Debreceni Egyetem

Informatikai Kar

Webalkalmazás fejlesztése

Témavezető:
Adamkó Attila
Parragh Szabolcs

Készítette:
Fodor Attila
Programozó matematikus
Esti tagozat

Debrecen
2008

Tartalomjegyzék

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek Adamkó Attilának, aki hasznos tanácsokkal és észrevételekkel segítette a szakdolgozat megírását.

Köszönettel tartozom Parragh Szabolcsnak, amiért elvállalta a külső konzulensi feladatokat. A vele folytatott beszélgetések, úgy érzem, jó irányba befolyásolták a dolgozat megszületését.

Mindenképpen hálával tartozom Pribék Ádámnak, amiért bevont az első olyan projektbe, ahol lehetőségem volt megismerkedni a Spring keretrendszerrel.

Továbbá köszönöm Molnár Brigittának, feleségemnek, amiért türelemmel elviselte, hogy sokszor nem vele, hanem a szakdolgozatommal és a tanulmányaimmal kellett foglalkoznom.

Bevezetés

Az Internet a megjelenése óta töretlenül fejlődik, sőt, a fejlődés máig szinte semmit sem veszített kezdeti lendületéből. A fejlődésnek természetesen ára van és ennek az összegnek a jelentős részét különböző vállalkozások milliói teremtik elő azáltal, hogy befektetnek olyan megoldásokba, amelyek az interneten bárki által elérhetők. Ezek a megoldások túlnyomó részt valamilyen webes alkalmazást jelentenek, ezért bátran állíthatjuk, hogy jelenleg az Internet és ezen belül a web fejlődésének a kulcsa az a hatalmas piaci igény, amely a webalkalmazások iránt jelentkezik.

Témaválasztásomnak így kézenfekvő okai vannak. Egyrészt érthető módon olyan technológiákba kívánom befektetni az időmet és szellemi energiámat, amely iránt nagy a piaci érdeklődés és ez által a fejlődés mértéke is nagy, másrészt ez a téma szorosan összefügg azzal, amit a munkahelyemen is csinálok. Azonban, hogy egy percre se szakadjunk el a piaci igényektől, fontos megemlíteni a harmadik és egyben a legnyomósabb érvet, ami egy ilyen témájú szakdolgozat megírásához vezetett, ez pedig nem más, mint az a megrendelés, ami az anyagi hátteret is biztosította a szakdolgozat mellékleteként benyújtott szoftver kifejlesztéséhez.

Egy megrendelés általában nem írja elő, hogy milyen technológiával hozzuk létre a kívánt terméket, a megrendelőt a legtöbb esetben csupán az érdekli, hogy az általa megálmodott szolgáltatásokat az általa megálmodott módon érje el. Ezért a feladat megismerése után az első lépés az alkalmazandó technológia kiválasztása.

Jelen helyzetben tehát az igény az, hogy elkészítsünk egy olyan egyszerű alkalmazást, amelyen keresztül nagyszámú elektronikus levél küldhető ki azok számára, akik a megrendelő hírlevél szolgáltatására igényt tartanak. Fontos hogy ezt az email küldő alkalmazást a megrendelő a weben keresztül érje el, ami nem csak a megrendelő elvárásai miatt szükséges, hanem azért is mert a fejlesztés megkezdése előtt mások is jelezték, hogy szívesen igénybe vennének egy ilyen szolgáltatást.

Mivel az alkalmazásnak viszonylag egyszerű feladatot kell ellátnia, így a fejlesztést egy ember is kényelmesen el tudja végezni, ezért a továbbiakban egyedül kell hozzálátnom a program elkészítéséhez. Bár a technológia kiválasztásához szabad kezet kaptam, a választási lehetőségek széles skálájának gyorsan gátat szab a határidő, ami az én esetemben azt jelenti, hogy korábbi ismereteimre alapozva valamilyen Java technológiát kell választanom, amely alkalmas webalkalmazás létrehozására, nem túl nagy a tanulási görbéje, de elég komplex ahhoz hogy a későbbiek során más, összetettebb alkalmazások kifejlesztéséhez is igénybe vehessem.

A szakdolgozat célja tehát megtalálni azt az eszközt, amely megfelelő nem csak egy kisebb webes projekt elkészítéséhez, hanem nagyobb, komplexebb webalkalmazások megírásához is.

Ennek megfelelően az első fejezetben röviden néhány elterjedtebb, nem Java alapú megoldásról lesz szó, majd az azt követő fejezetben néhány standard Java-s eszközt is bemutatok. A harmadik fejezetben a választott technológia, a Spring Framework ismertetése következik. Mivel a program elkészítése során az akkor elérhető 2.5-ös verziót használtam, a fejezetben leírtak a Spring ezen verziójához igazodnak. Nagyobb komplexitású alkalmazásoknál, illetve ott, ahol nagyobb terhelés miatt valamilyen elosztott megoldásban gondolkodunk, jó választás lehet az EJB technológia használata. A negyedik fejezetben röviden az EJB-vel foglalkozom, de csak a perzisztenciakezelés szempontjából. A leírások során itt is az általam használt 3.0-ás verzió került ismerttetésre. Végül röviden az elkészült levelező alkalmazás is bemutatásra kerül.

Fontos megjegyezni, hogy az itt felsorolt témák mindegyike csak és kizárólag gyakorlati szempontok alapján került tárgyalásra, így szinte mindent igyekeztem forráskódrészleteken keresztül is bemutatni. Ahol lehetett, ott az elkészült program forráskódjából merítettem a példákat.

1. fejezet

A webalkalmazások kialakulása fejlődése

1.1. A WWW rövid története

A 60-as években az Egyesült Államok hadseregének megbízásából kifejlesztettek egy ARPANET-nek (Advanced Research Projects Agency Network of the Department of Defense) keresztelt számítógépes hálózatot, amelynek lényeges jellemvonása volt, hogy a hálózatban a kommunikáció zavartalan működése akkor is biztosított volt, ha a hálózat egyes elemei üzemképtelenné váltak. A 70-es évekre több hasonló elven működő hálózat is létrejött, amelyek egymástól elszigetelten működtek, de hamarosan felmerült az igény, hogy ezeket a hálózatokat egymáshoz kapcsolják és egyesítsék az adatátviteli módszereket. A 80-as években százával csatlakoztak ehhez a hálózathoz az egyetemek, főiskolák, kutatóintézetek, valamint az állami hivatalok. Még ebben az évtizedben megjelentek a magánfelhasználók is és a hálózat mérete robbanásszerű ütemben kezdett növekedni. Ma ezt a hálózatot nevezzük Internetnek és az egész világot behálózza.

Az Interneten számos szolgáltatás érhető el, de ezek között kétségtelenül a World Wide Web (vagy röviden WWW, illetve Web) a legnépszerűbb szolgáltatás, ami nem más, mint egy hipertext- és hipermédia-állományok által alkotott, világméretű információs rendszer. A Web története 1989-ben kezdődött, amikor a genfi CERN-ben (Centre Européen de la Recherche Nucléaire) ezen a néven elindult egy informatikai projekt. Tim Berners-Lee és csoportja 1991-re kidolgozták a hiperlinkeken alapuló információs rendszer koncepcióját. Az elképzelést az NCSA (National Center for Supercomputing Applications), az amerikai felsőoktatási intézmények számítástechnikai támogatását biztosító szervezet felkarolta, munkatársai Marc Andreessen vezetésével 1992-ben megalkották a Mosaic nevű Web-böngésző programot, amelyet hamarosan követtek az egyéb ma közismert böngészők (Netscape Navigator†, Microsoft Explorer, HotJava, Opera, stb.).

