Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Programování webových aplikací v jazyku Java

Roman Šíma

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Roman Šíma

Informatika

Název práce

Programování webových aplikací v jazyku Java

Název anglicky

Web application programming in Java

Cíle práce

Bakalářská práce je tematicky zaměřena na problematiku vývoje aplikací v jazyce Java. Hlavním cílem práce je charakterizovat základní aspekty, které jazyk Java programátorům nabízí.
Dílčí cíle bakalářské práce jsou:
- analyzovat obecné požadavky na samotný programovací jazyk,
- charakterizovat různé problémy, které s sebou vývoj aplikací přináší a ukázat možná řešení v praxi,
- navrhnout nejschůdnější možné řešení určitých problémů při návrhu a vývoji aplikací,
- zpracovat zkušenosti s vývojem vlastních aplikací.

Metodika

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačních zdrojů. Dále také na vlastních zkušenostech autora BP získaných při tvorbě aplikací.

Vlastní řešení je realizováno formou návrhů různých metod, kterými lze řešit různé situace a problémy.

Na základě syntézy teoretických poznatků, praktických zkušeností a výsledků vlastního řešení budou formulovány závěry bakalářské práce.

Doporučený rozsah práce

30-40

Klíčová slova

Java aplikace, příkazy, vzory, modely, situace, možnosti, hlediska, problémy, řešení, plánování, návrh

---

Doporučené zdroje informací

BURD, B. JSP: JavaServer Pages Podrobný průvodce. Praha: Computer Press. 2003. ISBN 80-7226-804-X

ECKEL, B. Myslíme v jazyku Java: knihovna programátora. Praha: Grada. 2001. ISBN 80-247-9010-6.

ECKEL, B. Myslíme v jazyku Java: knihovna zkušeného programátora. Praha: Grada. 2001. ISBN 80-247-0027-1.

HALL, M. JAVA servlety a stránky JSP. Praha: Neocortex. 2001. ISBN 80-86330-06-0

HOGAN, B P. HTML5 a CSS3: výukový kurz webového vývojáře. Brno: Computer Press. 2011. ISBN 978-80-251-3576-1.

Předběžný termín obhajoby

2015/16 LS – PEF

Vedoucí práce

Ing. Čestmír Halbich, CSc.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 28. 10. 2015

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 10. 11. 2015

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 02. 2016

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Programování webových aplikací v jazyku Java" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 14. 3. 2016 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Čestmíru Halbichovi, CSc. za přínosné rady a konzultace během vedení této bakalářské práce. Další, komu bych chtěl poděkovat je rodina za trpělivost a podporu.

Programování webových aplikací v jazyku Java

Souhrn

Tato bakalářská práce popisuje základní technologie platformy Java EE, která slouží k programování webových a podnikových aplikací. V teoretické části je uveden stručný přehled platformy, porovnání s konkurenty nebo HTTP protokol. Dále je pak hlouběji probrána technologie servletů, sloužící jako kontrolér v třívrstvé architektuře. Věnoval jsem se i dynamicky generovaným stránkám Java Server Pages. Nakonec teoretické části jsem popsal základní aspekty technologie Java Server Faces. Všechny probrané témata jsem doplnil o názorné příklady.

V praktické části se věnuji vytvoření aplikace, která slouží jako evidence skladu. Popisuji v ní jaké má aplikace funkce a grafické rozhraní.

Klíčová slova: Java aplikace, příkazy, vzory, modely, situace, možnosti, hlediska, problémy, řešení, plánování, návrh

Web application programming in Java

Summary

This bachelor's thesis describes a basic technologies of Java EE, which is, used for programming web and enterprise applications. The theoretical part provides a brief overview of the platform, compared to competitors or HTTP protocol. Further, it is deeply discussed servlet technology, serving as a controller in a three-tier architecture. I have devoted to dynamically generated pages called Java Server Pages. Finally, the theoretical part, I described the basic aspects of Java Server Faces. All discussed topics I added illustrative examples.

In the practical part is devoted to developing an application that serves as an evidence of repository. I describe what it has application functions and graphical interface.

Keywords: Java application, commands, patterns, models, situation, options, prespective, problems, solutions, planning, suggestion

Obsah

Obsah **3**

1 Úvod **6**

2 Cíl práce a metodika **7**

2.1 Cíl práce 7

2.2 Metodika 7

3 Platforma EE **8**

3.1 Konkurenti 8

3.1.1 PHP 8

3.1.2 ASP.NET 8

3.1.3 Ruby on Rails 9

3.2 MVC architektura 9

3.3 Potřebný software 10

3.4 Souborová struktura java aplikace v Eclipse 10

4 HTTP protokol **11**

4.1 Formy zaslání požadavků 12

4.2 MIME typy 12

4.3 Stavové kódy 12

4.4 Záhlaví 13

5 Servlety **14**

5.1 Zpracování požadavku a vytvoření odpovědi 14

5.1.1 Získaní záhlaví z HTTP 14

5.1.2 Získání formulářových dat 15

5.1.3 Vytvoření odpovědi 16

5.1.4 Nastavení Stavového kódu 16

5.2 Relace (Session) 17

5.2.1 Cookies 17

5.2.2 Přepis URL 18

5.2.3 Práce s relacemi 18

5.3 Servlet Kontext 19

5.3.1 Rozsah proměnných 19

5.4 Nastavení 20

5.4.1 Web.xml 20

5.4.2 Web-fragment 21

5.4.3 Anotace 22

5.4.4 Posloupnost 22

5.5 Filtry ... 23
5.6 Posluchače ... 24
5.7 Přesměrování ... 25

6 JavaBean ... 25

7 Expression language (EL) ... 26

8 JSP ... 27
8.1 Syntaxe ... 28
8.1.1 Implicitní objekty ... 28
8.1.2 Výrazy ... 29
8.1.3 Deklarace ... 29
8.1.4 Skriptlety ... 30
8.1.5 Direktiva ... 30
8.1.6 Standartní instrukce ... 32
8.2 Značky XML ... 33
8.2.1 Standardní knihovna značek (JSTL) ... 34
8.2.2 Vytváření vlastních knihoven značek ... 34

9 Java Server Faces ... 37
9.1 Základní knihovny značek ... 37
9.2 Managed Bean ... 39
9.3 Konvertory ... 40
9.4 Zpracování událostí ... 41
9.5 Validátory ... 43
9.5.1 Tvorba vlastních validátorů ... 44
9.6 Vkládání závislostí ... 45
9.7 Životní cyklus JavaServer Faces požadavku ... 45
9.7.1 Restore View ... 45
9.7.2 Apply Request Value ... 46
9.7.3 Process Validation ... 46
9.7.4 Update Model Values ... 46
9.7.5 Invoke Application ... 46
9.7.6 Render Response ... 46
9.8 Konfigurace ... 47
9.9 Navigace ... 47

10 Vlastní práce ... 48
10.1 Aplikace sklad ... 48
10.2 Návrh aplikace ... 48
10.3 Návrh databáze ... 49
10.4 Vytvoření aplikace ... 50

10.4.1 Listeners 50
10.4.2 Validators 50
10.4.3 Database 51
10.4.4 Filtry 52
10.4.5 Grafické rozhraní 53
10.5 Funkce aplikace 53
10.6 Nasazení aplikace 56

11 Závěr 57

12 Seznam použitých zdrojů 58
12.1 Seznam zdrojů 58
12.2 Seznam tabulek 59
12.3 Seznam obrázků 59
12.4 Seznam příloh 59

1 Úvod

Platforma Java EE (Enterprise Edition) je používána pro vývoj podnikových nebo webových aplikací. Vychází z Java SE (Standard Edition), kterou doplňuje velikým množstvím knihoven. Výhodou Javy EE je, že může být nasazena na jakémkoliv operačním systému. Pomocí této platformy vyvíjíme program, který běží na serveru a generuje nám odpovědi podle přijatých informací. Odpovědí většinou bývá webová stránka.

Takto dynamicky generované stránky mají několik výhod oproti statickým. Například si představte e-shop, kde je nabízeno velké množství zboží. Pomocí statických stránek by muselo existovat tolik stránek, kolik je zboží. V dynamicky generovaných stránkách stačí pouze jedna, která načítá informace z databáze podle požadavku uživatele. Další výhodou je uživatelsky přívětivé prostředí, kdy si uživatel může upravit prostředí podle sebe a při příští návštěvě je mu vygenerována stránka dle jeho nastavení. Nevýhodou je o něco těžší implementace.

V dnešní době jsou webové aplikace vyvíjeny čím dál častěji. Takovéto aplikace jsou nezávislé na operačním systému. Uživateli stačí pouze webový prohlížeč a může pracovat z jakéhokoliv operačního systému. Dalším důvodem je mobilita. K aplikaci lze přistupovat odkudkoliv, kde máme připojení k internetu, ať už z domova nebo vlaku. Nemalou výhodou je to i pro administrátory ve velkých firmách. Pokud bude v aplikaci nalezena chyba a bude nutná aktualizace, nemusí se provádět na několika počítačích, ale pouze na jednom serveru.

2 Cíl práce a metodika

2.1 Cíl práce

Bakalářská práce je tematicky zaměřena na problematiku vývoje aplikací v jazyce Java. Hlavním cílem práce je charakterizovat základní aspekty, které jazyk Java programátorům nabízí a seznámení s platformou Java EE pro vývoj webových a podnikových aplikací.

Dílčím cílem je vytvořit ukázkovou aplikaci, která bude stavět na získaných zkušenostech z teoretické části.

2.2 Metodika

Metodika řešené problematiky bakalářské práce je založena na studiu a analýze odborných informačních zdrojů. Dále také na vlastních zkušenostech při tvorbě aplikací. Na základě syntézy teoretických poznatků, praktických zkušeností a výsledků vlastního řešení budou formulovány závěry bakalářské práce.

3 Platforma EE

Platforma Java EE obsahuje mnoho knihoven pro práci s různými technologiemi, které se postupem času do platformy zařadily. Jsou to například:

Technologie	Zkratka	Význam
Servlet	-	Třídy obsluhující požadavky od uživatelů
Java Server Pages	JSP	Dynamicky generované stránky
Java Server Faces	JSF	MVC Framework
Enterprise Java Bean	EJB	Práce s komponentami, zajišťují logiku aplikace
Java Presistence API	JPA	Ukládání objektů do databáze
Java Transaction API	JTA	Práce s transakcemi
Java Messaging Services	JMS	Posílání zpráv mezi aplikacemi
SOAP Web Services	JAX-WS	Tvorba SOAP webových služeb
REST Web Services	JAX-RS	Tvorba REST webových služeb
XML	-	Pro práci s XML soubory

Tabulka 1 – seznam vybraných technologií [Autor] podle [1]

3.1 Konkurenti

Samozřejmě že Java není jedinou technologii, kterou lze vyvíjet webové aplikace. To, kterou technologii použít, zvažujeme u každé aplikace individuálně, například podle velikosti, bezpečnosti, oblíbenosti, škálovatelnosti nebo třeba ceny.

3.1.1 PHP

Asi nejznámější technologii pro tvorbu webových aplikací je PHP. Je to skriptovací programovací jazyk, používán pro menší aplikace jako jsou e-shopy nebo blogy. Spolu s operačním systémem Linux, webovým serverem Apache a databázovým serverem MySql je nejpoužívanější kombinací, často nazývanou LAMP (Linux, Apache, MySql, PHP). [2]

3.1.2 ASP.NET

Je hlavním konkurentem Javy ve vývoji podnikových aplikací. Díky .NET prostředí

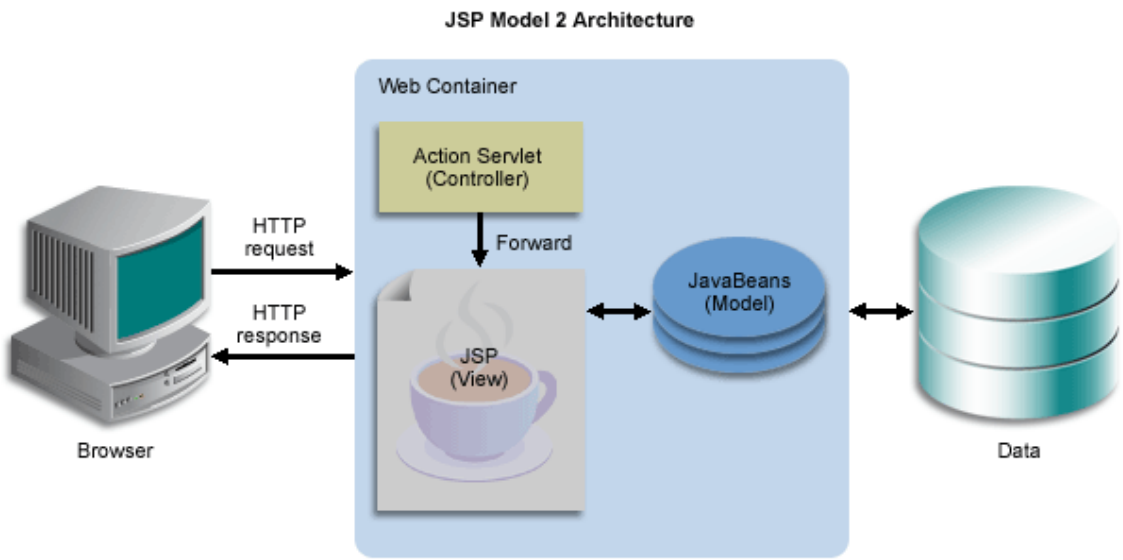
lze programovat pomocí různých programovacích jazyků, většinou se ale používá C#. Nevýhodou ASP.NET aplikací je, že pro jejich nasazení na server jsme vázáni produkty Microsoft. Nasazují se na operační systém Windows Server a aplikační server IIS (Internet Information Services). [3]

