© Kiskapu Kft. Minden jog fenntartva

# JavaServer Pages

Bemutatjuk a JSP-ket, amelyek mind a programozók, mind a laikusok számára lehetővé teszik servletek és dinamikus lapok készítését, Java és HTML kód keverésével.

Előző írásunkban vetettünk egy pillantást a kiszolgálóoldali Javára, és éppen csak lábujjainkat mártva a mélyvízbe, írtunk pár servletet. Ezek olyan Java programok, amelyek dinamikus webtartalmat készítenek. A CGI programok olyan végrehajtható állományok, amelyek nem tartoznak közvetlenül a webkiszolgálóhoz, és minden meghíváskor a legelejüktől kezdve végrehajtódnak. Ezzel szemben a Java servletek a Java virtuális gépben (JVM), a servlettárolóban találhatók, amely közvetlenül csatlakozik a http-kiszolgálóhoz. Amikor a webkiszolgálónak dinamikus tartalomra van szüksége, a servlettárolóhoz fordul.
A Java servlet készítés sok szempontból hasonlít egy mod_perl kezelő írásához: jókora előnyöket nyújt, de megkövetel bizonyos fegyelmezettséget is. Lehangoló lehet olyan servletet írni, amely kilencven százalékig statikus HTML kódot és mindössze tízszázaléknyi Java kódot tartalmaz. Ilyen sokszor meghívni az out.println()-t, már egyenesen őrjítő.
Egyre népszerűbb megoldás erre a gondra a JavaServer Pages azaz a JSP. Ez hasonló szellemiségben készült, mint a Microsoft ASP, vagy akár a nyílt forráskódú PHP nyelv és Mason kiegészítő rendszer. A JSP lehetővé teszi, hogy a Java és a HTML nyelvet számos módon keverjük. Tulajdonképpen a JSP, akárcsak a legtöbb Java program, figyelemreméltóan felületfüggetlen, ez azt jelenti, hogy írhatunk JSP-ket windowsos gépen, amit aztán fejlesztés közben linuxos kiszolgálón alkalmazunk, végül pedig Solarisra telepítjük.
Ebben az írásban gyors pillantást vetünk a JSP-re, ez jó alkalmat teremt arra, hogy egy kis Javát tanuljunk, és különösebb erőfeszítések nélkül, könnyű servletkészítési lehetőséget tárjunk a Java-programozók elé.

## Hogyan működik a JSP

A JSP-k alapötlete igen figyelemreméltó módon egyszerű: tulajdonképpen servletek álruhában. Amikor a JSP-t első ízben hívják meg önműködően servletté alakul. Ez a servlet fordítódik le aztán egy Java .class fájlba, amely ezután a servlettároló belsejéből hajtódik végre. A JSP első meghívásakor kicsit lassabban fognak az adatok a felhasználóhoz jutni, hiszen a színfalak mögött az alábbiakban részletezett folyamatok játszódnak le.
JSP alatt, minden alapvetően statikus tartalomnak tekintendő, ami nincs különleges zárójelek közé helyezve: <% és %>. Az ilyen tagok közé zárt kódrészletet „scriptlet”-nek nevezik a JSP tolvajnyelvén.
A következő HTML-fájl – amit main.jsp-nek neveztünk el – is teljesen megfelel a JSP szabványnak:

```
<HTML>
    <Head>
        <Title>Static JSP Title</Title>
    </Head>

    <Body>
        <P>Static JSP Content</P>
    </Body>
</HTML>
```

