ANTAL MARGIT

JAVA ALAPÚ WEBTECHNOLÓGIÁK

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MŰSZAKI ÉS HUMÁNTUDOMÁNYOK KAR, MAROSVÁSÁRHELY
METEMATIKA–INFORMATIKA TANSZÉK

ANTAL MARGIT

JAVA ALAPÚ WEBTECHNOLÓGIÁK

Scientia Kiadó
Kolozsvár·2009

A kiadvány megjelenését támogatta:

SAPIENTIA
ALAPÍTVÁNY

Lektor:
Ruff Laura (Kolozsvár)

Sorozatborító:
Miklósi Dénes

Descrierea CIP a Bibliotecii Naționale a României
ANTAL MARGIT
Java alapú webtechnológiák / Antal Margit. – Cluj-Napoca: Scientia, 2009.
Bibliogr.
ISBN 978-973-1970-21-9

004.43 Java

TARTALOM

CONTENTS

CUPRINS

ELŐSZÓ

Napjainkban a webtechnológiák a legerőteljesebben fejlődő technológiák közé tartoznak. Nagyon rövid idő alatt ezek a technológiák megváltoztatták nemcsak a szoftveripart, hanem az emberi gondolkodás- és viselkedésmódot is. Ennek a tananyagnak az a célja, hogy bemutassa a Java technológiák segítségével történő webfejlesztést. Hangsúlyosan szerveroldali fejlesztésről van szó, tehát feltételezzük, hogy az olvasó jártas az alapvető ügyféloldali technológiákban. Összesítve, a tananyag megértéséhez a következő előismeretekre van szükség: a Java programozási nyelv ismerete, HTML, HTTP alapismeretek, adatbázis-kezelési alapismeretek.

A tananyag a „learning by doing" filozófiát követi. A második fejezettől kezdődően végig ugyanazon a webalkalmazáson dolgozunk, amely az utolsó fejezet után válik teljessé. Az alkalmazásfejlesztésnél arra törekedtünk, hogy már a legelejétől betartsuk a Model–View–Controller architektúrát és a komponenseket ennek alapján tervezzük meg. Ez lehetővé teszi, hogy az alkalmazásban a funkciókat jól elkülönítsük, így megnövekszik a komponenseinek újrafelhasználhatósága is.

A fejezetek végén két típusú feladattal találkozhatunk: ezek a tesztkérdések és programozási feladatok. A tesztkérdések célja az elméleti ismeretek elmélyítése, itt lehetőség van a megoldások ellenőrzésére is: a helyes válaszokat a C. függelék tartalmazza. A programozási feladatok nagy része megoldott feladat, amelyek újabb komponensek hozzáadását, illetve már létezők módosítását szemléltetik az adott fejezet nyújtotta ismeretek segítségével.

A programozási feladatok megvalósításához ajánlott egy integrált fejlesztői környezet (IDE) használata. Bár a tananyagban a komponensek megadásánál ezek telepítésleíróban történő konfigurációját is megadtuk, ezt a munkát elvégezheti helyettünk egy integrált fejlesztői környezet is. Ebben a tananyagban a NetBeans fejlesztői környezet használatát javasoljuk, esetenként erre hivatkozunk is. A telepítés úgy történik, hogy először a JDK legutolsó változatát töltjük le, majd ennek telepítése után ajánlott a NetBeans telepítése. Ajánlott a legutolsó NetBeans változatot letölteni. A 6.5 változat például tartalmazza a Tomcat webkonténert, a

Glassfish alkalmazásszervert, illetve a Derby adatbázis-kezelőt is. A tananyaghoz szükséges szoftverek a világhálón megtalálhatók és szabadon letölthetők.

2009. július 14. *A szerző*

1. FEJEZET

WEBALKALMAZÁSOK FEJLESZTÉSE JAVA TECHNOLÓGIÁKKAL

1.1. Webalkalmazás technológiák

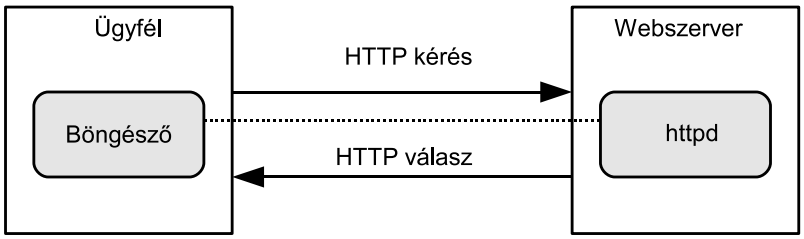
HTTP

A HTTP (Hypertext Transfer Protocol) protokoll egy kliens-szerver technológiát támogató protokoll. Ez a protokoll abban különbözik más internetes protokolloktól, hogy állapotmentes. Az állapotmentesség azt jelenti, hogy minden egyes kapcsolat csak egy kérés végrehajtását engedélyezi, ellentétben más internetes protokollal (pl. SMTP, FTP), amelyek esetében egy kapcsolat több kérés–válasz lebonyolítására is alkalmas. Például ha fájlműveleteket akarunk végrehajtani egy FTP szerverrel, akkor a kapcsolat megteremtése után korlátlan mennyiségű feltöltést, illetve letöltést végezhetünk. A HTTP protokoll állapotmentességének egyik következménye a nagyfokú hatékonyság. Bizonyos alkalmazások esetében az állapotmentesség egy hátrányos tulajdonság, amelyet majd a munkamenetekkel lehet megoldani.

HTML

A HTML (Hypertext Markup Language) egy dokumentummegjelenítő nyelv, amely lehetővé teszi a dokumentumok összekapcsolását hiperkapcsolatok segítségével. A nyelv lehetővé teszi különböző médiaelemek beágyazását a dokumentumba. A HTTP lehetővé teszi bármilyen MIME (Multipurpose Internet Mail Extensions) formátumú állomány átvitelét.

A HTTP és a HTML együttesen alkotják a WWW (World Wide Web) alaptechnológiáit.



1.1. ábra. *HTTP kliens-szerver architektúra [7]*

HTTP kliens-szerver architektúra

Minden egyes HTTP protokollt használó kommunikáció esetében van egy kérés és egy válasz, ahogy ezt az 1.1. ábra is mutatja. Az ügyfél böngészője egyetlen kérést intéz a szerverhez, amely alapján a szerver meghatározza a kért erőforrást és válaszul visszaküldi a kért állomány tartalmát. Ezt követően az ügyfél böngészője értelmezi a választ és megjeleníti a kapott tartalmat. A kérés az igényelt erőforráson kívül más információkat is tartalmaz. Ilyen információ például az ügyfél böngészőjének a típusa. A kérés általában sima szöveges információ, a válasz pedig lehet szöveg, illetve bináris adat is.

Webhely

HTML lapok és más médiafájlok gyűjteménye, amelyek egy webszerveren helyezkednek el és láthatóak az ügyfelek számára.

URL – Uniform Resource Locator

Az URI (Uniform Resource Identifier, egységes erőforrás-azonosító), egy olyan karaktersorozat, amely azonosít egy erőforrást. Az URI az erőforrást kétféleképpen azonosíthatja, hely (URL – Uniform Resource Locator) vagy név (URN – Uniform Resource Name) szerint. Az URL-t webcímnek is nevezzük, és a következő részekből tevődik össze:

```
protocol://host:port/path/file
```

Példa:

```
http://www.ms.sapientia.ro:80/~manyi/index_oop.html

protocol (protokoll) : http
host (gazdagép) : www.ms.sapientia.ro
port : 80
path (útvonal): ~manyi
file (erőforrás): index_oop.html
```

Amennyiben standard porton fut a webszerver (jelen esetben 80), akkor a port hiányozhat az URL-ből.

1.2. Java EE 5 platform

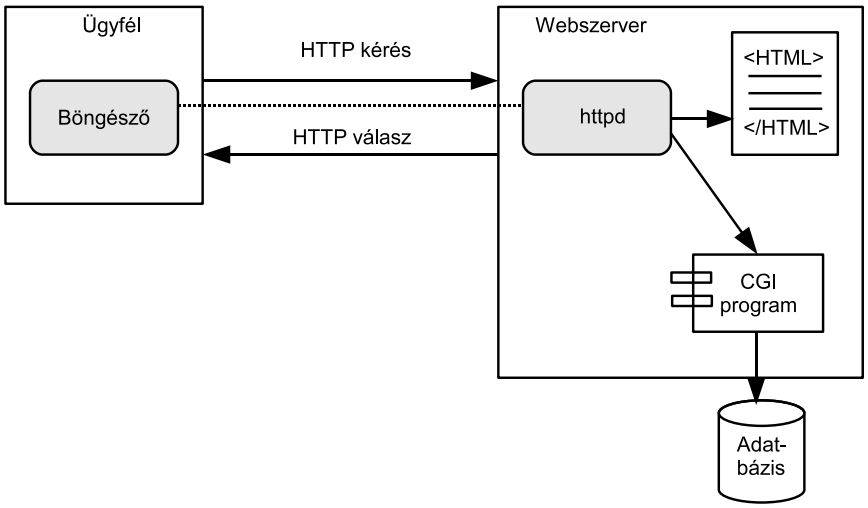
A Java EE (Enterprise Edition) egy ipari szabvány hordozható, bővíthető, skálázható és biztonságos szerveroldali alkalmazások készítéséhez. A Java SE (Standard Edition) platformra épül és magába foglal egy alkalmazásszervert is. Ez az alkalmazásszerver egyben webkonténer is, azaz képes szervletek végrehajtására.

1.3. Webalkalmazások végrehajtása

Webhely = Statikus állományok gyűjteménye, amely HTML, kép és más formátumú állományokat tartalmaz.

Webalkalmazás = webhely + szerveroldali programok

Az alapvető különbség a webhely és a webalkalmazás között az, hogy az előbbi csak statikus tartalmat szolgáltat, az utóbbi pedig dinamikus tartalom előállítására is képes. A dinamikus tartalom előállítását szerveroldalon futó programok végzik. Ezeket különböző programozási nyelvekben lehet elkészíteni, leggyakoribbak a Perlben megírt szkriptek, de használható akár a C/C++ nyelv is. A szerveroldali programot a webszerver külön folyamatként vagy folyamatszálként lefuttatja minden egyes kérésre, a program eredményét pedig visszaküldi a kérést indítványozó böngészőnek.



1.2. ábra. *Webszerver és CGI program [7]*

CGI programok

A CGI (Common Gateway Interface) az első mechanizmus, amely lehetővé tette szerveroldali program végrehajtását. A webszerver a programot külön burokban futtatja. Ez azt jelenti, hogy ahány kérés, annyi külön burokban lefuttatott CGI program (l. 1.2. ábra). Nyilván ennek a módszernek megvannak az előnyei, illetve a hátrányai is. Kezdjük a jobbikkal, éspedig az előnyökkel:

- A CGI program különböző programozási nyelvekben készíthető, bár a Perl a legelterjedtebb.
- Hibás CGI programtól nem fagy le a webszerver, hiszen a CGI program külön burokban fut.
- Nagyon egyszerű a hivatkozás egy weboldal keretén belül a CGI programra, egyszerűen egy sorban megadható a hívás.
- Nincsenek konkurencia problémák, hiszen minden kérés kiszolgálását egy külön burokban lefuttatott CGI program végzi.
- Minden webkiszolgáló támogatja a CGI programokat.

És most folytassuk a hátrányokkal:

- Hosszú válaszidő, ami a külön burok létrehozásának tulajdonítható.
- Nem skálázható, hiszen az egy gépen létrehozható különböző folyamatok száma korlátozott.
- Azok a programozási nyelvek, amelyekben a CGI programokat készítik, nem biztonságosak.
- CGI programban nehéz elválasztani a megjelenítést az üzleti logikától.
- A szkript nyelvek esetenként platformfüggőek.

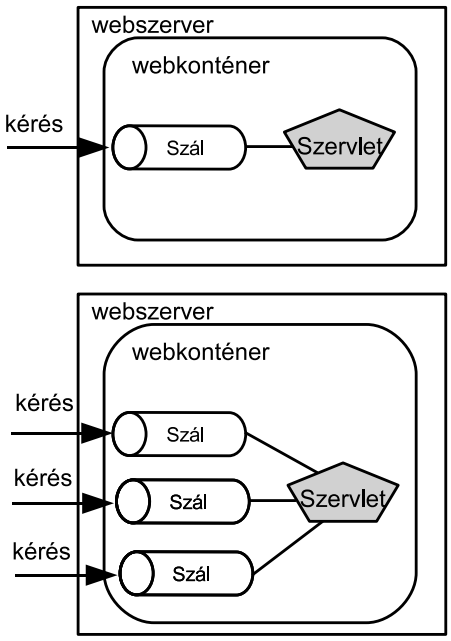
1.4. Szervletek

A CGI hátrányainak elkerülése végett született meg a szervlet technológia. A legfontosabb különbség a CGI-hez képest az, hogy a szervlet futtatása nem külön burokban (azaz külön folyamatként) történik, hanem csak külön szálon, amelyet a webszerver hoz létre minden egyes kérésre (l. 1.3. ábra). Mivel a szál létrehozása gyorsabb, mint a folyamaté, ezért ez a fajta kiszolgálási mód gyorsabb. Most pedig tekintsük át a szervletek előnyeit és hátrányait.

A szervlet technológia előnyei:

- Gyorsaság. Minden kérés kiszolgálása külön szálon fut, amely gyorsabb, mint a CGI külön folyamatai.
- Hibatűrés, objektumorientáltság. A szervletek Java nyelven íródnak, így használható a Java nyelv összes adottsága, mint például a kivételkezelés, amely a hibatűrő tulajdonság alapvető nyelvi eszköze.
- Skálázhatóság. Megnövekedett terhelést is kezelni tud, anélkül, hogy a felhasználók jelentős válaszidő növekedést tapasztalnának (több gépből álló klaszter kialakítását támogató technológia).
- Platformfüggetlenség. A Java nyelv biztosítja.
- Naplózás. A szervleteknek lehetőségük van naplózási műveleteket végezni.
- A webkonténer által nyújtott szolgáltatások igénybe vétele. Ide tartozik a hibakezelés és a biztonság.

A szervlet technológia hátrányai:



1.3. ábra. *Webszerver és szervletek [7]*

- Ha csak szervleteket használunk a webalkalmazás felépítésére, akkor itt is keveredik a megjelenítést végző kód az üzleti logikával. Keretrendszerek használatával a probléma kiküszöbölhető (pl. Struts, JSF).
- Konkurencia problémák. Ugyanaz a szervlet párhuzamosan hajtódhat végre különböző szálakon, több egyidejű kérés kiszolgálása esetén. Ilyen esetben bizonyos változókhoz való hozzáférést szinkronizálni kell.

A Java szervleteket a webkonténer futtatja, amelyet még szervletmotornak is nevezünk. A webkonténer tulajdonképpen egy Java virtuális gép (JVM), amelyet egy Java Servlet API egészít ki. A Java szervlet egy webkomponens, amelynek életciklusát a webkonténer kezeli. A webkonténer működhet a webszerver részeként, illetve lehet egy különálló HTTP szolgáltatás.

A Java szervletek a legalapvetőbb webkomponensek, az összes többi Java webkomponens erre épül. Egy szervlet, ahogyan a neve is mutatja,

alapvetően kiszolgálói feladatot lát el. Egy kérés feldolgozása során előállítja az ennek megfelelő választ, tehát olyan Java program, amely HTML kódot állít elő. Mivel karbantartása programozási ismereteket igényel, egyszerű HTML ismeretekkel rendelkező személy nehezen boldogul el a szervletekkel. Pontosan ezért dolgozták ki a JSP technológiát, amelynek használata egyszerűbb, és programozási ismeretekkel nem rendelkező személyek is könnyen elboldogulnak a készítéssel, illetve karbantartással.

1.5. JSP technológia

Egy JSP lap, ellentétben a szervletekkel, HTML oldalba beágyazott Java kódot tartalmaz. Egy korszerű JSP lap pedig a Java kódot beágyazott elemek formájában tartalmazza, így maga a Java kód rejtve maradhat a felhasználó elől. A JSP technológia is a szervlet technológiára épül. A webkonténer minden egyes JSP lapból egy szervletet állít elő, így a kérés kiszolgálását ismét Java kód végzi. Ennek következtében JSP lapokkal is minden olyan problémát meg lehet oldani, amely szervletekkel megoldható.

A HTML lapba beágyazott kód szép példái a nem Java alapú szkript nyelvek, mint például a PHP (Hypertext Preprocessor), ASP (Active Server Pages), illetve a Ruby on Rails. Most pedig bevezetésként tekinsünk egy nagyon rövid példát nem túl korszerű JSP lapra. Ennek a JSP lapnak az a feladata, hogy megjelenítse az első 10 természetes számot és ezek négyzeteit.

```
1 <html>
2    <body>
3       <% for(int i=0; i<10; ++i ){ %>
4       <p>
5          <%=i%>   <%=(i*i)%>
6       </p>
7       <%}%>
8    </body>
9 </html>
```

A fenti kódrészlet összesen négy darab szkript kódot tartalmaz, egyet a 3. sorban, kettőt az 5. sorban és végül még egyet a 7. sorban. A 3. sor a for ciklus kezdősora, a 7. sor pedig a törzsrész bezárását végző sor.

A JSP lapoknak az a nagy előnyük a szervletekhez képest, hogy egyszerű a szintaxisuk és, amennyiben nem használnak szkripteket, elkészítésüket olyan személy is végezheti, aki minimális programozási ismerettel rendelkezik. Egy modern Java webalkalmazás mind szervleteket, mind pedig JSP lapokat használ. Az előbbiek leggyakrabban vezérlési funkciókat látnak el, míg az utóbbiak a megjelenítésre fókuszálnak. Így kölcsönösen kiegészítik egymást.

1.6. Háromrétegű alkalmazások

A háromrétegű architektúrát általában vízszintesen szokták ábrázolni, és a következő rétegeket tartalmazza:

- ügyfélréteg,
- üzleti logika réteg,
- adatréteg (perzisztens réteg).

A háromrétegű alkalmazások nagy előnye, hogy elválasztják az ügyfélréteget az adatrétegtől, az ügyfél csak az üzleti rétegen keresztül tud kommunikálni az adatréteggel. Ennek következménye, hogy az alkalmazás rugalmassá válik. Így például ha módosítjuk az adatréteget, akkor elégséges az üzleti réteg aktualizálása, az ügyfélréteget nem kell módosítani. Egy másik előny, hogy a meglévő üzleti réteget felhasználhatjuk egy új alkalmazás elkészítéséhez, ezáltal megvalósítva a komponensek újrafelhasználását.

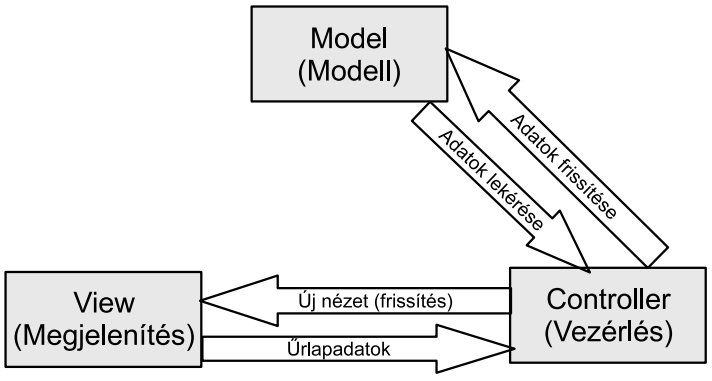
1.7. MVC architektúra

Elsőként a Xerox írta le a 1980-as évek végén, azóta más GUI alkalmazások készítésére alkalmas környezet is átvette ezt a modellt. Az alapgondolat az, hogy egy alkalmazás adatait, ezek megjelenítését, illetve ezek kapcsolatát három önálló egységre bontja szét, amelyek nevei: Model (modell), View (megjelenítés), illetve Controller (vezérlés). A szétbontás célja az, hogy rugalmas alkalmazásokat készíthessünk, például kicserélhessük az adatok megjelenítését a többi rész változtatása nélkül.

Egyre szélesebb körben terjednek el a mobiltelefonok, PDA-k, amelyek internetkapcsolat segítségével alkalmassá váltak tartalom megjelenítésére [1]. Csakhogy ezek mérete különböző, így ugyanazon tartalmat

más és más formában kell megjeleníteni. Gyakorlatilag úgy kell elkészíteni az alkalmazást, hogy ez képes legyen a kommunikációs eszköznek megfelelő tartalmat előállítani. Ennek megvalósítása szükségessé tette az alkalmazásoknak három részre bontását: modell (Model), megjelenítés (View) és vezérlés (Controller). A modell részt rendszerint az adatok és ezek elérési logikája képezi. Ez a rész az, amely a legritkábban változik. A vezérlés a közvetítő a modell és a megjelenítés között, amely vagy aktulizálja a modellt, vagy pedig egy új nézetet jelenít meg. A megjelenítés pedig sokféle lehet, ahogyan azt az előzőekben is szemléltettük. Ha időközben ugyanazon alkalmazáshoz egy új kommunikációs eszközt akarunk csatlakoztatni, akkor csak az ennek megfelelő megjelenítési réteggel kell kiegészíteni az alkalmazást. Tehát ennek a tervezési mintának a betartása is nagymértékben hozzájárul az alkalmazások rugalmasságának növeléséhez.

Az 1.4. ábra egy űrlap kitöltésének MVC architektúráját szemlélteti, amely során az űrlapadatok a megjelenítési rétegből átkerülnek a vezérlési réteghez, ahol megtörténik az adatok ellenőrzése. Hibás adatok esetén a vezérlés új megjelenítést generál, amely figyelmezteti a felhasználót a hibákra. Ha helyesek voltak az adatok, eltárolódnak a modell rétegben. Egy másik lehetséges forgatókönyv az, amikor az űrlap bizonyos adatok lekérdezését valósítja meg. Ebben az esetben a vezérlés feladata az adatok lekérése a modell rétegből és az ezt tartalmazó új nézetet előállítása és eljuttatása az ügyfélhez.



1.4. ábra. *Az MVC architektúra*

Az MVC tervezési minta először grafikus interfészek (desktop alkalmazások) készítésével kapcsolatban jelent meg, később alkalmazták

például webalkalmazások architektúrájaként is. Webalkalmazások esetében a vezérlési réteg feladata a kérések feldolgozása és továbbítása a megfelelő komponens fele. Így a vezérlési rész felügyeli az alkalmazást, a modell pedig az alkalmazás állapotát tárolja.

Összefoglalva, a következő előnyökkel rendelkezik egy MVC architektúrájú alkalmazás:

- Lehetővé teszi több, különböző megjelenítés illesztését ugyanazon modellhez.
- Szétválasztja az alkalmazás üzleti logika, adatelérés, illetve megjelenítés részeit, amely növeli az alkalmazás rugalmasságát.

Modell 2 architektúra

Egy Java alkalmazást modell 2 architektúrájúnak nevezünk, ha a következő tulajdonságokkal rendelkezik:

- Tartalmaz egy központi szervletet, amely a vezérlés szerepét látja el. A beérkező kéréseket a megfelelő feldolgozó részhez irányítja, amely ellenőrzi az űrlapadatokat, elvégzi a szükséges frissítéseket a modellben, majd ezt követően kiválasztja a megfelelő megjelenítési komponenst válaszként.
- A megjelenítést végző komponensek JSP lapok, amennyiben dinamikus tartalomra van szükség, illetve egyszerű HTML lapok statikus tartalom esetében.
- A modellelemeket egyszerű Java osztályok, illetve Java babok valósítják meg.

Modell 2 architektúrájú Java keretrendszer nagyon sok van, például:

- Struts – Jakarta
- JavaServer Faces – Sun Microsystems

A Java EE konténerek feladata bizonyos szolgáltatások biztosítása az alakalmazások számára. Ilyen szolgáltatások például:

- tranzakciókezelés,
- többszálúság,
- erőforrás-készletek kezelése.

A biztosított szolgáltatások kényelmesebbé teszik az alkalmazások fejlesztését, hiszen ezek fejlesztői mentesülnek bizonyos problémák megoldása alól, amelyeket a konténer biztosít minden egyes Java EE alkalmazás számára.

A Java EE komponensek futtatása feltételezi, hogy előzetesen a komponenst elhelyeztük egy Java EE modulban, amelyet telepítettünk egy

Java EE konténerbe. A komponensre vonatkozóan bizonyos megkötéseket (például biztonsági megkötések) tehetünk az adott modul telepítésleírójában. Ennek következtében a komponens viselkedését ezek a megkötések is befolyásolják, ugyanaz a komponens többféleképpen telepíthető egy konténerbe, és a telepítés módja határozza meg a komponens viselkedését.

Most pedig ismét felsoroljuk a legfontosabb alkalmazásokat, amelyek nélkülözhetetlenek egy Java webalkalmazás fejlesztése során:

- *Webszerver:* olyan szerver alkalmazás, amely HTTP kéréseket fogad, értelmezi ezeket, majd a megfelelő válaszokat visszaküldi az ügyfél programoknak (böngészőknek). Példa: Apache Web Server.
- *Webkonténer:* olyan Java EE szabványnak megfelelő implementáció, amely lehetővé teszi szervletek és JSP lapok futtatását. Gyakorlatilag egy szervlet- és egy JSP motor. Ha egy HTTP kérés egy Java webkomponensre irányul (szervlet vagy JSP), akkor a webszerver a kérést a webkonténerhez irányítja, és a feldolgozás eredményét a webkonténer visszaadja a webszervernek, amely eljuttatja a böngészőhöz. Példa: Tomcat Web Container.
- *Alkalmazásszerver:* olyan szerver, amely lehetőséget teremt üzleti komponensek (pl. EJB) futtatására is. Ezenfelül rendelkezik a webszerver és a webkonténer képességeivel is. Példa: Sun Application Server (Glassfish), Bea WebLogic, IBM WebSphere, Oracle Application Server stb.

1.8. Feladatok

1.8.1. Programozási feladatok

1.1. feladat: A Glassfish alkalmazásszerver tanulmányozása

- Indítsa el a Java EE (Glassfish) alkalmazásszervert (indulás után kiírja, hogy az adminisztrációs felület milyen porton érhető el).
- Indítsa el az alkalmazásszerver adminisztrációs felületét (egy lehetséges eset: http://localhost:8080).
- Lépjen be az alkalmazásszerver adminisztrációs felületébe (Application Server Admin Console). Az alapértelmezett felhasználó és jelszó: (username: admin, password: adminadmin).
- Tanulmányozza a felület nyújtotta lehetőségeket.

1.2. feladat: Készítsen NetBeans környezetben egy webalkalmazást, ezt követően tanulmányozza az alkalmazás szerkezetét. Mi a különbség a `File` és a `Project` nézet között?

1.3. feladat: Az előző feladatban elkészített webalkalmazáshoz adjon hozzá egy új JSP lapot: `datum.jsp`.

1.4. feladat: Helyezzen el az `index.jsp` lapon egy hiperlinket a `datum.jsp` komponensre.

```
<a href="datum.jsp">DATUM</a>
```

1.5. feladat: Ellenőrizze a hiperlink működését!

1.6. feladat: Módosítsa a `datum.jsp` tartalmát a következőképpen:

```
<body>
  <jsp:useBean id="naptar"
               class="java.util.Date" />
  Aktuális dátum a szerveren:
  <ul>
    <li>ÉV   :<jsp:getProperty name="naptar"
               property="year"/></li>
    <li>HÓNAP:<jsp:getProperty name="naptar"
               property="month"/></li>
    <li>NAP  :<jsp:getProperty name="naptar"
               property="date"/></li>
    <li>ÓRA  :<jsp:getProperty name="naptar"
               property="hours"/></li>
    <li>PERC :<jsp:getProperty name="naptar"
               property="minutes"/></li>
  </ul>
</body>
```

1.7. feladat: Helyes-e az eredmény? Hiba esetén az eredmény értelmezésében a `java.util.Date` osztály segíthet.

1.8. feladat: Módosítsa a `datum.jsp` komponens törzsét a következőképpen:

```
<body>
  <jsp:useBean id="naptar"
               class="java.util.Date" />
  Aktualis dátum a szerveren:
  <ul>
    <li>ÉV    :${naptar.year+1900}</li>
    <li>HÓNAP :${naptar.month+1}</li>
    <li>NAP   :${naptar.date}</li>
    <li>ÓRA   :${naptar.hours}</li>
    <li>PERC  :${naptar.minutes}</li>
  </ul>
</body>
```

A $ karakterrel kezdődő kifejezések ún. EL (Expression Language) kifejezések, amelyeket a JSP komponensekről szóló fejezetben ismertetünk.

1.8.2. Tesztkérdések

1.1. kérdés: Melyik protokoll állapotmentes? (1 helyes)

A. FTP
B. HTTP
C. SMTP
D. telnet

1.2. kérdés: Mely technológiapáros alkotja a WWW alapját? (1 helyes)

A. Model-View-Controller (MVC) és a háromrétegű architektúra
B. Java Server Pages (JSP) technológia és az Apache Struts keretrendszer
C. Simple Mail Transfer Protocol (SMTP) és a File Transfer Protocol (FTP)
D. HTTP protokoll és a HTML jelölőnyelv

1.3. kérdés: A szervlet technológia megjelenése előtt milyen technológiát használtak szerveroldali alkalmazások végrehajtására? (1 helyes)

A. Common Gateway Interface (CGI)

B. Uniform Resource Locator (URL)
C. JSP Standard Tag Library (JSTL)
D. JavaServer Faces technológia

1.4. kérdés: A Modell 2 architektúra egy webalkalmazás tervezési minta, amelyben Java osztályok és Java babok alkotják a modellt és ... (1 helyes)

A. minden új kérésre létrejön egy új folyamat, amelynek feladata a kérés kiszolgálása.
B. a válasz HTML lapok és különböző média típusú állományok gyűjteménye.
C. a vezérlést egy szervlet végzi, a megjelenítést pedig JSP lapok.
D. mind a vezérlést, mind a megjelenítést szervletek végzik.

2. FEJEZET

SZERVLETEK

2.1. HTTP kérés–válasz modell

A HTTP (HyperText Transfer Protocol) alapját egy egyszerű és egyben hatékony kommunikációs modell képezi. Ez a modell úgy működik, hogy az ügyfél (általában böngésző) egy kérést küld a kiszolgáló (szerver) adott erőforrásához. Amennyiben az erőforrás elérhető az ügyfél számára, akkor ezt a kiszolgáló elküldi neki, ellenkező esetben pedig hibajelzést küld. A kért erőforrás lehet egy egyszerű HTML, ebben az esetben válaszul ezt fogja megkapni az ügyfél, vagy lehet egy program (webkomponens), ebben az esetben a komponens lefut és előállít egy dinamikus tartalmat, amit visszajuttat az ügyfélhez. A HTTP protokoll tervezésekor a hatékonyság volt a lényeges szempont, ezért állapotmentesre tervezték. Ez azt jelenti, hogy pontosan egy kérés–válasz lebonyolítására alkalmas, azaz a kiszolgáló semmilyen információt nem őriz meg az ügyfélről, miután kiszolgálta azt. Tehát azt sem észleli, ha ugyanazon ügyféltől több egymást követő kérés érkezik.

Az ügyfél többféleképpen is intézhet kérést a kiszolgáló fele:

- beír egy címet a böngésző címsorába,
- rákattint egy weblap-hivatkozásra,
- elküld egy űrlapot.

Ezeken kívül a böngésző egy weblap feldolgozása közben minden egyes kép, stíluslap, applet vagy más médiafájl esetén kérést intéz a böngészőhöz. Több típusú HTTP kérés létezik, a leggyakrabban használtak a GET, illetve a POST kérések. A fenti esetek mindenike HTTP GET kérést küld a kiszolgáló fele, kivételt képeznek az űrlapok. Ebben az esetben az űrlap készítője határozza meg a metódust, amelyre általában HTTP POST metódust használnak, de ez is lehet esetenként GET.

Egy HTTP kérésnek három különböző része lehet:

- kérés sora,
- kérés fejlécek,
- kérés törzse.

A 2.1. táblázat a HTTP metódusokat foglalja össze.

2.1. táblázat. *HTTP metódusok*

HTTP metódus	Magyarázat
OPTIONS	Lekérdezi a szerver kommunikációs opcióit.
GET	A megadott erőforrás letöltését kezdeményezi.
HEAD	Ugyanaz, mint a GET, csak az üzenet törzsét kihagyja a válaszból.
POST	Feldolgozandó adatot küld a szervernek.
PUT	Feltölti a megadott erőforrást.
DELETE	Törli a megadott erőforrást.
TRACE	Visszaküldi a kapott kérést. Ez azt ellenőrzi, hogy a köztes gépek változtatnak-e a kérésen.
CONNECT	Átalakítja a kérést transzparens TCP/IP csatornává. Ezt a metódust jellemzően SSL kommunikáció megvalósításához használják.

2.1.1. A HTTP kérés

Mivel a GET kérés egyike a leggyakrabban használt HTTP metódusoknak, ezért a következőkben erre adunk egy példát:

A HTTP GET kérés a következő sorral kezdődik:

```
GET /distedu/list_courses.view HTTP/1.0
```

A sorban három elem van, ezek jelentése a következő:

- GET: a HTTP metódus neve,
- /distedu/list_courses.view : erőforrás-azonosító,
- HTTP/1.0: a HTTP verziószáma.

A kérés sorát fejlécsorok követik, amelyek száma nincs korlátozva. Minden fejlécsor tartalmazza a fejléc nevét, ezt a kettőspont (`:`) karakter követi, majd a fejlécértékek. Ezután következik egy üres sor, majd ezt követheti a kérés törzse. A kérés törzse opcionális. Az alábbi példa egy érvényes HTTP GET kérés:

```
GET /index.html HTTP/1.1
```

```
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; en-US;)  Firefox/3.0.3
Accept: text/html,application/xhtml+xml,
        application/xml;q=0.9,*/*;q=0.8
Accept-language: en-us,en;q=0.5
Accept-encoding: gzip,deflate
Accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

A kérés első sora arra kéri a kiszolgálót, hogy küldje el az `index.html` erőforrást HTTP/1.1 protokollt használva. A `Host` sorban a kiszolgáló neve és portszáma van. A `User-Agent` sor a kérést küldő ügyfél információkat tartalmazza. A kiszolgáló ennek alapján megfelelő tartalmat tud küldeni a böngészőnek. Ugyancsak ebből tudja kideríteni a kiszolgáló, hogy milyen eszközről érkezett a kérés, amely lehet például mobiltelefon is. Az `Accept` kulcsszóval kezdődő sorok a böngésző képességeiről értesítik a kiszolgálót. Ilyenek például a fájlformátumok, nyelvek, karakterkódolások stb.

A fontosabb HTTP kérés fejléceket a 2.2. táblázat tartamazza:

2.2. táblázat. *Fontosabb HTTP kérés fejlécek*

Fejléc	**Magyarázat**
Accept	Milyen MIME típusokat fogad az ügyfél
Host	A kért erőforrás adatai: szerver és portszám
Referer	Az ügyfél címe
User-agent	Információk az ügyfélről

2.1.2. A HTTP válasz

A kérés megérkezésekor a kiszolgáló először az URI-vel megadott erőforrást azonosítja, majd a kiegészítő információk alapján eldönti a kérés kezelésének módját. A válasz szerkezete a kérés szerkezetéhez hasonló. Ez is három egységből áll: állapotsor, fejlécek és törzs. A következő egy HTTP válasz példa:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

```
Date: Tue, 8 Mar. 2009 10:00:03 GMT
Server: Apache Tomcat/6.0-bl

<HTML>
  <BODY>
    <H1>Hello, itt vagyok!</H1>
  </BODY>
</HTML>
```

Az állapotsor a protokoll nevével kezdődik, ezt követi az eredménykód és az eredmény szöveges alakja. A mi esetünkben az eredménykód 200, ez pedig sikeres kérésteljesítést jelent. Az OK jelentése is hasonló. Az állapotsort a fejlécsorok követeik, ebből akárhány lehet. A válasz fejlécsorok feladata a kérés fejlécsorokéhoz hasonló. A fejléceket a törzstől egy üres sor választja el. A 2.3. táblázat a leggyakrabban használt válasz fejléceket szemlélteti.

2.3. táblázat. *Fontosabb HTTP válasz fejlécek*

Fejléc	**Magyarázat**
Content-Type	MIME típus
Content-Length	A hasznos válasz hossza
Server	A választ küldő szerver neve
Cache-Control	A böngészőnek küldött direktíva arról, hogy a válasz betehető-e a gyorsító tárba (cache)

2.1.3. HTTP válaszkódok

A webszerver a válasz első sorában egy kódot küld, amely a válasz sikerességére vonatkozik [6]. Ezek a kódok három számjegyű számok. Az első számjegy azonosítja a válasz kategóriáját:

- 1-gyel kezdődőek: „információt adnak a kérés kezelésének módosításáról" [6],
- 2-vel kezdődőek: a kérés kiszolgálása sikeres volt,
- 3-mal kezdődőek: átirányítás történt, további lépésekre van szükség a kérés kiszolgálásához,

– 4-gyel kezdődőek: kliensoldali hiba, például hibás szintaxis vagy nem teljesíthető kérés,
– 5-tel kezdődőek: szerveroldali hiba, például a dinamilus tartalom előállítása közben hiba lépett fel.

Gyakran előforduló válaszkódok:

– 200: sikeres válasz,
– 401: a kliensenek nincs jogosultsága a kért erőforráshoz,
– 404: a kért erőforrás nem található,
– 500: szerverhiba, például le nem kezelt kivétel kiváltódása esetén.

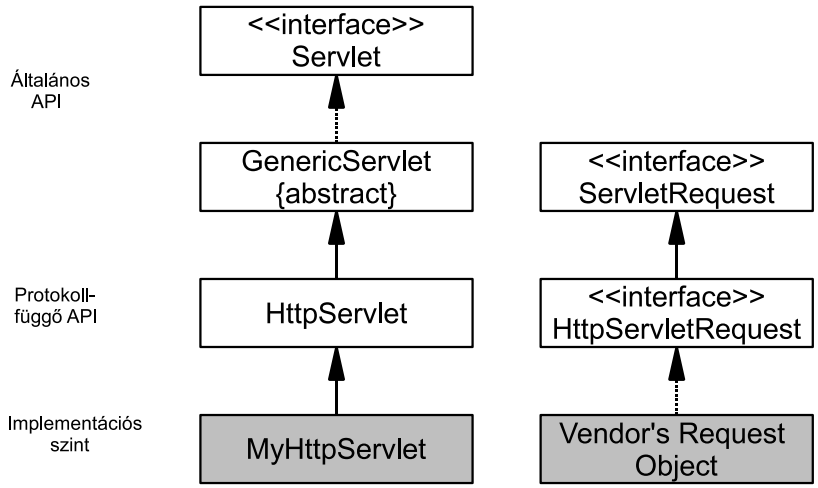
2.2. Servlet API

A Servlet API különböző szintű programozást tesz lehetővé, ezt szemlélteti a 2.1. ábra. Minden egyes szervlet egy webkomponens, amely paraméterezhető, akárcsak az önálló Java alkalmazások. A szervletek viszont nem parancssorból kapják paramétereiket, hanem a `web.xml` telepítésleíró állomány tartalmazza ezeket. A Servlet API metódusokat biztosít ezen paraméterek feldolgozására. Hasonlóképpen műveleteket biztosít a kérés feldolgozására és a válasz előállítására. A protokollfüggő (jelen estben HTTP) funkciók közül megemlítjük a munkamenetek (szessziók) kezelését biztosító osztályokat.

A szervlet API három szintjét a következőképpen jellemezhetnénk:

– Az első szint protokollfüggetlen interfészeket és absztrakt osztályokat tartalmaz: `Servlet`, `GenericServlet`, `ServletRequest` stb.
– A második szinten levő típusok protokollfüggő kiterjesztései a felső szintű általános típusoknak. Jelen pillanatban csak a HTTP protokollfüggő típusokat tartalmaz ez a szint: `HttpServlet`, `HttpServletRequest` stb.
– A harmadik szint a felhasználói szint, ezeket a típusokat a webalkalmazás készítője határozza meg, illetve itt vannak a webkonténer specifikus típusok is.

Minden olyan osztály, amely vagy a `Servlet` interfészt implementálja, vagy pedig a `GenericServlet` osztályt terjeszti ki, szervlet. Tipikusan mégis a protokollfüggő osztályokat szokás használni, hiszen ebben az esetben extra szolgáltatások válnak elérhetővé, mint például a munkamentek kezelését elősegítő `HttpSession` osztály.



2.1. ábra. *A Servlet API szintjei*

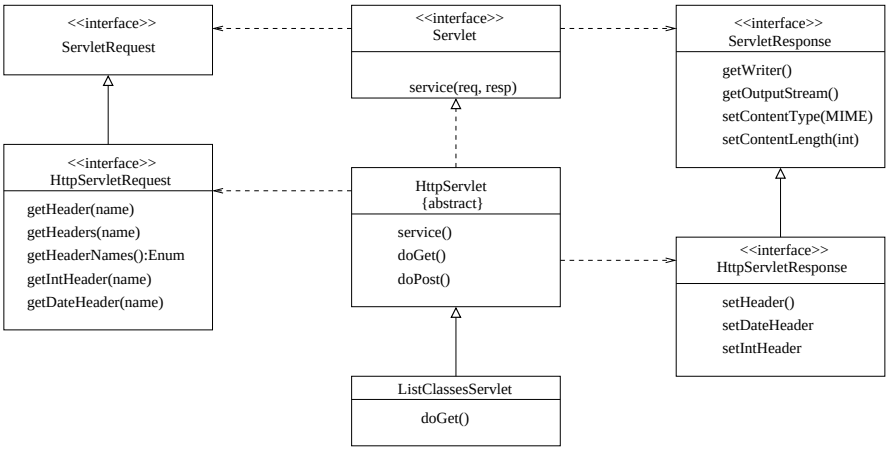
2.2.1. A HttpServlet osztály

A Servlet API legfontosabb osztályait a 2.2. ábra szemlélteti.

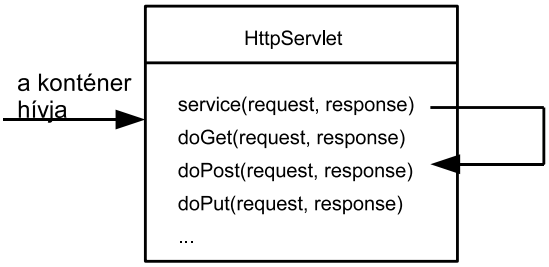
A `HttpServlet` osztály implementálja a `Servlet` interfészben deklarált `service` metódust (l. 2.3. ábra). A webkonténer mindig ezt a metódust hívja, ha az adott szervlethez érkezik a kérés. A `service` metódus pedig a kérés típusának függvényében delegálja a kérés feldolgozását a megfelelő metódushoz. Így például HTTP GET kérés estén a `doGet` metódus, illetve HTTP POST kérés estén a `doPost` metódus kapja meg a vezérlést.

A `service` metódust nem ajánlott felülírni a származtatott osztályban. Leggyakrabban a `doGet`, illetve a `doPost` metódusokat szokás felülírni, esetleg mindkettőt. Amennyiben a szervletnek mind a GET, mind pedig a POST metódusra ugyanazt a választ kell generálni, akkor megoldás lehet, hogy definiálunk például egy `processRequest` nevű metódust az adott szervlet osztályban, és a `doGet`, illetve `doPost` metódusok ehhez irányítják a kérést. Ezt szemlélteti a 2.4. ábra, és a következő kódrészlet is.

```
package view;
import javax.servlet.*;
import java.io.*;
```



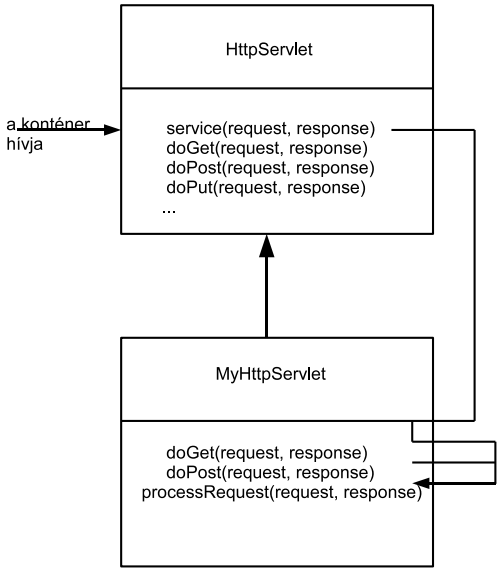
2.2. ábra. *A Servlet API*



2.3. ábra. *A* `service` *metódus*

```
public class ListCourses extends HttpServlet {
  public void processRequest(HttpServletRequest request,
  HttpServletResponse response)throws ServletException,
                                      IOException{
  }
  public void doGet(HttpServletRequest request,
  HttpServletResponse response)throws ServletException,
                                      IOException{
      processRequest(request, response);
  }
  public void doPost(HttpServletRequest request,
  HttpServletResponse response)throws ServletException,
```

```
                                        IOException{
      processRequest(request, response);
  }
}
```



2.4. ábra. *A GET és POST metódusok együttes kezelése*

2.2.2. A szervlet konfigurálása

Minden szervletet konfigurálni kell a telepítésleíróban. A szervlet konfigurálásának két része van. Az első rész (`<servlet>`) a szervletosztályhoz rendel egy nevet, a második rész (`<servlet-mapping>`) pedig ehhez a névhez társít egy URL-t, és ezt a hozzárendelt URL-t fogjuk a továbbiakban használni a szervlet elérésére. A telepítésleíró `web.xml` állomány kötelező módon a WEB-INF katalógusban helyezkedik el. A következő egy érvényes szervletkonfiguráció:

```
<servlet>
   <servlet-name>ListCourses</servlet-name>
   <servlet-class>view.ListCourses</servlet-class>
</servlet>
```

```
<servlet-mapping>
   <servlet-name>ListCourses</servlet-name>
   <url-pattern>/list_courses.view</url-pattern>
</servlet-mapping>
```

Figyeljük meg, hogy a `servlet-mapping`, `url-pattern` tagja / jellel kezdődik. Mindig ezzel a szimbólummal kezdődik, ha *egy-az-egyhez* típusú leképzést akarunk megvalósítani. Lehetőség van *egy-a-sokhoz* típusú leképzésre is, azaz ugyanazon szervlethez több URL mintát is társíthatunk, ebben az esetben kiterjesztés szerinti asszociációt végzünk, ezért nem kell a / jel. A következő leképzés erre ad egy mintát:

```
<servlet-mapping>
   <servlet-name>ListCourses</servlet-name>
   <url-pattern>*.view</url-pattern>
</servlet-mapping>
```

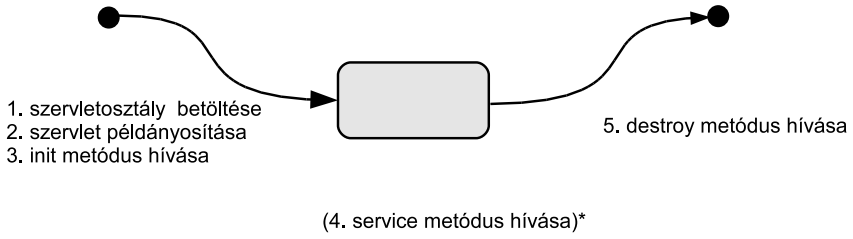
A fenti leképzés alapján minden `.view`-ban végződő kérés URL mintát a `ListCourses` szervlet fog kiszolgálni.

2.2.3. A szervlet életciklusa

A szervletet mindig a webkonténer hozza létre, betöltve a konfigurációban megadott osztályt és példányosítva ezt. Ezután meghívja a szervletet inicializáló `init` metódust. Minden egyes szervletből csak egy példányt készít. A párhuzamos kiszolgálás úgy valósul meg, hogy minden egyes kérésre a webkonténer külön szálat hoz létre, amelyben meghívja az adott szervlet `service` metódusát. A szervlet életciklusában az `init` metódus csak egyszer hívódik meg, a `service` metódus pedig akárhányszor meghívódhat. A szervlet megsemmisítése előtt meghívódik a `destroy` metódus, ez utóbbi is csak egyszer (l. 2.5. ábra). Az init–destroy metóduspár hívása az objektum konstruktor–destruktor műveletpárhoz hasonló.

Amennyiben a webkonténernek erőforrásra van szüksége, megsemmisítheti a szervleteket. A webkonténer ugyanakkor garantálja, hogy ha kérés érkezik a szervlethez, akkor a szervletet ismételten létrehozza és inicializálja.

A szervlet életciklusát teljes egészében a webkonténer kezeli, ez azt jelenti, hogy a szervlet metódusokat is a webkonténer hívja meg.



2.5. ábra. *A szervlet életciklusa*

2.2.4. A kérésobjektum

A kérésobjektumot a webkonténer hozza létre, `HttpServlet` esetében ez `HttpServletRequest` típusú. Ez az objektum tartalmazza a HTTP kérésre vonatkozó összes információt és műveleteket biztosít ezek kinyerésére. Továbbá lehetővé teszi, hogy a kérés hatókörébe adatokat regisztráljunk. Ezen adatok elérhetővé válnak az összes olyan webkomponensben, amelyek ugyanazon kérésen dolgoznak.

A kérésobjektumból kinyerhetjük különböző formátumban a fejléc elemeket. A `HttpServletRequest` fejlécekre vonatkozó metódusai (a fejléc megnevezésekben nem tesznek különbséget kis- és nagybetűk között):

- `String getHeader(String name)`: az adott nevű fejlécnek megfelelő értékkel tér vissza. Nem létező név esetén a visszatérített érték null.
- `Enumeration getHeaders(String name)`: adott nevű fejléchez tartozó értékek halmaza.
- `Enumeration getHeaderNames()`: visszatéríti a fejlécneveket.
- `int getIntHeader(String name)`: adott nevű fejléchez tartozó egész érték lekérdezése. Nem létező fejléc esetén az érték -1.
- `long getDateHeader(String name)`: adott nevű fejléchez tartozó dátum érték lekérdezése. Nem létező fejléc esetén az érték -1. Ha a fejléc nem alakítható át dátum típussá, kivétel váltódik ki.

A következő szervlet feladata az ügyféltől érkező fejlécelemeket megjeleníteni:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
      response.setContentType("text/plain");
      PrintWriter out = response.getWriter();
```

```
      out.println("Request's headers");
      out.println();
      Enumeration names = request.getHeaderNames();
      while( names.hasMoreElements()){
            String name =(String)names.nextElement();
            Enumeration values =request.getHeaders(name);
            while( values.hasMoreElements()){
                String value=(String)values.nextElement();
                out.println(name+": "+value);
            }
      }
      out.close();
}
```

A fenti szervlet eredményét (válaszát) láthatjuk alább:

```
host: localhost:8099
user-agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US;
            rv:1.9.0.3) Gecko/2008092417 Firefox/3.0.3
accept: text/html,application/xhtml+xml,application/xml;
        q=0.9,*/*;q=0.8
accept-language: en-us,en;q=0.5
accept-encoding: gzip,deflate
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
keep-alive: 300
connection: keep-alive
referer: http://localhost:8099/fejleclekerdezo/
cookie: JSESSIONID=d4ae10118c8ca0419b224d5b432b
```

Szintén a kérésobjektumból nyerhetjük ki a kérésben elküldött űrlapadatokat is a `getParameter(String name)` metódussal.

