služby jako správu verzí, e-mail a kategorizování.

Turbine

Turbine je jeden z nejobsáhlejších aplikačních frameworků poskytující služby pro celý vývojový cyklus aplikace, díky čemuž Turbine umožňuje vytvářet webové aplikace velmi snadno a rychle.

Turbine obsahuje přes 200 tříd rozdělených do pěti základních modulů.

Komponenty Turbine mohou být použity i v rámci jiných frameworků.

Niggle

Niggle je komplexní aplikační framework zaměřený na flexibilitu implementace uživatelského rozhraní a perzistence dat. Umožňuje vytvářet servletově založené aplikace s daleko méně řádky kódu pomocí služeb vysoko-úrovňových knihoven.

realMethods

Framework realMethods je komerční produkt založený na velmi důkladném šablonovacím návrhu.

3.4.2 Frameworky zaměřené na prezentaci

Většina těchto frameworků je založena na myšlence zjednodušení tvorby uživatelského rozhraní.

Mezi nejvýznamnější frameworky tohoto typu patří **JavaServer Faces (JSF)** a **Struts**, které jsou v této práci diskutovány detailně později.

Maverick

Maverick je framework peo vývoj webových aplikací založený na modelu MVC.

Scope

Scope framework nabízí odlišný způsob prezentace ve webových aplikacích. Jeho architektonický model není pro webové aplikace příliš vhodný, neboť je dost „těžkopádný".

Scope poskytuje nástroje pro validaci dat, odchytávání událostí a lokalizaci obsahu.

Tapestry

Tapestry framework je alternativou JSP a podobných nástrojů. Vývojář vytváří šablony ve formě HTML stránek, do jejichž tagů lze vložit speciální atribut `jwcid`, tento tag potom Tapestry nahradí kódem vykreslujícím určitou komponentu.

Barracuda

Barracuda je prezentační framework speciálně navržený pro webové aplikace se striktním oddělením prezentace a aplikační logiky prostředky řízení událostmi.

Barracuda je velmi modulární framework obsahující množství měnitelných komponent pro vysokou flexibilitu při vývoji.

Barracuda kompiluje šablony pro prezentaci do Java tříd, s jejichž instancemi je posléze manipulováno pomocí standardního W3C DOM rozhraní.

Echo

Echo framework nabízí poněkud odlišný náhled na podporu vývoje webových aplikací. Umožňuje vývojáři soustředit se na aplikační logiku. Poskytuje událostmi řízený vývojový rámec pro zpracování HTTP žádostí uživatelskými akcemi jakoby byly událostmi v aplikační logice.

Echo nabízí rozsáhlou knihovnu vizuálních komponent a jejich kombinací pro sestavení uživatelských rozhraní aplikace bez nutnosti psaní jakéhokoliv HTML kódu.

3.4.3 Aplikačně specifické frameworky

Následující frameworky jsou zaměřené na určitou problémovou oblast nebo na určitou část ve vývojovém procesu (jako perzistence, databázový přístup apod.).

Aplikačně specifické frameworky jsou zaměřeny na určitý druh problému, nabízejí však pouze část prostředků potřebných pro kompletní vývoj aplikace.

Xerces

Xerces je framework přinášející funkcionalitu zpracování XML dokumentů.

Slide

Apache Slide projekt poskytuje framework pro konstrukce manažera správce obsahu.

JUnit

JUnit je framework pro snadnou definici jednotkových testů a testovacích souborů.

Jetspeed

Jetspeed je framework zaměřený na stavbu webových portálů s obsahem založeným na XML dokumentech.

Simper

Simper framework je zaměřený zejména na perzistenci dat pomocí Enterprise JavaBeans (EJB).

Castor

Castor je framework zaměřený na dolování dat z více datových zdrojů včetně XML dokumentů, relačních databází a LDAP.

jRelational Framework

jRelational Framework je navržen pro snadnou tvorbu objektově-relačního mapování v aplikacích.

Batik

Batik je framework zaměřený na vektorovou grafiku, respektive manipulaci, generování, prohlížení a konverzi souborů typu Scalable Vector Graphics (SVG).

4 JSF framework

4.1 Technologie JavaServer Faces (JSF)

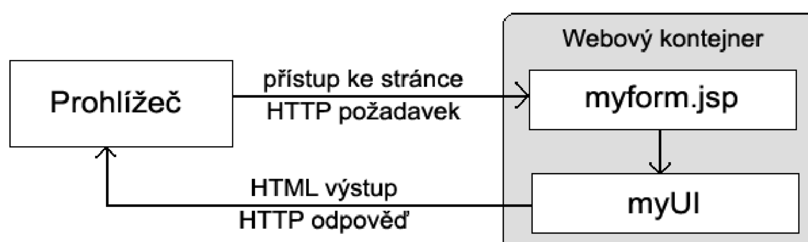
JSF technologie je komponentový framework pro tvorbu uživatelských rozhraní (déle jen **UI**) webových Java aplikací běžící na straně serveru.

Hlavní komponenty JSF jsou následující:

- API (aplikační programovací rozhraní) pro reprezentaci komponent UI a správy jejich stavu; odchytávání událostí, validace a konverze dat na straně serveru; navigace mezi stránkami; podpora více jazyčných mutací a přístupnosti stránek; a možnost rozšíření všech těchto prvků.
- Dvě uživatelské knihovny tagů (značek) pro vytváření JSF uživatelských komponent vně stránky JSP a pro vytváření komponent pro objekty na straně serveru.

Dobře definovaný programovací model a knihovna tagů podstatně usnadňují práci při vývoji webových aplikací s UI na straně serveru. Pomocí JSF lze snadno:

- Obsluhovat události ze strany klienta pomocí kódu na straně serveru.
- Provázat komponenty UI na stránce s daty na straně serveru.
- Vytvořit UI se znovupoužitelnými a rozšiřitelnými komponentami.
- Uložit a znovu získat stav UI po požadavku na server.



Obrázek 4-1: UI vytvořené pomocí JSF běží na straně serveru a prezentuje klientovi

V příkladě na obrázku 4-1 stránka JSP, `myform.jsp`, obsahuje tagy JSF, je tedy stránkou JSF. UI webové aplikace (na obrázku reprezentované `myUI`) spravuje objekty odkazované z JSP stránky. Tyto objekty obsahují:

- UI komponenty, které jsou mapovány do tagů na JSP stránce.
- Naslouchače událostí, validátory a konvertery dat, které jsou registrované s komponenty.
- Objekty zapouzdřující data a aplikací definované funkce komponent.

Tato kapitola čerpá zejména ze zdroje [5], použité ukázky zdrojových kódů jsou vzaty z ukázkové aplikace implementovaná v tomto frameworku, popisované v kapitole *Ukázková aplikace*.

4.2 Výhody JSF technologie

K největším výhodám JSF patří možnost striktního oddělení aplikační logiky a prezentace. Webové aplikace vytvořené pomocí JSP technologie dosahují této separace částečně také. JSP aplikace však nemohou mapovat http požadavek na naslouchač události UI komponent ani spravovat prvky UI jako stavové objekty na straně serveru, což jsou další přednosti JSF aplikace. JSF technologie umožňuje vytvořit webové aplikace s čistě oddělenou aplikační logikou a prezentaci, která tradičně poskytuje UI pro klienta.

Oddělení logiky od prezentace umožňuje a usnadňuje práci v týmech, kdy může být vývoj aplikace rozdělen do dvou nezávislých procesů, ve kterých tvůrce UI pracuje na své části projektu pomocí JSF bez nutné znalosti programování aplikační logiky a aniž by byl nucen psát skriptlety.

Další důležitou výhodou JSF je působení důvěrně známých UI komponent a konceptů webové vrstvy na detaily skriptování nebo značkovacího jazyka. Ačkoliv JSF zahrnuje JSP knihovnu uživatelských tagů pro reprezentaci komponent na JSP stránce, JSF technologie AIP je přímo navrchu programovacího rozhraní servletů, což umožňuje různé případy aplikačního užití, jako například vytváření vlastních uživatelských komponent přímo z komponentových tříd, a nebo generování výstupu pro různá klientská zařízení.

Nejdůležitějším přínosem jsou již zmiňované možnosti technologie JSF pro správu stavů komponent, zpracování uživatelského vstupu (validace, konverze dat) a odchytávání událostí.

4.3 Aplikace JSF

JSF aplikace jsou obyčejné Java webové aplikace běžící v servlet-kontejneru typicky obsahující:

- JavaBeans komponenty zahrnující aplikačně definovaná data a funkcionalitu
- Naslouchače událostí
- Stránky, většinou JSP stránky
- Pomocné třídy na straně serveru, jako jsou beany pro přístup k databázi apod.

Mimoto JSF navíc obsahují:

- Uživatelskou knihovnu tagů pro vytváření UI komponent na stránce
- Uživatelskou knihovnu tagů pro reprezentaci naslouchačů událostí, validátorů, a dalších akcí
- UI komponenty reprezentující stavové objekty na straně serveru

- *Backing beany*, což jsou JavaBeans komponenty definující vlastnosti a funkcionalitu UI komponent na stránce
- Validátory a konvertery
- Soubor pro konfiguraci aplikačních zdrojů – *vývojový deskriptor* (soubor `web.xml`)

Typická JSF aplikace používající JSP pro generování HTML musí obsahovat uživatelskou knihovnu tagů definující tagy, které reprezentují UI komponenty. Musí také obsahovat uživatelskou knihovnu tagů pro reprezentaci dalších akcí, jako jsou validátory a obsluha událostí. Obě tyto knihovny jsou součástí implementace JSF.

4.4 MVC architektura ve frameworku JSF

Framework JSF implementuje architekturu MVC, ale neposkytuje plné prostředky pro implementaci modelu dat. Datovou doménu je tedy potřeba implementoval za použití jiné technologie. JSF poskytuje dobrou podporu pro integraci s EJB – přístupovat k EJB session lze pomocí tzv. *dependency injection*, tedy anotace `@javax.ejb.EJB TridaSessionBeany mojeBeana`.

Pro přístup k datovému modelu a implementaci řadičů aplikace lze využít *backing beanů* (viz podkapitola 4.5.6).

Vrstva pohledu je v JSF implementována pomocí JSP stránek s tagy JSF, tyto stránky se souhrnně označují jako stránky JSF.

4.5 Role ve frameworku JSF

Protože JSF framework podporuje dělení týmové práce na projektu, usnadňuje a zrychluje tak práci designérům, aplikačním vývojářům a projektovým managerům. V mnoha týmech může jedinec hrát více než jednu z následujících rolí, nicméně by měly být brány v zřetel při určování primární zodpovědnosti na projektu. Typičtí členové týmu vývoje jsou:

- **Autor stránek**, který používá značkovací jazyk, jako je HTML, k vytváření stránek pro webové aplikace. Je také většinou autorem grafické designu stránek. Při používání JSF frameworku je autor stránek hlavním uživatelem uživatelské knihovny tagů obsažené v JSF technologii.
- **Aplikační vývojář**, který programuje objekty, obsluhu událostí, konvertery a validátory. Aplikační vývojář také často vytváří speciální pomocné třídy.
- **Tvůrce komponent**, který má zkušenosti s programováním UI a preferuje vytváření uživatelských UI komponent pomocí programovacího jazyka. Tvůrci komponent vytvářejí vlastní komponenty přímo z tříd UI komponent nebo rozšiřují standardní komponenty poskytované JSF.

- **Aplikační architekt**, který navrhuje webovou aplikaci, zajišťuje rozšiřitelnost, definuje navigaci mezi stránkami, konfiguruje beany a provazuje objekty s aplikací.

Hlavní uživatelé JSF jsou autoři stránek, aplikační vývojáři a aplikační architekti.

4.6 Vytváření aplikací v JSF

4.6.1 Kroky ve vývojovém procesu

Vývoj JSF aplikace obvykle sestává z následujících kroků:

- Mapování instance `FacesServlet`
- Vytvoření stránek užitím UI komponent a tagů výkonného jádra (core) JSF.
- Definice navigace stránek v aplikaci pomocí konfiguračního souboru.
- Vývoj backing beanů.
- Vložení řízení beanů (managed bean) do konfiguračního souboru.

Tyto úlohy mohou být provedeny souběžně nebo v jiném pořadí.

4.6.2 Mapování instance FacesServlet

Všechny JSF aplikace musí obsahovat mapování na `FacesServlet` instanci ve svém konfiguračním souboru (vývojovém deskriptoru). Tato instance přijímá přicházející požadavky, zpracovává je, a inicializuje zdroje.

Následující příklad vývojového deskriptoru ukazuje mapování `FacesServlet` instance:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    ...
</servlet>
<servlet-mapping>
    <servlet-name>FacesServlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Tento deskriptor ukazuje mapování JSP stránek využívajících JSF komponent pomocí prefixového volání. URL pro JSP stránku s JSF komponentami musí obsahovat prefix definovaný hodnotou atributu `url-pattern` (v tomto příkladě „`faces/`“):

```
<a href="faces/index.jsp"> ... </a>
```

4.6.3 Vytváření stránek

Vytváření stránek je práce autora stránek. Tato úloha zahrnuje sestavení komponent ve stránce, mapování komponent na beany, vkládání tagů registrujících validátory, konvertory a naslouchače událostí.

4.6.3.1 Deklarace knihoven tagů

JSP stránky s komponentami JSF musí před jejich užitím deklarovat knihovny tagů pomocí `taglib` deklarace:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http:.//java.sun.com/jsf/core" prefix="f" %>
```

První deklarace deklaruje knihovnu tagů pro komponenty HTML s prefixem `h`. Všechny tagy komponent ve stránce mají tento prefix. Knihovna tagů komponent jádra (core) je deklarována s prefixem `f`. Všechny tagy jádra JSF na stránce mají tento prefix.

4.6.3.2 Příklad vložení textového formulářového pole do stránky

Tag `inputText` reprezentuje komponentu textového pole. Následující příklad zobrazuje pole přijímající číselnou hodnotu, jeho instance má atributy: `id`, `label`, `value`.

```
<h:inputText
      id="userNameInput" label="Uživatelské jméno"
      value="#{authenticationJSFBean.inputUsername}">
      ...
</h:inputText>
```

Atribut `id` zde koresponduje s ID komponentového objektu reprezentovaného tagem. V tomto případě je `id` atribut povinný, protože `message` tag (užívaný pro zobrazování chybových zpráv při validaci) potřebuje referenci na tuto komponentu.

Atribut `label` určuje jméno užité při chybových zprávách k referenci na komponentu. V uvedeném příkladu je hodnota atributu nastavena na `User number`, chybová zpráva potom může vypadat takto:

```
User Number: Validation Error: Value is greater than allowable maximum
```

Atribut `value` svazuje komponentu s vlastností beany `UserNumberBean.userNumber`, která obsahuje data vložená do textového pole.

4.6.3.3 Registrování validátoru na textové pole

Vložení tagu `validateLongRange` do těla tagu komponenty textového pole registruje autor stránky `LongRangeValidator` k textovému poli. Tento validátor kontroluje aktuálně zadanou hodnotu pole a konfrontuje ji se zadanou hranicí, definovanou atributy `minimum` a `maximum` tagu `validateLongRange`. Přidáním validátoru se obsah stránky změní následovně:

```
<h:inputText
      id="userNameInput" label="Uživatelské jméno"
      value="#{authenticationJSFBean.inputUsername}">
      <f:validateLength minimum="5" maximum="20"/>
</h:inputText>
```

4.6.3.4 Zobrazování chybových zpráv

Tag `message` se používá k zobrazování chybových zpráv na stránce v případě, že selže konvertor nebo validátor dat. Vložením tagu pro zobrazování zpráv pro pole s `id="userNameInput"` se obsah stránky změní následovně:

```
<h:inputText
      id="userNameInput" label="Uživatelské jméno"
      value="#{authenticationJSFBean.inputUsername}">
      <f:validateLength minimum="5" maximum="20"/>
</h:inputText>
<h:message style="color: red" id="errors1" for="userNameInput"/>
```

4.6.4 Definice navigace stránek

Definice navigace stránek zahrnuje chování všech stránek po tom, co uživatel klikne na formulářové tlačítko nebo hyperlink. Navigace aplikace je definována v konfiguračním souboru aplikace s využitím efektního systému založeném na pravidlech (rule-based system). Následuje příklad pravidel pro definici navigace stránek:

```
<navigation-rule>
        <from-view-id>/admin/login.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
```

```
            <to-view-id>/admin/index.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/admin/login.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
```

Dvě uvedená pravidla definují navigaci ze stránky `/admin/login.jsp` obsahující formulářové tlačítko volající metodu pro zkontrolování zadaných údajů. Tato metoda při úspěšném přihlášení vrátí hodnotu `success`, na základě které se uplatní první pravidlo a dojde k přesměrování navigace na stránku `/admin/index.jsp`, v opačném případě vrátí metoda hodnotu `failure` a uplatní se druhé pravidlo navigace, které přesměruje opět na stránku `login.jsp`, pro možnost opětovného zadaní přihlašovacích údajů. Pro zobrazení komponenty popisovaného formulářového tlačítka se obsah stránky modifikuje přidáním následujícího tagu z JSF knihovny:

```
<h:commandButton
   action="#{authenticationJSFBean.login}" value="Přihlásit se"/>
```

Tento tag vytvoří standardní formulářové tlačítko, jehož akcí bude zavolání metody uvedené v parametru `action`. Tato metoda následně vrací textový řetězec řídící navigaci stránky (v tomto příkladě `success` nebo `failure`).

4.6.5 Vytváření backing beanů

Vývoj beanů je prací aplikačního vývojáře. Typické JSF aplikace párují backing beanu s každou stránkou aplikace. Backing beany definují vlastnosti a metody spojené s UI komponentami užitými na stránce.

V architektuře MVC plní backing beana funkci řadiče.

Autor stránek naváže hodnotu komponenty na vlastnost beany atributu `value` tagu komponenty:

```
<h:inputText
      id="userNameInput" label="Uživatelské jméno"
      value="#{authenticationJSFBean.inputUsername}">
      ...
</h:inputText>
```

Vlastnost backing beany `inputUsername`, která je mapována na data pro komponentu `userNameInput`, může vypadat následovně:

```
private String inputUsername = "";
...
public String getInputUsername() {
        return inputUsername;
}
public void setInputUsername(String inputUsername) {
       this.inputUsername = inputUsername;
}
```

4.6.6 Vložení řídících beanů (managed bean) do konfiguračního souboru

Po vytvoření backing beany pro užití v aplikaci je potřeba nakonfigurovat tyto beany v konfiguračním souboru:

```
<managed-bean>
 <managed-bean-name>authenticationJSFBean</managed-bean-name>
 <managed-bean-class>jsf.AuthenticationController</managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

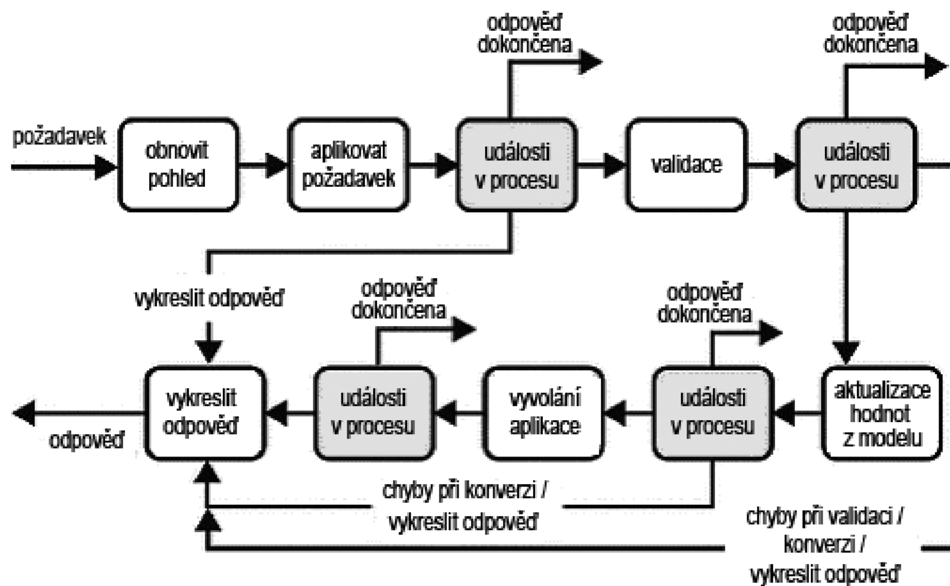
Tato deklarace napuje beanu `authenticationJSFBean` na třídu `jsf.AuthenticationController` tak, že po vytvoření je její životnost určena rozsahem uživatelského sezení (session scope).

