TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2612 – Elektrotechnika a informatika
Studijní obor: 1802R022 – Informatika a logistika

Využití JSP a TagLibs v prostředí dynamické webové aplikace

Use JSP and TagLibs in dynamic web applications

Bakalářská práce

Autor: **Lukáš Bardon**
Vedoucí práce: Ing. Igor Kopetschke
Konzultant: Ing. Igor Kopetschke

V Liberci 1. 4. 2009

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Poděkování

Poděkování patří hlavně vedoucímu mé bakalářské práce Ing. Igoru Kopetschkemu a to nejen za jeho čas a trpělivost, ale také za vědomosti, které mi během konzultací předal a bez kterých bych měl velké potíže s dokončením této bakalářské práce.

Děkuji také Bc. Luboši Remplíkovi za čas se mnou strávený při osvojování jazyka HTML a CSS.

Abstrakt

Obsahem této práce je zmapování možností programovacího jazyka Java v prostředí dynamických webových aplikací. Po přečtení dokumentu by měl čtenář mít jasný pohled na problematiku spojenou s JSP neboli Java Server Pages, s funkcí aplikačního serveru Tomcat spojenou s JSP a dále by měl porozumět výrazům jako jsou například Servlet, JDBC a v neposlední řadě TagLibs.

Druhou částí bakalářské práce je praktická ukázka dynamické webové aplikace využívající výše zmíněné výrazy doplněné o standardní Javovské technologie. Tato webová aplikace by měla sloužit jako inzertní portál pro studenty s tematikou spolujízdy. Aplikace umožňuje interaktivní vyhledávání a zadávání inzerátů prostřednictvím propojení JSP stránek a databáze MySQL.

Abstrakt

Subject of this bachelor thesis is mapping of Java programming language opportunities in dynamic web application environment. Reader should have clear notion of problems associated with JSP (Java Server Pages), and with function of application server Tomcat connected with JSP after reading this document. Further he should understand terms like Servlet, JDBC as well as TagLibs.

Second part of the thesis is practical demonstration of dynamic web application utilizing terms mentioned above completed with standard Java Technologies. This web application should serve as a student advertising portal for sharing a ride.(carpool). The application enables interactive searching and posting ads through interconnection of JSP sites and MySQL database.

Obsah:

Poděkování **4**

Abstrakt **5**

Úvod **8**

Slovník zkratek **9**

1. JSP **10**

2. Odesílání a překlad JSP dokumentu **11**

3. Servlety **12**

3.1 Použití servletu **12**
3.2 Jak vypadá servlet **13**

4. Princip JSP **15**

4.1 Základní stavební prvky JSP **15**
4.1.1 Deklarace 15
4.1.2 Výrazy 16
4.1.3 Skriplety 16
4.2 Implicitní objekty **18**
4.2.1 Objekt Request 18
4.2.2 Objekt response 19
4.2.3 Objekt out 19
4.2.4 Objekt session 19
4.2.5 Další implicitní objekty 20
4.3 Direktivy JSP **20**
4.3.1 Direktiva include 21
4.3.2 Direktiva page 22
4.3.3 Direktiva taglib 24

5. Srovnání JSP a Servlet **25**

6. JSP a databáze **27**

6.1 Komunikace databáze server **27**
6.2 MySQL **28**
6.3 Tvorba datového spojení **28**
6.3.1 Inicializace ovladače 28
6.3.2 Inicializace spojení 28
6.4 Vytvoření příkazu **29**

7. Tvorba vlastních elementů **31**

7.1 Konfigurace vlastní značky **32**
7.2 Obslužná třída elementu **34**
7.2.1 Obslužná třída značky prázdného elementu 34

7.3 Atributy vlastních elementů 36
7.4 Vlastní značka elementu s tělem 37
7.5 Zpracování těla elementu 38
7.5.1 Zpracování těla pomocí TagSupport 38
7.5.2 Zpracování těla pomocí BodyTagSupport 39
8. Apache Tomcat 40
9. Praktická část (webová aplikace Spolujízda) 41
10. Závěr 43
Seznam použité literatury 44

Úvod

Java jako taková je jedním z nejrozšířenějších programovacích jazyků na světě. Využívají ji hlavně vývojáři desktopových aplikací, programátoři aplikací pro mobilní telefony, ale nyní i vývojáři dynamických webových aplikací. S nástupem JSP se programátorům Javy otevírají další možnosti sebezdokonalování a zlepšení indexu na trhu práce.

A právě programování dynamických webových aplikací s využitím JSP je náplní této bakalářské práce. Co se postupně čtenář dozví při pročítání této práce. Samozřejmě zde bude objasněno jak funguje komunikace mezi stránkami JSP a webovým prohlížečem pomocí aplikačního serveru Tomcat. Budou zde zmapovány možnosti používání různých skriptovacích prvků JSP, implicitních objektů, direktiv a JSP akcí.

V další části dokumentu se zaměříme na komunikaci JSP s databází. Právě komunikace JSP s databází je nezbytnou součástí praktické části bakalářské práce. V databázi jsou uloženy inzeráty a za pomoci formuláře se z databáze inzeráty načítají nebo se do ní ukládají.

V závěrečné části práce se dozvíme základní informace o vytváření a použití vlastních elementů na stránkách JSP. Jedná se o technologii TagLibs. Vytváření vlastních tagů je silným nástrojem pro zpřehlednění kódu a pro zlepšení komunikace ve vývojářském týmu, např. mezi programátorem a grafikem.

Slovník zkratek

JSP	Java Server Pages
JDBC	Java Database Connectivity
ODBC	Open Database Connectivity
MySQL	relační databáze typu DBMS (database managment system)
SQL	Structured Query Language
Taglibs	JSP tag libraries define declarative
API	application programming interface
HTML	Hypertext markup Language
TAG	Formátovací značka HTML
URL	Uniform Resource Locator
TCP/IP	Protokol transportní vrstvy sítě
Cookies	Soubor vytvořený webovou stránkou uložený na disku uživatele
ICQ	Komunikační proprogram

1. JSP

Co je to vlastně JSP? Jakou hraje roli v prostředí dynamické webové aplikace? JSP je silným programovacím nástrojem na straně serveru.

Internetová stránka jde samozřejmě napsat jen za pomoci jazyka HTML a pomocí elementů, které HTML poskytuje (tzv. tagy). Takže není překvapením, že nám vznikne pouze statická webová aplikace. HTML totiž neposkytuje žádný mechanismus pro kontrolu toku zpracování.

Právě v těchto situacích, kdy potřebujeme za určitých okolností použít například cyklus, podmínku nebo přiřadit hodnotu odeslanou z nějakého formuláře do proměnné, přichází ke slovu JSP.

```
<% if (session.isNew()) { %>
     <p> Jsme rádi že jste k nám zavítal.</p>
<% } else {%>
     <p> Vítejte zpět. </p>
<%  } %>
```

výpis 1.1: ***Příklad jednoduchého dokumentu JSP***

JSP je tedy nástroj, který dává stránkám dynamiku a inteligenci. V této části si rozebereme nástroje, které JSP používá pro udílení dynamiky webovým aplikacím. Dále si vysvětlíme jak se přeloží JSP dokument, aby ho mohl internetový prohlížeč bez problému zobrazit.

2. Odesílání a překlad JSP dokumentu

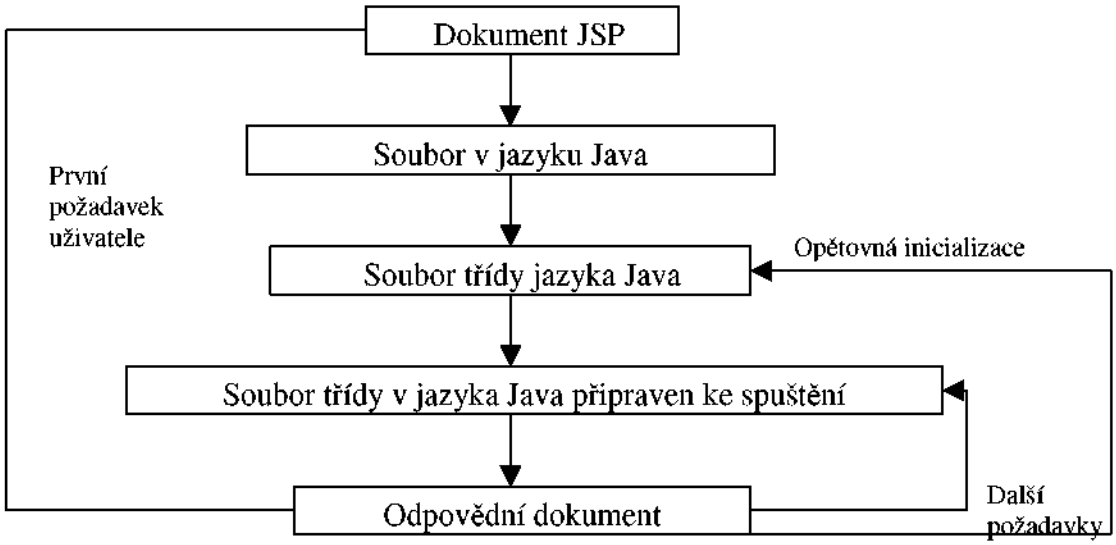
Jak jsme již zjistili v předchozí kapitole, je soubor JSP směsí Javy a jazyka HTML. Samotný prohlížeč si s touto kombinací neporadí. Webový prohlížeč očekává pouze značky (tagy) jazyka HTML. Jak tedy zajistit aby prohlížeč správně přeložil a zobrazil daný dokument JSP?

