

JSP elemkönyvtárak

Áttekintés

- ▶ Saját elemkönyvtárak (Custom Tags)
- ▶ JSP standard elemkönyvtár – JSTL
 - ▶ alap elemkönyvtár (core)
 - ▶ nemzetköziesítés (internationalization)

Saját elemkönyvtárak

- ▶ A JSP technológia segítségével könnyen lehet Java kódot HTML dokumentumokba beágyazni.
- ▶ Ez a megoldás nem ideális HTML tartalom fejlesztők számára, akik pl. nem tudnak Java-ul.

- ▶ A JSP technológia biztosítja, hogy elemkönyvtárakba szervezett **saját elemeket** hozzunk létre.
- ▶ A Java-fejlesztő kibővítheti a JSP oldalakat saját elemek írásával és HTML-be ágyazásával.
- ▶ A saját elemek tehát jobb "csomagolást" biztosítanak azáltal, hogy az üzleti logikát elválasztják a megjelenítési logikától, de nem biztosítanak több funkcionalitást, mint a szkriptletek.

A saját elemek–

- ▶ fejlesztő által definált JSP nyelvi elemek, amelyek ismétlődő feladatokat oldanak meg
- ▶ elemkönyvtárakba (tag library) vannak szervezve, amelyek
 - ▶ több (általában összefüggő) elemet definiálhatnak
 - ▶ tartalmazzák az elem implementációját is

Saját elemek előnyei

Saját elemek előnyei

- ▶ **Csökkentik vagy kiküszöbölik a szkriptlet kódot** a JSP-ben. Az elem működése függhet az attribútumok értékétől valamint a törzs (body) tartalmától
- ▶ **Egyszerűbb a szintaxis.** A szkriptletek java kódot tartalmaznak a saját elemek HTML-hez hasonló kódot használnak.
- ▶ **A termelékenységét javíthatják** azáltal, hogy a (programozni nem feltétlenül tudó) tartalom-fejlesztők elemkönyvtárak segítségével dinamikus tartalmat hozhatnak létre Java kód használata nélkül.
- ▶ **Újrafelhasználhatók:** fejlesztési és tesztelési időt takaríthatunk meg a segítségükkel. A szkriptletek nem újrafelhasználhatók (hacsak a copy-paste módszert nem nevezzük annak).

Szintaxis

Szintaxis:

- ▶ törzs nélküli:
`<prefix:tag attr1="value1"...attrN="valueN"/>`
- ▶ törzs tartalommal:
`<prefix:tag attr1="value1"...attrN="valueN">`
 body
`</prefix:tag>`

ahol prefix a könyvtárat jelenti,
a tag az elem azonosítót,
attr1 ... attrN pedig az attribútum nevek, amelyek módosítják az
elem viselkedését.

Saját elem fejlesztésének lépései

Egy saját elem fejlesztésének lépései:

1. Elemkezelő (tag handler) osztály implementálása
2. Elem könyvtár leíró (tag library descriptor) létrehozása
3. Elem használata

1. Saját elem definiálása

Az elem egy Java osztály, amely egy bizonyos interfészt implementál.

Egy elem lehet

- ▶ törzs nélküli
 - ▶ törzset tartalmazó
-
- ▶ Törzs nélküli elem esetében (`bodycontent = empty`) vagy ha a törzset módosítás nélkül használjuk fel vagy egyáltalán nem használjuk fel, azaz ejtjük (`bodycontent = JSP`), a **Tag** interfészt kell implementálni.
 - ▶ Törzset tartalmazó és azt fel is dolgozó elem esetében (`bodycontent = tagdependent` vagy `JSP`) pedig a **BodyTag** interfészt kell implementálni.