A hipertext hálózati hivatkozásokat tartalmazó dokumentum; a kapcsolattal ("link"-kel) ellátott szövegrészre kattintva más - gyakran másik gépen, másik országban tárolt - dokumentumhoz jutunk el. A hipermédia pedig képekkel, hangokkal, mozgóképekkel és/vagy szövegen belül indítható programokkal megtűzdelt hipertext. A hipertextek létrehozására fejlesztették ki a HTML-t, ami egy olyan jelölő nyelv, amely alkalmas a hiperlinkek definiálására és egyúttal

a létrehozott dokumentum kinézetét is szabályozza. A HTML az SGML (Standardized General Markup Language, szabványosított általános jelölőnyelv) egy leegyszerűsített részhalmazaként jött létre, illetve tartalmazza azt a néhány olyan kiegészítő elemet, amelyek segítségével az egyes HTML oldalak képesek egymásra hivatkozni a már említett hiperlinkek segítségével, valamint képes nem HTML-adatforrásokat, például képállományokat, szkripteket és stíluslapokat is használni. Az SGML 1985-ben vált hivatalos szabvánnyá. Elterjedését az okozta, hogy az egyes intézményekben egyre több elektronikus dokumentum halmozódott fel, amelyek különböző egymással inkompatibilis formátumban voltak tárolva. Elsőként, talán nem meglepő módon, az Egyesült Államok hadügyminisztériuma vezette be elektronikus dokumentumainak tárolására. A WWW megalkotói hasonló problémával kerültek szembe, hiszen az Internet egy olyan környezetet teremtett, amelyben bárki hozzáférhet az ott tárolt információkhoz, így elengedhetetlen volt, hogy ezek az adatok egy mindenki által használható formában kerüljenek ábrázolásra.

1.2. A webalkalmazások megjelenése

A HTML1 egyszerű lehetőséget nyújtott csupán a felhasználóval való interakcióra. A felhasználó linkek segítségével kiválaszthatta, hogy mely dokumentumot, HTML oldalt kívánja megtekinteni, a böngésző pedig betöltötte azt az oldalt, amelyre a kiválasztott link mutatott. A HTML2 megjelenésével lehetővé vált űrlapelemek elhelyezésére a weboldalakban, ezzel megvalósulhatott a valódi kétirányú kommunikáció, vagyis elhárult minden akadály az elől, hogy teljes értékű webes alkalmazásokat hozzanak létre.

Természetesen a webalkalmazások elkészítéséhez nem csak azt kellett megoldani, hogy hogyan juttat a felhasználó adatokat a webszerverhez, hanem azt is, hogy a webszerver hogyan tudja feldolgozni ezeket az adatokat és azok alapján hogyan tud dinamikus tartalmakat szolgáltatni.

CGI

Az első széles körben elterjedt és de facto szabvánnyá vált eljárás az NCSA (National Center for Supercomputing Applications) által kifejlesztett CGI (Common Gateway Interface) volt. A CGI létrehozásával egy olyan szabványos eljárást kívántak létrehozni, amely a kiszolgáló (ami nem feltétlenül egy webszerver) és a külső programok közötti információcserét határozza meg. Ennél a megoldásnál tehát a webszerver egy programhoz továbbította a kérést, majd a program által szolgáltatott eredményt jelenítette meg.

A bevezetőben említett feladat elvégzéséhez meg kell vizsgálnunk, hogy alkalmas lenne e ez a technológia a levélküldő alkalmazás elkészítéséhez. Bár a szövegfeldolgozási képességei miatt a Perl terjedt el leginkább mint olyan nyelv, amelyet CGI szkriptek írására használtak, ezek a programok bármilyen nyelven megírhatók, amely képes adatokat fogadni az STDIN-ről és adatokat továbbítani az STDOUT-ra, így a Java nyelv is alkalmas erre a feladatra. A későbbiekben látni fogjuk, hogy a Java erre a feladatra sokkal jobb lehetőségeket is nyújt, azonban fontos

megemlíteni, hogy miért nem ajánlott jelenleg CGI szkripteket használni webes alkalmazások létrehozására.

Amikor a kiszolgálóhoz olyan kérés érkezik, amely CGI programot igényel, a kiszolgálónak új processzt kell indítania a CGI program futtatásához, majd környezeti változókon és a standard bemeneten keresztül át kell adnia az összes olyan információt, amelyre a válasz elkészítésére szükség lehet. Az a körülmény, hogy minden egyes kéréshez új processzt kell indítani, jelentős mértékben igénybe veszi a kiszolgáló idejét és egyéb erőforrásait, ezért a kiszolgáló csak korlátozott számú kérésnek tud egyidejűleg eleget tenni.

FastCGI

Az Open Market cég által kifejlesztett FastCGI orvosolta a standard CGI azon hibáját, hogy minden kéréshez új processzt kellett létrehozni, úgy, hogy folyamatosan a memóriában maradó processzeket hoz létre. Azonban továbbra is megmaradt az a gond, ami a CGI-re jellemző, hogy nem tud interaktív módon együttműködni a webkiszolgálóval, mert külön processzben fut. Így például a CGI program nem képes írni a kiszolgáló naplófájljába.

PHP

1995-ben Rasmus Lerdorf elkészítette a PHP/FI (Personal Home Page / Forms Interpreter) nevű Perl szkript gyűjteményt, hogy nyomon kövesse az online önéletrajzához való hozzáférések számát. Később ezt újraírta C nyelven és kibővítette úgy, hogy bárki egyszerűen alkalmazhassa ezt az eszközt dinamikus webes alkalmazások készítésére.

1997-ben Andi Gutmans és Zeev Suraski a PHP/FI folytatásaként elkészítette a PHP 3.0-át, amely azonban csak 1998-ban jelent meg. Az egyik legnagyobb erősségét a PHP 3.0-nak a kiterjeszthetősége jelentette. Azon kívül, hogy a végfelhasználókat egy szilárd, sok különböző adatbázist támogató infrastruktúrával, protokollokkal és API-kkal szolgálja ki, a PHP 3.0 kiterjeszthetőségének lehetősége több tucat fejlesztőben ébresztett vonzalmat az iránt, hogy csatlakozzon és új kiterjesztéseket írjon hozzá.

A PHP a 2004-ban megjelent 5-ös verziótól kezdve támogatja az objektumorientált programfejlesztést. Korábban már készítettem egyszerű weboldalakat PHP segítségével, amely egyértelműen alkalmas egyszerű webes alkalmazások kifejlesztésére, és bár számos komplex webalkalmazás is készült már PHP-ban, mégis vannak bizonyos fenntartásaim vele kapcsolatban:

- A PHP alkalmazásokban keveredik a PHP és a HTML kód, ezért nehéz a kódot tisztán tartani.
- A PHP-ban írt kód biztonságossága erősen függ attól, hogy a programozó mennyire ismeri a különböző PHP modulokat, eszközöket, illetve magát a nyelvet.
- A PHP-ban nincs szálkezelés.

Nyilvánvalóan létezhetnek más prioritások is, amely miatt éppen a PHP a legmegfelelőbb választás egy projekt elkészítéséhez, de a jelenlegi alkalmazás elkészítésénél, ahol az egyes felhasználók több tízezer levelet küldhetnek ki naponta, valahogyan meg kellene oldani, hogy a levélküldés a háttérben történjen, ez pedig a szálkezelés teljes hiánya miatt legalábbis problematikus.