Celý cyklus je poměrně náročný, takže si jednotlivé části postupně rozebereme. Dokument JSP musíme mít uložen na počítači, na kterém běží serverový software, jehož součástí bývá i JSP kontejner, který umí zpracovávat dokumenty JSP (například Apache Tomcat 6.0.16).

Při prvním dotazu uživatele na JSP stránku je dokument přeložen kontejnerem na standardní program v Javě. Vzniká tedy soubor **_nazev.java**, který označujeme jako servlet. Tento servlet, stejně jako ostatní přeložené JSP dokumenty, se ukládá do zvláštního formuláře daného projektu. V dalším kroku jsou servlety přečteny překladačem jazyku Java a vznikají z nich binární soubory třídy **_nazev.class**. V souboru tříd jsou obsaženy veškeré příkazy pro tvorbu odpovědních dokumentů a pravidla jejich odesílání na klientský webový prohlížeč.

Kontejner načte soubor třídy a na základě jeho definice vytvoří nový objekt a nastaví všechny jeho proměnné. Dále vytvoří odpovědní dokument, jež se skládá pouze z HTML tagů a odešle ho na uživatelův prohlížeč.

Obrovskou výhodou je, že tento náročný cyklus se odehrává pouze pro první dotaz na daný dokument. Jakmile by, ten samý nebo jiný uživatel poslal stejný dotaz, už se bude vytvářet pouze nový odpovědní dokument. Tak to může kontejner opakovat dokud příslušný soubor třídy neodstraní. Dojde-li k odstranění souboru, pak musí kontejner znovu nastavit všechny proměnné, vytvořit odpovědní dokument a odeslat ho na klientský prohlížeč.



Obr.1: ***Zpracování dokumentu JSP před odesláním***

3. Servlety

V předchozí kapitole jsme se dozvěděli, že servlet není nic jiného než program v Javě, který vznikne přeložením dokumentu JSP kontejnerem. V této kapitole si rozebereme servlety podrobněji. Hlavně co se využití a formy týče.

3.1 Použití servletu

Výčet použití servetu je opravdu rozsahlý, takže vyberu jen některé případy, které se mi hodily při sestavování praktické části bakalářské práce.

- Pomocí metody POST může přenášet klientem zadaná data na server a dále je zpracovávat
- Umožňuje uložit data do databáze pomocí JDBC
- Vyhledat další informace o požadavku např. z Cookies
- Dokáže zformátovat výstup, vypočíst výsledky, atd.

Servlety vlastně fungují na principu dotaz-odpověď což je typické pro komunikaci klient-server. Pro komunikaci mezi klientem a serverem musejí mít servlety množinu tříd definující standardní rozhraní pro obsluhu dotazů a odpovědí. O tuto problematiku se stará Java Servlet API.

Java Servlet API se skládá ze dvou balíčků: *javax.servlet* a *javax.servlet.http*. Použití těchto dvou balíčků si ukážeme v kapitole 3.2.

3.2 Jak vypadá servlet

Standardní servlet se dědí od třídy *HttpServlet* a jeho nejpoužívanější akcí je překrytí metody *doGet* nebo *doPost*. Tyto metody se starají o zpracování požadavků GET a POST. Požadavky odeslané metodou GET jsou předávány v hlavičce HTTP protokolu, zatímco požadavky odeslané metodou POST se předávají přímo v těle dotazu. Obě metody *doGet* a *doPost* mají dva stejné parametry. Jsou jimi instance rozhraní *HttpServletRequest* jež nese veškeré potřebné informace o dotazu na server a instance rozhraní *HttpServletResponse*, která obsahuje veškeré informace o odpovědi složené serverem na daný dotaz.

Nyní si uvedeme nejjednodušší možný servlet. Výsledkem servletu nebude nic jiného, než že se na klientském prohlížeči objeví známý zformátovaný text Hello world.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet
{
public void doGet (HttpServletRequest request,
HttpServletResponse response) throws
ServletException, IOException
   {
     PrintWriter out;
     response.setContentType("text/html");
     out = response.getWriter();
     out.println("<HTML><HEAD><TITLE>");
     out.println("Hello world!!");
     out.println("</TITLE></HEAD><BODY>");
     out.println("<H1><CENTER>Hello
     world!!</CENTER></H1>");
```

```
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

výpis 3.1: ***Příklad jednoduchého servletu _HelloWorld.java***

4. Princip JSP

Jak už jsme se dozvěděli z předchozích částí této práce, je JSP stránka směsí HTML tagů a kódů v jazyce Java. Existují však další komponenty, které se starají o dynamický chod webové stránky. Jsou jimi například direktivy a akce JSP, implicitní objekty atd.

Touto tematikou se budeme zabývat v následujících řádcích dokumentu. Právě kódy v Javě a výše zmíněné komponenty se starají o dynamiku JSP stránek.

4.1 Základní stavební prvky JSP

Dokument JSP je složen ze dvou částí: značek HTML a JSP příkazů. Obě tyto části musí být jednoznačně rozlišeny, aby je JSP kontejner při kompilaci rozpoznal. Proto jsou tagy HTML ohraničeny hranatými závorkami (výpis 4.1) a příkazy JSP jsou označeny hranatými závorkami se znakem % (výpis 4.2).

```
<b> toto bude tučně zvýrazněný text v okně prohlížeče
</b>
```

výpis 4.1: ***Příklad jednoduché HTML značky***

```
<% for ( int i=0; i<=10; i++)  %>
```

výpis 4.2: ***Příklad jednoduchého JSP příkazu***

4.1.1 Deklarace

JSP deklarací rozumíme zapsáním jedné nebo více deklarací v jazyce Java a tento kód musíme ohraničit značkami <%! a %>.

Víme tedy že JSP deklarace se může skládat z více deklarací, proto se každá deklarace musí ukončit středníkem.

Jak taková JSP deklarace vypadá si ukážeme na příkladu.

```
<%!
     int i=10;
     String s="retezec";
     boolean  b=true ;
%>
```

výpis 4.3: ***Příklad JSP deklarace***

Ve výpisu (4.3) jsme deklarovali a inicializovali několik proměnných různých typů. Nejprve jsme deklarovali proměnnou *i* typu Integer a přiřadili jsme ji hodnotu *10*. Další proměnná je typu String, pojmenovali jsme ji *s* a přiřadili jsme ji hodnotu *řetězec*. No a konečně poslední proměnná *b* je typu boolean a byla ji přiřazena hodnota *true*.

4.1.2 Výrazy

Vytvoření JSP výrazu je velice jednoduché. Chceme-li vytvořit JSP výraz, stačí jen výraz v Javě ohraničit značkami <%= a %>.

Zpracování JSP příkazu probíhá tak, že server zjistí jakou hodnotu má daný příkaz a tuto hodnotu zobrazí na klientově prohlížeči.

```
<%! int i=10 %>
<%= i++ %>
```

výpis 4.4: ***Příklad JSP deklarace a příkazu***

Kdyby byl výpis (4.4) jediným obsahem těla JSP stránky, pak by klient po jejím navštívení spatřil na svém prohlížeči pouze inkrementovanou hodnotu proměnné *i*. Tudíž v okně webového prohlížeče spatří hodnotu 11.

4.1.3 Skriplety

Skriplet je asi nejvšestrannějším elementem JSP. Skriplet je v dokumentu JSP ohraničen dvojicí značek <% a %>. Jedná se o opravdu silný element z toho důvodu, že do skripletu můžeme napsat jakýkoli kód v Javě.