Egyszerűbb a **TagSupport** illetve a **BodyTagSupport** absztrakt osztályokat kibővíteni, hogy bizonyos standard metódusokat ne kelljen

Metódusok

set-/get metódusok:

- ▶ **setPageContext**: az elemet implementáló osztály a `javax.servlet.jsp.PageContext` objektumon keresztül hozzáfér a JSP oldalhoz. Ebből az objektumból az összes többi implicit objektum kinyerhető (request, session és application) valamint természetesen az ezekhez rendelt hatókörben eltárolt attribútumok is a `[set|get]Attribute` metódusok által.
- ▶ **get|setParent**: ha az elem beágyazott, a gyerek ezen keresztül fér hozzá a szülő elemhez.
- ▶ minden attribútumhoz tartoznia kell egy azonos nevű propertynek (elég ha **setProperty** metódus van), ezek az attribútumok aktuális értékeit kapják paraméterül. Ha engedélyezzük a TLD-ben a futásidejű paraméterek használatát, akkor az attribútumok értékeit eltároló `setProperty` metódusok paramétere konkrét típus is lehet (int, boolean, stb.), nem csak String.

Megjegyzés:

- ▶ Az első két metódust **TagSupport** illetve **BodyTagSupport** osztály is implementálja.
- ▶ A fenti `setXxx` metódusok meghívódnak rögtön az elemkezelő osztály példányosítása után, így a beállított értékek a `getXxx` metódusok segítségével vagy közvetlenül minden más metódusból elérhetők.

További metódusok:

- ▶ típustól függően (Tag vagy BodyTag), azok a metódusok, amelyeken keresztül a JSP oldalból készült servlet az elemkezelő osztállyal kommunikál (lásd a következőket)

doStartTag:

- ▶ a JSP oldalból készített servletben ez hívódik meg a nyitóelem feldolgozásakor.
- ▶ a metódus törzsében felhasználhatjuk a `pageContext`, a `parent` és az attribútumok értékeit.

Az eljárás visszatérési értéke:

- ▶ Tag interfész implementálása esetén `SKIP_BODY` vagy `EVAL_BODY_INCLUDE` lehet, az előbbi, alapértelmezés szerinti esetben a törzs nem kerül feldolgozásra, az utóbbi esetben viszont igen (de nem az elem hanem a JSP által).
- ▶ `BodyTag` interfész esetén ezeken kívül a törzs feldolgozása esetén `EVAL_BODY_BUFFERED` vagy `EVAL_BODY_AGAIN` (az `EVAL_BODY_TAG` elavult (deprecated))

doEndTag:

- ▶ a záró elem feldolgozásakor hívódik meg.
- ▶ Két lehetséges visszatérési értéke:
EVAL_PAGE, és SKIP_PAGE.
- ▶ Az előbbi esetben, ami egyben az alapértelmezés is, folytatódik az oldal végrehajtása, az utóbbi esetben pedig befejeződik.

release:

- ▶ ez akkor hívódik meg, ha a servlet már végzett az elem feldolgozásával.

Speciális BodyTag metódusok: doInitBody és a doAfterBody

A **doInitBody** egyszer, a törzs első kiértékelése előtt hívódik meg, a **doAfterBody** viszont a törzs minden kiértékelése után meghívódik. (Ha nem volt törzs megadva, vagy a doStartTag metódus SKIP_BODY értéket adott vissza, akkor persze egyik sem hívódik meg.)

Mód van a törzs többszöri kiértékelésére:

- ▶ amennyiben a doAfterBody EVAL_BODY_AGAIN értékkel tér vissza, akkor a törzs újra kiértékelődik,
- ▶ ha SKIP_BODY-t ad vissza, akkor a törzs feldolgozása véget ér. (Az out implicit objektum értéke is visszaállítódik a külső JspWriter-re.)

get|setBodyContent:

- ▶ A JSP oldalból készült servlet, mielőtt végrehajtaná a `BodyTag` típusú elemek törzsét, elmenti az implicit `out` objektum értékét, és egy `BodyContent` osztályú objektummal helyettesíti.
(A `BodyContent` egy végtelen nagy pufferrel rendelkező `JSPWriter`)
- ▶ Ebbe írnak a saját elem törzsében lévő JSP elemek, és ez az, amit aztán a servlet a szülő elemkezelőnek átad.
(A `BodyTagSupport` ezt a `bodyContent` változóban tárolja el.)
- ▶ A `bodyContent` tartalma már a kiértékelés eredménye, tehát nem tartalmaz esetleges saját elemeket vagy más JSP elemeket, hanem ezek kiértékelésének eredményét.