ASP

A Microsoft Active Server Pages (ASP) születése 1996 októberére tehető, amikor megjelent az 1.0- ás publikus béta, mint frissítés az Internet Information Server (IIS) 2.0-hoz. Ettől a ponttól az ASP lassan felfejlődött 2.x-é, majd végül 3.0-vá.

A kezdeti három verzióban az ASP az alapértelmezett nyelvként egy szkriptnyelvet használt: a VBScriptet. A szkriptnyelv használatának vannak hibái, a kód értelmezése meglehetősen nehézkes, ezért a VBScript mint alapértelmezett nyelv használatát sokan ki is kapcsolták (bár technikailag megoldható, hogy az ASP más nyelvet pl.: JScirpt vagy Perl-t használjon, de ez nem általános megoldás). Az így értelmezett ASP kódmodell komoly teljesítménybeli korlátozottságot jelentett. 2000 elején, Microsoft bevezette az új .NET Framewok-ot és vele együtt az ASP továbbfejlesztését az ASP.NET 1.0-t (eleinte ASP+ néven).

Mivel az ASP lapok nem Windows platformokon való futtatása nagyobb feladatok esetén komoly nehézségekkel járhat, az ilyen technológiával készült alkalmazások hordozhatóságának megoldása nehéz, ha egyáltalán lehetséges. Véleményem szerint a megrendelők körét is szűkíti az a tény, hogy alkalmazásunk megfelelő működése csak meghatározott platformokon garantálható.

2. fejezet

Java alapú webalkalmazások készítése

2.1. Java szervletek

A Java szervlet tulajdonképpen a kiszolgáló generikus bővítése - egy Java osztály, amely dinamikusan betölthető és így bővíti a kiszolgáló funkcióit.

A szervletek - eltérően a CGI és a FastCGI eljárástól, amelyek csak több processzben tudják kezelni a különböző programokat és/vagy kéréseket - ugyanazon processzen belül különböző szálakban vagy a háttérben működő kiszolgálók között megosztott több processszen belüli szálakban kezelhetők.

A Servlet API

A `javax.servlet` és a `javax.servlet.http` csomagok együtt alkotják a Servlet API-t. A javax.servlet csomagban a generikus, protokolloktól független szervleteket támogató osztályok és interfészek találhatók. Ezeket az osztályokat a javax.servlet.http csomagban lévő osztályok bővítik, hogy HTTP-specifikus funkciókkal lássák el az előbbieket. Egy protokolloktól független szervletnek a javax.servlet.GenericServlet alosztályt, míg egy HTTP-szervletnek a javax.servlet.http.HttpServlet alosztályt kell bővítenie, amely maga is a GenericServlet alosztály HTTP-specifikus tulajdonságokkal felruházott változata.

Az önállóan futtatható Java alkalmazásoktól eltérően a szervleteknek nem kell tartalmazniuk `main()` metódust. Ehelyett a kiszolgáló a kérések kezelésekor a szervlet más-más metódusait hívja meg. Minden olyan esetben, amikor egy kiszolgáló egy szervlethez továbbít egy kérést, meghívja a szervlet `service()` metódusát.

Ha tehát egy generikus szervletet készítünk, akkor a szervlet `service()` metódusát kell felülírni, hogy a kérést a saját feladatának megfelelően kezelje. A `service()` metódus két paramétert vesz át: egy kérési objektumot és egy válasz objektumot.

Egy HTTP szervletben a generikustól eltérően nem a `service()` metódust írjuk felül, hanem a HTTP kéréstől függően GET kérés esetén a `doGet()`, POST kérés esetén a `doPost()` metódust.

A szervletek életciklusa

Egy szervletkonténer többnyire úgy hajtja végre a szervleteket, hogy az összes szervletet egyetlen Java virtuális gépben futtattja azért, hogy maximális mértékben kihasználhassa a szervletek közötti adatmegosztást. Persze léteznek olyan konténerek, amelyekre ez nem igaz, hiszen vannak, amelyek támogatják a szervletek több háttérkiszolgálón történő megosztott végrehajtását is.

A szervletek az egyes kérések között objektumpéldányokként továbbra is megmaradnak, így az egyes kérések között nem kell új példányokat legyártani. Ez rendkívül hatékonnyá teszi a szervleteket a CGI szkriptekkel szemben, hiszen egy szervlet objektum sokkal kevesebb memóriaterületet igényel, mint a CGI szkriptek processzei, ráadásul nem kell minden kéréskor erőforrásokat mozgósítani a létrehozásukhoz.

A fentiek alapján egy szervlet életciklusa a következőképpen alakul:

1. A szervlet létrehozása inicializálása
2. A kérések kiszolgálása
3. A szervlet megsemmisítése, szemétgyűjtés

A kérések feldolgozásakor a már korábban említett `service()`, illetve `doGet()`, `doPost()` metódusok hívódnak meg. Azonban az appletekhez hasonlóan van lehetőség a szervletek inicializálását és megsemmisítését is befolyásolni. Inicializáláskor egyszerűen az `init()` metódus hívódik meg, a szervlet megsemmisülése, memóriából történő törlődése előtt pedig a `destroy()` metódus.

Webalkalmazás létrehozása Java szervlet segítségével

A Java nyelven írt webalkalmazások szigorúan meghatározott fájlszerkezettel rendelkeznek annak érdekében, hogy telepítésük a lehető legegyszerűbb módon a kiszolgáló mindenféle különleges beállítása nélkül elvégezhető legyen. Egy webalkalmazás fájlszerkezetének tehát a következőképpen kell kinéznie:

1. A webalkalmazásnak egy külön könyvtárban kell elhelyezkednie, vagy pedig egy .war kiterjesztésű tömörített állományban, amely tömörített állomány egyébként egy JAR fájl, csak más a kiterjesztése.
2. Rendelkeznie kell egy WEB-INF könyvtárral, amely tartalmazza a webalkalmazás konfigurációs fájljait, többek között egy web.xml fájlt, amely az alkalmazás telepítésleíró fájlja, továbbá ebben a könyvtárban vannak azok a könyvtárak, amelyek a webalkalmazás osztályfájljait tartalmazzák. Az osztályfájlok lehetnek JAR fájlokba tömörítetten, ekkor a WEB-INF/lib könyvtárba kell őket elhelyezni, vagy pedig tömörítetlenül, ebben az esetben pedig a WEB-INF/classes könyvtárba kell tenni őket.