4.7 Životní cyklus stránky JSF

Životní cyklus stránky JSF se podobá životnímu cyklu stránky JSP: Klient zašle HTTP požadavek stránce a server následně odpoví stránkou přeloženou do HTML. Životní cyklus JSF je však rozdělen do vícenásobných fází v pořadí zajištující sofistikovaný komponentový model UI. Tento model vyžaduje konverzi a validaci dat komponent, obsluhy komponentových událostí a ukládání komponentových dat do beanů.

Stránka JSF je reprezentována stromem UI komponent zvaným pohled (view). Skrz životní cyklus vytváří JSF pohled za základě uloženého stavu z předchozí stránky. Když klient odešle požadavek, JSF provede několik úloh, jako validaci dat vstupu z komponent v pohledu a konverzi těchto dat do typů specifikovaných na straně serveru.

JSF vykoná všechny tyto úlohy v sérii kroků v JSF životním cyklu typu požadavek/odpověď.



Obrázek 4-2: JSF životní cyklus typu požadavek/odpověď

Obrázek 4-2 ilustruje životního cyklus stránky JSF skládající se z několika fází:

- **obnovit pohled** – při přijetí požadavku na JSF (například klik na tlačítko nebo odkaz); během této fáze vytvoří JSF pohled, prováže naslouchače událostí a validátory s komponentami a uloží pohled do instance `FacesContext`.
- **aplikovat požadavek** – po obnovení pohledu každá komponenta získá novou hodnotu z parametrů požadavku; pokud se při konverzi těchto hodnot objeví chyba, je vygenerována chybová hláška svávaná s komponentou. Zažádá-li dekódovací metoda nebo naslouchač událostí o vykreslení, jsou přeskočeny ostatní fáze přímo do fáze vykreslení odpovědi. Potřebuje-li aplikace v této fázi provést přesměrování na zdroj jiné aplikace nebo generovat odpověď neobsahující žádnou JSF komponentu, může provést dokončení odpovědi pomocí zavolání `FacesContext.responseComplete`. Na obrázku 4-2 je tento stav znázorněn jako **události v procesu** a během životního cyklu může nastat vícekrát.
- **validace** – během této fáze jsou zpracovány všechny validátory registrované s komponentami. Selže-li validace, jsou přeskočeny ostatní fáze požadavkem na vykreslení (typicky obsahující vypsání chybových zpráv). V této fázi je opět možné provést dokončení odpovědi.
- **aktualizace hodnot z modelu** – po legalizaci všech dat lze nastavit hodnoty komponent na hodnoty korespondujících objektů na straně serveru. V případě chyb při konverzi hodnot nebo při žádosti o vykreslení, jsou přeskočeny ostatní fáze přímo do fáze vykreslení odpovědi. V této fázi je možné provést dokončení odpovědi.

- **vyvolání aplikace** – v této fázi jsou zachyceny všechny aplikační události, jako odeslání formuláře odkázání na jinou stránku apod. Je opět možné provést dokončení odpovědi.
- **vykreslit odpověď** – v této fázi JSF vykreslí stránku do JSP kontejneru.

4.8 Internacionalizace dat

JSF rozlišuje tři typy dat, u nichž je možná internacionalizace:

- **Statický text**
- **Chybové zprávy**
- **Dynamická data** nastavovaná dynamicky přes objekty na straně serveru jakými jsou backing beany

Veškerá lokalizovaná data jsou uložena ve zdrojových svazcích (Resource Bundles) reprezentovaných buď třídou `ResourceBundle` nebo textovými soubory, většinou s příponou `.properties`.

4.8.1 Používání zdrojových svazků

Pomocí zdrojových svazků lze oddělit některá textová data ze zdrojového kódu do samostatného souboru. Tento přístup umožňuje snadnou úpravu aplikace a je výhodný pro internacionalizaci.

Pro zpřístupnění lokalizovaných dat ze zdrojových balíků je potřeba jedna ze dvou věcí:

- Registrovat zdrojový balík v aplikaci pomocí konfiguračního souboru použitím elementu `resource-bundle`.
- Načíst zdrojový balík do aktuálního pohledu pomocí tagu `loadBundle`.

Příkladem pro první variantu je:

```
<resource-bundle>
  <base-name>
       com.sun.mypackage.resources.CustomMessages
  </base-name>
  <var>customMessages</var>
</resource-bundle>
```

Příklad pro druhou variantu s použitím tagu `loadBundle`:

```
<f:loadBundle var="bundle" basename="messages.CustomMessages" />
```

Hodnota atributu basename v tomto příkladě specifikuje jméno třídy rozšiřující abstraktní třídu `ResourceBundle`.

4.8.2 Internacionalizace statických dat

K nahrazení statických dat lokalizací ze zdrojových svazků se používá hodnota výrazu atributu komponentového tagu, který zobrazí lokalizovaná data:

```
<h:outputText value="#{bundle.Text}"/>
```

4.8.3 Internacionalizace chybových zpráv

K internacionalizaci chybových zpráv pomocí zdrojových svazků se používá element `message-bundle` v konfiguračním souboru.

Příklad lokalizace chybové zprávy:

```
<h:inputText
      id="userNameInput" label="Uživatelské jméno"
      value="#{authenticationJSFBean.inputUsername}"
      required="true"
      requiredMessage="#{customMessages.ReqMessage}" >
      ...
</h:inputText>
<h:message for="userNameInput" />
```

Hodnota výrazu atributu `requiredMessage` je v tomto příkladě použita k nahrazení chybové zprávy s identifikátorem `ReqMessage` ve zdrojovém svazku `customMessages`. Tato zpráva nahradí korespondující zprávu pro komponentu a bude zobrazena kdekoliv bude ve stránce umístěn tag `message` nebo `messages`.

4.9 Standardní validátory

JSF nabízí sadu standardních tříd a s nimi asociovaných tagů, které umožňují autorům stránek a aplikačním vývojářům validovat data komponent:

- Třída `DoubleRangeValidator`, tag `validateDoubleRange` – kontroluje, zda hodnota komponenty má požadovaný rozsah a je konvertibilní na desetinné (typu `Double`) číslo.
- Třída `LongRangeValidator`, tag `validateLongRange` - kontroluje, zda hodnota komponenty má požadovaný rozsah a je konvertibilní na celé (typu `Long`) číslo.

- Třída `LengthValidator`, tag `validateLength` – kontroluje, zda hodnota komponenty je v požadovaných mezích délky a je konvertibilní na řetězec (typu `java.lang.String`).

Všechny tyto třídy validátorů implementují rozhraní `Validator`. Tvůrci komponent a aplikační vývojáři často také implementují toto rozhraní pro svoje vlastní komponenty.

Následuje příklad použití validátoru `validateLongRange` s atributem minimální hodnoty čerpané z beany `ShowCartBean` a její vlastnosti `minimum`:

```
<f:validateLongRange minimum="#{ShowCartBean.minimum}" />
```

4.10 Standardní konvertery

JSF nabízí sadu standardních tříd a s nimi asociovaných tagů, které umožňují autorům stránek a aplikačním vývojářům konvertovat data komponent.

Standardní implementace nacházející se v balíčku `javax.faces.convert`:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`
- `NumberConverter`
- `ShortConverter`

Všechny tyto konvertory mají asociované vlastní chybové zprávy.

Příklad použití konvertoru, který konvertuje data komponenty na typ (`java.lang.Integer`):

```
<h:inputText converter="javax.faces.convert.IntegerConverter" />
```

Dva ze standardních konvertorů (`DateTimeConverter` a `NumberConverter`) mají svoje vlastní tagy, které jim umožňují nastavit formát dat komponenty, se kterou jsou použity:

```
<h:outputText value="#{MyBean.myDate}">
        <f:convertDateTime pattern="MM/dd/yy"/>
</h:outputText>
```

Konvertor z příkladu konvertuje data z komponenty do formátu MM/dd/yy, tedy například datum 21.2.2008 konvertuje na 02/21/08.

4.11 Naslouchače událostí (event listeners)

Aplikační vývojáři mohou implementovat naslouchače událostí jako třídy a nebo pomocí metod backing bean. Je-li naslouchač metodou beany, autor stránek na metodu odkazuje buďto atributem komponenty `valueChangeListener`, nebo atributem `actionListener`. Je-li naslouchač třídou, autor stánek může odkazovat na naslouchač buďto atributem komponenty `valueChangeListener`, nebo tagem `actionListener` a vložit tento tag dovnitř tagu komponenty v pořadí registrace naslouchačů na komponentu:

```
<h:inputText id="userNameInput" label="Uživatelské jméno"
             value="#{authenticationJSFBean.inputUsername}"
   <f:valueChangeListener type="listeners.UserNameChanged" />
</h:inputText>
```

4.12 JavaServer Pages Standard Tag Library (JSTL)

JavaServer Pages Standard Tag Library (dále jen JSTL) zahrnuje základní (core) funkcionalitu běžnou pro mnoho JSP aplikací. Namísto míchání tagů od více poskytovatelů v jedné JSP aplikaci, umožňuje pracovat s jedinou standardní sadou tagů.

JSTL obsahuje tagy jako iterátory a podmíněnost pro řízení toku programu, tagy pro manipulaci s XML dokumenty, tagy pro přístup k databázi pomocí jazyka SQL, a další běžně používané funkce.

4.12.1 Tag spolupráce

Tagy obvykle spolupracují se svým okolím implicitně nebo explicitně.

Implicitní spolupráce je vykonávána prostředky dobře definovaného rozhraní. Tento druh spolupráce používají JSTL podmíněné tagy.

Explicitní spolupráce nastává, když vloží informace do svého okolí. Na příkladu tag `forEach` vkládá aktuální položku nákupního košíku pomocí iterací:

```
<c:forEach var="item" items="${kosikJSFBean.list}">
    ...
</c:forEach>
```

4.12.2 Tag podpory proměnných

Tag `set` nastavuje hodnotu proměnné nebo vlastnosti proměnné v jiných rozsazích životnosti (page, request, session, nebo application). Pokud proměnná doposud neexistuje, bude vytvořena.

Hodnota nebo vlastnost může být nastavena pomocí atributu `value`:

```
<c:set var="foo" scope="session" value="..."/>
```

nebo z těla tagu:

```
<c:set var="foo">
    ...
</c:set>
```

Příkladem použití může být následující kód:

```
<c:set var="produktId" scope="page" value="#{produktyBean.produkt}"/>
...
<h:inputText id="produktId" value="#{produktId}"/>
```

4.12.3 Tagy řízení toku programu

Pro řízení toku programu není díky JSTL nutné používat skriptlety.

4.12.3.1 Podmíněný tag

Tag `if` umožňuje vykonání svého těla na základě podmínky zadané hodnotou atributu test.

V následující příkladu se testuje, zda je vlastnost beany `authenticationJSFBean` logged má hodnotu „`no`“. Pokud ano, provede se kód uvnitř tagu:

```
<c:if test="${authenticationJSFBean.logged=='no'}">
...
</c:if>
```

4.12.3.2 Tag iterace

Tag `forEach` umožňuje iterovat skrze kolekci objektů. Kolekce je specifikována pomocí atributu `item`, aktuální položka je přístupná skrze proměnnou pojmenovanou atributem `var`:

```
<c:forEach var="item" items="${kosikJSFBean.list}">
    <tr>
         <td align="right" bgcolor="#ffffff">
         ${item.pocet}
    </td>
    ...
</c:forEach>
```

4.13 Vytváření vlastních komponent UI

JSF poskytuje základní sadu standardních, znovupoužitelných komponent UI, které umožňují autorům stránek a aplikačním vývojářům snadno a rychle vytvářet UI pro webové aplikace. Často však aplikace vyžaduje komponentu se speciální doplňkovou funkcionalitou, a nebo je potřeba vytvořit komponentu zcela novou. JSF proto nabízí možnost rozšiřování standardních komponent a vytváření vlastních komponent.

4.13.1 Kroky pro vytvoření vlastní komponenty

Při vytváření vlastních komponent je třeba aplikovat následující kroky:

Vytvoření třídy pro vlastní komponentu

Chování komponenty definované její třídou by mělo obsahovat následující:

- Dekódování (konverze parametru požadavku na lokální hodnotu komponenty)
- Zakódování (konverze lokální hodnotu komponenty do odpovídajícího značkování)
- Uložení stavu komponenty
- Aktualizace hodnoty beany lokální hodnotou
- Validace lokální hodnoty
- Seřazení událostí do fronty

Třída `javax.faces.component.UIComponentBase` definuje výchozí chování komponenty. Všechny třídy reprezentující standardní komponenty rozšiřují tuto třídu. Stejně tak nová komponenta musí rozšiřovat přímo tuto třídu nebo třídy standardních komponent.

Delegování vykreslování na vykreslovač (render)

Pro delegování vykreslování je třeba řešit následující úlohy:

- Vytvořit třídu vykreslovače `Renderer`
- Registrovat vykreslovač s vykreslovacím nástrojem
- Identifikovat typ vykreslovače v ovladači tagů komponent

Zachytávání událostí komponenty

Ve standardních komponentách jsou události automaticky řazeny do fronty. Nová vlastní komponenta si musí události řadit sama tak, jak budou události přicházet.

Vytvoření ovladače komponentového tagu

Po vytvoření komponenty a třídy vykreslovače je třeba definovat, jak ovladač tagu provede reprezentaci komponenty a její vykreslení.

Definice tagu vlastní komponenty v knihovně deskriptorů tagů

K definici tagu je potřeba jej deklarovat v knihovně deskriptorů tagů (Tag Library Descriptor – dále jen **TLD**). TLD je používána webovým kontejnerem pro validaci tagů.

4.14 Konfigurační soubor JSF aplikace

JSF technologie poskytuje pro konfiguraci přenositelný formát souboru založený na XML. Architektura aplikace vytváří jeden nebo více souborů zvaných konfigurační soubory aplikace (Application Configuration Resource File), které registrují a konfigurují objekty a definují pravidla navigace stránek. Konfigurační soubor JSF aplikace je obvykle pojmenován `faces-config.xml`.

Každý konfigurační soubor musí při zachování pořadí obsahovat následující:

- Číslo verze XML:

```
<?xml version="1.0"?>
```

- Tag `faces-config` uzavírající všechny ostatní deklarace:

```
<faces-config xmlns=http://java.sun.com/xml/ns/javaee
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xsi:schemaLocation=http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd
   version="1.2">
   ...
</faces-config>
```

Je možné mít více konfiguračních souborů, JSF je najde následujícím způsobem:

- Soubor /META-INF/faces-config.xml v nějakém JAR souboru ve složce aplikace /WEB-INF/lib/.
- Parametr inicializace obsahu javax.faces.application.CONFIG_FILES, který specifikuje jednu nebo více (oddělené čárkou) cest ke konfiguračním souborům webové aplikace.
- Soubor faces-config.xml ve složce /WEB-INF/ webové aplikace.

Pro přístup ke zdrojům registrovaným v aplikaci je možné použít instanci třídy Application, která je automaticky vytvořena pro každou aplikaci při startu aplikace.

5 Struts framework

Struts, součást Apache Jakarta Project, je open-source komponentový framework pro webové aplikace v Javě založený na filozofii architektury MVC. Autorem rámce Struts je Craig R. McClanahan [6].

Tato kapitola se věnuje zejména aktuální verzi Struts 2 s občasným srovnáním s předchozí verzí Struts 1.x v aspektech, ve kterých se verze podstatným způsobem liší.

Kapitola čerpá zejména ze zdroje [6] a [7].

5.1 Deklarativní přístup

Tento speciální typ konfigurace využívaný v implementaci frameworku Struts umožňuje vývojáři popsat architekturu aplikace na nejvyšší úrovni pomocí konfiguračních nastavení komponent bez nutnosti zásahů přímo ze zdrojového kódu. Pro tyto konfigurace jsou ve frameworku Struts 2 využívány dva různé mechanismy – *deklarace založené na XML* a *deklarace založené na Java anotacích*.

5.1.1 Deklarace založené na XML

Struts 2 umožňuje využívat deklarativní přístup založený na XML. Většina aplikací obsahuje více takovýchto XML souborů, které dohromady tvoří celou deklaraci aplikace. Jako vstupní bod do této deklarace využívá framework Struts jeden XML soubor. Ve Struts 2 je tento konfigurační soubor pojmenován `struts.xml`. Základní struktura souboru je vidět v následujícím kódu:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "http://struts.apache.org/dtds/struts-2.0.dtd"
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN">
<struts>
...
</struts>
```

Tag **`struts`** obsahuje všechny specifické konfigurace Struts 2. Pro vložení externích konfiguračních souborů je možné využít tag **`include`** s atributem `file` určujícím umístění konfiguračního souboru:

```
<include file="modul1-config.xml" />
```

Jednotlivé skupiny konfigurací lze oddělit vložením do těla tagu **package**. Uvnitř tohoto tagu jsou sdíleny atributy jako interceptory (viz níže) nebo URL jmenné prostory.