Azt már a saját elemkezelőnk döntheti el, hogy mit tesz a `bodyContent` tartalmával:

- ▶ lekérdezheti (`getReader`, `getString`)

- ▶ módosíthatja

- ▶ kiírhatja a külső `JSPWriter`-re:

```
bodyContent.writeOut(getPreviousOut()) vagy
```

```
bodyContent.writeOut(bodyContent.getEnclosingWriter())
```

- ▶ eldobhatja (`clearBody`)

Szkriptváltozók bevezetése

- ▶ Felmerül az igény arra, hogy valamilyen iterált típus (vektor, lista, sorozat stb.) elemein végiglépkedve egy listát vagy táblázatot jelenítsünk meg.
- ▶ A cél egy olyan saját elem készítése, ami annyiszor hajtja végre a törzsét, ahány feldolgozandó elem van, és a törzsében egy szkriptváltozón keresztül elérhetővé teszi az aktuális elemet.
- ▶ Szükség van tehát valami olyan módszerre, amellyel új szkriptváltozókat definiálhatunk, hasonlóan a standard `jsp:useBean` elemhez.

- ▶ Nehézséget jelent ugyanakkor, hogy a JSP fordítónak már fordítási időben tudnia kell, hogy milyen nevű és típusú változókat akarunk bevezetni, hiszen az erre szolgáló programsorokat el kell helyeznie a JSP oldalból készülő servlet kódjában.
- ▶ Az erre vonatkozó információkat elhelyezhetjük
 - ▶ a TLD `variable` elemében (kevésbé rugalmas, hardkódolt változó nevek), vagy
 - ▶ egy külön osztályban (rugalmasabb, mivel az attribútumok alapján definiálhatjuk a változókat): egy, a `TagExtraInfo`-t kibővítő osztályban, ennek a nevét kell megadni a TLD `teiclass` elemében.

- ▶ lásd pl. `iterate.jsp`, `iterateTei.java`, `iterateTag.java`

- ▶ A származtatáskor a **getVariableInfo** metódust kell felüldefiniálni, a JSP fordító ennek a meghívásával kérdezi le az új változók jellemzőit.
- ▶ Az eljárás paraméterül kapja a TagData típusú objektumot, melyen keresztül elérhetők az elem megadott attribútumainak értékei, amire szükség van, ha például az egyik attribútum értéke adja meg, hogy milyen néven is kell a változót létrehozni.
- ▶ A getVariableInfo visszatérési értéke VariableInfo objektumokat tartalmazó tömb kell legyen, ahol minden egyes VariableInfo egy új változó adatait tartalmazza.

A VariableInfo konstruktorának négy dolgot kell megadni:

- ▶ az új változó nevét
- ▶ osztályát
- ▶ azt, hogy tényleg új-e, vagy csak frissíteni kell az értékét
- ▶ a változó láthatóságát (VariableInfo.NESTED, VariableInfo.AT_BEGIN vagy VariableInfo.AT_END)

Értékadás a változóknak:

- ▶ a pageContext objektumon keresztül a page hatókörben (scope) tároljuk a változó neve mellé a beállítandó értéket:

```
pageContext.setAttribute(valtozoNev, valtozoObjektum,  
PageContext.PAGE_SCOPE);
```

ahol a valtozoObjektum lehet **pl.** iterator.next() azaz egy lista következő elemét teszi be a page hatókörbe.

Egymásbaágyazott elemek készítése

A saját elemek együttműködésének egyik módja, hogy az egyik elem bevezet egy új szkriptváltozót, amit a másik elem felhasznál. Ez nem jelenti feltétlenül az elemek egymásba ágyazását. Közvetlen kapcsolat nincs az elemek között.