3.1.3 Ruby on Rails

Je to webový framework založený na jazyce Ruby. Používá více vrstvé architektury MVC. Je to velice jednoduchý a produktivní framework, kterému na rozdíl od ostatních programovací jazyků stačí napsat daleko méně kódu. Nevýhodou může být jeho umíněnost, kde tlačí vývojáře určitým směrem. [4]

3.2 MVC architektura

Je to softwarová architektura, která rozděluje vývoj aplikace do 3 oddělených vrstev, jimiž jsou Model, View, Controller (MVC). Tímto přístupem se řídí i Java. Vrstva view se stará pouze o to jak bude uživateli zobrazeno dané uživatelské rozhraní. V javě je view typicky JSP nebo JSF stránky. Controller obsluhuje požadavky od uživatele, ukládá je do modelu a poté předá řízení view vrstvě. O tuto funkci se stará servlet. Poslední vrstva je model, která zajišťuje logiku aplikace, jedná se například o javaBean a EJB. [1]



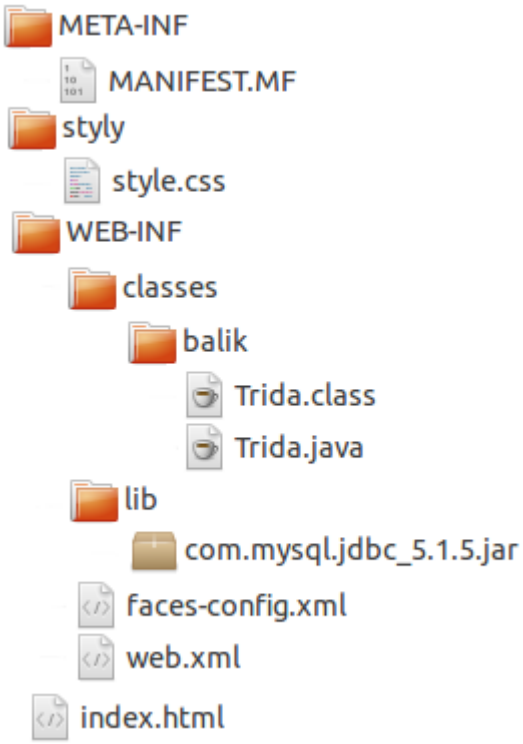
Obrázek 1 - MVC architektura v Java EE [5]

3.3 Potřebný software

Abychom mohly vyvíjet aplikace, potřebujeme nějaké vývojové prostředí. Mezi nejznámější patří NetBeans, Eclipse nebo IntelliJ IDEA. Já jsem se rozhodl používat pro své příklady Eclipse díky jeho přehlednosti. Jako další budeme potřebovat nějaký aplikační server, na kterém budou naše aplikace spuštěny. Existuje mnoho aplikačních serverů ať už komerční nebo open source. Mezi nejznámější open source servery patří GlassFish, Jboss nebo Apache Tomcat. Komerční potom WebLogic (Oracle) nebo WebShare (IBM). Já jsem pro testování příkladů zvolil Tomcat, kvůli jeho rozšířenosti a jednoduchosti.

3.4 Souborová struktura java aplikace v Eclipse

Námi vytvořená aplikace je většinou nahrávána na server jako webový archiv, který je potom na serveru rozbalen. Takovýto archiv má koncovku .war. Ve vývojovém prostředí eclipse jsou pak části aplikace umístěny ve složce WebContent. Obsahuje složky META-INF a WEB-INF, kde jsou uloženy konfigurační soubory jako například web.xml nebo faces-config.xml. Dále pak je v WEB-INF složka classes s třídami a složka lib s vloženými knihovnami třetích stran, které jsou v aplikaci použité. Ve složce WebContent jsou uloženy i webové stránky nebo vlastní složky.



Obrázek 2 - souborová struktura v Eclipse [Autor]

4 HTTP protokol

HTTP (HyperText Transfer Protocol) je internetový protokol, který slouží ke komunikaci klienta se serverem. Funguje formou zpráv (požadavek / odpověď). Protokol obsahuje pravidla, která zajistí, že server správně přečte zprávu, kterou sestavil klient a naopak klient bude rozumět odpovědi od serveru. Požadavek obsahuje metodu zaslání, URI identifikátor a seznam záhlaví klienta. [6] [9]

GET /cs/ HTTP/1.1
Host: www.czu.cz
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: cs-CZ,cs;q=0.8

Každá odpověď serveru obsahuje verzi protokolu, stavový kód, seznam záhlaví specifikované serverem a obsah zprávy. [6]

HTTP/1.1 200 OK

Date: Fri, 06 Nov 2015 12:02:19 GMT
Server: Apache/2.4.12 (Win32) OpenSSL/1.0.1m PHP/5.2.17
X-Powered-By: PHP/5.2.17
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html

<!DOCTYPE html>
...

4.1 Formy zaslání požadavků

Protokol HTTP 1.1 má osm metod pro komunikaci se serverem, ale nejpoužívanější je *GET* a *POST*. V požadavku typu *GET* se formulářová data odesílaná na server ke zpracování připojují za URL adresu. Pokud bychom ale chtěli posílat například přihlašovací údaje, není požadavek typu *GET* příliš vhodný. Proto máme lepší variantu a to požadavek typu *POST* kde se data nepřipojují za URL adresu, ale na vlastní řádek v požadavku. Mezi další metody patří *PUT, HEAD, OPTIONS, TRACE, DELETE* a *CONNECT*. [7] [9]

4.2 MIME typy

Ne všechny odpovědi serveru musí být webové stránky. Server může zaslat prostý text, obrázek nebo tabulku v Excelu. Aby mohl klient data správně interpretovat, musí vědět, jakého typu jsou. Pro určení jiného obsahu než webové stránky přepíšeme hodnotu záhlaví *Content-Type* příslušným MIME typem. MIME (Multipurpose Internet Mail Extensions) identifikuje formát souboru používaný na internetu. Skládá se ze dvou částí *typ/podtyp*. Mezi základní typy patří *application, text, audio, video* nebo *image*. Například MIME formát pro webovou stránku je *text/html*. [6]

4.3 Stavové kódy

Stavový kód je reprezentovaný 3 čísly, z nich první specifikuje určitý status odpovědi. Rozlišujeme 5 druhů statusů. Každý kód je většinou následován slovem popisující kód (*OK, NOT_FOUND*, …). Kódy v rozsahu:

Kód	Význam
100 až 199	Informativní
200 až 299	Označují, že požadavek byl úspěšně proveden
300 až 399	Vyžaduje od klienta další činnost
400 až 499	Chyba klienta
500 až 599	Chyba serveru

Tabulka 1 - seznam stavových kódů [Autor] podle [7]

4.4 Záhlaví

Jsou to informace poskytnuté klientem nebo serverem. Každé záhlaví má svůj vlastní význam. Klient většinou specifikuje, které MIME typy podporuje, jaký jazyk a prohlížeč používá a na jaký server odesílá požadavek. Naopak server nám určuje, jaký formát dat bude mít odpověď, jakou délku má a datum kdy byla odeslána. [6]

Záhlaví	Zpráva	Význam
Accept	Požadavek	Podporované MIME typy klientem
Accept-Encoding	Požadavek	Podporované kódování klientem
Accept-Language	Požadavek	Podporované jazyky klientem
Allow	Odpověď	Podporované metody serverem
Connection	Oba	Klient: druh připojení Server: pokud nepodporuje trvalé, posílá close
Cookie	Požadavek	Odesílá cookie soubor danému serveru
Content-Encoding	Odpověď	Sděluje klientovi použité kódování
Content-Length	Oba	Sděluje délku těla zprávy
Content-Type	Oba	Sděluje použitý MIME typ
Date	Oba	Čas odeslání zprávy
Host	Požadavek	Doménové jméno serveru a port
If-Modified-Since	Požadavek	Ptá se, zda byl dokument změněn
Last-Modified	Odpověď	Poslední datum změny požadovaného objektu
Server	Odpověď	Informace o softwaru, který přijímá požadavek
Set-Cookie	Odpověď	Nastavuje soubor cookie
User-Agent	Požadavek	Obsahuje informace o klientovi

Tabulka 2 - seznam vybraných záhlaví [Autor] podle [7]

5 Servlety

Jsou to obyčejné Java třídy, které obsluhují příchozí požadavky od uživatelů. Implementují rozhraní *javax.servlet*, ale většinou jsou servlety využívány pro komunikaci přes HTTP protokol a místo *javax.servlet* implementují jejího potomka *javax.servlet.http.HttpServlet.* Servlety v MVC architektuře zastávají funkci Kontroléru. Příchozí požadavek je v servletu zachycen, jsou zpracovány příchozí data ze záhlaví a formulářů, a nakonec je předáno řízení dál. [6]

Když je aplikace spuštěna a instance servletu je načtena do paměti, je volána její metoda *init,* ve které se mohou provést určité operace pro načtení zdrojů, navázaní spojení s databází apod. Jakmile přijde požadavek je spuštěna metoda *service*, která dále volá metodu *doGet, doPost* nebo jinou (podle typu požadavku), ve které se provádí většina operací. Před ukončením aplikace nebo odstraněním instance z paměti je zavolána metoda *destroy*, ve které se zavírají navázaná spojení s databází anebo zavření zdrojů. [6]

5.1 Zpracování požadavku a vytvoření odpovědi

Pokaždé když klient zašle požadavek, server spustí nové vlákno daného servletu, vyvolá metodu *service* a ta příslušnou metodu *doXXX* podle typu požadavku. Metodě je serverem předán jako argument objekt rozhraní *javax.servlet.http.HttpServletRequest* (požadavek) a *javax.servlet.http.HttpServletResponse* (odpověď). Signatura metod vypadá takto:

```
protected void doXXX(HttpServletRequest request, HttpServletResponse response)
```

Objekt *request* reprezentuje všechny záhlaví a formulářová data poskytnuté klientem, ale i metody pro získání, mazání nebo ukládání proměnných. U objektu *response* nastavujeme záhlaví, popřípadě stavový kód nebo i přesměrování na jinou stránku. [6]

5.1.1 Získaní záhlaví z HTTP

Jak už bylo řečeno, záhlaví je získáno z objektu *request*. Předáním názvu záhlaví metodě *getHeader* získáme *String* hodnotu záhlaví nebo *null* pokud není specifikováno. Jestliže má být hodnota záhlaví datum, můžeme použít metodu *getDateHeader*, který vrací long reprezentující čas, který uběhl od 1. ledna 1970 nebo -1 když není určeno. Pokud chceme přečíst záhlaví, které vrací číslo, použijeme metodu *getIntHeader* vracející *int* proměnou nebo

-1. Nevíme-li, jaká záhlaví jsou specifikována klientem, máme metodu *getHeaderNames* vracejí *Enumeration*, který obsahuje výčet názvu záhlaví poskytnutých klientem. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String kodovani = request.getHeader("Accept-Charset");
        int delka = request.getIntHeader("Content-Length");
        long odeslano = request.getDateHeader("Date");
}
```

5.1.2 Získání formulářových dat

Formulář je reprezentován pomocí HTML značek. Pokud je odesílán na server metodou *GET*, jsou tyto data přiřazena za URL adresu a ‚?'. Následuje jeden nebo více parametrů z formuláře ve formátu *název=hodnota*. Jednotlivé parametry jsou odděleny ‚&' a na jeden název parametru lze navázat více hodnot. URL adresa by mohla vypadat například nějak takto:

http://www.server.cz/servlet?jmeno=Roman&prijmeni=Sima

Většinou se používá metoda *POST*, kde jsou parametry zaslány na samostatné řádce.

V servletu jsou formulářová data získána pomocí objektu *request* a její metodě *getParameter*. Metodě předáme jako argument název parametru, který chceme získat. Pokud byl zaslán, metoda nám vrátí *String* hodnotu parametru, v opačném případě *null*. Jestliže je na parametr vázáno více hodnot, metoda vrátí pouze první hodnotu. Chceme-li získat všechny hodnoty, voláme *getParameterValues*. Metodě předáme jako argument název parametru a ona nám vrátí pole *String* hodnot, nebo *null* pokud parametr neexistuje. V některých případech neznáme názvy obdržených parametrů. Proto má *request* metodu *getParameterNames*, která vrací výčet obsahující názvy parametrů předaných uživatelem. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String jmeno = request.getParameter("jmeno");
        String prijmeni = request.getParameter("prijmeni");
}
```

5.1.3 Vytvoření odpovědi

Když máme zpracované všechny příchozí informace, můžeme vytvořit odpověď pro klienta. Odpověď je vázána na objekt *response*. Záhlaví jsou nastavena pomocí metod *setHeader, setDateHeader* a *setIntHeader*. Jako první argument předáme metodám název záhlaví. Druhý argument je hodnota záhlaví. [6]

Servlet není vhodný pro vytváření obsahu odpovědi, nicméně tu možnost má. Ovšem před zasíláním jakéhokoliv obsahu do výstupního proudu se doporučuje provést všechny nezbytné nastavení odpovědi, protože HTTP zpráva obsahuje nejprve verzi protokolu a stavoví kód potom jednotlivé záhlaví a pak až obsah odpovědi. To jak mají být data reprezentována, nám právě určuje jediné povinné záhlaví *Content-Type*. Jelikož je záhlaví nastavováno při vždy byla přidána metoda *setContentType*. Podle formy odpovědi voláme buďto *getWriter* vracející *PrintWriter*, nebo *getOutputStream*, který nám vrací *ServletOutputStream*. Většinou jsou do PrintWriteru vkládány HTML značky jako *String* řetězce. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter vystup = response.getWriter();
        vystup.println("<html>");
        vystup.println("<head><title>odpoved</title></he<p>ad>");
        vystup.println("<body><b>Jmeno : </b>" +
        request.getParameter("jmeno") + "<body>");
        vystup.println("</html>");
}
```

5.1.4 Nastavení Stavového kódu

Jako odpověď můžeme nastavit i HTTP status, kterým odpovíme. Objekt *response* má metodu *setStatus*, které předáváme konstantu z rozhraní *HttpServletResponse*, představující stavoví kód. Pokud by se mělo jednat o nějaký chybový status, je vhodnější použít přetěžovanou metodu *sendError*. Kdybychom měli v aplikaci určenou nějakou chybovou stránku, metoda nás na tuto stránku přesměruje zatímco *setStatus* ne. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String jmeno = request.getParameter("jmeno");
        if(jmeno.equals("Roman")){
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<html><body>Vitej Romane</body></html>");
        }else{
                response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        }
}
```