Tekintsük a következő űrlaprészletet:

```
<form action="add_course.do" method="POST">
   <input type="text" name="nev"><br><br>
   <input type="submit" value="Elkuld">
</form>
```

Az űrlap a `form` elem `action` attribútumával határozza meg, hogy mely webkompones fogadja a kérés paramétereket. Ez a mi esetünkben az a szervlet lesz, amelyhez a telepítésleíróban az `add_course.do` URL-t társítottunk. A kérés paramétereket feldolgozó szervlet a `nev` paramétert így nyeri ki:

```
String name = request.getParameter("nev");
```

Összesen három metódus áll rendelkezésre a paraméterek kinyerésére:

- `String getParameter( String paraméternév)`: adott nevű paraméter kinyerésére szolgál. Nem létező paraméternév esetén `null` a visszatérési érték.
- `String[] getParameterValues( String paraméternév )`: adott nevű paraméterhez tartozó értékek listája. Akkor használjuk, ha például legördülő listából több értéket is ki lehet választani.
- `Enumeration getParameterNames()`: a kérésben található paraméterek nevét adja vissza.

Adatokat a kérés hatókörébe a `setAttribute( String kulcs, Object value)` metódussal regisztrálhatunk. Ezeket a továbbiakban attribútumoknak fogjuk nevezni. Az attribútumok tárolása a kérés objektumban történik, amely tartalmaz egy asszociatív tömböt ezen attribútumok nevének, illetve értékeinek tárolására (attribútum-név, attribútum-érték). Az attribútum nevet még kulcsnak is nevezzük, ez `String`, az érték pedig `Object` típusú, tehát végeredményben bármilyen objektumot betehetünk ebbe a tömbbe. A lekérdezés a `getAttribute( String kulcs)` metódus segítségével bármely webkomponensben történhet, ahol még érvényes a megfelelő hatókör. Mivel a visszatérített érték `Object` típusú, használat előtt explicit konverziót kell végezni.

Most pedig tekintsünk egy példát, amelyben két szervletünk van, mindkét szervlet ugyanazon kérés kiszolgálásában vesz részt. Az első szervlet feladata regisztrálni egy listaobjektumot a kérés hatókörébe, majd átadni a vezérlést a második szervletnek:

```
List errorMsgs = new ArrayList();
request.setAttribute("errorMsgs",errorMsgs);
//vezérlés átadása a következő szervletnek
```

A vezérlés átadásával a kérés objektum is továbbítódik a második szervlethez. Ez a kérésobjektum már tartalmazza az első szervlet által elhelyezett listaobjektumot is, és ezt fogja a második szervlet kinyerni.

```
List errorMsgs =(ArrayList)request.getAttribute("errorMsgs");
```

Adott hatókörbe nemcsak betenni és lekérdezni lehet attribútumokat, hanem ezeket törölhetjük is. Lehetőség van az összes attribútum

nevének lekérdezésére is. Összesítve a következő négy művelet végezhető attribútumokkal, hatókörtől függetlenül:

- `void setAttribute( String name, Object value )`
- `Object getAttribute( String name )`
- `Enumeration getAttributeNames()`
- `void removeAttribute( String name )`

2.2.5. A válaszobjektum

A válaszobjektum típusa `HttpServletResponse`, a továbbiakban ezen osztály fontosabb metódusait ismertetjük. A válasz előállításának első lépése a küldendő válasz típusának beállítása. Az alapértelmezett típus a `text/html` MIME típus, amely a `setContentType( String type)` metódussal állítható be.

A következő lépés az, hogy a válasz típusától függően kinyerjük a kimeneti adatfolyamot. Nyilvánvaló, hogy a válasz vagy szöveges, vagy pedig bináris lesz. Mindkét típusra létezik egy-egy metódus, de a kettő közül csak az egyiket használhatjuk. Ha mégis megpróbáljuk mindkét metódust meghívni egymás után, a második hívásra `IllegalStateException` kivétel fog kiváltódni.

```
java.io.PrintWriter getWriter()
javax.servlet.ServletOutputStream getOutputStream()
```

Szöveges adatfolyamot a `getWriter()`, illetve binárisat `getOutputStream()` metódussal kell kinyerni. A lenti szervlet egy HTTP GET kérésre adott választ fog előállítani. A szervlet 4. sora beállítja a válasz típusát, az 5. sora kinyeri a szöveges típusú kimeneti adatfolyamot és ebbe beírja a Hello szöveget. Az adatfolyam lezárásával a válasz befejeződik és eljut a HTTP GET kérést intéző ügyfélhez.

```
1 public void doGet(HttpServletRequest request,
2                   HttpServletResponse response)
3                    throws ServletException, IOException {
4       response.setContentType("text/plain");
5       PrintWriter out = response.getWriter();
6       out.println("Hello");
7       out.close();
8 }
```

Szükség esetén a `setHeader, addHeader` metódusokkal a válasz fejléceket is beállíthatjuk. A `setHeader` egyetlen fejlécelemet állít be, törölve annak előző értékét. Az `addHeader` metódus lehetővé teszi, hogy ugyanazon fejlécelemhez több értéket is felvegyünk. A `HttpServletResponse` fejlécekre vonatkozó metódusai:

- `void addHeader( String name, String value)`: adott nevű fejléc hozzáadása,
- `boolean containsHeader( String name)`: jelzi, ha egy adott nevű fejléc már szerepel,
- `void setHeader( String name, String value)`: adott nevű fejléc beállítása. Ha már van adott nevű fejléc, azt felülírja,
- `void setIntHeader( String name, int value)`: ugyanaz, mint a setHeader, viszont csak egész értékű fejléc beállítására használható,
- `void setDateHeader( String name, long date)`: dátum típusú fejléc értékének beállítására használható. Az időpontot, 1970. január 1., 00:00:00 óta eltelt ezredmásodpercek számával kell megadni,
- `void addIntHeader( String name, int value)`,
- `void addDateHeader( String name, long date)`.

Még megemlítjük a `sendRedirect` metódust, amely segítségével átirányítást végezhetünk egy megadott URL-re. Az URL a metódus paramétere lesz és megadható abszolút, illetve relatív módon is. Amennyiben relatív URL-t használunk, ezt a webkonténer átalakítja abszolút URL-re és utána végzi az átirányítást. Az átirányítás mind a szerver-, mind pedig a kliensoldalon érzékelhető lesz. Kliensoldalon a böngésző címsorában megjelenik az általunk megadott URL. A metódus szintaxisa:

```
public void sendRedirect(java.lang.String location)
                  throws java.io.IOException
```

2.3. Szervletek közötti kommunikáció

Az ügyféltől érkező kérést továbbíthatjuk (delegálhatjuk) egy másik webkomponens felé. Ez teljes egészében a szerveroldalon zajlik, így az ügyfél ezt nem is észleli, hiszen a kérés URL nem változik a böngészőben. A delegálást egy `RequestDispatcher` objektum segítségével végezhetjük, amelyre referenciát a kérésobjektumtól szerezhetünk.

A `getRequestDispatcher` metódusnak át kell adni annak a webkomponensnek az URL-jét, amelyhez továbbítjuk a kérést. A következő kódsor referenciát szerez a `RequestDispatcher` objektumra, megadva egyben annak a webkomponensnek a nevét, amelyhez a kérést továbbítani szeretnénk:

```
RequestDispatcher comp =
       request.getRequestDispatcher("error.jsp");
```

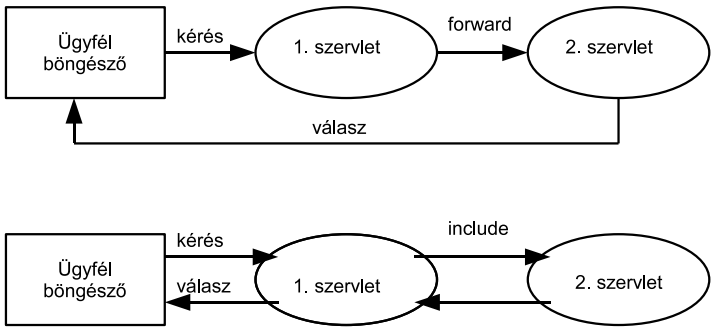
A kérés továbbítása kétféleképpen történhet, `forward`, illetve `include` hívással (l. 2.6. ábra):

- `forward`: a kérés továbbítódik a megadott webkomponenshez. A válasz generálása a fogadó komponens feladata, mi több, a továbbítás előtt a válaszpuffer tartalma automatikusan törlődik. Ezért ha a továbbítás előtt írtunk is adatokat a válaszpufferbe, ezek elvesztődnek. Ha a továbbítás előtt lezárjuk a választ, akkor `IllegalStateException` futásidejű hibát kapunk.

  ```
  comp.forward(request,response);
  ```

- `include`: lehetővé teszi más erőforrás tartalmának beillesztését a válaszba. Az így beillesztett erőforrások nem változtathatják meg a válasz fejléceket és nem zárhatják le a választ.

  ```
  comp.include(request,response);
  ```



2.6. ábra. *A kérés továbbítása `forward`, illetve `include` metódusokkal*

A ServletContext interfész

A `ServletContext` interfész rögzíti a szervletek és a szervletkonténer közötti kommunikáció metódusait. Tulajdonképpen egy-egy `ServletContext` típusú objektum képviseli a webalkalmazásokat a webkonténeren belül. Segítségével például a következőket végezhetjük:

1. attribútumokat tárolhatunk a webalkalmazás hatókörében,
2. szervlet paramétereket dolgozhatunk fel,
3. kéréseket továbbíthatunk ugyanazon webalkalmazás más komponensei felé,
4. erőforrásokhoz férhetünk hozzá,
5. naplózást valósíthatunk meg.

1. Az alkalmazás hatóköréhez ugyanúgy adunk hozzá attribútumokat, mint a kérés hatóköréhez. A következő kódrészlet egy szervlet típusú osztályhoz tartozik:

```
ServletContext sc = getServletConfig().getServletContext();
String name ="Your Name";
sc.setAttribute("name", name);
```

2. A szervletek is kaphatnak paramétereket, akárcsak az önálló Java alkalmazások. A paramétereket az alkalmazás telepítésleírójában helyezhetjük el a következőképpen:

```
<servlet>
  <servlet-name>AdminViewServlet</servlet-name>
  <servlet-class>
     distedu.view.AdminViewServlet.class
  </servlet-class>
  <init-param>
     <param-name>email</param-name>
     <param-value>admin@distedu.org</param-value>
  </init-param>
</servlet>
```

A paramétereket egyszer kell feldolgozni egy szervlet életciklusa során, éspedig rögtön a szervlet példányosítása után. Ezért a paraméterek feldolgozását a szervlet `init` metódusában indokolt végezni.

```
ServletContext sc = getServletConfig().getServletContext();
String email = sc.getInitParameter("email");
```

Egy szervlet összes paraméterének nevét is lekérdezhetjük:

```
ServletContext sc = getServletConfig().getServletContext();
Enumeration parameters = sc.getInitParameterNames();
```

3. A `ServletContext` segítségével is továbbíthatók a kérések más erőforrások fele. A kérés továbbításának módját már szemléltettük az előző alfejezetben. Az előző alfejezetben a kérés továbbítását úgy végeztük, hogy a `HttpRequest getRequestDispatcher()` metódusát használtuk. Most a `ServletContext getRequestDispatcher()` metódusát fogjuk használni. A különbség a célerőforrás URL-jének megadásában van:

- `ServletRequest` típusú objektum esetében a `getRequestDispatcher(String url)` metódust használhatjuk, ahol az url megadása történhet abszolút vagy relatív módon,
- `ServletContext` típusú objektum esetében a `getRequestDispatcher(String url)` metódust használhatjuk, ahol az url megadása csak abszolút módon történhet.

4. A webalkalmazás erőforrásait általában virtuális nevekkel azonosítjuk. Ez növeli az alkalmazás rugalmasságát is, hiszen nem lesznek bedrótozott nevek a webalkalmazásban. Bizonyos esetekben szükség lehet az erőforrás fizikai jellemzőire is. A fizikai jellemzők kinyerését a következő metódusok hivatottak segíteni:

- `String getRealPath( String path )`: egy virtuális névnek megfelelő fizikai nevet ad meg. Fájl erőforrás esetében ez a fájlnév lesz a fájlrendszerbeli abszolút elérési útvonallal együtt.
- `Set getResourcePaths( String path )`: a paraméterként megadott útvonalon található erőforrások listáját adja vissza. Az útvonalnak '/' karakterrel kell kezdődnie.
- `InputStream getResourceAsStream( String path )`: a paraméterként megadott erőforráshoz megnyit egy bemeneti csatornát. Így akár bájtonként is feldolgozhatjuk az adott erőforrást.
- `java.net.URL getResource( String path )`: a paraméterként megadott erőforrás URL-jét adja vissza.

5. Naplózás. A `ServletContext` interfész két metódust tartalmaz a naplózási funkció megvalósításához:

- `log(String message)`

– log(String message, Throwable e)

Az egyetlen probléma ezzel a fajta naplózással, hogy ez a lehetőség egy webalkalmazás keretén belül csak azon komponensekben vehető igénybe, amelyek hozzáférnek a ServletContext objektumhoz. A közönséges Java osztályokban, amelyekkel modell elemeket valósítunk meg, nincs lehetőség elérni a ServletContext objektumot.

Legyen például egy eduservice nevű webalkalmazásunk. Ekkor a naplózást végző kódrészlet, illetve az általa előállított kimenet alább látható:

Programrészlet:

```
ServletContext sc = getServletConfig().getServletContext();
String resource ="/index.jsp";
java.net.URL url = sc.getResource( resource );
sc.log("Resource: "+resource+"\t"+" URL: "+url);
```

Eredmény:

```
Resource: /index.jsp  URL: jndi:/server/eduservice/index.jsp
```

Bizonyos alkalmazásokban szükség lehet átmeneti állományok tárolására. Például ha a webalkalmazásnak valamilyen műveletet kell végeznie egy feltöltött fájlon, akkor ezt a fájlt el kell tárolni előzetesen a szerveroldalon. A szervletspecifikáció lehetővé teszi, hogy hozzáférjünk egy olyan könyvtárhoz, amelyhez írásjogunk is van. Bármely webkonténert használjuk, az írható könyvtárhoz a következőképpen férhetünk hozzá:

```
File tempdir = (File)getServletContext().
  getAttribute("javax.servlet.context.tempdir");
```

2.4. Távoktatás alkalmazás

Ettől a fejezettől kezdődően egy távoktatási alkalmazás egyes moduljait fogjuk megtervezni és implementálni. Alapvetően egy ilyen alkalmazásnak két típusú felhasználója létezik, az egyik az adminisztrátor, aki feltölti az alkalmazást adatokkal, és a másik a hallgató, aki bizonyos tanfolyamokra feliratkozik. Egy nagyon leegyszerűsített változattal fogunk dolgozni, az adminisztrátor a tanfolyamok listáját fogja karbantartani,

a hallgató pedig tanfolyamokat listázhat és igény szerint feliratkozhat ezekre. Ebben az egyszerűsített változatban időpontokkal nem foglalkozunk.

Fontosnak tartjuk megjegyezni, hogy a programozási feladatok megoldásai nem lesznek tökéletes megoldások. Minden fejezetben az adott fejezetben közölt ismeretek segítségével próbáljuk megoldani a feladatot, amelyet később lecserélünk egy jobb megoldásra. Igyekszünk az adott megoldás hiányosságaira is felhívni a figyelmet.

Feladatok:

- Készítsük el a használati eset diagramot.
- Implementáljuk a tanfolyamok listázását.
- Implementáljuk az új tanfolyam felvitelét.

2.4.1. Használati eset diagram

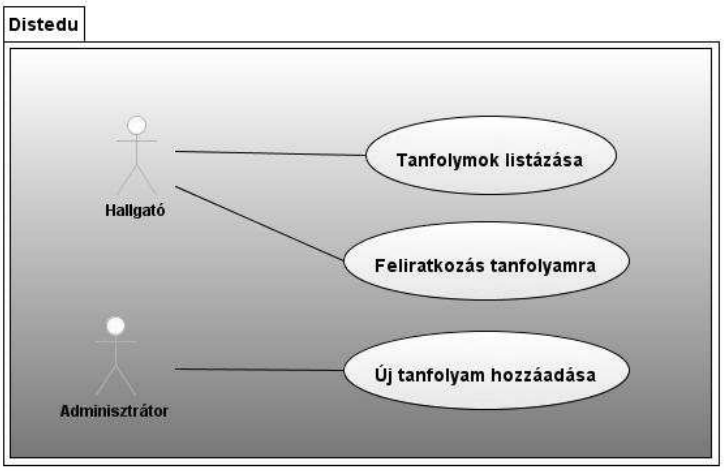
A 2.7. ábra a használati eseteket szemlélteti. A Hallgató típusú felhasználó két műveletet végezhet a rendszerrel: vagy a tanfolyamokat listázza, vagy pedig kiválaszthatja azokat a tanfolyamokat, amelyeket végig szeretne hallgatni és feliratkozhat ezekre. Ezt a második funkciót csak egy későbbi fejezetben fogjuk implementálni. Az adminisztrátor típusú felhasználónak egyelőre csak a tanfolyam hozzáadása műveletet engedélyezzük. A későbbiekben majd más, tanfolyamok karbantartására vonatkozó műveleteket is bevezetünk.

Az alkalmazás tervezésénél betartjuk a MVC architektúrát. Minden egyes komponenstípusnak, amelyet Java osztállyal valósítunk meg, külön csomagnevet fogunk fentartani. A modell elemek, esetünkben egyelőre a `Course` osztály, a `model` csomagban lesznek elhelyezve. A programozási feladatokhoz a NetBeans integrált fejlesztői környezetet ajánljuk. E jegyzet készítésekor a 6.5-ös változat a legfrissebb, és erre a változatra fogunk hivatkozni.

Első lépésként a NetBeans IDE-t használva hozzunk létre egy Java webalkalmazást.

2.4.2. Modell komponens készítése

A webalkalmazásunk tanfolyamokkal és ezekre bejelentkezett hallgatókkal fog dolgozni. Ezért ezt a két entitást kell modelleznünk. Kezdetben csak a tanfolyam modellezésével foglalkozunk, a hallgatókat későbbre halasztjuk. Úgy tervezzük meg a webalkalmazásunkat, hogy



2.7. ábra. *Distedu használati eset diagram*

a tanfolyamok megtekintése (listázása) bejelentekezés nélkül elérhető funkciója lesz a rendszernek.

A legelső dolog, amit tisztázni kell, hogy hol tároljuk a tanfolyamadatokat. Ideális esetben ezeket az adatokat adatbázisban kell tárolni. Mivel még ezek használatát sem ismertettük, maradjunk egyelőre a szöveges állománynál. Tehát a tanfolyamadatok olyan szöveges állományban lesznek elhelyezve, amelynek minden sora egy-egy tanfolyamot tartalmaz, és egy soron belül a # karaktert fogjuk használni a tanfolyam attribútumok elválasztására. Az alkalmazás valamely szervletének inicializálása során beolvassuk a szöveges állomány tartalmát és létrehozunk a beolvasott adatokból egy tanfolyamokat tartalmazó listát. Ezt a listát regisztráljuk az alkalmazás hatókörébe attribútumként, majd a továbbiakban az alkalmazás minden komponense ebből a listából fogja venni a tanfolyam adatokat. Nyilvánvaló, hogy a lista módosulhat az alkalmazás futása során, ha az adminisztrátor típusú felhasználónk új tanfolyamot ad hozzá a rendszerhez. Ezért szükséges, hogy azon szervlet, amely a tanfolyamlista beolvasását végezte az `init` művelete során, a `destroy` művelete során frissítse a szöveges állomány tartalmát.

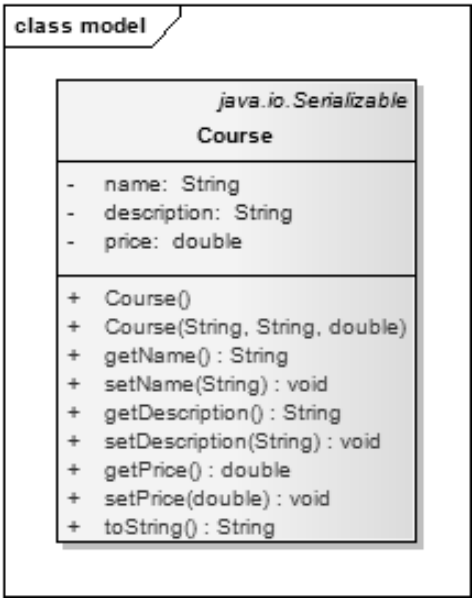
Egy tanfolyamot egy `Course` típusú objektummal fogunk modellezni. Ez alapvetően egy Java bab (`JavaBeans`) típusú komponens lesz. A Java

bab komponens egy olyan Java osztály, amely a következő tulajdonságokkal rendelkezik:

- Minden egyes tulajdonsághoz van `get` és/vagy `set` metódus: ha például van egy `name` nevű mezőnk, akkor ezt `getName` és `setName` metódusokkal olvashatjuk, illetve írhatjuk. Vigyázni kell a névadási konvenciók betartására.
- Az osztály implementálja a `java.io.Serializable` interfészt.
- Az osztály nem tartalmaz publikus mezőt (attribútumot).
- Az osztálynak van paraméter nélkül hívható konstruktora.

A Course osztály

Adjunk hozzá az alkalmazáshoz egy új Java osztályt, ezt helyezzük el egy `model` nevű csomagban. Az osztálydiagram a 2.8. ábrán látható.



2.8. ábra. *A Course osztály*

```
package model;

public class Course implements java.io.Serializable{
    private String name;
```

```
    private String description;
    private double price;

    public Course(){
        this.name = "";
        this.description = "";
        this.price = 0.0;
    }

    public Course( String name,
                   String description, double price){
        this.name = name;
        this.description = description;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String toString(){
        return name+"#"+description+"#"+price;
```

```
    }
}
```

2.4.3. Megjelenítési komponens készítése

A tanfolyamok listázását, bár megjelenítést végző művelet, szervlet segítségével fogjuk megvalósítani. Azért szervlettel, mert még nem ismertettük a JSP lapokat. A későbbiek során majd lecseréljük a szervletet JSP lapra. Legyen ennek a szervletosztálynak a neve `ListCourses`.

Adjunk hozzá az alkalmazáshoz egy új szervletet:

```
File-->New--->Web-->Servlet
```

A szervlet hozzáadását végző dialógusdobozban a szervlet nevéhez írjuk be a `ListCourses` nevet, az URL Pattern(s)-be pedig:

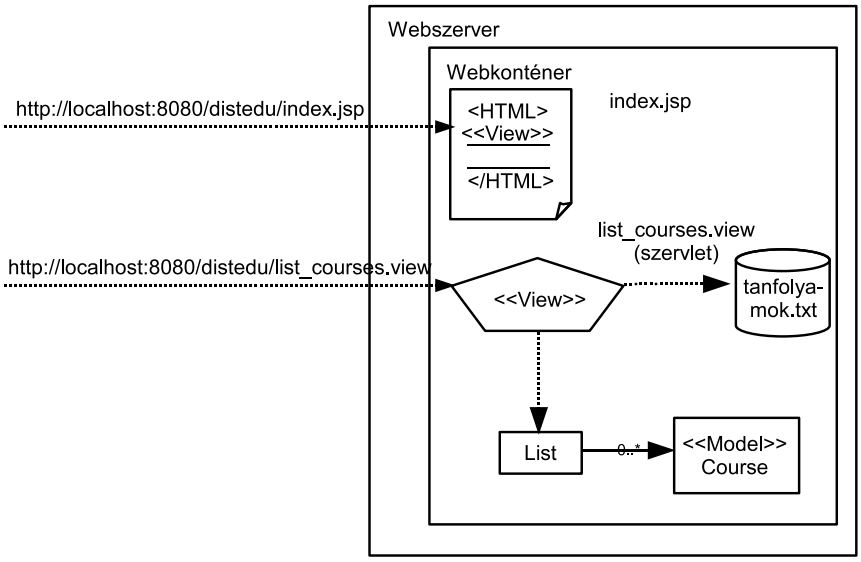
```
/list_courses.view
```

Ezeket az adatokat a szervlet konfigurációjához fogja felhasználni az IDE, előállítva a `web.xml` telepítésleíróban a szervlet konfigurációját:

```
<servlet>
  <servlet-name>ListCourses</servlet-name>
  <servlet-class>view.ListCourses</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ListCourses</servlet-name>
  <url-pattern>/list_courses.view</url-pattern>
</servlet-mapping>
```

A szervlet meghívását az `index.jsp` állományból végezzük, elhelyezve ebbe egy hiperlinket:

```
<body>
  <ul>
    <li><a href="list_courses.view">List Courses</a></li>
  </ul>
</body>
```



2.9. ábra. *A distedu alkalmazás komponensei*

Ha ezzel megvagyunk, rögtön ki is próbálhatjuk a webalkalmazást, hiszen a NetBeans minden egyes webkomponenshez generál egy alapértelmezett tartalmat. Így a szervletünk is egy minimális tartalmat fog előállítani, amit a későbbiek során átírunk.

Felmerül a kérdés, hogy mikor olvassuk be a tanfolyamadatokat az állományból. Ideális itt is az lenne, hogy az alkalmazás indításakor. Úgy fogjuk konfigurálni a szervletünket, hogy az alkalmazás indításakor biztosan betöltődjön és inicializálódjon, egyúttal kiolvasva a tanfolyamadatokat is egy `tanfolyamok.txt` állományból. Tehát az állomány feldolgozását és a tanfolyamok listájának összeállítását a szervlet `init` metódusában fogjuk végezni.

Hogy jobban átlássuk a szervlet szerepét az alkalmazásban, tanulmányozzuk a 2.9. ábrát.

A szöveges állományt úgy kell elhelyeznünk a webalkalmazásban, hogy az csak a webalkalmazás komponensei számára legyen elérhető. Ezért a WEB-INF könyvtárban fogjuk elhelyezni. A következő logikai

lépés az, hogy adjunk hozzá projektünkhöz egy szöveges állományt, ezt mentsük le a WEB-INF katalógusba és töltsük fel adatokkal. Minden tanfolyam külön sorban fog szerepelni és adatait a # szimbólummal választjuk el. Egy lehetséges tartalom a következő:

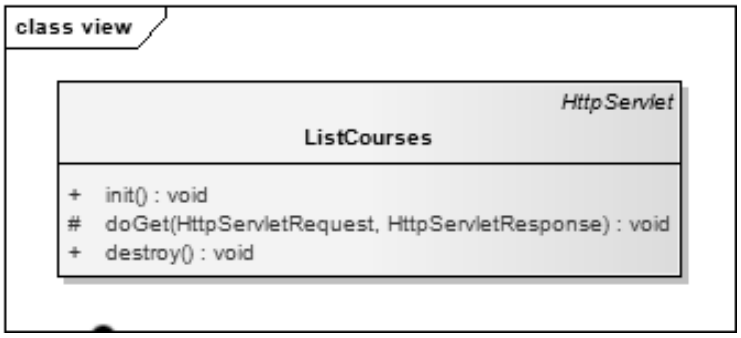
```
Java SE#Java Standard Edition#1000
Java Servlets and JSP#Java Web Programming#1500
Java EE#Java Enterprise Edition#2000
```

Mivel ez az első szervletünk és bizonyos inicializáló műveleteket is el kell végeznie, ezért a kódja elég hosszú. Pontosan ezért darabokban fogjuk az osztályt megadni. Először az osztály vázát ismertetjük, majd rendre a metódusokat.

Az osztály váza

```
package view;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import model.*;

public class ListCourses extends HttpServlet {
   public void init(){
   //...
   }
   protected void doGet(
     HttpServletRequest request,
     HttpServletResponse response)
          throws ServletException, IOException{
   //...
   }
   public void destroy(){
   //...
   }
}
```

A `ListCourses` osztály (l. 2.10. ábra) alapvetően egy megjelenítési funkciót megvalósító szervlet.



2.10. ábra. *A `ListCourses` szervlet*

A megjelenítési funkciót a `doGet` metódus végzi. A szervletre még két extra feladatot bízunk: az `init` metódusában betölti egy szöveges állomány tartalmát és ebből felépít egy tanfolyamokat tartalmazó listaobjektumot, majd a `destroy` metódusában lementi a listaobjektumot, ha erre szükség van. Mentésre csak akkor van szükség, ha futásidőben megváltozik a tanfolyamok száma.

Az init metódus

```
public void init(){
    List<Course> courselist = new ArrayList();

    String resource= "/WEB-INF/tanfolyamok.txt";
    InputStream is= this.getServletContext().
          getResourceAsStream(resource);
    BufferedReader br = new BufferedReader(
                new InputStreamReader(is));
    while( true ){
      String line = null;
      try{
        line = br.readLine();
        if( line == null ) break;
        StringTokenizer stk =
               new StringTokenizer(line,"#");
        Course course = new Course();
        course.setName(stk.nextToken());
        course.setDescription(stk.nextToken());
```

```
        course.setPrice(
           Double.parseDouble(stk.nextToken()));
        courselist.add( course );
      }
      catch( IOException e){
               e.printStackTrace();;
      }
    }

    this.getServletContext().
      setAttribute("coursecounter", courselist.size());
    this.getServletContext().
      setAttribute("courselist", courselist);
    try{
      br.close();
    }
    catch( Exception e){
      e.printStackTrace();
    }
 }
```

Az `init` metódus csak egyszer fut le egy szervlet életciklusában. Miután feldolgozta a megadott szövegállományt és felépítette a listaobjektumot, ezt lementi egy `courselist` nevű attribútumként az alkalmazás hatókörébe. Ezenfelül egy `coursecounter` nevű attribútumot is lement, ez a betöltött tanfolyamok számát tartalmazza. Ezt a második attribútumot azért tároljuk, hogy a tanfolyok mentését csak akkor végezzük el, ha a futásidő alatt megváltozott a tanfolyamok száma.

A doGet metódus

```
protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet ListCourses</title>");
      out.println("</head>");
      out.println("<body>");
```

```
      out.println("<h1> Course list </h1>");
      out.println("<ul>");
      List<Course> courselist =
      (List<Course>)this.getServletContext().
       getAttribute("courselist");
      Iterator<Course> it = courselist.iterator();
      while( it.hasNext()){
        out.println( "<li>"+it.next()+"</li>");
      }
      out.println("</ul>");
      out.println("</body>");
      out.println("</html>");
    }
    finally {
      out.close();
    }
}
```

Ez a metódus gyakorlatilag előállítja a válaszobjektumot. A válasz szöveges típusú lesz, tehát az első lépés, hogy ezt beállítjuk, utána pedig kinyerjük a kimeneti adatfolyamot.

```
response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();
```

Az ezt követő utasítások HTML formátumú szöveget írnak az adatfolyamba, majd lezárják ezt.

A destroy metódus

```
public void destroy(){
     int coursecounter = (Integer)this.
       getServletContext().getAttribute("coursecounter");
     List<Course> courselist = (List<Course>)
       this.getServletContext().getAttribute("courselist");
     if( coursecounter != courselist.size()){
       String resource= "/WEB-INF/tanfolyamok.txt";
       String path =
         this.getServletContext().getRealPath(resource);
       System.out.println("destroy-PATH: "+path);
       try{
         PrintWriter pw =
```

```
            new PrintWriter( new FileWriter(path));
          Iterator<Course> it = courselist.iterator();
          while( it.hasNext() )
            pw.println( it.next());
          pw.close();
        }
        catch(Exception e){
          e.printStackTrace();
        }
      }
}
```

A `destroy` metódus is csak egyszer hajtódik végre egy szervlet életciklusában. Ebben a metódusban ellenőrizzük, hogy történt-e változás a tanfolyamok számát illetően, és ha igen, akkor lementjük a listaobjektumot a `tanfolyamok.txt` szöveges állományba. Mivel szervletünkre a webalkalmazás inicializálását is rábíztuk, vigyáznunk kell, hogy ez mindig elsőként betöltődjön. Ezt a telepítésleíróban oldhatjuk meg a `load-on-startup` sorral, amelynek 1 értéket adva a szervlet elsőkénti betöltését garantálja:

```
<servlet>
  <servlet-name>ListCourses</servlet-name>
  <servlet-class>view.ListCourses</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Próbálja ki a következőket:

- Indítsa el az alkalmazást. Ha minden beállítás rendben van, akkor a böngészőt automatikusan indítja a NetBeans. A NetBeans IDE webalkalmazás futtatásakor az Output ablakban megjeleníti a webalkalmazás böngészőből való elérhetőségét. Például:

  ```
  Browsing: http://localhost:8099/distedu/
  ```

  Ha nem indul automatikusan az alkalmazás, írjuk be a böngészőbe a fenti URL-t.
- Módosítsa a `ListCourses` szervlet `doGet` metódusát. Miután lekérte az alkalmazás hatóköréből a `courselist` listaobjektumot, adjon hozzá egy új Course objektumot:

```
courselist.add(new Course("Proba"+
  (int)(Math.random()*1000),"proba",0.0));
```

Ahányszor rákattint a tanfolyamok listázását végző hiperlinkre, annyiszor lefut a szervlet `doGet` metódusa, bővítve a listaobjektumot. Két listázás után már így nézhet ki a kimenet:

```
Course list


    * Java SE#Java Standard Edition#1000.0
    * Java Servlets and JSP#Java Web Programming#1500.0
    * Java EE#Java Enterprise Edition#2000.0
    * Proba212#proba#0.0
    * Proba385#proba#0.0
```

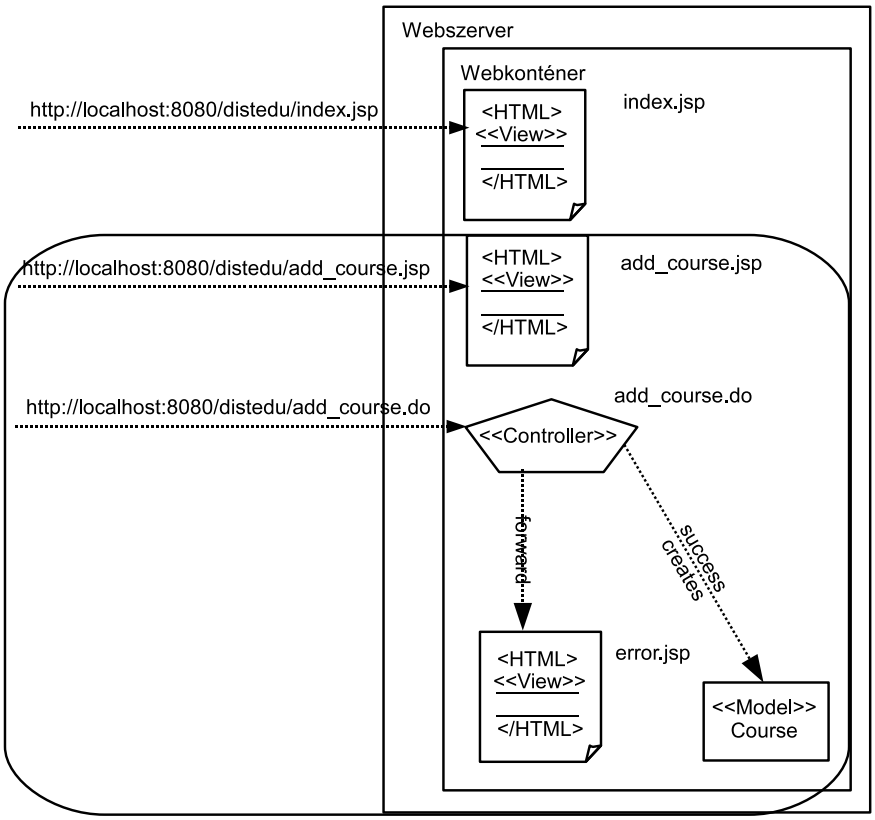
– Állítsa le a webkonténert és utána ellenőrizze a `tanfolyamok.txt` szöveges állomány tartalmát. Ha jól működik az alkalmazása, akkor itt már észlelnie kell a változásokat, vagyis az állomány mérete annyi sorral lett hosszabb, ahányszor lefutott a szervlet.
– Törölje ki a `doGet`-ben végzett módosítást (az előbb hozzáadott kódsort).

2.5. Űrlapok feldolgozása

2.5.1. Vezérlési komponens készítése

A használati eset diagram következő megvalósítandó eleme az új tanfolyam hozzáadása. Ez az új funkció két új webkomponens bevezetését jelenti alkalmazásunkba. Szükségünk van egy űrlapra, amely megjelenítési komponens, továbbá pedig egy űrlapadatokat feldolgozó komponensre, ez egy vezérlési komponens lesz. Az űrlapot egy `add_course.jsp` komponenssel valósítjuk meg, habár egyelőre csak HTML tartalommal. A vezérlésre egy új, `AddCourse` nevű szervletet fogunk hozzáadni az alkalmazáshoz. Az új tanfolyam adatainak beírása után az adatokat elküldjük a webkonténernek, amely lefuttatja a feldolgozást végző szervletet. Itt először ellenőrizzük az adatokat. Hibás adatok esetén átadjuk a vezérlést egy hibalapnak, amely egyszerűen kiírja a hibákat. Helyes adatok esetében végrehajtódik az üzleti logika: létrehozunk egy új `Course` típusú objektumot és ezt hozzáadjuk a tanfolyamlistához.

A 2.11. ábra szemlélteti az alkalmazás szerkezetét.



2.11. ábra. *A distedu alkalmazás komponensei*

2.12. ábra. *Egyszerű űrlap*

2.5.2. HTML űrlap készítése

Készítsünk egy olyan űrlapot, amely egy új előadás adatainak bevitelét teszi lehetővé. Az előadás paramétereit már rögzítettük, ezt modellezi a `Course` osztály. Az űrlapunk egy `Course` típusú objektum inicializálásához szükséges adatok bevitelét teszi lehetővé. Helyezzük az alábbi űrlap részletet egy `add_course.jsp` webkomponensbe.

A form elem

```
<form action="add_course.do" method="POST">
  Name: <input type="text" name="name"
                           size="50">
  <br><br>
  Description: <input type="text" name="description"
                                   size="75">
  <br><br>
  Price: <input type="text" name="price"
                                   size="10">
  <br><br>
  <input type="submit" value="Add Course">
</form>
```

Ez az űrlap böngészőben a 2.12. ábrán látható módon jelenik meg.

A böngésző feladata az űrlap megjelenítése, amely esetünkben három szövegdobozt és egy űrlap adatainak elküldésére szolgáló `Submit`

nyomógombot tartalmaz. A form elem körülzárja a többi űrlapelemet, amelyek együttesen az űrlap tartalmát jelentik.

A form elemnek két fontos attribútuma van:

- `action:` Ezzel az attribútummal adjuk meg azon szerveroldali komponensnek a relatív URL-jét, amely az űrlap adatainak feldolgozásáért felelős. Pontosabban azt jelenti, hogy amikor a böngészőben megnyomjuk az elküldést végző nyomógombot, akkor szerveroldalon a megadott komponens fog lefutni.
- `method:` Ezen attribútum segítségével megadhatjuk az adatok elküldésénél használni kívánt HTTP metódust. Ez GET, illetve POST lehet, és az alapértelmezett értéke a GET. Ha tehát egy `form` elem nem tartalmaz method attribútumot, akkor az adatok elküldése mindig HTTP GET metódussal történik.

Egy weblapon több űrlapot is elhelyezhetünk, de ezek nem ágyazhatók egymásba. Az is lehetséges, hogy a `form` üres legyen és ne legyen Submit nyomógombja, ekkor viszont JavaScript kód szükséges a küldés megvalósítására. Ezeket rejtett űrlapoknak nevezzük.

Szövegdoboz komponens

Szövegdobozt HTML input elemmel hozhatunk létre. Az input elemet nemcsak szövegdoboz létrehozására lehet használni, ezért ha szövegdobozt akarunk, akkor szükség van egy `type` attribútum megadására, amelynek tartalma *'text'*. A szövegdoboz mérete a `size` attribútummal adható meg, a `name` attribútum pedig rögzíti a mező nevét. A mezőnév a bevitt értékkel együtt fog elküldésre kerülni, így lehetőség van szerveroldalon lekérdezni adott nevű paraméterek értékét.

```
Name: <input type="text" name="name" size="50"/>
```

Submit nyomógomb

Az űrlap adatok elküldését végző nyomógombot szintén `input` HTML elemmel helyezhetjük ki, ebben az esetben a `type` attribútum értéke *'submit'*. A nyomógomb címkéjét a `value` attribútum értéke határozza meg.

```
<input type="submit" value="Add Course"/>
```

A HTTP GET és POST

A `Submit` nyomógomb lenyomása az űrlapadatok elküldését jelenti. Most vizsgáljuk meg, hogyan is történik ez. Az adatok elküldése a böngésző felelőssége, ezenfelül a böngésző figyelembe veszi az űrlap `method` attribútumát és ennek megfelelően fogja csomagolni az adatokat a HTTP kérésbe.

Az űrlapadatok formátuma:

```
mezőnév1=érték1&mezőnév2=érték2...
```

A fenti szintaxisból látható, hogy minden mezőnévhez tartozó érték az = jel után következik, a mezők pedig & jellel vannak egymástól elválasztva. Ha egy mező értéke szóköz karaktert is tartalmaz, akkor ez + jelként jelenik meg a kérésben, úgy, ahogyan az alábbi példa is mutatja:

```
name=Java+SE&description=Java+Standard+Edition&price=1000
```

Az űrlapadatoknál használt speciális karakterek:

- = : mezőnév = mezőérték
- & : elválasztó karakter név–érték párok között
- + : szóköz-helyettesítő
- ? : elválasztó karakter a GET kérés URL része és az űrlap adatai között

HTTP GET

Az űrlapadatokat az URL tartalmazza.

```
GET /add_course.do?name=Java+SE&description=Java+Standard+
                        Edition&price=1000
HTTP/1.1
Host: localhost:8099
User-Agent: ...
Accept:
text/xml,application/xml, ...
Accept-Language: en-us,...
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8;...
Keep-Alive: 300
Connection: keep-alive
```

HTTP POST

Az űrlapadatok a kérés törzsében vannak, a fejlécelemek után, ezektől egy üres sor választja el.

```
POST /add_course.do
HTTP/1.1
...
Ugyanaz, mint a HTTP GET eseteben
...
Referer:http://localhost:8099/distedu/add_course.jsp
Content-Type: application/x-www-form-urlencoded
Content-Length: 57

name=Java+SE&description=Java+Standard+Edition&price=1000
```

HTTP GET és POST összehasonlítása

- HTTP GET használata:
  - Ha a HTTP kérésnek nincs mellékhatása a szerveren.
  - Az űrlap kevés adatot tartalmaz.
  - Megengedhető a kérés URL-jének lementése (bookmark).
- HTTP POST használata:
  - A HTTP kérés feldolgozása megváltoztatja a szerver állapotát (pl. adatokat tárol egy adatbázisban).
  - Az űrlap sok adatot tartalmaz.
  - Az űrlap adatait nem jeleníthetjük meg az URL-ben (pl. jelszó).

2.5.3. HTML űrlap adatainak feldolgozása

Az űrlapadatokat feldolgozó komponenst az MVC modellben a Controller (vezérlő) kategóriába soroljuk, és ezt egy szervlettel szokás megvalósítani. Már jeleztük az űrlap készítésénél, hogy az `action` attribútum egy szerveroldali komponens URL-jének megadására szolgál. Így gyakorlatilag már rögzítve van, hogy milyen URL-t kell hozzárendelnünk szervletünkhöz.

Adjunk hozzá az alkalmazáshoz egy szervletet, amelynek a `doPost` metódusban a következő műveleteket kell elvégeznie:

- az űrlap paramétereinek kinyerése,
- az űrlap paramétereinek átalakítása a kívánt formátumba,
- az űrlap paramétereinek ellenőrzése,
- üzleti logika meghívása.

Most pedig vegyük sorba a fenti műveleteket és nézzük meg, hogyan valósíthatók meg. Kezdjük az űrlap paramétereinek kinyerésével. A `ServletRequest` interfész négy metódussal segíti a paraméterek kinyerését:

- `String getParameter(String name)`
- `String[] getParameterValues(String name)`
- `Enumeration getParameterNames()`
- `Map getParameterMap()`

A metódusnevek önmagukért beszélnek: a `getParameter` adott nevű paraméter értékét téríti vissza. Ha nem létező paramétert kérünk, `null` értéket térít vissza. Ha egy paraméterhez több érték tartozik (ez gyakori jelölőnégyzeteket tartalmazó űrlapokon), akkor a `getParameterValues` metódust ajánlott használni. A `getParameterNames` a paraméterneveket szolgáltatja, a `getParameterMap` pedig asszociatív tömbként adja vissza a paramétereket.

Az előző alfejezetben bevezetett űrlap paramétereit az alábbi kódrészlettel nyerhetjük ki.

```
String name = request.getParameter("name");
String description = request.getParameter("description");
String priceStr = request.getParameter("price");
double price=0;
try{
  price = Double.parseDouble( priceStr );
}
catch( NumberFormatException e){
           //Hibakezeles
}
```

A fenti kódrészletben nemcsak a paraméterek kinyerését végeztük, hanem esetenként átalakítottuk a kívánt formátumba. Most következne a paraméterek ellenőrzése. Habár az űrlap paramétereket kliensoldalon is ellenőrizhetjük, például Javascript segítségével, a szerveroldali ellenőrzést mégis szükséges elvégezni. Megtörténhet ugyanis, hogy az ügyfél letiltja a böngészőben a Javascriptek végrehajtását, és akkor az adatok ellenőrzés nélkül kerülnek szerveroldalra.

Az adatok ellenőrzését úgy fogjuk végezni, hogy az ellenőrzés során észlelt hibákat összegyűjtjük egy karkterlánclistába. A tanfolyam esetében kötelezővé tesszük a név és ár megadását, és az árnak kötelező módon pozitív valós számnak kell lennie.