Természetesen a fenti JSP meglehetősen unalmas, hiszen kizárólag statikus tartalommal bír. De a JSP-motor egyáltalán nem törődik azzal, hogy egy JSP mennyi dinamikus tartalmat foglal magába; az egész dolgot servletté alakítja, függetlenül attól, hogy mennyire összetett. A fenti JSP esetében az eredményül kapott servlet nem sokkal lesz több, mint egy hosszú karaktersorozat és egy out.println() utasítás egy doGet() eljárás belsejében.
A saját rendszeremen, a fenti HTML kódot a main.jsp fájlba mentettem, a Tomcattel együtt érkező /usr/java/jakarta-tomcat-3.2.1/webapps/examples/jsp/ könyvtár példakönyvtárába.
Ez nyilvánvalóan nem a legmegfelelőbb hely, de ez volt számomra a legegyszerűbb megoldás.
Ha a JSP-t már feltelepítettem, nem is kell többet tennem; a rendszer önműködően fogja servlet forrássá (.java) alakítani, és Java .class fájlba fordítani.
Végre is hajthatjuk és megtekinthetjük az új JSP-nket a Tomcat kiszolgálón keresztül, amely az alapértelmezés szerint a 8080-as kapura figyel ➲ http://localhost:8080/examples/jsp/main.jsp/.
Ha az Apache-t és a mod_jk-t úgy állítottuk be, hogy a továbbítsa a servlet és a JSP kérelmeket a Tomcat kiszolgálóhoz, akkor a main.jsp-t megtekinthetjük a
➲ http://localhost/examples/jsp/main.jsp/ címen is.
Az én gépemen a JSP-rendszer által a main.jsp-hez készített .java és a .class fájlok a következő könyvtárban helyezkednek el: /usr/java/jakarta-tomcat-3.2.1/work/localhost_8080%2Fexamples.
Ha kilistázom a könyvtár tartalmát, a következőket látom:

```
_0002fjsp_0002fmain_0002ejspmain.class
_0002fjsp_0002fmain_0002ejspmain_jsp_0.java
_0002fjsp_0002fmain_0002ejspmain_jsp_1.java
_0002fjsp_0002fmain_0002ejspmain_jsp_2.java
_0002fjsp_0002fmain_0002ejspmain_jsp_3.java
_0002fjsp_0002fmain_0002ejspmain_jsp_4.java
_0002fjsp_0002fmain_0002ejspmain_jsp_5.java
_0002fjsp_0002fmain_0002ejspmain_jsp_6.java
```

Amint látható, jó néhány különféle .java fájl helyezkedik itt el, mindegyik az eredeti JSP egy-egy változata. Akárhányszor csak megváltoztatom a JSP-t, a rendszernek új .java fájlt kell készítenie. A Tomcat alapértelmezés szerint megtartja az előző JSP-alapú servleteket, de egyszerre csak egyetlen .class fájl létezhet, akárcsak ebben az esetben.
A .java és a .class fájlnevek meglehetősen hosszúak és nem arra szánták, hogy a böngészőbe közvetlenül begépeljük őket. A JSP varázslat része ez is. A Tomcat képes arra, hogy az adott URL-lel összerendelt servletet értelmesen és önműködően megtalálja, és elkészítse a Java forrásfájlokat, ha szükséges.
Érdemes egy pillantást vetni a JSP által létrehozott Java forráskódra, hogy képet alkothassunk arról, milyen komoly munka megy végbe a háttérben. Az egyszerű, statikus JSP kódunk servletté alakult, amely több mint száz soros Java forráskódból áll. Ha vesszük a fáradtságot, és visszafejtjük a JSP kódunkat a lefordított servlet forrásból – ami igen nagy kihívás, legalább akkora mintha a Perl HTML::Mason-t próbálnánk visszafejteni – a servletben megjegyzéssorokat találhatunk, amelyek az eredeti JSP sorok sorszámai és a lefordított kód közötti alapvető kapcsolat feltérképezésében nyújthatnak segítséget.

*1. lista* A <% %> tagok használata

```
<HTML>
  <Head>
    <Title>Mini-dynamic JSP Title</Title>
  </Head>
  <Body>

    <P>You are connecting from the host

    <% if (request.getRemoteHost().equals(""))
          { %>

          <%= request.getRemoteHost() %>.</P>
    <% } else { %>

          <%= request.getRemoteAddr() %>.</P>
    <% }  %>

  </Body>
</HTML>
```

*2. lista* Az Include irányelv használata

```
<HTML>
   <Head>
      <Title>Mini-dynamic JSP Title</Title>
   </Head>
   <Body>

      <%@ include file="menubar.jsp" %>

      <P>You are connecting from the host

      <% if (request.getRemoteHost().equals(""))
            { %>
            <%= request.getRemoteHost() %>.</P>
      <% } else { %>
            <%= request.getRemoteAddr() %>.</P>
      <% }  %>

   </Body>
</HTML>
```

© Kiskapu Kft. Minden jog fenntartva

## Dinamikus tartalom

Kicsit érdekesebbé tehetjük a dolgokat, ha egy különleges JSP tagot szúrunk be a main.jsp-be. A tag valamilyen Java kód eredményét szúrja be a felhasználóhoz küldendő kimenetbe:

```
<HTML>
    <Head>
        <Title>Mini-dynamic JSP Title</Title>
    </Head>
    <Body>
        <P>You are connecting from
            <%= request.getRemoteHost() %>.</P>
    </Body>
</HTML>
```