5.2 Relace (Session)

Protokol HTTP sám nijak nerozlišuje požadavky od jednotlivých uživatelů, ale někdy je potřeba identifikovat uživatele, který už na stránkách byl, nějak si je nastavil nebo přidal zboží do košíku a podobně. Proto existuje vestavěná podpora session. Ta je dosažena buďto ukládáním záznamů cookies nebo přepisem URL. [6]

5.2.1 Cookies

Jsou to malé soubory vytvořené aplikací a předány v záhlaví odpovědi. Následně jsou automaticky ukládány do počítače. Při další návštěvě jsou opět předány v záhlaví požadavku. Každé cookies je vázáno pouze na server kde bylo vytvořeno. V Javě jsou reprezentovány třídou *Cookie*, která umožňuje práci s nimi. Například nastavení životnosti souborů, přiřazení nové hodnoty cookie a podobně. Získáme ho přes objekt *request.* Jde vlastně jenom o to přečíst záhlaví s názvem *Cookie*. Je jedno z nejvíce používaných a proto má pro ukládání nebo získávání vlastní metody. Metoda *getCookies* vrací pole *Cookie* objektů nebo *null* pokud požadavek žádné neobsahuje. Naopak pro uložení použijeme objekt *response* a jeho metodu *addCookie*, která bere jako parametr objekt *Cookie*. Samotné cookies je vytvořeno v konstruktoru třídy, který bere dva *String* parametry první je název a druhý hodnota. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String nastaveni;
        Cookie[] vsechny = request.getCookies();
        for(Cookie pom : vsechny){
                if(pom.getName().equals("nastaveni")){
                        nastaveni = pom.getValue();
                }else{
                        Cookie novy = new Cookie("nastaveni", "vychozi");
                        nastaveni = "vychozi";
                        response.addCookie(novy);
                }
        }
}
```

5.2.2 Přepis URL

Ačkoliv jsou cookies účinný nástroj pro identifikaci uživatele, některé internetové prohlížeče mohou mít uživatelem zakázáno ukládání těchto souborů. Proto tu máme další způsob jak toho dosáhnout. Přepis URL funguje tak, že se za URL adresu přidají údaje, které identifikují relaci a server přiřadí data, jenž jsou k této relaci uložena. Nevýhodou je, že pokud uživatel zadá URL adresu znovu bez přidaných údajů na konci nebo bude přistupovat na stránku z nějakého odkazu, nebude uživatel spojen s žádnou relací. [6]

5.2.3 Práce s relacemi

To jaký způsob máme použít, tím se nemusíme vůbec zabývat, protože Java nabízí řešení a tím je API pro práci s relacemi. Většinou server používá cookies, ale když jsou zakázána, přejde automaticky na přepis URL. Vše se odehrává v pozadí, my pouze pracujeme s *HttpSession* objektem, který reprezentuje relaci, a nestaráme se o detaily. Ten získáme z *request* a jeho metody *getSession*, která bere jako parametr *boolean* hodnotu. Pokud je předáno metodě *true* a relace s daným uživatelem dosud neexistuje, vytvoří se nová. Jestliže je předáno *false* a relace neexistuje, vrací *null*. Když máme objekt relace, můžeme do ni ukládat proměnné. Pomocí setAttribute do ní vložíme jakýkoliv objekt. Metoda bere dva parametry. První je *String*, který reprezentuje název a druhy je *Objekt* reprezentující hodnotu. Naopak pokud už máme v relaci nějakou proměnnou a chceme jí přečíst, voláme *getAttribute* a jako parametr předáme název. Tato metoda vrací *Objekt* hodnotu, která musí být přetypována na danou třídu. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String nastaveni;
        HttpSession relace = request.getSession(true);
        if(relace.isNew()){
                relace.setAttribute("nastaveni", "vychozi");
                nastaveni = "vychozi";
        }else{
                nastaveni = (String)relace.getAttribute("nastaveni");
        }
}
```

5.3 Servlet Kontext

Webový server může obsahovat více webových aplikací a každá tato aplikace má svůj vlastní servlet kontext. Je to prostor, který reprezentuje celou aplikaci. Kontextu lze nastavit inicializační proměnné. Tyto proměnné se nastavují v souboru web.xml a výhodou je, že při jejich změně nemusíme nijak zasahovat do kódu. Proměnné jsou dostupné všemi servlety v aplikaci. Pro získání kontextu zavoláme *getServletContext* na *request*. Metoda nám vrací objekt *ServletContext*. Pomocí objektu lze číst nebo zapisovat proměnné, programově ukládat servlety, filtry a posluchače, nebo zjistit typ a verzi servletového kontejneru.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        ServletContext ser = request.getServletContext();
        ServletRegistration.Dynamic reg = ser.addServlet("nazev", new TridaServletu());
        reg.addMapping("/servlet");
}
```

5.3.1 Rozsah proměnných

Máme 4 rozsahy proměnných. První je rozsah pro celou aplikaci. Ukládáme do kontextu servletu. Druhý je pro všechny volání jednoho uživatele. Proměnnou uložíme do *HttpSession*. Třetí je po dobu trvání jednoho požadavku. Nastavujme pomocí *request.setAttribute* a poslední rozsah je definován v JSP a platí pouze pro jednu JSP stránku. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        HttpSession session = request.getSession();
        request.getServletContext().setAttribute("kontextPromena", "hodnota");
        session.setAttribute("relacePromena", new String("hodnota"));
        request.setAttribute("pozadavekPromena", new Object());
        //posledni rozsah je pouze v JSP stránkách
}
```

5.4 Nastavení

Webová aplikace obsahuje konfigurační soubory nebo anotace, které říkají servletovému kontejneru např. jaká má být uvítací nebo chybová stránka, které proměnné mají být uloženy v kontextu nebo jaké servlety mají být načteny.

5.4.1 Web.xml

Je to konfigurační soubor celé aplikace, který se nachází ve /WEB-INF/web.xml. Je napsán v XML jazyku pro lepší čitelnost. Na prvním řádku je deklarována verze a kódování souboru. Následuje kořenová značka <*web-app*>, ve které se provádí veškeré nastavení. Jako atributy u značky nastavujeme jmenný prostor. Mezi nejdůležitější značky patří <*servlet*>. Uvnitř spojujeme název servletu s danou třídou. Když máme deklarovaný servlet tak ho můžeme namapovat na určitou URL adresu. Pomocí značky <*servlet-mapping*> můžeme spojit název servletu s určitým URL vzorem, takže pokud uživatel zadá adresu, která bude shodná se vzorem, tak bude požadavek předán danému servletu. Vzor může být přímo nějaký řetězec anebo regulární výraz. To samé platí pro filtry a jeho značky <*filter*> a <*filter-mapping*>. Jedinou výjimkou je, že můžeme filtr navázat na jiný servlet. Takže pokud je na servlet navázaný filtr a uživatel zadá URL adresu na kterou je namapovaný servlet, nejdříve se provede filtr a ten pak předá požadavek dál servletu. Mezi často používané značky patří <*error-page*> a <*welcome-file-list*>. První značka nám deklaruje chybovou stránku, kam jsou zasílány chybové kódy nebo výjimky, které v aplikaci nastanou. Druhá značka nám určuje uvítací stránku. Můžeme také např. nastavovat zabezpečení, popis aplikace nebo různé posluchače událostí. [6][8]

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>nazevApp</display-name>
  <welcome-file-list>
        <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  <error-page>
        <error-code>401</error-code>
        <location>/chybovaStranka.jsp</location>
  </error-page>
  <servlet>
        <servlet-name>nazevServletu</servlet-name>
        <servlet-class>TridaServletu</servlet-class>
  </servlet>
  <servlet-mapping>
        <servlet-name>nazevServletu</servlet-name>
        <url-pattern>/servlet</url-pattern>
  </servlet-mapping>
</web-app>
```

5.4.2 Web-fragment

Web-fragment je část nebo celý konfigurační soubor. Je obsažen spolu s třídami v JAR archivu knihovny nebo frameworku, který je do aplikace přidán. V archivu je umístěn ve složce /META-INF/web-fragment.xml. Aby se nemusel přepisovat soubor web.xml pokaždé, když je do aplikace přidán nějaký framework, pro který je potřeba definovat a namapovat nějaký servlet a podobně, je přibalen do archivu vlastní konfigurační soubor web-fragment.xml. Tento soubor rozšiřuje původní web.xml. Může obsahovat skoro všechny značky jako web.xml, ale jeho kořenová značka musí být <*web-fragment*> namísto <*web-app*>. Uvnitř lze deklarovat <*servlet*>, <*filter*> a podobné úplně stejně jako v web.xml. [8]

```
<?xml version="1.0" encoding="UTF-8"?>
<web-fragment id="WebFragment_ID" version="3.0" 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-
fragment_3_0.xsd">
        <display-name>nazevFragmentu</display-name>
        <filter>
                <filter-name>nazevFiltru</filter-name>
                <filter-class>TridaFiltru</filter-class>
        </filter>
        <filter-mapping>
                <filter-name>nazevFiltru</filter-name>
                <url-pattern>/*</url-pattern>
                <servlet-name>NazevServletu</servlet-name>
        </filter-mapping>
 </web-fragment>
```

5.4.3 Anotace

Anotace jsou třetí způsob konfigurace. Byli přidány do servletů ve verzi 3.0 a nachází se v balíku javax.servlet.annotation. Výhodou je, že nemusíme psát zdlouhavě konfigurační soubor, ale pouze přidáme anotaci přímo k servletu nebo filtru. Nevýhoda spočívá v tom, že pokud chceme změnit nějak konfiguraci, musíme danou třídu znova zkompilovat. *@WebServlet* odpovídá značkám <*servlet*> a <*servlet-mapping*> v souboru web.xml. V této anotaci můžeme nastavit název servletu, který poté namapujeme na nějaký URL vzor, nebo nastavit inicializační proměnné a podobně. Úplně stejně je na tom *@WebFilter*, který odpovídá značkám <*filter*> a <*filter-mapping*>. [8]

```
@WebServlet("/servlet")
public class TridaServletu extends HttpServlet {
        ...
}
```

5.4.4 Posloupnost

Když spouštíme webovou aplikaci, tak se pro konfiguraci nahlédne do souboru web.xml, poté se prohledávají JAR knihovny jestli nemají svoje vlastní konfigurační soubory pro vlastní třídy, a nakonec se prohledávají všechny třídy a hledají se anotace. Takovéto hledání je na spuštění aplikace poněkud časově náročné, natož když ladíme aplikaci a potřebujeme často restartovat aplikační server. Proto můžeme v souboru web.xml zapsat do značky <*web-app*> atribut *metadata-complete="true"* a žádné jiné konfigurace se hledat nebudou. Pokud chceme načíst i konfiguraci z JAR archivů, ale nechceme hledat anotace, atribut *metadata-complete="true"* zapíšeme až v souboru web-fragment.xml. Když dojde ke kolizi v konfiguraci, například web-fragment.xml bude chtít nastavit uvítací stránku a web.xml taky, použije se uvítací stránka z konfiguračního souboru web.xml. On má totiž přednost před web-fragmentem, ale ten má zas přednost před anotacemi. [8]

V některých případech musíme určit, které třídy mají být načteny dříve před ostatními. Například když máme více filtrů, musíme určit, v jakém pořadí budou vykonávány. V souboru web.xml na to máme párovou značku <*absolute-ordering*>, do níž se zapisují <*name*> značky, v kterých jsou názvy posluchačů nebo filtrů. Tak jak jdou za sebou, jsou i vykonávány. Mimo značky <*name*> můžeme vložit i nepárovou značku <*others/*>, která reprezentuje zbylé prvky. [8]

```
<web-app>
	<absolute-ordering>
		<name>Prvni</name>
		<name>Druhy</name>
		<others/>
		<name>Posledni</name>
	</absolute-ordering>
</web-app>
```

U souboru web-fragment.xml je pořadí udáváno relativně a to pomocí párové značky <*ordering*> a jejích vnitřních značek <*after*>, <*before*> nebo nepárové značky <*other/*>. [8]

```
<web-fragment>
	<name>Druhy</name>
	<ordering>
		<after>Prvni</after>
		<before><others/></before>
	</ordering>
 </web-fragment>
```

5.5 Filtry

Filtry jsou Java třídy, které implementují rozhraní javax.servlet.Filter a implementují metodu *doFilter*. Jsou ukládány spolu se servlety. Filtry se využívají pro zpracování požadavku nebo i nastavení odpovědi ještě než přijde požadavek k servletům. Slouží např. pro zpracování záhlaví a formulářových dat, logování požadavků nebo jako autentifikace uživatele při přihlašování. Nejsou vhodné pro odesílání odpovědi. Lze zařadit několik filtrů za sebe, kde každý bude mít jinou funkci. Takovému zařazení se říká zřetězení filtrů. Filtry a servlety jsou vždy vykonávány ve stejném vlákně. Stejně jako servlety musejí být i filtry uvedeny buďto v web.xml pomocí značek <*filter*> a <*filter-mapping*> a nebo pomocí anotací *@WebFilter* u dané třídy. Veškeré zpracování probíhá v metodě *doFilter*, která má parametry *ServletRequest* (požadavek), *ServletResponse* (odpověď) se kterými se pracuje stejně jako u metod *doXXX* v servletech. Navíc má parametr *FilterChain* pro zřetězení filtrů. [8]