```
<package name="struts2" extends="struts-default" namespace="/struts2">
...
</package>
```

V předchozím příkladu je definován balíček s názvem určeným atributem `name`, který rozšiřuje jiný balíček určený atributem `extends` a vytváří URL jmenný prostor daný atributem `namespace`. (URL volané akce z tohoto prostoru pak musí začínat tímto jménem).

Jiným příkladem deklarace umístěné v těle tagu `struts` konfiguračního souboru může být následující kód:

```
<action name="Login" class="action.Login">
  <result>/index.jsp</result>
  <result name="input">/login.jsp</result>
</action>
```

V tomto příkladě je provedena deklarace akční třídy `action.Login` pomocí tagu `action` a mapování pohledů na výstupní řetězce z této akce pomocí tagů `result`, které obsahují jako tělo JSP stránku reprezentující zvolený pohled a atribut `name`, jehož hodnota koresponduje s navráceným řetězcem z metody `execute` akční třídy a který určuje výběr pohledu. Pokud atribut `name` není v tagu obsažen, je pohled označen jako výchozí a bude proveden, pokud se nenajde jiný pohled s atributem `name` korespondujícím s hodnotou navráceného řetězce.

5.1.2 Deklarace založené na Java anotacích

Druhým možných přístupem, tedy alternativou k XML deklaracím, jsou deklarace založené na Java anotacích; tento přístup je někdy označován jako *nulová konfigurace* (zero configuration). Tato vlastnost je ve frameworku Struts 2 nová a ve dřívější verzi Struts 1.x neznámá.

Tento přístup umožňuje vkládat deklarativní metadata přímo do zdrojových souborů.

Hlavním cílem Java anotací je podpora pro některé nástroje umožňující čtení metadat z Java tříd a takto získané informace nějakým způsobem využívající. Takto využívá Java anotace i Struts 2.

Příkladem využití Java anotace může být následující kód:

```
@Results({
  @Result(name="input", value="/login.jsp" )
```

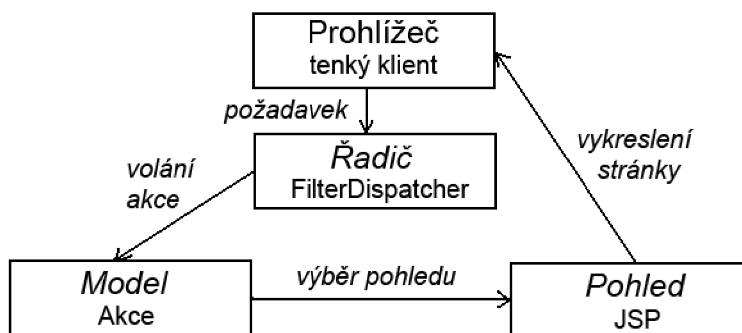
```
  @Result(value="/index.jsp" )
})
public class Login {
  public String execute() {
  ...
```

V tomto příkladě jsou Java anotace využity ve zdrojovém kódu třídy `Login`, které plní funkci akční třídy. Význam anotace je stejný jako v předchozím příkladě s XML deklarací.

Tato práce bude využívat zejména první popsaný způsob, deklarace založené na XML.

5.2 MVC komponentový model rámce Struts

Struts framework implementuje návrhový vzor MVC, resp. jeho specializaci zvanou *Front controller*. V tomto návrhovém vzoru je řadičem servlet poskytující centralizované řízení pro HTTP požadavky všech stránek; mapuje požadavek na jednotku zpracování označovanou jako *akce* (action), akce jsou implementovány jako metody tříd – odtud *akční třída*. Tyto komponenty komunikují s datovým modelem za účelem získání dat pro vykreslení v JSP stránkách implementujících komponenty pohledů. Aplikační logika tedy proběhne ještě před předáním řízení pohledu – tato strategie je nazývána *Service to worker*.



Obrázek 5-1: MVC architektura Struts 2

Obrázek 5-1 ukazuje MVC architekturu frameworku Struts 2 na nejvyšší úrovni. Funkci řadiče přebírá objekt typu `FilterDispatcher`, který filtruje každý příchozí požadavek a deterministicky jej provazuje s nějakou akcí; definice mapování požadavků na akce je provedena deklarativním způsobem v XML konfiguračním souboru frameworku Struts 2. Modelem je myšlen vnitřní stav datového modelu a aplikační logiky; ve Struts frameworku je implementován pomocí akcí.

Protože Struts je framework zaměřený na prezentaci, nenabízí prostředky pro implementaci datového modelu, pro tento je proto třeba využít jiných technologií jako EJB apod.

5.2.1 Komponenty řadiče

Protože varianta vzoru MVC využívaná ve Struts frameworku je Front controller, hraje řadič ve zpracování první úlohu. Jeho práce je mapovat požadavky na konkrétní akce. Toto mapování je prováděno na základě definicí v konfiguračním souboru.

5.2.1.1 Třída `FilterDispatcher`

Role řadiče je ve Struts 2 frameworku dána třídě `FilterDispatcher`, v dřívějších verzích Struts 1.x měla podobnou úlohu třída `ActionServlet`.

5.2.1.2 Interceptory

Koncept *interceptorů* ve Struts 2 frameworku se podobá servletovým filtrům Java EE (rozhraní `javax.servlet.Filter`), provádí tedy před a po zpracování akcí.

Užitím interceptorů lze dosáhnout:

- předzpracování ještě před zavoláním akce
- ovlivnění akcí, poskytnutí informací pro vykonání akcí, nastavení parametrů požadavku pro akce
- následné zpracování po zavolání akce
- modifikace vráceného výsledku, změny pohledu vráceného uživateli
- odchytávání výjimek a jejich alternativní zpracování

Interceptory jsou novým pojmem ve Struts 2 frameworku, který nabízí několik standardních interceptorů s různou funkcionalitou a možnost definování vlastních. V dřívějších verzích Struts 1.x bylo podobné funkcionality možno dosáhnout odvozováním od třídy `RequestProcessor`.

Definice vlastních interceptorů

Pro vytvoření vlastních interceptorů je potřeba implementovat rozhraní `Interceptor`:

```
public interface Interceptor extends Serializable {
  void init();
  void destroy();
  String intercept(ActionInvocation invocation) throws Exception;
}
```

Metoda `init` umožňuje provést inicializaci interceptoru a metoda `destroy` její zrušení a dodatečný úklid. Na rozdíl od akcí jsou interceptory používány skrze požadavky, a proto musí být vláknově bezpečné. Pokud nejsou vyžadovány žádné inicializace ani úklidové akce, lze odvozovat od abstraktní třídy `AbstractInterceptor`, která definuje prázdné metody `init` a `destroy`.

Nejdůležitější metodou, kterou je třeba implementovat, je metoda `intercept`. Parametr této metody je objekt typu `ActionInvocation`, který poskytuje přístup do prostředí běhu aplikace – do vlastní akce (parametry požadavku, sezení, lokalizace apod.), k výsledkům akce; a poskytuje metody pro zavolání akce či její ukončení.

Po vytvoření implementace interceptoru je tento potřeba nakonfigurovat v konfiguračním souboru následujícím způsobem (obdobně i při použití standardních interceptorů):

```
<interceptors>
  ...
  <interceptor name="myintercep" class="interceptor.MyInterceptor" />
</interceptors>
```

V příkladu je vidět konfigurace interceptoru reprezentovaného třídou `MyInterceptor` s názvem daným atributem `name`, který lze dále využít v konfiguraci pro určitou akci (počet interceptorů pro konkrétní akci není omezen):

```
<action name="myAction" class="action.MyAction" >
  ...
  <interceptor-ref name="myintercep" />
</action>
```

Pro každý balíček lze nakonfigurovat výchozí interceptor pomocí tagu `default-interceptor-ref`, jak ukazuje následující příklad:

```
<package ...>
<default-interceptor-ref name="myintercep" />
</package>
```

5.2.2 Komponenty modelu

Hlavní funkcí akcí ve Struts frameworku je zapouzdření zpracování příchozích požadavků. Dále mají na starost přenos dat do uživatelského pohledu a výběr pohledu pro vykreslení uživateli na základě jeho požadavku.

5.2.2.1 Rozhraní `Action`

Akční třídy ve frameworku Struts 2 implementují rozhraní `Action`, které obsahuje jedinou metodu – `execute`. V dřívějších verzích Struts 1.x byla situace o poznání jiná, akční třídy neimplementovaly žádné rozhraní, ale rozšiřovaly třídu `Action` a přetěžovaly její metody, včetně metody `execute`.

Zavoláním akce je spuštěna metoda `execute`, kterou implementují všechny akční třídy, mající jako návratovou hodnotu textový řetězec, který je následně použit pro výběr pohledu. Tento řetězec je zvolen zcela libovolně vývojářem, musí však být zajištěna konzistence mezi návratovými hodnotami akcí a deklaracemi pro mapování pohledů v konfiguračním souboru.

```
public String execute() {
  ...
  return SUCCESS;
}
```

Jak je vidět v příkladě, lze pro některé typické návratové hodnoty akčních tříd použít konstanty předdefinované v rozhraní akce:

```
public static final String ERROR     "error"
public static final String INPUT     "input"
public static final String LOGIN     "login"
public static final String NONE      "none"
public static final String SUCCESS   "success"
```

Přenos dat do pohledů

Protože akční třídy poskytují pohledům data, je potřeba tato data umístit do akční třídy ve formě atributů, které budou pohledem využívány pomocí veřejných metod *getter* a *setter*. Framework následně přenese data z parametrů požadavku do atributů akce automaticky.

S touto problematikou souvisí úzce i podkapitola *Akce ModelDriven* (viz níže).

Konvertery vstupních dat

Framework Struts nabízí několik standardních konverterů pro konverzi typů vstupních dat na typy atributů akcí. Standardní konvertery podporují typy `String`, `boolean/Boolean`, `char/Character`, `Date`, `List`, `Map`, a pole (jejich položky musí být opět konvertabilní).

Pokud máme následující HTML formulář:

```
<form …>
Username: <input name="username" /><br />
Věk s des. přesností: <input name="age" /><br />
Datum narození: <input name="birthday" /><br />
Záliby: <br />
<input name="hobby" /> <input name="hobby" /> <input name="hobby" />
...
```

Můžeme jím naplnit následující atributy akční třídy:

```
private String username;
private Double age;
private Date birthday;
private List hobby;
```

Ve Struts 2 frameworku lze vytvářet i vlastní konvertery pro datové vstupy různých typů. Tyto konvertery musí být implementovány rozšířením abstraktní třídy `StrutsTypeConverter`.

5.2.2.2 Třída `ActionSupport`

Tato pomocná třída poskytuje výchozí implementace rozhraní `Action` a několika dalších užitečných rozhraní a nabízí některé funkce jako například validaci vstupních dat.

Validace vstupních dat

Pro možnost validace vstupních dat lze třídou `ActionSupport` implementovat rozhraní `Validateable` obsahující metodu `validate`, kterou lze implementovat pro konkrétní validaci konkrétních dat, jak ukazuje následující příklad, ve kterém se kontroluje neprázdnost uživatelského jména daného atributem `username`:

```
public void validate() {
  if (getUsername().length() == 0) {
    addFieldError("username", "Username is required.");
  }
...
```

Metoda `addFieldError` je využita k zobrazení textové zprávy dané druhým parametrem k formulářovému poli daném prvním parametrem v pohledu.

Sofistikovanější přístup k validaci vstupních dat je popsán v kapitole *Validace vstupních dat*.

Využití zdrojových svazků pro textové zprávy

`ActionSupport` poskytuje metodu `getText` pro možnost využití textových zpráv ze zdrojových svazků s identifikátorem zprávy jako parametrem.

Využití v metodě `addFieldError` z minulého příkladu by mohlo vypadat následovně:

```
addFieldError("username", getText("username.required"));
```

Zdrojovými svazky textových zpráv se podrobněji zabývá podkapitola *Internacionalizace*.

5.2.2.3 Rozhraní akčních tříd ModelDriven

Akční třídy implementující rozhraní *ModelDriven* upouštějí od využívání vlastností JavaBeanů jako mechanismu pro vytváření modelu dat v aplikaci, místo toho vytvářejí doménový model aplikace skrz metodu `getModel` deklarovanou v rozhraní `ModelDriven`. Implementace tohoto rozhraní způsobí, že data přicházející skrz požadavek budou zpřístupněna aplikační logice pro další zpřístupnění v pohledech.

Následující příklad ukazuje akční třídu implementující rozhraní `ModelDriven`:

```
public class ModelDrivenRegister extends ActionSupport
       implements ModelDriven {
  public String execute() {
    getPortfolioService().createAccount(user);
    return SUCCESS;
  }
  private User user = new User();
  public Object getModel() {
    return user;
  }
...
```

Akční třída v uvedeném příkladě získá modelový objekt zvolené třídy `User` skrz metodu `getModel` z rozhraní `ModelDriven`. Data v požadavku by měla být v korespondenci, pokud by třída `User` obsahovala atributy `username` a `password`, mohl by vstupní formulář vypadat následovně:

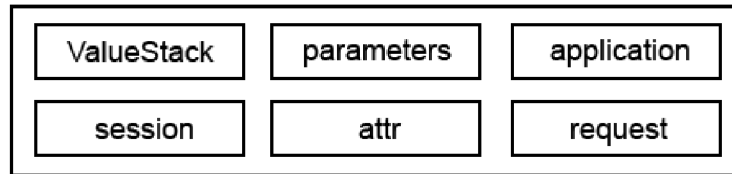
```
<form ...>
Username: <input name="user.username" ... /><br />
Heslo: <input name="user.password" ... /><br />
...
```

5.2.2.4 Kontejner akcí reprezentovaný třídou ActionContext

Jednou z prvních věcí, které framework Struts 2 udělá po obdržení požadavku, je, že vytvoří objekt typu `ActionContext` a uloží v něm všechna data důležitá pro požadavek. Tento objekt potom reprezentuje kontext akce, jednoduchý kontejner tvořící prostředí vykonávání akcí.

Tento kontejner se využívá při integraci dat doménového modelu aplikace s uživatelskými pohledy (viz níže), získávání parametrů požadavku či přístupu k session z akčních tříd.

Na obrázku 5-2 je vidět nejdůležitější části, které `ActionContext` obsahuje, tyto jsou popsány v následující tabulce:



Obrázek 5-2: obsah `ActionContext`

Název	**Popis**
`ValueStack`	obsahuje všechna doménová data aplikace (pro přenos do atributů akce)
`parameters`	seznam parametrů pro aktuální požadavek
`application`	seznam atributů aplikace
`session`	seznam atributů sezení
`attr`	první výskyt atributu stránky, požadavku, sezení nebo aplikace (v tomto pořadí)
`request`	seznam atributů požadavku

5.2.3 Komponenty pohledu

Struts poskytuje knihovny tagů pro vytváření webových stránek umožňující dynamické vykreslování a integraci s daty doménového modelu aplikace prostřednictvím kontejneru akcí (zejména jeho částí `ValueStack`).

Tagy poskytované Struts 2 lze rozdělit do čtyřech kategorií:

- **datové tagy** pro získávání či nastavování dat z/do objektu `ValueStack`,
- **kontrolní tagy** poskytující prostředky pro podmíněné vykreslování stránek,
- **tagy uživatelského rozhraní** (UI tagy) pro vykreslování prvků UI a
- **ostatní tagy** s různou funkcionalitou

Následuje soupis některých nejdůležitějších tagů využívaných v ukázkách kódu dále.

5.2.3.1 Používání JSP k prezentaci

JSP stránky tvoří hlavní součást komponent pohledu ve Struts. Společně s uživatelskými tagy a s HTML usnadňuje JSP v aplikaci práci s pohledy.

Pro použití Struts 2 tagů je potřeba do JSP stránky vložit následující direktivu:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

Syntaxe pro použití tagu je potom následující:

```
<s:názevTagu atribut="hodnotaAtributu"/>
```

JSP syntaxe není jedinou možností pro využití Struts 2 tagů, existují i jiné verze pro použití v různých technologiích.

5.2.3.2 Datové tagy

Datové tagy slouží pro manipulaci s daty v kontejneru akcí, resp. jeho součástí zvané `ValueStack`. Data lze z `ValueStack` získávat či je v něm nastavovat.

Nejdůležitější tagy z kategorie datových tagů udává následující tabulka:

Název tagu	**Popis**
`property`	slouží k vypsání hodnoty atributu akční třídy do HTML stránky.
`set`	slouží k přiřazení atributu akční třídy k jinému jménu, použití pro zjednodušení přístupu k vlastnosti (kratší název) apod.
`push`	uloží vlastnost do `ValueStack`
`bean`	kombinace tagů `set` a `push`. Uloží vlastnost pod zadaným názvem do `ValueStack`.
`action`	umožňuje spustit akci z pohledu. Výsledek akce může být zobrazen na místě tagu ve stránce.

Tabulka 5-1: Datové tagy Struts 2

5.2.3.3 Kontrolní tagy

Kontrolní tagy umožňují řízení vykreslování stránky. Obsahuje tagy pro podmínění, cykly apod.

Nejdůležitější tagy z kategorie kontrolních tagů udává následující tabulka:

Název tagu	**Popis**
`iterator`	slouží k vykreslení prvků kolekce v cyklech. Je možné využít i vlastnosti iterací (pořadové číslo apod.).
`if & else`	tyto tagy vytváří podmíněné toky na základě logické podmínky.

Tabulka 5-2: Kontrolní tagy Struts 2

5.2.3.4 Tagy uživatelského rozhraní

Tagy uživatelského rozhraní poskytují prostředky pro tvorbu uživatelských rozhraní pohledů, zejména pro tvorbu vstupních HTML formulářů.

Nejdůležitější tagy z kategorie tagů uživatelského rozhraní udává následující tabulka:

Název tagu	**Popis**

`table`	vykresluje HTML tabulku
`div`	vykresluje HTML blok
`actionerror`	zobrazí chybové zprávy z akční třídy, pokud nějaké jsou
`actionmessages`	zobrazí informační zprávy z akční třídy, pokud nějaké jsou
`head`	vykresluje hlavičku HTML dokumentu
`form`	vykresluje vstupní HTML formulář
`textfield`	vykresluje vstupní textové pole HTML formuláře
`hidden`	vykresluje skryté pole HTML formuláře
`password`	vykresluje vstupní textové pole HTML formuláře pro zadání hesla
`textarea`	vykresluje víceřádkové vstupní textové pole HTML formuláře
`checkbox`	vykresluje zaškrtávací pole HTML formuláře
`radio`	vykresluje přepínací pole HTML formuláře
`select`	vykresluje výběrové pole HTML formuláře
`file`	vykresluje vstupní textové pole HTML formuláře pro lokální soubor
`submit`	vykresluje odesílací tlačítko HTML formuláře
`label`	vykresluje textový popis pole HTML formuláře

Tabulka 5-3: Tagy UI Struts 2

5.2.3.5 Ostatní tagy

Ostatní tagy mají různá užitečná použití.

Nejdůležitější tagy z kategorie ostatních tagů udává následující tabulka:

Název tagu	**Popis**
`include`	slouží pro vložení výstupu z nějakého webového zdroje do vykreslované stránky. Umožňuje průchod parametrů požadavku do vkládaného zdroje. Použití tohoto tagu má oproti JSP tagu `include` výhodu v integraci s frameworkem – umožňuje přímý přístup do `ValueStack`.
`url`	usnadňuje práci s URL zdroji v aplikaci. Pomocí tohoto tagu je možné zjistit URL jakéhokoliv zdroje, akce apod.
`text`	slouží k zobrazení národnostně specifických textových řetězců, je založen na klíči do svazku zdrojových textů.
`i18n`	specifikuje zdrojový svazek textových zpráv pro použití s tagem `text`. Pokud není uveden, použije se výchozí svazek frameworku.
`param`	nastavuje parametry v tělech ostatních tagů. Sám o sobě nemá význam.

Tabulka 5-4: Kontrolní tagy Struts 2

5.2.3.6 OGNL

OGNL (Object Graph Navigation Language) je výrazový jazyk užívaný v dokumentech založených na textu jako jsou stránky HTML, XML a JSP k odkazování na kontext běžící Java aplikace. Ve Struts 2 frameworku je tento jazyk využívaný v pohledech, resp. v uživatelských tazích Struts 2, pro referenci na datové vlastnosti z prostředí Java aplikace. OGNL je také využíván pro konverzi mezi řetězci zasílanými skrze HTTP a Java datovými typy vlastností.

Protože tvorba uživatelských pohledů ve Struts 2 se bez znalosti OGNL syntaxe neobjede, budou v této podkapitole popsány základní vlastnosti výrazového jazyka OGNL a jeho použití..

Reference na vlastnost beany

OGNL referuje pomocí mechanismu označovaného jako *řetězení vlastností*. Jedná se o přístup k vlastnostem pomocí jejich názvů oddělených tečkou, jak ukazuje následující příklad:

```
osoba.otec.otec.jmeno
```

V tomto příkladě chceme zjistit jméno otce otce není osoby, tedy jméno dědečka této osoby.

Práce s kolekcemi

V OGNL se nerozlišuje mezi kolekcemi typu `List` a poli. Následující tabulka demonstruje práci se seznamy a poli pomocí OGNL:

Java kód	**OGNL výraz**
`list.get(0)`	`list[0]`
`array[0]`	`array[0]`
`((User)list.get(0)).getName()`	`list[0].name`
`array.length`	`array.length`
`list.size()`	`list.size`
`list.isEmpty()`	`list.isEmpty`

Tabulka 5-5: Základní práce se seznamy a poli

Poněkud specifická je práce s kolekcemi typu `Map`. Následující tabulka demonstruje základní práci s těmito kolekcemi:

Java kód	**OGNL výraz**
`map.get("foo")`	`map['foo']`
`map.get(new Integer(1))`	`map[1]`
`User user = (User)map.get("userA");` `return user.getName();`	`map['userA'].name`
`map.size()`	`map.size`

`map.isEmpty()`	`map.isEmpty`
`map.get("foo")`	`map.foo`

Tabulka 5-6: Základní práce s kolekcemi typu `Map`

OGNL také podporuje několik speciálních operací pro filtrování a projekci kolekcí.

Filtrování umožňuje výběr prvků z kolekce na základě nějakého pravidla/podmínky. Syntaxe pro filtrování je následující:

```
kolekce.{? výraz}
```

Ve výrazu *výraz* lze použít `#this` pro referenci na objekt, pro který je filtrování prováděno.

Projekce umožňuje redukovat kolekci pouze na některé její vlastnosti. Syntaxe pro provedení projekce je následující:

```
kolekce.{výraz}
```

Následující tabulka demonstruje některá použití filtrování a projekce nad kolekcemi:

OGNL výraz	**Popis**
`users.{?#this.age > 30}`	filtrování vracející novou kolekci obsahující pouze uživatele, jejichž věk je větší třiceti let
`users.{username}`	projekce vracející novou kolekci obsahující pouze uživatelské jméno pro každého uživatele
`users.{firstName+ ' ' +lastName}`	projekce vracející novou kolekci řetězců reprezentujících celé jméno pro každého uživatele
`users.{?#this.age > 30}.{username}`	projekce, vracející novou kolekci obsahující pouze uživatelská jména, provedená nad filtrovanou kolekcí uživatelů nad třicet let

Tabulka 5-7: Filtrování a projekce nad kolekcemi v OGNL

5.3 Správa výjimek

Pomocí mechanismu výjimek upozorňuje Java závislé klienty na výskyt neočekávané události nebo nenormálního stavu. Klient je seznámen s typem problému pomocí instance konkrétní výjimky a záleží čistě na něm, jak na výjimku zareaguje.

Správa výjimek se v aplikacích Struts příliš neliší od běžných Java aplikací, jiné je však to, k jaké akci při výjimce dojde a jak je o těchto výjimkách informován koncový uživatel.

5.3.1 Výjimky v Javě

Pro lepší pochopení zpracování výjimek ve frameworku Struts je třeba znát základní detaily správy výjimek v aplikacích Java.

V Javě jsou výjimky objekty, které se vytvářejí, když dojde k nenormálním podmínkám během zpracování aplikace. Když aplikace Javy vyvolá výjimku, vytvoří objekt, který je potomkem třídy `java.lang.Throwable`. Tato třída má dvě podtřídy: `java.lang.Error` a `java.lang.Exception`. Od těchto dvou podtříd je dále odvozeno více než 100 přímých a nepřímých podtříd.

Prvky z větve `Exception` jsou obvykle vyvolány, aby oznámily nenormální podmínky, se kterými se aplikace většinou umí vyrovnat. Všechny výjimky vytvářené a vyvolávané uživatelskou aplikací by měly být rozšířením této třídy nebo jejích podtříd.

Třída `Error` a její potomci se zabývají vážnějšími problémy, které se při běhu aplikace mohou vyskytnout. Tyto chyby bývají tak zásadní, že se s nimi většinou nedá nic dělat, proto se klienti o výjimky třídy `Error` příliš nezajímají a jejich odchytávání není povinné.

5.3.2 Správa výjimek uvnitř frameworku Struts

Od verze Struts 1.1 se ve frameworku objevil rámec pro účinnou správu výjimek umožňující vývojářům používat deklarativní i programový přístup. Ošetřování výjimek ve Struts 2 je velmi podobné, výrazněji se liší pouze syntaxí a názvy značek pro deklaraci.

5.3.2.1 Deklarativní ošetření výjimek

Deklarativní ošetření výjimek spočívá ve vyjádření pravidel pro správu výjimek v textovém (většinou XML) souboru včetně toho, které výjimky jsou vyvolány a jak mají být ošetřeny. Tento soubor je zpravidla oddělen od kódu aplikace.

Takovýto přístup usnadňuje úpravy v logice ošetřování výjimek bez velké rekompilace kódu.

Ve Struts 2 je možné deklarovat globální ošetření výjimek v konfiguračním souboru pro každý tag `package` přidáním tagu `global-exception-mappings` do jeho těla. Tento tag obsahuje obecně více tagů `exception-mapping` se dvěma atributy, `exception` udávající plný název třídy výjimky a `result` určující přesměrování při příchodu této výjimky. Toto přesměrování může být určeno pomocí tagu `global-results` definující výsledný pohled. Celou situaci demonstruje následující příklad (při příchodu výjimky typu `Exception` je určeno přesměrování `error` a tomu je dán pohled `error.jsp`):

```
<global-results>
  <result name="error">/error.jsp</result>
</global-results>
```

```
<global-exception-mappings>
  <exception-mapping exception="java.lang.Exception" result="error" />
</global-exception-mappings>
```

Tag `exception-mapping` je možné použít i ke zpracování výjimek pro konkrétní akci, jak ukazuje následující příklad, který deklaruje výjimku, která může být vyvolána z akce `login`.

```
<action name="login" class="action.LoginAction">
  <result>/login.jsp</result>
  <exception-mapping
       result="loginInvalid"
       exception="exception.InvalidLoginException" />
</action>
```

Deklarace v příkladě říká, že pokud bude při zpracování akční třídou `LoginAction` vyvolána výjimka `InvalidLoginException`, bude provedeno přesměrování na pohled `loginInvalid`.

5.3.2.2 Programové ošetření výjimek

Pravým opakem je programové ošetření výjimek, což je tradiční způsob ošetřování výjimek v Javě pomocí bloků `try-catch-finally`.

5.4 Internacionalizace

Internacionalizace je proces tvorby softwaru, který předem počítá s mnoha jazyky a geografickými oblastmi, takže pro podporu dalšího jazyka není třeba měnit zdrojový kód aplikace.

Aplikace s podporou internacionalizace má následující vlastnosti:

- podpora dalších jazyků se obejde bez zásahů do kódu
- textové elementy, zprávy a obrázky se ukládají mimo zdrojový kód
- data vycházející z dané kultury, jako je čas, datum, číslice a měna, jsou formátovány správně vzhledem k uživatelově jazyku a geografické poloze
- podpora nestandardních znakových sad
- snadná úprava aplikace pro nové jazyky a oblasti

Dokonce i aplikace nepodporující internacionalizaci mohou využít výhod některých jejích rysů., jako např. použití zdrojových svazků pro veškerý statický text ušetří mnoho času při údržbě aplikace.

5.4.1 Podpora internacionalizace v Javě

Java poskytuje v základní knihovně rozsáhlý soubor internacionalizačních rysů. Podpora internacionalizace ve frameworku Struts spočívá do značné míry právě na těchto komponentách.

5.4.1.1 Třída Locale

Třída `java.util.Locale` je nesporně nejdůležitější internacionalizační třídou v knihovně Javy. Poskytuje aplikaci instance pro lokaci. Každá jednotlivá instance této třídy reprezentuje jeden jazyk a oblast. Objekty `Locale` neprovádějí žádné formátování za účelem internacionalizace, ale třídy, které rozlišují lokaci, je používají jako identifikátory.

Následující příklad vytváří dva objekty třídy Locale; jeden pro USA a druhý pro Velkou Británii:

```
Locale usLocale = new Locale ("en", "US");
Locale gbLocale = new Locale ("en", "GB");
```

Instanci třídy Locale lze také získat pomocí některých užitečných konstant:

```
Local japanLocal = Locale.JAPAN;
```

Výchozí lokaci daného prostředí lze získat pomocí metody `getDefault()` ze třídy `Locale`:

```
Locale defaultLocale = Locale. getDefault();
```

Pro získání lokace klienta lze použít metodu `getLocale()`, jak ukazuje následující příklad, ve které je klientova lokace získána z těla servletu:

```
...
Locale userLocale = request.getLocale();
```

5.4.1.2 Zdrojové svazky (*resource bundles*) v Javě

Třída `java.util.ResourceBundle` umožňuje seskupovat zdroje pro danou lokaci. Zdroji jsou většinou textové elementy, jako popisky polí, zprávy nebo názvy stránek apod.

Pro zdroje lze využít i dynamického textu s využitím funkcí třídy `java.text.MessageFormat`. Formát zpráv oddělených novým řádkem je `"jméno.zprávy=text zprávy"`. Zdrojový text svazku by mohl potom vypadat takto:

```
...
error.requiredField=Zadejte prosím položku {0}.
label.name=Jméno
```

```
label.surname=Příjmení
```

Číslo ve složených závorkách slouží jako index pro `Object[]`, který se předá se zprávou `format()` třídy `MessageFormat`.

5.4.2 Internacionalizace ve Struts frameworku

Podpora internacionalizace ve frameworku Struts se zaměřuje téměř výhradně na prezentaci textu a obrázků dané aplikace. Neodpovídá za příjem vstupů v netradičních jazycích apod.

Struts 2 poskytuje podporu internacionalizace na základě zdrojových svazků, interceptorů a uživatelských tagů.

5.4.2.1 Zdrojový svazek frameworku Struts

Framework Struts 2 používá pro zdrojové svazky textové soubory, jejichž jména korespondují se jmény akčních tříd, s koncovkou `.properties`.

Při vyhledávání zdrojové zprávy je proveden následující postup, dokud není nalezen odpovídající klíč zdrojové zprávy:

1. prohledání souboru `NázevAkčníTřídy.properties`
2. prohledání souboru s názvem třídy o úroveň výš v hierarchii tříd až k třídě `Object`
3. prohledání souboru pro název každého rozhraní a pod-rozhraní akční třídy
4. pokud je akce typu model-driven, prohledání souborů s názvem tříd jejího modelu
5. prohledání souboru s názvem balíčku pro každý balíček akční třídy vzestupně
6. prohledání souborů konfigurovaných v `struts.properties` pod klíčem `struts.custom.i18n.resources`

V dřívějších verzích Struts 1.x byla situace poněkud odlišná. Byla zde možnost definování více zdrojových svazků v konfiguračním souboru globálně, nikoliv pro samostatnou třídu.

Pojmenování souborů se liší od každé lokalizace. Soubor `NázevTřídy.properties` reprezentuje výchozí lokalizaci pro třídu `NázevTřídy`. Pro každou podporovanou lokalizaci/jazyk je potřeba vytvořit nový lokalizovaný soubor zpráv, např.: `NázevTřídy_en.properties`, `NázevTřídy_de.properties` apod. Všechny tyto třídy by měly obsahovat stejné klíče s rozdílnými hodnotami.

5.4.2.2 Určení lokace pomocí interceptorů

Framework Struts nastavuje lokaci pro uživatelské sezení na základě objektu `HttpServletRequest`. Tato informace je získána přes HTTP přímo z webového prohlížeče uživatele. Pro nastavení závislosti webové aplikace na lokaci je třeba použít interceptor `i18n`.

Tento interceptor kontroluje parametry požadavku a ukládá tyto informace do uživatelského sezení.

5.4.2.3 Internacionalizace v uživatelských pohledech

Pro zobrazení lokalizovaných zpráv v pohledech je možné použít tag `text`, který má jako hodnotu atributu `name` klíč do zdrojového svazku.

Lokalizovaný uvítací text může vypadat takto:

```
<h1><s:text name="label.welcome" /></h1>
```

Jinou možností je využití tagu `property` zavoláním metody `getText` ze třídy `ActionSupport`. Výše uvedený příklad by za použití této metody vypadal takto:

```
<h1><s:property value="getText('label.welcome')" /></h1>
```

Tento způsob lze využít například při lokalizaci návěští formulářových polí, jak ukazuje následující příklad (hodnotu atributu label je potřeba převést na OGNL výraz použitím notace `%{`*`výraz`*`}`):

```
<s:textfield label="%{getText('label.welcome')}" />
```

5.5 Validace vstupních dat

Pro sofistikovaný přístup k validaci vstupních dat se ve frameworku Struts 1.x využíval rámec Validator připojený k frameworku jako plugin. Tento rámec byl využitelný i samostatně v aplikacích, které nebyly přímo postaveny na frameworku Struts. Rámec Validator je dále využitelný i v nové verzi frameworku Struts 2, který nicméně poskytuje svoje vlastní prostředky pro sofistikovanou validaci vstupních dat, velmi podobné mechanismu rámce Validator.

5.5.1 Vytváření validátorů

Validátor představuje souhrn pravidel pro validaci. Tato pravidla jsou ukládána do XML konfiguračního souboru se jménem `NázevAkčníTřídy-validation.xml`. Tento soubor se vztahuje k akční třídě, podle které je pojmenován, a obsahuje definice validačních pravidel pro atributy akční třídy, kterých je třeba validaci provádět.

Máme-li akční třídu `UzivatelAction` obsahující dva datové atributy `jmeno` typu `String` a `vek` typu `int`, můžeme pro načtení hodnot těchto atributů využít pohled se vstupním formulářem v následujícím znění:

```
<s:form method="post">
    <s:textfield label="Jméno:" name="jmeno" />
    <s:textfield label="Věk:" name="vek" />
    <s:submit/>
</s:form>
```

Obsah souboru s validátory `UzivatelAction-validation.xml` bude vypadat následovně:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
    <field name="jmeno">
        <field-validator type="requiredstring">
            <message>Zadejte prosím jméno</message>
        </field-validator>
    </field>
    <field name="vek">
        <field-validator type="requiredstring">
            <message>Zadejte prosím svůj věk</message>
        </field-validator>
        <field-validator type="int">
            <param name="min">18</param>
            <message>Pouze pro uživatele nad 18 let</message>
        </field-validator>
    </field>
</validators>
```

V příkladu jsou vidět validátory definované v těle tag `field` pro prvky `jmeno` a `vek`. Validátory jsou definované tagem `field-validator`. Prvek `jmeno` obsahuje jediný validátor kontrolující vyplnění jeho hodnoty, stejný validátor je pak použit pro prvek `vek`, ve kterém se navíc objevuje validátor pro hodnoty typu `int` s parametrem pro zadání minimální hodnoty (18).

Validátory `requiredstring` a `int` patří ke standardním validátorům, které Struts 2 poskytuje. Výpis základních standardních validátorů pro formulářové prvky udává následující tabulka:

Název validátoru	**Popis**
`conversion`	kontroluje výskyt chyby v konverzi zadané hodnoty prvku na požadovaný typ atributu akční třídy

`date`	kontroluje platnost zadaného data do zadaného rozsahu (parametry `min` a `max`)
`double`	kontroluje platnost hodnoty desetinného čísla do zadaného rozsahu (parametry `min` a `max`)
`email`	kontroluje zadaný řetězec, pokud není prázdný, na obsah e-mailové adresy
`fieldexpression`	kontroluje hodnotu prvku na zadaný OGNL výraz (parametrem `expression`)
`int`	kontroluje platnost hodnoty celého čísla do zadaného rozsahu (parametry `min` a `max`)
`regex`	kontroluje hodnotu prvku na zadaný regulární výraz (parametrem `expression`)
`required`	kontroluje, zda byla zadána hodnota prvku
`requiredstring`	kontroluje, zda byla zadána hodnota textového prvku a zda tato hodnota není prázdná
`stringlength`	kontroluje délku zadaného řetězce (pomocí parametrů `minLength` a `maxLength`)
`url`	kontroluje hodnotu prvku na obsah URL adresy

Tabulka 5-8: Standardní validátory pro formulářové prvky

5.5.1.1 Lokalizace zpráv validátorů

Pro lokalizaci zpráv daných parametrem `message` validátoru lze využít metodu `getText` uvnitř OGNL výrazu pro načtení zprávy ze zdrojového svazku zpráv:

```
<message>${getText("validation.inputrequired")}</message>
```

nebo pomocí atributu `key`, který ukazuje na klíč zprávy ze zdrojového svazku:

```
<message key="validation.inputrequired" />
```

5.5.2 Registrace validátorů

Všechny validátory využívané v aplikaci je třeba zaregistrovat pomocí konfiguračního XML souboru `validators.xml`:

```
<!DOCTYPE validators PUBLIC
   "-//OpenSymphony Group//XWork Validator Config 1.0//EN"
   "http://www.opensymphony.com/xwork/xwork-validator-config-1.0.dtd">
<validators>
  <validator name="requiredstring"
```

```
class="com.opensymphony.xwork2.validator.validators.RequiredStringVali
dator" />
   <validator name="int"
class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValid
ator" />
...
</validators>
```

5.5.2.1 Zprovoznění validace

Zprovoznění validace pro akční třídy vyžaduje deklaraci interceptoru pro validaci:

```
<interceptor name="validation"
class="org.apache.struts2.interceptor.validation.AnnotationValidationI
nterceptor" />
```

Takto deklarovaný interceptor je následně potřeba přidat do deklarace každé akční třídy, ve které bude validace prováděna.

6 Ukázková aplikace

Pro demonstraci zvolených frameworků byla navržena typická ukázková aplikace skýtající rozličnou funkcionalitu včetně databázové struktury používající všechny základní typy vztahů mezi entitami.

Touto demo aplikací je jednoduchý obecný **elektronický obchod** a jeho administrační část, což je téma natolik zažité, že pochopení idey aplikace a její implementace ve zvoleném frameworku bude velmi názorné. Ukázková aplikace implementuje pouze základní funkčnost pro demonstraci možností a schopností zvoleného frameworku. Vytvoření komplexní aplikace by nepřineslo do problematiky nic nového a spíše by dělalo ukázku nepřehlednější, což je zde nežádoucí.

6.1 Určení požadavků

6.1.1 Neformální specifikace

Ukázková aplikace implementuje jednoduchý elektronický obchod. Běžný uživatel (potenciální zákazník) má možnost prohlížet nabízené zboží členěné do kategorií. Zboží může po zobrazení detailu zboží umístit do nákupního košíku. Množství u jednotlivého zboží v košíku je možné měnit. Zboží z košíku je možné objednat po přihlášení k již existujícímu účtu nebo registrací se zadáním kontaktních údajů (fakturační a případné dodací adresy) a volbou uživatelského jména a hesla pro pozdější přihlášení. Provozovatel obchodu bude mít možnost správy veškerého zboží nabízeného obchodem včetně kategorií obchodu a jejich obsahu.

6.1.1.1 Slovníček pojmů

Jednoznačnost obchodní terminologie v projektu zavádí slovníček obchodních pojmů. Uvedený slovníček obsahuje pouze termíny z oblasti prvního popisu požadavků, pro použití v dalších fázích vývoje by musel být dále rozšiřován.

Pojem	**Definice**
Obchod	Zákaznická část informačního systému poskytující nabídku zboží s možností přidání do nákupního košíku a objednání.
Zboží	Reprezentuje nabízené produkty včetně jednotlivých variant zboží.
Detail zboží	Výpis souhrnných informací o konkrétním nabízeném zboží.
Varianta zboží	Specifikace zboží (např. rozměr, barva, apod.) jednoznačně přiřazená k určitému zboží, charakterizuje zboží zejména cenou. Jedno zboží obsahuje jednu nebo více variant.
Kategorie	Skupina administrátorem určeného zboží s jednoznačným názvem.

Nákupní košík	Kolekce zákazníkem zvolených variant zboží s možností objednání.
Množství	Tento údaj se týká zejména položek v nákupním košíku. Informace o množství udává hodnotu počtu jednotek daného zboží. Týká se konkrétní varianty zboží.
Zákazník	Náhodný návštěvník obchodu, potencionální zájemce o nabízené zboží.
Administrátor	Provozovatel obchodu znající přihlašovací údaje pro správu obchodu z administrační sekce.

6.1.2 Analýza požadavků

Aplikace elektronického obchodu je rozdělena do dvou částí:

- **vlastní virtuální obchod** s nabídkou produktů a možností přidání do košíku a objednání
- **administrační modul** umožňující provozovateli po přihlášení zpravovat kategorie obchodu a produkty

6.1.2.1 Vlastní elektronický obchod

Potenciální zákazník resp. běžný uživatel (dále jen **zákazník**) má zde bez nutnosti registrace možnost prohlížet nabízené zboží a nezávazně je vkládat do virtuálního nákupního košíku (dále jen **košíku**), ze kterého lze produkty po přihlášení objednat. Před prvním přihlášením do systému musí uživatel provést vytvoření nového účtu, kam zadá informace o své osobě a fakturační a dodací adresu.

6.1.2.2 Administrace obchodu

Provozovatel obchodu (dále jen **admin**) má zde po přihlášení možnost spravovat obsah obchodu v plném rozsahu (přidávání, mazání, editace). Tato správa se týká záznamů reprezentujících zboží a kategorie obchodu.