Další výhodou je, že skriplet nemusí být celistvý a soběstačný. Právě této výhody využívám často v praktické části bakalářské práce. Názorný příklad si předvedeme na situaci, kdy uživateli nabízím možnost zadat do formuláře **formZadani.jsp** informaci o počtu volných míst skrze rolovací seznam s hodnotami 1 až 10.

```
<th align="center">
  <select name="osoba">
     <% for (int i=1 ; i <10 ; i++){%>
       <option><% = i %></option>
     <% } %>
  </select>
</th>
```

výpis 4.5: ***Příklad skriplet*****ů**

Ve výpisu (4.5) jsou zřetelné dva skriplety ohraničené značkami <% a %> a dále HTML tagy pro vytvoření řádku tabulky a pro vytvoření rolovacího seznamu. První skriplet obsahuje cyklus *for* starající se o inkrementaci proměnné *i* při každém průchodu a závorku, která poukazuje na začátek těla cyklu. Druhý skriplet obsahuje pouze závorku, která poukazuje na konec těla cyklu *for*. Tímto způsobem jsem si zajistil, že v těle cyklu *for* mohu použít jakékoli HTML tagy.

Výsledkem toho snažení je zpřehlednění a kultivovanost kódu. Kdybychom totiž v tomto případě nevyužili skripletů, vypadal by kód jako ve výpisu (4.6).

```
<th align="center">
        <select name="osoba">
<option>1</option><option>2</option><option>.....
        </select>
</th>
```

výpis 4.6: ***Příklad kódu bez použití skriplet*****ů**

Ještě je nutno poukázat na deklarace prováděné ve skripletech. Pokud ve skripletu deklaruji proměnnou, pak je proměnná inicializována při každém dotazu na stránku. Naproti tomu JSP deklarace vytvoří proměnnou a nastaví její hodnotu jen během inicializace dokumentu. Další rozdíl mezi těmito deklaracemi je, že proměnná deklarovaná v deklaraci JSP má platnost třídy, zatímco proměnná deklarovaná ve skripletu má platnost metody. Proto je potřeba pečlivě uvážit, v kterých situacích použít jakou deklaraci.

4.2 Implicitní objekty

Implicitní objekty jsou dalším velice silným nástrojem pro vývojáře, kteří programují JSP stránky. Tyto objekty jsou implicitní v tom smyslu,že je lze používat aniž by jste je museli nějakým způsobem definovat. Tyto objekty vznikají automaticky jako součást procesu překladu JSP dokumentu na servlet. Zároveň jsou tyto objekty dostupné v jakýchkoli výrazech a skripletech použitých v dokumentu JSP. V JSP odpovídá každý implicitní objekt určité třídě nebo rozhraní Javy.

4.2.1 Objekt request

Pomocí metod objektu *request* dokážeme zpracovávat informace zaslané klientem pomocí webového požadavku na server. Opět se budu odkazovat na praktickou část bakalářské práce, kde pomocí metod implicitního objektu *request* získávám hodnoty, které klient zadal do formuláře a odeslal k dalšímu zpracování.

Výpis (4.7) je část dokumentu **vyrizeni.jsp** (hodnota proměnné *action* elementu *<form>*), který se stará o přiřazení hodnot získaných z formuláře daným proměnným.

```
<%String start= request.getParameter("od");
String cil= request.getParameter("do");
String datum=request.getParameter("datum");
String cas=request.getParameter("cas");
String osoba=request.getParameter("osoba");
String telefon=request.getParameter("phone");
String icq=request.getParameter("icq");
String mail=request.getParameter("mail");
String poznamky=request.getParameter("poznamky");
%>
```

výpis 4.7: ***Příklad příklad použití implicitního objektu request***

4.2.2 Objekt response

Při přijetí klientského dotazu server okamžitě skládá a posílá odpověď na tento dotaz. Kontejner JSP ukládá informace o této odpovědi právě do implicitního objektu *response*. Vlastnosti této odpovědi můžeme měnit pomocí metod objektu *response*.

4.2.3 Objekt out

Implicitní objekt *out* je instancí třídy *JspWriter* a má dvě důležité metody *out.print()* a *out.println()*. Voláním těchto metod se text (který je zároveň parametrem metod) odesílá do webového dokumentu. Rozdíl mezi těmito dvěma metodami je, že metoda *println()* přidává ke konci řetězce znak nového řádku.

Jinak se objekt *out* používá také k tomu abychom nemuseli "trhat" skriplety na části. Například výpis (4.5) by s použitím implicitního objektu *out* vypadal následovně.

```
<th align="center">
     <select name="osoba">
          <% for (int i=1 ; i <10 ; i++)
               <option>out.print(<% = i %>);</option>
          } %>
     </select>
</th>
```

výpis 4.8: ***Příklad příklad použití implicitního objektu out***

4.2.4 Objekt session

Implicitní objekt *session* je vytvořen kontejnerem JSP v případě, kdy na server dorazí požadavek nového uživatele. Dále při každé další návštěvě už kontejner používá informace ze *session* vytvořené při prvním dotazu uživatele na server.

Ke každé *seseion* lze přiřadit atributy které se pak ukládají jako soubory *cookie* na pevný disk uživatele. Programátoři přistupují k těmto atributům pomocí metod objektu *session*. Pro nastavení atributu slouží metoda *setAttribute()* a pro získání hodnoty atributu slouží metoda *getAttribute()* objektu *session*.

Implicitní objekt *session* obsahuje také další zajímavé metody. Jejich částečný výčet je znázorněn v tabulce (tab 4.1)

Metoda	Význam
Object getAtrtibute (String atributeName)	Vrací objekt shodný s názvem parametru. Pokud zadaný název atributu neexistuje, pak vrací hodnotu null
Void setAttribute (String Name, Object hodnota)	Přiřadí objekt k dané relaci pod zadaným názvem
getCreationTime()	Vrací čas vzniku uživatelské relace
getLastAccessedTime()	Vrací okamžik poslední návštěvy daného uživatele
getMaxInactiveInterval()	Vrací časový údaj, po jehož uplynutí je relace ukončena. (časový limit nečinnosti)
setMaxInactiveInterval()	Umožňuje nastavit časový limit nečinnosti

tab 4.1: ***Výčet metod objektu session***

4.2.5 Další implicitní objekty

Každý dokument JSP má k dispozici objekt *application*, který slouží k ukládání informací o cestě, ve které je daný dokument uložen.

Objekt *config* obsahuje informace o konfiguraci daného servletu.

Objekt *page* odkazuje na aktuálně zpracovávanou stránku. Slouží například pro nastavení typu obsahu a kódování stránky.

4.3 Direktivy JSP

Direktiva na rozdíl od skriptovacího elementu neobsahuje žádný kód v jazyce Java.

Direktivy JSP si lze představit jako zprávy, které stránka posílá kontejneru JSP a tím žádá o vyřízení určité akce ve prospěch dané stránky.

Direktivu zapíšeme následujícím způsobem:

```
<%@ nazev direktivy nazev-atributu=hodnota nazev-
                atributu=hodnota ...%>
```

výpis 4.9: ***Formát zápisu direktivy***

Direktiva je ohraničena značkami <%@ a %>. Po počáteční značce následuje název direktivy.

Máme k dispozici tři druhy direktiv JSP: *include*, *page* a *taglib*. Každá direktiva se stará o splnění vlastní instrukce, pro kterou byla vytvořena. Za názvem direktivy následuje seznam atributů a jejich hodnot. Každá direktiva má vlastní výčet atributů. Výčtem atributů se budeme zabývat v následujících podkapitolách.

4.3.1 Direktiva include

Direktiva *include* je velice silným a důležitým nástrojem JSP. Jedná se o nástroj, s jehož použitím dosáhneme především usnadnění práce a přehlednosti kódu.

Programátor stránek často naráží na problém, že se v dokumentech JSP často opakují stejné pasáže kódu nebo že je nucen do poměrně přehledného dokumentu vepsat složitý a zdlouhavý kód.

Právě v těchto situacích je potřeba využít direktivy *include*. Direktiva *include* má pouze jeden atribut, a to atribut *file*. Jako hodnota atributu *file* se udává název dokumentu JSP, který chceme do příslušného dokumentu vložit právě prostřednictvím direktivy *include*.