```
@WebFilter("/*")
public class Filtr implements Filter {

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        String jmeno = request.getParameter("jmeno");
        String mobil = request.getParameter("mobil");
        String adresa = request.getParameter("adresa");
        Zaznam z = new Zaznam(jmeno, mobil, adresa);
        ulozitDoDB(z);
        chain.doFilter(request, response);
}
        ...
}
```

5.6 Posluchače

Posluchače umožňují zachytit událost a reagovat na ní. Jsou to Java třídy, které implementují dané rozhraní podle toho, jakou událost chceme zachytit. Daná třída musí být uvedena v konfiguračním souboru značkou *<listener>* a nebo pomocí anotací *@WebListener*. Události dělíme na 3 druhy podle toho, kde nastaly. Jsou to kontextové, požadavkové anebo události relací. [8]

Posluchače (rozhraní)	Metody posluchačů
ServletContextListener	contextInitialized
	contextDestroyed
ServletContextAttributeListener	attributeAdded
	attributeRemoved
	attributeReplaced
HttpSessionListener	sessionCreated
	sessionDestroyed
HttpSessionAttributeListener	attributeAdded
	attributeRemoved
	attributeReplaced
ServletRequestListener	requestInitialized
	requestDestroyed

Tabulka 3 - seznam vybraných posluchačů [Autor] podle [8]

5.7 Přesměrování

Jak už bylo řečeno servlety v MVC architektuře zastupují pouze funkci kontroléru. Data tu jsou zpracována a uložena. Potom kontrolér předá řízení někam jinam a jeho funkce končí. [6]

Pro předání řízení nejprve musíme získat objekt *RequestDispatcher*. Ten získáme tím, že voláme metodu *getRequestDispatcher* na objektu request anebo na objektu *servletContext*. Obě berou jako parametr *String* cestu k cílové stránce nebo servletu, kterému chceme předat řízení. Rozdíl mezi nimi je, že metoda na objektu *request* může brát i relativní cestu. Když máme RequestDispatcher, stačí zavolat metodu *forward* nebo *include*. Metoda *forward* předá plné řízení cílovému prvku, zatímco *include* začlení cílový prvek a po jeho skončení je řízení vráceno zpět. Obě metody předávají jako parametry požadavek a odpověď. [6]

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        RequestDispatcher r = request.getRequestDispatcher("/index.jsp");
        r.forward(request, response);
}
```

6 JavaBean

Takzvané softwarové komponenty. Jsou to Java třídy, které podléhají určité konvenci zápisu. Používají se pro práci s webovými aplikacemi, ale i k vytváření grafického uživatelského rozhraní. JavaBean komponenta je například tlačítko nebo textové pole, které ve vývojovém prostředí přetáhneme z menu do naší aplikace. Ve webových aplikacích se instance JavaBean komponent ukládají do kontejneru aplikačního serveru, kde jsou poskytovány pod svým názvem ostatním technologiím. Pro JSP stránky se komponenty deklarují na každé stránce, kde jsou používány. [6] [9]

```
<jsp:useBean id="nazev" class="skola.Uzivatel" scope="session"/>
```

Pro JSF jsou deklarovány globálně v souboru faces-config.xml, nebo pomocí anotace *@ManagedBean*. [8]

```
<managed-bean>
        <managed-bean-name>nazev</managed-bean-name>
        <managed-bean-class>skola.Uzivatel</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Konvence je v souladu s pravidly objektově orientovaného programování. Smí obsahovat pouze bezargumentový konstruktor. Veškeré proměnné mají private identifikátor přístupu, a každá proměnná musí mít *setPromena, getPromena* metodu. Pokud se jedná o *boolean*, místo *getPromena* se použije *isPromena*. Většinou se stav uživatelovi komponenty ukládá do nějakého uložiště, aby při příští návštěvě aplikace mohl začít tam, kde minule skončil. Proto třídy JavaBean implementují rozhraní *Serializable*. [6] [9]

```
public class Uzivatel implements Serializable{
        private static final long serialVersionUID = 1L;

        private String jmeno;               //private proměnná
        private String adresa;
        private String mobil;

        public Uzivatel() {}                //default konstruktor

        public String getJmeno() {                  //getter metoda
                return jmeno;
        }
        public void setJmeno(String jmeno) {        //setter metoda
                this.jmeno = jmeno;
        }
        public String getAdresa() {
                return adresa;
        }
        public void setAdresa(String adresa) {
                this.adresa = adresa;
        }
        public String getMobil() {
                return mobil;
        }
        public void setMobil(String mobil) {
                this.mobil = mobil;
        }
}
```

7 Expression language (EL)

Původně byl EL jazyk součástí JSTL. Sloužil pouze pro vkládání hodnot do značek pomocí syntaxe *${výraz}*. S příchodem JSP 2.0 byl představen jako standardní jazyk pro JSP. Při vzniku JSF byl jazyk vložen i sem, ale s trochu odlišnou syntaxí *#{výraz}*. Později byli

oba tyto jazyky spojeny do jednoho, a vznikla samostatná specifikace. Dnes se používá syntaxe *${}* spíše v JSP stránkách a *#{}* v JSF. Zatímco *${}* se provede okamžitě při překladu a vloží na místo výrazu nějakou hodnotu, tak *#{}* může být provedeno v různých fázích cyklu zpracování. [1]

```
...
<body>
        <jsp:useBean id="uzivatel" class="skola.Uzivatel" scope="page"/>
        <c:out value="${uzivatel.jmeno}"/>
</body>
...
```

Výrazy *#{}* mohou nastavovat proměnné pomocí metod anebo spouštět různé metody i s argumenty. [1]

```
...
<f:view>
        <jsp:useBean id="uzivatel" class="skola.Uzivatel" scope="application"/>
        <h:commandButton value="Uloz" action="#{uzivatel.uloz(uzivatel.jmeno)}"/>
</f:view>
...
```

K proměnným přistupujeme pres tečkovou notaci nebo pomocí [ ]. Stačí pouze proměnou zavolat, a na pozadí se provede volání *getPromena*. Jazyk má několik předdefinovaných objektů, ke kterým může přistupovat ve stránce. V EL je zavedena automatická konverze, takže čísla jsou automaticky převáděna na String a obráceně, kdykoliv je potřeba. Ve výrazu můžeme použít aritmetické a logické operace nebo dokonce podmínku. [1]

```
<h:commandButton value="Nadruhou" action="#{cislo*cislo}"/>
```

8 JSP

Pokud bychom měli vytvářet výstup ze servletů bylo by to zdlouhavé, nepřehledné a náchylné na chyby. Proto vznikly JSP (Java Server Pages) stránky. V MVC architektuře zastupují funkci View. Vytvářejí se podobně jako HTML stránky, ale mezi HTML jsou přimíchány i značky JSP, které generují dynamický obsah. Takovýto soubor má pak příponu .jsp a je uložen ve složce WebContent nebo jeho podsložkách. Při prvním požadavku je soubor převeden na servlet, v jehož výstupním proudu jsou uloženy HTML značky i s dynamicky vygenerovaným obsahem. To vše se vykonává automaticky a v pozadí. V takto

přeloženém servletu se při každém požadavku volá místo metody *doXXX* metoda *_jspService*, která požadavky obsluhuje. [6] [9]

8.1 Syntaxe

Jak už bylo řečeno, mezi HTML jsou přimíchány i značky JSP a celý dokument je pak převeden na servlet. Příkazy JSP mohou být buď ohraničeny hranatými závorkami a znaky procent anebo pomocí xml značek. Existují různé druhy JSP značek a každá je převedena na něco jiného nebo i jinam. Na začátku jsp dokumentu je obvykle uveden typ (*contentType*) a kódování dokumentu (*pageEncoding*). [6] [9]

```
<%@ page contentType="text/html" pageEncoding="windows-1250"%>
```

Pro stránky JSP platí také rozsah proměnných jako u servletu plus navíc má ještě rozsah proměnné pro stránku JSP. [9]

8.1.1 Implicitní objekty

V metodě *_jspService* jsou při překladu automaticky vytvořené některé objekty. To nám usnadňuje práci v JSP, protože je můžeme na stánkách používat, tedy pouze v některých typech příkazů. V deklaraci tyto objekty používat nelze. [6] [9]

Objekt	Popis
request	Objekt typu HttpServletRequest – požadavek
response	Objekt typu HttpServletResponse – odpověď
out	Objekt typu JspWriter – výstupní proud
session	Objekt typu HttpSession – práce s relacemi
application	Objekt typu ServletContext – kontext aplikace
config	Objekt typu ServletConfig – Přístup ke konfiguračním informacím
page	Objekt typu Object – totožné s this v Javě
pageContext	Objekt typu PageContext – Přístup ke kontextu stránky
exception	Objekt typu JspException – pouze v chybových stránkách

Tabulka 4 - seznam implicitních objektů [Autor] podle [9]

8.1.2 Výrazy

Jsou určeny pro vytisknutí nějaké proměnné a doplnění statického nějakým dynamický obsahem. Například vyplnění obsahu tabulky seznamem osob a k nim přidružených informací, které se nacházejí v databázi. Tento příkaz je analogický s příkazem *out.print()* v servletu. Při tvorbě dokumentu HTML, který servlet odesílá je výsledný text vložen na stejné místo, jako byl příkaz ve stránce JSP. Výrazy jsou vkládány za rovnítko mezi značky *<%= ... %>* nebo mezi párovou xml značku *<jsp:expression> … </jsp:expression>*. Mezi tyto značky lze vkládat i EL výrazy. Většinou je používáme pro vytisknutí nějaké proměnné JavaBean objektu. [6] [9]

```
...
<body>
        <b>Jméno: </b><%= request.getParameter("jmeno")%>
        <b>Heslo: </b>
        <jsp:expression>
                request.getParameter("heslo")
        </jsp:expression>
</body>
...
```

8.1.3 Deklarace

Do deklarace je ukládán Java kód, který je určen mimo metodu pro obsluhu požadavků. Jedná se většinou o deklaraci třídních proměnných nebo napsání pomocných metod. Ačkoliv psaní metod do deklarací není správný přístup, ta možnost tu je. Deklarace jsou vkládány mezi značky *<%! ... %>* anebo párové xml značky *<jsp:declaration> … </jsp:declaration>*. Nezapomeňme však, že je vytvořena pouze jedna instance servletu, tudíž veškeré tyto proměnné jsou sdíleny všemi vlákny (požadavky) daného servletu. [6] [9]

```
...
<body>
        <%! int jablek = 10;%>
        <jsp:declaration>
                int lidi = 5;
        </jsp:declaration>
        <p>Každy člověk dostane <%= jablek/lidi%> jablka.</p>
</body>
...
```

8.1.4 Skriptlety

V dnešních JSP by se neměli používat, jsou již zavržené, ale pro zpětnou kompatibilitu byly zachovány. Do skriptletu můžeme napsat Java kód, který je vkládán do obslužné metody *_jspService*. Skriptlety jsou psány mezi značky *<% ... %>* anebo párové xml *<jsp:scriptlet> … </jsp:scriptlet>*. Je možné napsat na první řádek část skriptletu, na druhý potom nějakou HTML značku a na třetí zbytek skriptletu. Toto se například používalo při procházení cyklů, kde při každém cyklu byl vytisknut další řádek tabulky. Jsou zavrženy, protože v MVC architektuře by měla být veškerá logika aplikace uložena v Modelu (JavaBean, EJB, …) a zároveň se může takovýto dokument stát hůře čitelným. [6] [9]

```
...
<body>
        <table>
                <tr>
                        <td colspan="2">Mocniny</td>
                </tr>
                        <% for(int i=1; i<=10; i++){%>
                <tr>
                        <td><%= i%></td><td><%= i*i%></td>
                </tr>
                        <% }%>
        </table>
</body>
...
```

8.1.5 Direktiva

Direktiva říká kontejneru jakou strukturu má mít servlet, který je vytvořen z JSP dokumentu. Sděluje nám, jaké třídy mají být importovány, jestli má být vložena nějaká knihovna tagů, nebo má být do dokumentu začleněn ještě jiný dokument. Direktivu zapisujeme mezi značky *<%@ … %>*. Veškerá konfigurace se provádí pomocí atributů. Jsou 3 druhy direktiv: *page, include, taglib*. [6] [9]

Direktiva page

Může být v jakékoliv části dokumentu a kromě standartního zápisu, ji lze zapsat i pomocí xml značky *<jsp:directive.page ... />*. Direktiva *page* má více než 10 atributů pro různé nastavení. Nejpoužívanější jsou *contentType* a *pageEncoding* pro určení typu dokumentu a jeho kódování. Atribut *import* slouží ke sdělení jaké třídy má servlet importovat,

naopak *extends* určuje nadtřídu servletu. Pokud máme v aplikaci chybovou stránku, může být určena atributem *isErrorPage*, jestli je chybovou stránkou pak přidělíme atributu *true*. Když máme chybovou stránku, můžeme na některé ze stránek nastavit jaká to je a to se provádí pomocí atributu *errorPage*. Dalšími atributy jsou *isThreadSafe, info, language, buffer* aj. [9]