(**PI.** az egyik elem kikeresi egy táblázat adatait az adatbázisból, a másik –a listát bejáró elem– pedig a megfelelő formában kiírja azt.)

Az együttműködés másik módja az elemek *egymásbaágyazása*, ami közvetlenebbül fejezi ki az elemek összetartozását.

- ▶ Minden elemkezelő megkapja a JSP oldaltól a szülő elem osztályának referenciáját, ahol szülőnek azt az osztályt nevezzük, aminek a törzsében az elem található.
- ▶ A Support osztályok ezt a referenciát, (ami lehet null is), eltárolják a parent osztályváltozóban.
- ▶ Az információátadás egyoldalú lehet csak, a gyerek meghívhatja a szülő metódusait, a szülő viszont nem tud a törzsében lévő elemekről. (Feldolgozhatja ugyan a törzsét karakterről karakterre, de ez igencsak körülményes ...)
- ▶ Arra sincs mód, hogy két gyerekelem közvetlenül elérje egymást, de a szülőn keresztül már megoszthatják az adataikat.
- ▶ A szülő megkeresésére a TagSupport osztály `findAncestorWithClass` metódusa használható, (a szülő és a gyerek közé további saját elemek ékelődhetnek). Ez a parent értékéből kiindulva addig lépked felfelé a hierarchiában, amíg az adott osztályt meg nem találja.

Példák

- ▶ if.jsp, IFTag.java, IfConditionTag.java, IfThenTag.java, IfElseTag.java:
egymásbaágyazott elemek, melyek a szülő attribútumain keresztül kommunikálnak egymással
- ▶ repeat.jsp, RepeatTag.java, GreetingTag.java:
egymásbaágyazott elemek, melyek nem kommunikálnak egymással

2. Könyvtár leíró létrehozása

Fontosabb attribútumok a teljesség igénye nélkül:

```
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>first</shortname>
  <uri>anAbsoluteUri</uri>
  <info>Egyszeru példakat tartalmazó elemkönyvtár</info>
  <tag>
    <name>hello</name>
    <tagclass>fully_qualified_type</tagclass>
    <teiclass>fully_qualified_type</teiclass>
    <bodycontent>
      empty | tagdependent | JSP
    </bodycontent>
    <info>Köszön</info>
```

```
<attribute>
  <name>attr1</name>
  <required>true|false|yes|no</required>
  <rtexprvalue>true|false|yes|no</rtexprvalue>
  <type>fully_qualified_type</type>
</attribute>
<variable>
  <name-from-attribute>id</name-from-attribute>
  <variable-class>
    fully_qualified_class
  </variable-class>
  <declare>true|false</declare>
  <scope>NESTED | AT_BEGIN | AT_END </scope>
</variable>
</tag>
</taglib>
```


3. Elem használata

Ahhoz, hogy egy saját elemet egy JSP-ben használhassunk, deklarálni kell az elemkönyvtárat: `<%@taglib prefix="tt" uri="URI"%>`

A prefix attribútum

– meghatározza azt a prefixet, amelyik megkülönbözteti egy bizonyos elemkönyvtárban definiált elemeket a más könyvtárban definiált elemektől.

Az uri attribútum

– megadja az URI-t, amely azonosítja az elemkönyvtár leíróját (TLD). A leíró fájlok kiterjesztése `.tld`.