V praktické části této bakalářské práce jsem direktivy *include* hojně využíval hned v několika případech. Vzhledem k tomu, že v horní části každé aplikace je obsaženo menu, je zde využito direktivy *include*. Dále jsem této pomůcky využil pro vkládání formulářů do dokumentu **zadani.jsp** a **hledani.jsp**.

Ve výpisu (4.9) uvidíme kód obsahující HTML tagy charakterizující menu aplikace. Tento dokument je nazván jako **menu.jsp**.

```
<html>
   <ul>
       <li><a href="index.jsp">Úvod</a></li>
       <li><a href="zadani.jsp">Zadání inzerátu</a></li>
       <li><a href="hledani.jsp">Vyhledání inzerátu</a>
       </li>
  </ul>
</html>
```

*Výpis 4.9: **Obsah souboru menu.jsp***

Ve výpisu (4.10) je uvedena část kódu, která je shodná v dokumentech **index.jsp**, **hledani.jsp** a **zadani.jsp**. V této části kódu je využita direktiva *include*, u které je hodnota parametru shodná s názvem dokumentu obsahující kód z výpisu (4.9)

```
<div id="container">
<%@ include file="menu.jsp" %>
.
.
.
</div>
```

výpis 4.10: ***Použití direktivy include***

Výsledkem tohoto snažení bude, že se v klientském prohlížeči na stránkách **index.jsp, zadani.jsp** a **hledani.jsp** na pozici *container* objeví aplikační menu.

4.3.2 Direktiva page

Direktiva *page* je asi nejvšetranější direktivou. Obsahuje nemalý výčet atributů, které upravují spoustu vlastností stránek. Formát zápisu direktivy *page* je podobný jako u direktivy *include*. Direktiva *page* je ohraničena značkami <%@ a %>, a za počáteční značkou se uvádí název *page* a pak následuje výčet atributů a jejich hodnot.

Atributy direktivy *page*:

- *import*
- *errorPage a isErrorPage*
- *session*
- *info*
- *language*
- *contentType*
- *isThreadSafe*
- *buffer*
- *autoFlush*
- *extense*

Podrobně rozeberu pouze atributy, které často používám v praktické části bakalářské práce.

Atribut *import* direktivy *page* slouží ke zkrácení zápisu jednotlivých tříd ve skripletech na dané stránce. Dá se též říci, že pomocí atributu *import* nahrajeme potřebné knihovny. Ve výpisu (4.11) si ukážeme využití atributu *import* direktivy *page* na reálném příkladu.

```
<%@ page import="java.text.*, java.io.*,java.sql.*" %>
```

Výpis 4.11: ***Použití atributu import***

Je-li tento kus kódu přítomen na začátku dokumentu JSP, pak je možno v tomto dokumentu využívat veškerých metod z balíčků *java.text, java.io* a *java.sql.*

Balíček *java.text* obsahuje různé metody pro formátování textu, *java.io* umožňuje využívat různých nástrojů pro vstupně-výstupní operace a balíček *java.sql* je velice potřebný pro komunikaci JSP s databází.

Zbytek atributů a jejich využití bude popsán v tabulce (4.2)

errorPage	Zachycuje výjimky
isErrorPage	Zpřístupňuje implicitní objekt exception
session	Stará se o zpřístupnění relací
info	Pro zdokumentování aplikace
language	Říká kontejneru v jakém jazyce jsou psané deklarace, skriplety a výrazy JSP
contentType	Sděluje klientskému prohlížeči jakého typu jsou příchozí byty
isThreadSafe	Povoluje vytváření kopií dané stránky (na stránce může být v dané chvíli jen jeden nebo více klientů)
buffer	Umožňuje nastavit velikost vyrovnávací paměti pro odpovědní dokumenty
autoFlush	Stará se o přetečení vyrovnávací paměti
extends	Stará se o rozšíření třídy HttpJspBase

tab 4.2: ***Další atributy direktivy page***

4.3.3 Direktiva taglib

JSP umožňuje definovat vlastní uživatelské značky. Značku je nejprve potřeba definovat v souboru deskriptoru knihovny značek (*Tag Library Deskriptor-TLD*), ve kterém se definuje propojení mezi značkami a jejich obslužnými třídami. Dále je třeba značce přiřadit funkci, kterou má značka vykonávat. Tato funkce se zapíše do obslužné třídy elementu.

Aby mohla být značka implementována do dokumentu JSP, musíme nejdříve nastavit, který soubor deskriptoru knihovny značek se má použít. A právě k tomu slouží direktiva *taglib*.

Direktiva *taglib* má dva důležité parametry. Atribut *uri* má hodnotu odpovídající relativní URL cestě vedoucí k souboru s koncovkou *.tld*. Druhý parametr *prefix* má hodnotu obsahující název skupiny určitých tagů. Tato předpona se bude používat v dokumentu JSP před jménem vlastního tagu.

Podrobněji se budeme tvorbou vlastních elementů zabývat v samostatné kapitole.

Příklad využití direktivy *taglib* je ve výpisu (4.12).

```
<%@ taglib uri="tagy" prefix="mojetagy"%>
<div id="contact">
   <mojetagy:nadpis1Tag text="Spolujízda"/>
   <mojetagy:nadpis2Tag text="O čem to zde je? :"/>
   <mojetagy:odstavecTag text="Taky se Vám nechce
   jezdit vlakem nebo autobusem?"/>
   <mojetagy:odstavecTag text="Pak je tento webový
   server určen právě Vám."/>
   <mojetagy:odstavecTag text="Výhodou je interaktivní
   zadávání i vyhledávání inzerátů spolujízdy."/>
</div>
```

Výpis 4.12: ***Příklad použití direktivy taglib***

5. Srovnání JSP a Servlet

Servlety a JSP. Na jednu stranu rozdílné technologie, na druhou stranu zase neodlučitelná a vzájemně provázaná dvojice pro fungování JSP stránek.

Psát servlety znamená psát jedině čistý kód v Javě. Neobjevují se zde JSP značky a HTML značky se zde objevují pouze ve formě výstupního řetězce v metodě *out.print()*. Servlet je vlastně program v jazyce Java spuštěný na webovém serveru, který k tvorbě odpovědi na uživatelský dotaz používá balíčky *javax.servlet* a *javax.servlet.http*. Server spustí metodu *doGet()* jako odpověď na http požadavek. Pro vytvoření a otevření komunikačního kanálu slouží metoda *getWriter()* objektu *response*. Jeli komunikační kanál vytvořen, můžeme použít metodu *println()* k uložení textového řetězce do výstupního proudu. Příklad jednoduchého servletu je ve výpisu (5.1).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class index extends HttpServlet
{
  public void doGet (HttpServletRequest request,
  HttpServletResponse response)
  throws
  ServletException, IOException
  {
     PrintWriter out;
     response.setContentType("text/html");
     out = response.getWriter();
     out.println("<HTML><BODY>");
     out.println("<H1><CENTER>Spolujízda</CENTER></H1>");
     out.println("</BODY></HTML>");
     out.close();
   }
}
```

Výpis 5.1: ***Příklad jednoduchého servletu***

Tento servlet má za důsledek, že se na klientově prohlížeči objeví nápis Spolujízda zformátovaný jako nadpis 1.

Z výpisu (5.1) je patrné, že psaní servetů je poměrně náročné, nepřehledné a poměrně zastaralé. Naproti tomu JSP je mnohem vyspělejší technologií. Dalo by se říci, že JSP nabízí snadnější, přehlednější a hlavně velice intuitivní technologii tvorby servletů. Jak už bylo několikrát zmíněno, největší předností JSP je vzájemné provázání HTML značek s částmi kódu v jazyce Java. JSP nabízí implicitní objekty a velké množství dalších nástrojů pro snadnější programování webových aplikací.Je tedy mnohem lepší psát webovou aplikaci v JSP a servletu přenechat funkci backendové logiky.

Abychom dosáhli stejného výsledku v okně klientského webového prohlížeče jako v případě servletu z výpisu (5.1), stačí v JSP napsat pouze kód obsažený ve výpisu (5.2).