```
List errorMsgs = new ArrayList();

String name = request.getParameter("name").trim();
if( name == null || name.length() == 0){
   errorMsgs.add("Please eneter the name of the course");
}

String description =
       request.getParameter("description").trim();

String priceStr = request.getParameter("price").trim();
double price=0;
try{
   price = Double.parseDouble( priceStr );
   if( price <0 ){
      errorMsgs.add("Price must be a positive number");
   }
}
catch( NumberFormatException e){
   errorMsgs.add("Price must be a number");
}
```

Az üzleti logikát csak akkor kell végrehajtani, ha az adatok ellenőrzése sikeres volt. Ellenkező esetben jelezni kell az ügyfélnek, hogy hiba történt.

```
if( !errorMsgs.isEmpty()){
  //Hibakezelés
}
else{
  //Üzleti logika
}
```

Ezt úgy kell megoldani, hogy a vezérlést vissza kell adni az űrlapra a hibaüzenetekkel együtt. Ezt az elegáns megvalósítást későbbre hagyjuk, hiszen ehhez űrlapunkat át kellene alakítanunk úgy, hogy lehetővé tegyük a hibák megjelenítését. Egyelőre készítsünk egy webkomponenst,

amely egyszerűen csak kiírja a hibákat. Mivel ezt JSP lappal egyszerűbb megoldani, ezért most ezt használjuk.

Az error.jsp lap

```
<body>
  <% java.util.List errorMsgs=
     (java.util.List)request.getAttribute("errorMsgs");
     out.println(errorMsgs);
  %>
</body>
```

Most pedig egészítsük ki a szervletünket úgy, hogy hiba esetén a fenti komponensnek adjuk át a vezérlést. Ezt a `RequestDispatcher forward` metódusa segítségével végezzük. Nyilván az `error.jsp` komponensben hozzáférésünk kell legyen az `errorMsgs` listához. Ez most egyelőre a szervletünk kérésfeldolgozó metódusának lokális változója. Regisztráljuk a változót a kérés hatókörébe, ez a hatókör elérhető az összes olyan webkomponensben, amely a kérés feldolgozásában valamilyen módon részt vesz.

```
if( !errorMsgs.isEmpty()){
  request.setAttribute("errorMsgs", errorMsgs);
  RequestDispatcher r =
    request.getRequestDispatcher("error.jsp");
  r.forward(request, response);
}
else{
  Course course = new Course(name, description, price);
  List<Course> courselist =(List<Course>)
    this.getServletContext().getAttribute("courselist");
  courselist.add(course);
 }
```

2.6. Tesztkérdések

2.1. kérdés: Milyen metódus hívódik meg a Submit címkéjű nyomógomb lenyomásakor? (1 helyes)

```
<form action="myservlet">
```

```
   <input type=submit name="Submit"/>
</form>
```

A. doPost
B. doHead
C. doGet
D. submit

2.2. kérdés: A következő kijelentések közül válassza ki az egyetlen helyeset!

A. A service metódus csak egyszer hívódik, a legelső kérés kiszolgálásakor.
B. A service metódus minden egyes kérés kiszolgálásakor meghívódik.
C. A service metódust a webkonténer a doGet, illetve a doPost metódus után hívja.
D. A service metódus csak a HTTP POST kérés feldolgozásakor hívódik.

2.3. kérdés: Adott a következő űrlap:

```
<form action="myservlet" method=POST>
  <input type="text" name = "nev"/>
  <input type="submit" name="Submit"/>
</form>
```

Feltételezve, hogy a myservlet egy érvényes szervlet, a request pedig egy érvényes hivatkozás a kérésobjektumra, a következő kódrészletekből melyik segítségével kaphatjuk meg a nev paraméter értékét az űrlap elküldése után? (1 helyes)

A. request.getHttpParameter("nev")
B. request.getFormParam("nev")
C. request.getValue("nev")
D. request.getAttribute("nev")
E. request.getParameter("nev")

2.4. kérdés: Adott a következő HTML részlet:

```
<html>
  <body>
    <a href="HelloServlet">HelloServlet>POST</a>
  </body>
</html>
```

A HelloServlet URL-el azonosított szervletnek mely metódusa fog meghívódni, amikor a HTML kódot megjelenítő böngészőben a hiperlinkre kattintunk? (1 helyes)

A. doLink
B. doGet
C. doPost
D. doPOST
E. init

2.5. kérdés: Melyik az a metódus, amely HTTP fejléc értékek lekérdezésére használható? (1 helyes)

A. A GenericServlet osztály getHeader(String name) metódusa
B. A HttpsServlet osztály getHeader(String name) metódusa
C. A HttpServletRequest osztály getHttpHeader(String name) metódusa
D. A HttpServletRequest osztály getHeader(String name) metódusa
E. A HttpServletResponse osztály getHeader(String name) metódusa

2.6. kérdés: Mely kódsort kell használnunk egy szervletben, ha a válasz szöveges típusú lesz? (1 helyes)

A. PrintWriter out = response.getWriter();
B. OuputStream out = response.getStream();
C. OutputReader out = response.getStream();
D. ServletWriter out =response.getWriterStream();
E. StreamWriter out = response.getStreamWriter();

2.7. kérdés: Válassza ki a helyes kijelentéseket! (2 helyes)

A. A sendRedirect metódusnak csak abszolút URL adható át

paraméterként.
B. A sendRedirect metódushívás után a böngésző visszatér az eredeti, hívás előtti URL-hez.
C. Ha a sendRedirect metódus hívása a válasz elküldése után történik, kivétel keletkezik.
D. A sendRedirect a HttpServletResponse osztály metódusa
E. A sendRedirect a HttpServletRequest osztály metódusa

2.8. kérdés: Adott a következő kódrészlet:

```
import javax.servlet.*;
import javax.servlet.http.*;
public class WWServlet extends GenericServlet{
}
```

Mely metódust kell a fenti WWServlet osztálynak implementálnia ahhoz, hogy a szervlet fordítása hibátlan legyen? (1 helyes)

A. doGet
B. doService
C. service
D. az összes doXXX metódust
E. Egyetlen metódust sem, hiszen a GenericServlet osztály már tartalmazza a metódusok implementációit.

2.9. kérdés: Adott a következő TestServlet osztály. Válasszuk ki a helyes kijelentéseket! (2 helyes)

```
public class TestServlet extends HttpServlet
{
  public void init()
  {
  }
  public void service(HttpServletRequest req,
              HttpServletResponse res)
  {
    super.service();
  }
  public void doGet(HttpServletRequest req,
              HttpServletResponse res)
  {
    //do something
```

```
  }
  public void doPost(HttpServletRequest req,
              HttpServletResponse res)
  {
    //do something
  }
}
```

A. Egy HTTP PUT kérésre kivételt dob.
B. Egy HTTP PUT kérés esetén a szervletosztály egyetlen metódusa sem fog meghívódni.
C. Bármilyen HTTP kérés esetén meghívódik a service metódus.
D. Bármelyik és minden HTTP kérés esetén az osztály legalább egy metódusa meghívódik.
E. Bármelyik és minden HTTP kérés esetén a szervletosztály legfeljebb 2 metódusa hívódik meg.

2.10. kérdés: Mely sor használható az alábbi `init` metódusban a szervlet `dbname` paramétere elérésére? (2 helyes)

```
public void init()
{
 String dbname = //Melyik sor?
}
```

A. `getInitParameterValue("dbname");`
B. `getInitParameter("dbname");`
C. `getServletContext().getInitParameter("dbname");`
D. `getServletConfig().getInitParameter("dbname");`
E. `getServletConfig().getParameter("dbname");`

2.11. kérdés: A szervlet feladata egy gif formátumú állomány átküldése válaszként. Hogyan lehet kinyerni a küldéshez szükséges kimeneti adatfolyamot? (1 helyes)

```
A. ServletOutputStream out =
       response.getServletOutputStream("image/gif");
B. ServletOutputStream out=response.getOutputStream();
C. FileOutputStream out=
```

```
                    response.getServletOutputStream();
D. PrintWriter out = response.getWriter();
E. PrintWriter out = response.getPrintWriter();
```

2.12. kérdés: A szervletkonténer a szervletpéldány `init` metódusát végrehajtja... (1 helyes)

A. először, amikor létrejön a szervletpéldány, utána pedig minden kérés végrehajtása előtt.
B. ha a kérés egy olyan felhasználótól érkezik, amelyre érvénytelenítődött a menet.
C. legfeljebb egyszer hajtja végre a szervlet életciklusa során.
D. minden egyes szervlethez érkező kérés esetében, amelyre új szál jön létre.
E. azokra a kérésekre, amelyekre új menetobjektumot is létre kell hozni.
F. minden egyes szervlethez érkező kérésre.

2.13. kérdés: Melyik HTTP metódust kell használni az űrlap paramétereinek küldésére, ha ezek nem lehetnek láthatók a böngésző címsorában? (1 helyes)

A. GET
B. POST
C. HEAD
D. HIDDEN
E. PUT

2.14. kérdés: Adott az alábbi űrlap és a hozzá tartozó, űrlapot feldolgozó szervlet. Mit fog kiírni a szervlet? (1 helyes)

```
<form action="/MyFirstServlet" method=POST>
        <input type=text name='param1'>
        <input type=submit>
</form>
```

```
import java.io.*;
import javax.http.*;
import javax.servlet.http.*;
```

```
public class MyFirstServlet extends HttpServlet
{
  public void service(HttpServletRequest req,
                      HttpServletResponse res)
    throws ServletException, IOException{
    res.setContentType("text/html");
  }

  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
    throws ServletException, IOException{
    res.getWriter().println("This is doGet!");
  }

  public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
    throws ServletException, IOException{
    res.getWriter().println("This is doPost!");
  }
}
```

A. This is doPost
B. This is doGet
C. "This is doPost" és "This is doGet"
D. Nem ír ki semmit.

2.15. kérdés: Egy HTTP válasz a következő fejlécet tartalmazza:

```
CustomHeader: 5
```

A következő kódsorok közül melyik használható a fejlécelem lekérdezésére? (2 helyes)

A. request.getHeader("CustomHeader");
B. request.getIntHeader("CustomHeader");
C. request.getHeaders("CustomHeader")[0];
D. request.getHeader("CustomHeader").get(0);
E. request.getHeaderValue("CustomHeader");

2.16. kérdés: Adott a következő szervlet. Mely metódust kell ennek implementálnia, ahhoz, hogy fordításakor ne keletkezzen hiba?
(1 helyes)

```
public class WWServlet extends HttpServlet
{
  //...
}
```

A. service
B. doService
C. doGet
D. minden doXXX metódust
E. egyet sem

3. FEJEZET

MUNKAMENETEK KEZELÉSE

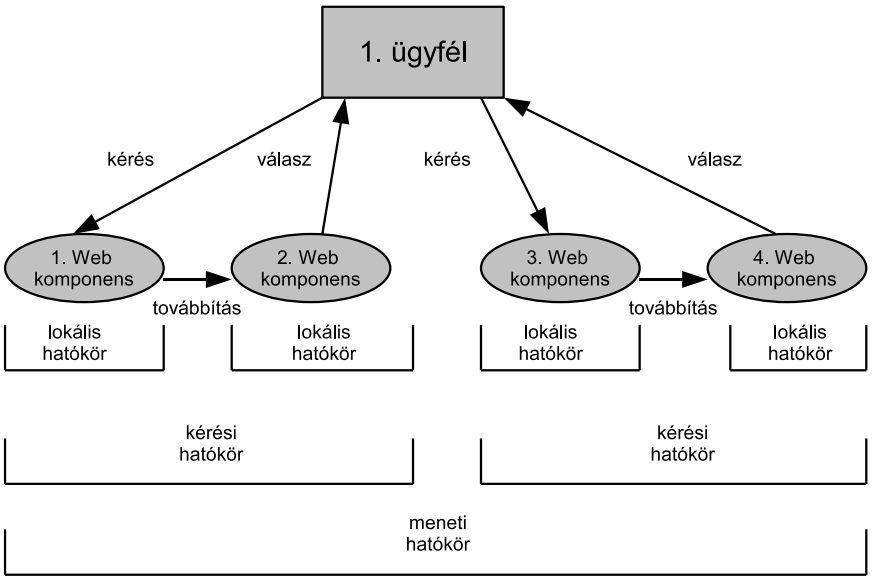
3.1. Munkamenetek

A HTTP protokoll egy állapotmentes protokoll, minden egyes kérés–válasz egymástól függetlenül történik. Tehát a kiszolgáló még ugyanazon kliens két egymást követő kérését is úgy válaszolja meg, mintha két különböző kliens lenne. Ez a modell nagyon hatékony, viszont bizonyos alkalmazások, mint például webáruházak esetében a kiszolgálónak tárolnia kell az egymást követő kérések során kiválasztott árucikkeket. Tehát szükség van egy olyan mechanizmusra, amely képes megoldani ezt a problémát.

A webkonténerek úgy oldják meg ezt a problémát, hogy az ügyfélről információt tárolnak, amelyet megőriznek az egymást követő kérések, azaz a menet idejére. Az információkat menetobjektumokban tárolják, minden egyes aktív ügyfélhez tartozik egy egyedi menetobjektum, amelyet az első kérés alkalmával hoznak létre és amely érvényes az egész menet idejére. Minden egyes menetobjektumnak van egy egyedi azonosítója (SESSIONID). Ezt nevezzük menetazonosítónak. A 3.1. ábra három különböző hatókör közötti különbségeket szemlélteti: ezek a lokális, a kérési, illetve a meneti hatókörök.

Amikor egy böngésző kérést intéz egy webszerverhez, az generál egy egyedi azonosítót és visszaküldi a böngészőnek, amely ezt eltárolja. Ezt követően az összes további kérésben, amit ugyanazon webszerverhez intéz, a kérés adataihoz csatolja a menetazonosítót is. Így a szervernek lehetősége van minden egyes kérést egy menetazonosítóhoz asszociálni. Minden egyes menetazonosító a webszerver számára egy egyedi felhasználót jelent. Lehetőség van a menetazonosító mellett más adatokat is tárolni az adott felhasználóról, például egy webáruház alkalmazás esetében ide helyezhetők az ügyfél által kiválasztott áruk adatai.

Megfigyelhetjük, hogy webprogramozás esetében a változók hatókörei mások, mint ahogy ezt eddig programozásból ismertük. Ez azért van, mert itt alapjában véve egy többfelhasználós rendszerrel van dolgunk, és



3.1. ábra. *Meneti hatókör [5]*

ez a rendszer más hatóköröket igényel. Most is minden szerveroldali változó, memóriában van, de a hozzáférések megváltoztak. Eddig háromféle hatókörről beszéltünk:

– kérési hatókör: egy adott kérést feldolgozó webkomponensek férhetnek hozzá. Ha például egy szervletből átadjuk a vezérlést egy másik webkomponensnek, akkor ezzel együtt a másik webkomponens hozzáférést kap a kérés hatókörébe lementett vátozókhoz. Ezt objektumorientált programozásban nagyon egyszerű megvalósítani, az egységbezárás tulajdonság következményeként. Ha a kérésobjektum egységbezárja a kérés adatait, akkor ugyanígy tesz az ezen hatókörbe tartozó változókkal is. Így aki hozzáfér a kérésobjektumhoz, az hozzáfér ezen változókhoz is. Ezek az adatok csak a kérés kiszolgálásának befejeztéig érvényesek. Olyanok, mint a függvények lokális változói: addig élnek, amíg a függvény végrehajtása tart, és annyiszor jönnek létre, majd halnak el, ahányszor meghívjuk a függvényt. Tipikusan ilyen hatókörben tárolhatunk egy hibalistát, amit egy üzleti logikát végző komponens állít elő, majd ezt továbbítja egy hibalapnak.

- meneti hatókör: egyazon klienstől érkező, egymást követő kérések során elérhető adatok. Ez azt jelenti, hogy két különböző kliens nem látja egymás adatait. Ismét a webáruházzal példálózva, amit az egyik ügyfél vásárol, arról a másik ügyfélnek nem lehet tudomása. Ezek az adatok a menet érvényességi idejéhez kötődnek. Ha megszűnik a menetobjektum, megszűnnek ezen hatókörbe lementett változók is. Tipikusan interaktív alkalmazásoknál használják, például bevásárlókosár megvalósítására.
- alkalmazási hatókör: a webalkalmazás bármelyik ügyfele hozzáfér, és egyben ez a legtágabb hatókör. Az ebbe a hatókörbe lementett változó hasonló a más programnyelvekből ismert globális változóhoz. Érvényességi körük a webalkalmazás futásidejéhez kötődik. Ha leáll a webalkalmazás, megsemmisülnek ezen változók. Tipikusan adatbázis-kapcsolatokat, az összes bejelentkezett felhasználó adatait tárolják alkalmazás hatókörben, szóval mindent, amihez minden ügyfél hozzáférhet.

Nyilvánvaló, hogy bizonyos hatókörökben tárolt változók esetén konkurencia-problémák adódnak. A kérés hatóköre kivétel, mert a kérésobjektum mindig egy felhasználó egy kéréséhez tartozik.

A Servlet API két interfészt biztosít (l. 3.2. ábra), amelyek segítségével kezelhetjük a menetobjektumokat. A menet hatókörébe, ugyanúgy mint a kérés vagy az alkalmazás hatókörébe, (név, érték) párokat tárolhatunk. Ezen értékpárok írása, olvasása, illetve törlése a megszokott módon történik. Például a következő két kódrészlet a menet hatókörébe lementett `Course` típusú objektum mentését, illetve olvasását szemlélteti.

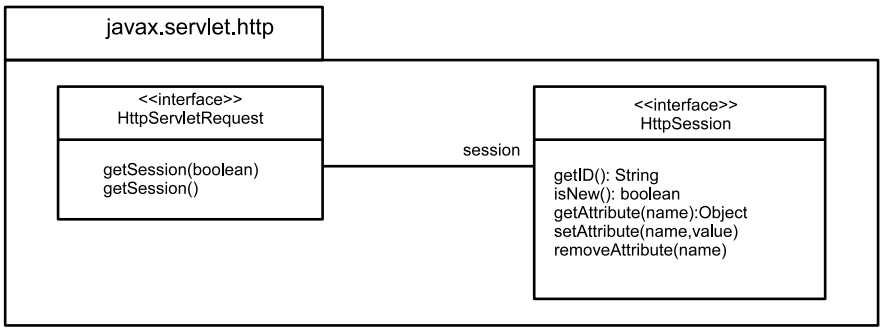
A következő két programrészletben feltételezzük, hogy a `request` egy referencia a kérés objektumra.

1. szervlet

```
HttpSession session = request.getSession();
Course course = new Course("Java SE",
                           "Java Standard Edition", 1000);
session.setAttribute("course", course);
```

2. szervlet

```
HttpSession session = request.getSession();
Course c = (Course)session.getAttribute("course");
```



3.2. ábra. *A HttpSession API [3]*

A fenti kódrészletben használt `getSession` metódusnak két túlterhelt alakja van, egy paraméter nélküli és egy logikai paramétert tartamazó. A paraméter nélküli alak visszatéríti a már létező menetobjektumot (ezt a mentazonosító alapján teszi, ennek átadása automatikus, nem programozói feladat), vagy ha nem létezik, létrehoz egyet és ezt téríti vissza. Hasonlóképpen viselkedik a második alak is `true` aktuális paraméterrel. Ha viszont `false` a paraméter, és nem létezik a menetobjektum, akkor `null` a visszatérített érték.

A menetobjektumok automatikusan jönnek létre, megsemmisítésük viszont történhet automatikusan vagy expliciten. A szervletspecifikáció rögzíti a menetazonosító nevét, amely kötelező módon JSESSIONID. A menetek érvényességi idejét szabályozhatjuk a `web.xml` telepítésleíró, illetve a `HttpSession` interfész metódusai segítségével.

1. A telepítésleíróban megadható a menet lejárati ideje percben:

```
<session-config>
   <session-timeout> 10 </session-timeout>
</session-config>
```

2. A programban megadható a menet lejárati ideje másodpercben:

```
public void setMaxInactiveInterval(int seconds)
```

Amennyiben a programban megadott érték negatív, a menetobjektum soha nem veszíti el érvényességét, vagyis örök életű lesz.

A menetobjektumok megsemmisítése történhet automatikusan a beállított menetidő lejárta után, vagy programozottan. Ez utóbbi a

`HttpSesssion invalidate` metódusának hívásával történik. Hatására azonnal megszűnik a menetobjektum.

Nyilvánvaló, hogy ahhoz, hogy a kiszolgáló érzékelje, hogy két egymást követő kérés ugyanattól az ügyféltől származik, valamilyen formában el kell juttatni a menet azonosítóját (és esetleg más kiegészítő információkat) az ügyfélhez, az ügyfélnek pedig minden egyes kérésbe, amit a kiszolgálóhoz intéz, bele kell illesztenie ezt a menetazonosítót. A Servlet API erre két kényelmes lehetőséget kínál: a sütiket és az URL újraírást.

3.2. Sütik

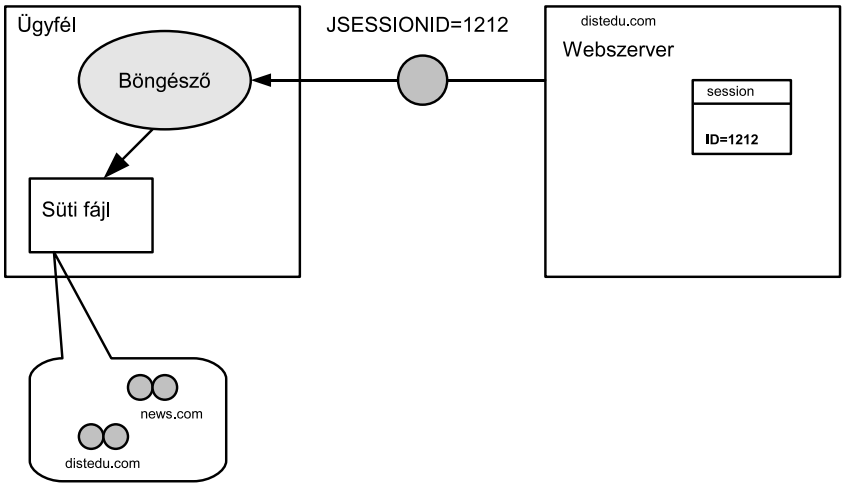
Az előző alfejezetben bemutattuk a menetobjektumok létrehozási, megsemmisítési módját, illetve azt, hogy hogyan oszthatunk meg adatokat egy menethez tartozó webkomponensek között. Nagyrészt a webkonténer végezte a feladatot, nekünk, programozóknak nagyon kevés dolgunk maradt. Jó azonban megismerkednünk a menetazonosító továbbításának módozatával a kiszolgáló és az ügyfél között. Erre az alapértelmezett mechanizmust a sütik (cookie) biztosítják. A `Cookie` technológiát elsőként a Netscape cég vezette be.

A sütik ügyféloldalon tárolt információdarabkák. A böngésző minden egyes webszerver nevéhez több sütit is társíthat és ezeket hozzáilleszti minden egyes olyan kéréshez, amit az adott webszerverhez intéz. Nyilván a sütik sem kötelező módon örök életűek. Ezek életciklusát a böngészőprogramon keresztül szabályozhatjuk, sőt azt is szabályozhatjuk, hogy a böngészőnk fogadjon-e egyáltalán sütiket. Amennyiben ezt letiltjuk, a webalkalmazásnak nem lesz lehetősége sütiken keresztül küldeni a menetazonosítót, így a második módszerhez kell folyamodnia, az URL újraíráshoz. Összefoglalva, a sütiket a következőképpen jellemezhetjük:

- A sütiket a webszerver küldi a böngészőknek.
- A sütik ügyféloldalon vannak tárolva.
- A sütiket tartományokba csoportosítva tárolja a böngésző. A tartományokat a webszerverek határozzák meg. (Példák tartománynevekre: dot.com, news.com stb.)
- Minden egyes kérésben, amit a böngésző egy ismert webszerverhez továbbít, küldi a hozzá tartozó sütiket is.

– A sütiknek van életciklusuk, amelyet a böngészővel is szabályozhatunk.

A 3.3. és 3.4. ábrák a sütik működését szemléltetik.



3.3. ábra. *A süti elküldése az ügyfélprogramnak*

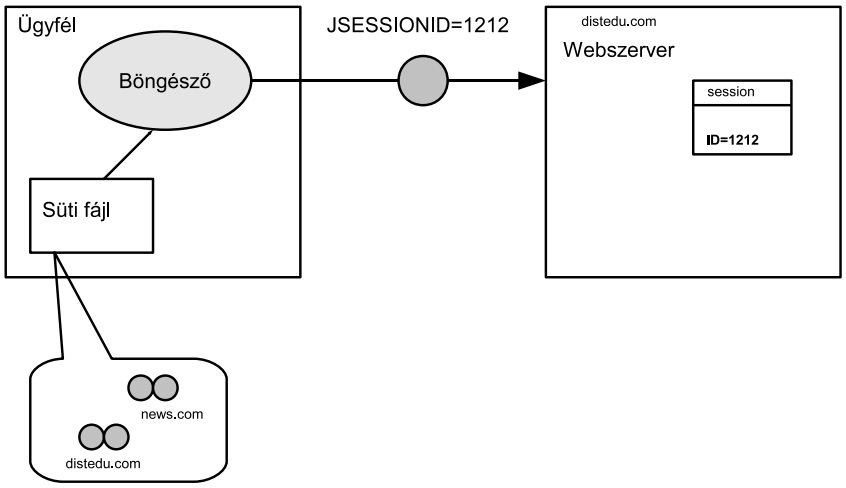
Azt láthattuk, hogy a menetazonosítót tartalmazó süti küldése automatikusan történik. Most nézzük meg, hogy egyéb információt hogyan küldhetünk sütiként. Tételezzük fel, hogy a felhasználó nevét akarjuk sütiként tárolni azért, hogy a következő alkalommal név szerint üdvözölhessük.

Süti küldése:

```
String name = request.getParameter("name");
Cookie c = new Cookie("yourname",name);
response.addCookie(c);
```

Süti fogadása:

```
String name;
Cookie[] cookies = request.getCookies();
for( int i=0; i<cookies.length; ++i ){
  if( cookies[i].getName().equals("yourname"))
    name = cookies[i].getValue();
}
```



3.4. ábra. *A süti visszaküldése a webszervernek*

Amikor egy szervletben meghívjuk a `getSession` metódust, a webkonténer a sütiből kinyert menetazonosító alapján előkeresi a menetobjektumot. Ezután tetszőleges számú és típusú attribútumot regisztrálhatunk meneti hatókörbe. Ezek az attribútumok egy gyors keresést biztosító tárolóban vannak elhelyezve a menetobjektumban.

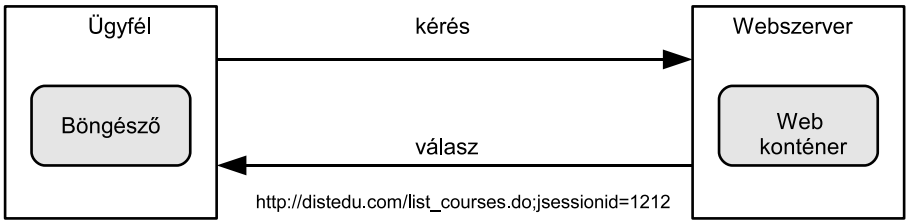
A sütik képezik az alapértelmezett menetkezelési stratégiát, a szervlet készítőjének ennek érdekében semmit sem kell tennie. A böngészők használói tolakodásnak tarthatják, hogy a gépükre bármilyen információ érkezzen sütik formájában, ezért letilthatják ezeket. Ebben az esetben más menetkezelési mechanizmusra van szükség. Lehetőségünk van lekérdezni, hogy a böngésző, amelytől a kérés érkezett, támogatja-e a sütiket. Erre az `isRequestedSessionIdFromCookie` metódus használható. Ha az alábbi kódrészletet elhelyezi egy szervletben, válaszként megtudja a kéréseket intéző böngészők beállításait. A szervlet csak a második kérésre ad helyes választ, hiszen az első kérésre kapja meg a webkonténertől a menetazonosítót tartalmazó sütit, így a második kérésben ezt már vissza is küldi a kiszolgálónak.

```
if( request.isRequestedSessionIdFromCookie()){
  out.println("<p>"+ "Az ön böngészője fogad sütiket!"+
              "</p>");
}
```

```
else{
   out.println("<p>"+ "Az ön böngészője NEM fogad sütiket!"+
               "</p>");
}
```

3.3. URL újraírás

Az URL újraírás egy alternatív menetkezelési mechanizmus. Először minden webkonténer megpróbál sütiket használni. Ha ez nem sikerül, akkor URL újraírást fog használni. Amint a neve is jelzi, itt tulajdonképpen az történik, hogy szerveroldalon átíródnak az URL-ek, kiegészítődnek a menetazonosítóval. Így minden egyes URL tartalmazni fogja a menet azonosítóját. Ezt szemlélteti a 3.5. ábra is.



3.5. ábra. *URL újraírás [7]*

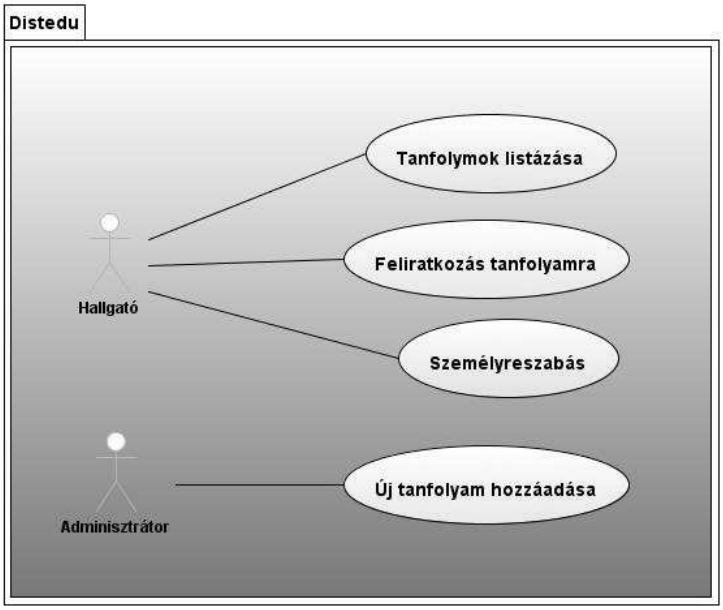
Azok a webalkalmazások, amelyek menetkezelést igényelnek, kell számoljanak azzal, hogy nem minden ügyfelük fogja megengedni a sütik fogadását. Ha ezeket az ügyfeleket is kezelni akarják, akkor az alkalmazást úgy kell elkészíteni, hogy amennyiben az ügyfél böngészője nem támogatja a sütiket, akkor URL újraírást valósítsanak meg. Az URL újraírás nem annyira áttetsző, mint a sütik, a programozóra is több munka hárul. A `HttpServletResponse` osztály `encodeURL` és `encodeRedirectURL` metódusai segítségével elvégezhetjük az URL újraírásokat. Ezen metódusoknak értelemszeűen átadjuk az átírandó URL-t, és visszatérítik az újraírt URL-t. Szervletben a következőképpen valósíthatjuk meg az URL újraírását:

```
out.println("<form
   action='"+response.encodeURL("add_item.do")+"'
   method ='POST'>");
```

3.4. Feladatok

3.4.1. Programozási feladatok

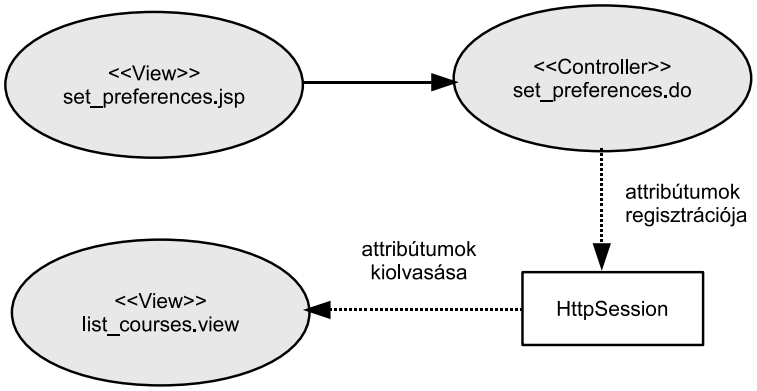
A *Szervletek* című fejezetben bevezetett távoktatási alkalmazásunkat egészítsük ki egy új funkcióval. Tegyük lehetővé, hogy minden hallgató maga állíthassa be, milyen adatok jelenjenek meg a tanfolyamokról, amikor ezeket listázza. Nyilván a tanfolyam neve kötelező módon megjelenik, de a leírás és ár már legyen választható. Kezdjük azzal, hogy kiegészítjük a használati eset diagramot. Az új változatot a 3.6. ábra szemlélteti.



3.6. ábra. *Distedu használati eset diagram*

A személyreszabás funkció egyelőre két logikai érték beállítását vezérli. Ezt űrlap segítségével tudjuk megvalósítani, a melyet egy `set_preferences.jsp` lapon helyezünk el. Az űrlapadatokat egy szervlet segítségével dolgozzuk fel, amelyhez a `set_preferences.do` URL-t rendeljük hozzá. Az űrlapadatokat a feldolgozó szervlet lementi a menet hatókörébe, így a menet során ezek alapján listázhatja felhasználónk a

tanfolyamokat. A komponensek közötti adatok áramlását szemlélteti a 3.7. ábra.



3.7. ábra. *Személyreszabás*

A `set_preferences.jsp` *lap törzse*

```
<h2>Course list preferences</h2>
<form action="set_preferences.do" method="GET">
  <input  type="checkbox" name="showName" value="true"
          checked disabled="true">Name<br><br>

  <input  type="checkbox" name="showDesc"
          value="true">Description<br><br>

  <input type="checkbox" name="showPrice"
         value="true">Price<br><br>

  <input type="submit"  value="Submit">
</form>
```

A fenti JSP lap csak HTML elemeket tartalmaz, amelyben az input elemet jelölőnégyzetre használjuk, ezt jelzi a `type=checkbox`.

A `SetPreferences` szervlet

```
package controller;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SetPreferences extends HttpServlet {

  @Override
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException {
    boolean showDesc = false, showPrice = false;
    HttpSession s = request.getSession();

    if (request.getParameter("showDesc") != null)
            showDesc = true;
    if (request.getParameter("showPrice") != null)
            showPrice = true;

    s.setAttribute("showDesc", showDesc);
    s.setAttribute("showPrice", showPrice);
  }
}
```

A szervlet elkéri az űrlap paramétereket és regisztrálja ezeket egy-egy logikai változóként a menet hatókörébe. A jelölőnégyzetek esetén a böngésző csak azokat küldi a kérésben, amelyeket bejelöltünk. Tehát ha csak az elsőt jelöljük be, akkor a kérés URL így néz ki:

```
http://localhost:8099/distedu/set_preferences.do?showDesc=true
```

Most pedig módosítanunk kell a tanfolyamok listázását végző szervletet és figyelembe kell vennünk a menet hatókörében tárolt listázási opciókat. Nyilván a listázás akkor is helyesen kell működjön, ha a felhasználó egyáltalán nem állít be listázási opciókat. Ilyenkor csak a tanfolyam nevét jelenítjük meg.

A ListCourses szervletben csak a doGet metódusban változtatunk, így csak a módosított részletet adjuk meg. Alapvetően abból áll a módosítás, hogy listázás előtt lekérjük a menet hatóköréből a listázási opciókat, amelyeket figyelembe veszünk a kiíratásnál.

```
HttpSession session = request.getSession();

boolean showDesc = false;
if( session.getAttribute("showDesc")!= null )
  showDesc = (Boolean) session.getAttribute("showDesc");

boolean showPrice = false;
if( session.getAttribute("showPrice")!= null )
  showPrice = (Boolean) session.getAttribute("showPrice");

while( it.hasNext()){
  out.println( "<li>");

    Course c = it.next();
    out.print(c.getName()+"\t");
    if( showDesc ){
      out.print(c.getDescription()+"\t");
    }
    if( showPrice ){
      out.print(c.getPrice()+"\t");
    }

  out.println( "</li>");
}
```

3.4.2. Tesztkérdések

3.1. kérdés: Válasszuk ki az egyetlen igaz állítást!

A. A menet attribútuma lehet primitív, illetve Object típusú is.
B. A menet attribútuma Object típusú.
C. A menet attribútuma csak szerializálható típusú lehet.
D. A menet hatókörébe attribútumot a setSessionAttribute metódussal regisztrálhatunk.

3.2. kérdés: Válasszuk ki a két igaz állítást!

A. A menet lejárati idejét csak a telepítésleíróban állíthatjuk be.
B. A menet lejárati idejét mind telepítésleíróban, mind pedig kódban beállíthatjuk.
C. A session-timeout telepítésleíró elem percben határozza meg a menet lejárati idejét.
D. A session-timeout telepítésleíró elem másodpercben határozza meg a menet lejárati idejét.

3.3. kérdés: Válasszuk ki az egyetlen igaz állítást!

A. Ha a böngészőben le van tiltva a sütik fogadása, akkor az URL újraírás automatikusan történik.
B. URL újraírást a HttpResponse rewriteURL metódusával végezhetünk.
C. URL újraírást a HttpResponse encodeURL metódusával végezhetünk.
D. URL újraírást a HttpSession encodeURL metódusával végezhetünk.

3.4. kérdés: Adott a következő részlet a `web.xml` telepítésleíróból:

```
<session-config>
   <session-timeout>300</session-timeout>
</session-config>
```

Mennyi lesz a menet lejárati ideje ebben a webalkalmazásban? (1 helyes)

A. 300 ezredmásodperc
B. 300 másodperc
C. 300 perc
D. 300 óra

3.5. kérdés: Egy szervlet `doGet` metódusa a következő kódsort tartalmazza:

```
String sid = request.getParameter("jsessionid");
```

A következő kódsorok közül melyek helyesek a `HttpSession` objektum lekérésére? (3 helyes)

A. request.getSession();
B. HttpSession.getSession(sid);
C. request.getSession(sid);
D. request.getSession(true);
E. request.getSession(false);

3.6. kérdés: Mikor lesz a menetobjektum végérvényesen megsemmisítve? (3 helyes)

A. A webkonténert leállítjuk és újraindítjuk.
B. Az utolsó kérés óta több idő telt el, mint a menet lejárati ideje (session timeout).
C. A szervlet meghívja az invalidate() metódust a menetobjektumra.
D. A menet lejárati idejét 0 értékre állítjuk a setMaxInactiveInterval() metódussal.

3.7. kérdés: Válassza ki az egyetlen igaz kijelentést!

A. URL újraírást úgy használhatunk, ha a telepítésleíróban megadjuk a következő sort:
<url-rewriting>true</url-rewriting>
B. URL újraírást a válaszobjektum encodeURL metódusával végezhetünk.
C. A ,,jsessionid'' nevű süti (cookie) használható menetazonosításra, hiszen az ügyfél böngészője tárolja ezt, és akárhányszor az ügyfél visszaküldi a sütit, megvalósítja egyben a bejelentkezést is.

3.8. kérdés: Adott egy webalkalmazás a következő két szervlettel. Feltételezve, hogy a `loginUser` és `generateReport` érvényes metódusok, a következő kijelentések közül válasszuk ki az egyetlen helyeset!

```
//In file LoginServlet.java
public class LoginServlet extends HttpServlet
{
  public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
  {
    String userid = loginUser(req);
    req.getSession().setAttribute("userid", userid);
```

```
  }
}

//In file ReportServlet.java
public class ReportServlet extends HttpServlet
{
  public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
                     throws IOException
  {
     String userid = (String) req.getSession().
      getAttribute("userid");
     if(userid != null) generateReport(req, res);
  }
}
```

A. A ReportServlet.java fordítási hibát ad.
B. A generateReport() soha nem hajtódik végre.
C. A generateReport() metódus csak akkor hajtódik végre, ha a LoginServlethez a ReportServlet előtt érkezik egy HTTP POST kérés.
D. A web.xml telepítésleíróban a share-session tulajdonságot true értékűre kell állítani ahhoz, hogy a ReportServlet hozzáférjen a userid attribútumhoz.
E. Egyik sem a fentiek közül.

3.9. kérdés: Válasszuk ki azt a sort a következők közül, amely egy szervlet doPost() metódusában elhelyezve URL újraírást végez! (1 helyes)

```
A. request.useURLRewriting();
B. out.println( response.rewrite("<a href='/servlet/XServlet'>
                        Click here</a>"));
C. out.println("<a href=' "+request.rewrite(
                    "/servlet/XServlet")+"'>Click here</a>"));
D. out.println("<a href=' "+response.encodeURL(
                   "/servlet/XServlet")+" '>Click here</a>"));
E. out.println("<a href=' "+request.encodeURL(
                   "/servlet/XServlet")+" '>Click here</a>"));
```

3.10. kérdés: Egy webalkalmazás azonosítja felhasználóit. Az azonosítás pillanatában létrejön minden egyes felhasználónak egy új menet. Azt szeretnénk, hogy maximum 20 percig legyen érvényes a menet, utána automatikusan kelljen újra bejelentkezni. A következő `HttpSession` metódusok közül melyiket lehetne használni erre a célra? (1 helyes)

A. `getMaxInactiveInterval()`
B. `getMaxActiveInterval()`
C. `getLastAccessTime()`
D. `getLastAccessedTime()`
E. `getCreationTime()`

3.11. kérdés: Mit nem lehet menetkezelésre (menet fenntartására) használni? (1 helyes)

A. `HttpSession objektum`
B. `URL újraírás`
C. `Sütik (Cookies)`
D. `HttpSessionActivationListener`
E. `Rejtett űrlapmezők`

3.12. kérdés: A következő kódsorok közül melyik teszi állandó érvényűvé a menetobjektumot? (1 helyes)

A. `setTimeOut(-1)`
B. `setTimeOut(Integer.MAX_INT)`
C. `setTimeOut(0)`
D. `setMaxInactiveInterval(-1)`
E. `setMaxInactiveInterval(Integer.MAX_INT)`

3.13. kérdés: Hogyan lehet expliciten megsemmisíteni a menetobjektumot? (1 helyes)

A. `Nem lehetséges.`
B. `Az invalidate() metódus meghívásával.`
C. `Az expunge() metódus meghívásával.`
D. `A delete() metódus meghívásával.`
E. `A finalize() metódus meghívásával.`

3.14. kérdés: Adott egy webalkalmazás az alábbi két szervlettel. Válasszuk ki az egyetlen igaz kijelentést!

```
//File: LoginServlet.java
public class LoginServlet extends HttpServlet
{
   public void doPost(HttpServletRequest req,
                      HttpServletResponse res)
   {
      String userid = loginUser(req);
      req.getSession().setAttribute("userid",userid);
   }
}
```

```
//File: ReportServlet.java
public class ReportServlet extends HttpServlet
{
  public void doPost(HttpServletRequest req,
    HttpServletResponse res) throws IOException
  {
     String userid = req.getSession().
       getAttribute("userid");
     if(userid != null)
       generateReport(req, res);
  }
}
```

A. A LoginServlet.java fordítási hibát ad.
B. A ReportServlet.java fordítási hibát ad.
C. A LoginServlet kivételt dob.
D. A ReportServlet kivételt dob.
E. Mindekttő hibátlanul fordítható és futtatható.

3.15. kérdés: Adott egy menet hatókörében tárolt `interestRate` nevű és `Double` típusú attribútum. Feltételezve, hogy a `session` egy érvényes referencia a menetobjektumra, mely kódsor használható az attribútum lekérésére? (1 helyes)

A. `session.getObject("interestRate");`
B. `session.get("interestRate");`
C. `(Double)session.getAttribute("interestRate");`
D. `session.getDouble("interestRate");`
E. `session.getDoubleAttribute("interestRate");`

4. FEJEZET

ESEMÉNYKEZELŐK ÉS SZŰRŐK

4.1. Eseménykezelők

Mivel a webkonténerek kezelik a webkomponensek életciklusát, néha hasznos lehet, ha az életciklushoz kapcsolódó események figyelését is rájuk bízhatjuk. Ennek a problémának a megoldására vezették be a 2.4-es szervletspecifikációban az eseménykezelőket. A következő objektumokhoz lehet eseményfigyelőt illeszteni:

- kérés
- kérés szintű attribútum
- munkamenet
- munkamenet szintű attribútum
- munkamenet aktiválása
- webalkalmazás
- webalkalmazás szintű attribútum

A fenti objektumok figyelését a következő interfészek biztosítják:

- `javax.servlet.ServletRequestListener`
- `javax.servlet.ServletRequestAttributeListener`
- `javax.servlet.http.HttpSessionListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.servlet.http.HttpSessionActivationListener`
- `javax.servlet.ServletContextListener`
- `javax.servlet.ServletContextAttributeListener`

A `HttpSessionActivation` figyelőnek csak akkor van értelme, ha a munkamenet egy osztott webalkalmazáshoz tartozik, és lehetőség van a munkamenet átvitelére egy másik Java virtuális gépre. Ezt a figyelőt nem kell konfigurálni a telepítésleíróban, az összes többi felsoroltat viszont igen.

Terjedelmi okok miatt arra nincs lehetőség, hogy minden egyes figyelő interfész használatára példát adjunk. A figyelő komponensek szemléltetésére két példát fogunk adni. Ezek jól szemléltetik az ilyen típusú komponensek szerepét, majd ennek alapján, és felhasználva a Servlet API dokumentációt, képesek leszünk más figyelők készítésére.

Az első példánkat kössük a távoktatás alkalmazásunkhoz. Emlékezzünk vissza, hogy a tanfolyamok inicializálását nem a legelegánsabban végeztük. Bár az alkalmazás minden komponense használhatja a kurzuslistát, mi mégis egyetlen szervlethez kötöttük ennek inicializálását. Most viszont megvan a lehetőségünk, hogy kijavítsuk hibánkat. Tehát a tanfolyamok listája egy alkalmazás szintű változó, így ennek kezelése az alkalmazáshoz, és nem pedig valamely komponensének életciklusához kötődik.

A következő lépéseket kell végrehajtanunk:

- hozzáadunk az alkalmazáshoz egy figyelő típusú komponenst, a mi esetünkben ez a `ServletContextListener` interfészt implementálja,
- módosítjuk a `ListCourses` szervlet osztályunkat, kitöröljük az `init`, illetve a `destroy` metódusokat.

Az új figyelőosztályunkba átmentjük a `ListCourses` szervlet `init`, illetve a `destroy` metódusok tartalmát, a megfelelő metódusokba. A `ServletContextListener` interfész két metódusdeklarációt tartalmaz:

- `void contextInitialized(ServletContextEvent sce)`
- `void contextDestroyed(ServletContextEvent sce)`

A `contextInitialized` metódus a webalkalmazás telepítése után rögtön végrehajtódik, tehát még a kérések kiszolgálása előtt. A `contextDestroyed` metódus pedig a webalkalmazás leállítása előtt hajtódik végre. Mind az inicializáló, mind a megsemmisítő metódus csak egyszer hajtódik végre egy webalkalmazás életciklusában.

Ha NetBeans fejlesztői környezetet használunk, akkor nagyon könnyű dolgunk van. Minden egyes webkomponens típusra van egy egy sablon, így nekünk leggyakrabban csak tartalommal kell kitöltenünk a metódusokat, a vázat készen kapjuk. Egy másik előnye az integrált környezet használatának az, hogy amikor egy új komponenst adunk hozzá a webalkalmazáshoz, és ez telepítésleíró konfigurációt igényel, ezt ismét automatikusan előállítja a rendszer.

Készítsünk egy `ApplicationListener` nevű webalkalmazás figyelőt, majd másoljuk át a `ListCourses.java` szervlet `init` metódusának tartalmát a `contextInitialised` metódusba, illetve a `ListCourses.java` szervlet `destroy` metódusának tartalmát a `contexDestroyed` metódusba.

Még egy apró módosítást kell végeznünk a `contextInitialised` és a `contextDestroyed` metódusokban. Másképpen férünk hozzá az alkalmazási hatókörhöz, mint szervletekben:

– szervlet esetében az alkalmazás hatóköre így érhető el: `this.getServletContext()`
– figyelő esetében az eseményobjektumból fogjuk kinyerni az alkalmazás hatókörét: `sce.getServletContext()`

ApplicationListener.java

```
package listeners;

import javax.servlet.*;
import java.io.*;
import java.util.*;
import model.*;

public class ApplicationListener
             implements ServletContextListener {

  public void contextInitialized(ServletContextEvent sce) {

    List<Course> courselist = new ArrayList();

    String resource= "/WEB-INF/tanfolyamok.txt";
    InputStream is=
          sce.getServletContext().
          getResourceAsStream(resource);
    BufferedReader br = new BufferedReader(
                new InputStreamReader(is));
    while( true ){
      String line = null;
      try{
        line = br.readLine();
        if( line == null ) break;
        StringTokenizer stk =
               new StringTokenizer(line,"#");
        Course course = new Course();
        course.setName(stk.nextToken());
        course.setDescription(stk.nextToken());
        course.setPrice(
          Double.parseDouble(stk.nextToken()));
        courselist.add( course );
      }
      catch( IOException e){
               e.printStackTrace();;
```

```
      }
    }
    sce.getServletContext().
      setAttribute("coursecounter", courselist.size());
    sce.getServletContext().
      setAttribute("courselist", courselist);
    try{
      br.close();
    }
    catch( Exception e){
      e.printStackTrace();
    }
 }

 public void contextDestroyed(ServletContextEvent sce) {
   int coursecounter=(Integer)
     sce.getServletContext().getAttribute("coursecounter");
   List<Course> courselist = (List<Course>)
   sce.getServletContext().getAttribute("courselist");
   if( coursecounter != courselist.size()){
       String resource= "/WEB-INF/tanfolyamok.txt";
       String path = sce.getServletContext().
         getRealPath(resource);
       System.out.println("destroy-PATH: "+path);
       try{
         PrintWriter pw = new PrintWriter(
           new FileWriter(path) );
         Iterator<Course> it = courselist.iterator();
         while( it.hasNext() )
           pw.println( it.next());
         pw.close();
       }
       catch(Exception e){
         e.printStackTrace();
       }
   }
 }
}
```

A figyelőobjektumot a telepítésleíróban (`web.xml`) a következőképpen konfigurálhatjuk:

```
<listener>
```

```
  <description>ServletContextListener</description>
  <listener-class>
    listeners.ApplicationListener
  </listener-class>
</listener>
```

A webkonténer az alkalmazás betöltésekor meghatározott sorrendben hozza létre a telepítésleíróban megadott webkomponenseket. Az eddig ismertetett webkomponesek közül a szervleteket és az eseményfigyelőket kötelező telepítésleíróban konfigurálni. Először az eseményfigyelők példányosítása történik meg, és ezután következnek a szervletek. Az eseményfigyelők a deklarálás sorrendjében kapják meg az eseményeket. Ha két eseményfigyelő függ egymástól, akkor ezt a megfelelő sorrendben kell konfigurálni.

Most pedig készítsünk egy másik eseményfigyelőt, amely nyilvántartja a munkamenetek számát [5]. Ebben az esetben nem az alkalmazás életciklusának eseményeire kell figyelnünk, hanem a munkamentekre. Erre a Servlet API a `HttpSessionListener` interfészt biztosítja, ezt az interfészt kell implementálnunk.