Egy szemléletes példa kedvéért nézzük meg, hogy hogyan nézne ki egy Java szervlet segítségével megvalósított webalkalmazás, amely egy HTML űrlap kitöltése után dinamikusan generál egy újabb HTML oldalt. Először elkészítjük a szervletet (*HelloServlet.java*):

2.1.1. forráskód. *HelloServlet.java*

```
package hu.szakdolgozat.hello;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class HelloServlet extends HttpServlet {

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

        String mezo = request.getParameter("mezo");
        response.setContentType("text/html; charset=UTF-8");
        response.setHeader("Content-Language", "hu");
        PrintWriter writer = response.getWriter();
        writer.println("Ön a következőket gépelte be az űrlapmezőbe: <b>"+mezo+"</b>");
    }

}
```

Ezután elkészíthetjük az űrlapot tartalmazó HTML fájlt (*index.html*):

2.1.2. forráskód. *index.html*

```
<!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>Hello Servlet</title>
    </head>
    <body>
    <h1>Hello Servlet</h1>
    <form action="hello" method="post">
        <input type="text" name="mezo"><br/>
        <input type="submit" value="Küldés">
    </form>
    </body>
</html>
```

És végül a telepítésleíró fájl (*web.xml*):

2.1.3. forráskód. *web.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>hu.szakdolgozat.hello.HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
</web-app>
```

A fenti fájlokból már egyszerűen összeállíthatjuk a webalkalmazást, csupán a már ismertetett fájlszerkezetet kell kialakítani:

2.1.4. forráskód. *Egy webalkalmazás fájlszerkezete*

```
/WEB-INF/classes/HelloServlet.class
/WEB-INF/web.xml
/index.html
```

A fent vázolt példa alapján látható, hogy a szervletek segítségével egyszerűen hozhatunk létre webalkalmazásokat, de azt is láthatjuk, hogy csak alacsony szintű eszközöket biztosít a kérések és válaszok kezelésére. A request objektumból egyenként kell kiszedni az elküldött űrlapmezők értékeit, magunknak kell gondoskodnunk a validálásukról. Ami viszont még rosszabb, a HTML kimenet esetén a HTML tag-eket is a java kódba kell bedrótozni, így például még az a lehetőség sem adatik meg, hogy ezt egy HTML szerkesztővel készítsük el. Szerencsére nem kell bajlódnunk ezekkel az alacsony szintű megoldásokkal, hiszen számos Java API érhető el, amelyek kényelmes eszközöket biztosítanak, mind a kimenet megformázásához, mind a HTTP kérések és válaszok elkészítéséhez, ráadásul úgy, hogy nem kell lemondanunk a szervletek előnyeiről sem.

2.2. JSP

A JavaServer Pages vagy röviden a JSP a Sun Microsystem által kifejlesztett, szorosan a szervletekhez kötődő technológia. A JSP deklarált célja, hogy lehetővé tegye a dinamikus és a statikus tartalom szétválasztását. Fontos tudni, hogy ez csupán lehetőség, a JSP semmilyen módon nem kényszeríti ki a tartalom és a megjelenítés szétválasztását, így minden lehetőség adott arra, hogy akár üzleti logikákat kódoljunk JSP lapokon, ezzel végül zavaros és áttekinthetetlen kódot hozva létre.

JSP alkalmazások kifejlesztése

Egy JSP lap pontosan olyan, mint egy szokásos weboldal, azzal a különbséggel, hogy a normál HTML kód mellett JSP elemeket is tartalmaz, amelyek a lapnak a beérkező kérésektől függően változó tartalmát állítják elő. Mielőtt a kiszolgáló elküld egy ilyen lapot fel kell azt dolgoznia, ami azt jelenti, hogy a JSP lapot először átalakítja szervletté, majd azt végrehajtja.

A JSP lapok felépítésének megértéséhez nézzük végig, hogy milyen elemekből áll. A JSP elemeknek három típusa van: direktívák, akciók és szkript elemek.

- **A direktívák** olyan, a lappal kapcsolatos információkat specifikál, amelyek a lap elkérései között nem változnak: mint például a lapon használt szkript nyelv, a menet figyelemmel kísérésének az igénye, stb.
- **Az akcióelemek** valamilyen akciót hajtanak végre azon információk alapján, amelyek pontosan akkor állnak rendelkezésre, amikor az ügyfél elkéri a JSP lapot. Így például egy akcióelem az ügyféltől kapott paraméterek alapján keresést hajthat végre egy adatbázisban, vagy dinamikusan létrehozhat egy HTML táblázatot, amelyet valamilyen külső rendszerből beolvasott adatokkal tölt ki.
- **A szkript elemek** segítségével rövid kódokat vehetünk fel egy JSP lapra, így például egy `if` utasítást, vagy egy `for` ciklust, vagy bármilyen más Java kódot.

Ahelyett, hogy túlságosan is belemerülnénk a részletekbe álljon itt egy egyszerű (és nem túl sok értelemmel bíró) példa. A következő jsp oldalon 10-szer kiíratjuk az aktuális dátumot formázás nélkül:

2.2.1. forráskód. *index.jsp*

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!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>
    </head>
    <body>
```

```
    <h1>JSP Page</h1>
    <jsp:useBean id="date" class="java.util.Date" scope="page"/>
    <% for (int i=0; i<10; i++) { %>
    <%= date %><br/>
    <% } %>
    </body>
</html>
```

A webalkalmazás elkészítésénél ugyanolyan a fájlszerkezetet kell megvalósítani, mint amilyet a szervletek esetében már leírtam, sőt, ha az MVC (Model - View - Control) tervezési minta figyelembevételével szeretnénk elkészíteni a webalkalmazást, akkor szervletet is ajánlott készítenünk, amely a controller szerepét fogja betölteni. Ebben az esetben a JSP lapok természetesen a View részhez fognak tartozni, míg a Model-hez egyszerű Java bean-ek. Minden adott tehát ahhoz, hogy úgy hozzunk létre egy webalkalmazást, hogy közben jól strukturált kódot is alkossunk. De vajon hogyan fog kinézni a kódunk, ha mégis ennél a megoldásnál maradunk akkor, ha a webalkalmazásunk mérete kicsit nagyobb lesz, mint a korábban leírt szervletes példa. Miután nem érdemes egy webalkalmazáson belül minden kérést más és más szerlethez küldeni, ezért a legjobb amit tehetünk, hogy egyetlen szervletet alkalmazunk. Innentől kezdve minden GET kérés az egyetlen `doGet()` metódushoz és minden POST kérés az egyetlen `doPost()` fog érkezni. A legrosszabb, amit tehetnénk, hogy valamilyen elágaztató utasítást alkalmazva az összes kérés feldolgozó utasítást ebbe a két metódusba zsúfolnánk. Ehelyett a következő tervezési mintát szokták javasolni: Minden egyes kérés feldolgozására hozzunk létre egy Java osztályt, amelynek legyen egy publikus metódusa, amelyet a szervletből később meg tudunk hívni. Az egyszerűség kedvéért jó, ha ez a publikus metódus minden egyes kérést feldolgozó osztályban ugyanolyan szignatúrájú, és, hogy ettől nehogy eltérjünk, javasolt egy interfész elkészítése, amely tartalmazza ezen metódus deklarációját, és amelyet minden szóban forgó osztálynak implementálnia kell. Ha ezzel készen vagyunk, akkor a szervlet `init()` metódusában minden egyes kérést feldolgozó osztályt példányosítunk, és ezeket az objektumokat eltároljuk egy `Map`-ben, miközben a kulcsok egyszerű sztringek lesznek, amelyek utalnak a szóban forgó osztályokban implementált tevékenységekre. Például:

2.2.2. forráskód. *HelloServlet.java*

```
public class HelloServlet extends HttpServlet {
    private Map<Action> actionMap;

    @Override
    public void init() throws ServletException {
        actionMap = new HashMap<Action>();
        actionMap.put("login", new LoginAction());
        actionMap.put("logout", new LogoutAction());
    }

    ...
}
```

Tegyük fel, hogy a `LoginAction` és `LogoutAction` kérésfeldolgozó osztályok az `Action` nevű interfészünket implementálják, amely egy `doAction()` metódust deklarál. Ekkor a megfelelő kérésfeldolgozó műveletek elindítása például a `doPost()` metódusból a következő módon történhet:

2.2.3. forráskód. *HelloServlet.java*

```
...

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

    String action = request.getParameter("action");
    Action actionObject = actionMap.get(action);
    action.doAction();
}

...
```

Ahhoz, hogy az egész rendszer működőképes legyen már csak annyit kell tennünk, hogy minden egyes HTTP kérésnél egy `request` paraméterben el kell küldeni, hogy melyik kérés feldolgozó objektumot kívánjuk aktivizálni.