Ezek a TLD fájlok a következő helyeken lehetnek tárolva:

- ▶ a `WEB-INF` katalógusban vagy ennek egy alkatalógusában (főleg a sajátkezűleg írt elemek)
- ▶ JAR-ba csomagolt alkatalógusokban (a mások által előre megírt általánosan használható elemek)

Egy TLD-re direkt vagy indirekt módon hivatkozhatunk:

- ▶ Név szerint (direkt módon):

```
<%@taglib prefix="tlt" uri="/WEB-INF/iterator.tld"%>
```

- ▶ Logikai néven keresztül (indirekt módon):

```
<%@taglib prefix="tlt" uri="/tlt"%>
```

- ▶ A TLD logikai név egy abszolút helyre való map-elését a web-alkalmazás leírójában (deployment descriptor) adhatjuk meg.
- ▶ A /tlt URI-nak a tlds/iterator.tld felel meg.

```
<taglib>  
  <taglib-uri>tlt</taglib-uri>  
  <taglib-location>tlds/iterator.tld </taglib-location>  
</taglib>
```

Az URI lehet abszolút is:

Például a Core JSTL könyvtár abszolút URI-ja a következő:

- ▶ Core: `http://java.sun.com/jsp/jstl/core`
`<%@taglib prefix="tlt"`
`uri="http://java.sun.com/jsp/jstl/core"%>`

Ha az elemkönyvtárra abszolút URI-val hivatkozunk, ami pontosan megfelel a TLD-ben deklarált `taglib` elem URI-jának, akkor a `web.xml`-ben már nem is kell a `taglib` elemet deklarálni, mivel a JSP konténer automatikusan megtalálja a TLD-t, az elemkönyvtár implementációjában (.jar-ban).

JSP Standard elem könyvtár

A JSP Standard elem könyvtár (JSTL) magába foglalja a JSP-k alapfunktionalitásait:

- ▶ ahelyett hogy különböző szolgáltató elemeit keverjük a JSP alkalmazásokban, a JSTL segítségével egy egységes elem-csomagot használunk.
- ▶ ezáltal az alkalmazás bármely JSTL-kompatibilis alkalmazásszerverre telepíthető lesz és az elemek implementációja is optimalizálva van.

A JSTL széles elem-választékot kínál a különböző területekre.

A különböző elem könyvtárak URI-jai:

- ▶ Core: <http://java.sun.com/jsp/jstl/core>
- ▶ Internationalization: <http://java.sun.com/jsp/jstl/fmt>
- ▶ XML: <http://java.sun.com/jsp/jstl/xml>
- ▶ SQL: <http://java.sun.com/jsp/jstl/sql>

- ▶ **Core:** folyamatvezérlés (flow control), URL management, egyéb
- ▶ **l18n:** lokalizálás, üzenet formázás, szám ill. dátum formázás
- ▶ **XML:** alap (core), folyamatvezérlés (flow control), transzformációk,
- ▶ **Adatbázis:** Sql lekérdezések

A JSP-ben a következőképpen hivatkozunk egy elem könyvtárra:

```
<%@taglib  
    uri=http://java.sun.com/jsp/jstl/core prefix="c"  
%>
```

Együtműködés elemek között

Az elemek **implicit** vagy **explicit** módon működnek együtt környezetükkel.

- ▶ *Implicit együttműködés* egy jól meghatározott interfészen keresztül történik, amely által a beágyazott elemek együttműködnek a szülő elemekkel.
Ilyen együttműködést használnak például a JSTL feltételes elemek.
- ▶ *Explicit együttműködésről* beszélünk, ha egy elem információt kínál fel a környezetének EL változó formájában, amelynek a nevét a `var` attribútummal adjuk meg.

PI. a `forEach` elem az aktuális értékét az `item` változóban a következőképpen kínálja fel:

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

Alap (Core) elem könyvtár – Változók használatát támogató elemek

Változók használatát támogató elemek

- ▶ A **set** elem beállít egy változót egy EL kifejezés alapján egy bizonyos hatókörben (oldal, kérés, szesszió, vagy alkalmazás).
- ▶ Ha a változó még nem létezik, akkor létrehozza.