```
package listeners;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class SessionCounterListener
       implements HttpSessionListener{
    private static int sessionCounter = 0;

    public void sessionCreated(HttpSessionEvent arg0) {
        sessionCounter++;
    }
    public void sessionDestroyed(HttpSessionEvent arg0) {
        sessionCounter--;
    }
    public static int getSessionNumber() {
        return sessionCounter;
    }
}
```

A figyelőt konfigurálni kell a `web.xml` telepítésleíróban. Ennek hatására az alkalmazás telepítésekor a webkonténer még a szervletek példányosítása előtt létrehozza az összes figyelő objektumot. A figyelők

regisztrálásához a `listener` tagot használjuk, amelyben opcionálisan megadhatunk egy leírást is, és kötelező módon meg kell adnunk a figyelő osztályának a nevét.

```
<listener>
  <description>HttpSessionListener</description>
  <listener-class>
    listeners.SessionCounterListener
  </listener-class>
</listener>
```

Most pedig használjuk a menetszámláló osztályunkat és jelenítsük meg az alkalmazáshoz tartozó menetek számát. Ehhez az szükséges, hogy például az `index.jsp` lapon helyezzük el a következő kódrészletet:

```
<h1> Menetek száma:
<%=listeners.SessionCounterListener.getSessionNumber()%>
</h1>
```

A fenti implementáció tökéletesen működik, amennyiben nincs egyidejű hozzáférés a figyelő objektumhoz. Valós alkalmazásokban viszont nagyon gyakran előfordul az ilyesmi, ezért ha a valós menetek számát akarjuk nyilvántartani, akkor szálbiztosra kell készítenünk a figyelőt. Ezt szemlélteti a következő kódrészlet:

```
package listeners;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class SessionCounterListener
       implements HttpSessionListener {
    private static int sessionCounter = 0;
    public void sessionCreated(HttpSessionEvent arg0) {
        synchronized( this ){
            sessionCounter++;
        }
    }
    public void sessionDestroyed(HttpSessionEvent arg0) {
        synchronized( this ){
            sessionCounter--;
        }
    }
```

```
    public static int getSessionNumber(){
        return sessionCounter;
    }
}
```

4.2. Szűrők

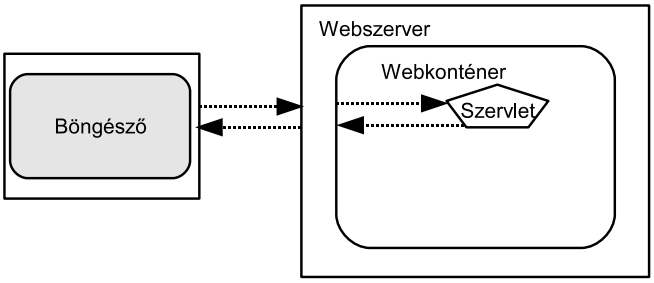
A HTTP kérések mindig a webkonténerhez érkeznek, és csak utána adódnak át a megfelelő webkomponensnek. Ez lehetőséget biztosít a webkonténernek, hogy előzetes feldolgozást végezzen a kérésen. A HTTP válaszok esetén is hasonló a helyzet. A választ előállító webkomponens a választ először a webkonténernek továbbítja, majd ez juttatja el az ügyfél böngészőjéhez. Így nemcsak elő-, hanem utófeldolgozásra is lehetőségünk van. Tipikus példa előfeldolgozásra, amikor bizonyos weblapok a felhasználó hitelesítését igénylik. Ilyen esetben szükséges a kérés kiszolgálása elé beiktatni a hitelesítést végző komponenst, amely majd továbbítja a kérést a weblaphoz. Az ilyen tevékenységek végzéséhez vezették be a szűrőket. A szűrők azért is fontosak, mert az ismétlődő tevékenységeket újrafelhasználható formában kínálják.

A leggyakoribb tevékenységek, amelyekre ajánlott szűrőt használni:

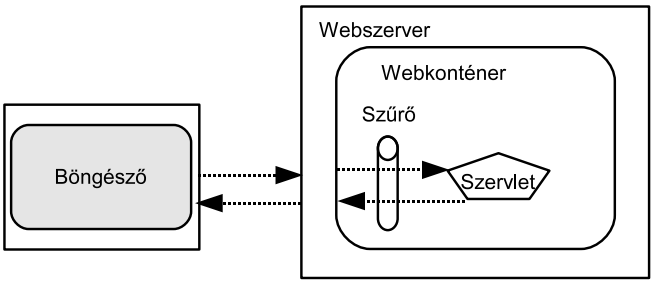
- jogosultságellenőrzés,
- szervletek hatékonyságának mérése és naplózása,
- a válasz tömörítése,
- lokalizáció.

Fontos megérteni a szűrők működési elvét, ezért először tekintsük át a webkonténer szerepét a kérésfeldolgozásban, majd nézzük meg, hogyan változik ez meg, ha szűrőt helyezünk a webkomponens elé. A kérésfeldolgozás szűrő nélküli, illetve a szűrőt használó változatát a 4.1. ábra, illetve a 4.2. ábra szemlélteti.

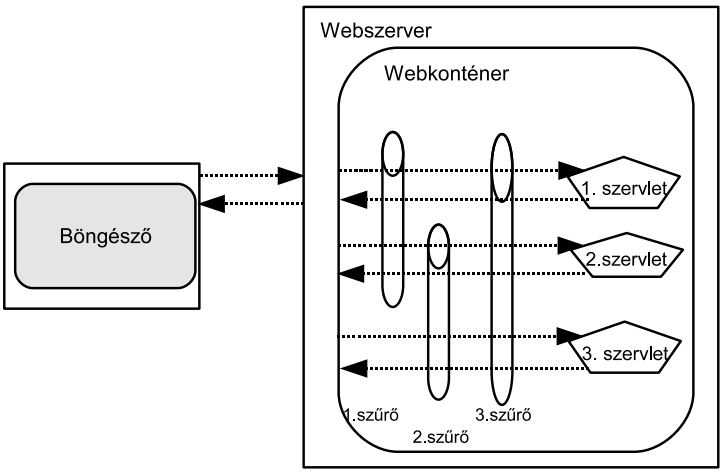
A kérés mindig először a webkonténerhez érkezik, amely előfeldolgozást végezhet a kérésen. Ezután a webkonténer ellenőrzi a kérés URL alapján, hogy kell-e szűrőt alkalmaznia a kérésre. Az URL mintára illeszkedő szűrőket a webkonténer egymás után alkalmazza a kérésre. Ha mindez hibamentes volt, akkor a szűrők után átkerül a kérés a cél webkomponenshez, a mi esetünkben ez most egy szervlet. A szervlet az előállított választ a legutolsóként alkalmazott szűrőhöz továbbítja,



4.1. ábra. *Webkonténer és kérésfeldolgozás [7]*



4.2. ábra. *Webkonténer és kérésfeldolgozás szűrő esetén [7]*



4.3. ábra. *Szűrőkonfiguráció [7]*

majd végigmegy fordított sorrendben a szűrőkön, és végül visszakerül a webkonténerhez.

A szűrők konfigurálása lehetővé teszi, hogy pontosan meghatározzuk, melyik szűrőt milyen esetben kell alkalmazni. Egy érdekesebb szűrőkonfigurációt szemléltet a 4.3. ábra.

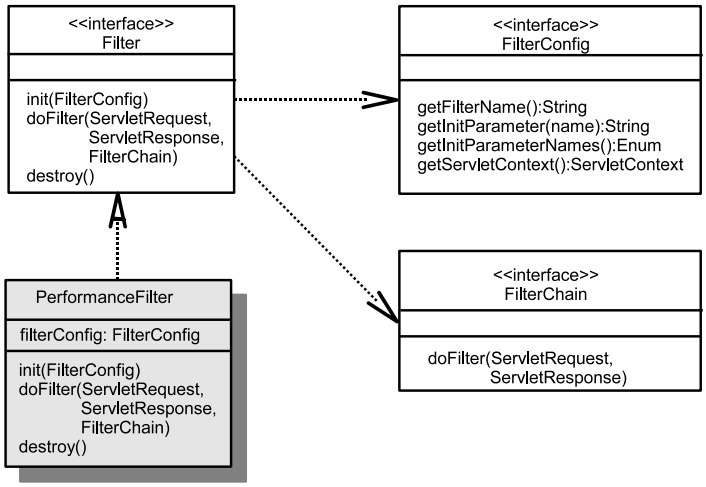
A szűrőket nem csak webkomponens elé lehet beilleszteni, hanem lehetőség van arra is, hogy valamely webkomponens szűrőhöz továbbítsa a kérést. Nyilván ezt a lehetőséget ritkábban szokás használni.

4.2.1. Filter API

A szűrőket a szervlet specifikáció 2.3-as változatában vezették be. Az API-hoz tartozó interfészek a `javax.servlet` csomagban találhatók, ezek közül a három legfontosabbat a 4.4. ábra szemlélteti.

A következő szűrő a kérés paramétereit naplózza:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import java.util.logging.Logger;

public class LogRequestParametersFilter implements Filter {
```



4.4. ábra. *Filter API [7]*

```
private FilterConfig filterConfig = null;

public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
    throws IOException, ServletException {
  Logger log = Logger.getLogger("Request parameters:");
  Enumeration parameters = request.getParameterNames();
  while( parameters.hasMoreElements()){
     String p = (String)parameters.nextElement();
     log.info(p+": ");
     String[] ps = request.getParameterValues(p);
     for( int i=0; i<ps.length; ++i )
        log.info("\t"+ps[i]);
  }
  chain.doFilter( request,response);
}

public void destroy() {
}

public void init(FilterConfig filterConfig) {
   this.filterConfig = filterConfig;
```

```
    }
}
```

A szűrőt a telepítésleíróban konfigurálni kell. A szűrők konfigurálása a szervletekéhez hasonló, van egy `<filter>` és egy `<filter-mapping>` tag. Az utóbbival határozzuk meg, hogy mikor kell alkalmazni a szűrőt. Amennyiben a fenti szűrőt minden kérés elé be szeretnénk helyezni, akkor a következőket kell elhelyeznünk a `web.xml` telepítésleíróban:

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogRequestParametersFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

A szűrők leképzése kétféleképpen történhet: szervletre, illetve URL mintára. A szervletre való leképzés azt jelenti, hogy a szervlet előtt mindig le fog futni a szűrő. Az URL mintára való leképzés esetében minden egyes alkalommal, amikor a kérés egyezik a mintával, le fog futni a szűrő. A szűrők a web.xml telepítésleíróban megadott deklarációs sorrendben futnak le, de az URL leképzésű szűrők előnyt élveznek a szervlet leképzésűekkel szemben.

- Szervlet leképzés:

  ```
  <filter-mapping>
     <filter-name>PerfFilter</filter-name>
     <servlet-name>FrontController</servlet-name>
  </filter-mapping>
  ```

- URL leképzés:

  ```
  <filter-mapping>
     <filter-name>PerfFilter</filter-name>
     <url-pattern>/*.do</url-pattern>
  </filter-mapping>
  ```

A következő szűrő meghívódik minden egyes tevékenységre és méri annak időigényét.

```
package filters;

import javax.servlet.*;
import java.io.*;

public class PerformanceFilter implements Filter {
    private FilterConfig filterConfig = null;

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

        long begin = System.currentTimeMillis();
        chain.doFilter(request, response);
        long end = System.currentTimeMillis();

        filterConfig.getServletContext().log(
         "Ellapsed time: "+(end-begin)+" ms");
    }

    public void destroy() {
    }

    public void init(FilterConfig filterConfig) {
       this.filterConfig = filterConfig;
    }
}
```

A szűrők esetében is, akárcsak a szervleteknél, az `init` és a `destroy` metódus csak egyszer hajtódik végre a szűrő életciklusa során. A `doFilter` pedig minden egyes olyan kérésre, amelyre a szűrő meghívódik. A `doFilter` metódusnak három paramétere van, az első kettő a kérés-, illetve a válaszobjektum, a harmadik `FilterChain` típusú, a kérés továbbítására szolgál. Ha a szűrő blokkolja a kérést, akkor ennek a harmadik paraméternek nincs további jelentősége.

A fenti szűrőt a következőképpen konfigurálhatjuk:

```
<filter>
   <filter-name>PerformanceFilter</filter-name>
   <filter-class>filters.PerformanceFilter</filter-class>
```

```
</filter>

<filter-mapping>
   <filter-name>PerformanceFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Hogy jobban megértsük a szűrők végrehajtási sorrendjét, oldjuk meg a következő feladatot:

4.1. feladat. Szűrők: Adott a következő telepítésleíró részlet:

```
<servlet-mapping>
   <servlet-name>FrontController</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>

<filter-mapping>
   <filter-name>perfFilter</filter-name>
   <servlet-name>FrontController</servlet-name>
</filter-mapping>

<filter-mapping>
   <filter-name>auditFilter</filter-name>
   <url-pattern>*.do</url-patern>
</filter-mapping>

<filter-mapping>
   <filter-name>transFilter</filter-name>
   <url-pattern>*.do</url-patern>
</filter-mapping>
```

Legyen a kérés URL: /admin/add_league.do. Mely szűrők és milyen sorrendben hívódnak meg?
Válasz:

1. auditFilter
2. transFilter
3. perfFilter

Az eddig bevezetett szűrőket a klienstől érkező kérésekre alkalmaztuk. Lehetőség van szűrőt használni akkor is, ha a komponenshez a vezérlés `include` vagy `forward` hívással kerül. Ezt ismét a telepítésleíróban állítjuk be, `dispatcher` tag segítségével:

- `<dispatcher>REQUEST</dispatcher>` azt jelenti, hogy a szűrő csak klienstől érkező kérés esetében lesz alkalmazva,
- `<dispatcher>INCLUDE</dispatcher>` azt jelenti, hogy a szűrő csak `RequestDispatcher.include` hívás esetében lesz alkalmazva,
- `<dispatcher>FORWARD</dispatcher>` azt jelenti, hogy a szűrő csak `RequestDispatcher.forward` hívás esetében lesz alkalmazva,
- `<dispatcher>ERROR </dispatcher>` azt jelenti, hogy a szűrő csak hiba esetében lesz alkalmazva.

Például a következő szűrő csak belső vezérlésátadás esetében lesz alkalmazva:

```
<filter-mapping>
   <filter-name>myFilter</filter-name>
   <url-pattern>*.do</url-patern>
   <dispatcher>INCLUDE</dispatcher>
   <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

4.3. Tesztkérdések

4.1. kérdés: Válasszuk ki azokat az XML kódrészleteket, amelyek helyesen konfigurálnak egy eseményfigyelőt a telepítésleíróban! (1 helyes)

```
A. <contextlistener>
      <listener-class>mypackage.SomeListener</listener-class>
   </contextlistener>

B. <listener>
      <listener-class>mypackage.SomeListener</listener-class>
   </listener>

C. <context-listener>
      <listener-class>mypackage.SomeListener</listener-class>
   </context-listener>
```

```
D. <listener>
      <class>mypackage.SomeListener</class>
   </listener>
```

4.2. kérdés: Válasszuk ki a helyes kijelentéseket! (3 helyes)

```
A. A szűrő init metódusa csak egyszer hívódik meg a szűrő
   életciklusa során.
B. A doFilter metódusban a szűrőnek át kell adnia
   a vezérlést a hozzárendelt erőforrásnak.
C. Ha egy erőforráshoz több szűrő is hozzá van rendelve,
   akkor ezek hívási sorrendjét a telepítésleíróban
   megadott sorrend határozza meg.
D. Ha egy szűrő UnavailableException kivételt dob az init
   metódusban, többé nem hívódik meg.
E. Minden egyes telepítésleíróban deklarált szűrőből csak
   egy példányt hoz létre a webkonténer.
```

4.3. kérdés: Adott egy `PreferenceServlet` szervlet, amely a felhasználó azonosítója (userid) alapján megjeleníti a felhasználói beállításokat. A következő alakú URL szolgál a hívására:

```
/myapp/prefservlet?userid=laci
```

A szervlethez érkező minden kérés esetén naplózni kell a felhasználó azonosítóját. Melyik a legelegánsabb megoldás erre a problémára? (1 helyes)

```
A. Módosítsuk a szervletet és ebben végezzük el a
   naplózást.
B. Használjuk a HTTP szerver naplóját a kérések
   követésére.
C. Rendeljünk egy szűrőt a szervlethez és ebben végezzük
   a naplózást.
D. Rendeljük egy szűrőt a webalkalmazáshoz és ebben
   végezzük a naplózást.
```

4.4. kérdés: Melyik metódus hívódik az alkalmazás indításakor? (1 helyes)

A. ServletContextListener objektum contextInitialized() metódusa
B. ServletContextListener objektum contextCreated() metódusa
C. ServletContextListener objektum contextStateChanged() metódsa
D. ServletContextListener objektum init() metódusa
E. ServletContextListener objektum initialized() metódus

4.5. kérdés: Válasszuk ki a helyes kijelentéseket! (2 helyes)

A. Egy telepítésleíróban több listener típusú osztályt lehet konfigurálni.
B. Egy listener osztály több típusú eseménynek is lehet figyelője.
C. Egy listener osztály nem implemetálhat egynél több figyelőinterfészt.
D. Egyik sem az előbbiek közül.

4.6. kérdés: Adott a következő két osztály, amelyek a nevükben megadott figyelőinterfészeket implementálják. Melyik telepítésleíró részlet konfigurálja helyesen a két figyelőt? (1 helyes)

A.
```
<listener>
   <listener-class>
      MyServletContextAttributesListener
   </listener-class>
   <listener-class>
      MyHttpSessionListener
   </listener-class>
</listener>
```

B.
```
<listener-class>
   MyServletContextAttributesListener
</listener-class>
<listener-class>
   MyHttpSessionListener
</listener-class>
```

C.
```
<listener>
   <listener-class>
      MyServletContextAttributesListener
   </listener-class>
</listener>
<listener>
   <listener-class>
      MyHttpSessionListener
   </listener-class>
</listener>
```

D.
```
<context-listener>
   <listener-class>
      MyServletContextAttributesListener
   </listener-class>
</context-listener>
<session-listener>
   <listener-class>
      MyHttpSessionListener
   </listener-class>
</session-listener>
```

E.
```
<listener>
   MyServletContextAttributesListener
</listener>
<listener>
   MyHttpSessionListener
</listener>
```

5. FEJEZET

JSP TECHNOLÓGIA

5.1. JSP alapfogalmak

Mielőtt a JSP technológiát ismertetnénk, hasonlítsuk össze az ugyanazon kimenetet előállító szervletet és JSP lapot [7]. Egy olyan szerveroldali webkomponenst készítünk, amely egy üdvözlő szöveget állít elő a kérésben megadott név alapján, majd ezt visszaküldi a böngészőnek. Az üdvözlő szöveg a `Hello` szó és utána a paraméterként kapott név. Mindkét webkomponenst úgy készítjük el, hogy amennyiben a kérésben nincs név paraméter, akkor egy alapértelmezett szöveget fogunk visszaküldeni.

A szervlet

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorldServlet extends HttpServlet {

    private static final String DEFNAME ="Vilag";

    protected void generateResponse
    (HttpServletRequest request,
     HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String name = request.getParameter("name");
        if( name == null || name.length() == 0 )
            name = DEFNAME;
        out.println("Hello, "+name+"!");
        out.close();
    }

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

```
    throws ServletException, IOException {
        generateResponse(request, response);
    }

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

A szervlethez hozzárendeljük a `hello.view` URL-t (web.xml telepítésleíróban):

```
<servlet-mapping>
   <servlet-name>HelloWorldServlet</servlet-name>
   <url-pattern>/hello.view</url-pattern>
</servlet-mapping>
```

A szervlet hívását úgy végezzük, hogy a böngésző címsorába begépeljük a következő sort. Ez a sor egy HTTP GET kérést fog kiváltani, amelyben nemcsak az erőforrást azonosítjuk, hanem a `name` nevű paramétert is elküldjük a szervletnek.

```
http://localhost:13585/HelloWorldServlet/hello.view?name=Kati
```

A fenti URL-ben a `HelloWorldServlet` a webalkalmazás neve, előtte pedig egy port száma található. Ez a szám attól függ, hogy milyen porton fut a webkonténer (alkalmazásszerver). Ha NetBeans környezetben fejlesztünk, akkor webalkalmazás futtatásakor megjelenik az URL, ahol elérhető az adott webalkalmazás.

A szervlet kódja elég hosszú, attól függetlenül, hogy a mai modern fejlesztőeszközök ezt a típusú szervletvázat már automtikusan előállítják (pl. NetBeans).

A JSP lap

```
<%! private static final String DEFNAME = "vilag"; %>
<html>
    <head>
        <title>JSP Page</title>
```

```
    </head>
    <body>
        <%
           String name = request.getParameter("name");
           if( name == null || name.length() == 0 )
              name = DEFNAME;
        %>
        <h2>Hello,  <%= name %> !</h2>
    </body>
</html>
```

A JSP lap hívását pedig így is végezhetjük:

```
http://localhost:13585/HelloWorldServlet/hello.jsp?name=Kati
```

A JSP – JavaServer Pages szerveroldali, Java alapú technológia, amelynek fő célja felváltani a szervleteket, amikor ezekkel megjelenítést végzünk. A szervletekkel foglalkozó fejezetben láthattuk, hogy mennyire kényelmetlen HTML tartalmat előállítani szervlet segítségével. Még programozónak is nehéz feladat, így nem várhatjuk el, hogy egy weblapkarbantartó könnyedén módosítsa az így előállított tartalmat. A JSP lapok bevezetésének éppen ez az egyik célja, hogy HTML készítésében jártas szakembereknek lehetőséget biztosítson dinamikus tartalom előállítására. Amíg a szervletet úgy tekinthetjük, mint Java kódba ágyazott HTML, addig a JSP lesz a HTML-be ágyazott Java kód. A dinamikus tartalmat úgynevezett JSP elemek fogják előállítani, amelyek jelölésben hasonlítanak a HTML elemekhez. A JSP nem helyettesíti a szervleteket, sőt, minden JSP lapból egy szervlet állítódik elő, és a háttérben ez fogja a dinamikus tartalmat előállítani. Ebből következik, hogy JSP lapokkal is megoldhatók ugyanazok a feladatok, mint szervletekkel. Nyilván minden egyes feladathoz a megfelelő Java webkomponenst (szervlet, illetve JSP lap) fogjuk választani.

Amikor kérés érkezik egy JSP laphoz, akkor ezt a JSP lapból előállított szervlet fogja feldolgozni. Amennyiben a JSP lap újabb, mint a neki megfelelő szervlet, akkor a webkonténer újra elvégzi a lap szervletté alakítását, lefordítja ezt és a kérést is továbbítja ennek a szervletnek. A JSP lapból előállított szervletosztály implementál egy `HttpJspPage` interfészt, ezért minden egyes JSP lapból készített szervlet tartalmazza a következő metódusokat:

- `jspInit`
- `_jspService(HttpServletRequest, HttpServletResponse)`

– jspDestroy

A jspInit metódus a Servlet interfész init metódusához hasonló, ez csak egyszer hajtódik végre a komponens betöltésekor. A jspDestroy az ezzel ellentétes művelet, ez a komponens eltávolításakor kerül végrehajtásra. A _jspService meghívódik minden egyes kérésre, amely a JSP komponenshez érkezik. Láthatjuk, hogy a működési mechanizmus a szervleteknél ismertetettel egyezik meg. Most pedig kövessük végig egy JSP lappal történő műveleteket:

1. JSP lap átalakítása (transzlációja) szervletté: foo.jsp → foo_jsp.java Ez a lépés minden esetben megtörténik, ha módosítottuk a JSP lapot, vagyis a JSP lap újabb, mint a neki megfelelő szervlet. Egy foo.jsp nevű lapból mindig egy foo_jsp.java nevű állományt fog előállítani.
2. a szervlet fordítása: foo_jsp.java → foo_jsp.class
3. a foo_jsp.class fájl betöltése
4. a foo_jsp osztály példányosítása
5. a jspInit metódus meghívása a létrehozott példányra
6. HTTP kérések kiszolgálása: _jspService metódus hívásával (ez a lépés annyiszor ismétlődik, ahányszor kérés érkezik a JSP laphoz)
7. a jspDestroy metódus hívása a komponens eltávolításakor

A JSP lapok telepítése ugyanúgy történik, mint a statikus HTML lapoké, egyszerűen bemásoljuk a telepített alkalmazás könyvtárába.

5.2. JSP szkriptelemek

A JSP lapba ágyazott szkriptelemeket (< % % >) a JSP motor dolgozza fel a JSP lap szervletté alakításakor. A szkriptelemeken kívüli részt szintén a JSP motor veszi át, módosítás nélkül, hiszen ennek bele kell kerülnie a válaszba. A lapnak azon részét, amely nem JSP elem, sablonszövegnek nevezik. Ez lehet közönséges szöveg, HTML, WML vagy XML is. A tartalom előállítása szerveroldalon úgy történik, hogy a szerver a sablonszöveghez hozzáilleszti a JSP elemek által előállított dinamikus tartalmat, majd ezt visszaküldi a böngészőnek.

Bár a szkriptelemek kerülendők a JSP lapok készítésekor, a teljesség kedvéért ezeket is ismertetjük. Gyakorlatilag azért nem ajánlott szkriptelemeket használni, mert ezek megértése Java programozási ismereteket feltétez, amelyet egy webdesignernek nem kötelező tudnia. Mivel a webalkalmazás megjelenítési elemeivel a honlaptervező (webdesigner) is

dolgozik, illik olyan tartalmat tenni a JSP lapokba, hogy programozási ismeretekkel nem rendelkező is könnyedén módosíthassa. A szkriptelemek felválthatók standard JSP elemekkel, elemkönyvtárak elemeivel, illetve Expression Language elemekkel (l. 5.3., 5.4. alfejezetek).

5.1. táblázat. *JSP szkriptelemek*

Szkriptelem	**Példa**
Megjegyzés	$<\%--megjegyzes--\%>$
Direktíva	$<\%@\quad direktiva \quad \%>$
Deklaráció	$<\%!\quad deklaracio \quad \%>$
Szkriptlet	$<\%\quad szkriptlet \quad \%>$
Kifejezés	$<\%=\quad kifejezes \quad \%>$

A JSP lapba beágyazható szkriptelemeket az 5.1. táblázat foglalja össze. Először a megjegyzéseket ismertetjük, ebből háromféle tehető JSP lapba.

Megjegyzések

- HTML megjegyzés: $<!--$ HTML megjegyzés $-->$, a HTML megjegyzés része lesz a HTTP válasznak, tehát a kiszolgáló ezt is elküldi a böngészőnek,
- JSP megjegyzés: $<\%--$ JSP megjegyzés $--\%>$, a JSP megjegyzés csak a JSP lapon látható, ez nem kerül bele a lapból előállított szervletosztályba, illetve a HTTP válaszba sem,
- Java megjegyzés: $<\%/*$ Java megjegyzés $*/\%>$, a Java megjegyzés belekerül a JSP lapból előállított szervletosztály kódjába, de nem kerül bele a HTTP válaszba.

Direktívák

A direktívák olyan információk beállítását szolgálják, amelyek befolyásolják a JSP lap szervletté alakítását. A direktívák szintaxisa a következő:

$$<\%@direktivaNev \quad [attributum = "ertek"] * \%>$$

A $*$ a fenti szintaxisban azt jelzi, hogy az `attributum=ertek` páros akárhányszor ismétlődhet. Összesen három típusú direktíva van:

- `page`: ezt a direktívát használhatjuk például Java import utasításokra, ezáltal az importált csomagokban szereplő Java osztályok rövid névvel is elérhetővé válnak:

$$< \%@page \quad import = "java.util.*, java.io.*" \% >$$

- `include`: egy másik komponens tartalmának beszúrását végzi az adott JSP komponensbe:

$$< \%@includefile = "banner.jsp" \% >$$

- `taglib`: elemkönyvtárak használatát teszi lehetővé:

$$< \%@taglib \quad uri = "tlds/taglib.tld" \quad prefix = "mytag" \% >$$

Deklarációk

A deklarációk segítségével a JSP lapból előállított osztályhoz adhatunk hozzá attribútumokat, illetve metódusokat.

$$< \%! \quad deklaracio \quad \% >$$

Ne feledjük, hogy az így meghatározott attribútumok a JSP lapból előállított szervletosztályhoz tartoznak, amelyből majd a webkonténer pontosan egy példányt készít. Ez az egy szervletpéldány fogja az összes, akár párhuzamosan érkező kérést kiszolgálni (külön szálon mindenik kérést), ezért konkurencia-problémákkal kell számolni.

Leggyakrabban a deklarációkat a `jspInit` és a `jspDestroy` metódusok bevezetésére használják. Mivel ezek a metódusok pontosan egyszer hajtódnak végre egy JSP lap életcikusában, létrehozáskor, illetve megsemmisítéskor, ez nem képezi tárgyát a párhuzamos végrehajtásnak.

Példák deklarációra:

```
<%! int counter = 0; %>

<%!
   public void jspInit(){
      //
   }
%>
```

```
<%!
   public void jspDestroy(){
      //
   }
%>
```

Szkriptletek

Szkriptletek segítségével tetszőleges Java kód illeszthető be JSP lapba. A szkriptletként megadott kódrészletek, a JSP lapból előállított szervlet `_jspService` metódusában fognak szerepelni.

Példák szkriptletekre

- 10-szer megjeleníti a `Hello, World!` szöveget

```
<% for( int i=0; i<10; ++i){%>
      <h2>Hello World!</h2>
<% }%>
```

- előállít 2 véletlen számot és növekvő sorrendben kiírja

```
<h1>Veletlen szamok</h1>
<%
   int a = (int)(Math.random()*100);
   int b = (int)(Math.random()*100);
   if( a <b ){
%>
     <p><%=a%>, <%=b%></p>
<% }else{ %>
     <p><%=b%>, <%=a%></p>
 <%}%>
```

Kifejezések

A kifejezések kiértékelése futásidőben történik, és a kifejezés értéke beillesztődik a válaszba. A szkriptletpéldák tartalmaztak kifejezéseket is, például a véletlenszámok megjelenítése:

$$< \% = a\% >, < \% = b\% >$$

A fenti példában az `a` és `b` változók a `_jspService` metódus lokális változóiként jönnek létre, ezen változók értékei illesztődnek be a válaszba.

XML szintaxisú szkriptelemek

Minden szkriptelemeknek van XML szintaxisú megfelelője:

- Szkriptlet (<% ... %>)

```
<jsp:scriptlet> ... </jsp:scriptlet>
```

- Kifejezés (<%= ... %>)

```
<jsp:expression> ... </jsp:expression>
```

- Deklaráció (<%! ... %>)

```
<jsp:declaration> ... </jsp:declaration>
```

- Direktíva (<%@ page import="java.util.*" %>)

```
<jsp:directive.page import="java.util.*" />
```

A JSP sablonszövegnek is van XML stílusú megfelelője:

```
<jsp:text>...</jsp:text>
```

Implicit objektumok

A JSP lapon elérhető implicit objektumokat az 5.2. táblázat foglalja össze. Ezeket az objektumokat a webkonténer hozza létre és teszi elérhetővé a webalkalmazáshoz tartozó JSP lapok számára.

A legfontosabb objektumokat részletesen ismertetjük:

- `request`: ez a HTTP kérésobjektum, segítségével hozzáférhetünk az aktuális kérés összes adatához, például kérési paraméterekhez, attribútumokhoz, fejlécekhez, sütikhez.
- `response`: ez a HTTP válaszobjektum, amelynek metódusaival beállíthatjuk a válasz fejléceket, sütiket küldhetünk.
- `session`: segítségével a menettel kapcsolatos információkhoz férhetünk hozzá, például a menet érvénytelenítése is ezen objektum segítségével valósítható meg.
- `application`: az alkalmazásszintű attribútumok kezelése, illetve más alkalmazásszintű adatok kezelésére szolgáló objektum.

5.2. táblázat. *JSP implicit objektumok*

Objektum	Típus
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.HttpServletConfig
pageContext	javax.servlet.jsp.PageContext
page	java.lang.Object
exception	java.lang.Throwable

- `out`: a válasz írását lehetővé tevő objektum, a `print()` és `println()` metódusok biztosítják szövegek beillesztését a válaszba.
- `exception`: ez az objektum csak olyan JSP lapokon érhető el, amelyek hibakezelő lapok. Egy JSP lapot direktíva segítségével lehet beállítani hibakezelésre.

Hangsúlyozzuk, hogy az implicit objektumokat csak szkriptletekben és kifejezésekben használhatjuk, hatókörük a `_jspService` metódusra korlátozódik. Ezért nem használhatjuk például deklarációkban. A következő kódrészlet fordítási hibát eredményez, mert használja az `out` implicit objektumot:

```
<%!
  public void m(){
     out.println("Hello");
  }
%>
```

A `request` objektum JSP lapokban

Legyen a következő JSP kód egy `show_request_data.jsp` lap része:

```
<h2>A kérés adatai: </h2>
<ul>
  <li>Kérési metódus  : <%= request.getMethod()%></li>
  <li>Kérési URI      : <%= request.getRequestURI()%></li>
  <li>Kérési protokoll: <%= request.getProtocol()%></li>
  <li>Szervlet elérési útja:
                    <%= request.getServletPath()%></li>
  <li>Lekérdező karakterlánc:
                    <%= request.getQueryString()%></li>
  <li>Kiszolgáló neve: <%= request.getServerName()%></li>
  <li>A kérés portja: <%= request.getServerPort()%></li>
  <li>Távoli cím: <%= request.getRemoteAddr()%></li>
  <li>Távoli gazda: <%= request.getRemoteHost()%></li>
  <li>Böngésző típusa:
               <%= request.getHeader("User-Agent")%></li>
</ul>
```

A fenti JSP lapot elkérve egy lehetséges tartalom:

```
A kérés adatai:

    * Kérési metódus : GET
    * Kérési URI : /distedu/show_request_data.jsp
    * Kérési protokoll: HTTP/1.1
    * Szervlet elérési útja: /show_request_data.jsp
    * Lekérdező karakterlánc: null
    * Kiszolgáló neve: localhost
    * Kérési portja: 8099
    * Távoli cím: 127.0.0.1
    * Távoli gazda: 127.0.0.1
    * Böngésző típusa: Mozilla/5.0 (Windows; U; Windows NT 6.0;
      en-US; rv:1.9.0.6) Gecko/2009011913 Firefox/3.0.6
```

Most pedig részletesen kifejtjük a `page`, illetve az `include` direktívákat.

A page direktíva

A `page` direktíva segítségével olyan üzeneteket küldhetünk a webkonténernek, amelyeket a JSP lap szervletté alakításakor vesz figyelembe.

Egy JSP lapon belül többször is szerepelhet a `page` direktíva, de minden egyes attribútum csak egyszer. Kivételt képez az `import` attribútum, ez akárhányszor szerepelhet ugyanazon a JSP lapon. A lapon belül bárhova helyezhetjük, viszont ajánlott a lap elejére tenni a direktívákat.

- `language` – A lapon használt szkript nyelvet rögzíti. Pillanatnyilag a `java` az egyetlen megengedett érték.
- `extends` – Az ősosztály neve adható meg. Használata nem ajánlott, mert hordozhatósági problémákat vethet fel.
- `buffer` – Meghatározza a JspWriter objektum kimeneti adatfolyamánál használt puffer méretét. Az alapértelmezett méret 8 KB. Pl. `buffer="18kb"` vagy `buffer="none"`.
- `autoFlush` – A puffer automatikus ürítését szabályozó attribútum. Amennyiben értéke true (ez az alapértelmezett), a puffer megtelés után automatikusan ürül, különben kivétel váltódik ki.
- `session` – Ez az attribútum határozza meg, hogy a lap részt vesz-e a munkamenetben vagy sem. Az alapértelmezett érték `true`.
- `import` – A JSP lapból előállított Java szervlethez szükséges Java osztályok és interfészek. Több osztály (vagy interfész), illetve csomag megadható egyetlen import direktíva segítségével, amennyiben ezeket vesszővel választjuk el. Példa: `import="java.sql.*,java.util.Vector"`
- `isThreadSafe` – Értéke true (alapértelmezett) vagy false. Ha értéke false, akkor olyan szervletet kell a webkonténernek a JSP lapból előállítani, amely egyidejűleg csak egy kérést fog feldolgozni, tehát szálbiztos.
- `info` – JSP laphoz rendelhető szöveges információ.
- `contentType` – A kimeneti adatfolyam MIME típusa. Alapértelmezett értéke a `text/html`.
- `pageEncoding` – A kimeneti adatfolyam karakter kódolása. Alapértelmezett értéke: `ISO-8859-1`.
- `isELIgnored` – Az Expression Language részek feldolgozását szabályozó attribútum. Ha értéke `true`, az EL kifejezések nem értékelődnek ki. Az alapértelmezett érték `false`.
- `errorPage` – Egy JSP lap ezzel az attribútummal határozhatja meg, hogy mely másik JSP lapot óhajtja hibalapként használni. Hiba esetében a vezérlés automatikusan átadódik a hibalapnak. Pl. `errorPage="/errors/studentError.jsp"`
- `isErrorPage` – Ez az attribútum jelzi, hogy a lap egy hibalap, és más JSP lapok használhatják hibalapként.

Az include direktíva

Segítségével ugyanazon szöveg (JSP lap) beilleszthető több másik JSP lapba. Akkor használjuk, ha több JSP lapnak van közös része, például a fejléc rész azonos. Ekkor a közös részt elhelyezzük például egy `header.jsp` állományba, majd beillesztjük az `include` direktíva segítségével az összes többi lapba.

```
<%@include file="header.jsp"%>
```

A beillesztés a JSP lap szervletté alakítása előtt történik, és semmi garancia nincs arra nézve, hogy ha időközben megváltozik a beszúrandó állomány tartalma, akkor új szervlet keletkezzen. A szövegbeillesztés után keletkezett JSP lapból egyetlen szervlet keletkezik.

5.3. Standard JSP tagok

A standard JSP tagok lehetővé teszik, hogy XML szintaxishoz hasonló JSP lapokat készítsünk. Használatuk növeli a kód olvashatóságát.

Standard tagok általános alakja:

```
<jsp:prefix .../>
```

vagy

```
<jsp:prefix ...>
   ...
</jsp:prefix>
```

Java babok

A Java bab (`JavaBeans`) egy olyan Java osztály, amely teljesíti a következő feltételeket:

- nincs publikus tulajdonság (példánymező),
- minden tulajdonság get/set metódusok segítségével kezelhető (például a `name` attribútum a `setName, getName` metóduspár segítségével kezelhető),
- van paraméter nélkül hívható konstruktora,
- az osztály szerializálható, tehát implementálja a `java.io.Serializable` interfészt.

Java bab

```
package model;

public class Course implements java.io.Serializable{
    private String name;
    private String description;
    private double price;

    public Course(){
        this.name = "";
        this.description = "";
        this.price = 0.0;
    }

    public Course( String name,
            String description, double price){
        this.name = name;
        this.description = description;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
```

```
        this.price = price;
    }

    @Override
    public String toString(){
        return name+"#"+description+"#"+price;
    }
}
```

A useBean tag

Ha JSP lapon Java babot szeretnénk használni, akkor ezt deklarálni kell használat előtt.

Szintaxis:

```
<jsp:useBean
   id="beanName"
   scope="page|request|session|application"
   class="className"
   type="typeName"
/>
```

Az `id` tulajdonság meghatározza a bab nevét, ez az egyedi azonosítója a babnak. A `scope` tulajdonsággal a bab hatókörét határozhatjuk meg, amennyiben ez hiányzik, az alapértelmezett hatókör a `page`. A `class` tulajdonságban meg kell adnunk a teljes osztálynevet, ez a tulajdonság a bab példányosításához szolgáltatja az osztálynevet. Amennyiben a bab már létezik (nem kell példányosítani) valamely hatókörben, a `class` tulajdonság helyett a `type` is használható. A `type` tulajdonsággal a referencia, nem pedig az objektum típusa adható meg. Például ha létezik a kérés hatókörében egy `ArrayList` típusú Java bab, akkor erre így is hivatkozhatunk:

```
<jsp:useBean  id="list" scope="request" type="java.util.List"/>
```

Java babot a `Course` osztályból a következőképpen készíthetünk:

```
<jsp:useBean  id="course" scope="request"
              class="model.Course"/>
```

Ha szkriptet használtunk volna, akkor a fentivel ekvivalens kód a következő lenne:

```
<%@page import="model.*"%>
<%
   Course course = (Course) request.getAttribute("course");
   if( course == null ){
      course = new Course();
      request.setAttribute("course",course);
   }
%>
```

A useBean tagot használhajuk törzs résszel is, ebben az esetben a törzs csak akkor hajtódik végre, ha a bab még nem létezik, és azt létre kell hozni. A törzs részben a babot inicializáló kódot lehet elhelyezni:

```
<jsp:useBean  id="course" scope="request" class="model.Course">
  <%
     course.setName ( request.getParameter("name"));
     course.setPrice( Double.parseDouble(
                       request.getParameter("price")));
  %>
</jsp:useBean>
```

Egy bab tulajdonságai beállítását, illetve lekérdezését a setProperty, illetve a getProperty tagokkal végezhetjük.

A setProperty tag

Adatok tárolását a babban a setProperty teszi lehetővé, amelynek szintaxisa:

```
<jsp:setProperty name="beanName" propertyexpresion />
```

A propertyexpression pedig a következőképpen nézhet ki:
- property="*"
- property="propertyName"
- property="propertyName" param="paramName"
- property="propertyName" value="propertyValue"

Példák:

1. Beállítja a bab name tulajdonságát felhasználva a HTTP kérés azonos nevű paraméterét.

   ```
   <jsp:setProperty name="course" property="name"/>
   ```

   Ekvivalens Java kód:

   ```
   course.setName(request.getParameter("name"));
   ```

2. Beállítja a bab name tulajdonságát felhasználva a HTTP kérés nev nevű paraméterét. Ezt az alakot abban az esetben szükséges használni, ha a bab tulajdonsága nem egyezik a HTTP kérés paraméter nevével.

   ```
   <jsp:setProperty name="course" property="name"
                    param="nev"/>
   ```

   Ekvivalens Java kód:

   ```
   course.setName(request.getParameter("nev"));
   ```

3. Beállítja a bab name tulajdonságát a value-ban megadott értékre. A value meghatározható kifejezés segítségével is. Az alábbi példában egy metódushívás fogja szolgáltatni a kifejezés értékét:

   ```
   <jsp:setProperty
        name="course"
        property="name"
        value='<%= metodus() %>'
   />
   ```

   Ekvivalens Java kód:

   ```
   course.setName(metodus());
   ```

4. Ha * a property értéke, akkor ez azt jelenti, hogy a bab minden egyes tulajdonsága felveszi az azonos nevű HTTP kérés paraméter értéket.

   ```
   <jsp:setProperty name="course" property="*"/>
   ```

A getProperty tag

A `getProperty` segítségével valamely tulajdonság értékét nyerhetjük ki a babból és helyezhetjük a kimeneti adatfolyamba.

```
<jsp:getProperty name="beanName" property="propertyName"/>
```

A következő tag a tanfolyam bab árát nyeri ki.