```
<html>
  <body>
    <h1>Spolujízda</h1>
  </body>
</html>
```

Výpis 4.9: ***Obsah souboru index.jsp***

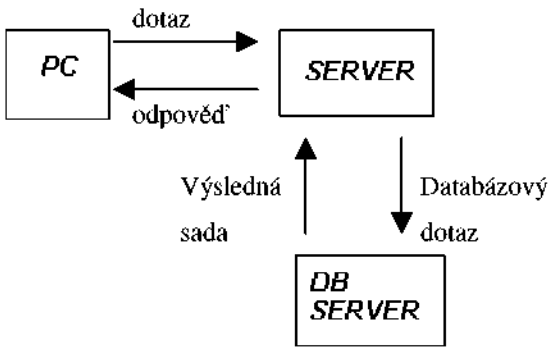
6. JSP a databáze

Možnost propojení JSP s databází je samozřejmostí. Velké procento webových aplikací zpracovává obrovské množství dat, proto je komunikace s databází velice důležitou součástí JSP. V praktické části této bakalářské práce také využívám databázi. Do databáze ukládám data získané po odeslání formuláře starajícího se o odesílání inzerátů. Ale také v databázi vyhledávám inzeráty podle dat odeslaných ve formuláři pro hledání inzerátů.

Databázi si lze představit jako seznam tabulek různě propojených mezi sebou. V mé databázi je pouze jedna tabulka obsahující veškerá data o vloženém inzerátu (start, cíl, datum odjezdu, čas odjezdu, počet volných míst, telefon, icq, e-mail, poznámky a datum vložení inzerátu).

6.1 Komunikace databáze server

Při příchodu klientského dotazu na webový server se dotaz vyřizuje standardním způsobem. Webový server spustí příslušný soubor *.class*, jenž vznikne překladem stránky JSP. Během zpracování dotazu serverový počítač narazí na dotaz, který se odvolává na databázi. Serverový počítač předá příslušná volání softwaru, který je uložen na databázovém serveru (tato komunikace probíhá povětšinou pomocí protokolu TCP/IP). Databázový server požadavek zpracuje, vytvoří výstupní sadu (výsledek) a odešle ji zpět na serverový počítač. Ten dokončí odpovědní dokument a odešle odpověď na klientský prohlížeč.



Obr.2: ***Komunikace databáze-server***

6.2 MySQL

MySQL je multiplatformní databáze. Komunikace s ní probíhá pomocí jazyka SQL. Podobně jako u ostatních SQL databází se jedná o dialekt tohoto jazyka s některými rozšířeními.

Pro svou snadnou implementovatelnost, výkon a především pro to, že se jedná o volně šiřitelný software, má vysoký podíl na v současné době používaných databázích. Velmi oblíbená a často nasazovaná je kombinace MySQL, JSP (PHP) a Apache jako základní software webového serveru.

MySQL bylo od počátku optimalizováno především na rychlost, a to i za cenu některých zjednodušení. Má jen jednoduché způsoby zálohování a až donedávna nepodporovalo pohledy, triggery a uložené procedury. Tyto vlastnosti jsou doplňovány teprve v posledních letech, kdy začaly programátorům již poněkud scházet.

6.3 Tvorba datového spojení

K vytvoření spojení nám postačí dva příkazy. Jeden se stará o inicializaci ovladače a druhý zabezpečuje inicializaci spojení.

6.3.1 Inicializace ovladače

Ovladač je software, který umožňuje komunikaci s určitou databází. V praktické části používám databázi MySQL.

Volání *Classe.forName()* vyhledá ovladač a načte instanci dané třídy do paměti. Jelikož používám databázi MySQL, musím jako parametr metody *Classe.forName()* použít třídu *com.mysql.jdbc.Driver*. Inicializace ovladače je znázorněna ve výpisu (6.1).

```
<%Class.forName("com.mysql.jdbc.Driver").newInstance(); %>
```

Výpis 6.1: ***Inicializace ovladače***

6.3.2 Inicializace spojení

Prostřednictvím databázového ovladače, který je již inicializován, můžeme otevřít datové spojení k dané databázi. Příkaz, který se o tuto práci postará je obsažen ve výpisu (6.2).

```
<% String connectionURL =
"jdbc:mysql://localhost:3306/project?user=root&password=
password";
  Connection connection = null;
  Connection=DriverManager.getConnection(connectionURL);
%>
```

Výpis 6.2: ***Inicializace spojení***

Connection v sobě ponese spojení k dané databázi. *DriverManager* je onen zinicializovaný ovladač. Pro otevření komunikace zavoláme jeho metodu *getConnection*, která má tři parametry: *url, user* a *password*, kde *url* reprezentuje databázové URL ve formátu jdbc:*protokol:jmeno*. *User* obsahuje jméno a *password* obsahuje heslo pro přístup k databázi.

6.4 Vytvoření příkazu

Pro vytváření příkazů slouží rozhraní definovaná v balíčku *java.sgl*. V praktické části bakalářské práce používám rozhraní *Connection*, *Resulset* a velice mocný nástroj *PreparedStatement* pomocí něhož můžeme ve zdroji dat vykonávat SQL příkazy. Další úlohou rozhraní *PreparedStatement* je úspora při použití vázaných proměnných. Ve výpisu (6.3) si ukážeme jak se takový SQL příkaz sestavuje.

```
<%String connectionURL =
"jdbc:mysql://localhost:3306/project?user=root&password
=password";
 Connection connection = null;
 Class.forName("com.mysql.jdbc.Driver").newInstance();
 Connection=DriverManager.getConnection(connectionURL);
 PreparedStatement
 pstm=connection.prepareStatement("SELECT * FROM tab
 WHERE adatum=?");
 java.util.Date n= new java.util.Date();
 pstm.setDate(1, new java.sql.Date(n.getTime()));
 ResultSet rs = pstm.executeQuery();
```

```
%>
```

Výpis 6.3: ***Příklad vytvoření příkazu jazyka SQL***

Tím, že použijeme stejný objekt *PreparedStatement* šetříme nejen zdroje na úrovni aplikace, ale též databázového serveru. Aplikace tímto postupem explicitně databázovému serveru přikazuje použít dříve vytvořený kursor, pro který byl prováděcí plán již sestaven. Server tak s výjimkou prvního volání nemusí provádět optimalizaci dotazu.

Výsledkem výpisu (6.3) je načtení veškerých řádků v databázi, které splňují podmínku, že se sloupeček *adatum* musí shodovat s aktuálním systémovým datem. Tato data jsou pak dále zpracována pomocí instance *rs* typu *ResultSet*.

7. Tvorba vlastních elementů

Základním stavebním prvkem webových stránek jsou bez pochyb HTML značky. Těchto značek je poměrně široká škála a většina z nich se při tvorbě internetových stránek hojně využívá.

Neohrabané propojení rozvržení textu s aplikační logikou často vede ke špatné čitelnosti dokumentu a k velké pracnosti při údržbě dokumentu. Z tohoto důvodu je žádoucí, aby se programátor JSP stránek snažil oddělit aplikační logiku od prezentace obsahu. Taglibs však není jen o vytváření vlastních značek. Taglibs také připravuje stránky pro strojové zpracování a v aplikaci může nahrazovat nejen tagy, ale i aplikační logiku (například cykly).

Tvorbu vlastních značek lze ve své podstatě srovnat s kaskádovými styly. Kaskádové styly sjednocují stránku vizuálně, zatímco TagLibs (JSP tag libraries define declarative) sjednocují generování jednotlivých tagů (HTML značek).

TagLibs je jedním z nejmocnějších nástrojů JSP. V tom má JSP velkou výhodu oproti PHP. Knihovnu značek, kterou jednou vytvoříme, můžeme přenášet a používat v dalších projektech.

Vytváření a používání vlastních značek je tak oblíbené, že vedlo k vytvoření projektu Jakata TagLibs. Tento projekt vznikl pro podporu používání vlastních značek. Je zde volně ke stažení na 25 samostatných knihoven značek. Každá z nich poskytuje určitou specifickou funkci, kterou dříve či později při psaní JSP stránek použijeme. Samozřejmostí je i zdrojový kód tříd, který si můžeme upravit přímo pro naši aplikaci na míru. Veškeré knihovny, které poskytuje projekt Jakata Taglibs jsou ke stažení na http://jakata.apache.org/taglibs/.

7.1 Konfigurace vlastní značky

Úvodem je třeba si říci, že každá značka je vlastně entitou, která popisuje určitý element. Element má určitý formát. Bezprostředně za první lomenou závorkou je název elementu (občas název elementu začíná prefixem: *prefix:název*). Hned za názvem elementu následují názvy a hodnoty atributů elementu.

Jak se zachová JSP kontejner, pokud při překladu stránky narazí na vlastní značku si pro názornost ukážeme na jednoduchém příkladu. Řekněme, že JSP kontejner narazí v dokumentu JSP na značku z výpisu (7.1).