Bár a fent vázolt technikában érezhető egyfajta jól megkomponáltság, ugyanakkor észrevehető benne egy kis fapadosság is, amely valami jobb megoldás után kiált. Felmerülhet a kérdés, hogy miért is kell az `init()` metódusban kézzel elvégezni a példányosításokat és a `Map` feltöltését és miért ne jöhetne ez az egész egy konfigurációs állományból? Miért kell egyáltalán újból és újból létrehoznom egy szervletet, ha tudjuk, hogy annak mindig ugyanazt és ugyanúgy kell csinálnia? Többek között ezek azok a kérdések, amelyek miatt jobb technológia után kell néznünk, még akkor is, ha nem hiányoljuk azt a tengernyi kézreálló funkciót, amit más technológiák kínálnak, és a JSP nem.

2.3. JSF

A JavaServer Faces (JSF) keretrendszer 2004 márciusában jelent meg és egy lényegesen magasabb szintű eszközt adott a webfejlesztők kezébe, mint amilyet a JSP és a szervletek jelentettek. A JSF bizonyos értelemben a webes alkalmazások fejlesztését az asztali alkalmazások fejlesztéséhez hasonló módon teszi lehetővé. Fejlesztés során különböző állapottal rendelkező felületi elemeket, ún. komponenseket használhatunk, amelyekhez az asztali alkalmazásoknál megszokott módon eseménykezelőket rendelhetünk.

Az adatok megjelenítésében és kezelésében is számos új lehetőséget biztosít a JSP-hez képest. Az adatok tárolására itt is Javabeaneket használhatunk, azonban nem kell a HTTP requestekben turkálnunk, hogy hozzáférjünk ezekhez az adatokhoz. Egy űrlap elemeit képes elmenteni

egy beanben úgy, hogy automatikusan végrehajtja a szükséges adatkonverziókat, miközben beépített lehetőségeket biztosít az adatok validálására is. A felületen a megjelenítésnél és a szerveroldalon az adatok feldolgozásnál is egyaránt beanekben gondolkozhatunk, ami lényegesen megkönnyíti a fejlesztést, illetve a későbbiekben a program karbantartását.

A JSF használata során nem kell szervletet írnunk ahhoz, hogy megvalósítsuk az MVC tervezési mintát, mivel a JSF már készen tartalmazza azt a szervletet, amely a kéréseket feldolgozza. Fejlesztés során így csupán az a dolgunk, hogy megfelelően konfiguráljuk ezt a készen kapott szervletet és természetesen az, hogy implementáljuk az üzleti logikát. A konfigurációt alapértelmezetten a faces-config.xml-ben végezhetjük el, ahol mindenképpen meg kell adnunk azokat a beaneket, amelyekkel az alkalmazásnak dolgoznia kell.

Csak a legegyszerűbb lehetőségekre koncentrálva ismét egy egyszerű példán keresztül mutatom be, hogy hogyan lehet webes alkalmazásokat kifejleszteni a JSF segítségével.

Az alkalmazás egy nevet és egy jelszót kér be. Ha jelszó megfelelő, akkor továbbenged, ha nem, akkor ismét a bejelentkezési oldalt kapjuk meg, és egy kis üzenet figyelmezteti a felhasználót, hogy rossz jelszót adott meg.

Először megadjuk a JSF által biztosított FacesServlet szervletet a telepítés leíró fájlban, hogy a szervlet konténer elindíthassa (*web.xml*).

2.3.1. forráskód. *web.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.jsp</welcome-file>
    </welcome-file-list>
    <servlet>
        <servlet-name>FacesServlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>FacesServlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Majd létrehozzuk azokat a beaneket, amelyeket adattárolóként fogunk használni. Itt most az alkalmazás mérete miatt csak egyet hozunk létre (*UserBean.java*)

2.3.2. forráskód. *UserBean.java*

```
package jsf.pelda;

public class UserBean {
    private String name;
    private String password;
    private String wrongPassword;

    public String getName() {
        return name;
    }

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

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getWrongPassword() {
        return wrongPassword;
    }

    public void setWrongPassword(String wrongPassword) {
        this.wrongPassword = wrongPassword;
    }
}
```

Az űrlapok feldolgozását elvégezhetjük úgy, hogy az ún. dinamikus navigációt használjuk, illetve úgy is, hogy kihasználjuk a JSF azon tudását, hogy képes a felületen bekövetkezett eseményekre reagálni, úgy mint egy gomb lenyomására. Viszont a JSF esetében az eseménykezelők nem képesek arra, hogy az űrlap feldolgozása után egy másik oldalra navigáljanak, így a szerepük inkább az, hogy az oldalon bekövetkezett események hatására az adott felületen lehet különböző változásokat eszközölni a segítségükkel, ezért a továbbiakban nem tárgyalom a használatát.

A feladat elvégzéséhez ezek után létre kell hoznunk egy osztályt, amelyben implementáljuk az űrlap feldolgozásához szükséges metódusokat. Abban az esetben, ha az űrlap mezőket egyetlen bean-ben képezzük le, akkor elvben a feldolgozást végző metódusokat is elhelyezhetnénk a bean-ben, vagyis a jelenlegi példában akár a `UserBean` osztályban létrehozhatnánk egy metódust, amely ellenőrzi a beírt adatok helyességét. Mivel azonban elképzelhető, hogy olyan kódot kell írnunk, ahol egyidejűleg több bean-ben tárolt adatot egymással összefüggésben kell

feldolgozni, ezért most a példánkban egy olyan új osztályt hozunk létre (*Login.java*) , amely akár képes is lehet erre, bár a példában csak egy beant fog feldolgozni:

2.3.3. forráskód. *Login.java*

```
public class Login {
    private UserBean userBean;

    public String auth() {
        if (userBean.getPassword().equals("secret")) {
            return "success";
        }

        userBean.setWrongPassword("Hibás jelszó!");
        return "error";
    }

    public UserBean getUserBean() {
        return userBean;
    }

    public void setUserBean(UserBean userBean) {
        this.userBean = userBean;
    }
}
```

A további bonyolultság elkerülésének érdekében csak a *secret* szót fogadjuk el helyes jelszónak és a felhasználó nevet nem ellenőrizzük.

El kell még készítenünk a bejelentkeztető képernyőt (*index.jsp*):

2.3.4. forráskód. *index.jsp*

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <f:view>
        <head>
            <title>JSF Példa</title>
        </head>
        <body>
            <h:form>
```

```
                <h3>Adja meg az azonosítóját és a jelszavát</h3>
                <table>
                    <tr>
                        <td>Azonosító:</td>
                        <td>
                            <h:inputText value="#{user.name}"/>
                        </td>
                    </tr>
                    <tr>
                        <td>Jelszó:</td>
                        <td>
                            <h:inputSecret value="#{user.password}"/>
                             
                            <h:outputText value="#{user.wrongPassword}"
                                          style="color: #f00;"/>
                        </td>
                    </tr>
                </table>
                <p>
                    <h:commandButton value="Bejelentkezés"
                                     action="#{login.auth}"/>
                </p>
            </h:form>
        </body>
    </f:view>
</html>
```

és a sikeres bejelentkezést prezentáló oldalt (*welcome.jsp*):

2.3.5. forráskód. *welcome.jsp*

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<!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>JSF Példa</title>
    </head>
    <body>
    <h1>JSF Példa</h1>
    <f:view>
        Üdv az oldalon: <h:outputText value="#{user.name}"/>!
    </f:view>
```

```
    </body>
</html>
```

Ahhoz, hogy a webalkalmazásunk működjön a FacesServlet-el tudatnunk kell, hogy hogyan értelmezze a

```
#{user.name}
```

-hez hasonló kódrészleteket. Már fentebb említettem, hogy a szervlet konfigurációját alapértelmezett módon a *faces-config.xml* fájlban adhatjuk meg, úgyhogy most nézzük meg, hogy hogyan is néz ki ez a fájl.