A <%= és a %> jelek közötti kérelem végrehajtódik, és visszatérési értéke a kimenetre kerül. Mivel a JSP tulajdonképpen egy álruhás servlet, minden olyan objektumhoz hozzáférhet, amihez egyébként a rendes servlet elér. Ilyenek például a „request” és a „response”. Figyeljük meg, hogy a <%= és a %> között elhelyezkedő Java kódot nem zárja le pontosvessző! Saját tapasztalatból tudom, hogy igen nehéz leszokni arról, hogy pontosvesszőt tegyünk ide, de a JSP kiakad, ha ragaszkodunk hozzá.

Amennyiben egy vagy több Java műveletet szeretnénk elvégezni anélkül, hogy az eredményt elküldenénk a felhasználó böngészőjére, használjuk az egyszerű <% és %> tagokat. Ezek összevegyíthetők a HTML-lel, ami lehetővé teszi, hogy feltételes szövegek jelenjenek meg a válaszban (lásd az *1. listát*).

Ha a felhasználó gépének neve elérhető, kiírjuk azt, egyébként a gazdagép IP-címét jelenítjük meg. Figyeljük meg, hogyan lehet az if/then/else blokkokat összeelegyíteni a statikus HTML kóddal. A request.getRemoteAddr() hívás csak akkor kerül végrehajtásra, ha a request.getRemoteHost() üres karaktersorozatot adott vissza ("").

Számos JSP irányelvet (directivet) a <%@ és %> tagpárral adunk meg. Minden ilyen irányelv a JSP-ből servletté való alakítás során kerül feldolgozásra. A `directive` kulcsszó közvetlenül a @ jel mögött helyezkedik el, amit aztán nulla vagy több elem követ. Például, tételezzük fel, hogy van egy honlapszerte előforduló menüsorunk, a menubar.jsp nevű JSP fájlban:

```
<table>
<tr>
   <td><a href="one">Option 1</a></td>
</tr>
<tr>
   <td><a href="two">Option 2</a></td>

</tr>
</table>
```

Ezt a részletet könnyen beépíthetjük a dokumentumunkba az „include” irányelv használatával (lásd a *2. listát*).

Fontos azonban tudni, hogy az irányelvek a JSP fordításakor hajtódnak végre és nem futásidőben. Így a fenti példa csak addig fog kifogástalanul működni, amíg meg nem változtatjuk a menubar.jsp fájlt. Mivel a menubar.jsp tartalma már akkor beépült a main.jsp-be mikor servletté alakult, az <%@ include %> tag pedig többé már nem létezik, így aztán, a dolgok nem úgy fognak frissülni, ahogy azt elvárnánk. A megoldás későbbiekben ismertetett, futás idejű JSP művelet használata.

Van még két különleges JSP tag. Az egyik közülük a <%-- --%>, mely megjegyzésként használható. Bár feleslegesnek tűnhet JSP megjegyzések léte, mikor HTML megjegyzések már régóta léteznek, de egy alapvető különbségre mindenképpen szeretném felhívni a figyelmet: a JSP megjegyzéseket a JSP-motor távolítja el, a servlet készítése közben. Ezzel szemben a HTML megjegyzéseket érintetlenül továbbengedi, így azok minden végfelhasználó számára láthatók, ha a „view source” menüpontot választják a böngészőjükben. Mivel lehet választani, többnyire magam is inkább JSP tagokat használok a megjegyzéseimhez, kivéve azokat, amelyek a JSP kódom HTML forrás kimenetének hibakereséséhez használhatók.

Az utolsó JSP-tag a <%! %>. Ennek segítségével a futó servlet számára példányváltozókat (úgynevezett mezőket) vezethetünk be. Habár igen csábító lehetőségnek tűnik, hogy a JSP-ben használt összes változót előre bevezessük, emlékezzünk viszont arra is, hogy a mezők alkalmazása azt is jelenti, hogy magunknak kell ügyelnünk a szálak biztonságos kezelésére is. Mivel a szálkezeléshez többnyire fejfájás társul, ahol lehet, célszerűbb elkerülni alkalmazását.

A <%! %> tagot helyi eljárások meghatározására is lehet használni, ám nem vagyok meggyőződve arról, hogy ez olyan jó ötlet.