6.1.3 Business model požadavků

Aplikace operuje s pěti základními entitami jejichž hodnoty jsou ukládány do databáze:

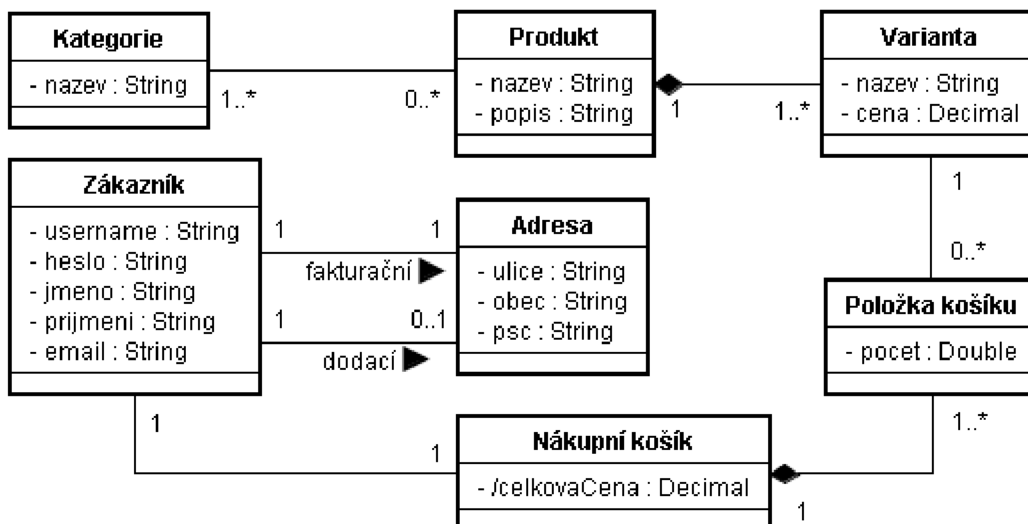
- **`Kategorie`** – kategorie produktů; v každé kategorii může být více produktů (entita `Produkt`, níže), každý produkt může být přiřazen k více kategoriím. Z toho plyne násobnost vztahu *Kategorie:Produkt* – N:M. Kategorie nemusí obsahovat žádné produkty, ale produkt musí být zařazen nejméně v jedné. Entita bude mít jediný atribut `nazev` nesoucí jednoznačné textové označení kategorie (např.: „Technická literatura“ pro obchod s nabídkou knih apod.)
- **`Produkt`** – základní entita reprezentující souhrnné informace o produktu. Entita má, krom výše popsané vazby s entitou `kategorie`, vazbu s entitou `Varianta` (níže). Násobnost tohoto vztahu je 1:N – produkt může obsahovat více variant (nejméně jednu),

ale každá varianta patří právě do jednoho produktu. Entita obsahuje dva atributy: `nazev`, textové označení produktu, a `popis`, což je textová charakteristika produktu.

- **`Varianta`** – detailní specifikace produktu, objednatelné zboží je charakterizováno právě touto entitou; udává variantu nabízeného produktu, jakou může být velikost, rozměr, barva či jiná konkretizace; obsahuje jediný atribut, `nazev`, popisující stručně variantu v kontextu produktu (pro prodej obrazů může být variantou nějakého obrazu např.: „rozměr 1000x520 cm, rám z lipového dřeva“).
- **`Zákazník`** – entita nesoucí informace o zákazníkovi resp. registrovaném uživateli. Obsahuje vazbu na entitu `Nákupní košík` (níže) s násobností 1:1, která říká, že každý zákazník může vlastnit pouze jediný virtuální nákupní košík; vazba s entitou Adresa (níže) je násobnosti 1:2, protože zákazník může mít zároveň dvě adresy (dodací a fakturační), v případě nevyplnění dodací adresy, je tato shodná s fakturační adresou (nedochází tedy k redundanci dat). Entita `Zakaznik` má několik atributů udávajících uživatelské přihlašovací jméno do systému, přihlašovací heslo (obě zvolená při registraci), křestní jméno a příjmení zákazníka, jeho e-mailovou adresu.
- **`Adresa`** – fakturační a dodací adresa zákazníka. Obsahuje tři atributy: název ulice, název obce, a PSČ.
- Další entitou, jejíž životnost je určena existencí uživatelského sezení je entita **`Nákupní košík`**. Tato entita se neukládá do databáze, nevyskytuje se tedy v databázové struktuře na obrázku 7-2. Virtuální nákupní košík může obsahovat neomezený počet variant produktů, přičemž u každé položky nese hodnotu množství. Tato situace je modelována vazbou 1:N mezi entitou `Nákupní košík` a `Varianta`, která říká, že nákupní košík může obsahovat více variant, ale každá varianta je v košíku pouze jednou. Informace o množství, ve kterém je varianta obsažena v košíku, je modelováno pomocí asociační třídy, `obsahuje` s atributem `pocet` nesoucím hodnotu udávající toto množství. Entita sama obsahuje jediný odvozený atribut, `celkovaCena`, který udává agregovanou hodnotu celkové ceny vypočítané z položek v košíku a jejich množství.

6.1.3.1 Business diagram tříd

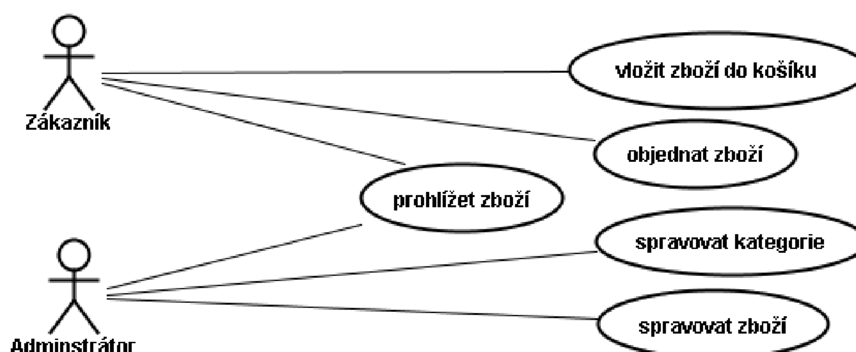
Business entity popsané výše jsou v implementaci informačního systému reprezentovány třídami, jak ukazuje diagram business tříd na obrázku 6-1:



Obrázek 6-1: Business diagram tříd

6.1.3.2 Business diagram případů užití

Business diagram případů použití na obrázku 6-2 ukazuje základní funkce obchodu na úrovni nejvyšší abstrakce. Aktor `Administrátor` zde značí autorizovaného správce serveru, provozovatele a aktor `Zákazník` značí náhodného uživatele, potenciálního nákupčího. Oba uživatelé mohou **prohlížet zboží**, administrátor pro možnost **spravovat zboží** zahrnující editaci, přidání nového a smazání stávajícího a zákazník pro možnost **vložit zboží do nákupního košíku** a následně **objednat zboží**. Administrátor může také **spravovat kategorie** pro škálování nabízeného zboží v obchodě.

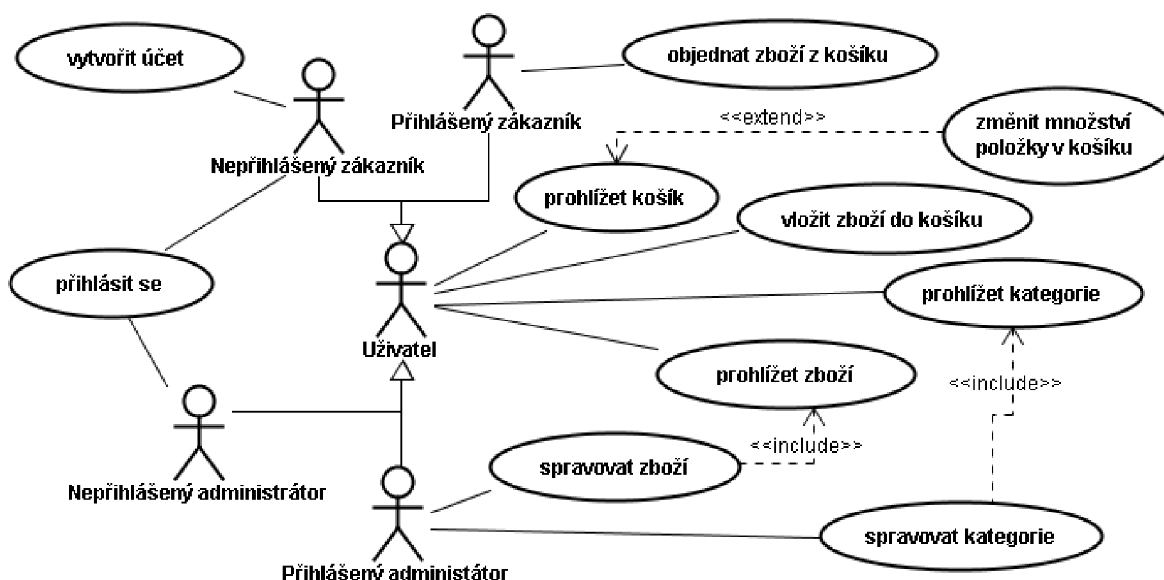


Obrázek 6-2: Business diagram případů užití

6.2 Specifikace požadavků

6.2.1 Diagram případů užití

Diagram případů použití na obrázku 6-3 specifikuje podrobněji funkce, které by měl systém ukázkové aplikace implementovat. Vychází z business diagramu případů užití, který doplňuje a rozšiřuje o některé případy použití.



Obrázek 6-3: Diagram případů užití

Klíčovou změnou oproti business modelu je zavedení abstraktního aktora `Uživatel`, který je dále specializován na *přihlášeného* a *nepřihlášeného* zákazníka. Tento model zdůrazňuje aspekt systému, že některé akce jsou k dispozici pouze přihlášenému zákazníkovi. Jedná se zejména o případ užití objednat zboží z košíku – objednání zboží je v systému podmíněno registrací a následným přihlášením, zatímco přidávání zboží do košíku je nezávazné, a je tedy zpřístupněno jak přihlášenému, tak nepřihlášenému zákazníkovi.

Diagram na obrázku 6-3 ukazuje následující případy použití:

- **vytvořit účet** – registrace nového zákazníka do systému při vytvoření nového uživatelského účtu obsahující zvolené uživatelské jméno a heslo, použité pro pozdější přihlášení k existujícímu účtu. Součástí úspěšné registrace je automatické přihlášení.
- **přihlásit se** – přihlášení zákazníka do systému na základě přihlašovacích údajů (zvolených při registraci / vytvoření účtu – pro zákazníka).

- **objednat zboží z košíku** – v případě že nákupní košík obsahuje alespoň jednu položku, je možné zboží z košíku objednat. Pro objednání zboží z košíku je vyžadováno přihlášení registrovaného zákazníka.
- **prohlížet košík** – obsahuje-li nákupní košík nějaké položky, budou tyto strukturovaně vypsány, v opačném případě bude zobrazena informativní zpráva o absenci položek v košíku.
- **změnit množství položky v košíku** – prohlížení košíku může být rozšířeno možností změnit množství, které, jak ukazuje diagram na obrázku 6-1, je implementováno atributem `pocet` třídy `Položka košíku`. Hodnoty tohoto atributu je možné měnit v zadaném rozsahu (hodnota množství musí být větší než nula apod.).
- **vložit zboží do košíku** – z detailu zboží je možné vložit položku zboží do nákupního košíku. V nákupním košíku bude vytvořena položka nesoucí informaci o zboží a jeho množství v košíku.
- **prohlížet zboží** – zobrazení zvoleného zboží s možností přidání zboží do košíku.
- **spravovat zboží** – správa zboží popsaná již v business modelu.
- **prohlížet kategorie** – zobrazení seznamu zboží ze zvolené kategorie.
- **spravovat kategorie** – správa kategorií popsaná již v business modelu.

6.2.2 Strukturovaný zápis některých případů použití

Následující tabulky zobrazují strukturovaný popis vybraných případů použití pro manipulaci s nákupním košíkem a objednání zboží. Ilustrují mechanismus nákupu v elektronickém obchodě včetně nutnosti přihlášení pro možnost objednání zboží.

Popis všech případů použití není klíčovou náplní této práce, a proto nebyl z prostorově úsporných důvodů uveden kompletní.

Případ použití: VložitZbožíDoKošíku
ID: 1
Stručný popis: Systém vloží zvolené zboží do košíku.
Primární aktéři: Nepřihlášený zákazník, přihlášený zákazník (dále jen Zákazník)
Sekundární aktéři: Žádný
Předpoklady: 1. Je zobrazen detail zboží s možností „přidat do košíku“.

Hlavní tok: 1. Případ použití se spustí, když Zákazník při prohlížení detailu zboží vybere „přidat do košíku". 2. Zákazník zadá množství přidávaného zboží 3. Systém ověří, zda nákupní košík již neobsahuje vkládané zboží 4. *Pokud* nákupní košík neobsahuje vkládané zboží 4.1 Systém vytvoří novou položku v nákupním košíku spolu s informací o množství.
Následné podmínky: 1. Košík obsahuje zvolené zboží v zadaném množství.
Alternativní toky: ZbožíJeJižObsaženo

Případ použití: VložitZbožíDoKošíku: ZbožíJeJižObsaženo
ID: 2
Stručný popis: Systém zvýší množství u zboží v košíku v závislosti na aktuální a vkládané hodnotě.
Primární aktéři: Nepřihlášený zákazník, přihlášený zákazník (dále jen Zákazník)
Sekundární aktéři: Žádný
Předpoklady: Nákupní košík již obsahuje vkládané zboží
Alternativní tok: 1. Alternativní tok se spustí po kroku 2 hlavního toku. 2. Systém přičte k aktuální hodnotě množství hodnotu vkládaného množství zboží.
Následné podmínky: 1. Množství u vkládaného zboží je zvýšeno

Případ použití: ObjednatZbožíZKošíku
ID: 3
Stručný popis: Systém založí novou objednávku zboží z košíku.
Primární aktéři: Přihlášený zákazník
Sekundární aktéři: Nepřihlášený zákazník

Předpoklady: 1. Košík obsahuje aspoň jednu položku.
Hlavní tok: 1. Případ použití se spustí, když Zákazník vybere „objednat zboží z košíku“. 2. *Dokud* není Zákazník přihlášen 2.1 Systém požaduje, aby uživatel zadal přihlašovací údaje (uživatelské jméno a heslo) nebo aby zrušil objednání 2.2 Systém ověří, zda je zákazník přihlášen 3. *Jestliže* je Zákazník přihlášen 3.1 Systém založí novou objednávku. 3.2 Systém informuje uživatele, že objednávka byla zpracována.
Následné podmínky: 1. Nová objednávka byla založena.
Alternativní toky: Žádný

6.3 Implementace

Jako vývojové prostředí jsem zvolil IDE Netbeans 6.1 a jako aplikační server GlassFish V2 UR2.

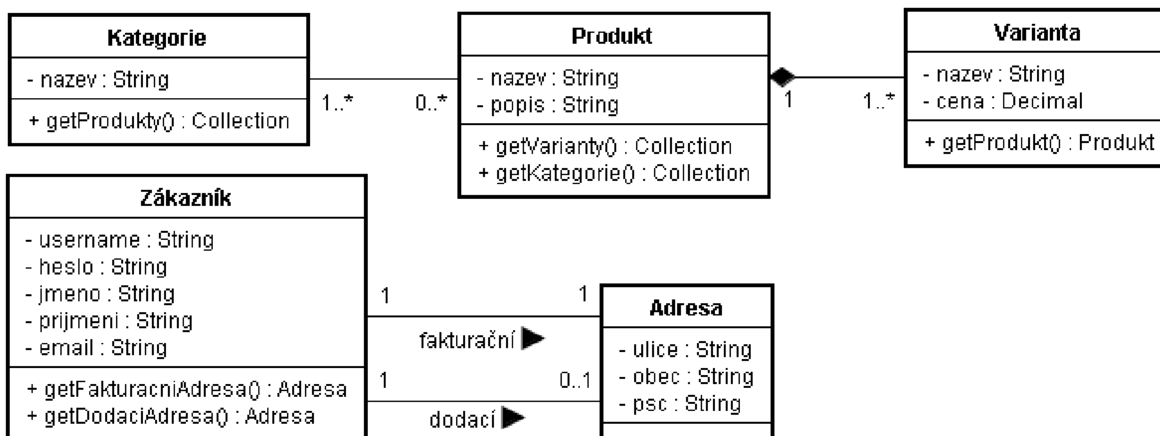
6.3.1 Implementace datového modelu v architektuře MVC

Protože oba frameworky, JSF i Struts, slouží zejména k prezentaci a neposkytují komplexní podporu pro implementaci datového modelu, zvolil jsem pro tuto část projektu technologii EJB, resp. EJB 3, která je využitelná jak v kombinaci s JSF, tak i s frameworkem Struts.

Následující diagram tříd vychází z business modelu, modeluje však pouze perzistentní třídy a je výchozím bodem pro návrh tříd EJB entit.

6.3.1.1 Diagram tříd

Diagram tříd na obrázku 6-4 zobrazuje třídy, které budou perzistentně uloženy v databázi:



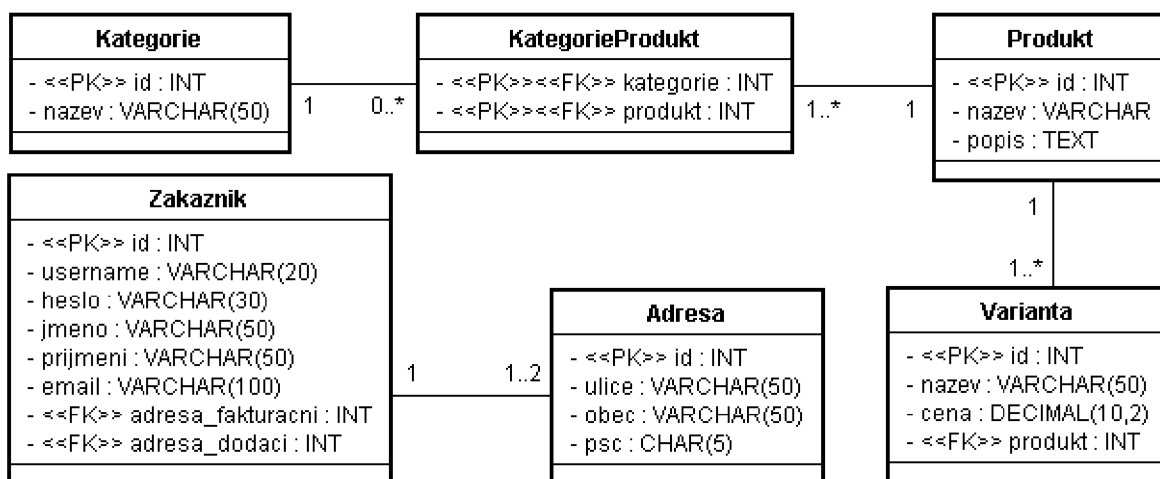
Obrázek 6-4: Diagram tříd

Uvedené třídy obsahují atributy se stejným významem jako v business modelu. Operace implementují zejména vazby mezi entitami. Tyto jsou implementovány pomocí kolekcí s přihlédnutím k povaze práce s vazbami mezi objekty v EJB, ta je totiž založená výhradně na kolekcích, v jiném případě by byly na místě spíše metody pro přidání prvku do vazby (např.: `addToKategorie` u třídy `Produkt` apod.).

Všechny třídy obsahují navíc operace pro práci s atributy (*getter* a *setter*), které v diagramu z úsporných důvodů nejsou uvedeny.