```
<%
<mojetagy:nadpislTag text="Spolujízda"/>
%>
```

*Výpis 7.1: **Vlastní značka v dokumentu JSP***

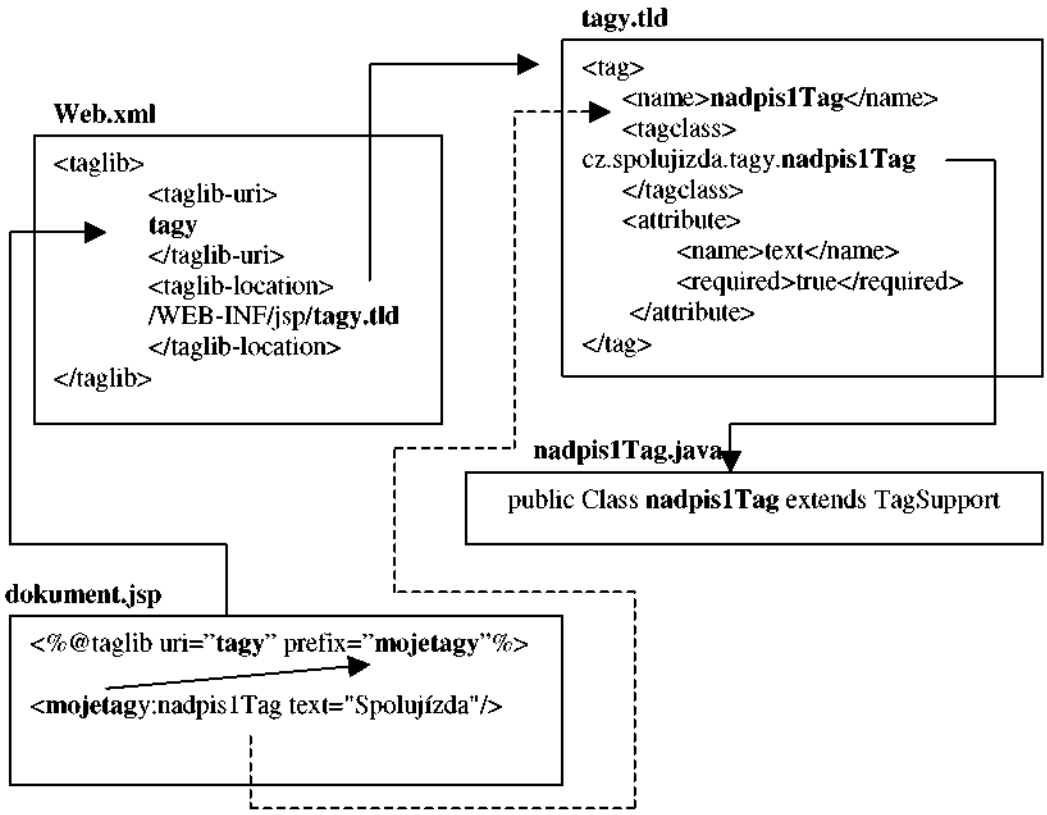
Každý uživatelsky definovaný element je spojen s třídou, která tento element reprezentuje. Tato třída je označována jako obslužná třída elementu. Značka elementu popisuje tento element v příslušném dokumentu JSP.

Samotný proces začínající zjištěním existence vlastní značky a končící sestrojením části odpovědního dokumentu má několik fází:

- Abychom mohli použít vlastní značku z výpisu (7.1), je nutné aby dokument JSP obsahoval direktivu taglib. Tato direktiva musí obsahovat dva parametry: parametr *prefix* s hodnotou *mojetagy* (musí se shodovat s prefixem použité značky) a atribut *uri* s hodnotou *tagy*. Atribut *uri* se odkazuje na element *<taglib-uri>tagy</tag-uri>* jež je součástí definičního souboru *web.xml* daného projektu.
- Element *<taglib-uri>* obsahuje vlastní vnořený element *<taglib-location>*, který ukazuje na příslušný deskriptor knihovny značek (tag library descriptor). V našem případě ukazuje na soubor *tagy.tld*.
- Deskriptor knihovny značek obsahuje elementy, které spojují značku elementu s jeho obslužnou třídou. Dále obsahuje další potřebné a užitečné elementy například pro vytvoření atributů značky atd. V našem případě je

značka elementu spojena se svou obslužnou třídou pomocí elementu *<tagclass>cz.spolujizda.tagy.nadpis1Tag</tagclass>*.

- Takže pokud kontejner JSP narazí na vlastní, uživatelsky definovanou značku, vyhledá podle výše zmíněného postupu obslužnou třídu a v ní nachází příslušné metody (*doStartTag()*, *doInitBody()*, atd.)
- Celý postup je znázorněn na obrázku (3).



Obr. 3: ***Spojení značky s obslužnou třídou***

7.2 Obslužná třída elementu

Jak jsme si již řekli dříve, užití vlastního elementu v dokumentu JSP je vlastně vykonání určitých metod nalezených v instanci obslužné třídy. Abychom mohli tyto metody použít v obslužné třídě, je nutné naimplementovat jedno ze dvou rozhraní z balíčku *javax.servlet.jsp.tagtext*. Těmito rozhraními se rozumí buď rozhraní *Tag* nebo rozhraní *BodyTag*. Při přímém implementování těchto rozhraní je zapotřebí nadefinovat všechny metody, které jsou v těchto rozhraních uvedené.

Definování veškerých metod rozhraní je složité a poměrně zdlouhavé. Balíček *javax.servlet.jsp.tagtex* poskytuje mimo jiné i implicitní implementace rozhraní *Tag* a *BodyTag*. Jsou jimi třídy *TagSupport* a *BodyTagSupport*. Jedná se tedy o předem připravené třídy, které používají programátoři pro tvorbu vlastních implementací rozhraní *Tag* a *BodyTag*. Toho dosáhnou děděním od již hotových implementací *TagSupport* a *BodyTagSupport*. S problémem, zda použít třídu *TagSupport* nebo *BodyTagSupport* se budeme zabývat později.

Vlastní značka by nám byla celkem k ničemu, pokud by nevracela nějakou odpověď. Pro sestavení a odeslání odpovědi do klientského prohlížeče má každý element dvě příležitosti. První příležitost se naskytne při vykonání metody *doStartTag()*. Ta nastává při indikaci počáteční značky elementu. Druhou příležitost dostává při provedení metody *doEndTag()* (při nalezení koncové značky elementu).

Pokud kontejner zpracovává prázdnou značku (značka elementu neobsahuje tělo) je bezprostředně po zpracování metody *doStartTag()* zpracována i metoda *doEndTag()*.

7.2.1 Obslužná třída značky prázdného elementu

V praktické části této bakalářské práce používám vlastní značky, které nemají žádné tělo. Obslužná třída elementu značky *<mojetagy:nadpi1Tag text="Spolujízda">* je znázorněna ve výpisu (7.2).

Ve výpisu (7.2) překrýváme metodu *doStartTag()* z čehož vyplývá že k sestavení odpovědi dochází při vykonání právě této metody. Všimněme si použití metody *pageContext.getOut()*. Pomocí této metody otvíráme výstupní proud do klientského

prohlížeče, a právě metoda *getOut()* vytváří objekt výstupního proudu, který používá užitečnou metodu *print()*.

Návratovou hodnotou metody *doStartTag()* je v tomto případě konstanta SKIP_BODY. Při práci s prázdným elementem je tato konstanta jedinou platnou návratovou hodnotou.

```
package cz.spolujizda.tagy;
import javax.servlet.jsp.tagext.TagSupport;
import java.io.*;
public class nadpis1Tag extends TagSupport {
    private String text="";
    @Override
 public int doStartTag(){
     try{
 pageContext.getOut().print("<h1>"+getText()+"</h1>");
     }
     catch(IOException e){
         e.printStackTrace();
     }
     return SKIP_BODY;
 }
    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }
}
```

Výpis 7.2: ***Obslužná třída elementu nadpis1Tag***

Stejného efektu bychom dosáhli i překrytím metody *doEndTag()*. Metoda *doEndTag()* může vracet dvě hodnoty, které jsou součástí třídy *javax.servlet.jsp.tagtex.Tag*. Pokud metoda vrací hodnotu *EVAL_PAGE*, sděluje tím kontejneru, že má zpracovávat vše co následuje za vlastní značkou elementu. Naopak

při použití návratové hodnoty SKIP_PAGE se kontejner už o nic, co je v dokumentu JSP za vlastní značkou, nestará.

7.3 Atributy vlastních elementů

Již ve výpisu (7.1) jsme si mohli všimnout, že za názvem značky je obsažen i atribut *text*, který má hodnotu "Spolujízda". Skrze atributy máme možnost dynamicky posílat informace obslužné třídě značky elementu. Tyto atributy se v obslužné třídě stávají vlastnostmi a hodnoty jsou těmto vlastnostem přiřazovány podle hodnot získaných z atributů.

Abychom mohli ve výpisu (7.1) použít atribut *text*, je třeba tuto skutečnost sdělit kontejneru JSP. Je tedy potřeba informaci o existenci atributu *text* přidat do deskriptoru elementu. Takže deskriptor daného elementu obohatíme o vnořený element *<attribute>*. Každý element *<attribute>* může obsahovat jeden nebo více vlastních elementů. Jeden povinný vlastní element je element *<name>*.

Ve výpisu (7.3) uvidíme jak vypadá deskriptor elementu *nadpis1Tag*, který obsahuje jeden atribut *text*.