```
...
<%@ page import="java.lang.Math, skola.Kruh"%>
<body>
        <%! int p = Kruh.getPolomer(); %>
        <jsp:directive.page errorPage="chybovaStranka.jsp"/>
        <b>Obsah kruhu je: </b><%= Math.PI*p*p %>
</body>
...
```

Directiva include

Slouží k vložení jiného dokumentu do stránky JSP v době překladu na servlet. Dokument je vložen do té části, kde je vložena direktiva. Lze ji zapsat i xml značkou *<jsp:directive.include ... />*. Obsahuje atribut *file*, v kterém je relativní cesta k vkládanému dokumentu. [9]

```
...
<body>
        <header>
                <%@ include file="hlavicka.jsp"%>
        </header>
        <p> Obsah stránky </p>
        <footer>
                <jsp:directive.include file="paticka.jsp"/>
        </footer>
</body>
...
```

Direktiva taglib

Umožňuje vkládání vlastních knihoven značek. Lze ji vložit i pomocí xml značky *<jsp:directive.taglib ... />*. Direktiva má atributy *uri, tagdir a prefix*. Do *uri* vkládáme cestu nebo název, na který je soubor namapován v web.xml. Atribut *prefix* nám specifikuje předponu pro knihovnu značek a *tagdir* udává cestu ke složce s .tag soubory. [9]

```
...
<head>
        <%@ taglib uri="namapovany" prefix="a"%>
        <jsp:directive.taglib tagdir="/WEB-INF/tags" prefix="b"/>
</head>
...
```

8.1.6 Standartní instrukce

Zapisují se pomocí xml značek, aby byl příkaz lépe čitelný. Každá značka standartní instrukce začíná předponou jsp následuje dvojtečka a název instrukce. Většinou mají atributy, a některé i další vnořené značky. Existuje 10 standartních instrukcí, které nám provádějí určitou operaci. [9]

Instrukce	Popis
jsp:useBean	Najde nebo vytvoří instanci třídy
jsp:setProperty	Nastaví proměnou instance
jsp:getProperty	Přečte proměnou instance
jsp:forward	Předá řízení jinému prvku
jsp:include	Začlení soubor do stránky
jsp:attribute	Definuje atribut v těle značky
jsp:body	Definuje tělo značky
jsp:element	Vytvoří element
jsp:text	Obsahuje text nebo EL příkaz
jsp:plugin	Slouží například pro vkládání apletů

Tabulka 5 - seznam standardních instrukcí [Autor] podle [9]

Nejpoužívanější instrukce jsou pro práci s JavaBean a přesměrovaní nebo předání řízení. Pro použití proměnných JavaBean objektu na stránce JSP musíme daný JavaBean nejdříve načíst. K načtení slouží instrukce *useBean* s atributy *id*, *class* a *scope*. V atributu *class* uvádíme název třídy popřípadě i cestu. Do atributu *id* uvedeme název instance, nejprve se hledá ve vytvořených, a pokud souhlasí název i rozsah je objekt přiřazen, pokud se žádný takový nenalezne, je vytvořen nový. Poslední povinný atribut je *scope*, který nám určuje

rozsah objektu. Můžeme si vybrat ze 4 rozsahů: *application, request, session, page*. Když máme načtený objekt JavaBean můžeme přistupovat k jeho proměnným. Instrukce *setProperty* nám pomocí atributů nastavuje proměnnou daného objektu. Do atributu *name* uvedeme název instance. V *property* uvádíme název proměnné a ve *value* hodnotu. Pokud chceme číst proměnou, použijeme instrukci *getProperty*, u které uvedeme atributy *name* a *property*. Další často používané instrukce jsou *include* a *forward*, které předávají řízení stejně jako metody *RequestDispatcher* v servletu. Do atributu *page* uvedeme prvek, kterému chceme řízení předat. [9]

```
...
<body>
        <jsp:useBean id="info" class="skola.Uzivatel" scope="request"/>
        <table>
                <tr>
                        <td>Jmeno</td>
                        <td><jsp:getProperty property="jmeno" name="info"/></td>
                </tr>
                <tr>
                        <td>Adresa</td>
                        <td><jsp:getProperty property="adresa" name="info"/></td>
                </tr>
                <tr>
                        <td>Telefon</td>
                        <td><jsp:getProperty property="mobil" name="info"/></td>
                </tr>
        </table>
        <footer>
                <jsp:include page="pocasi.jsp"/>
        </footer>
</body>
...
```

8.2 Značky XML

Do dokumentu nemusíme vkládat jen HTML značky nebo JSP příkazy, lze použít i předem definované xml značky, které provádí určitou činnost. Většinou nám zjednodušují práci nebo mohou zastupovat část dokumentu, který se používá opakovaně. Pokud chceme pracovat s těmito značkami, musíme je nejdříve do dokumentu vložit. Na začátku dokumentu proto vložíme direktivu *taglib*, ve které definujeme cestu k deskriptoru knihovny značek nebo složce s .tag soubory, a předponu, kterou budeme ke značkám v dokumentu přistupovat. [6] [9]

8.2.1 Standardní knihovna značek (JSTL)

Některé činnosti jsou v JSP prováděny často, a proto vznikly standartní knihovny značek (JSP Standart Tag Library), které zjednodušují práci s určitými činnostmi. Knihovna s těmito značkami musí být do aplikace naimportována. Existuje 5 standartních knihoven:

Knihovny
http://java.sun.com/jsp/jstl/core
http://java.sun.com/jsp/jstl/function
http://java.sun.com/jsp/jstl/fmt
http://java.sun.com/jsp/jstl/sql
http://java.sun.com/jsp/jstl/xml

Tabulka 6 - seznam standardních knihoven značek [Autor] podle [10]

Často používanou knihovnou je *core*, ve které jsou značky pro základní operace, jako jsou cykly, podmínky anebo práce s URL. Pro práci s řetězci můžeme použít knihovnu *function*. Další často používanou je *fmt* pro zobrazování čísel, časů a řetězců podle zvoleného jazyka. Ačkoliv máme k dispozici knihovnu pro práci s *sql*, neměla by být v JSP používána, protože porušuje architekturu MVC. Poslední knihovnou je *xml* pro práci s Xpath nebo transformacemi u xml dokumentů. [10]

```
...
<body>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
        <c:forEach var="pom" begin="1" end="10" step="1">
                <c:if test="${ pom%2==0 }">
                        <c:out value="${ pom }"/>
                </c:if>
        </c:forEach>
</body>
...
```

8.2.2 Vytváření vlastních knihoven značek

Značky xml si můžeme definovat i sami. To můžeme provést dvěma způsoby. Pomocí deskriptoru knihoven značek (Tag Library Descriptor - TLD) a jeho obslužné třídě nebo uložení čas JSP dokumentu do souboru s příponou .tag. [9] [10]

TLD a obslužná třída

Deskriptor knihovny značek (TLD) je xml konfigurační soubor, ve kterém definujeme název, strukturu a popřípadě atributy xml značky. Zde je určena i obslužná třída pro tuto značku. Tyto soubory ukládáme do složky /WEB-INF/ nebo do nějaké její podsložky například *tlds*. [9][10]

```
...
<tlib-version>1.0</tlib-version>
<short-name>zn</short-name>
<tag>
	<description>Vytiskne hodnotu atributu</description>
	<display-name>znacka</display-name>
	<name>znacka</name>
	<tag-class>skola.Obsluzna</tag-class>
	<body-content>empty</body-content>
	<attribute>
		<name>tisk</name>
		<type>java.lang.String</type>
		<required>true</required>
		<rtexprvalue>true</rtexprvalue>
		<elexprvalue>true</elexprvalue>
	</attribute>
</tag>
...
```

Obslužná třída naopak určuje, jakou akci tato značka vyvolá. Je to Java třída, která implementuje dané rozhraní. Dříve se používalo rozhraní *Tag* anebo *BodyTag*, dneska to je *SimpleTag*, které je jednodušší. Abychom nemuseli implementovat všechny jeho metody, můžeme zdědit třídu *SimpleTagSupport* a přepsat pouze metodu *doTag*, ve které generujeme obsah této značky. [9] [10]

```
public class Obsluzna extends SimpleTagSupport {
	private String tisk;

	public void setTisk(String tisk) {
		this.tisk = tisk;
	}

	public void doTag() throws JspException, IOException {
		if (tisk != null) {
			JspWriter out = getJspContext().getOut();
			out.println(tisk);
		}
	}
}
```

Jestliže chceme na stránkách JSP použít vlastní xml značky, musíme na začátek dokumentu vložit direktivu *taglib*. K značkám potom přistupujeme pomocí předpony definované v direktivě. [9]

```
...
<body>
        <%@ taglib uri="/WEB-INF/tlds/znacky.tld" prefix="zn"%>
        <zn:znacka tisk="vytisklý text"/>
</body>
...
```

Tag soubor

V tomto souboru je část JSP dokumentu, který může být vkládán na jiné JSP stránky. Soubory mají příponu .tag a ukládáme je do složky /WEB-INF/tags. Takovéto soubory mají vlastní direktivu, například místo *page* mají tyto soubory direktivu *tag*. Další často používané jsou *attribute* nebo *variable*. [10]

```
<%@ tag language="java" pageEncoding="windows-1250"%>
<%@ attribute name="a" required="true" type="java.lang.Integer"%>
<%@ attribute name="b" required="true" type="java.lang.Integer"%>
<jsp:text>
        Obsah čtverce je ${ a*b }
</jsp:text>
```

Pokud chceme k těmto souborům přistupovat z JSP stránek, musíme v direktivě *taglib* uvést *tagdir* atribut, který udává cestu k složce, kde jsou tyto soubory. Vkládaná značka potom obsahuje předponu nastavenou v direktivě, následovanou názvem souboru, který chceme začlenit. Popřípadě můžeme předat i atributy, pokud jsou v tomto souboru deklarované nebo dokonce vyžadované. [10]

```
...
<body>
        <%@ taglib tagdir="/WEB-INF/tags" prefix="zn"%>
        <zn:obsahCtverce a="5" b="10"/>
</body>
...
```

9 Java Server Faces

Je to MVC framework, který se stal standardem a byl zařazen do platformy Java EE. Je založený na, vkládaní znovupoužitelných komponent uživatelského rozhraní do souborů JSP nebo HTML. Abychom, mohly komponenty vkládat, musíme v dokumentu deklarovat knihovnu značek, kterou chceme používat. Takového soubory reprezentují view. Vrstvu model zastupují managed beany, které propojujeme pomocí EL s některými komponentami. Veškeré příchozí požadavky jdou přes FacesServlet, který řídí životní cyklus požadavku. Tento servlet zastupuje funkci kontroléru. JSF technologie obsahuje knihovny značek, pomocí kterých vkládáme komponenty na stránky a třídy, které zajišťují funkcionalitu těchto značek nebo také konverzi, validaci, navigaci či zpracování výjimek. Tyto značky jsou poté převedeny renderem do požadované technologie. Záleží jaký render kit použijeme. Jako výchozí pro JSF je HTML. [1] [8]

9.1 Základní knihovny značek

Knihovny můžeme vložit pomocí direktivy *taglib* do JSP souborů, nebo jako jmenný prostor v souboru HTML. Doporučovaná je práce s HTML souborem. JSF má 5 základních knihoven značek:

Knihovny
http://xmlns.jcp.org/jsf/html
http://xmlns.jcp.org/jsf/core
http://xmlns.jcp.org/jsf/facelets
http://xmlns.jcp.org/jsf/composite
http://xmlns.jcp.org/jsf/passthrough

Tabulka 7 - seznam základních knihoven značek [Autor] podle [8]

Knihovna *html* obsahuje značky základních komponent, jako jsou tlačítka, formuláře, obrázky, tabulky nebo vstupní textové pole. Duležitou značkou je <*message*> a <*messages*>. První nám zobrazí chybovou zprávu sdruženou s určitou komponentou. Druhá nám zobrazí všechny chybové zprávy. Všechny značky *html* obsahují atribut *id*, ve kterém je uveden unikátní název komponenty. [1][8]

V *core* jsou značky pro vkládání validátorů, konvertorů, posluchačů nebo značka <*f:view*>, do které jsou vkládány všechny JSF značky a je v ní vytvářena stromová struktura komponent. [1]

```
<!DOCTYPE html>
<html xmlns:h="http://xmlns.jcp.org/jsf/html"
        xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:body>
<f:view>
        <h:form>
                <h:outputLabel id="tx" value="Zadej heslo " />
                <h:inputSecret id="heslo" value="#{trida.heslo}">
                        <f:validateLength minimum="3" maximum="10" />
                </h:inputSecret>
                <h:commandButton id="tl" action="jina" value="Odeslat" />
                <h:message for="heslo" style="color:red" />
        </h:form>
</f:view>
</h:body>
</html>
```

Knihovna *facelets* slouží k tvorbě a použití šablon. Pokud bychom chtěli udělat vlastní komponentu, která se skládá z více menších prvků, můžeme použít knihovnu *composite*. Takovouto komponentu vytváříme v samostatném souboru HTML, který musí být uložen v libovolné podsložce složky resources. Mezi párovou značku *interface* vkládáme atributy, kterými uživatel předává komponentě informace, a v párové značce *implementation* jsou tyto informace použity k vytvoření komponenty podle daného vzoru. [1] [8]

```
<!DOCTYPE html>
<html xmlns:h="http://xmlns.jcp.org/jsf/html"
        xmlns:f="http://xmlns.jcp.org/jsf/core"
        xmlns:composite="http://xmlns.jcp.org/jsf/composite">

        <composite:interface>
                <composite:attribute name="jmeno"/>
        </composite:interface>

        <composite:implementation>
                <h:outputFormat value="Všechno nejlepší {0}">
                        <f:param value="#{cc.attrs.jmeno}"/>
                </h:outputFormat>
        </composite:implementation>
</html>
```

Pro použití takto vytvořené komponenty vložíme knihovnu *composite* následovanou názvem složky kde je komponenta uložena. Ke komponentě se poté přistupuje pomocí předložky a názvu souboru komponenty. [1]