6.3.1.2 Návrh struktury relační databáze

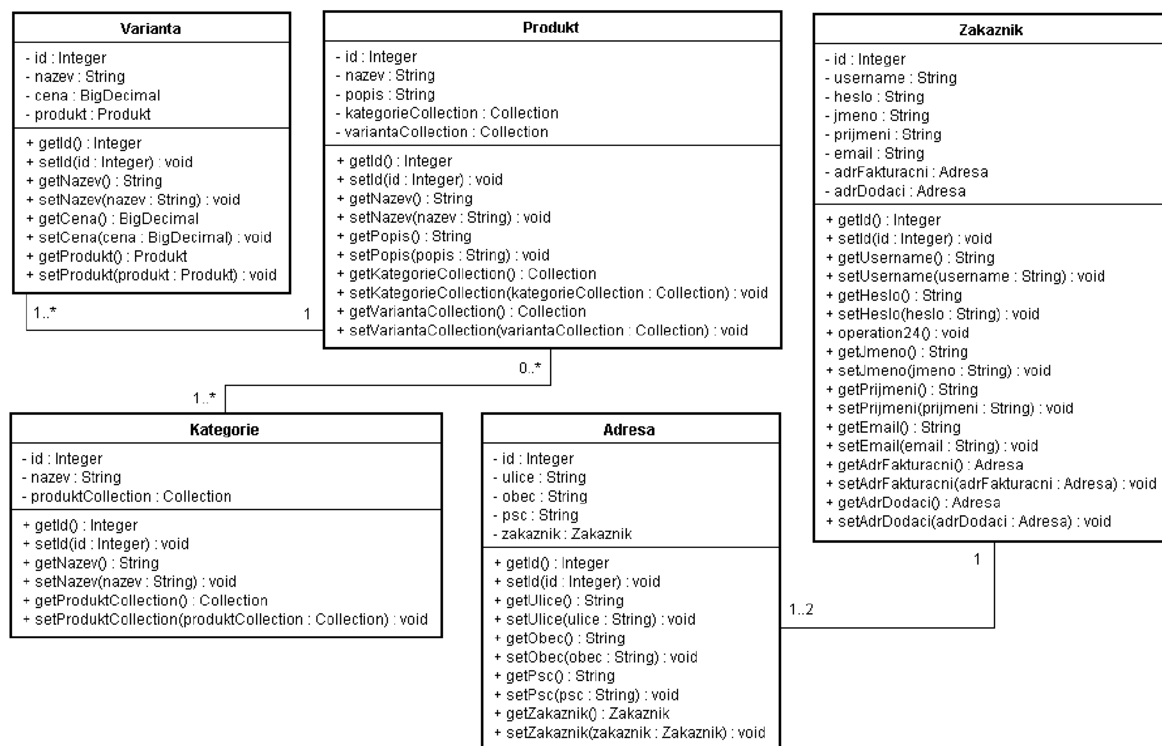
V systému je užito relační databáze, je tedy třeba zahrnout do návrhu vztahy relací a přidat atributy identifikující relace a vztahy mezi nimi pomocí primárních (`<<PK>>`) a cizích klíčů (`<<FK>>`), jak ukazuje databázová struktura na obrázku 6-5 . Struktura databáze vychází z perzistentních entit zobrazených na obrázku 6-4.



Obrázek 6-5: Struktura relační databáze pro uložení perzistentních entit aplikace

6.3.1.3 Datový model – vrstva perzistentních dat

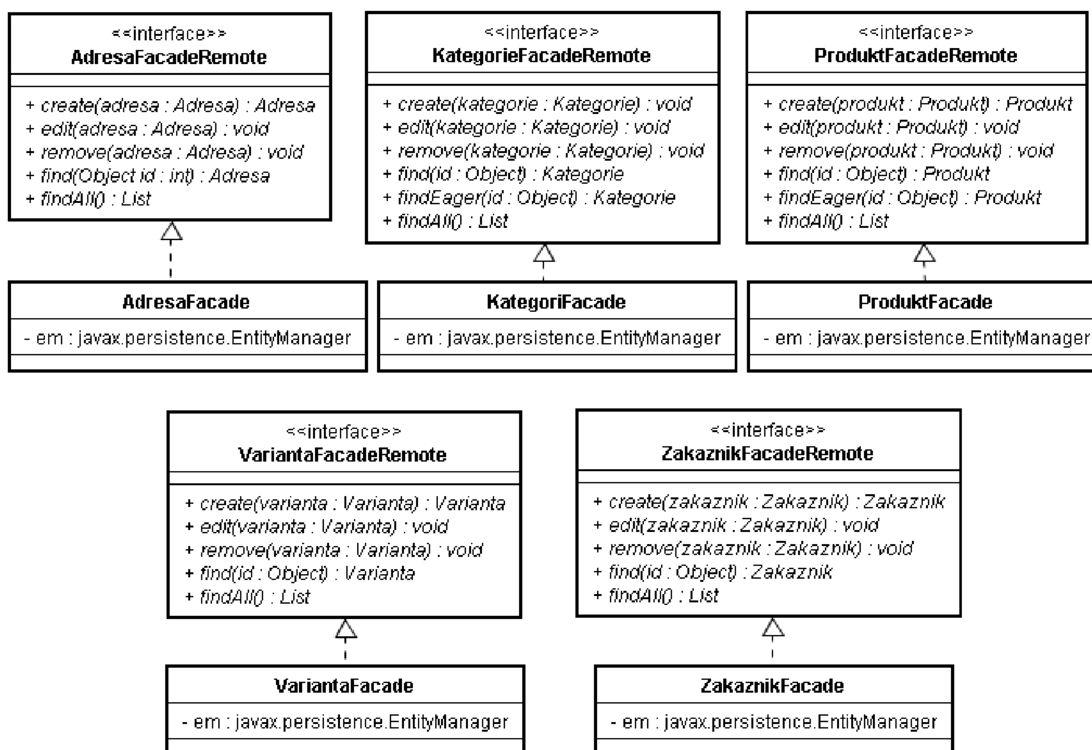
Strukturu modelu perzistentních dat implementovaného pomocí EJB 3 ukazuje diagram tříd na obrázku 6-6. Tento model vychází z modelu zobrazeném na diagramu tříd na obrázku 6-4.



Obrázek 6-6: Diagram EJB perzistentních entit

Model EJB entit rozšiřuje diagram tříd zejména o atributy a operace reprezentující vztahy. Práce se vztahy mezi entitami pomocí kolekcí vychází přímo z filozofie EJB, takový datový model je možné generovat pomocí sofistikovaného nástroje jako IDE Netbeans automaticky ze struktury databáze (uvedené na obrázku 6-5).

K jednotlivým perzistentním entitám popsaným na obrázku 6-6 je přistupováno přes session enterprise beany pomocí volání vzdálených metod. Tato rozhraní včetně jejich implementací lze generovat pomocí sofistikovaného nástroje jako IDE Netbeans automaticky přímo z modelu entit. Session beany popisuje obrázek 6-7:



Obrázek 6-7: Session beany EJB

Třídy realizující rozhraní session bean implementují všechny metody rozhraní (v diagramu nejsou z úsporných důvodů uváděny) pomocí rozhraní `javax.persistence.EntityManager` obsahujícího funkce pro práci s perzistentními entitami (vytváření, editaci, obnovu, vyhledání atd.).

Session beanám pracujícím s entitami, které dle obrázku 6-5 vstupují do vícenásobného vztahu jako majitelé vztahu (tzn. mohou vlastnit více než jednu instanci jiné entity), je kromě základních funkcí pro práci s perzistentními entitami (CRUD), přidána další funkce – metoda `findEager`, která má stejnou funkci jako metoda `find` (tedy vyhledání objektu dle parametru `id`) s tím, že navíc inicializuje násobný vztah. Tuto metodu bylo nutné přidat do session bean pro entity `Kategorie` (vícenásobný vztah s entitou `Produkt`) a `Produkt` (vícenásobný vztah s entitou `Kategorie` a `Varianta`).

Implementace metody `findEager` z rozhraní pro entitu `Produkt` (obdobně i pro entitu `Kategorie`) může vypadat následovně:

```
public Produkt findEager(Object id) {
    Produkt p = find(id);
    // inicializace vazby s entitou Kategorie
    p.getKategorieCollection().size();
    // inicializace vazby s entitou Varianta
    p.getVariantaCollection().size();
    return p;
```

```
}
```

Jak je vidět na příkladě, metoda `findEager` plní stejnou funkci jako metoda `find` s tím rozdílem, že inicializuje kolekce reprezentující násobné vztahy (zavoláním metody `size` pro každou kolekci).

Využití této metody ilustruje následující příklad vytvoření nové instance třídy `Produkt` a její provázání s objektem třídy `Kategorie` (tedy zařazení nového produktu do stávající kategorie):

```
// uložit produkt a získat jeho perzistentní podobu
Produkt p = produktFacade.findEager(
                  produktFacade.create(produkt).getId());
// získat stávající kategorii
Kategorie k = kategorieFacade.findEager(idKat));
// provázat produkt s kategorií = vložit produkt do kolekce produktů
k.getProduktCollection().add(p);
// uložit kategorii, změny v kolekci produktů
kategorieFacade.edit(k);
// provázat kategorii s produktem
p.getKategorieCollection().add(k);
```

Tento příklad (převzatý z ukázkové aplikace implementované v JSF – viz níže) ukazuje typický styl práce s vazbami entit implementovaných technologií EJB 3.

Kromě uvedených základních operací by datový model reprezentovaný EJB mohl obsahovat další aplikační logiku, protože ale EJB není hlavním tématem této práce, bude většina aplikační logiky přesunuta do řadičů implementovaných ve frameworcích, aby se tak více ukázaly možnosti a postupy, jaké lze ve frameworcích použít.

6.3.2 Implementace v JSF frameworku

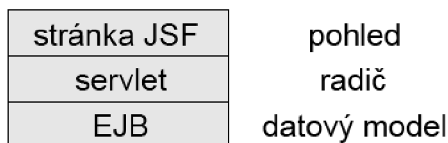
Podrobněji se implementací v JSF frameworku a postupy v návrhu aplikace zabývá samostatná příloha ***Tutoriál JavaServer Faces***.

6.3.2.1 MVC architektura aplikace

JSF framework implementuje architekturu MVC inspirovanou strategií *Dispatcher View* (volání aplikační logiky z pohledu). Protože je JSF framework zaměřen zejména k prezentaci dat a poskytování podpůrných prostředků pro tvorbu uživatelských rozhraní, tedy pohledů (*View*) v obecném smyslu MVC, bylo pro dosažení aplikační architektury typu MVC nutné zkombinovat jej s jinými nástroji. Pro implementaci datového modelu aplikace byla zvolena technologie EJB 3. Řídící

logika (řadiče) aplikace je naimplementována pomocí standardních Java tříd komunikujících s datovým modelem pomocí rozhraní `javax.ejb.EJB` a s JSF rozhraním pomocí JSF bean.

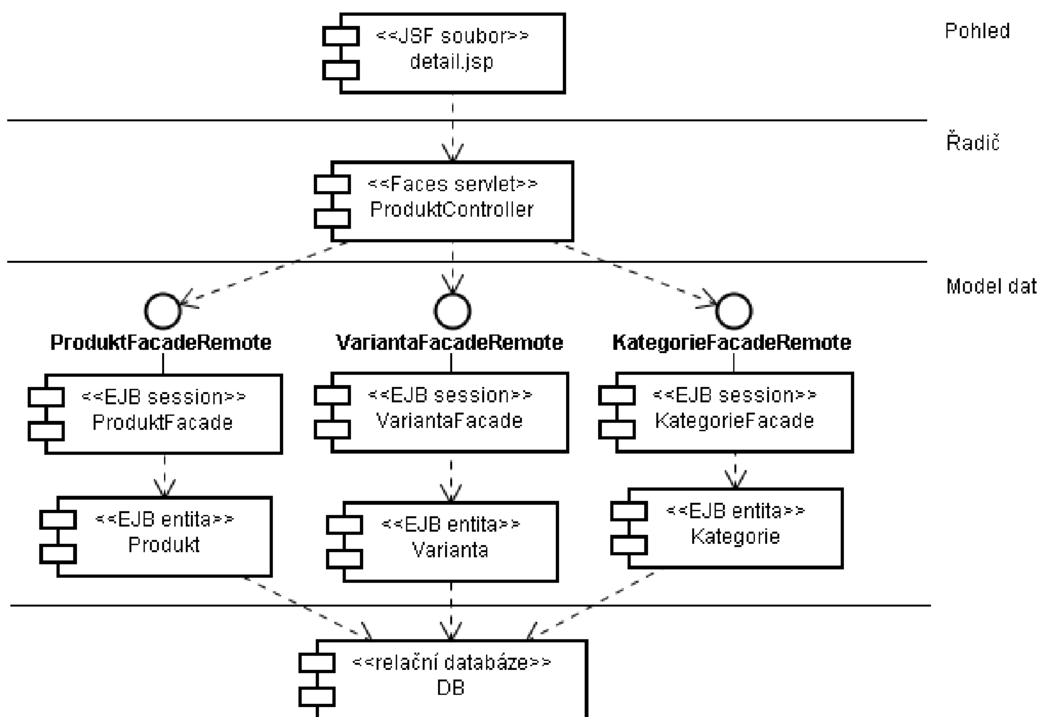
Takové spojení vytváří ideální celek pro tvorbu webových aplikací založených na architektuře MVC. Nezávislé oddělení jednotlivých modulů umožňuje separátní vývoj ve více týmech nezávisle na sobě.



Obrázek 6-8: Architektura MVC v ukázkové aplikaci využívající JSF

Na obrázku 6-8 jsou vidět jednotlivé vrstvy architektury MVC a technologie, kterými jsou v aplikaci implementovány.

Diagram komponent na obrázku 6-9 ilustruje implementaci architektury MVC v ukázkové aplikaci na příkladě *zobrazení detailu zboží*:



Obrázek 6-9: Komponentový diagram implementace zobrazení detailu zboží

Jak je vidět na tomto příkladě, pohled realizovaný JSF stránkou `detail.jsp` komunikuje s datovým modelem skrze řadič implementovaný třídou `ProduktController` pomocí rozhraní JSF bean (taková třída je také označována jako *Faces servlet*). Třída `ProduktController` tedy zastává funkci prostředníka mezi pohledem a modelem (ve smyslu MVC), takováto komponenta je

označována jako *business helper*. Mapování třídy `ProduktController` na JSF managed bean `produktyJSFBean` v konfiguračním souboru JSF frameworku potom vypadá následovně:

```
<managed-bean>
   <managed-bean-name>produktyJSFBean</managed-bean-name>
   <managed-bean-class>jsf.ProduktController</managed-bean-class>
   ...
</managed-bean>
```

Pohled (JSF stránka `detail.jsp`) využívá bean následovně:

```
<h:outputText value="#{produktyJSFBean.produkt.nazev}" /> ...
```

V příkladě přistupuje pohled k atributu `produkt` třídy `ProduktController` (viz dále).

Na diagramu z obrázku 6-9 je dále vidět závislost Faces servletu na EJB rozhraní `ProduktFacadeRemote`, `VariantaFacadeRemote` a `KategorieFacadeRemote`. Skrze tato rozhraní komunikuje řadič ve formě Faces servletu s datovým modelem ve formě EJB, získaná data potom předává pohledu mechanismem popsaným výše.

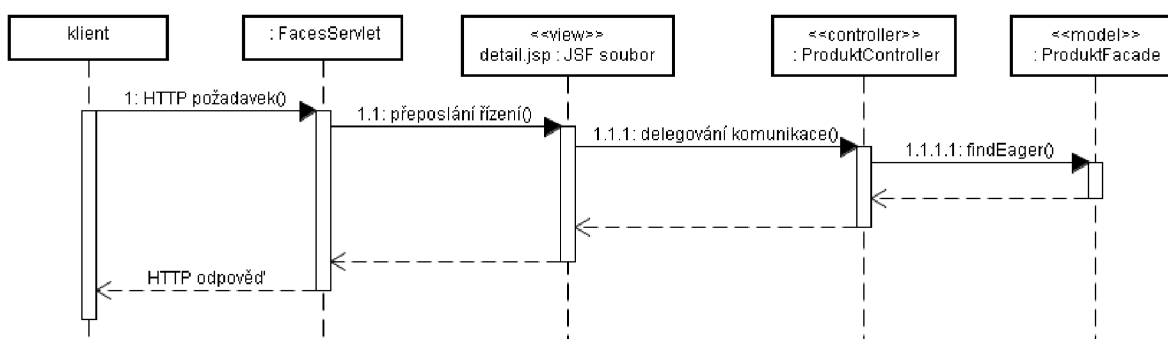
Příklad komunikace řadiče s EJB modelem je následující:

```
...
// získat rozhraní pomocí notace EJB 3 @EJB
@EJB
private ProduktFacadeRemote produktFacade;
...
// aktuálně zobrazovaný produkt
private Produkt produkt;
...
public String detail() {
   // získáme produkt pomoci předaneho ID
   ...
   // pomocí EJB rozhraní získáme instanci EJB entity s daným ID
   produkt = produktFacade.findEager(produktId);
   ...
   return "detail";      // návratový řetězec určuje pohled
}
...
public Produkt getProdukt() {
...
```