```
<jsp:getProperty name="course" property="price"/>
```

Ezzel ekvivalens Java kód:

```
out.print(course.getPrice());
```

JSP kódrészlet, amely megjeleníti egy tanfolyam adatait a `course` nevű babból:

```
<H3>Tanfolyam adatok:</H3>
Megnevezes: <jsp:getProperty name="course" property="name"/>
<br>
Ar        : <jsp:getProperty name="course" property="price"/>
<br>
```

Az include tag

Web erőforrások tartalmi beágyazását végezhetjük az `include` direktíva segítségével is. Ebben az esetben az egymásba ágyazás már a szervletté alakítás fázisa előtt megtörténik, és ennek következtében az egymásba ágyazott JSP lapból egyetlen szervlet keletkezik. Ennek a módszernek az a hátránya, hogy ha időközben megváltozik a beágyazott web erőforrás, ez nem fog tükröződni a végeredményben automatikusan, csak újrafordítás esetén.

Ha időben változó tartalmú JSP lapot akarunk beágyazni, akkor erre a standard `include` tag használata ajánlott. A beágyazás futásidőben történik, így ha módosítottuk a beágyazott JSP lapot, ez már tükröződni fog a lap kimenetében. Technikailag itt mind a beágyazó JSP lapból, mind pedig a beágyazott JSP lapból külön szervlet keletkezik.

Tekintsünk egy példát a kétféle beágyazásra, amelyben a beágyazó JSP lapban deklarált változók láthatóságát vizsgáljuk:

1. Az include direktíva

   beagyazo.jsp

   ```
   <%! String name="Hello"; %>
   <%@ include file="beagyazott.jsp" %>
   ```

   beagyazott.jsp

   ```
   <% out.print(name); %>
   ```

2. Az include standard tag

   beagyazo.jsp:

   ```
   <%! String name="Hello"; %>
   <jsp:include page="beagyazott.jsp" />
   ```

   beagyazott.jsp

   ```
   <% out.print(name); %>
   ```

Megfigyelhetjük, hogy mindkét esetben a beagyazott.jsp tartalma megegyezik, a különbség a beagyazo.jsp JSP lapon van: az első include direktívát használ, a második pedig JSP standard tagot. Az include direktíva használata esetében az eredmény a Hello kimenet előállítása lesz, a standard tag esetében viszont fordítási hibát kapunk, hiszen a beágyazott JSP lapon nem lesz elérhető a name nevű változó.

A param tag

A beágyazott JSP lapnak paramétereket is adhatunk át, ezek a HTTP kérés paraméterek szintaxisát használva lesznek elérhetők. Tekintsük erre a következő példát:
Beágyazó JSP lap:

```
<jsp:include page="beagyazott.jsp">
   <jsp:param name="username" value="jojo"/>
</jsp:include>
```

Beágyazott JSP lap:

```
A felhasznalo neve: <%= request.getParameter("username") %>
```

A forward tag

A forward tag a kérés feldolgozását egy másik webkomponenshez továbbítja. A cél webkomponens lehet HTML lap, JSP lap vagy szervlet. Alapértelmezetten ha a továbbító komponens már valamilyen választ előállított, és ez a pufferben van, akkor ez törlődik. A válasz előállítását az a webkomponens végzi, amelyhez továbbítódott a kérés.

Szintaxis:

```
<jsp:forward page={"relativeURL" | "<%= expression %>"}/>
```

vagy

```
<jsp:forward page={"pageURL" | "<%= expression %>"}>
   <jsp:param
         name="parameterName"
         value={"parameterValue" |"<%= expression %>"}
   />
</jsp:forward>
```

Példák:

```
<jsp:forward page="masik.jsp"/>

<jsp:forward page="masik.jsp">
   <jsp:param name="username" value="jojo"/>
</jsp:forward>
```

A kérés továbbításának célja többféle webkomponens is lehet, viszont ugyanazon webalkalmazáshoz kell tartoznia. A <jsp:forward> hívás utáni rész a hívó JSP lapból nem lesz feldolgozva. Hasonlóan az include taghoz, ebben az esetben is lehet paramétereket átadni a cél webkomponensnek. Csak JSP lapnak és szervletnek érdemes paramétert átadni. Az include és a forward közötti különbséget már a szervleteknél szemléltettük, ez JSP lapoknál is hasonló.

5.4. Expression Language

Gyakran EL rövidítést használunk az Expression Language helyett. Azzal a céllal vezették be a nyelvbe, hogy egyszerűen lehessen adatokhoz hozzáférni JSP lapokon, a szkriptelemek alternatívája kíván lenni. A JSF (Java Server Faces) technológia új követelményeket támasztott az EL kifejezések kiértékelésével szemben, ezért kidolgozták a JSF EL-t is, majd egységesítették, bevezetve az EL kifejezések kétféle kiértékelési módszerét.

- `${kifejezes}` – ezek a kifejezések azonnal kiértékelődnek az oldal generálásakor és csak egyszer hajtódnak végre. Példa:

  ```
  ${param.name}
  ```

  Ebben az esetben a `param` implicit objektum típusa `Map`, tehát asszociatív tömb, és a kifejezés a `name` névhez asszociált érték lekérését jelenti.
- `#{kifejezes}` – ezt késleltetett kiértékelésnek nevezzük, és azt jelenti, hogy a kiértékelést nem a JSP konténer végzi, hanem az azt használó technológia. Ebben az esetben a kiértékelés a JSF életciklusához kötődik. Egy másik különbség az azonnali kiértékelésű kifejezésekhez képest az, hogy ez az utóbbi bal, illetve jobb értékként is használható (képezheti értékadás bal és jobb oldalát egyaránt). Ez a jelölés csak a JSF keretrendszer esetében használható. Példa:

  ```
  #{param.name}
  ```

JSP lapokon az EL kifejezéseket a következő esetekben érdemes használni:

- standard JSP elemben valamely attribútum megadásakor:

  ```
  <jsp:include page='${URL_to_page}'/>
  ```

- HTML szövegben. Ebben az esetben az EL kiértékelődik, és az érték be lesz illesztve a kimenetbe a kifejezés helyére:

  ```
  <H2>Hello, ${param.name}</H2>
  ```

EL implicit objektumok

A JSP implicit objektumokhoz hasonlóan az EL-ben is használhatunk egy pár EL-specifikus implicit objektumot, ezeket fogjuk felsorolni a következő lépésben.

- `pageContext`: Ez egy referencia az aktuális PageContext objektumra. Hozzáférést biztosít a JSP lap hatóköréhez tartozó névterekhez.
- `pageScope`: Egy Map típusú objektum, amely a lap hatókörébe tartozó attribútumokat tárolja a hozzárendelt értékekkel együtt. (String – Object)
- `requestScope`: Egy Map típusú objektum, amely a kérés hatókörébe tartozó attribútumokat tárolja a hozzárendelt értékekkel együtt. (String – Object)
- `sessionScope`: Egy Map típusú objektum, amely a menet hatókörébe tartozó attribútumokat tárolja a hozzárendelt értékekkel együtt. (String – Object)
- `applicationScope`: Egy Map típusú objektum, amely az alkalmazás hatókörébe tartozó attribútumokat tárolja a hozzárendelt értékekkel együtt. (String – Object)
- `param`: Egy Map típusú objektum, amely a kérés paramétereket tárolja a hozzárendelt értékekkel együtt. Minden paraméterhez pontosan egy érték tartozik. (String – String)
- `paramValues`: Egy Map típusú objektum, amely a kérés paramétereket tárolja a hozzárendelt értéktömbbel együtt. Minden paraméterhez pontosan egy értéktömb tartozik. (String – String[])
- `header`: Egy Map típusú objektum, amely a fejlécneveket tárolja a hozzárendelt értékekkel együtt. Minden fejlécnévhez pontosan egy érték tartozik. (String – String)
- `headerValues`: Egy Map típusú objektum, amely a fejlécneveket tárolja a hozzárendelt értékekkel együtt. Minden fejlécnévhez pontosan egy értéktömb tartozik. (String – String[])
- `cookie`: Egy Map típusú objektum, amely a sütiket tárolja. (String – String)

Az EL bevezetésének célja, hogy kiszorítsa a Java nyelvű szkriptelemeket a JSP lapokról, ezért nem lehet használni EL kifejezést Java szkript kódban. A következő JSP kód fordítási hibát eredményez:

```
<%= request.getParameter( ${name}) %>
```

EL és a tárolók

Bármilyen Java tároló, amely tárolva van valamilyen hatókörben (lap, kérés, menet, alkalmazás), kényelmesen elérhető EL segítségével.

A tömbök tekinthetők a legegyszerűbb tárolóknak, de használható a Java Collections keretrendszer bármely tárolója (`Vector, List, Set, Map` stb.). Tekintsünk erre egy példát:

```
<%
  String[] allatok={"kutya", "macska", "elefant"};
  request.setAttribute("allatok", allatok);
%>

<p>Elso: ${allatok[0]}</p>
<p>Elso: ${requestScope.allatok[0]}</p>

<p>Masodik: ${allatok['1']}</p>
<p>Masodik: ${requestScope.allatok['1']}</p>

<p>Harmadik: ${allatok["2"]}</p>
<p>Harmadik: ${requestScope.allatok["2"]}</p>
```

A fenti pédában láthatjuk a tömbelemek elérésének sokféleségét. Először is nem kötelező megadni a tömb hatókörét. Amennyiben nem adjuk meg, akkor a tömb keresése először a lap hatókörében történik, majd a kérés, menet, illetve az alkalmazás hatókörével folytatódik. Ha nem létezik a megadott nevű tömb, vagy éppen a hivatkozott tömbelem hiányzik, az eredmény egy üres karakterlánc, és nem keletkezik futásidejű hiba. Másodsorban láthatjuk, hogy a tömbelem indexe megadható numerikus, karakter, illetve karakterlánc típusú kifejezéssel. Mindenik helyes EL kifejezés lesz.

Most pedig tekintsünk még egy példát, amelyben a kérés paramétereket feldolgozó lapban a `paramValues` implicit objektumot fogjuk használni. Legyen a feldolgozandó HTTP GET kérés a következő:

```
index.jsp?productid=12&productid=23
```

Legyen a kérést feldolgozó JSP lap tartalma a következő:

```
${paramValues.productid[0]}
${paramValues.productid[1]}
```

A fenti kódrészlet ekvivalens a következővel:

```
${paramValues['productid'][0]}
${paramValues['productid'][1]}
```

Java babok elérése EL segítségével

```
<jsp:useBean id="person" class="model.Person"/>
<jsp:setProperty name="person" property="firstName"
                 value="Margit"/>
<jsp:setProperty name="person" property="lastName"
                 value="Antal"/>


<p>First name: ${person.firstName} </p>
<p>Last  name: ${person.lastName}  </p>
```

A fenti példa első három sora létrehozza a babot (amennyiben ez nem létezik) és beállítja a tulajdonságait. Az utolsó két sor pedig egy-egy EL kifejezés a tulajdonságok lekérdezésére. A tulajdonságok lekérésére `babnév.tulajdonságnév` formátumot használtunk, de használható az index operátor is. Ez annak köszönhető, hogy a bab asszociatív tömbként is kezelhető, amely (kulcs,érték) párokat tartalmaz.

```
${person[firstName]}
${person['firstName']}
${person["firstName"]}
```

EL operátorok

Az 5.3., 5.4. és 5.5. táblázatokban az EL kifejezésekben használható operátorokat foglaltuk össze.

5.3. táblázat. *Aritmetikai operátorok*

Operátor neve	Operátor
összeadás	+
kivonás	-
szorzás	*
osztás	/
osztási hányados	%

5.4. táblázat. *Relációs operátorok*

Operátor neve	Operátor
egyenlő	== és eq
különböző	!= és ne
kisebb	< és lt
nagyobb	> és gt
kisebb vagy egyenlő	<= és le
nagyobb vagy egyenlő	>= és ge

5.5. táblázat. *Logikai operátorok*

Operátor neve	Operátor
és	&& és and
vagy	|| és or
nem	! és not

Az EL kifejezések kiértékelése felhasználóbarát módon történik. Az EL kiértékelő motor a következőket biztosítja:

- karakterlánc típusú paramétrek átalakítása numerikus értékké,
- nem létező paraméter estében a null értéket 0 numerikus értékké alakítja,
- ha a karakterlánc nem alakítható át numerikus értékké, kivétel váltódik ki.

JSP környezet beállítása

A web.xml telepítésleíróban szabályozhatjuk a JSP lapok viselkedését [4]. Erre a <jsp-config> elem használható. Itt pedig bizonyos, JSP csoportokra vonatkozó tulajdonságokat állíthatunk be. Tekintsük a következő példát:

```
<jsp-config>
  <jsp-property-group>
```

```
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>

  <jsp-property-group>
    <url-pattern>/scriptless/*</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

Amint a fenti példából is kitűnik, egy <jsp-config> tag több JSP csoportra adhat meg tulajdonságokat. Minden egyes csoportra meg kell adni először a csoport tagjait, vagyis mely JSP lapokra vonatkozik a konfiguráció. Ezt az <url-pattern> tag biztosítja, amit valamilyen konfiguráció követ. Az <el-ignored> true értéke azt jelenti, hogy a megadott JSP lapokon belül az EL kifejezések nem fognak kiértékelődni. Ha például a <scripting-invalid> értéke true, akkor fordítási hibát okoznak azon JSP lapok, amelyek szkriptkódot tartalmaznak.

5.5. Tesztkérdések

5.1. kérdés: Adott a következő JSP kód:

```
<html>
   <body>
     <% for (int i = 0; i < 5 ; i++)
     {  %>
         out.print(i);
     <% }  %>
   </body>
</html>
```

Mi lesz az előállított válasz? (1 helyes)

A. Fordítási hiba keletkezik.
B. Helyes fordítás, de nem állít elő kimenetet.
C. Kiírja 0-tól 4-ig a számokat.
D. Kivételt dob futás alatt.
E. Egyik sem az előbbiek közül.

5.2. kérdés: Egy `foo.jsp` nevű JSP lap adatbázishoz csatlakozik és a lekérdezett adatokat megjeleníti a kliensnek. A műveletek során kivétel léphet fel, amelyet egy `error.jsp` nevű kivétellapon szeretnénk kezelni. A következő JSP kódok közül melyek szükségesek a megvalósításhoz? (2 helyes)

```
A. <%@errorPage="error.jsp"%>  a foo.jsp lapon.
B. <%@page errorPage="error.jsp"%> a foo.jsp lapon.
C. <%@page isErrorPage="true"%>  az error.jsp lapon.
D. <%@isErrorPage="true"%> az error.jsp lapon.
```

5.3. kérdés: Melyik osztályból kell a JSP lapból előállított osztályt származtatni? (1 helyes)

```
A. javax.servlet.jsp.JspPage
B. javax.servlet.jsp.HttpJspPage
C. javax.servlet.jsp.JSP
D. javax.servlet.jsp.Page
E. Egyik sem ezek közül
```

5.4. kérdés: Egy JSP lap transzlációs fázisa során mely tevékenységek hajtódnak végre? (3 helyes)

```
A. Létrejön a JSP lapnak megfelelő implementációs osztály.
B. Lefordítódik  a JSP lapnak megfelelő implementációs
   osztály.
C. Betöltődik a JSP lapnak megfelelő implementációs osztály.
D. Szintaktikus ellenőrzés hajtódik végre a JSP lapon.
```

5.5. kérdés: A következő kódrészletek közül válasszuk ki azokat, amelyek kiíratják a kérésben szereplő összes paraméter értékét. (2 helyes)

```
A. <%
     Enumeration enum = request.getParameterNames();
     while(enum.hasMoreElements()) {
        Object obj = enum.nextElement();
        out.println(request.getParameter(obj));
     }
   %>
```

```
B. <%
     Enumeration enum = request.getParameters();
     while(enum.hasMoreElements()) {
        String obj = (String) enum.nextElement();
        out.println(request.getParameter(obj));
     }
   %>

C. <%
     Enumeration enum = request.getParameterNames();
     while(enum.hasMoreElements()) {
        String obj = (String) enum.nextElement();
        out.println(request.getParameter(obj));
     }
   %>

D. <%
     Enumeration enum = request.getParameterNames();
     while(enum.hasMoreElements()) {
        Object obj = enum.nextElement();
   %>
      <%=request.getParameter(obj); %>
   <% } %>

E. <%
     Enumeration enum = request.getParameterNames();
     while(enum.hasMoreElements()) {
        String obj = (String) enum.nextElement();
   %>
      <%=request.getParameter(obj)%>
   <% } %>
```

5.6. kérdés: A következő JSP kódrészletek közül mely használható metódus definícióra? (2 helyes)

```
A. <% public void m1() { ... } %>
B. <%! public void m1() { ... } %>
C. <%@ public void m1() { ... } %>
D. <%! public void m1() { ... }; %>
```

```
E. <% public void m1() { ... }; %>
```

5.7. kérdés: Adott két JSP fájl:

```
//File: companyhome.jsp:
<html><body>
Welcome to ABC Corp!
<%@ page errorPage="simpleerrorhandler.jsp" %>
<%@ include file="companynews.jsp" %>
</body></html>

//File: companynews.jsp:
<%@ page errorPage="advancederrorhandler.jsp" %>
<h3>Todays News</h3>
```

Válasszuk ki a helyes kijelentést! (1 helyes)

```
A. Ha a companyhome.jsp-hez érkezik kérés, a kimenet tartal-
   mazza a "Welcome...", illetve a "Todays News" szövegeket.
B. Fordítási hiba a  companyhome.jsp lapon.
C. Fordítási hiba a  companynews.jsp lapon.
D. Helyes fordítás mindkét JSP lapra és futásidejű kivétel.
```

5.8. kérdés: Melyek helyes iterációk JSP lapon? (3 helyes)

```
A. <% int i = 0;
      while(i<5){
        "Hello World"
         i++;
      } %>

B. <jsp:for loop='5'>
      "Hello World"
  </jsp:for>

C. <% int i = 0;
      for(;i<5; i++){
   %>
        "Hello World";
   <%
        i++;
```

```
      }
   %>

D. <table>
   <%
     java.util.Vector v = new java.util.Vector();
     v.add("alma");
     v.add("körte");
     v.add("szilva");
     java.util.Iterator it = v.iterator();
     int i = 0;
     while(it.hasNext())
     {
        out.println("<tr><td>"+(++i)+"</td>");
        out.println("<td>"+it.next()+"</td></tr>");
     }
   %>
   </table>

E. <jsp:scriptlet>
     <![CDATA[
     for(int i=0; i<5; i++){
     ]]>
   </jsp:scriptlet>
   <jsp:text>"Hello World!"</jsp:text>
   <jsp:scriptlet>
     }
   </jsp:scriptlet>
```

5.9. kérdés: Válasszuk ki a helyes JSP kódrészleteket! (2 helyes)

```
A. <%@ page import="java.util.*" autoFlush="true"%>
   <%@ page import="java.io.*" autoFlush="false"%>

B. <%Date d = new Date();
      out.println(d);%>

C. <%= String val = request.getParameter("hello");
      out.println(val);%>
```

```
D. <%! Hashtable ht = new Hashtable();
       {
           ht.put("max", "10");
       }%>

E. <%!
     Hashtable ht = new Hashtable();
     ht.put("max", "10");%>
```

5.10. kérdés: Válasszuk ki a helyes kijelentéseket! (2 helyes)

A. A session implicit objektum használható adatok megosztására ugyanazon alkalmazáshoz tartozó szervletek és JSP lapok között.
B. A request implicit objektum használható adatok megosztására szervletek és JSP lapok között különböző kérések során.
C. A válasz típusának beállítására használható a response implicit objektum.
D. Az out implicit objektum megegyezik a response.getWriter() által visszatérített objektummal.

5.11. kérdés: Válasszuk ki a helyes JSP direktívákat! (2 helyes)

```
A. <%@ page %>
B. <%! taglib uri="http://www.abc.com/tags/util"
            prefix="util" %>
C. <% include file="/copyright.html"%>
D. <%@ taglib uri="http://www.abc.com/tags/util"
            prefix="util" %>
E. <%$ page language="java" import="com.abc.*"%>
```

5.12. kérdés: Mely osztály használható a JSP lap hatókörében levő változók elérésére? (1 helyes)

A. HttpSession
B. Servlet
C. JspPage
D. ServletContext
E. PageContext

5.13. kérdés: Válasszuk ki a helyes JSP kifejezéseket! (2 helyes)

```
A. <%= 1+2 %>
B. <%= "1"+"2" %>
C. <%= Math.random(); %>
D. <%   = -2 %>
E. <%= boolean b = true %>
```

6. FEJEZET

JSP ELEMKÖNYVTÁRAK

6.1. JSP és JSTL

A JSTL a JSP Standard Tag Library kifejezés rövidítése. Az elemkönyvtárak bevezetésének az volt a célja, hogy olyan JSP lapok készítését tegyék lehetővé, amelyben nincs szkriptkód, és ezért könnyen karbantartható a programozáshoz kevésbé értő szakemberek által is. Bár programozók is készíthetnek saját elemkönyvtárat, mi csak a standard elemkönyvtár használatát mutatjuk be.

Az előnyök bemutatását kezdjük egy konkrét példával, amelyet elkészítünk mind szkriptkód, mind pedig JSTL használatával. Tételezzük fel, hogy egy `courses` nevű listaobjektum a kérés hatókörében van tárolva, nekünk pedig az a feladatunk, hogy egy nem sorszámozott HTML listaként megjelenítsük egy JSP lapon.

Szkriptkód

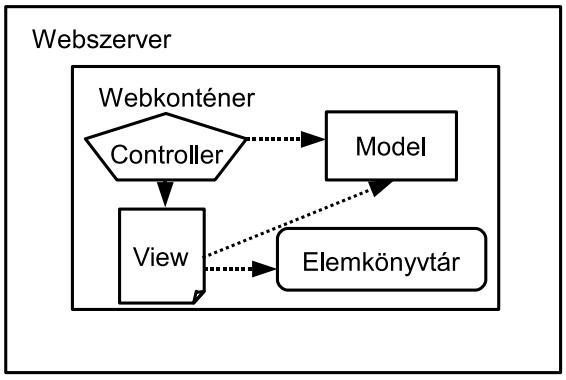
```
<%@page import = "java.util.*"%>
<%@page import = "model.*"%>

<ul>
<%
  List courses=(ArrayList)request.getAttribute("courses");
  if( courses != null && !courses.isEmpty()){
    Iterator it = courses.iterator();
    while( it.hasNext() ){
      Course course = (Course)it.next();
%>
   <li><%= course.getName()%></li>
<%
    }
  }
%>
</ul>
```

JSTL

```
<%@taglib prefix="c"
          uri="http://java.sun.com/jsp/jstl/core"%>
<ul>
<c:forEach var="course" items="${courses}">
   <li><c:out value="${course.name}"/> </li>
</c:forEach>
</ul>
```

Egyelőre nem részletezzük az elemkönyvtárat használó kódot, viszont könnyű belátni, hogy áttekinthető kódot eredményez, amelyet Java programozáshoz nem értő szakember is könnyedén karban tud tartani. A másik nagy előny, hogy újrafelhasználható elemekből építkezik.

6.2. A JSP elemkönyvtárak anatómiája



6.1. ábra. *Az elemkönyvtárak anatómiája [7]*

Amint a 6.1. ábra is mutatja, az elemkönyvtárakat a megjelenítési (interfész) komponensek használják. Az elemkönyvtárakban használható elemkezelőkben minden olyan objektum használható, ami JSP lapon is. Ezt a `pageContext` objektum teszi lehetővé, amely egyben a különböző hatókörökhöz való hozzáférést is biztosítja.

A JSTL elemeket az XML szintaxisnak megfelelően használjuk, a következők betartásával:

– Törzs részt tartalmazó elem szintaxisa:

```
<prefix:name {attribute={"value"|'value'}}*>
   body
</prefix:name>
```

Példa:

```
<c:forEach var="course" items="${courses}">
   //torzs
</c:forEach>
```

– Törzs nélküli elem szintaxisa:

```
<prefix:name {attribute={"value"|'value'}}*/>
```

Példa:

```
   <c:out value="${course.name}"/>
```

– Az elemnevek, attribútumok és prefixek kisbetű/nagybetű érzékenyek.
– Az elemeknek be kell tartaniuk az egymásba ágyazás szabályát, a beágyazott elemet mindig a beágyazó elem előtt kell lezárni:

```
<elem1>
   <elem2>
   </elem2>
</elem1>
```

A JSTL-t, bár egyként emlegetjük, több standard könyvtár alkotja. Ezek a következők:

– mag (core) - `<c:out.../>`
– XML kezelő elemek - `<x:out.../>`
– SQL parancsokat támogató elemek - `<sql:query.../>`
– nemzetköziesítés és formázás - `<fmt:out.../>`
– szövegkezelő függvények - `<fn:split.../>`

Amint már említettük, egy elemkönyvtár két részből áll:

– egy JAR fájl, amely a tagkezelőket tartalmazza (minden tagkezelő egy külön osztály),
– egy TLD – Tag Library Descriptor fájl, amely egy XML fájl és a tagok deklarációit tartalmazza.

Elemkönyvtárat csak úgy tudunk használni JSP lapon, ha ezt előzetesen bevezetjük a `taglib` direktívával. A direktívának két attribútuma van: TLD URI és a prefix. A következő kódrészlet szemlélteti a használatot:

```
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="sajat"
           uri="http://www.ms.sapientia.ro/sajatelemek.tld" %>
```

Egy JSP lapon akárhány elemkönyvtárt használhatunk. Minden elemkönyvtárhoz egyedi prefixet rendelünk a `taglib` direktívával, az elemkönyvtár elemeit pedig csak ezzel a prefixszel használhatjuk a JSP lapon. Az URI attribútum egy szimbolikus helyet határoz meg, amelynek nem kötelező a valóságban is léteznie. A szimbolikus névhez a fizikai elemkönyvtár hozzárendelését a telepítésleíróban végezzük a következőképpen:

```
<taglib>
   <taglib-uri>
      http://www.ms.sapientia.ro/sajatelemek.tld
   </taglib-uri>
   <taglib-location>
      /WEB-INF/sajatelemek.tld
   </taglib-location>
</taglib>
```

A fent felsorolt minden egyes elemkönyvtárhoz egy saját URI tartozik:

- mag - `http://java.sun.com/jsp/jstl/core`
- XML kezelő elemek - `http://java.sun.com/jsp/jstl/x`
- SQL parancselemek - `http://java.sun.com/jsp/jstl/sql`
- nemzetköziesítés és formázás - `http://java.sun.com/jsp/jstl/fmt`
- szövegkezelő függvények - `http://java.sun.com/jsp/jstl/fn`

6.3. JSTL standard elemek

Most pedig sorra vesszük a legfontosabb standard elemeket és példákat adunk használatukra.

Az out elem

Az a célja ennek az elemnek, hogy egy attribútumként megadott kifejezést kiértékeljen és eredményét a kimeneti adatfolyamba helyezze.

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

vagy

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

- `value` – a kiírandó kifejezés
- `escapeXml` – true (alapértelmezett érték) esetén a $<, >, \&, ', "$ értékek átalakítódnak $\&lt, \&gt$, `stb.` értékekké. Akkor hasznos, ha HTML forráskódot akarunk megjeleníteni, mert true érték esetén a böngésző nem értelmezi a HTML szöveget.
- `default` – mit kell kiírni, ha a `value` attribútumban megadott kifejezés értéke `null`.

Például egy űrlapból beolvasott paraméter megjelenítése így végezhető:

```
<c:out value="${param.name}">
  hianyzik a nev parameter
</c:out>
```

Példa az `escapeXml` attribútum használatára:

```
<c:out value="<h1>escapeXml=true </h1>" escapeXml="true" />
<c:out value="<h1>escapeXml=false</h1>" escapeXml="false"/>
```

Eredmény:

- <h1>escapeXml=true </h1>
- escapeXml=false

A set elem

A set elem célja változó értékének beállítása és tárolása adott hatókörben. Használható Java bab, illetve asszociatív tömb (Map) tulajdonság beállítására is. Akárcsak az out tag esetében, ezt is lehet törzs nélkül, illetve törzzsel használni.

```
<c:set
   var="varName"
   value="value"
   [scope="{page|request|session|application}"]
/>
```

vagy

```
<c:set
   var="varName"
   [scope="{page|request|session|application}"]
>
  value
</c:set>
```

- var : változó neve
- value : változó értéke
- scope : változó hatóköre (page - alapértelmezett)

Példa használatra:

```
<c:set var="appName">Distance Learning</c:set>
...
<h1>${appName}</h1>
```

Java bab, illetve asszociatív tömb esetén a szintaxis a következő lesz:

```
<c:set
   target   = "targetName"
   property = "propertyName"
   value    = "value"
/>
```

vagy

```
<c:set
   target   = "targetName"
   property = "propertyName"
>
  value
</c:set>
```

- `target` : létező Java bab vagy Map objektum
- `property` : tulajdonság neve
- `value` : tulajdonság értéke

Az alábbi kódrészlet a `course` nevű Java bab `name` nevű tulajdonságát állítja be valamely űrlap `name` nevű paraméterének értékére.

```
<c:set target="course" property="name" value="${param.name}"/>
```

Az `if` elem

Készítsünk egy JSP lapot, amely előállít egy véletlen egész számot 1 és 10 között, majd eldönti, hogy osztályzatként átmenő-e a jegy.

```
<%@taglib prefix="c"
          uri="http://java.sun.com/jsp/jstl/core"%>
<%
  int number =(int) (Math.random()*10)+1;
  pageContext.setAttribute("number", number);
%>

<c:if test="${number<5}">
  <c:out value="${number}, nem atmeno jegy"/>
</c:if>

<c:if test="${number>=5}">
  <c:out value="${number}, atmeno jegy"/>
</c:if>
```

A fenti példában szándékosan használtunk két `<c:if>` tagot, mert a szintaxis sajnos nem teszi lehetővé a programozási nyelvekben megszokott `else` ág használatát.

```
<c:if
   test   = "expression"
   [var   = "varName"]
```

```
   [scope = "{page|request|session|application}"]
>
  body
</c:if>
```

A var attribútumban megadhatunk egy változónevet, amelyben a test kifejezés értékét szeretnénk eltárolni. Nyilván ez egy logikai érték lesz. A változónak megadható a hatóköre is a scope attribútum segítségével.

A choose, when és az otherwise elem

Ezt az elemet is egy példával szemléltetjük. Ugyanaz a feladat, mint a <c:if> tagnál, csak egy picivel általánosabb. Feltételezzük, hogy most a number bármilyen egész szám lehet, és a döntésnek is három lehetséges eredménye lehet: átmenő, nem átmenő és a jegy kritériumainak nem megfelelő.

```
<c:choose>
  <c:when test="${number >0 && number<5 }">
      <c:out value="${number}, nem atmeno jegy"/>
  </c:when>

  <c:when test="${number>4 && number<=10 }">
          <c:out value="${number}, atmeno jegy"/>
  </c:when>

  <c:otherwise>
          <c:out value="Ilyen jegy nem letezik"/>
  </c:otherwise>
</c:choose>
```

A forEach elem

```
<c:forEach
   items="collection"
   [var="varName"]
   [varStatus="varStatusName"]
   [begin="begin"]
```

```
   [end="end"]
   //body
</forEach>
```

A `forEach` az egyik leggyakrabban használt elem, ciklusok megvalósítására alkalmas. Mivel sokrétű használatot tesz lehetővé, ezért részletesen átvesszük a szintaxisát, majd a különféle használatokra példákat adunk.

A következő attribútumok használhatók egy `forEach` elemben:

- `items`: Ez az attribútum azt a gyűjteményt határozza meg, amelyet a ciklusban feldolgozunk. A következők képezhetik az attribútum értékét:
  - gyűjtemény: `java.util.Collection`,
  - iterátor: `java.util.Iterator`,
  - asszociatív tároló: `java.util.Map`,
  - felsorolás objektum: `java.util.Enumeration`,
  - tömb,
  - vesszővel elválasztott karakterlánc sorozat.
- `var`: ebbe a változóba kerül az `items`-ben megadott gyűjtemény aktuális eleme. Típusa megegyezik a gyűjtemény típusával.
- `varStatus`: a ciklusváltozó szerepét betöltő objektum, amelyből információkat nyerhetünk ki a ciklus végrehajtása során. A leggyakrabban a `status.index`, a ciklusváltozó értéke (0 kezdőérték esetében), illetve `status.count` a ciklusváltozó értéke (1 kezdőérték esetében) formában használjuk.
- `begin`: a kezdőindex értékét adhatjuk meg – akkor hasznos, ha valamely gyűjteményt nem az első elemtől kezdődően szeretnénk feldolgozni.
- `end`: az iteráció ennél az indexnél fejeződik be (ezt is beleértve).
- `step`: az iterációnál az indexre használt növekmény.

Az opcionális elemeknél a `[...]` jelölést alkalmaztuk. Amint láthatjuk, az egyetlen kötelező attribútum az `items`. Ha csak ezt adjuk meg, akkor a ciklus törzsét az `items`-ben megadott entitás méreteszer ismétli. A következő kódrészlet, a `Hello` szöveget a `courses` tároló elemszámaszor jeleníti meg a válaszban.

```
<c:forEach items="${courses}">
   <p>Hello<p>
</c:forEach>
```

A következő kódrészletet csak azért adjuk meg, hogy ha valaki ki akarja próbálni a `forEach` elemmel kapcsolatos példákat, akkor a hivatkozott változók legyenek létrehozva a kérés hatókörében. Ez a kódrészlet létrehoz egy listaobjektumot, amelybe három `Course` típusú példányt helyez, és regisztrálja a listát `courses` név alatt a kérés (`request`) hatókörébe. Ezután lekérjük a lista iterátorát, amelyet `it` néven regisztrálunk ugyanazon hatókörbe.

```
<%@page import = "java.util.*"%>
<%@page import = "model.*"%>
<%
  List c = new ArrayList();
  c.add( new Course("Java SE","Java Standard Edition 6.0",
                    1000));
  c.add( new Course("Java EE","Java Enterprise Edition 5",
                    1000));
  c.add( new Course("Java Web","Java Web Development",
                    1000));
  request.setAttribute("courses", c);
  Iterator it = c.iterator();
  request.setAttribute("it", it);
%>
```

Az alábbi kódrészlet például iterátort használ, a tanfolyamnevek, illetve -árak megjelenítésére.

```
<c:forEach var ="course" items="${it}">
  <p>
     <c:out value="${course.name}"/>,
     <c:out value="${course.price}"/>
  </p>
</c:forEach>
```

Most pedig tekintsünk egy példát, amelyben felhasználjuk a `varStatus`-ból kinyerhető információt – jelen esetben a ciklus végrehajtásának sorszámát fogjuk kinyerni: `$status.count`

```
<ul>
   <c:forEach var ="course" varStatus="status"
                            items="${it}">
   <li>
        <c:out value="${status.count}"/>. 
        <c:out value="${course.name}"/>,
```

```
        <c:out value="${course.price}"/>
   </li>
   </c:forEach>
</ul>
```

A forTokens elem

Ez az elem a Java nyelv `StringTokenizer` típusához hasonló, amelynek feladata egy karakterfüzér halmaz feldolgozása. A karakterfüzérek a `delims` attribútummal megadott speciális karakterekkel vannak elválasztva. A következő példa az `items`-ben megadott, vesszővel elválasztott karakterlánc elemeit íratja ki.

```
<c:forTokens
   items="alma,szilva,eper,barack"
   delims=","
   var="token"  >
   <p><c:out value="${token}"  /></p>
</c:forTokens>
```

Az előző példában expliciten megadtuk a feldolgozandó karakterláncokat, a gyakorlatban ezt valamilyen hatókörbe regisztrált változóból vesszük. Tehát az `items` attribútum így nézne ki: `items="${valtozo}"`

A remove elem

Ezen elem segítségével változót törölhetünk adott érvényességi körből.

```
<c:remove
   var   = "varName"
   scope = "scope"
/>
```

Nyilván elgondolkozhatunk azon, hogy miért érdemes változót törölni adott hatókörből. A legfontosabb érv az, hogy minden egyes változó memóriát foglal, amellyel egy szerveroldali alkalmazásnak illik hatékonyan gazdálkodnia. Aki programozott valaha C++ nyelvben és készített memóriaigényes alkalmazásokat, az tudja, hogy mennyire fontos a helyes gazdálkodás a dinamikusan lefoglalt tárterülettel.

A catch elem

Ez a tag a kivételkezelés JSTL változata. Minden kivétel, amely ezen elem törzsében keletkezik, el lesz kapva és el lesz hanyagolva. Amennyiben használjuk a var attribútumot, lehetőségünk van a kivételobjektum tárolására és utólagos kezelésére. A szintaxis a következő:

```
<c:catch var="name">
  body content
</c:catch>
```

Most pedig tekintsünk egy példát a catch elem használatára:

```
<c:catch var="e">
   <%
     int nr = 0;
     out.print(100/nr);
   %>
</c:catch>

<c:out value="${e.message}"/>
```

Mivel a catch elem törzsében kivétel keletkezett, ez a megadott e változóban fog eltárolódni. A catch utáni out elem pedig kiírja a kivétel okát, mely jelen esetben nullával való osztás. Szándékosan használtunk szkriptet az osztás elvégzésére, ugyanis EL használatakor soha nem keletkezik kivétel. Ebben az esetben az osztás eredménye végtelen lett volna, és az infinity jelent volna meg a válaszban.

Az import elem

Funkcionalitásában a <jsp:include>-ra hasonlít. Azonban míg a <jsp:include> elem csak lokális erőforrások beszúrását teszi lehetővé, addig a <c:import> távoli erőforrások beszúrását is megengedi.

Helyi erőforrás beszúrása:

```
<c:import url="header.jsp">
   <c:param name="pageTitle" value="Distance Education"/>
</c:import>
```

Távoli erőforrás beszúrása:

```
<c:import url="http://www.sapientia.ro"/>
```

Az `url` elem

```
<c:url
    value   = "expression"
    context = "expression"
    var     = "name"
    scope   = "scope"
>
  <c:param name="expression" value="expression"/>
  ...
</c:url>
```

A `<c:url>` elem automatikus URL újraírást végez, amelyet akkor használunk, ha a sütik nem működnek (az ügyfél letiltotta a böngészőben). Az URL újraírás célja, hogy hozzáadja a menetazonósítót az URL-hez. Tehát ha a JSP konténer észleli a süti jelenlétét, nincs szükség URL újraírásra, és ez nem is fog megtörténni.

Ha jelen van a `var` attribútum is, akkor az itt megadott változó veszi fel az előállított URL értékét, különben az aktuális JspWriterbe kerül. Ennek következtében használható az `<a>` HTML elemben is:

```
<a href="<c:url value='/add_question.jsp'/>">Add a question</a>
```

Olyan URL is előállítható, amely tartalmazza a HTTP GET kérés paramétereket. Ehhez a beágyazható `param` elemet kell felhasználnunk.

```
<c:url value="/search.jsp">
  <c:param name="keyword" value="${searchTerm}"/>
  <c:param name="language" value="EN"/>
</c:url>
```

Ha a webalkalmazás neve `education`, a `searchTerm` paraméter értéke pedig „pros and cons", és létezik a menethez tartozó süti, akkor a fenti kódrészletből előállított URL így fog kinézni:

```
/education/search.jsp?keyword=pros+and+cons&language=EN
```

Megfigyelhetjük, hogy az URL előállításánál a paraméterek hozzáfűzése a GET szabályai szerint történik, a szóköz helyett a + karakter jelenik meg, a paraméter neve után az =, a paraméterek elválasztása pedig az & karakterrel történik.

Ha nincs menethez tartozó süti, akkor az URL-hez hozzáadódik a menetazonosító is, és körülbelül így fog kinézni:

```
/education/search.jsp;
jsessionid=233379C7CD2D0ED2E9F3963906DB4290
?keyword=pros+and+cons&language=EN
```

A redirect elem

Ez a JSTL elem ekvivalens a sendRedirect() metódushívással (HttpServletResponse).

```
<c:redirect url="expression" context="expression">
  <c:param name="expression" value="expression"/>
  ...
</c:redirect>
```

Átirányítást végezhetünk a <jsp:forward> elemmel is, ebben az esetben viszont csak egy másik szerveroldali komponensnek adhatjuk át a vezérlést. A redirect elem a böngészőt utasítja átirányításra, így a böngésző címsorában is meg fog jelenni az új URL. Ha szerveroldalon a <jsp:forward> elemet használjuk, akkor ez a böngésző címsorát nem érinti.

```
<c:catch var="exception">
  <c:import url="http://www.ms.sapientia.ro"/>
</c:catch>
<c:if test="${not empty exception}">
  <c:redirect url="/errors/error.jsp"/>
</c:if>
```

6.4. Feladatok

6.4.1. Programozási feladat

Egy MVC architektúrára épülő webalkalmazás esetében a megjelenítést JSP lapokkal illik végezni. Az eddig fejlesztett webalkalmazásunkban a tanfolyamok listázását szervlettel végztük. Most, miután megismerkedtünk a JSP lapokkal, helyettesítsük a szervletet JSP lappal. Nyilván az `index.jsp` lapot is frissíteni kell, hiszen más néven kell hivatkoznunk a JSP lapra. Adjunk hozzá a távoktatás projekthez egy új JSP lapot, legyen ennek neve: `list_courses.jsp`. A feladat ugyanaz, mint a szervletben, a listaobjektum tartalmát megjelenítjük, figyelembe véve a menet hatókörében elhelyezett megjelenítési beállításokat. HTML táblázatot fogunk használni a tanfolyamok megjelenítésére. Mivel a felhasználó beállításainak függvénye, hogy a táblázat egy, kettő vagy három oszlopot tartalmaz, már a táblázat fejlécénél feltételes utasításokat fogunk használni. Csak azokat a fejlécelemeket fogjuk megjeleníteni, amelyeket a felhasználó kért (ezek a `showDesc`, illetve a `showPrice` változókban vannak a menet hatókörében). Nyilván ha a tanfolyamok megjelenítését a felhasználói beállítások előtt végezzük, akkor a `showDesc` és a `showPrice` változók nem jönnek létre. Az EL gyönyörűen kezeli az ilyen helyzeteket, a nem létező logikai változók értéke false lesz. Ezért amíg a felhasználó be nem állítja a megjelenítési óhajait, addig minden tanfolyamnak csak a neve fog megjelenni.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="java.util.*, model.Course"%>
<%@taglib prefix="c"
          uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body>
  <table>
    <tr id="list-header">
      <td>Name</td>
      <c:if test="${showDesc}">
      <td>Description</td>
      </c:if>
      <c:if test="${showPrice}">
      <td>Price</td>
      </c:if>
    </tr>
```

```
    <c:forEach var="e" items="${courselist}">
    <tr>
      <td><c:out value="${e.name}"/></td>
      <c:if test="${showDesc}">
      <td><c:out value="${e.description}"/></td>
      </c:if>
      <c:if test="${showPrice}">
      <td><c:out value="${e.price}"/></td>
      </c:if>
    </tr>
    </c:forEach>
  </table>
</body>
</html>
```

Mielőtt kipróbálnánk alkalmazásunkat, ne feledjük a következő két lépést elvégezni:

- Az `index.jsp` lapon cseréljük le a hivatkozást szervletről JSP lapra.

  ```
  <a href="list_courses.jsp">List Courses</a>
  ```

- Töröljük a szervletet az alkalmazásból. Ennek legegyszerűbb módja a Code Refactoring, amelyet minden modern integrált fejlesztési környezet biztosít.

6.4.2. Tesztkérdések

6.1. kérdés: Válasszuk ki a szintaktikailag helyes taglib direktívát!

```
A. <%@taglib id="h" uri="taglib.tld"%>
B. <%@taglib prefix="h" uri="taglib.tld"%>
C. <%@taglib prefix="h" url="taglib.tld"%>
D. <%@taglib prefix="h" uri="/taglib.tld" scope="session"%>
```

6.2. kérdés: Helyes-e a következő JSTL kódrészlet?

```
<c:if test="${param.locale == 'UK'}" >
<jsp:include page="ukmenu.jsp" />
</c:if>
<c:else test="${param.locale == 'US'}" >
```

```
<jsp:include page="usmenu.jsp" />
</c:else>
```

A. Igen
B. Nem

6.3. kérdés: Feltételezve, hogy a taglib direktíva helyes, mi történik a következő kódrészlet fordításakor? (1 helyes)

```
<%@taglib uri="http://java.sun.com/jstl/core_rt"
          prefix="c"%>
<html>
    <head>
        <title>The out and catch JSTL tags</title>
    </head>
    <body>
        <c:catch var="e">
            <%
                out.print(10/0);
            %>
        <c:out value="${e.message}"/>
        </c:catch>
    </body>
</html>
```

A. Fordítási hiba. El kifejezések és a JSTL elemek nem használhatók együtt.
B. Helyes fordítás, majd futásidőben az e változó kiíratása.
C. Fordítási hiba. Hiányzik a try rész.
D. Helyes fordítás, de futásidőben nincs kimenet, mert az out elem a catch elemen belül van.

6.4. kérdés: Feltételezve, hogy a taglib direktíva helyes, mi történik a következő kódrészlet fordításakor és futtatásakor? (1 helyes)

```
<%@taglib uri="http://java.sun.com/jstl/core_rt"
          prefix="c"%>
<jsp:useBean id="users" class="java.util.Vector" />
<html>
  <body>
    <%
     users.add("Paul");
```

```
     users.add("Kathy");
     users.add("Frank");
    %>
    <c:forEach var="name" items ="${users}"
               varStatus="counter">
      <c:out value="${counter.count}"/>
      <c:out value="${name}"/>
    </c:forEach>
  </body>
</html>
```

A. Fordítási hiba. El kifejezések és a JSTL elemek nem használhatók együtt.
B. Fordítási hiba. Hiányzik a forEach elem begin attribútuma.
C. Fordítási hiba. Szkriptletben nem hivatkozhatunk Java babra.
D. Helyes fordítás és a kimenet: 1 Paul 2 Kathy 3 Frank

6.5. kérdés: Melyik kódsor használható URL újraírásra? (1 helyes)

A. <c:code var="url1" value="www.someurl.com"/>
B. <c:session var="url1" value="www.someurl.com"/>
C. <c:encode var="url1" value="www.someurl.com"/>
D. <c:url var="url1" value="www.someurl.com"/>

6.6. kérdés: Válasszuk ki a helyesen megadott import elemet! (1 helyes)

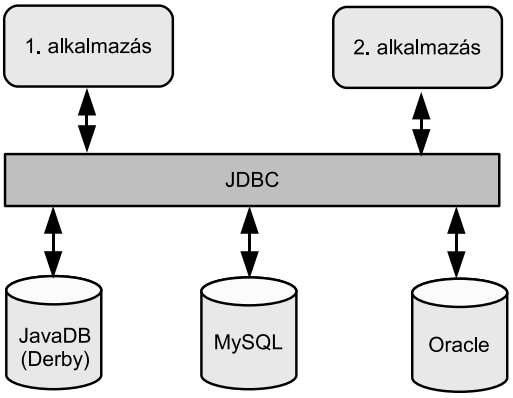
A. <c:import url="http://www.jchq.net"/>
B. <c:import file="http://www.jchq.net"/>
C. <c:import page="http://www.jchq.net"/>
D. <c:import target="http://www.jchq.net"/>

7. FEJEZET

ADATBÁZISOK HASZNÁLATA WEBALKALMAZÁSOKBAN

7.1. A JDBC API

A JDBC API megteremti a lehetőséget relációs adatbázisoknak platform- és adatbáziskezelő-független elérésére. Maga a JDBC API hangsúlyosan csak interfészeket deklarál, amelyeket az adatbázis-kezelő rendszereket szolgáltatók implementálnak driverek formájában. A 7.1. ábra a JDBC architektúrát szemlélteti, kihangsúlyozva, hogy bármelyik adatbázisszerver ugyanazon JDBC API segítségével érhető el.



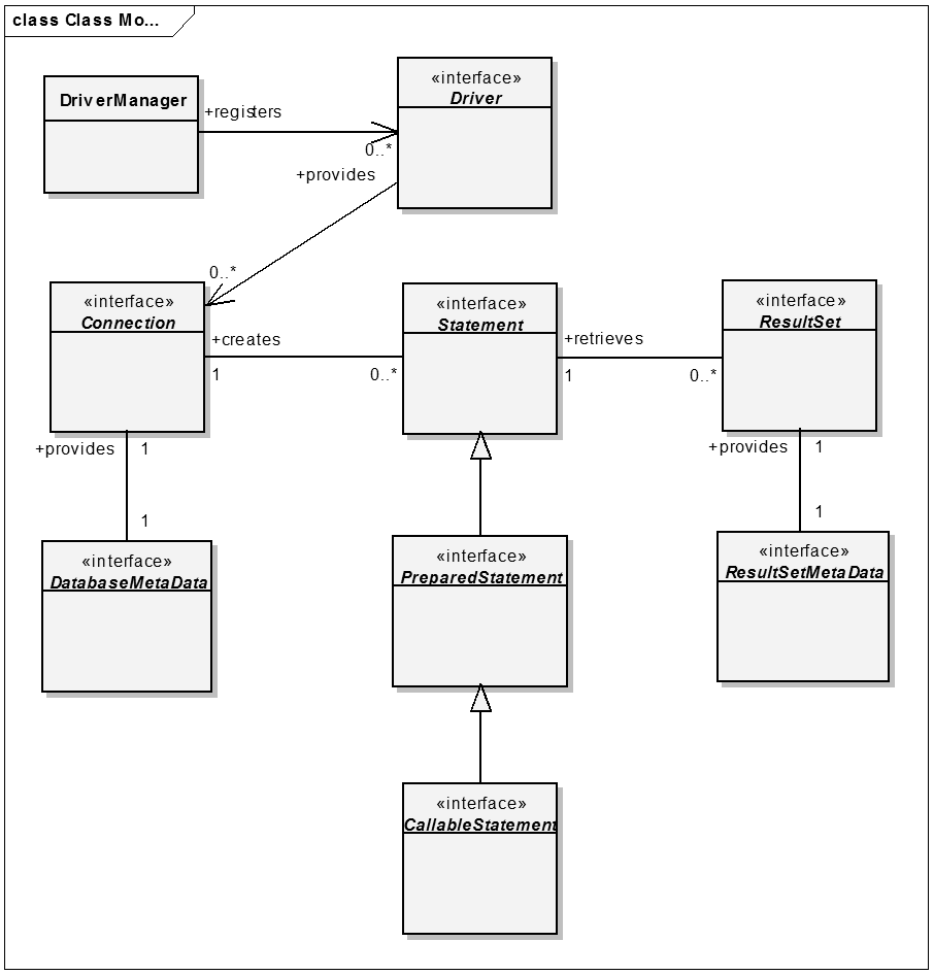
7.1. ábra. *A JDBC architektúra*

A JDBC API két szintjéről beszélhetünk, amelyek szervezésileg is két különböző csomagban vannak:

– A `java.sql` csomag az adatbázisok eléréséhez szükséges alapvető osztályokat tartalmazza.

– A `javax.sql` csomag haladó szintű osztályokat tartalmaz (pl. `DataSource`).

Az alapvető osztályokat a 7.2. ábra szemlélteti.



7.2. ábra. *A JDBC API, alapvető osztályok*

A JDBC API szolgáltatásait három kategóriába sorolhatjuk:
– adatbázis-kapcsolat megteremtését segítő típusok,
– SQL utasítások végrehajtását segítő típusok,
– SQL lekérdezések eredményeinek feldolgozását segítő típusok.

JDBC meghajtóprogramok

Nagyon sok adatbázis-kezelő rendszer (szerver) létezik. A Java specifikálja interfészek segítségével, hogy milyen típusokra van szükség az adatbázisok eléréséhez, az adatbázisszervert készítő pedig megvalósítja ezeket egy meghajtóprogram (driver) formájában. Ezeket a meghajóprogramokat `jar` csomagok formájában teszik közzé az adatbázisszerver készítői. Nyilván ahhoz, hogy az alkalmazásunk kommunikálni tudjon egy adott adatbázis-kezelővel, le kell tölteni a megfelelő meghajtóprogramot, és elérhetővé kell tenni a Java alkalmazás számára. (NetBeans projekt esetén ez úgy oldható meg, hogy ezt a `jar` állományt hozzáadjuk a projekthez.)

Példák adatbázis-kezelő meghajtóprogramokra:

- Derby (Java DB): `derbyclient.jar`
- MySql : `mysql-connector-java-5.1.7-bin.jar`

Egy JDBC adatbázis-kapcsolat a következő műveletek végrehajtását jelenti:

- adatbázis-meghajtó betöltése (driver regisztráció),
- adatbázis-kapcsolat kiépítése,
- egy SQL művelet végrehajtásához szükséges Statement példány létrehozása (ez lehet Statement, PreparedStatement, CallableStatement),
- SQL művelet eredményének feldolgozása (ez SQL utasításfüggő),
- az összes, adatbázissal kapcsolatos objektum lezárása: ResultSet, Statement és Connection.

Az első lépés az adatbázis driver betöltése, ezt két adatbázis-kezelő esetére megadjuk. Az egyik a Derby adatbázis-kezelő rendszer, ezt még Java DB-nek is nevezzük, és a NetBeans 6 fölötti változataival egyszerre letölthető és telepíthető. A nagy előnye ennek az adatbázis-kezelőnek, hogy hordozható. Minden egyes adatbázist egy külön mappában tárol, ha ezt átmásoljuk egy másik számítógépre a Derby adatbázisok közé, már meg is oldottuk a hordozhatóságot. A másik adatbázis-kezelő, amelyre még fogunk hivatkozni, a MySQL. Ennek is van szabadon letölthető változata, és nagy előszeretettel használják PHP webalkalmazások esetében is.

Meghajtó betöltése – Driver regisztráció

```
Derby: Class.forName("org.apache.derby.jdbc.ClientDriver");
```

```
MySql: Class.forName("com.mysql.jdbc.Driver");
```

Adatbázis-kapcsolat létrehozása

A JDBC programokban az adatbázisokra a már ismert, kibővített URL szintaxissal lehet hivatkozni:

```
jdbc:<alprotokoll>:<adatbázis-hivatkozás>
```

Az alprotokoll neve a meghajtók kiválasztásánál játszhat szerepet, hiszen a JDBC egyes kapcsolatokhoz más-más meghajtót használhat. Az adatbázis-hivatkozás pedig már meghajtóspecifikus szöveg lehet, bár az ajánlások szerint ahol lehet, használjuk a szokásos URL szintaxist. Példák:

- Derby adatbázis URL: `jdbc:derby://localhost:1527//distedu`
- MySql adatbázis URL: `jdbc:mysql://localhost:3306/distedu`

Egy Java alkalmazás több típusú adatbázis-kezelőt is használhat, betöltve mindenik esetében a megfelelő drivert, majd kiépítve az adatbázis-kapcsolatot. A kapcsolat kiépítése úgy történik, hogy meghívjuk a `DriverManager` osztálynak a `getConnection` metódusát, átadva az előző részben szemléltetett adatbázis URL-t, az adatbázishoz szükséges felhasználói nevet és jelszót.