© Kiskapu Kft. Minden jog fenntartva

*5. lista* showblog.jsp

```
<%@ page language="java" contentType="text/html"
%>
<%@ page import="java.sql.*" %>

<HTML>
  <Head>
     <Title>Weblog</Title>
  </Head>
  <Body>

  <H1>Weblog</H1>

<%
  // Néhány saját változóbevezetés az
  // adatbázis használatához

  String loginJdbc = "org.postgresql.Driver";
  String loginUser = "reuven";
  String loginPasswd = "";
  String loginUrl = "jdbc:postgresql:
//localhost/atf";

  // A PostgreSQL vezérlő betöltése
  Class.forName(loginJdbc);
  Connection con = DriverManager.getConnection
             (loginUrl, loginUser, loginPasswd);

  // A kifejezés megadása
  Statement statement = con.createStatement();

  // Bizonyosodjunk meg afelől, hogy egy időben
  // csak egy szál kezeli ezt a kifejezést

       String query = "SELECT entry_date,";
       query +=       "entry_headline, ";
       query +=       "       entry_text ";
       query +=       "FROM   BlogEntries ";
       query +=       "ORDER BY entry_date;
       query +=       "DESC;"

       // Lekérdezés végrehajtása
       ResultSet rs =
       statement.executeQuery(query);
%>

<table>

<% while (rs.next()) { %>

  <tr>
     <td> <%= rs.getString("entry_date") %>
          </td>
     <td> <%= rs.getString("entry_headline") %>
          </td>
     <td> <%= rs.getString("entry_text") %></td>
          </tr>

<% } %>

</table>

  </Body>
</HTML>
```

## JSP eljárások

Az irányelvek igen hasznosnak bizonyulhatnak, ha a servlet felépítésének folyamatát szeretnénk befolyásolni. De mi történik, ha a servlet eljárásait futásidőben szeretnénk megváltoztatni?

Természetesen belefoglalhatnánk (include) egy Java kódot, ami végrehajtja a szükséges eljárásokat. A JSP azonban számos különleges tagot tartalmaz, amelyek a servletben Java kódba fordítódnak, így bárki képes a kód megírására, még azok is, akik egyáltalán nem ismerik a Javát.

A JSP eljárástagok tulajdonképpen XML-nek megfelelők és tagkönyvtáraknak nevezett különleges XML dokumentumokban találhatók. Így, habár külsőre HTML-nek tűnnek, valójában nem azok, ami többnyire azt jelenti, hogy különösen nagy figyelmet kell fordítanunk az olyan elemekre, mint a bezáró tagok vagy a perjelek.

A beépített JSP tagkönyvtár számos függvényt is tartalmaz, amelyek közül az egyik gyanúsan hasonlít a második listában megismert include irányelvhez. A 3. lista a main.html (➲www.linuxvilag.hu/magazin/cikkekhez.html) egy olyan változatát mutatja be, amelyik a menubar.jsp irányelv felhozásához a <jsp:include> műveletet alkalmazza.

Az eltérés a két változat között apró, de igen jelentős: a beépítés irányelv ugyanis akkor fejti ki hatását az adott lapra mikor a JSP servletté alakul át, az include művelet ellenben futásidőben hajtódik végre. Ha a menubar.jsp-t megváltoztatjuk a main.jsp két futtatása közt, az irányelvet alkalmazó változat figyelmen kívül hagyná az új menüt, a műveletet használó változat viszont a legutóbbi változatot jeleníti meg. Természetesen mindennek megvan az ára; a <jsp:include> művelet futásidejű kérelmet hajt végre, ami lassabbá és kevésbé hatékonnyá teszi irányelvalapú testvérénél.

Mivel a <jsp:include> által beolvasott kód a legfelső szintű JSP minden kérelemadatához hozzáférhet, lehetőség nyílik dinamikusan változó menürendszerek, igény szerint alakítható rendszerek és adatbázis-elérést megvalósító könyvtárak készítésére.

Vannak más JSP műveletek is. Az egyik közülük a <jsp:forward>, amely a kérelmet egy másik JSP-hez irányítja. Akárcsak a <jsp:include> esetében, ez a művelet is a servletmotorban hajtódik végre, ami azt jelenti, hogy a http-kérelem és a felhasználói kapcsolatadatok továbbra is elérhetők maradnak.