Řádek `return "detail";` určuje pohled `detail.jsp` na základě definice navigace v JSF konfiguračním souboru:

```
<navigation-rule>
...
   <navigation-case>
       <from-outcome>detail</from-outcome>
       <to-view-id>/detail.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

Výše uvedený příklad z ukázkové aplikace doplňuje sekvenční diagram na obrázku 6-10 ukazující komunikaci mezi zmíněnými komponentami:



Obrázek 6-10: Sekvenční diagram ukazující komunikaci mezi komponentami na ukázkovém příkladě v aplikaci JSF

Jak je vidět na obrázku 6-10, prvním krokem ve zpracování HTTP požadavku v JSF frameworku je přeposlání řízení pohledu ve formě JSF souboru. Toto přeposlání provádí objekt třídy `FacesServlet`, který v JSF frameworku implementujícím vzor *Front Controller* (centrální zpracování požadavku v jedné komponentě řadiče) plní funkci hlavního řadiče. Pohled získá data delegováním aplikační logiky na třídu `ProduktController`, která plní funkci řadiče ve smyslu MVC architektury ukázkové aplikace – komunikuje přímo s datovým modelem, a odpověď je posléze vrácena klientovi.

Podkapitola 6.3.2.1 se věnovala popisu implementace architektury MVC v JSF ukázkové aplikaci a na vybraném příkladě ukázala propojení a komunikaci mezi jednotlivými vrstvami MVC. Následující dvě podkapitoly se věnují podrobněji implementaci řadičů v ukázkové JSF aplikaci a jejich komunikaci s pohledy.

6.3.2.2 Třídy řadičů – aplikační vrstva

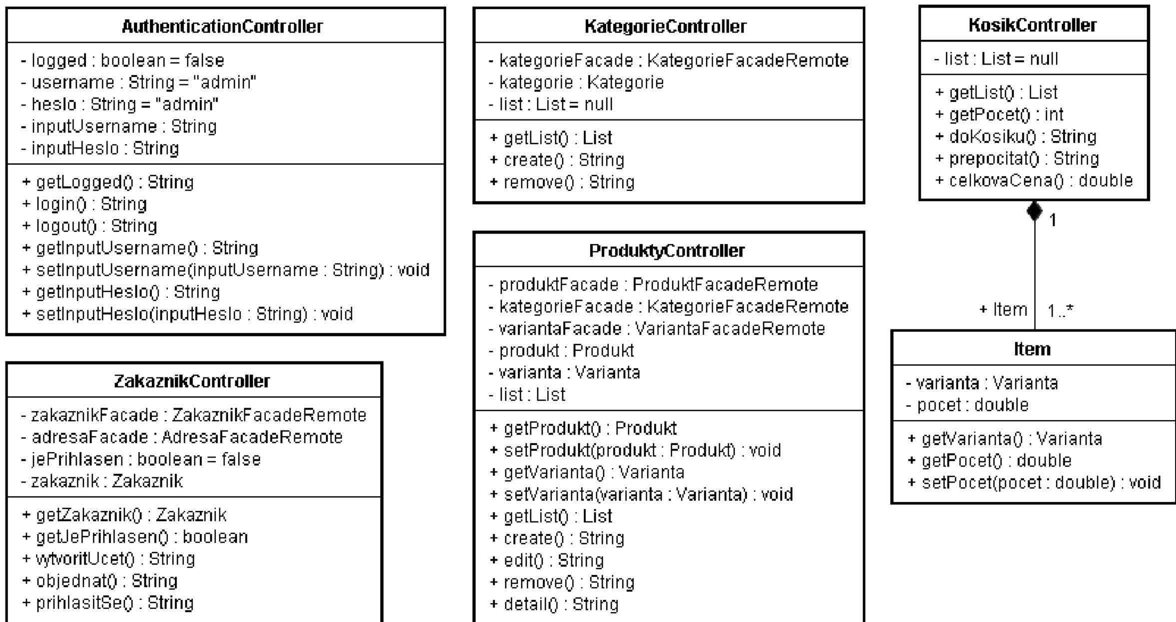
Třídy realizující funkci řadiče aplikace pracují s datovým modelem prostřednictvím rozhraní session bean EJB a vybírají uživatelské pohledy pomocí libovolných uživatelem zvolených návratových hodnot typu `String`, které zohledňuje navigační systém JSF – viz podkapitola 6.3.2.3.

Při návrhu řadičů byl zohledněn model případů užití na obrázku 6-3 v tom smyslu, že metody řadičů implementují jednotlivé případy užití, a business diagram tříd z obrázku 6-1 zejména pro přehledné rozdělení korespondující s datovými entitami, které jsou vlastníky násobných vztahů.

Aplikační logika ukázkové aplikace byla tedy rozdělena mezi následující řadiče:

- **`AuthenticationController`** slouží pro přihlašování v administrační sekci, uživatelské jméno a heslo je zde zadané přímo do kódu; lepším řešením pro reálnou aplikaci by bylo provázání se session beanou spravující perzistentní entitu udržující přihlašovací údaje jednotlivých administrátorů. Přihlašování zákazníka je zde realizováno třídou `ZakaznikController` (viz dále).
- **`ZakaznikController`** implementuje správu zákaznického účtu včetně přihlašování a objednání zboží z nákupního košíku. Tento řadič má na starost datové domény `Zakaznik` a `Adresa` ke kterým přistupuje skrz EJB rozhraní `ZakaznikFacadeRemote` a `AdresaFacadeRemote`. Řadič byl pojmenován podle entity `Zakaznik`, protože ta je majitelem násobného vztahu s entitou `Adresa` (udávající fakturační a dodací adresu registrovaného zákazníka).
- **`KategorieController`** má na starosti datovou entitu Kategorie, ke které přistupuje skrz rozhraní `KategorieFacadeRemote`; komunikuje s pohledem pro výpis kategorií v obchodě a realizuje správu kategorií v administrační sekci aplikace.
- **`ProduktyController`** implementuje všechny akce spojené se správou a výpisem zboží. Je nazván podle entity `Produkt`, která je majitelem násobného vztahu s entitami `Varianta` a `Kategorie`, s těmito entitami komunikuje skrz rozhraní `ProduktFacadeRemote`, `VariantaFacadeRemote` a `KategorieFacadeRemote`.
- **`KosikController`** je řadič implementující správu nákupního košíku pro uživatelské sezení. Obsahuje metody pro přidání položky do košíku, přepočítání resp. změnu množství u položek v košíku a metodu pro výpočet celkové ceny všech položek v košíku. Třída tohoto řadiče obsahuje veřejnou třídu **`Item`** reprezentující položku v nákupním košíku, tato je dána atributem `varianta`, ukazujícím na objekt typu `Varianta`, a `pocet` udávající množství této položky v košíku.

Diagram tříd na obrázku 6-11 ukazuje třídy implementující řadiče v ukázkové JSF aplikaci.

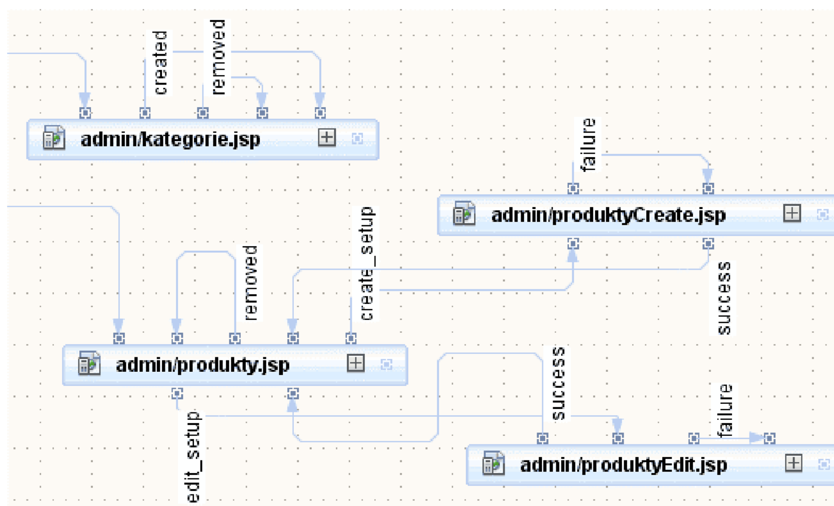


Obrázek 6-11: Třídy řadiče realizující aplikační logiku v ukázkové aplikaci

Metody tříd řadiče reprezentující funkčnost aplikace (korespondující s případy užití) na obrázku 6-11 a vrací vždy textový řetězec pro následné použití v navigaci (viz níže).

6.3.2.3 Pohledy uživatelského rozhraní – prezentační vrstva

Implementace pohledů uživatelského rozhraní pomocí JSF frameworku zpřehledňuje zdrojový kód rozhraní (díky použití parametrizovaných JSF tagů), ve velké míře vyjasňuje navigaci v aplikaci, která je striktně definovaná v samostatném souboru `faces-config.xml` a pro zjištění toku dat stačí projít tento jediný soubor bez nutnosti náročného prohledávání zdrojových textů všech souborů. IDE Netbeans navíc dokáží tuto navigaci znázornit a spravovat graficky pomocí tzn. PageFlow diagramu.



Obrázek 6-12: PageFlow diagram v IDE Netbeans

Na obrázku 6-12 je vidět tok navigace mezi stránkami pro správu, vytváření a mazání, kategorií – `admin/kategorie.jsp` a správu produktů zboží – `admin/produkty.jsp` včetně založení nového produktu – `admin/produktyCreate.jsp` a editace stávajícího – `admin/produktyEdit.jsp`. Textové řetězce jdoucí z navigačního toku (znázorněného šipkami) jsou uživatelem definované výstupy z metod řadiče, na základě kterých se provádí výběr JSF stránky pohledu.

Část příkladu na obrázku 6-12 by mohla být implementována následovně:

Na JSF stránce `admin/produkty.jsp` se vyskytuje následující JSF tag, který vytvoří hypertextový odkaz vyvolávající jako akci metodu `editSetup` řadiče mapovaného na bean `produktyJSFBean`, tedy třídy `ProduktController`:

```
<h:commandLink action="#{produktyJSFBean.editSetup}" value="editovat">
      <f:param name="editedId" value="#{item.id}" />
</h:commandLink>
```

Tento odkaz bude mít jako parametr `editedId` udávající hodnotu ID produktu určeného k editaci. Tento bude využit v metodě `editSetup` řadiče `ProduktController`:

```
public String editSetup () {
   // zíkání kontextu Faces servletu a jeho parametrů
   FacesContext context = FacesContext.getCurrentInstance();
   Map requestParams =
             context.getExternalContext().getRequestParameterMap();
   // zjistit hodnotu parametru editedId
Integer editedId =
       Integer.parseInt((String) requestParams.get("editedId"));
// příprava pro editaci produktu
...
// návratový řetězec
return "edit_setup";
```

Přesměrování z pohledu `admin/produkty.jsp` na pohled `admin/produktyEdit.jsp` znázorněné na obrázku 6-12 šipkou na základě řetězce `edit_setup` je deklarováno v JSF konfiguračním souboru následovně:

```
<navigation-rule>
    <from-view-id>/admin/produkty.jsp</from-view-id>
    <navigation-case>
        <from-outcome>edit_setup</from-outcome>
```

```
            <to-view-id>/admin/produktyEdit.jsp</to-view-id>
        </navigation-case>
        ...
    </navigation-rule>
```

Obdobně i pro ostatní příklady navigace.

6.3.2.4 Výhody JSF frameworku

- centralizovaná definice navigace v aplikaci
- přehlednost zdrojového kódu založeného na znovupoužitelných parametrizovaných komponentách volaných jako tagy XML
- možnost integrace s prakticky jakoukoliv jinou technologií
- standard Java EE – zaručen budoucí vývoj a vysoká podpora
- velké množství standardních komponent a nástrojů (validátory, konvertery, …)
- možnost definice vlastních komponent a nástrojů
- podpora technologie EJB ve formě tzv. *managed-beanů*

6.3.2.5 Nevýhody JSF frameworku

- použití JSF tagů v uživatelském rozhraní není striktní, je možné vkládat klasické příkazy JSP, což může vést k nepřehlednosti kódu a k nedodržování postupů architektury MVC.
- generuje uživatelské rozhraní založené na JavaScriptu, což může být pro některé aplikace nepřijatelné, po zakázání JavaScriptu v prohlížeči je aplikace zcela nepoužitelná
- některé aspekty nedotažené, např. špatná navigace skrze záložky v prohlížeči apod.
- prakticky nepoužitelné samostatně, pro sofistikovanější řešení (např. architekturu MVC) je nutná integrace s dalšími technologiemi či frameworky
- nepříliš vhodné pro některé funkce jako např. přihlašování uživatelů, nenabízí tedy komplexní služby pro tvorbu uživatelských rozhraní

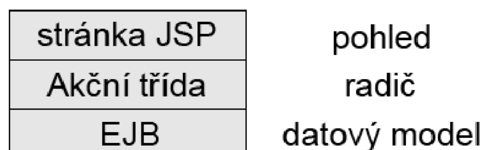
6.3.3 Implementace ve Struts 2 frameworku

Podrobněji se implementací ve Struts 2 frameworku a postupy v návrhu aplikace zabývá samostatná příloha ***Tutoriál Struts 2***.

6.3.3.1 MVC architektura aplikace

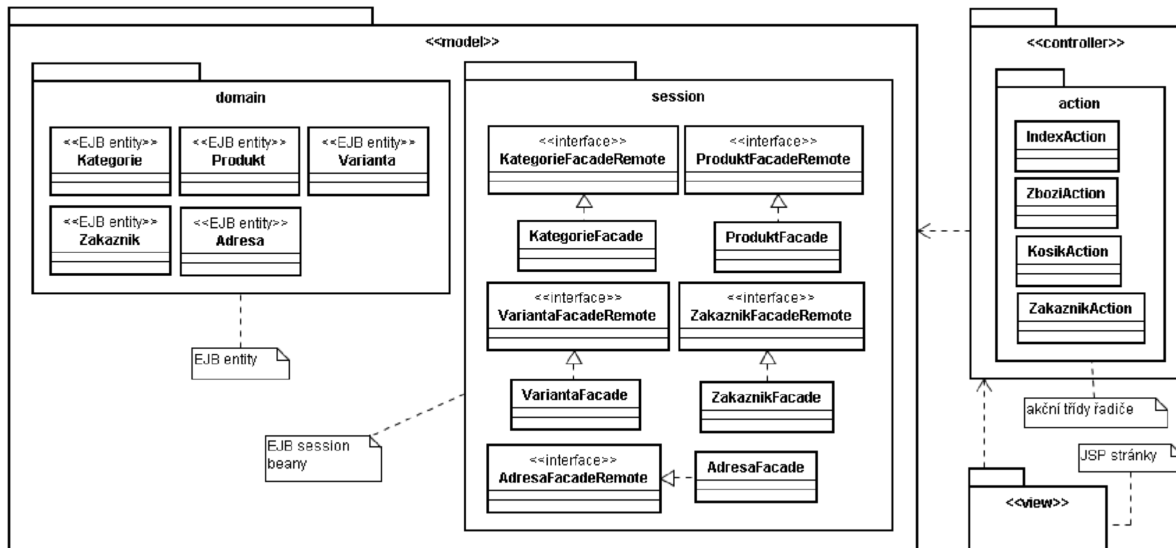
Framework Struts 2 implementuje návrhový vzor Front controller s využitím strategie Service to worker (volání aplikační logiky z řadiče). Dále je v něm využito vzoru *Intercepting filter* (přidání zodpovědnosti komponentě umístěné před zpracováním HTTP požadavku) implementovaného pomocí interceptorů.

Struts framework můžeme řadit spíše do prezentačních frameworků, protože neobsahuje žádnou podporu pro datové modelování. Pro datový model bylo tedy opět využito technologie EJB 3.0. Aplikaci vytvářenou ve Struts frameworku můžeme z hlediska MVC architektury rozdělit následovně: řadiče jsou implementovány pomocí akčních tříd a pomocných tříd, kterých akční třída využívá zejména ke komunikaci s prezentační a datovou vrstvou, prezentační vrstva je tvořena stránkami JSP s tagy z uživatelských knihoven. Pro komunikaci mezi vrstvami využijeme strategii Service to worker. Rozdělení technologií do vrstev MVC je zobrazeno na obrázku 6-13:



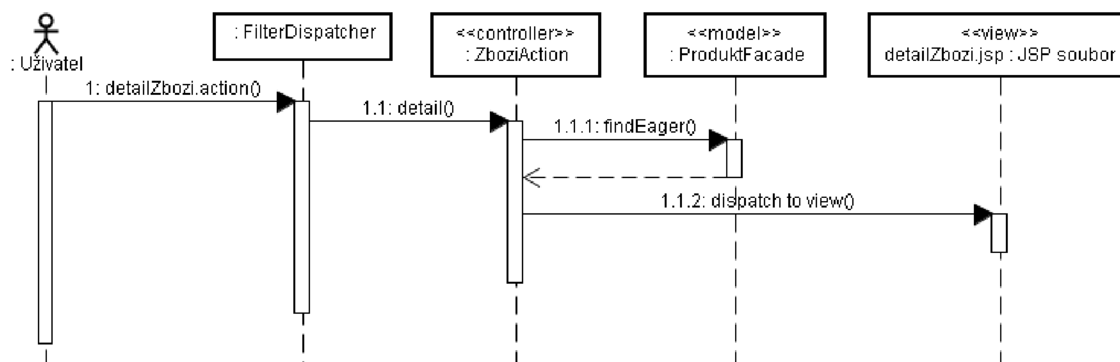
Obrázek 6-13: Architektura MVC v ukázkové aplikaci frameworku Struts 1

Na obrázku 6-14 je diagram balíčků znázorňující rozdělení aplikace do MVC architektonického modelu a závislosti jednotlivých MVC vrstev. Zobrazené třídy jsou uváděny pro přehlednost pouze jako hlavičky, kompletní diagramy tříd modelu jsou na obrázcích 6-6 a 6-7 (výše), diagram akčních tříd řadiče je na obrázku 6-16 (níže).



Obrázek 6-14: Diagram balíčků MVC architektury aplikace ve Struts 2 frameworku

Příklad komunikace mezi jednotlivými komponentami, zobrazení detailu zboží, je znázorněn sekvenčním diagramem na obrázku 6-15:



Obrázek 6-15: Sekvenční diagram zobrazení detailu zboží ve Struts 2

Zobrazení detailu zboží reprezentuje typický případ komunikace mezi vrstvami MVC architektury aplikace. Uživatel zašle požadavek na akci, v tomto případě akce `detailZbozi`, požadavek je zpracován hlavním řadičem (ve smyslu vzoru Front controller Struts 2 frameworku) reprezentovaným instancí třídy `FilterDispatcher`, který vyvolá spuštění metody akční třídy reprezentující řadič (ve smyslu MVC architektury aplikace). Akční třída komunikuje s modelem pro získání dat a následně vyvolá konkrétní pohled, zde JSP stránku `detailZbozi.jsp`, který je vrácen jako odpověď uživateli.

Z diagramu je dobře vidět postavení řadiče reprezentovaného akční třídou jako prostředníka získávajícího data z modelu a vyvolávající pohled, kterému jsou data předána, tedy implementace strategie Service to worker.

V ukázkové aplikaci je většina aplikační logiky přesunuta do řadičů (tedy akčních tříd) namísto modelu, aby víc ukázala práci ve frameworku (EJB není hlavním tématem této práce).

6.3.3.2 Datový model – vrstva EJB

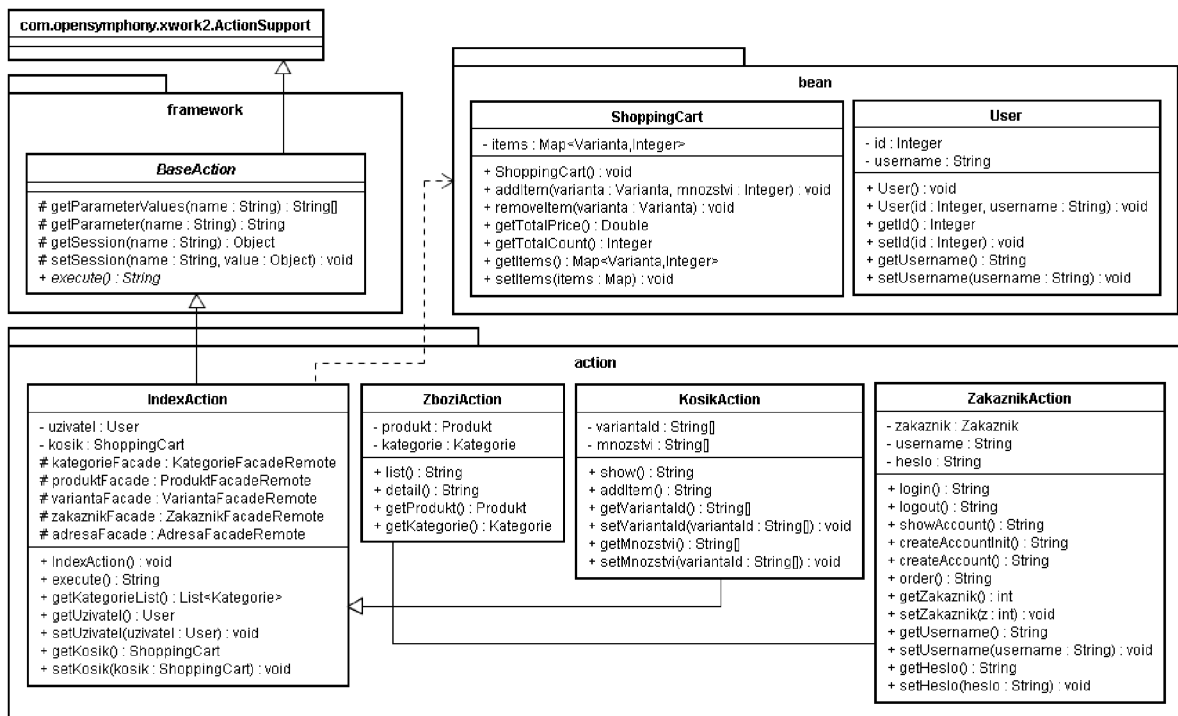
Protože bylo pro implementaci datového modelu využito technologie EJB, je potřeba získat přístup k EJB beanám. Bohužel nemůžeme použít standardní přístup (*dependency injection*) pomocí anotace `@javax.ejb.EJB TridaSessionBeany beana`, protože tento přístup lze použít pouze pro třídy v EJB nebo servlet kontejneru (servlety, JSP stránky, JFS *managed-beany*). V aplikaci založené na Struts frameworku musíme pro získání přístupu k EJB beaně využít *JNDI* (Java Naming and Directory Interface) kontext, jak je vidět na příkladu získání objektu EJB fasády:

```
Context c = new InitialContext();
ProduktFacadeRemote produktFacade = (ProduktFacadeRemote)
                    c.lookup(ProduktFacadeRemote.class.getName());
```

Tento postup získání přístupu k EJB musí využít všechny akční třídy, které potřebují přistupovat k datovému modelu. V ukázkové aplikaci toto obstarává třída `BaseAction` (viz dále), kterou rozšiřují všechny použité akční třídy.

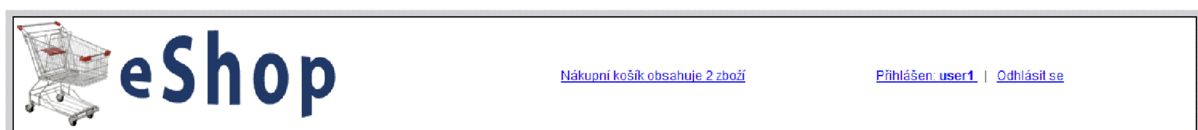
6.3.3.3 Třídy řadičů – aplikační vrstva

Diagram akčních tříd reprezentující řadiče je na obrázku 6-16:



Obrázek 6-16: Diagram tříd aplikační logiky

Na obrázku 6-16 jsou vidět všechny akční třídy vložené do balíčku `action`. Všechny rozšiřují třídu `IndexAction`, která obsahuje atributy a metody využívané všemi akčními třídami. Jedná se o atributy přistupující k datovému modelu prostřednictvím fasád a metody přístupu a nastavení nákupního košíku a údajů přihlášeného uživatele (informace o stavu nákupního košíku a přihlášení uživatele se zobrazuje na každé stránce, jak ukazuje obrázek 6-17, musí být tedy dostupné ze všech akčních tříd):



Obrázek 6-17: Informace o nákupním košíku a přihlášení uživatele se zobrazují na každé stránce

Třídy reprezentující nákupní košík a přihlášeného uživatele se nachází v balíčku `bean`, jedná se o třídy `ShoppingCart` pro nákupní košík a `User` pro uživatele.

Třída `IndexAction` dále rozšiřuje abstraktní třídu `BaseAction` z balíčku `framework` a implementuje její abstraktní metodu `execute` pro implicitní akci akční třídy. Tato implicitní akce je volána, pokud v konfiguračním souboru není uveden atribut `method` pro určení explicitní metody spojené s požadovanou akcí. Pro akci `home`, která zobrazí úvodní stránku vypadá konfigurace následovně:

```
<action name="home" class="action.IndexAction">
    <result>/index.jsp</result>
</action>
```

Pokud je atribut `method` uveden, je pro provedení akce volána metoda určená hodnotou tohoto atributu. Například pro akci `detailZbozi` zobrazující detailní výpis zboží vázanou na metodu detail akční třídy `ZboziAction` bude konfigurace vypadat takto:

```
<action name="detailZbozi" method="detail"
        class="action.ZboziAction">
       <result>/detailZbozi.jsp</result>
</action>
```

Abstraktní třída `BaseAction` z balíčku `framework` obsahuje sadu metod pro práci s parametry požadavku a session, k čemuž využívá třídu `ActionContext` popsanou výše v kapitole *Struts framework*. Tyto metody jsou hojně využívané ve všech akčních třídách, a proto je rozšiřování od třídy `BaseAction` velmi užitečné. Třída sama rozšiřuje jednu z implementací rozhraní akčních tříd, třídu `ActionSupport`, a tak zajišťuje, že všechny třídy od ní rozšířené, budou skutečně akčními třídami.

Akční třídy aplikace jsou příkladem více-jednotkových akčních tříd, což znamená, že obsahují implementace více než jedné akce (každá akce odpovídá jednomu případu použití). Tento přístup je výhodnější z hlediska přehlednosti a správy akcí. Akční třídy v tomto případě neimplementují pouze jednu implicitní metodu akce (`execute`), ale implementují více různých metod, které seskupují podle charakteru akce či závislostí na konkrétních částech modelu.

Aplikace byla v tomto smyslu rozdělena do akčních tříd **`ZboziAction`** obsahující akce pro výpis seznamu zboží z dané kategorie (metoda `list`), zobrazení detailních informací o produktu včetně možnosti přidání do košíku (metoda `detail`), **`KosikAction`** s akcemi pro zobrazení obsahu košíku (metoda `show`) a vložení položek do košíku (metoda `addItem`), **`ZakaznikAction`** zpracovávající akce spojené s uživatelským účtem, jako je objednání zboží z nákupního košíku

(metoda `order`), přihlášení a odhlášení uživatele (metody `login` a `logout`), vytvoření nového uživatelského účtu (metoda `createAccount` a `createAccountInit` pro přípravu datových struktur) a zobrazení údajů uživatelského účtu (metoda `showAccount`).

Akční třídy kromě metod reprezentujících akce obsahují také atributy a metody pro plnění pohledů daty. Pokud pohled pouze prezentuje data z akční třídy, stačí ke každému atributu připojit zapouzdřující metodu getter jako je to například u třídy `ZboziAction` a jejího atributu `produkt`, a příslušné metody `getProdukt`, jež slouží pro zobrazení zboží v pohledu. Obsahuje-li pohled formulář plnící datové struktury, je třeba k atributu připojit také odpovídající zapouzdřující metodu setter pro nastavení hodnoty atributu, jako je tomu například u třídy `ZakaznikAction`.

Zvláštním případem je třída `KosikAction`, jejíž atributy jsou pole (`String[]`). Je to proto, že detailní zobrazení zboží obsahuje formulář pro možnost přidání několika variant zboží do košíku najednou, jak je vidět na obrázku 6-18:

varianta 1 produktu 1	21 Kč	0	Přidat do košíku
varianta 2 produktu 1	22 Kč	0	Přidat do košíku
varianta 3 produktu 1	3 Kč	0	Přidat do košíku

Obrázek 6-18: Možnost přidání více variant do košíku v detailu zboží

6.3.3.4 Pohledy uživatelského rozhraní – prezentační vrstva

Prezentační vrstva je ve Struts frameworku tvořena stránkami JSP využívající tagy z knihoven uživatelských tagů. Navigace a obsah těchto stránek je řízen akčními třídami. Konfigurace provázání jednotlivých pohledů ke konkrétním akčním třídám se provádí v konfiguračním souboru frameworku Struts 2 `struts.xml`.

6.3.3.5 Výhody Struts frameworku

- výborná podpora, kvalitní dokumentace, silná komunita
- deklarativní přístup k návrhu aplikace prostřednictvím konfiguračního souboru včetně centralizované definice navigace
- možnost využití konfigurací pomocí Java anotací
- akční třídy s daty pro uživatelský pohled
- knihovny uživatelských tagů pro tvorbu uživatelského rozhraní
- využití zdrojových svazků textových zpráv pro možnost internacionalizace a centralizované správy zpráv v aplikaci (titulky stránek, návěští, popisky, chybové zprávy apod.)
- možnost využití šablonovacího rámce Tiles
- možnost využití rámce Validator pro validaci formulářových dat se standardními i uživatelsky definovanými ověřovacími pravidly

- globální a deklarativní ošetřování výjimek

6.3.3.6 Nevýhody Struts frameworku

- neúplná kompatibilita se starší verzí, verze Struts 2 není stále tak rozšířena
- složitá komunikace s datovým modelem (žádná podpora EJB apod.), nutnost využití JNDI
- pohledy jsou stránky JSP bez jakýchkoli omezení, což může vést k nedodržování postupů architektury MVC

7 Srovnání a zhodnocení

Každý ze zkoumaných frameworků je jiný a implementuje poměrně jinou strategii. Jak Struts tak i JSF jsou prezentační frameworky, poskytují tedy prostředky pro podobnou třídu úloh, i když každý svým vlastním způsobem. Z hlediska srovnání a zhodnocení je třeba zaměřit se na konkrétní aspekty poskytovaných prostředků pro vývoj a srovnat je v rámci dané funkcionality.

7.1 MVC přístup

Oba zkoumané frameworky implementují architektonický model MVC. Liší se však ve strategiích komunikace komponent.

JSF framework implementuje strategii Dispatcher view, zatímco Struts framework strategii Service to worker, což úzce souvisí se zaměřením frameworků.

JSF je zaměřen více na tvorbu uživatelských pohledů, které jsou tedy i vstupním bodem v komunikaci (aplikační logika je volána z pohledu). Výhodou tohoto přístupu je snadná definice navigace mezi stránkami, nevýhodou zůstává závislost na prezentační vrstvě.

Struts framework umísťuje řízení do řadičů ve formě akčních tříd, které komunikují s datovým modelem. Výsledný pohled je určen na základě vrácené hodnoty z řadiče. Výhodou tohoto přístupu je nezávislost na prezentační vrstvě.

Z tohoto hlediska jsem zastáncem strategie Service to worker, zejména pro nezávislost na prezentační vrstvě, což může být velmi užitečné chceme-li systém publikovat jako více nezávislých prezentací (například pomocí JSP, XML apod.).

7.2 Aplikační logika

K nejdůležitějším charakteristikám, které ovlivňují vývoj, patří prostředky pro implementaci aplikační logiky. Každý z frameworků nabízí v tomto smyslu odlišnou strategii.

JSF framework pro implementaci aplikační logiky využívá technologie backing beanů. Jedná se o obyčejné Java objekty, které přistupují k datovému modelu a které jsou prostřednictvím konfigurace dostupné pohledům. Ačkoliv z hlediska MVC architektury implementují backing beany vrstvu řadičů, neobsahují žádné prostředky pro řízení toku aplikace, který mají plně na starosti pohledy. Toto omezení může být nevýhodné, protože tak degraduje řadiče na pouhé přístupové body k datovému modelu. Výhodou backing beanů v JSF frameworku je dobrá podpora EJB: beany běží v JSF aplikačním kontejneru a mohou tak k EJB snadno přistupovat pomocí mechanismu dependency injection, tedy skrze anotace `@javax.ejb.EJB`. Tato výhoda ve Struts frameworku chybí.

Struts framework implementuje řadiče pomocí akčních tříd, které jsou výkonným jádrem aplikace. Akční třídy komunikují přímo s modelem, vybírají výsledný pohled, kterému poskytují data.

Velkou výhodou tohoto přístupu je centralizované řízení toku aplikace a snadné plnění a získávání dat z pohledů pomocí zapouzdřujících metod getter a setter ke každému atributu akční třídy. Nevýhodou akčních tříd je skutečnost, že se jedná o obyčejné Java třídy, které neběží v aplikačním kontejneru, což činí náročnější přístup k EJB (pomocí JNDI) či parametrům požadavku a session.

Další výhodou Struts frameworku je implementace návrhového vzoru Intercepting filter pomocí tzv. interceptorů, skrze které lze provádět předzpracování požadavku před jeho předáním řadiči a přednastavení akčních tříd. Tento přístup může být využit při zabezpečení (autentizace, ošetření vstupu od uživatele) apod. Ačkoliv je tento vzor implementován všemi Java EE aplikacemi (pomocí třídy `Filter`), je vlastní řešení nabízené Struts frameworkem velmi elegantní a patří k jeho nesporným výhodám.

I k validaci vstupních dat zaujaly oba frameworky rozdílný přístup. JSF umožňuje vkládat validační prvky přímo do pohledů, což může mít výhodu v definici umístění a formátování chybových zpráv, zatímco Struts framework nabízí strategii sofistikované validace definované v samostatném XML dokumentu pomocí rámce Validator. Tento přístup přináší nesporné výhody v centralizaci a přehledné správě validace.

7.3 Prezentační vrstva

Z hlediska prostředků pro tvorbu uživatelských rozhraní nabízí oba frameworky vlastní knihovny pro usnadnění a částečnou automatizaci vytváření pohledů prezentační vrstvy. V rozsahu standardně nabízených UI prvků jednoznačně vítězí JSF framework, který umožňuje i snadné definice či redefinice uživatelských komponent UI. Struts nabízí jen omezenou množinu pro tvorbu základních prvků IU, často se musí opírat o standardní postupy JSP stránek.

K nevýhodám UI nástrojů JSF frameworku však patří jejich značná provázanost na JavaScript (při vypnutí podpory JavaScriptu v prohlížeči se web stává nefunkčním). Další nevýhodou JSF je značná složitost nástrojů pro tvorbu UI, jejichž naučení zabere hodně času na rozdíl od intuitivních prvků navazujících na JSP poskytovaných frameworkem Struts.

7.4 Srovnání pomocí matice

Srovnání pomocí matice je standardní technikou pro porovnání a hodnocení různých frameworků pomocí jednotlivých aspektů.

Zavedeme následující systém hodnocení:

- **1:** chabé splnění, nebo není splněno vůbec
- **2:** vlastnost není bez podpory, tato je ale velmi slabá

- **3:** vlastnost je splněna
- **4:** vlastnost je slněna dobře
- **5:** vlastnost je splněna velmi dobře

Jako srovnávací kritéria zvolíme následující aspekty:

- **přístup k datovému modelu** – tato vlastnost říká, jakou podporu nabízí konkrétní framework pro prácí s datovým modelem.
- **podpora zabezpečení** – jak sofistikované prostředky pro zabezpečení framework poskytuje (řízení autentizace, validace vstupních dat, apod.)
- **implementace architektury MVC** – do jaké míry lze v aplikacích vyvíjených ve frameworku aplikovat postupy architektury MVC (dělba práce na jednotlivých vrstvách, zatížení vrstev, nezávislost komponent, apod.)
- **prostředky pro tvorbu UI** – jak kvalitní prostředky framework nabízí pro ulehčení a automatizaci tvorby uživatelských rozhraní
- **snadnost naučení** – jak jednoduchý je framework pro pochopení, rychlost naučení. Hodnocení jsem prováděl na základě vlastní zkušenosti v teoretické části.
- **dokumentace** – dostupnost, aktuálnost a pokrytí projektové dokumentace. Hodnocení jsem prováděl na základě vlastní zkušenosti v teoretické části.
- **podpora** – kvalita dostupných diskusních fór, velikost a rozšíření komunity uživatelů, oficiální podpora pro řešení problému, správa chyb. Hodnocení jsem prováděl na základě vlastní zkušenosti v teoretické části.
- **rozšíření a uplatnění** – množství pracovních nabídek, využití frameworku v praxi. Hodnocení na základě aktuálních informací pracovních nabídek na internetu.

Vlastnost	**JSF**	**Struts**	*Váha*
přístup k datovému modelu	4	1	5
podpora zabezpečení	3	5	5
implementace architektury MVC	3	5	4
prostředky pro tvorbu UI	5	3	3
snadnost naučení	3	5	2
dokumentace	5	3	3
podpora	4	5	2
rozšíření a uplatnění	3	5	3

Z těchto hodnot vypočítáme výsledné skóre:

Vlastnost	**JSF**	**Struts**	*Váha*
přístup k datovému modelu	4 × 5 = 20	2 × 5 = 10	5
podpora zabezpečení	3 × 5 = 15	5 × 5 = 25	5
implementace architektury MVC	3 × 4 = 12	5 × 4 = 20	4
prostředky pro tvorbu UI	5 × 3 = 15	3 × 3 = 9	3
snadnost naučení	3 × 2 = 6	5 × 5 = 25	2
dokumentace	5 × 3 = 15	3 × 3 = 9	3
podpora	4 × 2 = 8	5 × 2 = 10	2
rozšíření a uplatnění	3 × 3 = 9	5 × 3 = 15	3
Celkové skóre:	**100**	**123**	

Podle výpočtu celkového skóre byl na základě srovnání pomocí matice určen jako úspěšnější framework Struts. JSF framework za ním však zaostává jen přibližně o 20%, což není nijak markantní rozdíl. Ztráta JSF frameworku je zejména v oblasti implementace MVC modelu (JSF je zaměřen převážně na prezentaci), poměrné obtížnosti na naučení a také v jeho praktickém uplatnění z hlediska pracovní poptávky. Struts naopak zaostává v podpoře datového modelu (řešitelné přes JNDI) a podpory pro tvorbu prezentační vrstvy.

8 Závěr

Tato diplomová práce se zabývá webovými frameworky v jazyce Java. Popisuje problematiku webových frameworků obecně, nastiňuje možnosti srovnání a správného výběru frameworku pro konkrétní webovou aplikaci, věnuje se kategorizaci dle různých charakteristik a popisu některých vybraných frameworků a jejich srovnání.

Součástí práce je také analýza popisu požadavků, návrhu a implementace ukázkové aplikace.

Při popisu obou frameworků jsem vycházel z praktické znalosti vývoje ukázkových aplikacích v těchto frameworcích, jejichž implementace ve formě zdrojových kódů je součásti diplomové práce (přílohy 4 až 6). Většina komentovaných příkladů a ukázek zdrojového kódu v této práci pochází právě z těchto ukázkových aplikací. Tento teoretický úvod do problematiky jednotlivých frameworků dále podrobněji popisují programátorské tutoriály provázející implementacemi demo aplikací krok za krokem (přílohy 1 až 3).

Jako ukázková aplikace zvolených frameworků byla navržena typická webová aplikace skýtající rozličnou funkcionalitu včetně databázové struktury používající všechny základní typy vztahů mezi entitami. Tématem této aplikace je elektronický obchod, což je téma natolik zažité a známé, že pochopení idey aplikace a její implementace ve zvoleném frameworku je velmi názorné. Ukázková aplikace implementuje pouze základní funkčnost pro demonstraci možností a schopností zvoleného frameworku. Vytvoření komplexní aplikace by nepřineslo do problematiky nic nového a spíše by dělalo ukázku nepřehlednější, což je zde nežádoucí.

Na základě zkušeností s vývojem těchto aplikací bylo provedeno srovnání obou zvolených frameworků. Z tohoto celkového zhodnocení vyšel lépe framework Struts. Tento výsledek odpovídá i mému intuitivnímu odhadu dle zkušeností s vývojem demo aplikace. Vítězství Struts však nebylo zcela jednoznačné (123:100), z čehož vyplývá, že i JSF framework má své specifické výhody, které jej činí vhodnějším pro různé druhy aplikací. Ke srovnání bylo využito standardních metod popsaných obecně v této práci, konkrétně srovnání prostřednictvím tabulky.

Literatura

[1] The Java EE 5 Tutorial : For Sun Java System Application Server 9.1 [online]. September 2007 [cit. 2008-10-26]. Dostupný z WWW: <http://java.sun.com/javaee/5/docs/tutorial/doc>.

[2] BRANICKÝ, Marek. Java Servlets : predstavenie technológie. Interval.cz [online]. 2003 [cit. 2008-10-31]. Dostupný z WWW: <http://interval.cz/clanky/java-servlets-predstavenie-technologie>.

[3] EJB 3.0 Enterprise Beans [online]. [2008] [cit. 2008-10-23]. Dostupný z WWW: <http://www.netbeans.org/kb/55/ejb30.html>.

[4] NASH, Michael. Java Frameworks and Components : Accelerate Your Web Application Development. [s.l.] : [s.n.], 2003. 477 s. ISBN 0521520592.

[5] MAHMOUD, Qusay. Developing Web Applications with JavaServer Faces [online]. 2004 [cit. 2008-10-25]. Dostupný z WWW: <http://java.sun.com/developer/technicalArticles/GUI/JavaServerFaces>.

[6] Struts Tutorials : Jakarta Struts Tutorial [online]. c2008 [cit. 2008-10-28]. Dostupný z WWW: <http://www.roseindia.net/struts>.

[7] BROWN, Donald, CHAD, Davis. Struts 2 in Action. [s.l.] : [s.n.], 2001. 450 s. ISBN 193398807X, 97819.

Seznam příloh

Příloha 1. Tutoriál k JSF frameworku

Příloha 2. Tutoriál k Struts 1.x frameworku

Příloha 3. Tutoriál k Struts 2 frameworku

Příloha 4. Zdrojové kódy ukázkové aplikace v JSF frameworku (projekt v IDE Netbeans 6.0.1)

Příloha 5. Zdrojové kódy ukázkové aplikace ve Struts 1.x frameworku (projekt v IDE Netbeans 6.0.1)

Příloha 6. Zdrojové kódy ukázkové aplikace ve Struts 2 frameworku (projekt v IDE Netbeans 6.0.1)