Egy JSP EL változó beállítható:

- ▶ a value attribútummal:
`<c:set var="valtozonev" scope="session" value="..."/>`
- ▶ az elem törzsével:
`<c:set var="valtozonev">`
...
`</c:set>`

A `remove` elem segítségével eltávolítható egy EL változó

```
<c:remove var="valtozonev" scope="session"/>
```

Folyamatvezérlés (Flow Control) elemek

A folyamatvezérlés szkriptekkel nehézkes:

```
<%  
Iterator i = cart.getItems().iterator();  
while (i.hasNext()){  
    ShoppingCartItem item = (ShoppingCartItem)i.next();  
    ...  
}%>  
<tr>  
<td align="right" bgcolor="#ffffff">  
    ${item.quantity}  
</td>  
</tr>  
<% } %>
```

A folyamatvezérlés elemek kiküszöbölik a szkripteket.

Feltétel elemek

Az **if** elem a **test** attribútumában található kifejezés értelmezésének eredményeképpen a törzsében levő tartalmat kiértékeli vagy sem.

```
<c:if test="{empty param.Add}">  
...  
</c:if>
```

- ▶ A **choose** elem feltételes blokkokat hajt végre a beágyazott **when** elemek által.
- ▶ Az első olyan **when** elem törzsét értelmezi, melynek a feltétele true.
- ▶ Ha egyik törzs feltétel sem true, akkor az **otherwise** elem törzse értékelődik ki (ha van ilyen).

Pl.:

```
<c:choose>
<c:when test="\${customer.category =='proba'}">
...
</c:when>
<c:when test="\${customer.category =='ismeros'}">
...
</c:when>
<c:when test="\${customer.category =='kedvenc'}">
...
</c:when>
<c:otherwise>
...
</c:otherwise>
</c:choose>
```

az if-then-else a következőképpen szimulálható:

```
<c:choose>  
<c:when test="{count == 0}">  
Nincs egy találat sem.  
</c:when>  
<c:otherwise>  
A találatok száma:  {count}.  
</c:otherwise>  
</c:choose>
```

Elemek kollektciók bejárására

A `forEach` elem segítségével egy objektumkollektiót járhatunk be.

Többek között a következő típusú kollektiók járhatók be:

- ▶ `java.util.Collection`,
- ▶ `java.util.Map` implementációi (a `var` változó `java.util.Map.Entry` objektum lesz),
- ▶ tömbök (objektum vagy primitív elemek, a primitív elemek wrapper osztályokká lesznek átalakítva),
- ▶ `Iterator`,
- ▶ `Enumeration` implementációk,
- ▶ karaktersor (`String`) mely vesszővel elválasztott értékeket tartalmaz. (pl.: Monday, Tuesday, Wednesday, Thursday, Friday).

Az előbbi példa így alakul:

```
<c:forEach var="item" items="{sessionScope.cart.items}">
...
<tr>
<td align="right" bgcolor="#ffffff">
${item.quantity}
</td>
</tr>
</c:forEach>
```

URL elemek

Az **import** elem segítségével elérhetünk URL alapú erőforrásokat, melyeknek tartalma befűzhető vagy feldolgozható a JSP-ben.

- ▶ **PI.:** beolvasunk egy XML dokumentumot, aminek a tartalmát hozzárendeljük az xml nevű változóhoz, majd ezt felhasználjuk más elemekben:

```
<c:import url="/books.xml" var="xml"/>  
<x:parse doc="${xml}" var="booklist"  
scope="application"/>
```

- ▶ A szessziókövetésnél szó volt róla, hogy egy alkalmazás át kell írja az URL-eket arra az esetre, ha a felhasználó kikapcsolja a sütitet (cookie) (`response.encodeURL(...)`).
- ▶ Az `url` elem segítségével a JSP-ben található URL-k átíródnak.
- ▶ Ez az elem csak akkor fűzi hozzá a szesszió ID-t az URL-hez, ha a sütik ki vannak kapcsolva, egyébként nem módosítja az URL-t.

Pl.:

```
<c:url var="url" value="/catalog">  
<c:param name="Add" value="{bookId}"/>  
</c:url>  
<p><strong><a href="{url}">...</a>
```

- ▶ A **param** elem a `jsp:param` elemhez hasonlóan a többi URL elem törzsében használható és kérés (request) paramétereket specifikál.
- ▶ A **redirect** elem egy HTTP átirányítást végez el.