A 4. lista (➲www.linuxvilag.hu/magazin/cikkekhez.html) például, az eredeti JSP-nk egy olyan változatát mutatja be, amelyik másik lapra irányítja át a felhasználót, ha gazdagépének neve nem állapítható meg. Ha a kiszolgálónk esetleg nem tartalmaz no-reverse.jsp nevű lapot, a felhasználók 404-es (a fájl nem található) hibával találkoznának a böngészőjükben. Ugyanakkor a böngészőjük továbbra is az eredetileg kért, main.jsp lapra mutató címet írná ki forrásként. Ez azért van így, mert az átirányítás a motor belsejében megy végbe, külső http-átirányítás nélkül.

## Egyszerű webnaplózó JSP

Előző cikkemben (64. oldal) írtunk pár egyszerű servletet, amelyek segítségével webnaplókat (blogokat) készíthettünk, és nézhettünk meg. Mivel a JSP-k servletekké fordítódnak, semmi sem szól az

ellen, hogy olyan JSP-ket készítsünk, amelyek ugyanazt tudják, mint azok a servletek. Nyilvánvalóan kicsit másképp fognak kinézni, de a hatásuk ugyanaz lesz.
Az *5. lista* JSP-t mutat be (showblog.jsp), amely ugyanazt a feladatot hajtja végre, amit az előző cikkben ShowBlog servlet. Azaz, kiírja a PostgreSQL adatbázistáblában tárolt webnapló tartalmát, a legfrissebbtől a legrégebbi bejegyzésig rendezve.
Már most felhívom rá a figyelmet, hogy a showblog.jsp meglehetősen borzalmas példa arra, hogyan írjunk JSP-ket; a feladata egyszerűen az, hogy bemutassa, mi mindent lehet megvalósítani, és nem törekszik választékos megoldásokra. Ezekkel a dolgokkal a két következő részben fogunk foglalkozni, amikor a JavaBeans és a saját tagkönyvtárak kerülnek terítékre.
Menjünk végig lépésenként ezen a JSP-n, hogy pontosan megvizsgáljuk, miképpen működik. Két „page" irányelvvel kezdünk. Ezek teszik lehetővé számunkra a JSP-nk alapvető beállításának elvégzését, kezdve a lap által visszaadott MIME ContentType fejléc beállításával. Sőt, azt is lehetővé teszi, hogy meghatározzuk, milyen programozási nyelvet keverünk a nem programozott szövegbe. Bár ilyen megoldás egyelőre nem létezik, elméletileg lehetséges olyan JSP-t írni, amely Perlt használ az XML előállításához, vagy Pythont a PNG képek létrehozásához.
Figyeljük meg, hogy a page irányelvben egy vagy több értéket is megnevezhetünk. A showblog.jsp első sora egyszerre állítja be a nyelvet és a ContentType értéket. A második sor azt mutatja, hogy az eredményül kapott servletnek kell bejuttatni majd a java.sql-ben található csomagokat, melyek segítségével JDBC-n keresztül összekapcsolódhatunk a relációs adatbázis-kiszolgálókkal (jelen esetben a PostgreSQL-lel).
Egy rövid bevezető HTML kód után, mélyen elmerülünk a Javában. Készítünk egy SQL kapcsolat-objektumot, és segítségével hozzákapcsolódunk a PostgreSQL kiszolgálónkhoz. Letöltjük az adatokat az adatbázisból, majd sorról sorra végiglépkedünk az eredménykészleten (ResultSet).