2.3.6. forráskód. *faces-config.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
              version="1.2">

    <managed-bean>
        <managed-bean-name>user</managed-bean-name>
        <managed-bean-class>jsf.pelda.UserBean</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>login</managed-bean-name>
        <managed-bean-class>jsf.pelda.Login</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>userBean</property-name>
            <property-class>jsf.pelda.UserBean</property-class>
            <value>#{user}</value>
        </managed-property>
    </managed-bean>

    <navigation-rule>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/welcome.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>error</from-outcome>
            <to-view-id>/index.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>

</faces-config>
```

A <managed-bean> tag-ek között kell megadni azokat a beaneket, amelyeket a különböző weboldalakon valamilyen céllal el akarunk érni. Ennek megfelelően az *index.jsp* fájlban látott

```
<h:inputText value="#{user.name}"/>
```

tag annak a beannek a name attribútumát kívánja elérni, amely bean-t *user* néven hoztunk létre

```
<managed-bean-name>user</managed-bean-name>
```

és azt is láthatjuk, hogy ez a bean egy UserBean objektum

```
<managed-bean-class>jsf.pelda.UserBean</managed-bean-class>
```

Fentebb azt is láthattuk, hogy a `Login` osztály `public String auth()` metódusa egy `String`-et ad vissza, amelynek az értéke, vagy *success*, vagy pedig *error*. Ezek a "kódszavak" a navigation-rule szekcióban nyernek értelmet, vagyis ha a visszatérési érték *success*, akkor a *welcome.jsp*-t töltjük be, ha pedig *error* akkor az *index.jsp*-t.

Bár a JSF képességeinek csak nagyon kis részéről esett szó, láthattuk, hogy hogyan egyszerűsíti le a webalkalmazás fejlesztését. Nem kellett szervletet készítenünk, elég volt csupán konfigurálnunk a JSF által biztosított `FacesServlet`-et, így közvetlenül az üzleti logikára koncentrálhattunk. Ugyanakkor a JSF nem mondta meg nekünk, hogy hogyan válasszuk szét az üzleti logikát az adatokat tároló bean-ektől. A JSF-et minden további nélkül választhattam volna arra, hogy létrehozzam a megrendelt webalkalmazást, mégsem így tettem, és döntésem mögött igazából szubjektív okok voltak. Egyik ilyen ok, hogy a JSF-et egyelőre nem úgy képzelem el, mint olyan eszközt, amely teljes mértékig keretet adna egy webalkalmazás kifejlesztésére, hanem csak a megjelenítésben szánnék komolyabb szerepet neki. Ezen álláspontom igazolásához, vagy elvetéséhez még részletesebben tanulmányoznom kell a JSF által nyújtott lehetőségeket.

3. fejezet

Spring

A Spring egy sokoldalúan használható keretrendszer, amelyet Rod Jhonson fejlesztett ki, és először az *Expert One-on-One J2EE Design and Development* című könyvében publikált 2002-ben. A Spring célja leegyszerűsíteni a különböző üzleti alkalmazások kifejlesztését, így számos részből áll, de a teljes Spring ismeretére nincs szükség webalkalmazások kifejlesztéséhez.

A továbbiakban, ahogyan azt a bevezetőben is említettem a Spring 2.5-ös verziójának ismertetése következik, pontosabban a 2.x verzióé, mivel az itt leírtak mindegyike a 2.0-tól már elérhető.

3.1. Inversion of Control

Az Inversion of Control konténer (röviden IoC konténer) a Spring alap modulja, gyakorlatilag semmit sem tudunk csinálni ennek ismerete nélkül. Az IoC nem Spring sajátosság, hanem tervezési minta, amelyet számos keretrendszer használ, ezért ismerete mindenképpen hasznos, ha üzleti alkalmazásokat fejlesztünk. Népszerűsége Martin Fowler cikkének megjelenésétől kezdődött[11], de nézzük meg, hogy pontosan mit is takar ez a kifejezés.

Egy normál program esetén a fejlesztő a kódjában számos könyvtári függvényt hív meg, az IoC lényege ezzel szemben pont az, hogy különböző könyvtári függvények hívják meg a fejlesztő által írt függvényeket. Hogy mikor is hasznos ez, arra már láttunk egy példát a JSF-es mini alkalmazásban, akkor amikor a bean-eket definiáltam `faces-config.xml` fájlban, igaz akkor nem részleteztem, hogy mi is történik. A JSF-es példában tehát láthattuk, hogy a `Login` osztálynak volt egy `UserBean` típusú paramétere, amely beállításra került azáltal, hogy a konfigurációs XML fájlban megadtuk, hogy mi legyen ennek a paraméternek az értéke. Valójában a JSF azért tudta beállítani ezt a paramétert, mert a `Login` osztályban ehhez a paraméterhez egy set metódust is megadtunk és ezt a set metódust képes meghívni a JSF. Bár a paraméterek beállítását a különböző IoC konténerek leggyakrabban a set metódusok segítségével végzik, számos konténer támogatja a konstruktoron keresztüli befecskendezést is.

Az IoC-nek természetesen nem csak az az előnye, hogy inicializálhatjuk vele a beanjeinket, segítségével konfigurációs állományokon keresztül a kód újrafordítása nélkül befolyásolhatjuk az alkalmazás működését úgy, hogy kicserélhetjük az üzleti logikák implementációját. Példa-

ként tételezzük fel, hogy az alkalmazásunkban a perzisztenciakezelést egy osztály példányán keresztül végezzük. Ehhez az osztályhoz készíthetünk egy interfészt, így a perzisztencia kezelő logikának több megvalósítását is implementálhatjuk úgy, hogy a lényeges (kívülről is látható) metódusok szignatúrája minden megvalósításban ugyanaz lesz, így a hívó eljárásoknak nem kell ismerniük, hogy konkrétan melyik megvalósítás is lesz meghívva. Ahhoz, hogy egy IoC konténer segítségével a hívó osztályokban megváltoztathassuk a használt megvalósítást, a hívó osztályon belül mindenhol az interfészt kell szerepeltetnünk és a konfigurációs állományban kell megadni azt az osztályt, amelyben a a perzisztenciakezelés konkrét megvalósítása van.

A beanek konfigurálása

A JSF esetében azt láthattuk, hogy egy konfigurációs állományon keresztül megadtuk, hogy milyen objektumokat szeretnénk használni az alkalmazásban és azt is, hogy hogyan kell azokat inicializálni.

A Spring nagyon hasonló lehetőségeket kínál, azzal a különbséggel, hogy a konfigurációs állomány feldolgozására számos megoldást biztosít. Természetesen ezek a megoldásoknak egy jól meghatározott alap működés köré csoportosulnak, így mindegyik megoldásban nagyjából hasonló dolgok történnek. Ez az alap működés a `org.springframework.beans.factory.BeanFactory` interfészben van deklarálva. A `BeanFactory` interfész egyik legegyszerűbb és leggyakrabban használt megvalósítása a `org.springframework.beans.factory.xml.XmlBeanFactory` osztály. Használata rendkívül egyszerű:

3.1.1. forráskód. *BeanFactory objektum legyártása*