Egyéb elemek

- ▶ A **catch** elem a JSP hibakezelés mellé egy kiegészítő mechanizmust ad.
- ▶ A fontos műveleteket nem kell egy `catch` elembe beágyazni, hanem hagyni kell, hogy továbbmenjenek a hibaoldalra.
- ▶ Kevésbé fontos hibákat egy `catch` elembe kell beágyazni, ezáltal a hiba nem jut el a hibaoldalig, hanem hamarabb le lesz kezelve.
- ▶ Az elkapott hiba a `var` változóban lesz tárolva, melynek hatóköre mindig az aktuális oldal.
- ▶ Ha a `var` nincs specifikálva, a hiba el lesz kapva, de nem lesz eltárolva egy változóban.

Az **out** elem kiértékel egy kifejezést, majd az eredményt az aktuális `JspWriter` objektumba teszi:

```
<c:out value="value" [escapeXml="true|false"] [default=""]  
/>
```


Nemzetközivé tételt elősegítő (internationalization) elem könyvtár

JSTL elemeket határoz meg egy oldal nyelvspecifikus beállítására, nyelvspecifikus üzenetek létrehozására, számok, pénznemek, dátumok, idők nyelfüggő formázására és beolvasására.

- ▶ A JSTL i18n elemek egy nyelvfüggő kontextust használnak, hogy a megfelelő adatot elérjék.
- ▶ Egy ilyen kontextus egy `Locale` és egy `ResourceBundle` instanciából áll.
- ▶ Mikor egy kérés jön, a JSTL automatikusan beállítja a `locale`-t a kérés fejléce alapján és kiválasztja a helyes erőforrásfájlt felhasználva a JSTL elem paramétereiként megadott alapnevet.

A Locale beállítása

- ▶ A **setLocale** elem a kliens által a böngészőben specifikált locale felülírására használható.
- ▶ A **requestEncoding** elem segítségével beállíthatjuk a kérés objektum karakter kódolását (character encoding), hogy helyesen dekódoljuk azon kérés paramétereket, melyek kódolása nem ISO-8859-1

Üzenetet megjelenítő (messaging) elemek

- ▶ Alapértelmezés szerint a JSTL úgy van beállítva, hogy érzékelje a böngésző nyelvbeállításait.
- ▶ Ez azt jelenti, hogy a kliens a böngészőbeállítások segítségével meghatározza, hogy melyik nyelvet akarja használni.

A `setBundle` és `bundle` elemek

Az **`fmt:setBundle`** és **`fmt:bundle`** elemek segítségével beállíthatók a nyelvhez kötött erőforrásfájlok.

- ▶ Az **`fmt:setBundle`** az erőforrásfájlt egy hatókörhöz rendelt változóba menti le.
- ▶ Az **`fmt:bundle`** egy adott elem törzsében használandó erőforrásfájlt állít be.

A message elem

A message elem nyelvfüggő üzenetek megjelenítésére használható.

Pl.

```
<fmt:message key="Choose"/>
```

param elemekkel az üzenetnek további paramétereket adhatunk meg.

Formázó elemek

A JSTL több elemet biztosít nyelvfügő adatok (számok, dátumok) feldolgozására és formázására.

Ezek a következők:

- ▶ `formatNumber`
- ▶ `formatDate`
- ▶ `parseDate`
- ▶ `parseNumber`
- ▶ `setTimeZone`
- ▶ `timeZone`

Pl.

```
<fmt:formatNumber value="\${book.price}" type="currency"/>
```

```
<jsp:useBean id="now" class="java.util.Date"/>
```

```
<jsp:setProperty name="now" property="time"  
value="\${now.time +432000000}"/>
```

```
<fmt:message key="ShipDate"/>
```

```
<fmt:formatDate value="\${now}" type="date"  
dateStyle="full"/>
```