## Valami nincs rendben

Amint azt az imént említettem, ez csúnya JSP írási mód, ezenkívül rettenetesen rossz teljesítményt érünk el a rengeteg, ismételten felépített és lebontott adatbázis-kapcsolat miatt, ráadásul még iszonyatos mennyiségű HTML és programkód zagyvalékot is készítettünk. Tulajdonképpen az egyetlen, amit nyertünk a dologgal, hogy a HTML kiírásához nem kell out.println() eljárásokat használnunk.
Továbbá, a JSP-k alapja pont az lenne, hogy a HTML lapokból eltávolítsuk a kódot, és a nem programozók számára is lehetővé tegyük, hogy kevés erőfeszítéssel dinamikus HTML lapokat készítsenek. Ha ilyen sok kódot kell beszúrnunk a dinamikus lapok létrehozásához, elég kicsi az esélye, hogy egy nem programozó rászánja magát a webfejlesztésre.
Ezenkívül, miközben a ShowBlog servletünket JSP-vé alakítottuk, jó néhány kivételkezelő eljárást eltávolítottunk. A servletünk elég értelmes volt ahhoz, hogy a PostgreSQL kiszolgáló eltűnését is kezelje, és erről megfelelő hibajelzést adjon. A JSP-nk, ezzel szemben, mindössze egy hibaüzeneteket tartalmazó backtrace vonalat ad vissza. Ez a nyomvonal hasznos lehet a fejlesztőknek, de egyáltalán nem barátságos vagy hasznos a végfelhasználóknak, akik meglátogatják a lapunkat (az igazság kedvéért el kell mondani, hogy a lap irányelvei közt beállíthatunk egy errorPage értéket, így ezek a hibák egy másik JSP-hez irányíthatók).
A jó megoldás az lenne, ha annyi kódot távolítanánk el a JSP-ből, amennyit csak bírunk, így téve lehetővé a nem programozó számára, hogy hagyományos módon használhassa a kódot, illetve, hogy elválasszuk a programozott és a nem programozott tartalmat.
És valóban, a JSP együtt érkezik a JavaBeans támogatással, ahol minden egyes „bean" (babszem) tulajdonképpen egy számos tagfüggvénnyel rendelkező objektum, amelyet JSP-nkben a különleges <jsp: > művelettel használhatunk ki. A sikeres JSP fejlesztés titka, nem kétséges, a JavaBeans okos felhasználása. Továbbá a JSP lehetővé teszi számunkra, hogy saját műveleteket meghatározó tagkönyvtárakat fejlesszünk ki, így a megfelelő <jsp: > művelethez képest még több kódot tudunk tagokkal helyettesíteni.

## Következtetés

A JavaServer Pages, azaz a JSP, egy új írásmódot von a servletek köré, amit egyébként talán kicsit nehéz lenne egy nem programozónak megtanulnia. Ugyanakkor e hónap legbonyolultabb példája azt is megmutatta, hogy könnyen rossz útra tévedhetünk, ha majdnem annyi kód begépelésével érjük el célunkat, mint amennyi egy hagyományos servlet megírásához kellene. Bár a JSP-kkel általában véve könnyebb dolgozni, a HTML és kód szétválasztásából származó előny semmivé foszlik, ahogy a JSP egyre összetettebbé válik.
A következő hónapban belepillantunk a JavaBeans titkaiba, amely lehetővé teszi számunkra, hogy a sok programozást kiváltsuk a JSP-n kívül meghatározott és karbantartott osztályok segítségével. Ezután megismerkedünk az egyéni JSP tagkönyvtárak készítésének lépéseivel, amelyek lehetővé teszik, hogy saját kis nyelveinket használjuk a JSP-nken belül.

*Reuven M. Lerner* (reuven@lerner.co.il) kisebb, webes és internetes módszerekkel foglalkozó tanácsadó cég tulajdonosa és vezetője. A cikk megjelenésének időpontjában valószínűleg már végleg elkészült Core Perl című könyvével, melyet idén jelentet meg a Prentice-Hall. Az ATF honlapon érhetjük őt el ➲ http://www.lerner.co.il/atf/

### Források

Ahogy a JSP-k egyre népszerűbbekké válnak, újabb és újabb források állnak rendelkezésére annak, aki tanulni szeretne belőlük.

A *Jason Hunter* (O'Reilly and Associates kiadónál napvilágot látott Servlet Programming szerzője) által vezetett ➲ servlets.com nevű honlap sok fontos adatot tartalmaz a servletekről, a legtöbb igen hasznos lehet JSP készítők számára is. Talán még érdekesebb, hogy található itt két cikk a „JPS-k gondjai" témakörben. Bár néhány pontban nem értek egyet, kétségtelenül van ott néhány ötlet és megoldás, amit érdemes megfontolni, különösen, ha nem vagyunk biztosak benne, hogy JSP-ket vagy inkább servleteket használjunk-e.

Nemrégiben jelent meg az O'Reilly and Associates kiadásában *Hans Bergsten* kitűnő, JSP-ről szóló munkája, mely meglepő módon a „JavaServer Pages" címet viseli. A saját példányomat cikkem leadása előtt néhány nappal kaptam meg, és a mű teljesen lenyűgözött mélységével, világosságával és részletességével.

© Kiskapu Kft. Minden jog fenntartva