```
<!DOCTYPE html>
<html xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:k="http://xmlns.jcp.org/jsf/composite/komponenty">

<h:body>
      <k:prani jmeno="Roman"/>
</h:body>
</html>
```

Poslední knihovna *passthrough* je užitečná, pokud bychom chtěli vložit libovolný attribut do značky, který bude renderer ignorovat a vloží jej do již výsledné značky. Využívá se hlavně při práci s HTML5. [8]

```
<!DOCTYPE html>
<html xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:p="http://xmlns.jcp.org/jsf/passthrough">

<h:body>
      <h:commandButton value="Odeslat" p:nejaky="atribut"/>
</h:body>
</html>
```

Toto tlačítko by výsledně vypadalo takto:

```
<input type="submit" name="j_idt6" value="Odeslat" nejaky="atribut"/>
```

9.2 Managed Bean

Jsou to JavaBean třídy, které musí být registrovány v souboru faces-config.xml nebo pomocí anotací. Managed bean reprezentují model v MVC architektuře. Pro registraci v souboru faces-config.xml použijeme značku <*managed-bean*> a její pod značky, které nám určují, jakým názvem budeme k instanci přistupovat z JSF stránek, z jaké třídy má být vytvořena a jaký má mít rozsah. Minimálně tyto 3 pod značky musí být deklarované. [1]

```
<managed-bean>
      <managed-bean-name>nazev</managed-bean-name>
      <managed-bean-class>balik.Trida</managed-bean-class>
      <managed-bean-scope>Session</managed-bean-scope>
</managed-bean>
```

Alternativním řešením je registrace pomocí anotací. Nad třídou deklarujeme *@ManagedBean*, která může mít atributy *name* a *eager*. První nám určuje název, pod kterým

budeme k dané instanci přistupovat. Druhý určuje, zda má být instance vytvořena ještě před tím než je požadována. Spolu s touto anotací přidáváme ještě jednu, která nám určuje rozsah platnosti. Pro managed bean existuje 6 anotací s rozsahem platnosti: *@ApplicationScoped*, *@SessionScoped*, *@ViewScoped*, *@RequestScoped*, *@NoneScoped* a *@CustomScoped*. [1]

```
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="nazev", eager=true)
@SessionScoped
public class Trida implements Serializable{
	private static final long serialVersionUID = 1L;

	private String promena;
	private int cislo;

	//getter a setter metody ...
}
```

9.3 Konvertory

Jednou z výhod JSF jsou konvertory. Umožňují úpravu datového typu z JSF stránek do požadovaného v managed beanu a naopak. Takže pokud budeme do textového pole zadávat například cenu a toto pole bude spojeno s proměnou typu *int* v managed beanu, tak se automaticky použije implicitní konvertor *IntegerConverter*. Tyto implicitní konvertory jsou uloženy v balíčku *javax.faces.convert*. Jsou pro všechny základní datové typy jako je *int*, *double* nebo *boolean*, a některé pro speciální třídy jako je například *Date* nebo *BigDecimal*. Pokud nám však žádný nevyhovuje, můžeme si vytvořit vlastní pomocí rozhraní *javax.faces.convert.Converter*. Takovýto konvertor musí být registrován v souboru faces-config.xml nebo pomocí anotace *@FacesConverter*. [1] [8]

```
@FacesConverter(value="nazev")
public class Konvertor implements Converter{

@Override
public Object getAsObject(FacesContext a, UIComponent b, String c) {
		// implementace konverze z view do modelu
	}

@Override
public String getAsString(FacesContext a, UIComponent b, Object c) {
		// implementace konverze z modelu do view
	}
}
```

Jestliže chceme přiřadit konvertor přímo ke značce, použijeme atribut *converter* do něhož vložíme třídu konvertoru, který chceme použít nebo název vlastního konvertoru. Atribut *converter* je u všech vstupních a výstupních polích. Použít můžeme ještě atribut *converterMessage*, ve kterém zadáváme zprávu pro případ, že uživatel zadá znaky, které nelze převést na požadovaný datový typ. V takovémto případě je předána chybová hláška značce *message*. [1] [11]

```
...
<h:form>
        <h:inputText id="pole" converter="nazev"/>
        <h:commandButton id="tl" value="Odeslat"/>
</h:form>
...
```

Dalším způsobem jak přiřadit konvertor přímo k značce je použití značky konvertoru z knihovny *core*. Mezi párové značky vložíme značku *converter* a do jejího atributu *converterId* vložíme požadovaný typ, na který chceme hodnotu převést. [1] [11]

```
...
<h:form>
        <h:inputText id="pole">
                <f:converter converterId="nazev"/>
        </h:inputText>
        <h:commandButton id="tl" value="Odeslat"/>
</h:form>
...
```

Pro převod na datum a číslice byli vytvořeny vlastní značky. Když pracujeme s datem nebo časem můžeme použít značku *convertDateTime*, u které například použijeme atribut *pattern* nebo *locale* k nastavení zobrazovanému formátu. [1] [11]

Pokud pracujeme s číselnými hodnotami, můžeme použít značku *convertNumber*, kde pomocí atributů například nastavíme počet čísel zobrazených za desetinou čárkou nebo v jakém formátu se má zobrazovat. [1] [11]

9.4 Zpracování událostí

V JSF lze zachytávat události, které vznikly na stránce a reagovat na ně určitou akcí. Existují tři druhy událostí buďto *ActionEvent*, *ValueChangeEvent* nebo *PhaseEvent*. První vzniká při kliknutí u komponent, jako je tlačítko nebo odkaz. Druhá je způsobena změnou hodnoty komponenty, například u textového pole nebo checkboxu a třetí vzniká v jednotlivých fázích životního cyklu požadavku. Takovéto události můžeme zachytávat dvěma

způsoby. První je vytvořit třídu, která implementuje rozhraní buďto *javax.faces.event.ActionListener* ,*ValueChangeListener* nebo *PhaseListener*. V případě *ActionListener* implementujeme metodu *proccesListener,* která danou událost obsluhuje. To samé platí u *ValueChangeListener* s tou výjimkou, že implementuje metodu *proccesValueChange* a u *PhaseListener* implementujeme 3 metody. [1]

```
public class Posluchač implements ActionListener{

	@Override
	public void processAction(ActionEvent a) {
		FacesContext o = FacesContext.getCurrentInstance();
		o.addMessage("tl", new FacesMessage("chybová zpráva"));
	}
}
```

Ke každému posluchači je vytvořena i jeho značka v *core* knihovně, takže pokud chceme nějaké komponentě přiřadit posluchače, vložíme mezi párové značky této komponenty značku posluchače. Lze přidat i více než jeden. V takovémto případě jsou události zpracovávány podle pořadí přidaných posluchačů. [1] [11]

```
...
<h:form>
	<h:commandButton id="tl" value="Odeslat">
		<f:actionListener type="balik.Posluchač"/>
	</h:commandButton>
	<h:messages style="color:red"/>
</h:form>
...
```

Druhým způsobem jak zpracovávat události je v metodě managed beanu. Daná metoda musí mít *public* identifikátor přístupu, *void* jako návratový typ a v parametru třídu události, kterou bude metoda obsluhovat. [1]

```
...
public void zmena(ValueChangeEvent u){
	FacesContext context = FacesContext.getCurrentInstance();
	Osoba os = context.getApplication().evaluateExpressionGet(context, "#{osoba}", Osoba.class);
	os.setPohlavi((String)u.getNewValue());
}
...
```

Pokud chceme, aby komponenta zpracovávala události tímto způsobem, musíme do atributu *actionListener* nebo *valueChangeListener* vložit název metody, která bude danou událost zpracovávat. [11]

```
...
<h:form>
        <h:selectOneMenu id="menu" valueChangeListener="#{trida.zmena}">
                <f:selectItem itemValue="Muž" itemLabel="Muž" />
                <f:selectItem itemValue="Žena" itemLabel="Žena" />
        </h:selectOneMenu>
        <h:commandButton id="tl" value="Odeslat"/>
</h:form>
...
```

9.5 Validátory

Validátory slouží pro zajištění správnosti dat ukládaných do managed beanu. Pokud například zadáváme poštovní směrovací číslo, potřebujeme zajistit, aby mělo přesně 5 číslic ani o jednu víc ani míň. Existují validátory rozsahu, počtu znaků nebo podle vzoru. Tyto validátory jsou uloženy v balíčku *javax.faces.validator*. [1]

Všechny značky reprezentující nějaké vstupní pole mají atributy *validator*, *validatorMessage*, *required* a *requiredMessage*. Do *validator* vkládáme název našeho validátoru. V atributu *required* musí být *boolean* hodnota, která říká, zda pole musí být vyplněno nebo ne. Do *validatorMessage* a *requiredMessage* vkládáme chybovou zprávu, která se objeví ve značce *message*, když zadaná hodnota neodpovídá našemu validátoru nebo je vyžadováno vyplnění pole. [11]

```
...
<h:form>
        <h:inputText id="vek" validator="#{trida.zkontrolujVek}"/>
        <h:commandButton id="tl" value="Odeslat"/>
        <h:message for="vek"/>
</h:form>
...
```

Pro vložení validátoru můžeme použít i jednu ze značek z knihovny *core*, které lze vkládat mezi párové značky vstupních polí. Jednotlivé značky jsou navázány na validátory z balíčku *javax.faces.validator*. Například pomocí značky *validateLength* můžeme hodnotu omezit minimálním nebo maximálním počtem znaků. Značka *validateLongRange* nám určuje, v jakém rozmezí může být číselná hodnota. Samozřejmostí je i možnost vložení vlastního validátoru pomocí značky *validator*. [11]

```
...
<h:form>
	<h:inputText id="psc" required="true">
		<f:validator validatorId="validPSC"/>
	</h:inputText>
	<h:commandButton id="tl" value="Odeslat" />
	<h:message for="psc" />
</h:form>
...
```

9.5.1 Tvorba vlastních validátorů

Pokud nám žádný z validátorů nevyhovuje, můžeme si vytvořit vlastní. Každý vlastní validátor musí být zaregistrovaný buďto v souboru faces-config.xml nebo pomocí anotace *@FacesValidator*. Validátor vytvoříme tím, že třídě implementujeme rozhraní *javax.faces.validator.Validator* a jeho metodu *validate*. [1]

```
@FacesValidator(value="validPSC")
public class Validátor implements Validator{

@Override
public void validate(FacesContext a, UIComponent b, Object c) throws ValidatorException {
	String pom = c.toString();
	if(pom.length()!=5){
		throw new ValidatorException(new FacesMessage("chyba"));
	}
}
}
```

Dalším způsobem jak zkontrolovat data je pomocí metody v managed beanu. Daná metoda musí mít určitou signaturu. K takovéto metodě se z JSF stránek přistupuje pomocí atributu *validator*.

```
...
public void zkontrolujVek(FacesContext a, UIComponent b, Object c) throws ValidatorException {
	String pom = c.toString();
	int vek = Integer.parseInt(pom);
	if(vek<=0 || vek>120){
		throw new ValidatorException(new FacesMessage("chyba"));
	}
}
...
```

9.6 Vkládání závislostí

Jednou z předností tohoto frameworku je vkládání závislostí neboli dependency injection. Do managed beanu A lze vložit jakoukoliv proměnou z beanu B nebo samotný bean B. Toho lze dosáhnout, pokud do třídy beanu A vložíme anotaci *@ManagedProperty* a v atributu *value* uvedeme pomocí EL název beanu popřípadě i jeho proměnné. Pod touto anotací musí být deklarována třída, popřípadě proměnná, kterou do beanu A vkládáme. Tento vložený objekt nebo proměnná musí mít minimálně setter metodu jinak server vyhodí výjimku. [12]

```
@ManagedBean
@SessionScoped
public class BeanA {
	@ManagedProperty("#{beanB}")
	private BeanB beanB;