```
BeanFactory factory =
    new XmlBeanFactory(new FileSystemResource("c:/beans.xml"));
```

A konfigurációs állomány beolvasása után nem történik meg a beanek inicializálása, az csak akkor következik be ha egy adott beanre konkrétan szükségünk van. Egy bean elkérése a `BeanFactory`-tól a következőképpen történik:

3.1.2. forráskód. *Egy bean példányának az elkérése a BeanFactory objektumtól*

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

Egyszerű alkalmazásoknál, ahol az itt leírtaknál többre nincs szükség az `XmlBeanFactory` használata éppen megfelelő, ha azonban jobban ki akarjuk használni a Spring Framework által nyújtotta lehetőségeket akkor érdemes megismerkedni az `org.springframework.context.ApplicationContext` interfésszel, illetve az azt kiterjesztő osztályokkal. Az `ApplicationContext` interfész a `BeanFactory` interfész leszármazottja, így minden olyat tud, amit a `BeanFactory` de néhány új hasznos lehetőséget is biztosít:

- Eszközt biztosít szöveges üzentek feldolgozására, ezáltal könnyen nemzetköziesíthetjük az alkalmazásunkat.
- Egyszerűbb módot biztosít a fájlok eléréséhez.
- Közzétehetünk eseményeket, amelyek figyelésére eseményfigyelőket regisztrálhatunk.

Az `ApplicationContext` interfésznek is több megvalósítása van, ezek közül a legáltalánosabban használtak a `ClassPathXmlApplicationContext`, amely a konfigurációs állomány a class path-ból olvassa be, a `FileSystemXmlApplicationContext`, amely a fájlrendszerből olvassa be a konfigurációs állományt és a `XmlWebApplicationContext`, amely a webalkalmazásoknál használható. A konfigurációs állomány beolvasása bármelyik esetben a következő egyszerűséggel alakul:

3.1.3. forráskód. *Egy ApplicationContext objektum legyártása*

```
ApplicationContext context =
    new FileSystemXmlApplicationContext("c:/foo.xml");
```

Egy bean elérése a `BeanFactory`-nál leírtak szerint a `getBean` metódussal lehetséges. A beanek inicializálása azonban nem azok első elérésekor történik, hanem a konfigurációs állomány feldolgozásakor, így az alkalmazásnak már nem kell várnia a beanek legyártására.

Webalkalmazások fejlesztése esetén nem feltétlenül szükséges bajlódnunk a konfigurációs állomány feldolgozásával és a beanek elérésével, mivel a JSF-hez hasonlóan a Spring is tartalmaz előregyártott szervletet, amelyben már implementálták a beanek kezelésének logikáját. Sőt több ilyen szervletet is tartalmaz, amelyek közül a legáltalánosabban használt a `DispatcherServlet`.

A beanek legyártása

A továbbiakban maradjunk a `DispatcherServlet` használatánál, így már csak egy konfigurációs állományra van szükségünk. Ez a szervlet alapértelmezett módon a *dispatcher-servlet.xml* fájlból próbálja meg beolvasni a konfigurációs utasításokat, amelyet a WEB-INF könyvtárban keres. A konfigurációs állomány felépítése a következő:

3.1.4. forráskód. *dispatcher-servlet.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    ...

    <bean id="loginController"
          class="hu.nir.nmail.spring.controllers.LoginController">
        <property name="formView" value="login" />
```

```
        <property name="successView" value="base" />
        <property name="loginService" ref="loginService"/>
        <property name="downloadMailService" ref="downloadMailService"/>
        <property name="restrictionService" ref="restrictionService"/>
        <property name="commandName" value="loginData" />
        <property name="commandClass"
                  value="hu.nir.nmail.spring.beans.LoginData" />
        <property name="validator" ref="loginValidator" />
    </bean>

    ...
</beans>
```

Vagyis a <beans> gyökér tag-ek között felsoroljuk az összes olyan bean-t, amelyet a Spring IoC konténerén keresztül akarunk inicializálni. Minden egyes bean megadása <bean> tag-ekkel történik, amelynek két kötelező attribútuma van az egyik az id, az itt megadott néven kerül példányosításra a bean, a másik a class, ahol a bean típusát kell megadni. Az így létrehozott objektum példányváltozóit egyrészt a <property> tag-ek segítségével állíthatjuk be, ebben az esetben az IoC konténer a bean set metódusait használja a változók beállítására. Másik lehetőség a konstruktoron keresztüli inicializálás, ekkor <constructor-arg> tag-eket kell használni a <property> tag-ek helyett.

A fejlesztés során előfordulhat, hogy egy bizonyos bean-t több osztályba is be kell injektálni. Esetleg úgy érezzük, hogy jól automatizálható lenne a beanek konfigurálása ha a Spring el tudná dönteni, hogy hová milyen típusú beant kel injektálnia és nem kellene annyit gépelni. A Spring mindezekre képes is az ún. *autowiring* segítségével.

A Spring négyféle lehetőséget biztosít az autowiring megvalósítására. Bemutatásukhoz nézzük a következő egyszerű példát:

3.1.5. forráskód. *Felhasznalo.java, Cim.java*

```
public class Felhasznalo {
    private String nev;
    private Cim cim;
    ...

    public Felhasznalo(String nev, Cim cim) {
        ...
    }

    public void setNev(String nev) {...}
    public void setCim(Cim cim) {...}

    ...
}

public class Cim {...}
```

Adott tehát két osztály, amelyek közül most főleg a `Felhasznalo` érdekes, amelynek van egy `Cim` típusú változója. Az eddig ismertetett módszer alapján a konfigurációs állományba a következő sorokat kellene elhelyezni ahhoz, hogy inicializálhassuk ezeket a beaneket:

3.1.6. forráskód. *dispatcher-servlet.xml*

```
...

<bean id="cim" class="pelda.Cim"/>

<bean id="felhasznalo" class="pelda.Felhasznalo">
    <property name="nev" value="Gipsz Jakab"/>
<property name="cim" ref="cim"/>
</bean>

...
```

Most nézzük az autowiring négy esetét:

Név alapú: A Spring látja, hogy a `Felhasznalo` osztálynak van egy `cim` nevű változója és ilyen néven már létrehoztunk egy másik beant, ezért a `Felhasznalo` osztály `cim` nevű változóját ezzel a beannel inicializálja. A konfiguráció ezen módjához a következőt kell begépelnünk:

3.1.7. forráskód. *dispatcher-servlet.xml*

```
...

<bean id="cim" class="pelda.Cim"/>

<bean id="felhasznalo" class="pelda.Felhasznalo" autowire="byName">
    <property name="nev" value="Gipsz Jakab"/>
</bean>

...
```

Itt láthatóan a `<property name="cim" ref="cim"/>` sor vált feleslegessé azáltal, hogy használtuk a `<bean>` tag `autowire` paraméterét.

Típus alapú: A név alapján történő behelyettesítés természetesen okozhat zavarokat, ezért egy jobb módszer lehet, ha azt mondjuk a Springnek, hogy keressen olyan beaneket az inicializáláshoz, amelyek típusa megegyezik az adott példányváltozó típusával. A `Felhasznalo cim` példányváltozója tehát inicializálható egy `Cim` típusú beannel:

3.1.8. forráskód. *dispatcher-servlet.xml*

```
...

<bean id="cim" class="pelda.Cim"/>

<bean id="felhasznalo" class="pelda.Felhasznalo" autowire="byType">
    <property name="nev" value="Gipsz Jakab"/>
</bean>

...
```

Konstruktor alapú: Ez a módszer csak abban tér el a típus alapútól, hogy a függőség befecskendezés a konstruktoron keresztül fog történni.