```
String url = "jdbc:derby://localhost:1527//distedu";
Connection con = DriverManager.getConnection(
                  url,"username","password");
```

Adatbázis-lekérdezések végrehajtása

A következő lépés egy adatbázis-lekérdezés objektum létrehozása, amelyen keresztül majd végrehajtjuk a lekérdezéseket. Ez a következő három osztály valamelyikének a példánya kell legyen (mindhárom a `java.sql` csomagban található):

- statikus SQL utasítás: `Statement`,
- dinamikusan paraméterezett SQL utasítás: `PreparedStatement`,
- tárolt eljáráshívás: `CallableStatement`.

A példáknál a statikus SQL utasítások végrehajtásához szükséges `Statement` osztályt fogjuk használni. Az adatbázis- lekérdezéséhez szükséges objektumot a `Connection` objektumtól kaphatunk.

```
Statement stat  = con.createStatement();
```

A `Statement` osztály metódusainak segítségével lehet SQL utasításokat végrehajtani. Eredménytáblákat visszaadó SQL utasításokat (SELECT) az `executeQuery` metódussal hívunk. Adatmanipulációs (UPDATE, DELETE) és adatdefiníciós (CREATE TABLE, DROP TABLE) utasítások futtatására pedig az `executeUpdate` metódus használható.

```
ResultSet executeQuery( String sql )
int executeUpdate( String sql )
```

Eredménytáblák lekérdezése esetén egy `ResultSet` típusú objektumból nyerhetjük ki soronként a lekérdezés eredményét. Az `executeUpdate` metódus eredménye egy egész szám, amely az adatmanipulációban érintett rekordok számát jelenti.

A ResultSet objektumot úgy képzelhetjük el, mint egy tömböt, amely rekordokat tartalmaz, és amelyek között egy kurzor mozgatásával navigálhatunk. Kezdetben ez a kurzor az első rekord elé(!) mutat. A kurzor mozgatására a következő metódusok használhatók:

- `next()`: a következő rekordra lép, ha ez lehetséges, különben false,
- `previous()`: az előző rekordra lép, ha ez lehetséges, különben false,
- `last()`: az utolsó rekordra lép, ha nem üres a lekérdezés tábla, különben false,
- `first()`: az első rekordra lép, ha nem üres a lekérdezés tábla, különben false.

Most pedig tekintsünk példát egy egyszerű lekérdezés eredményének feldolgozására. Legyen egy `Course` nevű adattáblánk, amelyben az előző fejezetekben szemléltetett tanfolyamadatokat tárolunk. Tételezzük fel azt is, hogy az adattáblánkat a következőképpen hoztuk létre (Derby DB):

```
CREATE TABLE course (
   id int NOT NULL GENERATED ALWAYS AS IDENTITY,
   name varchar(100) NOT NULL,
   description varchar(200) NOT NULL,
   price double NOT NULL
)
```

Az adatbázis-kapcsolatot megteremtettük, a fenti adattáblát már feltöltöttük, most pedig lekérdezzük a tábla tartalmát:

```
try{
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("select * from course");
   while( rs.next() ){
    System.out.println(
    "ID: "+rs.getInt(1)+
    "NAME: "+rs.getString(2)+
    "DESCRIPTION: "+rs.getString(3)+
    "PRICE: "+rs.getDouble(4)
   );
  }
}
catch (Exception e) {
  e.printStackTrace();
}
```

A tábla összes adatának a lekérdezését az `executeQuery` művelettel végezzük. Ezt követően pedig egy ciklusban léptetjük a kurzort a lekérdezés következő sorára, kinyerjük, majd pedig kiíratjuk az adatokat. Az adatok kinyerése többféleképpen is végezhető: a lekérdezés oszlopaira hivatkozhatunk pozícióval, illetve az oszlop nevével. Mi az előbbit választottuk, de használhattuk volna a következőket is:

- `getInt("ID");`
- `getString("NAME");`
- `getString("DESCRIPTION");`
- `getDouble("PRICE");`

Nézzük meg, hogy mindig használható-e a lekérdezés feldolgozásához a `while( rs.next())...` séma. Mi történik például, ha a lekérdezés eredménye üres? Ilyen esetben bizony nem így kell használni.

```
try{
   Statement stmt = con.createStatement();
   ResultSet rs =
   stmt.executeQuery("select * from course where name='xx'");

   if( rs.next() ){
     System.out.println(
     "Az xx tanfolyam ára: "+rs.getDouble(4));
   }
```

```
   else{
     System.out.println("Nem létezik xx nevű tanfolyam");
   }
}
catch (Exception e) {
  e.printStackTrace();
}
```

Vegyünk most egy olyan példát, amelyben a lekérdezés eredménye nem üres, hanem NULL. Legyen az a feladatunk, hogy kérdezzük le a legdrágább tanfolyam árát. Ha például még üres a tanfolyamok táblája, akkor ennek a lekérdezésnek az eredménye egy 1 X 1-es tömb, NULL értékkel.

```
try{
   Statement stmt = con.createStatement();
   ResultSet rs =
     stmt.executeQuery("select max(price) from course");
   rs.next();
   double maxPrice = rs.getDouble(1);
   if( rs.wasNULL()){
      //Üres a tábla
   }
   else{
      //Mehet a feldolgozás
   }
}
catch (Exception e) {
  e.printStackTrace();
}
```

Felmerül a kérdés, hogy miért nem lehet a `ResultSet` objektumtól lekérdezni, hogy hány soros az eredmény. Ennek az az oka, hogy nem minden adatbázis-kezelő mondja meg ezt.

Most pedig tekintsünk egy példát adatmanipuláló SQL eljárásra, amelyen keresztül beszúrunk egy rekordot a `course` adattáblába:

```
try{
   Statement stmt = con.createStatement();
   int n = stmt.executeUpdate(
```

```
   "insert into course (name, description, price)
    values ('Java SE','Java Standard Edition', 1000)");
   //Ha n értéke 1, akkor sikeres volt a beszúrás
}
catch (Exception e) {
  e.printStackTrace();
}
```

Dinamikusan paraméterezett lekérdezések

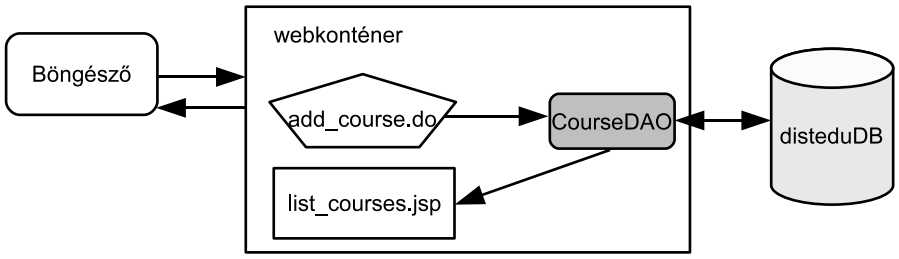
Két fontos különbség van a `Statement` és a `PreparedStatement` között. Az első különbség, hogy a `PreparedStatement` esetében már a konstruktornak meg kell adni az SQL utasítást, a második pedig az SQL utasításban szereplő ? karakterek. Minden kérdőjel egy paraméter helyén áll, és ezt a lekérdezés előtt be kell állítani. Az adatbázis oldalon maga a lekérdezés csak egyszer jön létre és minden végrehajtás előtt behelyettesíti a paramétereket. Ezek a típusú lekérdezések gyakran hatékonyabban vannak implementálva adatbázisokban és biztonságosabbak is.

Most pedig nézzük meg az előző példánkat, az új tanfolyam beszúrását, `PreparedStatement` segítségével.

```
String sql_comm = "insert into Course (name,description,price)
values (?,?,?)";
try {
   //Statement stmt = con.createStatement();
   PreparedStatement pstmt = con.prepareStatement(sql_comm);
   pstmt.setString(1, c.getName());
   pstmt.setString(2, c.getDescription());
   pstmt.setDouble(3, c.getPrice());
   int res = pstmt.executeUpdate();
}
catch (Exception e) {
   e.printStackTrace();
}
```

7.2. A Data Access Object tervezési minta

A Data Access Object (DAO) tervezési minta lényege az, hogy elválasztja az üzleti logikát (az adatok feldolgozását) az adattárolási logikától (l. 7.3. ábra). A DAO tartalmazza az összes adatelérési és manipulálási műveletet, az alkalmazás üzleti logikáját képviselő komponensek pedig nem tartalmaznak semmiféle ilyen műveletet. Ennek az a következménye, hogy ha megváltozik valami az adatrétegben, például más adatbázisszerverre helyezzük adatainakat, csak a DAO komponenseket kell módosítani, az üzleti logika változatlan marad. Ez nagymértékben növeli az alkalmazás rugalmasságát.



7.3. ábra. *A DAO szerepe*

Mit is tartalmaz egy ilyen DAO osztály? Legegyszerűbb esetben egy DAO osztály egy adattábával tartja a kapcsolatot és az adattáblával végezhető műveleteket tartalmazza. Például új rekord felvitele, az összes rekord lekérdezése és különböző keresési műveleteket. Egyszóval minden olyan műveletet, amely az alkalmazásban hasznos lehet, ezenfelül pedig az adatbázis-kapcsolatra vonatkozó műveleteket is itt helyezhetjük el.

Kezdjük a bemutatást ezen utóbbival. Az osztályban konstansként fogjuk bevezetni az adatbázis-kapcsolathoz szükséges adatokat: driver, url, felhasználónév, illetve jelszó. Két konstruktort használunk, az egyik megkapja paraméterként a fenti adatokat, a másik az osztályban bevezetett konstansokat fogja használni. Mi ezt az utóbbit fogjuk használni a későbbiekben. A konstruktorokat követő két metódus a `connect` és a `disconnect`, ezek az adatbázis-kapcsolat kiépítéséért, illetve megszüntetéséért lesznek felelősek.

```
package model;

import java.sql.*;
import java.util.*;

public class CourseDAO {
   public static final String DRIVER =
                     "org.apache.derby.jdbc.ClientDriver";
   public static final String URL =
                   "jdbc:derby://localhost:1527/disteduDB";
   public static final String USERNAME ="public";
   public static final String PASSWORD ="public";

   private String driver;
   private String url;
   private String userName;
   private String password;
   private Connection con;


   public CourseDAO(){
     driver =DRIVER;
     url = URL;
     userName = USERNAME;
     password = PASSWORD;
   }

   public CourseDAO( String driver, String url,
                     String userName, String password){
     this.driver = driver;
     this.url = url;
     this.userName = userName;
     this.password = password;
   }

   public void connect(){
     try{
       Class.forName( driver );
       con = DriverManager.getConnection(
              url,userName, password);
     }
     catch ( SQLException e1 ){
```

```
        System.out.println(
         "Hiba a kapcsolat kiepitesenel: "+url);
      }
      catch( ClassNotFoundException e2 ){
        System.out.println(
          "Hiba a driver betoltesenel: "+driver);
      }
    }


    public void disConnect(){
      if( con == null ) return;
      try{
        con.close();
      }
      catch( SQLException e){
        System.out.println(
         "Hiba a kapcsolat megszuntetesenel\n");
      }
  }

  //...
}
```

Most pedig készítsünk el két adatmanipulációs műveletet, az egyik egy új rekord felvitelét végzi, a másik pedig az összes rekord lekérdezését. Tételezzük fel azt is, hogy módosítottuk a `Course` modell osztályunkat, bevezettük az `id` egész típusú adatmezőt, ennek megfelelően módosítottuk a konstruktort, és bevezettünk egy ennek megfelelő `getId` műveletet. A `setId` műveletet azért nem vezettük be, mert az adattábla létrehozásakor ezt úgy kértük, hogy az adatbázis-kezelő rendszer automatikusan inkrementálja.

```
public List<Course> getAllCourses(){
      //Kapcsolat
      connect();
      if( con == null )  return null;
      //SQL parancs
      String query ="select * from course";
      //SQL parancs vegrehajtasa
      Statement stmt = null;
      ResultSet rs = null;
      List<Course> list = new ArrayList<Course>();
```

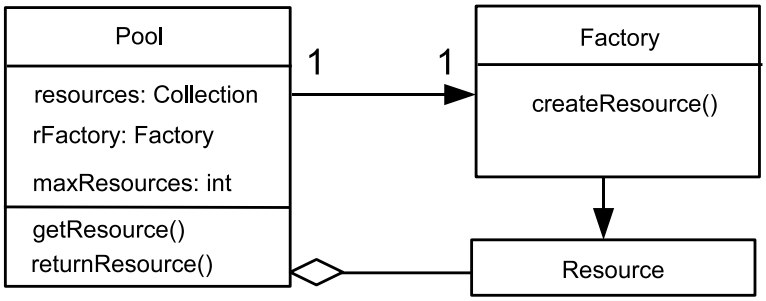
```
      try{
        stmt = con.createStatement();
        rs = stmt.executeQuery(query);
        while( rs.next()){
          int id = rs.getInt(1);
          String name = rs.getString(2);
          String description = rs.getString(3);
          double price = rs.getDouble(4);
          list.add(
            new Course( id, name, description, price));
        }
      }
      catch( SQLException e ){
        System.out.println("SQL hiba:"+query);
      }
      finally{
        //Lezarasok
        if( rs != null ){
            try{
                rs.close();
            }catch( SQLException e ){}
        }
        if( stmt != null ){
            try{
                stmt.close();
            }catch( SQLException e ){}
        }
        disConnect();
      }
      return list;
}


public boolean insertCourse(Course c) {
    //Kapcsolat
    connect();
    if(con == null) return false;
    //SQL parancs felepitese
    StringBuffer sql_comm = new StringBuffer("insert into
      Course (name, description, price) values ('");
    sql_comm.append(c.getName()+"','");
    sql_comm.append(c.getDescription()+"',");
```

```
    sql_comm.append(c.getPrice()+")");
    System.out.println(sql_comm);
    //SQL parancs vegrehajtasa
    Statement stmt=null;
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(sql_comm.toString());
    }
    catch (SQLException e) {
        System.out.println("SQL insert error - " + e);
        return false;
    }
    finally{
        //Lezarasok
        if( stmt != null ){
            try{
                stmt.close();
            }catch( SQLException e ){}
        }
        disConnect();
    }
    return true;
}
```

7.3. Adatforrás (DataSource)

Miért is van szükségünk egy újabb komponensre? Az előzőekben megismerkedtünk adatbázis-kapcsolatok létrehozásával, most pedig nézzük meg, hogy milyen problémák merülnek fel osztott rendszerek esetében. Először is egy osztott rendszert egyidejűleg több felhasználó is kezelhet. Minden új felhasználónak kiépíteni, majd megszüntetni az adatbázis-kapcsolatot teljesítményproblémákhoz vezethet. Jó lenne, ha az adatbázis-kapcsolatok újrafelhasználhatók lennének. Webalkalmazásokban lenne is erre egy megoldás, például az adatbázis-kapcsolatot azon szervlet inicializálásakor létrehozni, amely ezt igényli. Ez viszont megoldja ugyan azt a problémát, hogy nem kell minden egyes ügyfél kiszolgálásakor létrehozni a kapcsolatot, de párhuzamos kiszolgálások esetén szinkronizálást igényel. Egy második megoldás az lenne, ha az

alkalmazás indításakor már létrehoznánk egy kapcsolatkészletet (connection pool), ezt regisztrálnánk a webalkalmazás hatókörébe, és innen igény szerint lehetne kivenni, majd használat után visszatenni a már előzőleg kiépített adatbázis-kapcsolatokat. Ez a megoldás már működőképes is lenne, az egyetlen bökkenő az, hogy így csak a webréteg komponensei számára lesznek elérhetők az adatbázis-kapcsolatok.

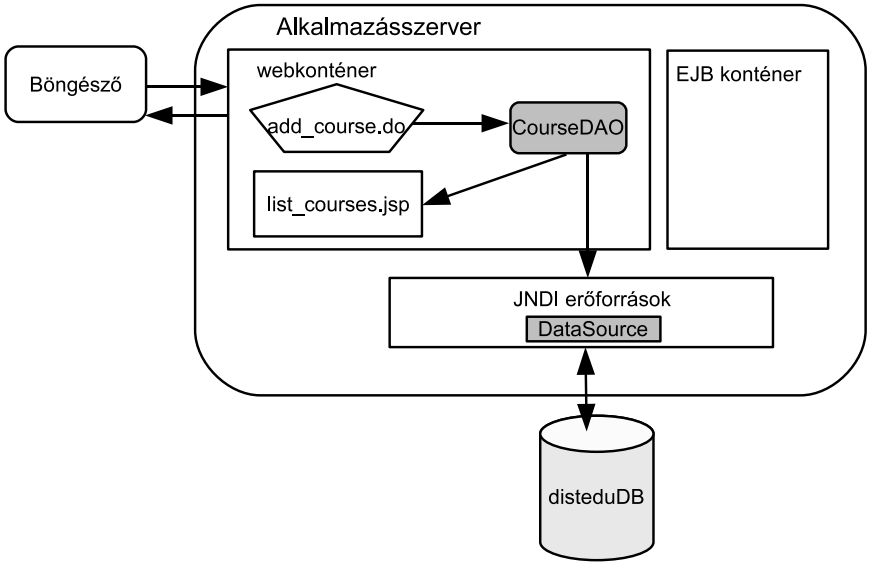


7.4. ábra. *A Connection Pool tervezési minta [2]*

Az elegáns megoldás pedig `DataSource` komponens használata `ConnectionPool` mechanizmussal együtt. Ez két dolgot jelent: elsősorban azt, hogy egy névszerverben regisztrálunk egy komponenst, amely név alapján elérhető lesz egy osztott alkalmazás összes komponense számára. Másodsorban pedig az adatforráshoz hozzárendelünk egy adatbázis-kapcsolathalmazt, amelyet igény szerint méretezhetünk. Az ötlet nem új, hiszen erre a problémára már tervezési minta is született, a Resource Pool (erőforrás-gyűjtemény). A 7.4. ábra egy ilyen tervezési minta megvalósításához szükséges osztályokat szemlélteti. A `Pool` osztály tárolja az erőforrásokat a `resources` gyűjteményben és használ egy erőforrás-gyártót (`Factory`), amely segítségével feltölti az erőforrás-gyűjteményét, illetve szükség esetén bővíti ezt. A `Resource` osztály maga a gyártott erőforrás absztrakciója.

A DataSource egy olyan komponens, amely egy adatbázis-kapcsolathoz szükséges információkat egységbe zárja: adatbázis URL, driver, felhasználónév és jelszó, és az adatbázis-kapcsolatkészlet kezelését is biztosítja egyúttal. Ezt a komponenst egy névszerverbe kell regisztrálni. A névszerver feladata erőforrások név alapján történő regisztrációja és név alapján történő előkeresése. A Java erre a célra kialakított technológiája a JNDI Java Naming and Directory Interface. Ez lehetővé teszi,

hogy az adatforrás karbantartása elkülönüljön az alkalmazástól. A JNDI szolgáltatást minden alkalmazászerver biztosítja, így a Sun Glassfish alkalmazásszervere is. A 7.5. ábrán jól látható, hogyan változik meg az adatbázis-hozzáférés adatforrás használatakor. Ugyanakkor az ábra azt is jól szemlélteti, hogy az adatforrás az alkalmazásszerver egy komponense lesz, nem pedig a webkonténeré, ezért elérhető az alkalmazásszerver bármely konténere számára.



7.5. ábra. *Adatforrás (DataSource)*

Milyen előnyökkel jár a DataSource használata? Először is kevesebb munkát kell a programozónak befektetnie, hiszen a rendszer skálázhatóságához égetően fontos Connection Pool mechanizmust nem kell implementálnia. Másodsorban az alkalmazás hordozható lesz, bármely Java EE alkalmazásszerveren működni fog. Ehhez csak annyit kell tenni, hogy az alkalmazásszerverben létre kell hozni a megfelelő nevű DataSource erőforrást, elérhetővé tenni az adatbázist és telepíteni a webalkalmazást. Ez utóbbi egyetlen `war` csomag bemásolását jelenti a megfelelő mappába. Ezek a műveletek kényelmesen végezhetők a Glassfish alkalmazásszerver adminisztrációs felületéről. Mivel a DataSource komponens elkészítése alkalmazásszerver-függő, ezért csak a használatát ismertetjük.

A DataSource interfészt a javax.sql csomag tartalmazza.

```
public interface DataSource{
Connection getConnection();
Connection getConnection(String name, String password);
...
}
```

Most pedig tekintsük át, hogyan is történik az adatbázis-kapcsolat megszerzése:

1. referencia megszerzése a DataSource komponensre JNDI segítségével:

   ```
   DataSource datasource = null;
   ...
   try{
     Context ctx = new InitialContext();
     if( ctx == null )
           throw new RuntimeException
             ("JNDI Context coluld not be found");
     datasource = (DataSource) ctx.lookup("jdbc/disteduDS");
   }
   catch (Exception e){
     e.printStackTrace();
   }
   ```

2. getConnection metódus hívása a DataSource objektumra, adatbázis-művelet elvégzése, majd a kapcsolat bezárása:

   ```
   Connection con = null;
   try {
     con = datasource.getConnection();
     //Muveletek
   }
   catch( SQLException e ){
   e.printStackTrace();
   }
   finally {
     if(con != null){
       try {
         con.close();
   ```

```
            }
            catch (Exception e) {
               e.printStackTrace();
            }
         }
      }
```

A kapcsolatokat azért a finally blokkban zárjuk le, mert ez a blokk mindenképpen végrehajtódik. Ennek következtében pedig minden lekötött erőforrást felszabadítunk.

7.4. Megoldott feladatok

7.1. feladat. Adatbázis létrehozása: Adatbázis létrehozására a Derby adatbázis-kezelőt használjuk. Ezt pedig a következőképpen kell létrehozni:

```
Services fül-->JavaDB-->Create Database
```

Ennél a lépésnél meg kell adni a következőket: adatbázisnév, felhasználónév és jelszó. A továbbiakban mi a következő értékeket használjuk:

```
Database Name: disteduDB
User Name: public
Password: public
```

A dialógusdobozban a fentiek megadása után vegyük észre azt is, hogy hol fogja az adatbázisszerver tárolni az adatbázisunkat. Erre azért érdemes figyelni, mert ha át akarjuk vinni az adatbázist egy másik gépre, akkor csak a fenti mappát kell átmásolni a célgépre. A Derby adatbázisok ilyen egyszerűen hordozhatók.

Az adatbázistábla létrehozása után a JavaDB alatti listában megjelenik az adatbázis neve. Erre jobb egérkattintással lehet kapcsolódni. Ha megvan a kapcsolódás, a felbukkanó menüből szintén jobb egérkattintással kiválasztjuk az Execute Command menüpontot. Ez lehetővé teszi, hogy egy parancsablakban SQL parancsokat hajtsunk végre.

Az első parancs legyen az adattábla létrehozása:

```
CREATE TABLE course (
   id int NOT NULL GENERATED ALWAYS AS IDENTITY,
   name varchar(100) NOT NULL,
   description varchar(200) NOT NULL,
   price double NOT NULL
)
```

7.2. feladat. Adattábla feltöltése: Az adattábla feltöltését végezzük szintén az előző pontban megnyitott parancsablakban, beírva a következőhöz hasonló SQL INSERT parancsokat.

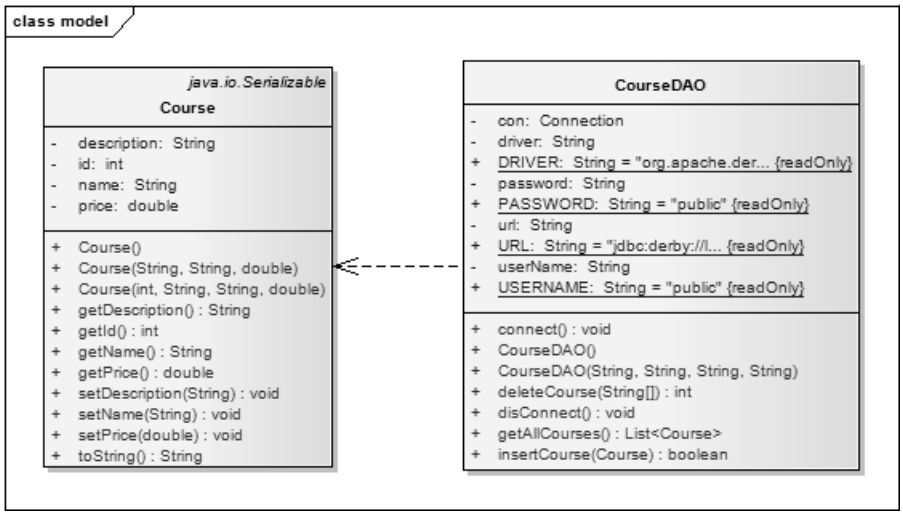
```
insert into course (name, description, price)
  values ('Java SE', 'Java Standard Edition', 1000)
```

Figyeljük meg, hogy mivel az ID generálását automatikusra állítottuk, ezért ennek nem adunk mi értéket.

7.3. feladat. A Course osztály módosítása: A későbbiekben szükségünk lesz a kurzusobjektum `id` mezőjének írására, illetve olvasására, ezért módosítsuk a `Course.java` osztályt a `model` csomagból. Adjuk hozzá az `id` attribútumot, illetve az ennek megfelelő `get` és `set` metódusokat. Elkészítünk egy 4 paraméteres konstruktort is.

```
class Course{
    private int id;
    ...
    public Course( int id, String name,
            String description, double price){
        this.id = id;
        this.name = name;
        this.description = description;
        this.price = price;
    }
    public void setId(int id){
        this.id = id;
    }

    public int getId(){
        return id;
    }
}
```

7.4. feladat. A CourseDAO osztály létrehozása: A DAO osztályt a 7.2. alfejezetben leírtak alapján készítjük el, felhasználva az ott megadott adattagokat, illetve műveleteket, kiegészítve még egy törlést végző metódussal. Az eddigi modell rész osztálydiagramját a 7.6. ábra szemlélteti.



7.6. ábra. *A CourseDAO és a Course osztályok*

Most pedig nézzük meg, hogy hogyan végezzük a törlést. Úgy tervezzük a metódust, hogy egyszerre több tanfolyam törlését is lehetővé tegyük. Ezért a `deleteCourse` metódusnak nem egy id, hanem egy id tömb paramétere van. A kényelmesség kedvéért ezeket az id-ket karakterláncok formájában fogjuk átadni:

```
public int deleteCourse(String[] ids) {
    //Kapcsolat
    connect();
    if(con == null) return -1;
    //SQL parancs eloallitasa
    Statement stmt = null;
    if (ids.length == 0) return 0;
    String sql = "delete from course where
                    (id in (" + ids[0];
    for (int i = 1; i < ids.length; ++i)
        sql += ", " + ids[i];
    sql += "))";
```

```
    //SQL parancs vegrehajtasa
    int result;
    try {
        stmt = con.createStatement();
        result = stmt.executeUpdate(sql);
    }
    catch (SQLException e) {
        System.out.println("SQL insert error - " + e);
        result = -1;
    }
    finally{
        //Lezarasok
        if( stmt != null ){
            try{
                stmt.close();
            }catch( SQLException e ){}
        }
        disConnect();
    }
    return result;
}
```

7.5. feladat. A list_courses.jsp lap módosítása: Mivel adatbázisból kell venni a tanfolyamadatokat, ezért módosítanunk kell a tanfolyamok listázását végző JSP lapot. A `courselist` objektumot a listázás előtt fogjuk előállítani a DAO objektum segítségével. Ezt egy rövid szkriptlettel fogjuk végezni, amely ugyan nem elegáns és az MVC elvet sem tartja be, de az egyszerűség végett ezt a megoldást választottuk.

```
<%
   CourseDAO dao = new CourseDAO();
   List<Course> list = dao.getAllCourses();
   pageContext.setAttribute("courselist", list);
%>
```

Ez lényegében példányosítja a `CourseDAO` osztályt, lekéri a tanfolyamok listáját, majd a lap hatókörében `courselist` név alatt regisztrálja. A teljes JSP lap pedig a következő:

```
<%@page contentType="text/html"
        pageEncoding="UTF-8"%>
```

```
<%@page import="java.util.*, model.*"%>
<%@taglib prefix="c"
          uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body>

  <%
    CourseDAO dao = new CourseDAO();
    List<Course> list = dao.getAllCourses();
    pageContext.setAttribute("courselist", list);

  %>
   <p><a href="index.jsp">Main Page</a></p>
  <table>
    <tr id="list-header">
    <td>Name</td>
    <c:if test="${showDesc}">
      <td>Description</td>
    </c:if>
    <c:if test="${showPrice}">
      <td>Price</td>
    </c:if>
    </tr>
    <c:forEach var="e" items="${courselist}">
    <tr>
      <td><c:out value="${e.name}"/></td>
      <c:if test="${showDesc}">
        <td><c:out value="${e.description}"/></td>
      </c:if>
      <c:if test="${showPrice}">
        <td><c:out value="${e.price}"/></td>
      </c:if>
    </tr>
    </c:forEach>
  </table>
</body>
</html>
```

7.6. feladat. Az AddCourse szervlet módosítása: Nézzük meg, milyen módosítások szükségesek a tanfolyamok halmazát bővítő szervletben. Az egyetlen különbség az, hogy most nem egy listához adjuk

hozzá az új tanfolyamot, hanem egy adatbázishoz. Ezt pedig úgy tudjuk megtenni, hogy példányosítjuk a `CourseDAO` osztályt és meghívjuk az `insertCourse` metódusát:

```
Course course = new Course(name, description, price);
CourseDAO dao = new CourseDAO();
boolean b = dao.insertCourse(course);
```

Figyeljük meg, hogy az `insertCourse` metódus hiba esetén (pl. adatbázis-kapcsolat) `false` értéket térít vissza. Így lehetőség van ilyen jellegű hiba esetén is a vezérlést egy hibalapnak továbbítani. Ennek megoldását az olvasóra bízzuk.

7.7. feladat. Tanfolyamok törlése: A `CourseDAO` osztály már elő van készítve törlésre, most már csak az ennek megfelelő megjelenítési, illetve vezérlési részt kell megvalósítanunk. Kezdjük az egyszerűbbel, a megjelenítéssel:

```
<%@page contentType="text/html"
        pageEncoding="UTF-8"%>
<%@taglib prefix="c"
          uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
   Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@ page import = "java.util.*, model.*" %>
<html>
    <body>
    <%
      CourseDAO dao = new CourseDAO();
      List<Course> cl = dao.getAllCourses();
      pageContext.setAttribute("courselist", cl);
    %>

    <h2>Which courses would you like to delete?</h2>
    <form action="delete_course.do" method="post">
       <c:forEach var = "v" items ="${courselist}" >
           <input type="checkbox" name="courses"
                  value="${v.id}" >
           <c:out value="${v.name}"/> <br><br>
       </c:forEach>
```

```
       <input type="submit" name="submit"
              value="Delete">
    </form>
    </body>
</html>
```

Figyeljük meg, hogy a JSP lap egy szkriptlettel kezdődik, amelyben a CorseDAO példánytól lekérdezzük a tanfolyamok listáját. Mivel JSTL elemek segítségével akarjuk ezt a listát feldolgozni, ezért regisztráljuk (betesszük) a listát courselist név alatt a lap hatókörébe.

```
<%
  CourseDAO dao = new CourseDAO();
  List<Course> cl = dao.getAllCourses();
  pageContext.setAttribute("courselist", cl);
%>
```

Most pedig elkészítjük azt a vezérlési komponenst, szervletet, amely feldolgozza az űrlapadatokat és elvégzi a törlést az adatbázisban. Ez egy nagyon rövid szervlet lesz, hiszen nincs szükség az űrlap paramétereinek ellenőrzésére. Az űrlapból érkező paramétereket egyszerűen továbbítjuk a DAO objektum törlési metódusának.

```
package controller;

import model.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class DeleteCourse extends HttpServlet {

   protected void doPost(HttpServletRequest request,
                         HttpServletResponse response)
   throws ServletException, IOException {
     String[] courseId =
              request.getParameterValues("courses");
     if( courseId != null ){
            CourseDAO dao = new CourseDAO();
            dao.deleteCourse( courseId);
     }
```

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

7.8. feladat. Az ApplicationListener törlése: Töröljük az `ApplicationListener` nevű komponenst. Eddig ez végezte a tanfolyamadatok betöltését, illetve mentését, a továbbiakban erre nincs szükségünk, hiszen adatbázissal dolgozunk. A törlést végezzük `Code Refactoring` művelettel, így nemcsak a forrásállomány törlődik, hanem annak összes előfordulása, például a telepítésleíróban megadott konfiguráció is.

7.9. feladat. DataSource készítése: Adjunk hozzá az alkalmazáshoz egy DataSource komponenst, majd módosítsuk a `CourseDAO` osztályt úgy, hogy használja ezt az adatbázis-kapcsolatokra.

`File -> New File ->Glassfish -> JDBC Resource`

A módosított `CourseDAO` osztály:

```
package model;

import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.naming.*;

public class CourseDAO {
   private DataSource datasource;

   public CourseDAO(){
     try{
       Context ctx = new InitialContext();
       datasource = (DataSource)
                    ctx.lookup("jdbc/disteduDS");
     } catch (Exception e) {
            System.out.println("DataSource hiba!!!");
            e.printStackTrace();
     }
```

```
   }

   public List<Course> getAllCourses(){
     String query ="select * from course";
     List<Course> list = new ArrayList<Course>();
     Connection con = null;
     Statement stmt = null;
     ResultSet rs = null;
     try{
       con = datasource.getConnection();
       stmt = con.createStatement();
       rs = stmt.executeQuery(query);
       while( rs.next()){
         int id = rs.getInt(1);
         String name = rs.getString(2);
         String description = rs.getString(3);
         double price = rs.getDouble(4);
         list.add(
          new Course( id, name, description, price));
       }
     }
     catch( SQLException se ){
         se.printStackTrace();
     }
     finally {
       if(rs != null) {
         try { rs.close(); }
         catch (Exception e) { e.printStackTrace(); }
       }
       if(stmt != null) {
         try { stmt.close(); }
         catch (Exception e) { e.printStackTrace(); }
       }
       if(con != null) {
         try { con.close(); } catch (Exception e) {
         catch (Exception e) { e.printStackTrace(); }
       }
     }
     return list;
}

public boolean insertCourse(Course c) {
```

```
    Connection con = null;
    Statement stmt = null;
    try{
        con = datasource.getConnection();
        stmt = con.createStatement();
        StringBuffer sql_comm = new StringBuffer(
           "insert into Course
           (name, description, price) values ('");
        sql_comm.append(c.getName()+"','");
        sql_comm.append(c.getDescription()+"',");
        sql_comm.append(c.getPrice()+")");
        System.out.println(sql_comm);
        stmt.executeUpdate(sql_comm.toString());
    }
    catch( SQLException se ){
         se.printStackTrace();
         return false;
    }
    finally {
      if(stmt != null) {
        try { stmt.close(); }
        catch (Exception e) { e.printStackTrace(); }
      }
      if(con != null) {
        try { con.close(); }
        catch (Exception e) { e.printStackTrace(); }
      }
   }
   return true;
}

public boolean deleteCourse(String[] ids) {
    if (ids.length == 0)
        return true;
    String sql = "delete from course where (id in ("
                 + ids[0];
    for (int i = 1; i < ids.length; ++i)
        sql += ", " + ids[i];
    sql += "))";
    System.out.println(sql);
    Connection con = null;
    Statement stmt = null;
```

```
    try{
        con = datasource.getConnection();
        stmt = con.createStatement();
        stmt.executeUpdate(sql);
    }
    catch (SQLException se) {
        se.printStackTrace();
        return false;
    }
    finally {
      if(stmt != null) {
        try { stmt.close(); }
        catch (Exception e) { e.printStackTrace(); }
      }
      if(con != null) {
        try { con.close(); }
        catch (Exception e) { e.printStackTrace(); }
      }
    }
    return true;
}
}
```

8. FEJEZET

A STRUTS KERETRENDSZER

8.1. A Struts MVC

Az Apache Struts keretrendszer szerzője Craig McClanahan, és az 1.0 verziója 2001. júliusban jelent meg. Ez egy mérföldkő volt a webfejlesztésben, mert biztosította a webkomponensek újrafelhasználhatóságát és a webalkalmazások karbantarthatóságát. A következő generáció a Struts2, amelynek célja a webfejlesztés megkönnyítése volt. Mivel is érkezett a 2.0-ás verzió? Kevesebb XML konfigurációra van szükség, mert használhatók az annotációk is helyette. Az akcióelemeket nem kell valamely rögzített osztályból származtatni, ezek egyszerű POJO-k (Plain Old Java Object) is lehetnek. Ez egyben csökkenti a függőséget is. Ezenfelül a kérésfeldolgozást is modulárisabbá tették, bevezették az interceptorok használatát. Továbbá a keretrendszer bizonyos osztályai is lecserélhetők saját osztályokkal.

Ebben a fejezetben a Struts 1.2.9 változatát mutatjuk be. Elsősorban azért választottuk ezt, mert ez együtt telepítődik a NetBeans 6.5 változattal, így használata nem igényel külön letöltést és telepítést. Másodsorban pedig használata egyszerűbb, ezért gyorsabban elsajátítható. Ez a keretrendszer egyszerűbb webalkalmazások ideális keretrendszere.

Milyen egy jó keretrendszer?

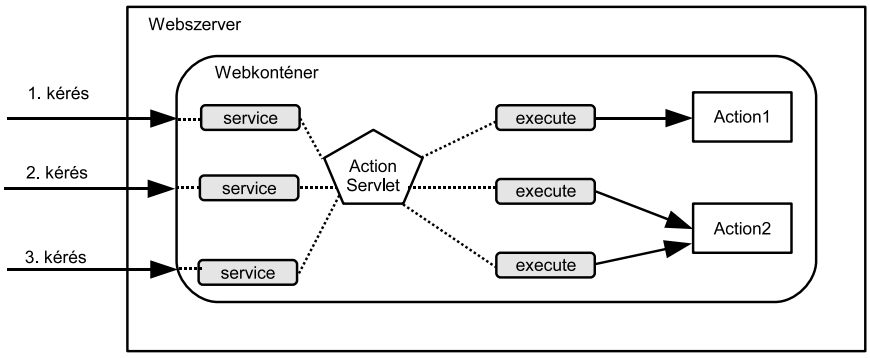
A keretrendszerek a szoftver újrafelhasználásának egyik formáját biztosítják. A Struts keretrendszer az Apache terméke és a Jakarta projekt keretében készült. A keretrendszer megvalósítja az MVC architektúrát, főképp a vezérlés részét támogatja úgy, hogy erre egy központi vezérlőszervletet biztosít. Olyan webalkalmazás esetében érdemes használni, amelyben az összes kérés esetében valamilyen elő-, illetve utófeldolgozást szükséges végezni. A Struts keretrendszer által biztosított központi vezérlőszervlet a Front Controller [2] tervezési minta megvalósítása. Először nézzük meg, hogy milyen problémákat old meg ez a tervezési minta:

– Elválasztja a vezérlési réteget a megjelenítési rétegtől. A vezérlő szervletnek viszont ismernie kell a megjelenítési komponensek URL-jeit, hiszen csak így valósíthatja meg a vezérlés átadását. A

keretrendszer ezt úgy oldja meg, hogy a megjelenítési komponensekhez szimbolikus neveket rendel, ezen szimbolikus neveket a Struts konfigurációs állományban képezhetjük le fizikai megjelenítési komponensekre.

– Bevezeti a vezérlő szervletet, amelyhez tetszőleges számú akcióelemet illeszthetünk. Ezek az akciók lesznek azok a vezérlési komponensek, amelyeket eddig szervletekkel oldottunk meg.

A Struts központi vezérlőszervletének (`ActionServlet`) működését szemlélteti a 8.1. ábra.



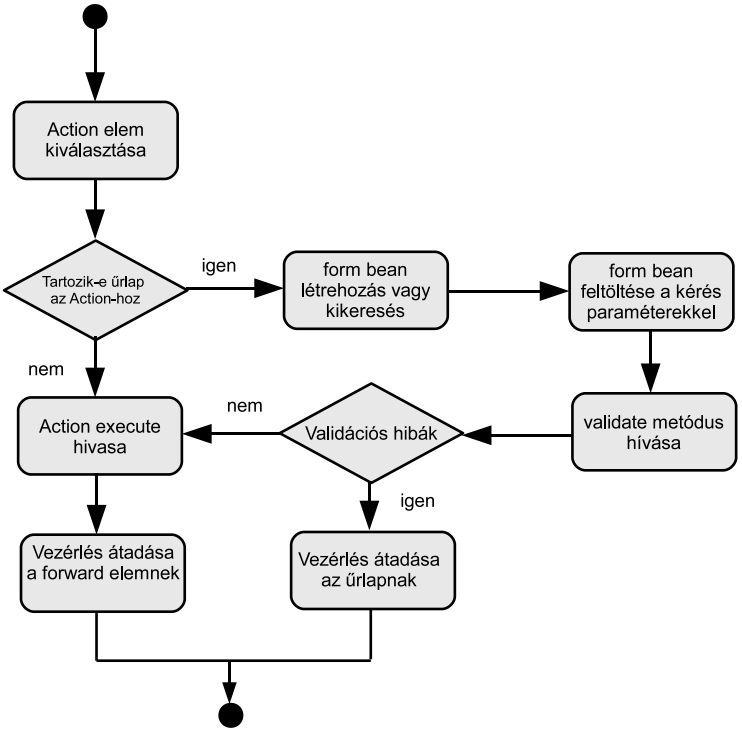
8.1. ábra. *Front Controller tervezési minta*

Egy keretrendszer olyan osztályok gyűjteménye, amelyek infrastruktúratámogatást biztosítanak az alkalmazásoknak. Erre az infrastruktúrára építhetjük rá az alkalmazásspecifikus osztályainkat. A Struts keretrendszer a következő elemeket tartalmazza:

– Központi vezérlő szervlet: ez egy `ActionServlet` típusú osztály, amelynek feladata a konfigurációs állomány beolvasása és ennek alapján a kérések leképzése a megfelelő akcióelemekre.
– Alapvető osztályok: a legfontosabb ezek közül az `Action` osztály, amelynek kiterjesztésével (származtatással) hozhatók létre az egyedi akcióknak megfelelő osztályok.
– Konfigurációs állományok: pl. a `struts-config.xml` állomány.

A 8.2. diagram szemlélteti a Struts keretrendszer működését. Amikor a kérés egy olyan webalkalmazáshoz érkezik, amely Struts keretrendszert használ, akkor a webkonténer a kérést továbbítja a központi vezérlőszervlethez:

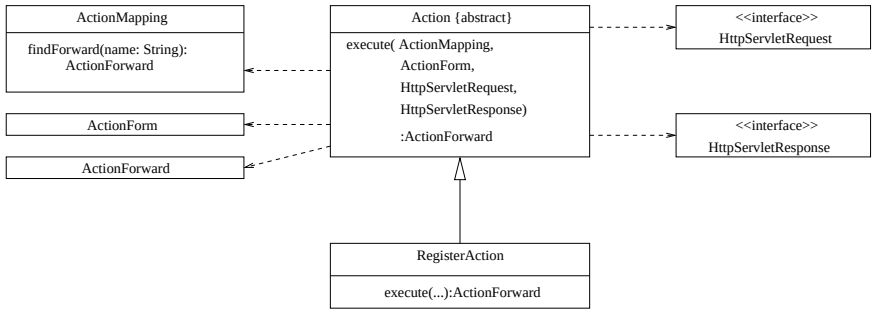
- A központi vezérlőszervlet a kérés URL alapján eldönti, hogy melyik akcióelemhez kell továbbítani a kérést.
- A következő lépés annak eldöntése, hogy tartozik-e űrlapbab (form-bean) az akcióelemhez. Ez nagyon egyszerű, hiszen minden akcióelem konfigurálva van, és a konfiguráció alapján ez egyértelműen eldönthető.
- Ha tartozik űrlapbab az akcióelemhez, akkor:
  - első használatkor létrehozza az űrlapbab objektumot, ha már létezik, akkor csak előkeresi,
  - az űrlapbabot feltölti a kérés paraméterekkel,
  - meghívja az űrlapbab `validate` metódusát,
  - validációs hibák esetén a vezérlés automatikusan visszakerül az űrlapra.
- Végrehajtódik az akcióelem `execute` metódusa, majd átadódik a vezérlés a megfelelő `forward` elemnek.



8.2. ábra. *Struts tevékenységi diagram*

8.2. Struts akcióosztály fejlesztése

A 8.3. ábra szemlélteti a Struts osztálykönyvtár fontosabb elemeit. Új `Action` elemet az `Action` absztrakt osztályból való örökítéssel lehet létrehozni, konkrét implementációt adva az `execute` metódusnak. Az `execute` metódus első argumentuma egy `ActionMapping` típusú objektum. A második egy `ActionForm`, amely az `Action` elemhez tartozó űrlapbabot reprezentálja. Ezt nem minden esetben szükséges használni. A harmadik és a negyedik argumentum a kérés-, illetve a válaszobjektum. Ezeket a webkonténer adja át a metódusnak. Az `execute` metódus egy `ActionForward` típust térít vissza, amely egy megjelenítési komponens szimbolikus neve és URL-je közötti leképzést reprezentál. A Struts konfigurációs fájlban minden egyes `Action` elemhez `forward` tagokat rendelhetünk. Ezek a tagok határozzák meg, hogy az adott akció után mely nézetek következhetnek. Futásidőben minden egyes `forward` taghoz létrejön egy `ActionForward` típusú objektum, amelyekre az `Action` osztályból a `findForward` metódussal kaphatunk referenciát.



8.3. ábra. *Struts Action API*

8.3. Struts akcióelemek konfigurálása

Egy Struts keretrendszert használó webalkalmazás konfigurálása a következő lépésekből áll:

1. A Struts infrastruktúra szervlet konfigurálása: `web.xml`
2. Az akcióelemek konfigurálása: `struts-config.xml`
3. A Struts osztálykönyvtárak telepítése: `/WEB-INF/lib`

A Struts infrastruktúra szervlet konfigurálása

A Struts infrastruktúra szervlet konfigurálása a `web.xml` állományban történik. Mint minden más szervlet esetében, itt is meg kell adni a szervlet nevét, amely legyen ebben az esetben `action`, és a szervlet osztályának nevét: `org.apache.struts.action.ActionServlet`. Egyetlen kezdőparamétere a `config`, amelynek értéke a Struts konfigurációs állomány neve: `/WEB-INF/struts-config.xml`. A `<load-on-startup>` paraméter azt jelenti, hogy ez a szervlet az alkalmazás indulásakor be fog töltődni. Ez lehetővé teszi, hogy rögtön az alkalmazás betöltődésekor beolvassa a konfigurációs állományt és beállítsa a kezdő állapotot.