	@ManagedProperty("#{beanC.promena}")
	private String promena;

	public void setBeanB(BeanB beanB) {
		this.beanB = beanB;
	}
	public void setPromena(String promena) {
		this.promena = promena;
	}
}
```

9.7 Životní cyklus JavaServer Faces požadavku

Každý cyklus začíná zasláním požadavku od uživatele. Ten pak prochází několika fázemi, kde jsou provedeny různé operace a následně je odeslána uživateli odpověď. Tyto požadavky dělíme na initial a postback. Initial požadavky nastávají při první návštěvě stránky, například pokud přijdeme z nějakého odkazu. Tento požadavek prochází pouze fázemi Restore view a Render response. Postback požadavek nastává, například když potvrzujeme formulář. Tento požadavek prochází všemi šesti fázemi. [8]

9.7.1 Restore View

Toto je první fáze cyklu. Během této fáze se vytváří obraz stránky, spojují se validátory a posluchače událostí s danými komponentami, a následně jsou tyto informace uloženy v instanci třídy *FacesContext*. Jestliže se jedná o initial požadavek je vytvořen

prázdný obraz a pokračuje se k fázi Render Response. Pokud je požadavek postback, daný obraz stránky už existuje a je obnoven do *FacesContext*. [8]

9.7.2 Apply Request Value

Když máme vytvořený obraz, přecházíme do druhé fáze. Jednotlivým komponentám z obrazu jsou nastaveny nové hodnoty podle parametrů požadavku. Tyto hodnoty jsou uloženy lokálně v každé komponentě. Pokud má některá komponenta nastavený atribut *immediate* na true, je provedena konverze, validace a budou zpracovány události týkající se této komponenty. Jestliže nastane chyba, bude chybová zpráva uložena v *FacesContext*. [8]

9.7.3 Process Validation

V této fázi jsou prováděny validace a konverze. Jsou načteny atributy, které určují pravidla pro validaci a jsou porovnávány s lokální hodnotou komponenty. Pokud dojde k chybě, je chybová zpráva uložena do *FacesContext* a přechází se k Render Response fázi. [8]

9.7.4 Update Model Values

Jakmile je zajištěno, že jsou data ve správném formátu, mohou být lokální data uložena do modelu. Při ukládání do modelu může dojít k chybě. V takovémto případě je vygenerována chybová zpráva, která je uložena v *FacesContext* a přechází se rovnou do Render Response fáze. [8]

9.7.5 Invoke Application

Tato fáze zpracovává události, které uživatel vyvolal, například zmáčknutí tlačítka. Je zachycena událost příslušnou metodou a následně provedená nějaká akce. Poté se přejde do Render Response fáze. [8]

9.7.6 Render Response

Toto je poslední fáze a je zde vygenerována požadovaná stránka. Pokud se jedná o initial požadavek, jsou komponenty přidány do stromu komponent a je vytvořena výsledná stránka. Jestliže se jedná o postback požadavek, strom komponent byl vytvořen už předtím. Pokud je v *FacesContext* nějaká chybová zpráva je načtena ta samá stránka jako předtím, spolu s chybovou zprávou, která je vykreslena místo *message* nebo *messages* značky. [8]

9.8 Konfigurace

Aby mohla JSF aplikace správně fungovat musí se nastavit. K tomu slouží konfigurační soubor faces-config.xml, který je uložen ve složce WEB-INF spolu s web.xml. Pokud budeme chtít používat JSF framework v naší aplikaci, musíme nejprve registrovat *FacesServlet* v souboru web.xml, přes který půjdou všechny požadavky určené pro JSF stránky. Soubor faces-config.xml začíná xml deklarací, následovanou párovou značkou <*faces-config*>, mezi kterou jsou prováděna veškerá nastavení. Zde jsou například registrovány managed bean, vlastní validátory nebo konvertory. Některé nastavení lze provést pomocí anotací přímo u dané třídy [1]

```
...
 <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
 </servlet-mapping>
...
```

9.9 Navigace

Mezi základní nastavení souboru faces-config.xml patří i navigace mezi stránkami. Pravidla pro přechod mezi stránkami jsou uvedena v párové značce <*navigation-rule*>. Tam musí být uvedeno, pro kterou stránku jsou tato pravidla nastavována. To se sděluje značce <*from-view-id*>. Dále tam je párová značka <*navigation-case*>, kde se uvede pravidlo pro jakou akci nebo spuštěnou metodu se přejde na jakou stránku. Tato značka může být přítomná více než jednou, na rozdíl od <*from-view-id*>. [1] [8]

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/webfacesconfig_2_2.xsd" version="2.2">

        <navigation-rule>
                <from-view-id>index.xhtml</from-view-id>
                <navigation-case>
                        <from-outcome>druha</from-outcome>
                        <to-view-id>druha.xhtml</to-view-id>
                </navigation-case>
        </navigation-rule>
</faces-config>
```

Tyto akce jsou předávány z JSF stránek a jsou určeny v komponentě pomocí jejího atributu *action*. Takže například pokud budeme mít tlačítko, u kterého bude nastaven atribut *action* na druha, najde se tato akce a přejde se na stránku určenou těmito pravidly. O hodnotě tohoto atributu je často rozhodnuto metodou z managed beanu pomocí EL jazyka.

```
...
<h:form>
        <h:commandButton id="tl" action="druha" value="Přejít" />
</h:form>
...
```

10 Vlastní práce

10.1 Aplikace sklad

Když jsem se rozhodoval, jakou aplikaci budu vytvářet, nejprve jsem si určil, jakou vytvářet nechci. Nechtěl jsem dělat další z řad internetových obchodů nebo fór, na které se více hodí jiné webové technologie. Jelikož jsem dříve pracoval v jedné firmě, kde jsme osazovali elektrické rozvaděče a kde jsme přistupovali každý den několikrát do skladu, rozhodl jsem se udělat aplikaci skladu.

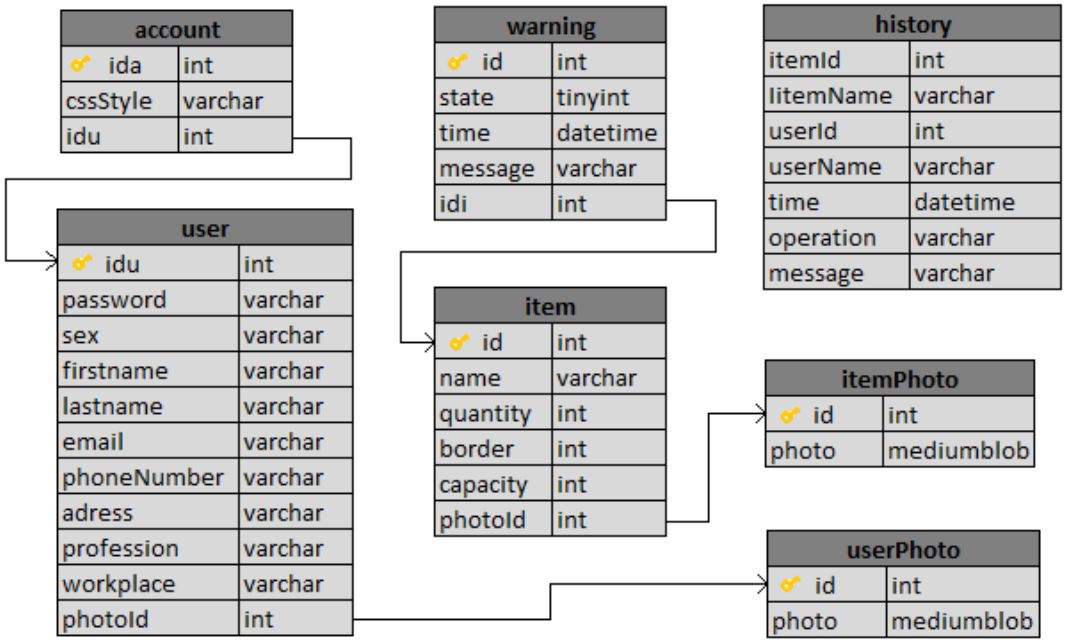
10.2 Návrh aplikace

Aplikace by měla být vhodná jak pro menší tak i pro střední firmy. Tudíž musí mít možnost, aby každý uživatel měl vlastní účet. Musí existovat i účet pro administrátora, který bude tyto účty spravovat. Jak už z názvu vypovídá, aplikace by měla mít nějaký sklad s položkami. Jednotlivé množství položek musí být možno měnit a měla by být hlídána

minimální hranice, kvůli tomu aby položka ve skladu nedošla. Zároveň by veškeré operace ve skladu měli být zaznamenávány.

10.3 Návrh databáze

Pro tuto aplikaci jsem použil databázový systém MySQL, jelikož je zdarma, lze snadno nasadit na různé operační systémy, má přehlednou dokumentaci a díky tomu, že je často používán pro technologii PHP, je většina problémů již vyřešena na některém fórum.



Obrázek 3 - návrh databáze [Autor]

V tabulce user jsou uchovávány informace o jednotlivých uživatelích, sloupec *idu* slouží jako primární klíč tabulky a sloupec *photoId* jako cizí klíč odkazující na tabulku *userPhoto* kde jsou uchovávány fotky uživatelů. Do tabulky *warning* jsou zaznamenávány varovné zprávy, pokud některá položka skladu prolomila minimální hranici množství. Obsahuje primární klíč *id,* stav v jakém se zpráva nachází, jestli je stále bez povšimnutí nebo už si jí někdo všimnul a doplnil zásoby, popřípadě čas kdy byla zpráva vytvořena nebo text varovné zprávy. Tabulka *item* obsahuje opět primární klíč *id*, dále informace o položce skladu a nakonec cizí klíč, který odkazuje na tabulku *itemPhoto*, kde jsou uchovány fotky položek. Poslední tabulkou je *history*, do které jsou ukládány záznamy o všech provedených operací.

10.4 Vytvoření aplikace

Pro tuto aplikaci jsem se rozhodl použít JSF, který jsem popsal v teoretické části. Spolu s JSF použiji PrimeFaces framework. Vybíral jsem mezi RichFaces, IceFaces a PrimeFaces. Jedná se o frameworky, které nabízejí nové komponenty. Vybral jsem si PrimeFaces, protože má nejvíce těchto komponent a k nim rozšířené API. Výslednou aplikaci budeme muset nasadit na nějaký aplikační server. Já jsem si vybral Apache Tomcat verze 7, jelikož je zdarma, jednoduchý a přehledný. Aplikace lze nasadit na jakýkoliv jiný aplikační server. V poslední řadě budeme potřebovat nějaké vývojové prostředí, vybíral jsem mezi NetBeans, Eclipse a IntelliJ IDEA. Zvolil jsem Eclipse, protože je přímo určený pro tvorbu Java EE aplikací, díky tomu je přehledný a dá se v něm dobře vyznat a orientovat, zároveň obsahuje vše potřebné k vývoji aplikace.

10.4.1 Listeners

Srdcem celé aplikace je Managed Bean *listeners*. Na většinu tlačítkových komponent je připnut posluchač, teda spíše metoda v Managed Beanu, která je vykonána při zmáčknutí tlačítka. Pokud se jedná o ajaxové tlačítko je po skončení metody aktualizována část uvedena v atributu update dané komponenty, ale pokud není ajaxové je po skončení metody stránka načtena znova s novými informacemi.

```
public void newItem() {
        item.setDefault();
        item.setId(database.nextIdOfItem());
        account.setPage("item");
        values.setNewItem(true);
}
```

10.4.2 Validators

Další důležitou části aplikace jsou validátory. Jsou přidány převážně všem vstupním komponentám. Slouží k dodržení správnosti vstupních dat, a protože většina vstupů putuje dál do databáze, zároveň kontrolují, aby neprošly nebezpečné znaky pro SQL syntaxi.

```
public void quantityOfItem(FacesContext con, UIComponent comp, Object in) throws ValidatorException {
        isDangerous(in.toString(),"Množství");
        char[] input = in.toString().toCharArray();
        int inputLength = input.length;

        if (inputLength > 10) {
                throw new ValidatorException(new
                FacesMessage(FacesMessage.SEVERITY_ERROR, "Množství max 10 číslic", ""));
        }
        for (char a : input) {
                if (!Character.isDigit(a)) {
                        throw new ValidatorException(new
                        FacesMessage(FacesMessage.SEVERITY_ERROR, "Množství musí být číslo", ""));
                }
        }
}
```

10.4.3 Database

V tomto Beanu probíhá veškerá komunikace s databází. Většina těchto metod je volána právě z *listeners* beanu. Zde je použita třída *Config* k nastavení proměnných pro přihlášení do databáze. V třídě *Config* je přečten soubor *config.properties* a jsou nastaveny statické proměnné.

```
public List<WarnMessage> listOfWarnMessage() {
String sql = "SELECT * FROM warning WHERE state=0 ORDER BY time DESC";
List<WarnMessage> messages = new ArrayList<>();
Connection c = null;
Statement stmt = null;
ResultSet res = null;

try {
        Class.forName("com.mysql.jdbc.Driver");
        c = DriverManager.getConnection(url, USERNAME, PASSWORD);
        stmt = c.createStatement();
        res = stmt.executeQuery(sql);

        while(res.next()){
                WarnMessage message = new WarnMessage();
                message.setId(res.getString("id"));
                message.setState(res.getBoolean("state"));
                message.setDate(res.getTimestamp("time"));
                message.setMessage(res.getString("message"));
                messages.add(message);
        }

} catch (ClassNotFoundException e) {
        e.getMessage();
} catch (SQLException e) {
        e.getMessage();
} finally {
        try {
                res.close();
                stmt.close();
                c.close();
        } catch (SQLException e) {
                e.getMessage();
        }
}
return messages;
}
```

10.4.4 Filtry

Aplikace obsahuje i několik filtrů, který kontrolují jestli je přihlášen nějaký uživatel a pokud ne je přesměrován na přihlašovací stránku. Nebo naopak pokud nějaký uživatel již přihlášen je, z přihlašovací stránky bude přesměrován do aplikace.

```
@WebFilter("/pages/login.xhtml")
public class AuthFilter implements Filter {

@Override
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
        throws IOException, ServletException {

        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;
        HttpSession sess = request.getSession(false);

        if (sess != null) {
                Account account = (Account) sess.getAttribute("account");
                if (account != null) {
                        if (account.isLogged()) {
                                response.sendRedirect(request.getContextPath() +
                                "/pages/secured/application.xhtml");
                        }
                }
        }
        chain.doFilter(req, resp);
}

@Override
public void init(FilterConfig arg0) throws ServletException {}
@Override
public void destroy() {}
}
```

10.4.5 Grafické rozhraní

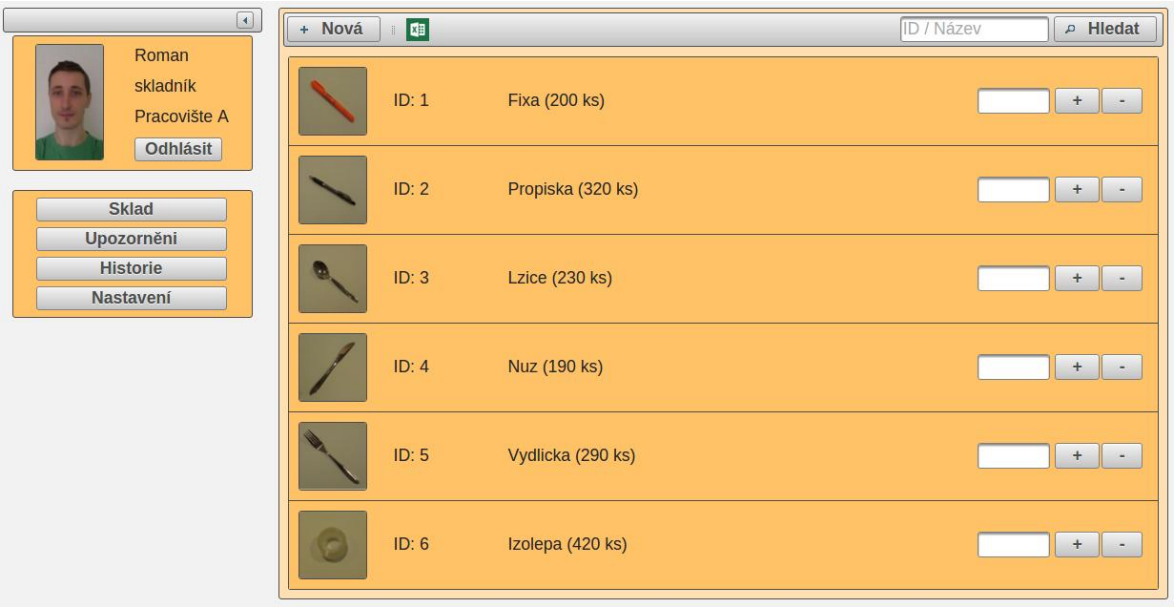
Grafické rozhraní je vytvořeno pomocí PrimeFaces frameworku. Většinu komponent a panelů, kde jsou komponenty uloženy, jsem si upravil pomocí CSS stylů. V aplikaci jsem vytvořil několik barevných kombinací, které si může uživatel zvolit, aby si upravil svůj vlastní účet podle sebe. Tato barevná kombinace je uložena v CSS souboru a pokud je zvolena daná kombinace, název souboru se uloží do tabulky *account* v databázi a pak je načten jako CSS styl v dokumentu. Pro administrátorský účet je vytvořeno trochu odlišné grafické rozhraní s jinými funkcemi než pro ostatní.