```
<tag>
   <name>nadpis1Tag</name>
    <tagclass>cz.spolujizda.tagy.nadpis1Tag</tagclass>
      <bodycontent>empty</bodycontent>
      <attribute>
        <name>text</name>
        <required>true</required>
       </attribute>
</tag>
```

Výpis 7.3: ***Deskriptor elementu nadpis1Tag***

Z výpisu (7.3) je patrné, že element *<attribute>* v sobě zapouzdřuje několik vlastních elementů. Nejpoužívanější elementy a jejich významy si shrnuty do tabulky (7.1).

<name>	Hodnota elementu odpovídá názvu atributu
<required>	Je-li hodnota tohoto elementu true, pak je atribut povinný
<rtexprvalue>	Je-li hodnota tohoto elementu true, pak je možné k nastavení hodnoty atributu použít výraz JSP

Tab 7.1: ***Elementy atributu a jejich význam***

7.4 Vlastní značka elementu s tělem

Vlastní značka elementu s tělem není v praktické části této bakalářské práce vůbec použita, ale i přesto si tuto kapitolu rozebereme trochu podrobněji.

Pokud vlastní značku elementu obohatíme o tělo, otevírají se nám další možnosti. Tělo elementu je možné úplně vynechat, odeslat ho rovnou do klientského prohlížeče, nebo je možné tělo elementu odeslat objektu obslužné třídy, kde lze toto tělo různě upravovat a pak ho lze opět odeslat do okna klientského prohlížeče.

Pokud se rozhodneme do elementu zahrnout tělo, je třeba o tom „informovat" kontejner JSP. A to lze provést opět skrze deskriptor elementu dané značky. Do deskriptoru elementu značky přidáme element *<bodycontent>*. Podle hodnoty elementu *<bodycontent>* zpracovává kontejner JSP tělo elementu vlastní značky. Hodnoty elementu *<bodycontent>* a jejich význam jsou popsány v tabulce (7.2)

Empty	Sděluje kontejneru JSP, že element vlastní značky neobsahuje žádné tělo
Tagdependent	Sděluje kontejneru JSP, že element vlastní značky obsahuje tělo bez značek JSP
JSP	Sděluje kontejneru JSP, že element vlastní značky obsahuje tělo se značkami JSP

Tab 7.2 : ***Hodnoty elementu <bodycontent> a jejich význam***

Tak tedy čistě z prezentačních důvodů si na výpise (7.4) ukážeme, jak by vypadal deskriptor elementu s tělem, a ve výpisu (7.5) uvidíme element značky *nadpis1Tag* s tělem.

```
<tag>
   <name>nadpis1Tag</name>
    <tagclass>cz.spolujizda.tagy.nadpis1Tag</tagclass>
      <bodycontent>tagdependent</bodycontent>
      <attribute>
        <name>text</name>
        <required>true</required>
      </attribute>
</tag>
```

Výpis 7.4: ***Deskriptor elementu nadpis1Tag s tělem***