```
<servlet>
   <servlet-name>action</servlet-name>
   <servlet-class>
      org.apache.struts.action.ActionServlet
   </servlet-class>
   <init-param>
      <param-name>config</param-name>
      <param-value>
         /WEB-INF/struts-config.xml
      </param-value>
   </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

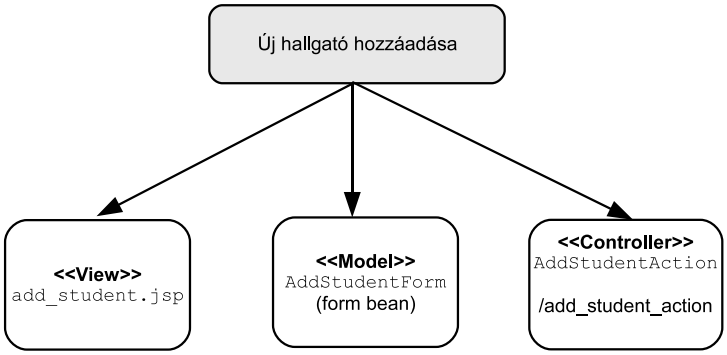
Szintén a `web.xml` állományban adjuk meg az URL mintát, amely a szervletre való leképzést szabályozza. A mi esetünkben minden egyes `*.do` alakú kérés az `action` nevű infrastruktúra szervletre fog leképződni:

```
<servlet-mapping>
   <servlet-name>action</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Az akcióelemek konfigurálása

Minden egyes akcióelemet konfigurálni kell a `struts-config.xml` állományban. A következő konfigurációs állományrészlet akciókat konfigurál. Az akcióelem feladata egy űrlap segítségével megadott hallgató

adatait beolvasni és rögzíteni adatbázisban. Az ehhez szükséges komponenseket a 8.4. ábra szemlélteti:



8.4. ábra. *Új hallgató felvitele*

```
<struts-config>
...
   <form-beans>
      <form-bean name="AddStudentForm"
                 type="view.AddStudentForm"/>
      ...
   </form-beans>
   ...
   <action-mappings>
        <action
             name="AddStudentForm"
             path="/add_student_action"
             type="controller.AddStudentAction"
             input="/add_student.jsp"
             scope="request">

            <forward name="success" path="/index.jsp"/>
            <forward name="error"
                     path="/error_add_student.jsp"/>
        </action>
        ...
   </action-mappings>
```

```
...
</struts-config>
```

Az `action` tag paramétereinek jelentése a következő:

- Kötelező paraméterek:
  - `path`: az akcióelemhez rendelt URL
  - `type`: az akció osztály neve, ezt kell példányosítni
- Opcionális paraméterek:
  - `name`: az űrlapbab elem, amely egységbe zárja az űrlap adatait és biztosíthatja ezek validációját
  - `input`: az űrlap, amelyről az adatok érkeznek az akcióelemhez
  - `scope`: az űrlapbab hatóköre (hol tárolódik ez a bab)
  - `validate`: logikai érték, végezzen-e validációt a bevitt adatokra

Az űrlapbabokat is konfigurálni kell a `struts-config.xml` állományban (NetBeans projektben ezt automatikusan megteszi az IDE). Íme egy példa:

```
<form-beans>
  <form-bean name="ActivateStudentForm"
             type="view.ActivateStudentForm"/>
  <form-bean name="SelectNameActionForm"
             type="view.SelectNameActionForm"/>
</form-beans>
```

A `forward` tagokkal határozzuk meg az egy adott akcióelem után következő lehetséges komponenseket. Az akció `execute` metódusában majd ezek közül fogjuk kiválasztani a következőt. Gyakorlatilag így oldjuk meg a navigációt a webalkalmazás komponensei között.

A Struts osztálykönyvtárak telepítése

Ha egy modern integrált fejlesztői környezettel dolgozunk, mint amilyen például a NetBeans, akkor az IDE új webalkalmazás létrehozásakor már felkínálja a létező keretrendszereket. Innen kiválasztva a Struts keretrendszert meg is történik az osztálykönyvtárak telepítése. Az osztálykönyvtárak a WEB-INF/lib mappába kerülnek. Ezek közül a következő három csomag a legfontosabb:

- `struts.jar`, ez a keretrendszer legfontosabb osztályait tartalmazza,

– `commons-beanutils.jar`, ez biztosítja a JavaBeans architektúrát és a tükrözést,
– `commons-digester.jar`, ez egy XML parsert biztosít.

8.4. A Struts `html` elemkönyvtár

A Struts nagyon sok JSP elemkönyvtárral rendelkezik. Mi csak a `html` elemkönyvtárat ismertetjük, mert ezt fogjuk használni a későbbiekben. Ha egy űrlapon hibás adatokat adunk meg, akkor az űrlap következő megjelenítésénél kényelmes, ha a bevitt adatok megjelennek a hibajelzésekkel együtt. Ennek legegyszerűbb módja, ha olyan űrlapelemeket használunk, amelyek ezt automatikusan elvégzik, amennyiben hozzáférnek az űrlapadatokat egységbezáró űrlapbabhoz.

A 8.1. táblázatban néhány Struts `html` elemet ismertetünk.

8.1. táblázat. *Néhány Struts `html` elem*

Elem	Magyarázat
`form`	HTML űrlap
`text`	szöveg típusú input elem
`radio`	rádiógomb típusú input elem
`submit`	submit típusú nyomógomb
`image`	kép típusú input elem
`img`	HTML img elem
`link`	HTML link
`errors`	hibaüzenetek megjelenítése

8.5. Megoldott feladatok

Ebben az alfejezetben átalakítjuk a távoktatás alkalmazásunkat Struts keretrendszert használó alkalmazássá.

8.1. feladat. Struts keretrendszer: Adja hozzá a projekthez a Struts keretrendszert. Ezt a projekt tulajdonságaiból kell kiválasztani.

`Properties -> Frameworks -> Add: Struts 1.2.9`

Próbálja ki az alkalmazást!

8.2. feladat. A DeleteCourseAction akcióelem: Le fogjuk cserélni az összes szervletet akcióelemre. Kezdjük az egyszerűbbekkel! Adjunk hozzá a webalkalmazáshoz egy új Struts akcióelemet:

1. lépés. `File -> New File-> Struts -> Struts Action`

- Class Name: `DeleteCourseAction`
- Package: `controller`
- Activation Path: `delete_course`

2. lépés. Másoljuk át a `DeleteCourse` szervletosztály `doPost` metódusának törzsét a `DeleteCourseAction execute` metódusának törzsébe.

3. lépés. Egészítsük ki a megfelelő import utasításokkal: `import model.*;` Ha helyesen dolgozott, a következő tartalmat kellett kapnia:

```
package controller;

import model.*;
import javax.servlet.http.*;;
import org.apache.struts.action.*;

public class DeleteCourseAction extends Action {

   private final static String SUCCESS="success";

   public ActionForward execute(
           ActionMapping mapping,
           ActionForm form,
           HttpServletRequest request,
           HttpServletResponse response)
           throws Exception {
     String[] courseId =
              request.getParameterValues("courses");
     if( courseId != null ){
            CourseDAO dao = new CourseDAO();
            dao.deleteCourse( courseId);
     }
     return mapping.findForward(SUCCESS);
```

```
   }
}
```

4. lépés. A `struts-config.xml` állományban található az akcióelem konfigurációja. Ha helyesen dolgozott, akkor az új komponensre ez a sor adódott hozzá a konfigurációs állományhoz:

```
<action path="/delete_course"
        type="controller.DeleteCourseAction"/>
```

Módosítani fogjuk az akcióelem konfigurációját, kiegészítve egy `forward` elemmel. A törlés esetében a vezérlést a főoldalnak fogjuk átadni, erre vezetjük be a `success` nevű `forward` elemet, amelyet beágyazunk az `action` elembe:

```
<action path="/delete_course"
        type="controller.DeleteCourseAction">
   <forward name="success" path="/index.jsp"/>
</action>
```

5. lépés. Ha hibakezelést is szeretnénk végezni és hiba esetén a vezérlést egy hibalapnak átadni, akkor be kell vezetnünk még egy `forward` elemet. Például:

```
<action path="/delete_course"
        type="controller.DeleteCourseAction">
   <forward name="success" path="/index.jsp"/>
   <forward name="error"
            path="/error_delete_course.jsp"/>
</action>
```

8.3. feladat. A SetPreferencesAction akcióelem: Készítsük el a fenti mintára a `SetPreferences` szervletet helyettesítő akcióelemet. Legyen ennek neve: `SetPreferencesAction.java`. Az akcióelem konfigurálása is a fentihez hasonló lesz.

8.4. feladat. Az AddCourseActionForm űrlapbab: Az űrlapbab egy sajátos szerveroldali Java bab komponens, amelynek feladata egységbe zárni egy űrlap adatait. Szintén erre a komponensre bízhatjuk az adatok konverzióját és validációját is. Konverzióra azért van szükség, mert

az űrlapparaméterek alapértelmezetten szöveges típusúak. A validációt az `action` elemben a `validate` attribútummal szabályozhatjuk. Az űrlapbab attribútumok megnevezésekor vigyáznunk kell arra, hogy ezek megegyezzenek a HTML űrlap elemeivel. Az űrlapbab elkészítése a következőképpen történik:

1. lépés. `File->New File->Struts->Struts ActionForm Bean`
- Class name: `AddCourseActionForm`
- Package: `view`

2. lépés. Az így kapott komponenst módosítsuk úgy, hogy az új tanfolyaműrlap adatainak megfelelő legyen. Ez így fog kinézni:

```
package view;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

public class AddCourseActionForm extends ActionForm {
    private String name;
    private String description;
    //a price mező az űrlapból
    private String priceStr;
    //a priceStr-nek megfelelő numerikus érték
    private double price;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getPriceStr() {
```

```
        return priceStr;
    }

    public void setPriceStr(String priceStr) {
        this.priceStr = priceStr;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }


    public ActionErrors validate(ActionMapping mapping,
                                 HttpServletRequest request) {
    ...
    }
}
```

8.5. feladat. Az add_course.jsp módosítása: Ahhoz, hogy a fenti bab helyes legyen, módosítanunk kell az `add_course.jsp` lapon az árnak megfelelő paraméter nevét `priceStr`-re. Nagy méretű űrlapoknál kimondottan előnyös, ha hiba esetén nem kell újra benépesítenünk az űrlapot, hanem ez automatikusan történik az űrlapbabból. Ez viszont úgy végezhető, ha lecseréljük a HTML `input` elemeket a Struts `html` elemkönyvtárban megadottakra. Az űrlap validációja során talált hibákat a lap tetején jelenítjük meg a `<html:errors/>` elem segítségével. A módosított JSP lap:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC
   "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@taglib prefix="html"
   uri="http://jakarta.apache.org/struts/tags-html" %>

<html:errors/>
<h1>New Course</h1>
<br/>
```

```
<html:form action="add_course.do"
           method="POST" focus="name">
  Name: <html:text property="name"/>
  <br><br>
  Description: <html:text property="description"
                          size="75"/>
  <br><br>
  Price: <html:text property="priceStr"/>
  <br><br>
  <html:submit value="Add Course"/>
</html:form>
```

Átírtuk a price mezőnevet priceStr nevűvé, ennek következtében helyesen fogja átvenni az űrlapbab az űrlapparamétereket. A price attribútumot a validate metódusban fogjuk beállítani.

8.6. feladat. Hibaüzenetek: A Struts keretrendszer nemcsak azt oldja meg, hogy validációs hiba esetén visszaadja a vezérlést az űrlapnak, hanem segédosztályokat is biztosít a hibák egységbe zárásához. Egy ilyen hibákat tároló gyűjtemény az ActionErrors osztály, amelybe ActionMessage típusú objektumokat helyezhetünk el. Az ActionErrors osztály (név, hibaüzenet) elempárok tárolását biztosítja, ahol a név String, illetve a hibaüzenet ActionMessage típusú.

Űrlapadatok ellenőrzése

```
public ActionErrors validate(ActionMapping mapping,
       HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();
  if (getName() == null ||
      getName().length() < 1) {
    errors.add("name",
    new ActionMessage("error.name.required"));
  }
  if (getDescription()  == null ||
      getDescription().length() < 1) {
    errors.add("description",
    new ActionMessage("error.description.required"));
  }
  if (getPriceStr()  == null ||
      getPriceStr().length() < 1) {
    errors.add("price",
```

```
    new ActionMessage("error.price.required"));

  }
  else{
    try{
      price = Double.parseDouble( getPriceStr());
      if( price < 0 )
        errors.add("price",
        new ActionMessage("error.price.negative"));
    }
    catch( NumberFormatException e ){
      errors.add("price",
      new ActionMessage("error.price.nonnumeric"));
    }
  }
  return errors;
}
```

Láthatjuk, hogy az `ActionMessage` példánynak nem egy igazi hibaüzenetet adtunk át, hanem egy azonosítót, amelynek alapján a hibaüzenet előkereshető egy erőforrásfájlból. Az erőforrásfájl egy tulajdonságfájl, amely alapértelmezetten az `ApplicationResources.property`, és amely a `com.myapp.struts` csomagban található. Ez minden olyan projektben megtalálható, amely használja a Struts keretrendszert. Ebbe a fájlba kell bevinni a `error.name.required` kulcsot a megfelelő hibaüzenettel. A hibaüzeneteket nem drótozzuk be az alkalmazásba, hanem egy külön erőforrásfájlban helyezzük el, amely lehetővé teszi, hogy könnyedén lefordítsuk az alkalmazásunkat egy másik nyelvre. Például a magyar fordításokat elhelyezhetjük egy `ApplicationResources_hu.property` fájlba, majd a `struts-config.xml` állományban beállítjuk ennek használatát. Alább egy példa látható egy ilyen erőforrásfájlra.

```
#Add Course
error.name.required=Az előadás nevét kötelező megadni!
error.description.required=Az előadás leírását
                           kötelező megadni!
error.price.required=Az előadás árát kötelező megadni!
error.price.nonnumeric=Az ár számérték kell legyen!
error.price.negative=Az ár nem lehet negatív érték!
```

8.7. feladat. Az AddCourseAction akcióelem: Most pedig készítsük el az új tanfolyam adatait feldolgozó akcióelemet:

```
File->New File->Struts->Struts Action
```

- Class name: `AddCourseAction`
- Package: `controller`
- Action Path: `/add_course`
- ActionForm Bean Name: `AddCourseActionForm`
- Input Resource: `/add_course.jsp`
- Scope: `Request`
- jelöljük be: `Validate ActionFormBean`

Az akcióelem `execute` metódusában ugyanazt fogjuk végezni, mint a hasonló célt szolgáló szervlet `doPost` metódusában. A különbség az űrlapparaméterek kinyerésének módja. Amíg a szervletben elvégeztük az űrlapparaméterek kinyerését, addig az akcióelemben ezt egységbe zárja az űrlapbab, amelyet az execute metódus paraméterként kap. Ez lesz az `execute` metódus `form` nevű és `ActionForm` típusú paramétere. Használat előtt ezt expliciten a megfelelő típusúvá kell konvertálni.

```
package controller;

import javax.servlet.http.*;
import org.apache.struts.action.*;

import view.*;
import model.*;

public class AddCourseAction extends Action {
    private final static String SUCCESS = "success";
    private final static String ERROR = "error";

    public ActionForward execute(
            ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws Exception {
        AddCourseActionForm myform =
                    (AddCourseActionForm)form;
```

```
        Course course = new Course(myform.getName(),
            myform.getDescription(),myform.getPrice());
        CourseDAO dao = new CourseDAO();
        if( dao.insertCourse(course) )
            return mapping.findForward(SUCCESS);
        else
            return mapping.findForward(ERROR);
    }
}
```

Az akcióelem konfigurációja alább látható. Két `forward` elemet használtunk: egyet ha sikeres az új tanfolyam felvitele, ebben az esetben a vezérlés a főoldalra kerül, és egyet abban az esetben, ha hiba történik. Erre bevezetünk egy hibalapot: `error_add_course.jsp`. Ez tetszőleges tartalmú lehet, a lényeg az, hogy a felhasználó értesüljön a hibáról.

```
<action-mappings>
  <action input="/add_course.jsp"
          name="AddCourseActionForm"
          path="/add_course"
          scope="request"
          type="controller.AddCourseAction"
          validate="true">
    <forward name="success" path="/index.jsp"/>
    <forward name="error" path="/error_add_course.jsp"/>
  </action>
  ...
</action-mappings>
```

8.8. feladat. Code refactoring: Töröljük ki az AddCourse szervletet a projektből (Code Refactoring). Ha helyesen dolgozott, a `web.xml` telepítésleíróban a szervlet konfigurációs résznek így kell kinéznie:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
```

```
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
    <param-name>detail</param-name>
    <param-value>2</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

A Struts konfigurációs állományban (`struts-config.xml`) az általunk létrehozott három akcióelem konfigurációjának pedig így kell kinéznie:

```
<form-beans>
   <form-bean name="AddCourseActionForm"
              type="view.AddCourseActionForm"/>
</form-beans>

<action-mappings>
   <action input="/add_course.jsp"
           name="AddCourseActionForm"
           path="/add_course"
           scope="request"
           type="controller.AddCourseAction">
      <forward name="success" path="/index.jsp"/>
      <forward name="error"
                   path="/error_add_course.jsp"/>
   </action>

   <action path="/set_preferences"
           type="controller.SetPreferencesAction">
```

```
      <forward name="success" path="/index.jsp"/>
   </action>

   <action path="/delete_course"
           type="controller.DeleteCourseAction">
      <forward name="success" path="/index.jsp"/>
   </action>
   ...
</action-mappings>
```

9. FEJEZET

WEBALKALMAZÁSOK BIZTONSÁGA

9.1. Biztonsági mechanizmusok

Az alkalmazások biztonságára több típusú mechanizmus létezik. Először ismertetjük ezeket a mechanizmusokat, majd bemutatjuk ezek megvalósításait mind a konténer szolgáltatásain keresztül, mind pedig programkód szintjén.

Négy fogalom értelmezésével kezdjük [1]:

1. Authentication – Hitelesítés, autentikáció
2. Authorization – Jogosultság-ellenőrzés, autorizáció
3. Data inetgrity – Adatintegritás
4. Confidentiality – Bizalmasság

A *hitelesítés* azt jelenti, hogy meggyőződünk arról, hogy az ügyfél tényleg az, akinek mondja magát. A hitelesítés során az ügyfélnek a *Ki vagy te?* kérdésre kell válaszolnia. Ez történhet különféle biometriás módszerrel, de a leggyakrabban alkalmazott módszer a felhasználónév és a jelszó bekérése.

A *jogosultság-ellenőrzés* lehetővé teszi, hogy különböző felhasználócsoportoknak különböző erőforrások elérését engedélyezzük. Tehát ez a biztonsági elem *Milyen erőforrásokat használhatsz?* kérdésre ad választ.

Az *adatintegritás* garantálja, hogy az adatok módosítás nélkül érkeznek meg a küldőtől a fogadóhoz, tehát sem véletlenül, sem pedig szándékosan nem voltak módosítva.

A *bizalmasság* szigorúbb követelmény, mint az adatintegritás, hiszen megköveteli, hogy az adatok olvasása csak az erre jogosultaknak legyen lehetséges. Ez azt jelenti, hogy ha a szállítás folyamán valaki hozzáfér az adatokhoz, ne tudja azokat értelmezni. Ez pedig csak úgy lehetséges, ha az adatokat titkosítjuk.

9.2. Konténer által nyújtott hitelesítés

A Java technológiával készült webalkalmazások biztonságát kétféleképpen szabályozhatjuk:

1. Deklaratív módszer: a biztonsági beállításokat az alkalmazás telepítésleírójában adjuk meg, és a biztonságot a webkonténer szabályozza.
2. Programozott módszer: a biztonsági beállításokat az alkalmazás komponenseiben, programkód szintjén valósítjuk meg.

Mindenképpen figyelemre méltó az első módszer, amelynek egyik nagy előnye, hogy a komponensek megírásakor nem kell törődnünk a biztonsági résszel, ezt majd a komponens illesztésénél fogjuk elvégezni. Így a komponensek általánosabbak lesznek, és a hibalehetőségek is csökkennek.

A Java EE specifikáció meghatározza, hogy az alkalmazásszervereknek támogatniuk kell a szerepkör alapú jogosultság-ellenőrzést [5]. Ez úgy történik, hogy a webalkalmazásban a komponensekhez absztrakt szerepeket rendelhetünk telepítésleírók vagy annotációk segítségével. Az absztrakt szerepkörök leképzése fizikai felhasználó csoportokra telepítési időben történik.

9.2.1. Alaphitelesítés és autorizáció

Ebben a részben a legegyszerűbb hitelesítési mechanizmusra adunk példát. A hitelesítésre a legyegyszerűbb módszer az, ha a webalkalmazás a böngészőre bízza a hitelesítési adatok bekérését. Ezt már a HTTP protokoll biztosítja. Ha a szerver a HTTP 401 státuszkódot küldi a böngészőnek, akkor a böngésző feladata megjeleníteni egy ablakot, amelybe a felhasználó beviheti a név–jelszó párost. Ezt a módszert BASIC, vagy alaphitelesítésnek nevezzük.

Most pedig vegyünk egy konkrét példát. Tételezzük fel, hogy van egy webalkalmazásunk, amely két komponenst tartalmaz, egy `index.jsp` és egy `admin.jsp` komponenst. Az `index.jsp` komponens nyilvános komponens lesz, azaz bárki hozzáférhet a komponenshez. Az `admin.jsp` komponenshez a hozzáférést csak adminisztrátori jogosultságú felhasználóknak szeretnénk lehetővé tenni. A legfontosabb pedig az, hogy nem akarjuk ezt az egészet leprogramozni, hanem a telepítésleíróban szeretnénk konfigurálni.

A példát konkrétan a Glassfish alkalmazásszerverre adjuk meg. A következő lépésekre van szükségünk:

1. Az alkalmazásszerverben (Glassfish) létre kell hoznunk az adminisztrátor csoportot a megfelelő felhasználókkal.
2. A webalkalmazás konfigurációs részében meg kell adnunk a következőket:
   a) web.xml:
      - Milyen hitelesítési mechanizmust óhajtunk használni?
      - Milyen absztrakt szerepköröket használ a webalkalmazás?
      - Milyen erőforrások elérését óhajtjuk korlátozni?
   b) sun-web.xml:
      - A webalkalmazás absztrakt szerepköreinek leképzése az alkalmazásszerverben létrehozott konkrét csoportokra.

1. Alkalmazászerver konfiguráció

Minden alkalmazásszerver biztosítja valamilyen formában csoportok és felhasználók létrehozását. A Sun alkalmazásszervere ezeket egy `realm` nevű elemben tárolja, amely az alkalmazásszerver adminisztrációs felületén keresztül szerkeszthető (realm=tartomány). Az alkalmazásszerver előre elkészített `realm` elemeket tartalmaz, mi például a `file` nevű `realm` elemhez fogjuk hozzáadni a csoportot és a felhasználókat. Az alkalmazásszerver adminisztrációs felületén válasszuk ki a `Configuration` menüpont `Security` almenüpontját, majd innen a `Realms`, illetve a `file` menüpontot. Ezután a `Manage Users` nyomógomb segítségével vihetünk fel felhasználókat és csoportokat. Tételezzük fel, hogy felvisszük az `adminisztrator` nevű felhasznalót és az `adminisztratorok` csoportot. A felhasználóhoz, jelen esetben az `adminisztrator`-hoz kötelező jelszót is megadni, amit az alkalmazásszerver hasító függvény segítségével tárol egy konfigurációs állományban.

```
User ID: adminisztrator
Group List: adminisztratorok
```

2. Alkalmazás konfiguráció: `web.xml` telepítésleíró

Ha NetBeans IDE-t használunk a fejlesztéshez, akkor lehetőségünk van a `web.xml` tartalmát grafikusan is szerkeszteni.

– Első lépésben a hitelesítési mechanizmust fogjuk beállítani:

```
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>file</realm-name>
</login-config>
```

A <login-config> tag szolgál a hitelesítés beállítására. Az <auth-method> meghatározza, hogy éppen melyik hitelesítési mechanizmust óhajtjuk használni. Lehetséges értékek: BASIC, DIGEST, FORM, CLIENT-CERT. Itt kötelező a <realm-name> tag megadása, amelyben meghatározzuk, hogy az alkalmazásszerverben hol vannak tárolva a felhasználók és csoportok.

– Második lépésben határozzuk meg az absztrakt szerepköröket, amelyeket a hozzáférési szabályok megadásánál fogunk használni. Vezessünk be egy admin nevű szerepkört.

```
<security-role>
   <description/>
   <role-name>admin</role-name>
</security-role>
```

– Harmadik lépésben vezessünk be korlátozást a védett erőforrásunkra:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>
         adminpages
      </web-resource-name>
      <url-pattern>/admin.jsp</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
```

A hozzáférési jogosultságokat a <security-constraint> taggal vezetjük be, amelyben jelen esetben megadjuk a korlátozott

erőforrást <web-resource-collection>, illetve azt, hogy milyen absztrakt szerepkörhöz tartozó felhasználók használhatják. Az erőforrás megadásánál a <web-resource-name> nevet rendel az erőforrás csoporthoz, majd az <url-pattern> taggal megadhatjuk az URL mintát. Az URL mintánál használhatjuk a * karaktert is. Például *.jsp az összes jsp fájlhoz való hozzáférést korlátozza. A <http-method> taggal határozzuk meg, hogy mely HTTP metódusokra vonatkoznak a korlátozások. És végül az <auth-constraint> tag segítségével hivatkozunk az absztrakt szerepkörre, amelyhez tartozó felhasználók jogosultak az erőforrás elérésére.

– Negyedik lépésben az absztrakt szerepkörök leképzését adjuk meg konkrét felhasználócsoportokra.
Az absztrakt szerepkörök leképzése konkrét csoportokra szintén konténerspecifikus művelet. A Sun alkalmazásszervere (Glassfish) erre a sun-web.xml állományt használja. A mi esetünkben ebben kell a következő leképzést elhelyezni:

```
<security-role-mapping>
   <role-name>admin</role-name>
   <group-name>adminisztratorok</group-name>
</security-role-mapping>
```

9.2.2. Hitelesítési módszerek

A Java EE háromféle hitelesítési típus meglétét követeli meg a webrétegtől.

– HTTP alaphitelesítés – BASIC
– Űrlap alapú – FORM
– HTTPS ügyfélhitelesítés – CLIENT-CERT

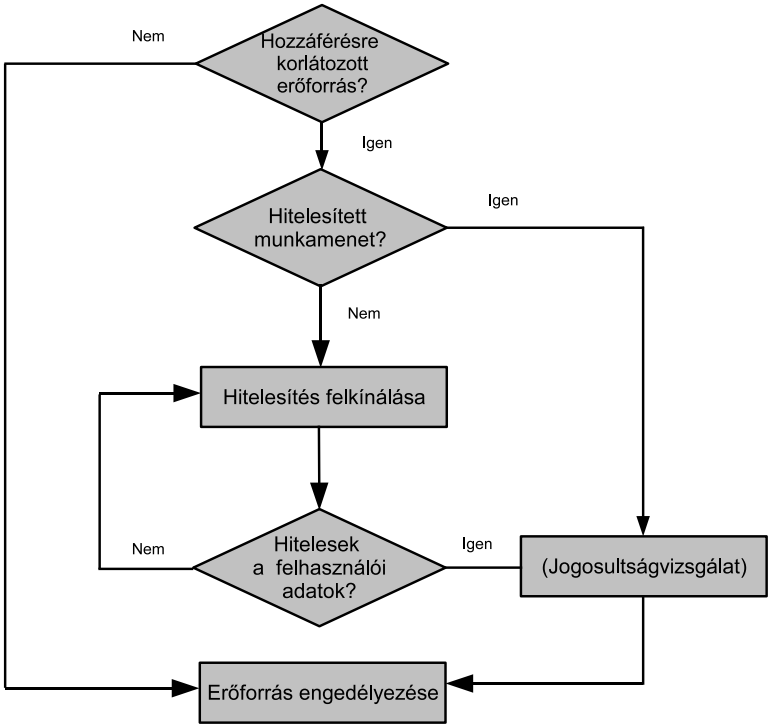
Létezik még egy negyedik, HTTP DIGEST hitelesítés is, de ez nem követelmény egy webkonténertől. A webalkalmazásban alkalmazott hitelesítési módszert a <login-config> tag segítségével adjuk meg, és az <auth-method> négyféle lehet:

– BASIC
– FORM
– DIGEST
– CLIENT-CERT

```
<login-config>
  <auth-method>
```

```
    BASIC|FORM|DIGEST|CLIENT-CERT
  </auth-method>
  <realm-name>file</realm-name>
</login-config>
```

A konténer által végzett hitelesítés menetét a 9.1. ábra szemlélteti. Megjegyezzük, hogy a konténerek lusta hitelesítést alkalmaznak, ez azt jelenti, hogy csak akkor végzik el a hitelesítést, ha a felhasználó egy korlátozott hozzáférésű erőforrást kér.



9.1. ábra. *Hitelesítés a webrétegben [3]*

BASIC hitelesítés

Az alaphitelesítést már ismertettük a 9.2.1. alfejezetben, ahol egy konkrét példát is adtunk. A módszer hátrányai a következők:

- Az alaphitelesítés a böngészőre bízza az ügyféladatok bekérését, így a bekérés módja böngészőspecifikus, és általában nem túl barátságos módon történik.
- Az igazi nagy probléma viszont az, hogy nehézkes a kijelentkezés. A böngésző bekérte és tárolta az ügyfél hitelesítéséhez szükséges információt, és lehetősége van ezt visszaküldeni a szervernek, amikor erre igényt tart a szerver. Léteznek ugyan megoldások arra, hogy a szerver kényszerítse a böngészőt, hogy újra bekérje a hitelesítési adatokat, de ezek csak részben hordozható megoldások.
- És végül meg kell említeni a biztonsági problémákat is, ugyanis ez a hitelesítési módszer az adatokat Base64 kódolással küldi a szervernek, amely nagyon könnyen dekódolható. Ez a hitelesítési módszer nyújtja a legalacsonyabb szintű biztonságot. Hasonló alacsony biztonsági szintet még az űrlap alapú hitelesítés nyújt.

Az alaphitelesítés esetében a `<realm-name>` megadása kötelező.

FORM hitelesítés

Az űrlap alapú hitelesítés biztonsági szintje megegyezik az alaphitelesítésével, ez a módszer viszont azzal az előnnyel rendelkezik az alaphitelesítéssel szemben, hogy lehetővé teszi a személyre szabott bejelentkezési űrlap megadását. A hitelesítési mód megadása az alaphitelesítéséhez hasonló:

Az alkalmazásszerverben szintén a `file` nevű `realm`-re fogunk hivatkozni, ezt az előző példában szemléltettük.

Tételezzük fel, hogy a webalkalmazásunk két JSP lapból áll: `index.jsp` és `seged.jsp`. Az `index.jsp` tartalmaz egy linket a `seged.jsp` JSP lapra, és tételezzük fel, hogy a `seged.jsp` lesz a védett erőforrás.

Az űrlap alapú hitelesítésnél szükség van még két webkomponens megadására, egy bejelentkezési űrlapra és egy sikertelen bejelenkezés esetén megjelenítendő hibalapra. Mindkettő lehet egyszerű HTML lap. Íme két egyszerű példa:

formlogin.html

```
<body>
   <form method="POST" action="j_security_check">
      Username: <input type="text" name="j_username"/>
      Password: <input type="password" name="j_password"/>
      <input type="submit" name="submit" value="Log in"/>
   </form>
</body>
```

Megjegyezzük, hogy a felhasználói név és a jelszó input mezőnevek rögzítettek: `j_username` és `j_password`. Hasonlóan az űrlaphoz rendelt akció neve is rögzített: `j_security_check`. Tulajdonképpen csak ennek a három mezőnévnek a meglétét követeli meg az űrlap alapú, konténer által nyújtott hitelesítés.

formerror.html

```
<body>
   <h1>Authentication failed</h1>
</body>
```

A webalkalmazás `web.xml` telepítésleírójában a `<login-config>` tag esetében a hitelesítés típusa és a használt realm mellett meg kell adni a hitelesítést végző űrlapot, illetve hibalapot is.

```
<login-config>
   <auth-method>FORM</auth-method>
      <realm-name>file</realm-name>
      <form-login-config>
         <form-login-page>/formlogin.html</form-login-page>
         <form-error-page>/formerror.html</form-error-page>
      </form-login-config>
</login-config>
```

A következő lépésben megadjuk az absztrakt szerepkört, ez ugyanúgy történik, mint az alaphitelesítésnél:

```
<security-role>
   <role-name>admin</role-name>
</security-role>
```

Most pedig beállítjuk a védett erőforrást és a szerepkört:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>folap</web-resource-name>
         <url-pattern>/seged.jsp</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
```

Végül az utolsó lépésben a `sun-web.xml` állományban elvégezzük az absztrakt szerepkör hozzárendelését egy konkrét felhasználói csoporthoz:

```
<security-role-mapping>
   <role-name>admin</role-name>
   <group-name>adminisztratorok</group-name>
</security-role-mapping>
```

CLIENT-CERT hitelesítés

A CLIENT-CERT vagy más néven HTTPS ügyfélhitelesítés az eddigi eljárások közül a legbiztonságosabb. Ez a módszer feltételezi, hogy a felhasználónak van egy nyilvános kulcsú bizonyítványa. Ez a hitelesítési mód HTTPS (=HTTP+SSL) protokollt használ. Az SSL (Secure Sockets Layer) technológia titkosítást, szerveroldali hitelesítést, adatintegritást és opcionális ügyfél oldali hitelesítést garantál TCP/IP kapcsolatoknak.

Ha azt szeretnénk, hogy a webréteg adatforgalma HTTPS protokollt használjon, ezt nagyon egyszerűen beállíthtjuk a `web.xml` telepítésleíróban. Itt gyakorlatilag a HTTPS protokoll használatát erőforráscsoportokra adhatjuk meg. Ha például azt szeretnénk, hogy minden szerepkörben levő felhasználó számára minden erőforrás HTTPS protokollt használjon, akkor a `<security-constraint>` elem `<web-resource-collection>` alelemében beállítjuk a kívánt URL mintát, kihagyjuk az `<auth-constraint>` elemet (azért, hogy bármilyen szerepkörben levő felhasználó hozzáférhessen) és beteszünk egy

user-data-constraint elemet. Ebben az utóbbiban kell megadni, hogy milyen szintű adatbiztonságot kell garantálnia a webkonténernek.

A user-data-constraint elemben a transport-guarantee alelem szabályozza a szállítási rétegre vonatkozó biztonságot. Ennek három lehetséges értéke van:

- NONE: nincs megkötés az adatforgalom biztonságára.
- INTEGRAL: a webkonténernek csak olyan kapcsolaton szabad kiszolgálni a kért erőforrásokat, amely biztosítja az adatok integritását. Gyakorlatilag ez HTTPS kapcsolatot jelent. Ebben az esetben megengedett a titkosítást nem végző változat is.
- CONFIDENTIAL: a webkonténernek ebben az esetben is kötelező a HTTPS protokollt használnia, itt viszont kötelező a titkosítás is a bizalmasság megvalósítása érdekében.

Íme egy konkrét példa:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>minden</web-resource-name>
      <url-pattern>*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
   </web-resource-collection>

   <user-data-constraint>
      <transport-guarantee>
         CONFIDENTIAL
      </transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

Megemlítjük, hogy ha a konténer által nyújtott hitelesítési mechanizmust választjuk, akkor figyelembe kell venni, hogy ez a mechanizmus nem alkalmazódik az include, illetve forward segítségével elért webkomponensekre. Tehát ha egy nem védett erőforrásból include vagy forward segítségével érjük el a védett erőforrást, akkor ezt a webkonténer nem ellenőrzi.

9.2.3. A biztonsági API

A biztonsági API csak olyan alkalmazásokban használható, amely a konténerre bízza a hitelesítést [3]. Az API információt szolgáltat a következőkről:

- a bejelentkezett felhasználó azonosságáról: `getUserPrincipal`,
- a bejelentkezett felhasználó szerepköréről: `isUserInRole`.

A webkonténer a biztonsági információkat a kérésobjektumon keresztül szolgáltatja, így a fenti két metódus a `HttpServletRequest` osztályhoz tartozik. A bejelentkezett felhasználó nevére, illetve szerepkörére akkor is szükségünk lehet, ha konténer által vezérelt hitelesítést használ az alkalmazásunk. A következő két kódrészlet rendre ezekre ad egy-egy példát.

A felhasználónév lekérdezése

```
String user = request.getUserPrincipal().getName();
String message = "Welcome, "+user;
```

A `getUserPrincipal` metódus egy `Pricipal` típusú objektumot térít vissza, amely egységbe zárja a felhasználó azonosságához tartozó adatokat.

A felhasználói szerepkör lekérdezése

```
if (request.isUserInRole("administrator")){
//adminisztrator menu megjelenitese
...
}
```

A fenti kódrészletben használt `administrator` egy absztrakt szerepkör, amelynek leképzését konkrét felhasználói csoportokra, Glassfish alkalmazásszerver esetén, a `sun-web.xml` állományban végezhetjük. Ezt már szemléltettük a BASIC hitelesítés alfejezetben.

9.3. Alkalmazásvezérelt hitelesítés

Bizonyos alkalmazások számára nem megfelelő, hogy a felhasználókat és a szerepköröket a rendszergazda rögzítse és ossza ki. Az ilyen

típusú alkalmazásokban azt szeretnénk, hogy az alkalmazás dinamikusan változtathassa a felhasználóit, ezt pedig csak programozottan lehet megvalósítani. Ennél a fajta hitelesítésnél a következőket kell programozottan megvalósítanunk:

- a felhasználó regisztrálása,
- a felhasználó hitelesítése és a hitelesítés adatainak tárolása meneti hatókörben,
- a meneti adatok érvényesítése azon webkomponensekben, amelyek korlátozott hozzáférést igényelnek.

9.3.1. A felhasználó regisztrálása

A felhasználó regisztrálása egy munkás részfeladat lesz. A lépéseket az alábbi felsorolás tartalmazza:

- adattábla létrehozása,
- modellelemek elkészítése,
- megjelenítési és vezérlési elemek elkészítése,
- hibaüzenetek bevitele a Struts erőforrásfájlba,
- akcióelem konfigurálása,
- az `index.jsp` aktualizálása.

A regisztráláshoz elő kell készítenünk az adatbázist a regisztrációs adatok tárolására. Ezért készítsünk egy `users` nevű táblát a következőképpen:

```
CREATE TABLE users (
   id varchar(25) NOT NULL UNIQUE,
   password varchar(50) NOT NULL,
   name varchar(60) NOT NULL,
   address varchar(120) NOT NULL,
   email varchar(40) NOT NULL
)
```

Az új táblához létrehozunk két modellelemet, egyet a felhasználó adatainak egységbe zárására (`User.java`), egyet pedig az adattáblával való kapcsolattartásra (`UserDAO.java`).

```
public class User {
    private String id; //username
    private String name;
    private String address;
    private String email;
    private String password;
```

```
public User(){}

public User( String id, String name, String address,
             String email, String password){
    this.id = id;
    this.name = name;
    this.address = address;
    this.email = email;
    this.password = password;
}

public void setId( String id ){
    this.id = id;
}

public String getId(){
    return id;
}

public void setName( String name ){
    this.name = name;
}

public String getName(){
    return name;
}

public void setAddress( String address ){
    this.address = address;
}

public String getAddress(){
    return address;
}

public void setEmail( String email ){
    this.email = email;
}

public String getEmail(){
    return email;
```

```
    }

    public void setPassword( String password ){
        this.password = password;
    }

    public String getPassword(){
        return password;
    }
}
```

Az adattáblával két fontos műveletet végezhetünk: új felhasználó adatainak beszúrását, illetve felhasználó hitelesítését a név és a jelszó alapján. Az adatbázis-kapcsolathoz most már az adatforrás (DataSource) komponenst használtuk, amelyet a 7.3. alfejezetben mutattunk be. Így egyszerűbb és áttekinthetőbb a DAO típusú osztály, ugyanakkor az alkalmazás is hordozhatóbb.

```
package model;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class UserDAO {

    private DataSource datasource;

    public UserDAO(){
        try{
            Context ctx = new InitialContext();
            if( ctx == null ){
                throw new RuntimeException(
             "JNDI context cannot be found");
            }
            datasource = (DataSource)
               ctx.lookup("jdbc/disteduDS");
        }catch(Exception e){
            e.printStackTrace();
        }
    }

// 0 - Sikeres beszuras
```

```
// 1 - Adatbazis-kapcsolat/SQL  hiba
// 2 - Letezo felhasznalo

public int insertUser(User u) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    String find_user = "select * from users where
                        id = '"+u.getId()+"'";
    try {
        con = datasource.getConnection();
        stmt = con.createStatement();
        rs = stmt.executeQuery(find_user);
        if (rs.next() == true) return 2;
        String sql_comm = "insert into Users
       (id, name, address, email, password) values
       ('" + u.getId() + "', '" + u.getName() + "', '" +
        u.getAddress() + "', '" + u.getEmail() +
        "', '" + u.getPassword() + "')";
        System.out.println(sql_comm);
        stmt.executeUpdate(sql_comm);
        return 0;
    }catch (SQLException e) {
      e.printStackTrace();
      return 1;
    }
    finally {
        if(rs != null) {
            try { rs.close(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        if(stmt != null) {
            try { stmt.close(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        if(con != null) {
            try { con.close(); }
            catch (Exception e) { e.printStackTrace(); }
        }
    }
}
```

```
// 0 - Sikeres bejelentkezes, helyes felhasznalonev
       es jelszo
// 1 - Adatbazis-kapcsolat/SQL  hiba
// 2 - Helytelen felhasznalonev vagy jelszo

public int loginUser(String username, String password) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    String sql_comm = "select * from users where
     id = '" + username + "') and
     (password = '" + password + "')";
    try {
        con = datasource.getConnection();
        stmt = con.createStatement();
        rs = stmt.executeQuery(sql_comm);
        if (rs.next() == false)
            return 2;
        else
            return 0;
    }catch (SQLException e) {
      e.printStackTrace();
      return 1;
    }
    finally {
        if(rs != null) {
            try { rs.close(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        if(stmt != null) {
            try { stmt.close(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        if(con != null) {
            try { con.close(); }
            catch (Exception e) { e.printStackTrace(); }
        }
    }
  }
}
```

Most pedig elkészítjük a bejelentkezési űrlapot (`register_user.jsp`) és az ezt feldolgozó Struts akcióelemet (`RegisterUserAction.java`), illetve az akcióelemnek megfelelő űrlapbabot (`RegisterUserActionForm.java`).

A további JSP lapokon egy nagyon egyszerű stíluslapot fogunk használni, ezért először ezt adjuk meg.