10.5 Funkce aplikace

Po přihlášení je uživateli zobrazen informační panel a menu. V informačním panelu jsou uvedeny základní informace o uživateli včetně jeho fotografie, pokud je vložena.

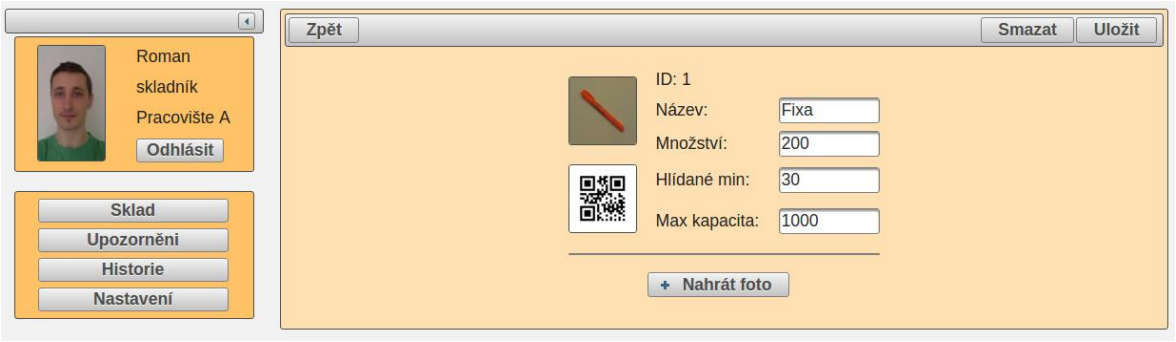
V záložce sklad je zobrazen aktualizovaný seznam položek, které se na skladě nacházejí. V tomto seznamu lze vyhledávat, nebo ho uložit do souboru Excel. U každé

položky je přidáno vstupní pole a dvě tlačítka plus a mínus pro rychlou manipulaci s položkou.



Obrázek 4 - grafické rozhraní (Sklad) [Autor]

Pro bližší informace o položce včetně zobrazení QR kódu slouží jako odkaz obrázek položky. Pro generování QR kódu a souborů Excel jsem použil API třetích stran. Jedná se o *barcode4j-light-2.0.jar*, *commons-codec-1.9.jar*, *core-3.0.0.0.jar*, *javase-3.0.0.0.jar*, *poi-3.13.jar*, *qrgen-1.4.jar*, *xml-apis-1.3.4.jar*, všechny jsou dostupné na stránkách http://mvnrepository.com.

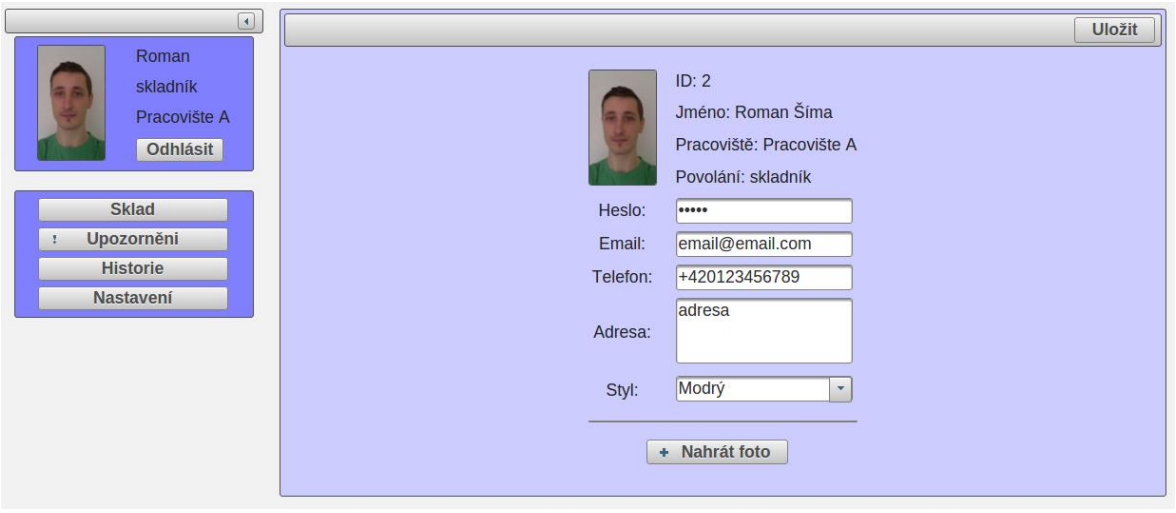


Obrázek 5 - grafické rozhraní (podrobnosti o položce) [Autor]

V záložce upozornění je seznam varovných zpráv. U každé položky musí být nastavena minimální hranice množství. Pokud uživatel vezme ze skladu nějaké množství a prolomí tuto hranici je vytvořena varovná zpráva a u tlačítka v menu je zobrazen vykřičník, který signalizuje, že dochází zásoby.

V historii jsou zaznamenávány všechny operace, které byli provedeny. Jedná se o jakoukoliv manipulaci s položkou, mazání varovných zpráv nebo mazání samotné historie.

Důležitou funkcí aplikace je i nastavení samotného profilu, ať už změna informací o uživateli nebo nastavení jiného barevného stylu.



Obrázek 6 - grafické rozhraní (Nastavení) [Autor]

Administrátorský učet slouží k správě ostatních účtů. Od ostatních uživatelů se liší jiným menu. Administrátor je automaticky každý, který má v kolonce povolání uvedeno

admin. Podobně jako u položek skladu tak i u uživatelů se přejde na podrobnější informace kliknutím na obrázek daného uživatele.

Obrázek 7 - grafické rozhraní (Uživatelé) [Autor]

10.6 Nasazení aplikace

Jelikož mám doma neveřejnou IP adresu a můj server by nebyl přístupný z internetu, rozhodl jsem se využít služby serveru openshift.com, který nabízí zdarma cloudové služby. Ve vytvořeném účtu jsem zvolil, aby byl nainstalován Apache Tomcat 7 a MySQL server. Samozřejmě účet tím, že je zdarma má některá omezení. První je, že URL adresa stránky nemůže být libovolná. Další je omezení místa na aplikačním serveru a databázi, které je sdíleno a může maximálně dosahovat 1Gb. Poslední omezením je, že pokud je aplikace více, jak jeden den v nečinnosti je server vypnut a při dalším požadavku na stránku musí uživatel počkat přibližně minutu, než se server opět nastartuje. Nicméně pokud nechceme platit za Java hosting, je toto jedno z pár řešení. Moje aplikace je dostupná na webové adrese bakalarka-romansima.rhcloud.com/sklad. První přihlášení a kroky v aplikaci mohou být pomalejší, je to kvůli tomu, že jsou vytvářeny session beany. Vytvořenou aplikaci pro tuto práci jsem před nahráním na server musel trochu poupravit a to kvůli tomu, aby nemohly být některé účty smazány nebo změněny. Je to účet s id 1 a povoláním admin. Druhým je účet s id 2 a povolání skladník. Tímto docílím toho, že budou vždy dostupné oba účty.

11 Závěr

Cílem práce bylo seznámit se základními aspekty, které tato platforma nabízí. Proto jsem na základě studiu odborných zdrojů zpracoval teoretickou část, kde jsem stručně představil platformu Java EE, její konkurenty a potřebný software pro vývoj a nasazení takovéto aplikace. Dále jsem se hlouběji zaměřil na vybrané technologie, jako jsou servlety, dynamicky generované JSP stránky a JSF framework, který je do této platformy také zařazen. V každé technologii jsem uvedl několik krátkých příkladů, které slouží pro bližší představu, jak daná technologie funguje. U všech příkladů jsem testoval jejich funkčnost.

Na základě získaných znalostí z teoretické části jsem vytvořil praktický příklad. Jedná se o webovou aplikaci pro evidenci skladu, kde každý uživatel má svůj vlastní účet a může manipulovat s položkami na skladě. To je doplněno o historii jednotlivých operací, nebo o seznam varovných zpráv. Aplikace také umožňuje tisk QR kódů nebo ukládání dat do Excelu. Tuto aplikaci jsem testoval v okruhu blízkých přátel a případné nedostatky jsem opravil. K řešení problémů, na které jsem při vývoji aplikace narazil, mi většinou pomohlo prohledat internet a internetová fóra.

Aplikaci jsem nasadil na server díky openshift.com, které poskytují zdarma cloudové služby. Jednou z nevýhod této cloudové služby tohoto serveru je, že pokud je aplikace více jak den nepoužívána je aplikační server vypnul. Další přístup na stránku trvá něco okolo 1 minuty, čeká se, než se aplikace znova spustí. Aplikace je dostupná na webové stránce bakalarka-romansima.rhcloud.com/sklad. Zdrojové kódy a aplikace jsou přiloženy v příloze.

12 Seznam použitých zdrojů

12.1 Seznam zdrojů

[1] - *The Java EE 6 Tutorial* [online], [1. březen 2016], Dostupné z: http://docs.oracle.com/javaee/6/tutorial/doc/

[2] *PHP: Hypertext Preprocessor* [online], [1. březen 2016], Dostupné z: http://php.net/

[3] *ASP.NET Overview* [online], [1. březen 2016], Dostupné z: https://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx

[4] *Getting Started with Rails — Ruby on Rails Guides* [online], [1. března 2016], Dostupné z: http://guides.rubyonrails.org/getting_started.html

[5] *About the Model 2 Versus Model 1 Architecture* [online], [1. března 2016], Dostupné z: http://download.oracle.com/otn_hosted_doc/jdeveloper/1012/developing_mvc_applications/adf_aboutmvc2.html

[6] HALL, M. *JAVA servlety a stránky JSP*. Praha: Neocortex. 2001. ISBN 8086330060

[7] *Hypertext Transfer Protocol -- HTTP/1.1* [online], [1. Března 2016], Dostupné z: https://www.w3.org/Protocols/rfc2616/rfc2616.html

[8] GUPTA, A. *Java EE Essentials*. O'Reilly Media, 2013. ISBN 9781449370176 1449370179

[9] BURD, B. JSP: *JavaServer Pages Podrobný průvodce*. Praha: Computer Press. 2003. ISBN 807226804X

[10] - *The Java EE 5 Tutorial* [online], [1. března 2016], Dostupné z: https://docs.oracle.com/javaee/5/tutorial/doc

[11] *JSF 2.1 View Declaration Language: Facelets Variant* [online], [1. března 2016], Dostupné z: http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/

[12] *Injecting Managed beans in JSF 2.0* [online], [1. března 2016], Dostupné z: http://www.mkyong.com/jsf2/injecting-managed-beans-in-jsf-2-0/

12.2 Seznam tabulek

Tabulka 1 - seznam stavových kódů [Autor] podle [7] 13
Tabulka 2 - seznam vybraných záhlaví [Autor] podle [7] 13
Tabulka 3 - seznam vybraných posluchačů [Autor] podle [8] 24
Tabulka 4 - seznam implicitních objektů [Autor] podle [9] 28
Tabulka 5 - seznam standardních instrukcí [Autor] podle [9] 32
Tabulka 6 - seznam standardních knihoven značek [Autor] podle [10] 34
Tabulka 7 - seznam základních knihoven značek [Autor] podle [8] 37

12.3 Seznam obrázků

Obrázek 1 - MVC architektura v Java EE [5] 9
Obrázek 2 - souborová struktura v Eclipse [Autor] 11
Obrázek 3 - návrh databáze [Autor] 49
Obrázek 4 - grafické rozhraní (Sklad) [Autor] 54
Obrázek 5 - grafické rozhraní (podrobnosti o položce) [Autor] 55
Obrázek 6 - grafické rozhraní (Nastavení) [Autor] 55
Obrázek 7 - grafické rozhraní (Uživatelé) [Autor] 56

12.4 Seznam příloh

Příloha A – zdrojové kódy
Příloha B – web archiv aplikace