```
<%
     <mojetagy:nadpis1Tag text="Spolujízda"/>
          tato značka vytvoří nadpis úrovně 1
%>
```

Výpis 7.5: ***Vlastní značka elementu s tělem***

7.5 Zpracování těla elementu

Zpracovat tělo elementu můžeme dvojím způsobem. Toto zpracování se liší podle toho, zda v obslužné třídě elementu rozšiřujeme třídu *TagSupport* nebo *BodyTagSupport*.

7.5.1 Zpracování těla pomocí TagSupport

Při rozšíření obslužné třídy neprázdného elementu třídou *TagSupport* používáme pro sestavení odpovědi překrytí metody *doStartTag()*. Jediným rozdílem oproti zpracování prázdného elementu je, že namísto návratové hodnoty metody *doStartTag()* SKIP_BODY použijeme návratovou hodnotu EVAL_BODY_INCLUDE. Touto návratovou hodnotou metody *doStartTag()* oznamujeme kontejneru JSP, že má tělo elementu přímo odeslat do klientského webového prohlížeče.

7.5.2 Zpracování těla pomocí BodyTagSupport

Při rozšíření obslužné třídy neprázdného elementu třídou *BodyTagSupport* používáme pro sestavení odpovědi překrytí metody *doAfterBody()*. Rozdíl mezi metodou *doStartTag()* a *doAfterBody* je v tom, že během prohledávání těla elementu se nalezený obsah neodesílá do okna klientského prohlížeče okamžitě. Namísto toho si kontejner tělo elementu přidrží a předá ho ke zpracování obslužnému objektu. K předání těla objektu dochází při vykonání metody *getBodyContent()*.

Pro otevření výstupního proudu nelze použít metodu *pageContext.getOut()*. Pro získání výstupního proudu do klientského prohlížeče musí metoda *doAfterBody()* zavolat metodu *getPreviousOut()*. Touto metodou požádáme o objekt, jenž byl aktivní ještě před zařazením výstupního proudu do zásobníku.

Jako návratovou hodnotu metody *doAfterBody()* můžeme použít i hodnotu SKIP_BODY. Tentokrát se tělo elementu nevynechá, ale je zpracováno kontejnerem JSP pouze jednou.

8. Apache Tomcat

Apache Tomcat je vlastně kontejner JSP, který obsahuje ještě další nástroje, které umožňují vývoj a nasazení webových aplikací.

Tomcat dokáže pracovat ve dvou modech. První mod odpovídá funkci stand-alone http server. Pokud Tomcat běží v tomto modu, znamená to, že zpracovává všechny požadavky na stránku (statické i dynamické požadavky). Druhá možnost spočívá v propojení Tomcatu se standardním http serverem Apache. Pomocí modulu *mod_jserv* nebo *mod_jk* se propojí zpracování požadavků a Apache je poté rozděluje podle potřeby. To má za důsledek, že statické požadavky zpracovává sám http server Apache a dynamické požadavky zpracovává Tomcat. Tomcat totiž běží na jiném portu než http servery. Http servery běží na portu 80, zatímco Tomcat pracuje na portu 8080. Výhodou propojení Apache a Tomcat je zřejmé. Při větší vytíženosti se Tomcat nemusí zabývat statickými požadavky.

Po nainstalování Tomcatu se do proměnné CATALINA_HOME nahraje hodnota odpovídající adrese domovského adresáře Tomcatu. V tomto adresáři se nacházejí další důležité podadresáře. Jejich výčet a význam si popíšeme v tabulce (8.1).

/bin	Obsahuje spustitelné soubory pro start a stop Tomcatu
/common/lib	Obsahuje JAR knihovny pro webové aplikace a interní kód Tomcatu
/conf	Obsahuje konfigurační soubory Tomcatu
/logs	Obsahuje losovací soubory
/shared/lib	Obsahuje JAR knihovny pouze pro webové aplikace
/webapps	Domovský adresář všech webových aplikací

Tab 8.1: ***Důležité adresáře Tomcatu***

9. Praktická část (webová aplikace Spolujízda)

Studenti této univerzity bydlící na kolejích občas samozřejmě dojíždějí domů. Ať už z důvodu stezku po rodinných příslušnících nebo třeba z důvodu mizících zásob čistého prádla. Spousta studentů vlastní automobil, většina ale ne a musí se přepravovat vlakem či autobusem.

Tato skutečnost mě přivedla na nápad, jak situaci vyřešit. Jako praktickou ukázku této bakalářské práce jsem vytvořil webovou aplikaci Spolujízda. Webová aplikace spolujízda je inzertní portál s tématem spolujízdy.

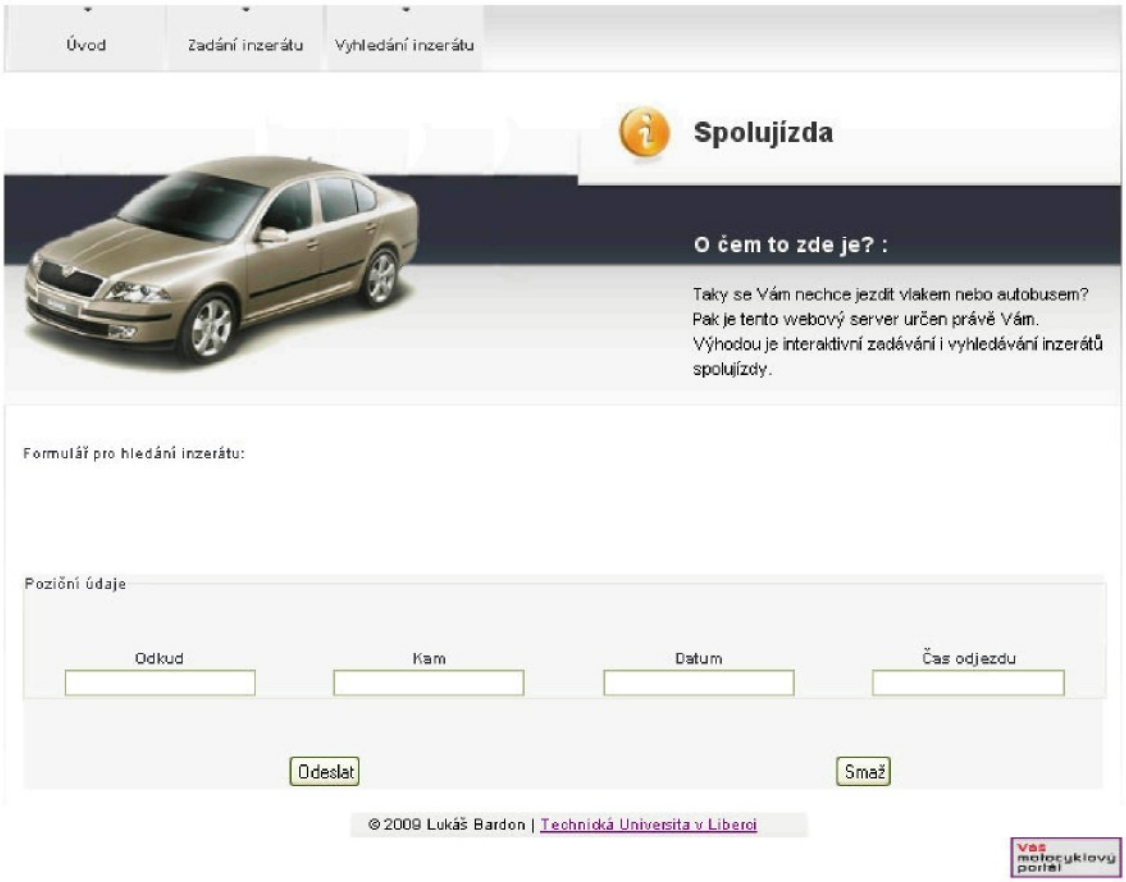
Aplikace tedy slouží k zadávání a vyhledávání inzerátů. Z uživatelského pohledu se aplikace skládá ze tří částí. Úvodní stránka obsahuje prezentační část, ve které se dozvíme k čemu a komu je inzertní portál určen. Dále je na úvodní stránce k vidění tabulka obsahující inzeráty, které byly vloženy v daném aktuálním datu.

Další částí aplikace je formulář pro zadávání inzerátů. Tento formulář obsahuje objekty, do kterých může uživatel zadávat důležité informace o inzerátu (start, cíl, datum odjezdu, čas odjezdu, počet volných míst, icq, telefon, e-mail a poznámky). Po odeslání formuláře dostává uživatel hlášku o úspěšném zařazení inzerátu do databáze.

Poslední částí aplikace je stránka obsahující formulář pro vyhledávání inzerátu. Formulář se skládá opět z objektů, do kterých uživatel zadává informace odpovídající jeho požadavkům (start, cíl, datum odjezdu, čas odjezdu). Po odeslání formuláře dostane uživatel odpověď ve formě tabulky, obsahující výpis požadovaných inzerátů se všemi důležitými informacemi.

Z hlediska vývojáře se aplikace skládá z mnoha dokumentů JSP a tříd starajících se o zpracování formulářů, o komunikaci s databází, o chování vlastních značek atd. V této aplikaci jsem velice často využíval skriptovacích elementů JSP pro implementaci dynamických vlastností stránek. Pro usnadnění čitelnosti kódu a pro případné změny ve formátování odstavců a nadpisů jsem využil technologie Taglibs.

Jelikož je Java multiplatformní a pro mě mnohem stravitelnější než například PHP, byla pro mě technologie JSP velice příjemným a silným nástrojem pro vytváření webové aplikace Spolujízda. Vzhledem k jednoduchosti zadávání a vyhledávání inzerátů je aplikace uživatelsky velice intuitivní.



Obr 9.1: ***Náhled na stránku hledani.jsp***

Obr 9.2: ***Náhled na stránku index.jsp***

10. Závěr

V této bakalářské práci jsem zmapoval možnosti použití programovacího jazyka Java v prostředí dynamických webových aplikaci. Popsal jsem zde rozdíly mezi JSP a servletem, dále jsem přiblížil nejpoužívanější skriptovací elementy JSP.

Výsledkem bylo zjištění, že technologie JSP je velice silným nástrojem pro vytváření dynamických webových aplikací. Její obrovskou výhodou je použití jazyka Java ve skripletech (multiplatformní). Další výhodou jsou implicitní objekty, které technologie JSP poskytuje pro usnadnění psaní dynamických webových aplikací. JSP se tedy hodí pro psaní větších aplikací se složitější backendovou logikou. Naopak nevýhodou JSP je prozatím málo rozšířená možnost hostingu. Další důležitou skutečností je, že psaní servletů je poměrně náročné a nepřehledné. Dle mého názoru je možné psát servlety pouze pro jednoduché internetové stránky prezentačního charakteru. Z těchto důvodů je v technologii JSP servletům přenechává funkce backendové logiky.

V textu je popsána i problematika komunikace JSP s databází. S propojením JSP a Databáze jsem měl asi největší problémy při vytváření praktické části bakalářské práce. Jednalo se především o pochopení *PreparedStatement*.

Dále jsem poměrně rozsáhle popsal vytváření vlastních značek elementu. Objasnil sem k čemu se vlastní značky používají a jak se vytvářejí.

Během zkoumání této problematiky jsem nenarazil na jedinou nevýhodu. Taglibs je velice silným nástrojem technologie JSP. Vytváření vlastních značek nejen že zpřehledňuje kód, usnadňuje komunikaci ve vývojářském týmu, připravuje stránky na strojové zpracování, ale zároveň umožňuje pří jedné změně v obslužné třídě elementu, zároveň změnit všechny objekty v aplikaci reprezentované vlastní značkou příslušného elementu. Taglibs může zastřešovat i aplikační logiku.

V závěrečné části dokumentu jsem napsal pár slov o JSP kontejneru Tomcat a o praktické ukázce této bakalářské práce. Tímto jsem splnil všechny body zadání. Veškeré teoretické podklady tohoto dokumentu jsem využil při vytváření dynamické webové aplikace „Spolujízda".

Seznam použité literatury

[1] Gary Bollinger, Bharathi Natarajan JSP Java Server Pages podrobný průvodce začínajícího tvůrce webu. Grada Publishing, a.s., 2003. Vydání první. ISBN: 80-247-0340-8. 420 str.

[2] Barry Burd JSP: Java Server Pages podrobný průvodce. Computer Press, a.s., 2003. Vydání první. ISBN: 807226804X. 416 str.

[3] Plew Donald R., Stephens Ryan K. Naučte se SQL za 21 dní. Computer Press, a.s., 2004. Vydání první. ISBN: 80-7226-870-8. 573 str.

[4] Vojtěch Patrný. Vlastní tagy v JSP. [online] . [2009-05-14].
URL: http://www.root.cz/clanky/vlastni-lttagy-v-jsp/

[5] David Krch. Tipy a triky pro Oracle XVI. – rychlejší aplikace i bez změn dotazů podruhé. [online] . [2009-05-14]. URL: http://www.dbsvet.cz/