A `style.css` stíluslap

```
.error
{
   color: red;
}

form
{
   background-color: Cornsilk;
   margin: 0;
   padding: 1em;
   border: 1px solid;
}

#button
{
   padding-top: 1em;
   margin-left: auto;
   margin-right: auto;
   width: 40%;
   position: relative;
   left: 0.9em;
}
```

A `register_user.jsp` komponens

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="html"
   uri="http://jakarta.apache.org/struts/tags-html"%>
<%@taglib prefix="bean"
   uri="http://jakarta.apache.org/struts/tags-bean"%>
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
              charset=UTF-8">
        <title>Register User</title>
        <LINK REL=StyleSheet HREF="style.css"
                             TYPE="text/css" MEDIA=screen>
    </head>
    <body>
        <html:form action="register_user.do" method="post">
            <fieldset>
            <legend>Register User</legend>
            <table>
                <tr>
                    <td>Username:
                        <html:text property="id"
                                   size="25"/>
                    </td>
                    <td class="error">
                        <html:errors property="id" />
                    </td>
                </tr>
                <tr>
                    <td>Name    :
                        <html:text property="name"
                                   size="60"/>
                    </td>
                    <td class="error">
                        <html:errors property="name"/>
                    </td>
                </tr>
                <tr>
                    <td>Address :
                        <html:text property="address"
                                   size="120"/>
                    </td>
                    <td class="error">
                        <html:errors property="address"/>
                    </td>
                </tr>
                <tr>
                    <td>Email   :
```

```
                        <html:text property="email"
                                   size="40"/>
                    </td>
                    <td class="error">
                        <html:errors property="email"/>
                    </td>
                </tr>
                <tr>
                    <td>Password:
                       <html:password property="password"
                                      size="50"/>
                    </td>
                    <td class="error">
                        <html:errors property="password"/>
                    </td>
                </tr>
                <tr>
                    <td>Confirm password:
                    <html:password property="repassword"
                                      size="50"/>
                    </td>
                    <td class="error">
                        <html:errors
                              property="repassword" />
                    </td>
                </tr>
            </table>
            </fieldset>
             <div id="button">
               <html:submit value="Register"/>
            </div>
        </html:form>
    </body>
</html>
```

Az űrlapbabot mindig az akcióelem előtt hozzuk létre, ez lehetővé teszi, hogy az akcióelem létrehozásakor már kiválaszthassuk az űrlapbabot és csatolhassuk az akcióelemhez.

A `RegisterUserActionForm` űrlapbab

```
package view;
```

```
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

public class RegisterUserActionForm extends
         org.apache.struts.action.ActionForm{
   private String id;
   private String name;
   private String address;
   private String email;
   private String password;
   private String repassword;

   //ide jönnek a get és set metódusok!!!
   //ezt az olvasóra bízzuk

   public ActionErrors validate(ActionMapping mapping,
                            HttpServletRequest request){
       ActionErrors errors = new ActionErrors();
       if (getName() == null || getName().length() < 1){
           errors.add("name",
             new ActionMessage("error.name.required"));
       }
       if (getId() == null || getId().length() < 1){
           errors.add("id",
             new ActionMessage("error.id.required"));
       }
       if (getAddress()==null || getAddress().length()<1){
           errors.add("address",
             new ActionMessage("error.address.required"));
       }
       if (getEmail() == null || getEmail().length()<1){
           errors.add("email",
             new ActionMessage("error.email.required"));
       }
       if (getPassword()==null ||
           getPassword().length()<1){
           errors.add("password",
             new ActionMessage("error.password.required"));
       }
       if (getRepassword()==null ||
           getRepassword().length()<1){
```

```
            errors.add("repassword",
            new ActionMessage("error.repassword.required"));
        }
        if (getId() != null && getId().length() > 25) {
            errors.add("id",
              new ActionMessage("error.id.toolong"));
        }
        if (getName() != null && getName().length() > 60){
            errors.add("name",
              new ActionMessage("error.name.toolong"));
        }
        if (getAddress()!=null &&
            getAddress().length()>120){
            errors.add("address",
              new ActionMessage("error.address.toolong"));
        }
        if (getEmail() != null && getEmail().length() > 40){
            errors.add("email",
              new ActionMessage("error.email.toolong"));
        }
        if (getPassword()!=null &&
            getPassword().length()>50){
            errors.add("password",
               new ActionMessage("error.password.toolong"));
        }
        if (getRepassword()!=null &&
            getRepassword().length()>50){
            errors.add("repassword",
               new ActionMessage(
                   "error.repassword.toolong"));
        }
        if (getRepassword()!=null && getPassword()!=null &&
            getRepassword().equals(getPassword())==false){
            errors.add("repassword",
            new ActionMessage(
                   "error.repassword.doesnotmatch"));
        }
        return errors;
    }
}
```

A RegisterUserAction akcióelem

Gyakorlatilag az űrlapbab elvégezte az adatok validációját, így két hibalehetőséget kell az akcióelemben figyelembe venni: hiba az adatbázis kapcsolódásnál, illetve ha már létező felhasználót akarunk bevinni. Az előbbinél egy hibalapra adjuk a vezérlést, az utóbbinál pedig visszaadjuk a vezérlést a regisztrációs űrlapnak, jelezve a megfelelő hibát.

Sikeres regisztráció esetén a vezérlést a főoldalra adjuk vissza. Ezt a későbbiekben lehet majd módosítani és a regisztráció után a bejelentkezési űrlapra adni tovább a vezérlést.

```
package controller;

import model.*;
import view.*;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class RegisterUserAction extends Action{
    private final static String SUCCESS = "success";
    private final static String DB_ERROR = "db_error";
    private final static String INVALID_ID= "invalid_id";

    public ActionForward execute(ActionMapping mapping,
           ActionForm  form,
           HttpServletRequest request,
           HttpServletResponse response)
           throws Exception {
        RegisterUserActionForm bean=
                        (RegisterUserActionForm)form;
        User u = new User();
        u.setId(bean.getId());
        u.setName(bean.getName());
        u.setAddress(bean.getAddress());
        u.setEmail(bean.getEmail());
        u.setPassword(bean.getPassword());

        UserDAO dao = new UserDAO();
        int result = dao.insertUser(u);

        if (result == 1)
```

```
            return mapping.findForward(DB_ERROR);

        if (result == 2){
            ActionErrors errors = new ActionErrors();
            errors.add("id",
              new ActionMessage("error.id.invalid"));
            this.saveErrors(request, errors);
            return mapping.findForward(INVALID_ID);
        }
        return mapping.findForward(SUCCESS);
    }
}
```

Most pedig elvégezzük az akcióelem és az űrlapbab konfigurációját a `struts-config.xml` állományban.

```
<form-beans>
  <form-bean name="RegisterUserActionForm"
             type="view.RegisterUserActionForm"/>
  ...
</form-beans>
...
<action-mappings>
  ...
  <action input = "/register_user.jsp"
          name  = "RegisterUserActionForm"
          path  = "/register_user"
          scope = "request"
          type  = "controller.RegisterUserAction">
    <forward name="success"    path= "/index.jsp"/>
    <forward name="invalid_id" path= "/register_user.jsp"/>
    <forward name="db_error"   path= "/dberror.jsp"/>
  </action>
  ...
</action-mappings>
```

A kipróbálás előtt még egy utolsó apróságot kell elvégeznünk. A Struts erőforrásfájlba be kell írnunk az űrlapbab validációjakor használt hibaazonosítókat a megfelelő üzenetekkel együtt. Ilyenre már adtunk példát az előző fejezetben az űrlapbabot használó akcióelemnél, ezért most ennek elvégzését az olvasóra bízzuk.

9.3.2. A felhasználó hitelesítése

A felhasználó hitelesítése jóval egyszerűbb lesz, mint a regisztrálása, mert mind az adattábla, mind a DAO komponens készen áll ezen feladat megoldására, tehát a modell részhez nem kell semmit hozzátenni. Az alábbi részfeladatokat kell elvégeznünk:

- a bejelentkezési űrlap elkészítése: `login_user.jsp`
- Struts ActionForm elem elkészítése az űrlap adatainak egységbezárására: `LoginUserActionForm.java`
- Struts Action vezérlési elem elkészítése: `LoginUserAction.java`
- az akcióelem konfigurálása: `struts-config.xml`
- hibaüzenetek elhelyezése a megfelelő Struts erőforrásfájlba: `com.myapp.struts.ApplicationResource.properties`

A bejelentkezési űrlap: `login_user.jsp`

```
<%@page contentType="text/html"
        pageEncoding="UTF-8"%>
<%@taglib prefix="html"
      uri="http://jakarta.apache.org/struts/tags-html" %>
<%@taglib prefix="bean"
      uri="http://jakarta.apache.org/struts/tags-bean" %>

<!DOCTYPE HTML PUBLIC
   "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>Login User</title>
        <LINK REL=StyleSheet HREF="style.css"
                             TYPE="text/css" MEDIA=screen>

    </head>
    <body>
        <html:form action="login_user.do" method="post">
            <fieldset>
                <legend>Login User</legend>
            <table>
```

```
                <tr>
                    <td>Username: <html:text property="id"
                                    size="25"/></td>
                    <td class="error"><html:errors
                                property="id" /></td>
                </tr>
                <tr>
                    <td>Password:
                        <html:password property="password"
                                    size="50"/>
                    </td>
                    <td class="error">
                        <html:errors property="password"/>
                    </td>
                </tr>
            </table>
            </fieldset>
            <div id="button">
                <html:submit value="Login"/>
            </div>
        </html:form>
    </body>
</html>
```

Az űrlapban már megadtuk az űrlapot feldolgozó komponens nevét: `login_user.do`, ezért ezt kell majd hozzárendelnünk a megfelelő akcióelemhez. Most pedig megadjuk az űrlapbab kódját:

A `LoginUserActionForm` űrlapbab

```
package view;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

public class LoginUserActionForm extends
        org.apache.struts.action.ActionForm {
   private String id;
   private String password;

   public String getId() {
       return id;
```

```
    }
    public void setId(String string) {
        id = string;
    }
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public ActionErrors validate(ActionMapping mapping,
            HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if (getId() == null || getId().length() < 1) {
            errors.add("id",
              new ActionMessage("error.id.required"));
        }
        if (getPassword() == null ||
            getPassword().length()<1){
            errors.add("id",
            new ActionMessage("error.password.required"));
        }
        return errors;
    }
}
```

A `LoginUserAction` akcióelem

A felhasználónév és jelszó ellenőrzése a `UserDAO loginUser` metódusában történik, és ha bármelyik is hibás, ugyanazt a hibakódot térítjük vissza. Ennek következtében nem teszünk különbséget hibás felhasználónév és hibás jelszó között. Amennyiben a bejelentkezés sikeres volt, a menet hatókörébe regisztrálunk két attribútumot: `username, login`, amelyeket majd a bejelentkezett státusz ellenőrzésére fogunk használni:

```
HttpSession s = request.getSession();
s.setAttribute("username", username);
s.setAttribute("login", true);
```

```
package controller;

import org.apache.struts.action.*;
import view.*;
import model.*;
import javax.servlet.http.*;

public class LoginUserAction
        extends org.apache.struts.action.Action {

    private final static String SUCCESS = "success";
    private final static String DB_ERROR = "db_error";
    private final static String INVALID_ID_PW =
                                        "invalid_id_pw";

    @Override
    public ActionForward execute(ActionMapping mapping,
            ActionForm  form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws Exception {

        LoginUserActionForm bean =
                          ( LoginUserActionForm )form;
        String username = bean.getId();
        String password = bean.getPassword();

        UserDAO dao = new UserDAO();
        int result = dao.loginUser(username, password);

        if (result == 1)
            return mapping.findForward(DB_ERROR);

        if (result == 2){
            ActionErrors errors = new ActionErrors();
            errors.add("id",
            new ActionMessage(
                      "error.id_or_password.invalid"));
            this.saveErrors(request, errors);
            return mapping.findForward(INVALID_ID_PW);

        }
```

```
        HttpSession s = request.getSession();
        s.setAttribute("username", username);
        s.setAttribute("login", true);
        return mapping.findForward(SUCCESS);
    }
}
```

A fenti kódrészletekben megjelent hibaüzeneteket be kell tenni a `com.myapp.struts.ApplicationResource.properties` Struts erőforrásfájlba. Ezt most az olvasóra bízzuk.

Befejezésül konfiguráljuk az akcióelemet a `struts-config.xml` fájlban:

```
<form-beans>
   ...
   <form-bean name="LoginUserActionForm"
              type="view.LoginUserActionForm"/>
   ...
</form-beans>
...
<action-mappings>
   ...
   <action input="/login_user.jsp"
           name="LoginUserActionForm"
           path="/login_user"
           scope="request"
           type="controller.LoginUserAction">
       <forward name="success" path="/index.jsp"/>
       <forward name="invalid_id_pw" path="/login_user.jsp"/>
       <forward name="db_error" path="/dberror.jsp"/>
   </action>
   ...
<action-mappings>
```

Most pedig a bejelentkezési komponens készen áll a kipróbálásra, ehhez helyezzen el egy linket a `login_user.jsp`-re az `index.jsp` fájlban.

9.3.3. A meneti adatok érvényesítése

A távoktatás alkalmazásunkban csak azon felhasználóknak engedjük, hogy feliratkozzanak tanfolyamra, akik bejelentkeztek. Úgy kell

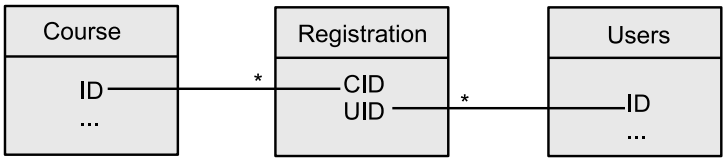
elkészítenünk a tanfolyamra való feliratkozást, hogy a feliratkozás előtt ellenőrizze a menet hatókörében a `login` nevű attribútum meglétét. Ha ez nem létezik, átadjuk a vezérlést a bejelentkezési oldalra.

A tanfolyamra való feliratkozás ismét egy munkás feladat lesz, hiszen nemcsak bizonyos adatok meglétét kell ellenőriznünk, hanem adatbázis-műveleteket is kell végeznünk.

A feliratkozáshoz a következő részfeladatokat kell megoldanunk:

- adattábla készítése a feliratkozáshoz: `REGISTRATION`
- DAO komponens az új adattáblához: `RegistrationDAO.java`
- a `CourseDAO` módosítása: `doSelectAvailableCourses( String user)` metódus hozzáadása, amely lekérdezi egy felhasználó számára elérhető kurzusok listáját
- a `register_course.jsp` elkészítése
- Struts ActionForm elem elkészítése az űrlap adatainak egységbezárására: `RegisterCourseActionForm.java`
- Struts Action vezérlési elem elkészítése: `RegisterCourseAction.java`
- az akcióelem konfigurálása: `struts-config.xml`
- hibaüzenetek elhelyezése a megfelelő Struts erőforrásfájlba: `com.myapp.struts.ApplicationResource.properties`

A feliratkozásokat adatbázisban fogjuk tárolni, ehhez viszont szükség van egy új adattáblára. Legyen ennek neve `Registration`. Ez a tábla egy asszociációs tábla, amely két azonosítót fog tartalmazni: egy UID (user ID) és egy CID (course ID) (l. 9.2. ábra).



9.2. ábra. *A `Registration` tábla*

Miután létrehoztuk a 9.2. ábrán látható táblát, készítsük el a neki megfelelő DAO komponenst, és legyen ennek neve `RegistrationDAO.java`. Nyilván ez egy modellelem lesz, ezért a `model` csomagban fogjuk elhelyezni. Egyelőre csak új rekord felvitelét fogja végezni, ezt később további műveletekkel ki lehet egészíteni.

A `RegistrationDAO` modell komponens

```
package model;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class RegistrationDAO {

    private DataSource datasource;

    public RegistrationDAO(){
        try{
            Context ctx = new InitialContext();
            if( ctx == null ){
                throw new RuntimeException(
                 "Nevszerver (JNDI) hiba");
            }
            datasource = (DataSource)
                        ctx.lookup("jdbc/disteduDS");
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }

   //0 - Sikeres vegrehajtas
   //1 - Adatbazis vagy SQL hiba

    public int newRegistrations(String uid,
                                String[] courses){
        Connection con = null;
        Statement stmt = null;
        try{
            con = datasource.getConnection();
        }
        catch( SQLException e){
            e.printStackTrace();
            return 1;
        }
        try{
```

```
            stmt = con.createStatement();
            for( int i=0; i<courses.length; ++i ){
                String sql_comm = "insert into
                    registration (uid, cid) values (" +
                  "'" + uid + "','" + courses[i] + "')";
                System.out.println(sql_comm);
                stmt.executeUpdate(sql_comm);

            }
            return 0;
        }catch (SQLException e) {
                e.printStackTrace();
                return 1;
        }
        finally {
          if(stmt != null) {
              try { stmt.close(); }
              catch (Exception e) { e.printStackTrace(); }
          }
          if(con != null) {
              try { con.close(); }
             catch (Exception e) { e.printStackTrace(); }
          }
        }
    }
}
```

Most pedig megadjuk a `CourseDAO` osztályhoz hozzáadandó metódust. Ez a metódus azon kurzusok listáját állítja össze, amelyekre a felhasználó még feliratkozhat (eddig még nem iratkozott fel).

```
public List<Course> doSelectAvailableCourses(
                                String userid) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs   = null;
    List<Course> cl = new ArrayList<Course>();
    try{
        con = datasource.getConnection();
    }
    catch( SQLException e){
        e.printStackTrace();
    }
```

```
    try{
       stmt = con.createStatement();
       String sql = "select * from Course where id not in "
                +"(select cid from registration where
                uid ='" + userid + "')";
       ResultSet rs = stmt.executeQuery(sql);
       while( rs.next() ){
          Course course = new Course();
          course.setId(rs.getInt("id"));
          course.setName(rs.getString("name"));
          course.setDescription(
                              rs.getString("description"));
          course.setPrice(
              Double.parseDouble(rs.getString("price")));
          cl.add(course);
       }
       return cl;
    }
    catch( SQLException e ){
       e.printStackTrace();
    }
    finally {
          if(stmt != null) {
              try { stmt.close(); }
              catch (Exception e) { e.printStackTrace(); }
          }
          if(con != null) {
              try { con.close(); }
              catch (Exception e) { e.printStackTrace(); }
          }
    }
}
```

A register_course.jsp megjelenítési komponens

Most pedig elkészítjük azt a JSP lapot (register_course.jsp), amely megjeleníti a fenti metódus által megadott kurzusok listáját a felhasználó számára egy űrlapon. Figyeljük meg, hogy a body elem a bejelentkezési státusz ellenőrzésével kezdődik. Amennyiben nincs bejelentkezve a felhasználó, átirányítjuk a bejelentkezési oldalra.

```
<%@page contentType="text/html"
```

```
         pageEncoding="UTF-8"%>
<%@page import = "java.util.*, model.*" %>
<%@taglib prefix="c"
       uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="html"
       uri="http://jakarta.apache.org/struts/tags-html" %>
<%@taglib prefix="bean"
       uri="http://jakarta.apache.org/struts/tags-bean" %>

<!DOCTYPE HTML PUBLIC
   "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
              charset=UTF-8">
        <title>JSP Page</title>
        <LINK REL=StyleSheet HREF="style.css"
                              TYPE="text/css" MEDIA=screen>

    </head>
    <body>
        <c:if test = "${login != true}">
            <c:redirect url="login_user.jsp"/>
        </c:if>

        <h1>Course Registration</h1>

        <%
            CourseDAO dao = new CourseDAO();
            HttpSession s = request.getSession();
            List<Course> cl = dao.doSelectAvailableCourses
                   (s.getAttribute("username").toString());
            pageContext.setAttribute("cl", cl);
            System.out.println("Courses:"+ cl.size());
        %>
        <table>
          <tr>
            <td class="error">
              <html:errors property="courses" />
            </td>
```

```
          </tr>
        </table>
        <html:form action="register_course.do"
                   method="get" >
            <fieldset>
                <legend>Register Course</legend>
                <table>
                <c:forEach var = "v" items ="${cl}" >
                    <tr>
                     <td>
                     <html:checkbox property="courses"
                                    value="${v.id}"/>
                     </td>
                     <td><c:out value="${v.name}"/></td>
                    </tr>
                </c:forEach>
                </table>
            </fieldset>
            <div id="button">
                  <html:submit value="Register"/>
            </div>
        </html:form>
    </body>
</html>
```

Következhet a Struts akcióelemhez szükséges űrlapbab, a RegisterCourseActionForm.java, amelyet a view csomagba helyezünk el.

A RegisterCourseActionForm űrlapbab

```
package view;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;

public class RegisterCourseActionForm
        extends org.apache.struts.action.ActionForm{

   private String[] courses;

   public String[] getCourses() {
```

```
        return courses;
    }
    public void setCourses(String[] strings) {
        courses = strings;
    }

    @Override
    public ActionErrors validate(ActionMapping mapping,
             HttpServletRequest request) {
        ActionErrors errors = new ActionErrors();
        if (getCourses()==null||getCourses().length==0){
            errors.add("courses",
              new ActionMessage("error.courses.required"));
        }
        return errors;
    }
}
```

A RegisterCourseAction akcióelem

```
package controller;

import javax.servlet.http.*;
import org.apache.struts.action.*;

import model.*;
import view.*;

public class RegisterCourseAction
        extends org.apache.struts.action.Action {

    private final static String SUCCESS = "success";
    private final static String DB_ERROR = "dberror";

    public ActionForward execute(ActionMapping mapping,
            ActionForm  form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws Exception {

        RegistrationDAO dao = new RegistrationDAO();
```

```
        HttpSession s = request.getSession();
        RegisterCourseActionForm bean =
                  (RegisterCourseActionForm)form;
        String[] courses = bean.getCourses();

        if( courses.length != 0 ){
            int result = dao.newRegistrations(
            (String) s.getAttribute("username"), courses);
            if( result == 1) {
                return mapping.findForward(DB_ERROR);
            }
        }
        return mapping.findForward(SUCCESS);
    }
}
```

Utolsó lépésként végezzük el az akcióelem konfigurálását. Figyeljünk az űrlapbab hatókörének megadására, amely ebben az esetben a kérés hatóköre lesz, hiszen egy felhasználó egy menet alatt többször is végezhet feliratkozást tanfolyamokra. Ezzel a beállítással az űrlapbabot minden egyes kérésre újra létrehozza a webkonténer.

```
<form-beans>
   ...
   <form-bean name="RegisterCourseActionForm"
              type="view.RegisterCourseActionForm"/>
   ...
</form-beans>

<action-mappings>
   ...
   <action input="/register_course.jsp"
        name="RegisterCourseActionForm"
        path="/register_course"
        scope="request"
        type="controller.RegisterCourseAction">
     <forward name="success" path="/index.jsp"/>
     <forward name="db_error" path="/dberror.jsp"/>
   </action>
   ...
</action-mappings>
```

9.3.4. Kijelentkezés

Mivel a bejelentkezési státuszt a menet hatókörében tároljuk, a `login` nevű attribútumban, ez automatikusan megszűnik a menet időtartamának lejártakor. Egy biztonságos alkalmazásnak viszont lehetővé kell tenni, hogy a felhasználó bármikor kijelentkezhessen. Ez egy nagyon egyszerű művelet lesz, amely két dolgot tartalmaz, a menet érvénytelenítését és a vezérlés átadását a bejelentkezési lapnak.

Oldjuk meg ezt a műveletet egy egyszerű akcióelemmel.

A `LogoutUserAction` akcióelem

```
package controller;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class LogoutUserAction extends Action {
    private final static String SUCCESS = "success";

    @Override
    public ActionForward execute(
            ActionMapping mapping, ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response)
            throws Exception {
        request.getSession().invalidate();
        return mapping.findForward(SUCCESS);
    }
}
```

Az akcióelem konfigurációja (`struts-config.xml`):

```
<action path="/logout_user"
        type="controller.LogoutUserAction">
   <forward name="success" path="/login_user.jsp"/>
</action>
```

9.4. Feladatok

9.4.1. Programozási feladatok

9.1. feladat. Deklaratív hitelesítés: Valósítsuk meg a távoktatás alkalmazásra, konténer által nyújtott, űrlap alapú hitelesítéssel az adminisztrátori feladatkört. Két védett erőforrásunk lesz: tanfolyamok törlése, illetve hozzáadása (`delete_course_jsp, add_course.jsp`).

9.2. feladat. CSS: Bővítsük ki a megadott CSS stíluslapot.

9.4.2. Tesztkérdések

9.1. kérdés: A <transport-guarantee> elem lehetséges értékei: NONE, INTEGRAL vagy CONFIDENTIAL.

A. igaz
B. hamis

9.2. kérdés: Válasszuk ki az igaz kijelentéseket! (1 helyes)

A. A hitelesítés annak eldöntését jelenti, hogy a felhasználó milyen erőforrásokhoz férhet hozzá.
B. A jogosultság-ellenőrzés azt jelenti, hogy meggyőződünk arról, hogy az ügyfél tényleg az, akinek mondja magát.
C. A hitelesítés és a jogosultság-ellenőrzés azonos jelentésű fogalmak.
D. A jogosultság-ellenőrzés annak eldöntését jelenti, hogy a felhasználó milyen erőforrásokhoz férhet hozzá.

9.3. kérdés: Válasszuk ki az igaz kijelentéseket! (3 helyes)

A. A bizalmasság azt jelenti, hogy az adatok értelmezése csak az arra jogosultaknak lehetséges.
B. A JSP lapok alapértelmezetten garantálják a bizalmasságot.
C. A bizalmasság megvalósítható az SSL protokoll használatával.
D. A bizalmasságot garantálja a HTTP protokoll.

E. Az adatintegritás azt jelenti, hogy az adatok szállítás közben nem sérülnek.

9.4. kérdés: Válasszuk ki az igaz kijelentéseket! (1 helyes)

A. Alapértelmezetten a felhasználókat és a szerepköröket a telepítésleíróban kell megadni.
B. A hitelesítési mechanizmust a telepítésleíró login-config elemében kell megadni.
C. A hitelesítési mechanizmust a telepítésleíró authentication elemében kell megadni.
D. A hitelesítési mechanizmust a telepítésleíró login elemében kell megadni.

9.5. kérdés:

Válasszuk ki az igaz kijelentéseket! (1 helyes)

A. A HttpServletRequest isUserInRole metódusa logikai értéket térít vissza.
B. A HttpServletRequest isMemberInRole metódusa logikai értéket térít vissza.
C. A HttpSession isUserInRole metódusa logikai értéket térít vissza.
D. A jogosultság-ellenőrzés csak a telepítésleíróban adható meg, programban nem.

9.6. kérdés: Adott a következő telepítésleíró részlet. Kinek van jogosultsága a /SecuredServlet erőforráshoz? (1 helyes)

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Sensitive</web-resource-name>
    <url-pattern>/SecuredServlet</url-pattern>
  </web-resource-collection>
  <auth-constraint>
        <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>
```

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Sensitive</web-resource-name>
    <url-pattern>/SecuredServlet</url-pattern>
  </web-resource-collection>
  <auth-constraint>
        <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Sensitive</web-resource-name>
    <url-pattern>/SecuredServlet</url-pattern>
  </web-resource-collection>
  <auth-constraint/>
</security-constraint>
```

A. Csak az admin szerepkörben levő felhasználóknak.
B. Csak a manager szerepkörben levő felhasználóknak.
C. Az admin és manager szerepkörű felhasználóknak.
D. Bármely szerepkörű felhasználónak, kivéve az admin és manager szerepkörűeknek.
E. Senkinek.

9.7. kérdés: Válasszuk ki az egyetlen igaz kijelentést!

A. A felhasználók azonosságát a telepítésleíró users elemében adhatjuk meg.
B. A user-data constraint elemben megadható, hogy mely felhasználók férnek hozzá az erőforrásokhoz.
C. Az auth-constraint elemben megadható, hogy mely szerepkörök férnek hozzá az erőforráshoz.
D. A telepítésleíróban csak szervlet típusú erőforráshoz való hozzáférés korlátozható, a JSP lapokhoz való nem.

9.8. kérdés: Válasszuk ki az igaz kijelentéseket! (3 helyes)

A. A FORM típusú hitelesítés nem biztosít titkosítást.
B. A CLIENT-CERT típusú hitelesítés titkosítást garantál.

C. A BASIC típusú hitelesítés lehetővé teszi a hitelesítési űrlap és hibalap megadását.
D. A hitelesítési mechanizmus a telepítésleíró login-config elemében adható meg.
E. A hitelesítési mechanizmus a telepítésleíró login elemében adható meg.

9.9. kérdés: Válasszuk ki az igaz kijelentéseket! (2 helyes)

A. A FORM típusú hitelesítésnél az űrlap neve LOGIN.
B. A FORM típusú hitelesítésnél az űrlap action elemének neve j_login_check.
C. A FORM típusú hitelesítésnél az űrlap action elemének neve j_security_check.
D. A FORM típusú hitelesítésnél az űrlapon kötelező módon legyen egy j_username és egy j_password nevű elem.
E. A FORM típusú hitelesítésnél az űrlapon kötelező módon legyen egy username és egy password nevű elem.

9.10. kérdés: Mely hitelesítést definiálja a HTTP protokoll? (1 helyes)

A. FORM
B. CLIENT-CERT
C. SECURE
D. DIGEST

A. FÜGGELÉK – A TELEPÍTÉSLEÍRÓ

A webalkalmazás telepítésleírója a `WEB-INF` mappában helyezkedik el, a neve `web.xml` és az alkalmazásra vonatkozó konfigurációkat tartalmazza. Amint látszik az állomány kiterjesztéséből is, ez egy XML formátumú állomány. A következőkben ezen konfigurációs állomány legfontosabb elemeit szemléltetjük és magyarázzuk:

<context-param>

Olyan inicializáló paraméterek adhatók meg ezen elem segítségével, amelyek az alkalmazás összes komponensében használhatók. Például:

```
<context-param>
   <param-name>fajlnev</param-name>
   <param-value>/WEB-INF/tanfolyamok.txt</param-value>
</context-param>
```

A webalkalmazás paramétereit egy szervlet a következőképpen dolgozhatja fel:

```
ServletContext ctx = this.getServletContext();
String fname = ctx.getInitParameter("fajlnev");
```

Ha ugyanezt egy JSP lapon akarjuk elérni, akkor ez így végezhető el:

```
<% String fname =
        application.getInitParameter("fajlnev);%>
```

<distributable>

Ez az elem csak törzs rész nélkül használható `<distributable/>` és közli a webkonténerrel (alkalmazásszerverrel), hogy az alkalmazás úgy van elkészítve, hogy elosztott webkonténerben is használható. Gyakorlatilag azt jelenti, hogy az alkalmazás nem egy JVM-ben fut, megtörténhet,

hogy a webkonténer egy menet adatait szerializálja és átviszi egy másik webkonténerbe. A nagy előnye az, hogy így lehetőség van terheléselosztásra.

<error-page>

A deklaratív hibakezelés lehetővé teszi, hogy a webalkalmazás készítője meghatározhassa az ügyfélnek visszaküldött tartalmat, hiba, illetve kivétel kiváltódása esetén. A deklaratív hibakezelést a `web.xml` állományban, az `error-page` elemmel végezzük. Ezen belül használható az `error-code` elem, illetve az `exception-type`. Az `error-code` elemmel a HTML hibakód adható meg, az `exception-type` elemmel pedig a kivétel típusa. Ez utóbbinál teljes típusnevet kell megadni, például `java.sql.SQLException`.

A `location` elemmel határozzuk meg, hogy hiba vagy kivétel esetén melyik lapnak adódjon át a vezérlés, vagyis mi jelenjen meg az ügyfélnek. A `location` elemben a hibalapot abszolút módon adjuk meg, ezért ennek kötelező a / karakterrel kezdődnie. A nagy előnye a deklaratív hibakezelésnek az, hogy a komponensek nem kell foglalkozzanak hibakezeléssel, ezért nem keveredik a hibakezelő kód az üzleti logikával vagy a megjelenítéssel. Ez a hibakezelési lehetőség is nagymértékben segíti a rugalmas alkalmazások készítését. Minden egyes hibakód, illetve kivételtípus csak egyszer szerepelhet a telepítésleíróban. Ha egy szervletben olyan hiba keletkezik, amely nincsen lekezelve a telepítésleíróban, akkor a webkonténer az 500-as hibakódot küldi vissza a böngészőnek.

```
<error-page>
   <exception-type>java.sql.SQLException</exception-type>
   <location>/sqlerrorpage.jsp</location>
</error-page>

<error-page>
   <error-code>404</error-code>
   <location>/errorpage.jsp</location>
</error-page>
```

<login-config>

A <login-config> elemmel a webalkalmazásban alkalmazott deklaratív hitelesítési mechanizmust konfigurálhatjuk. Ebben az esetben a hitelesítést a webkonténerre bízzuk, tehát a konkrét felhasználókat és csoportokat is a webkonténerben kell konfigurálni. A telepítésleíróban azt szabályozzuk, hogy melyik hitelesítési mechanizmust óhajtjuk használni (ezek részletes leírását a 9. fejezet ismerteti).

```
<login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>file</realm-name>
</login-config>
```

<security-constraint>

A <security-constraint> elemmel az erőforrásokhoz, illetve erőforrás-csoportokhoz való hozzáférést szabályozhatjuk. A három legfontosabb alelem:

- <web-resource-collection>: mit védünk?
- <auth-constraint>: mely absztrakt szerepköröknek engedjük meg a hozzáférést?
- <user-data-constraint>: milyen jellegű védelmet kell biztosítani a szállítandó adatoknak?

A részletes leírás szintén a 9. fejezetben található.

```
<security-constraint>

   <web-resource-collection>
      <web-resource-name>
         adminpages
      </web-resource-name>
      <url-pattern>/admin.jsp</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
   </web-resource-collection>

   <auth-constraint>
```

```
      <role-name>admin</role-name>
   </auth-constraint>

   <user-data-constraint>
      <transport-guarantee>
         CONFIDENTIAL
      </transport-guarantee>
   </user-data-constraint>

</security-constraint>
```

<security-role>

A <security-role> elemmel absztrakt szerepkörök határozhatók meg, amelyeket majd le kell képezni a webkonténerben meghatározott csoportokra. Az absztrakt szerepköröket felhasználjuk az erőforrás-hozzáférések szabályozásánál.

```
<security-role>
   <description/>
   <role-name>admin</role-name>
</security-role>
```

<servlet>

Egy webalkalmazás minden szervletét konfigurálni kell a telepítésleíróban, megadva egy nevet, illetve a hozzá tartozó bájtkód osztályt (.class fájl). Opcionálisan megadhatunk olyan paramétereket is, amelyek felhasználhatók például a szervlet inicializálására. Ellentétben az alkalmazás paramétereivel, ezek a paraméterek csak az adott szervletből érhetők el.

```
<servlet>
   <servlet-name>ListCourses</servlet-name>
   <servlet-class>view.ListCourses</servlet-class>
   <init-param>
      <param-name>jdbcURL</param-name>
```

```
      <param-value>
         jdbc:derby://myserver:1527//distedu
      </param-value>
   </init-param>
</servlet>
```

Lehetőség van JSP lapoknak is paramétert átadni, ebben az esetben ezeket is el kell helyezni a telepítésleíróban. A `<servlet-class>` helyett ilyenkor a `<jsp-file>` elemet használjuk:

```
<servlet>
   <servlet-name>rendeles</servlet-name>
   <jsp-file>
      /rendeles.jsp
   </jsp-file>
   <init-param>
      <param-name>jdbcURL</param-name>
      <param-value>
         jdbc:derby://myserver:1527//distedu
      </param-value>
   </init-param>
</servlet>
```

Ugyanaz a szervlet vagy JSP lap több névvel is konfigurálható, akár más inicializáló paraméterekkel is ellátható. A webkonténer minden egyes szervlet konfigurációra egy külön példányt hoz létre. A szervlet elemhez opcionálisan megadható a `load-on-startup` alelem is, amelynek segítségével közöljük a webkonténerrel, hogy induláskor töltse be a szervletet. Az alelemben megadható a betöltési sorrend is.

```
<servlet>
   <servlet-name>ListCourses</servlet-name>
   <servlet-class>view.ListCourses</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
```

<servlet-mapping>

A <servlet-mapping> elem hozzárendel egy URL mintát a megadott nevű szervlethez. A szervlet név társítása meg kell előzze az URL hozzárendelését, tehát először használjuk a <servlet> elemet és csak utána a <servlet-mapping> elemet. Az URL minta megadásánál a következő lehetőségek adottak:

- egy-az-egyhez típusú leképzés: egy adott kéréshez rendelt szervlet,
- kiterjesztés alapú leképzés: minden adott kiterjesztésű (pl. *.do) URL-t a megadott szervletre képzünk le,
- útvonal leképzése: pl. /admin/*, ez az admin mappára vonatkozó összes kérést a megadott szervlethez továbbítja,
- alapértelmezett szervlet megadása: / Ez a szervlet fogja kiszolgálni azokat a kéréseket, amelyekhez nem rendeltünk URL mintát.

```
<servlet-mapping>
   <servlet-name>ListCourses</servlet-name>
   <url-pattern>/list_courses.view</url-pattern>
</servlet-mapping>
```

<session-config>

A <session-config> elemmel a munkamenet (szesszió) lejárati ideje határozható meg percben.

```
<session-config>
   <session-timeout>
      30
   </session-timeout>
</session-config>
```

<welcome-file-list>

A <welcome-file-list> elemmel azokat az erőforrásokat határozzuk meg, amelyeket a webkonténer abban az esetben szolgáltat, ha a kérés URI-je egy mappát azonosít.

```
<welcome-file-list>
   <welcome-file>index.html</welcome-file>
   <welcome-file>welcome.html</welcome-file>
</welcome-file-list>
```

<taglib>

A <taglib> elemmel a fizikai elemkönyvtárnak egy szimbolikus névhez történő hozzárendelését végezzük:

```
<taglib>
   <taglib-uri>
      http://www.ms.sapientia.ro/sajatelemek.tld
   </taglib-uri>
   <taglib-location>
      /WEB-INF/sajatelemek.tld
   </taglib-location>
</taglib>
```

B. FÜGGELÉK – MAGYAR ÉKEZETES BETŰK

Olyan webalkalmazások esetén, amelyeket a Glassfish alkalmazásszerverre telepítünk és amelyekben a magyar ékezetes betűket akarjuk használni, szükséges a következő konfigurációt elhelyezni a `sun-web.xml` állományban:

```
<sun-web-app>
  ...

    <locale-charset-info default-locale="">
        <locale-charset-map locale="" charset=""/>
        <parameter-encoding default-charset="UTF-8"/>
    </locale-charset-info>

</sun-web-app>
```

C. FÜGGELÉK – HELYES VÁLASZOK

Webalkalmazások fejlesztése Java technológiákkal

1. B
2. D
3. A
4. C

Szervletek

1. C
2. B
3. E
4. B
5. D
6. A
7. C, D
8. C
9. C, D
10. B, D
11. B
12. C
13. B
14. D
15. A, B
16. E

Munkamenetek

1. B
2. B, C
3. C
4. C

5. A, D, E
6. B, C, D
7. B
8. C
9. D
10. E
11. D
12. D
13. B
14. B
15. C

Eseménykezelők és szűrők

1. B
2. A, C, E
3. C
4. A
5. A, B
6. C

JSP technológia

1. E
2. B, C
3. E
4. A, B, D
5. C, E
6. B, D
7. B
8. C, D, E
9. B, D
10. A, C
11. A, D
12. E
13. A, B

JSP elemkönyvtárak

1. B
2. B
3. D
4. D
5. D
6. A

Webalkalmazások biztonsága

1. A
2. D
3. A, C, E
4. B
5. A
6. E
7. C
8. A, B, D
9. C, D
10. D

SZAKIRODALOM

[1] BERGSTEN, Hans
2001 *JavaServer Pages.* Budapest, Kossuth
[2] CRAWFORD, William – KAPLAN, Jonathan
2003 *J2EE Design Patterns.* Sebastopol, O'Reilly
[3] ***
2006 *Developing Applications for the Java EE platform.* Broomfield, Sun Microsystems, FJ-310
[4] HALL, Marty – BROWN, Larry
2004 *Core Servlets and JavaServer Pages.* Sun Microsystems, Santa Clara, California
[5] IMRE Gabor (szerk.)
2007 *Szoftverfejlesztés Java EE platformon.* Budapest, SZAK Kiadó
[6] SZABÓ László
Webtechnológiák.
http://www.ms.sapientia.ro/~lszabo/webtechnologia/
[7] ***
2007 *Web Component Development with Servlet and JSP Technologies.* Broomfield, Sun Microsystems, SL-314

ABSTRACT

This course is suited for developers who know already the Java programming language and provides them with the skills to analyze, design, develop and deploy a web application. There are presented the basic technologies to create dynamic web content such as servlets and JSP and some auxiliary technologies such as JSP Tag Libraries, JDBC (for accessing databases) and web application security.

This course is intended to be a practical guide to the creation of small to medium scale web applications. One particularity of this material is that the web technologies are presented through a medium scale web application, whose components are developed starting from the second chapter and ending with the last chapter. Most chapters contain test questions intended for knowledge checking.

REZUMAT

Popularitatea platformei Java EE, care permite dezvoltarea aplicațiilor distribuite, este în continuă creştere. Aceasta se datorează multiplelor tehnologii Java care uşurează munca dezvoltatorilor de software distribuit. În acest curs, dintre aceste tehnologii, sunt prezentate cele strict necesare pentru dezvoltarea aplicațiilor web.

Primul capitol este o scurtă incursiune în tehnologiile de bază folosite în curs. Începând cu cel de al doilea capitol, sunt prezentate amănunțit tehnologiile de bază, cum ar fi servleturile şi paginile JSP, precum şi tehnologiile auxiliare, ca bibliotecile de elemente, JDBC pentru accesul bazelor de date, framework-ul Struts şi mecanismele de securitate.

Pentru parcurgerea acestui material, sunt necesare cunoştințe de programare în limbajul Java, cunoştințe de baze de date, precum şi cunoştințe fundamentale HTTP şi HTML. O caracteristică aparte a acestui material este că tehnologiile sunt prezentate prin prisma unei singure aplicații, ale cărei componente sunt dezvoltate începând cu cel de-al doilea capitol. Gradul de însuşire al conceptelor poate fi verificat cu ajutorul testelor de la sfârşitul capitolelor. Anexa C. conține şi răspunsurile corecte.

A SZERZŐRŐL

Antal Margit 1968-ban született Csíkkarcfalván. Középiskolai tanulmányait a csíkszeredai Matematika–Fizika Líceumban végezte 1982–1986 között, egyetemi tanulmányait pedig a Babeş–Bolyai Tudományegyetem Informatika Karán 1986–1991 között. Oktatói pályafutását a marosvásárhelyi Bolyai Farkas Líceumban kezdte, majd a Marosvásárhelyi Műszaki Egyetemen folytatta. 2003 óta főállású oktató a Sapientia – Erdélyi Magyar Tudományegyetem marosvásárhelyi karán, ahol programozási nyelvekkel kapcsolatos tantárgyakat tanít. Jelenleg minősített Sun oktató a Java programozás és a Java alapú webprogramozás témakörökben.

A SAPIENTIA –
ERDÉLYI MAGYAR TUDOMÁNYEGYETEM JEGYZETEI

Bege Antal
Számelméleti feladatgyűjtemény. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2002.

Bege Antal
Számelmélet. Bevezetés a számelméletbe. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2002.

Vofkori László
Gazdasági földrajz. Csíkszereda, Csíkszeredai Kar, Gazdaságtan Tanszék. 2002.

Tőkés Béla – Dónáth-Nagy Gabriella
Kémiai előadások és laboratóriumi gyakorlatok. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2002.

Irimiaş, George
Noţiuni de fonetică şi fonologie. Csíkszereda, Csíkszeredai Kar, Humán Tudományok Tanszék. 2002.

Szilágyi József
Mezőgazdasági termékek áruismerete. Csíkszereda, Csíkszeredai Kar, Gazdaságtan Tanszék. 2002.

Nagy Imola Katalin
A Practical Course in English. Marosvásárhely, Műszaki és Humán Tudományok Kar, Humán Tudományok Tanszék. 2002.

Balázs Lajos
Folclor. Noţiuni generale de folclor şi poetică populară. Csíkszereda, Csíkszeredai Kar, Humán Tudományok Tanszék. 2003.

Popa-Müller Izolda
Műszaki rajz. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Fodorpataki László – Szigyártó Lídia – Bartha Csaba
Növénytani ismeretek. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2004.

Marcuş, Andrei – Szántó Csaba – Tóth László
Logika és halmazelmélet. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2004.

Kakucs András
Műszaki hőtan. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Biró Béla
Drámaelmélet. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Biró Béla
Narratológia. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Márkos Zoltán
Anyagtechnológia. Marosvásárhely. Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Grecu, Victor
Istoria limbii române. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Varga Ibolya
Adatbázis-kezelő rendszerek elméleti alapjai. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2004.

Csapó János
Biokémia. Csíkszereda, Műszaki és Társadalomtudományi Kar, Műszaki és Természettudományi Tanszék. 2004.

Csapó János – Csapóné Kiss Zsuzsanna
Élelmiszer-kémia. Csíkszereda, Műszaki és Társadalomtudományi Kar, Műszaki és Természettudományi Tanszék. 2004.

Kátai Zoltán
Programozás C nyelven. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2004.

Weszely Tibor
Analitikus geometria és differenciálgeometria. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2005.

Györfi Jenő
A matematikai analízis elemei. Csíkszereda, Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2005.

Finta Béla – Kiss Elemér – Bartha Zsolt
Algebrai struktúrák – feladatgyűjtemény. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Antal Margit
Fejlett programozási technikák. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Csapó János – Salamon Rozália
Tejipari technológia és minőségellenőrzés. Csíkszereda, Műszaki és Társadalomtudományok Kar, Élelmiszertudományi Tanszék. 2006.

Oláh-Gál Róbert
Az informatika alapjai közgazdász- és mérnökhallgatóknak. Csíkszereda, Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Józon Mónika
Általános jogelméleti és polgári jogi ismeretek. Csíkszereda, Gazdaság- és Humántudományok Kar, Üzleti Tudományok Tanszék. 2007.

Kátai Zoltán
Algoritmusok felülnézetből. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2007.

Csapó János – Csapóné Kiss Zsuzsanna – Albert Csilla
Élelmiszer-fehérjék minősítése. Csíkszereda, Műszaki és Társadalomtudományi Kar, Élelmiszertudományi Tanszék. 2007.

Ágoston Katalin – Domokos József – Márton Lőrinc
Érzékelők és jelátalakítók. Laboratóriumi útmutató. Marosvásárhely, Műszaki és Humántudományok Kar, Villamosmérnöki Tanszék. 2007.

Szász Róbert
Komplex függvénytan. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2007.

Kakucs András
A végeselem-módszer alapjai. Marosvásárhely, Műszaki és Humántudományok Kar, Gépészmérnöki Tanszék. 2007.

Antal Margit
Objektumorientált programozás. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék, 2007.

Majdik Kornélia – Tonk Szende-Ágnes
Biokémiai alkalmazások. Kémiai laboratóriumi jegyzet. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék, 2007.

Györfy Jenő – András Szilárd
Valószínűségszámítás és lineáris programozás. A játékelmélet alapjai. Csíkszereda. Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2007.

Dimény Gábor
Minőségirányítási rendszerek. Marosvásárhely, Műszaki és Humántudományok Kar, Kertészmérnöki Tanszék. 2008.

Zsigmond Andrea
Minőségi és mennyiségi analitikai kémia laborkönyv. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2008.

Kátai Zoltán
Gráfelméleti algoritmusok. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2008.

Csapó János – Albert Csilla – Kiss Zsuzsanna
Élelmiszer-analitika. Válogatott fejezetek. Csíkszereda, Műszaki és Társadalomtudományi Kar, Élelmiszertudományi Tanszék. 2008.

Márton Gyöngyvér
Kriptográfiai alapismeretek. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2008.

Nagy Imola Katalin
A guidebook to Language Exams. English for Human Sciences. Marosvásárhely, Műszaki és Humántudományok Kar, Humántudományok Tanszék. 2008.

Gagyi József
Örökség és közkapcsolatok (PR). Marosvásárhely, Műszaki és Humántudományok Kar, Humántudományok Tanszék. 2008.

Fodor László
Szociálpedagógia. Marosvásárhely, Műszaki és Humántudományok Kar, Humántudományok Tanszék. 2008.

Fodorpataki László – Szigyártó Lídia – Bartha Csaba
Növénytani ismeretek. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2009.

Murádin János Kristóf
Nemzetközi kapcsolatok elmélete. Kolozsvár, Természettudományi és Művészeti Kar, Európai Tanulmányok Tanszék. 2009.

Biró Géza – Salamon Rozália Veronika
Élelmiszer-biztonság. Csíkszereda, Műszaki és Társadalomtudományi Kar, Élelmiszertudományi Tanszék. 2009.

A PARTIUMI KERESZTÉNY EGYETEM JEGYZETEI

Kovács Adalbert
: Alkalmazott matematika a közgazdaságtanban. Lineáris algebra. Nagyvárad, Alkalmazott Tudományok Kar, Közgazdaságtan Tanszék. 2002.

Horváth Gizella
: A vitatechnika alapjai. Nagyvárad, Bölcsészettudományi Kar, Filozófia Tanszék. 2002.

Angi István
: Zeneesztétikai előadások. I. Nagyvárad, Alkalmazott Tudományok Kar, Zenepedagógiai Tanszék. 2003.

Péter György – Kinter Tünde – Pajzos Csaba
: Makroökonómia. Feladatok. Nagyvárad, Alkalmazott Tudományok és Művészetek Kar, Közgazdaságtan Tanszék. 2003.

Angi István
: Zeneesztétikai előadások. II. Nagyvárad, Alkalmazott Tudományok Kar, Zenepedagógiai Tanszék. 2005.

Tonk Márton
: Bevezetés a középkori filozófia történetébe. Nagyvárad, Bölcsészettudományi Kar, Filozófiai Tanszék. 2005.

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-264-593694
E-mail: kpi@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szabó Beáta

Műszaki szerkesztés:
Antal Margit

Tipográfia:
Könczey Elemér

Készült a kolozsvári Gewalt nyomdában
100 példányban
Igazgató: Liliana Dadu