Automatikus: Ha végképp nem tudjuk eldönteni, hogy melyik módszert kívánjuk használni a fenti három lehetőség közül a Springre bízhatjuk a választást. Ebben az esetben a Spring először a konstruktor alapúval fog próbálkozni, és ha nem talál olyan konstruktort, amely megfelelő lenne, akkor a típus alapúval fog kísérletezni. Igazából a név alapút soha nem fogja megpróbálni, mivel az egy nem túl megbízható lehetőség. Az autowiring technikát a Spring tehát nem használja alapértelmezetten, csak azoknál a beaneknél, ahol az `autowire` attribútumnak értéket adunk. Ez a működés megváltoztatható ha a <beans> gyökér tag-ben állítjuk be, hogy melyik autowire típust szeretnénk használni:

3.1.9. forráskód. *dispatcher-servlet.xml*

```
<beans default-autowire="byType">
...
</beans>
```

Az autowiring tehát nagyon hasznos lehet, de természetesen veszélyeket is hordozhat, ezért használatát mindig mérlegelni kell.

Az eddigi példákban semmit sem mondtam arról, hogy az IoC konténer hányszor hozza létre ezeket a beaneket az alkalmazás futása során. Valójában ha másként nem rendelkezünk a Spring mindig singletonokat hoz létre, vagyis a beannek csak egyetlen példányát gyártja le és ezt az egy példányt használjuk az alkalmazás működése során. Ez nem mindig hasznos, gondoljunk csak a legutóbbi példánkra, nem lenne szerencsés ha minden felhasználóhoz ugyanaz a cím tartozna, ezért a konfigurációs leírásban megadhatjuk, hogy hogyan akarjuk példányosítani a beanjeinket. Ehhez a <bean> tag `scope` attribútumát kell használni. Az attribútum a következő értékeket veheti fel: `singleton`, `prototype`, `request`, `session`, `global-session`. Ha a `scope` attribútumnak a `prototype` értéket adjuk, akkor a `BeanFactory getBean` metódusának minden egyes meghívásakor a bean egy új példánya jön létre, vagyis a következő sor begépelésével biztosíthatjuk, hogy minden egyes `Felhasznalo` objektumhoz másik `Cim` objektum tartozzon:

3.1.10. forráskód. *dispatcher-servlet.xml*

```
<bean id="cim" class="pelda.Cim" scope="prototype"/>
```

A `request` és `session` értékeket csak akkor adhatjuk a `scope` attribútumnak, ha webalkalmazást fejlesztünk, ekkor egy HTTP request, vagy egy HTTP session objektum fog létrejönni a `getBean` metódus meghívódásakor. A `global-session` érték csak portletek esetén használható.

3.2. Spring MVC

Már többször esett szó az MVC-ről (Model-View-Control), de eddig nem fejtettem ki, hogy pontosan mi is ez.

Az MVC modellt először a Xerox írta le a Smalltalk nyelvvel kapcsolatos különböző tanulmányaiban. Azóta azonban számos más, népszerű programozási nyelveken megírt GUI alkalmazás is átvette ezt a modellt. A modell alapgondolata az, hogy az alkalmazás adatait és az üzleti logikát, az adatok megjelenítését, valamint ezek kapcsolatát három önálló egységbe bontja szét, amelyeknek Modellezés (Model), Megjelenítés (View) és Vezérlés (Controller) a nevük. Az ilyen fajta szétbontás rugalmassá teszi az alkalmazást, mert így az adatok többféle módon jeleníthetők meg, és egyszerűen módosítható a megjelenítésük, az üzleti logika vagy az adatok fizikai megvalósítása (modellezés) pedig anélkül változtatható, hogy hozzá kelljen nyúlni a felhasználói felület kódjához.

Korábban láthattuk, hogy a vezérlés szerepét a JSP és a JSF esetében is egy szervlet látta el. Bizonyos értelemben a Spring esetében is így van ez, de itt a szervlet csak annyi vezérlési feladatot lát el, hogy kiválasztja a megfelelő vezérlőt. A megfelelő vezérlő kiválasztásában valamilyen `HandlerMapping` objektumra van szüksége. A kiválasztott vezérlőnek, amely minden esetben egy `Controller` objektum, átadja a `request` és `response` objektumokat. Egy jól strukturált programban a vezérlő objektumok nem, vagy csak minimális üzleti logikát tartalmaznak, ehelyett az ilyen jellegű kódokat service objektumokban kell lekódolni. A vezérlő a feldolgozás eredményeként előálló információkat, a szükséges megjelenítési információkkal egy `ModelAndView` objektumba csomagolja és ezt az objektumot visszaküldi a szervletnek. A szervlet a `ModelAndView` objektumban tárolt megjelenítési információt, ami egy logikai név, átadja egy *ViewResolver* objektumnak, amely előállítja azt a `View` objektumot, amely a kívánt információkat megjeleníti.

A vezérlő kiválasztása

Egy webalkalmazás elkészítéséhez kicsit közelebbről is meg kell ismerkedni a fentebb ismertetett komponensekkel, ezért először nézzük meg, hogy hogyan történik a megfelelő vezérlő kiválasztása. Az előbb sejtelmesen csak annyit mondtam erről, hogy ezt a szervlet valamilyen `HandlerMapping` objektum segítségével végzi, valójában többféle `HandlerMapping` implementáció is létezik, ezek közül nézzünk meg most néhányat.

SimpleUrlHandlerMapping: Az egyik legintuitívebben használható megoldás. Egyszerűen URL-ekhez kötjük, hogy melyik vezérlőt akarjuk használni. A webalkalmazásunkban minden link egy URL-re mutat, és minden <form> tag rendelkezik egy `action` attribútummal, amely szintén egy URL-t tartalmaz. Minden egyes ilyen URL-hez a konfigurációs állományban hozzárendelünk egy vezérlőt, így a szervlet tudni fogja, hogy az adott űrlap feldolgozásakor, vagy az adott linkre kattintáskor hová kell továbbítani a kérést a további feldolgozás érdekében. Én szintén ezt a módszert használtam a vezérlők kiválasztásához, így nálam a következőképpen néz ki ennek a konfigurációja:

3.2.1. forráskód. *dispatcher-servlet.xml*

```
<bean class=
        "org.springframework.web.servlet.handler.
         SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/login.htm">loginController</prop>
            <prop key="/base.htm">messageDownloadController</prop>
            <prop key="/logout.htm">logoutController</prop>
            <prop key="/send.htm">sendMessageController</prop>
            <prop key="/settings.htm">settingsController</prop>
            <prop key="/import.htm">importController</prop>
            <prop key="/setpop3.htm">pop3Controller</prop>
            <prop key="/informations.htm">infoController</prop>
            <prop key="/updateprocestable.htm">
                procesController
            </prop>
        </props>
    </property>
</bean>
```

ControllerClassNameHandlerMapping: Nagyon hasonlóan működik az előzőhöz, azzal a különbséggel, hogy nem kell kézzel megadnunk, hogy melyik URL-hez melyik vezérlő tartozik, mert egyszerűen kikövetkezteti azt. A logikája nagyon egyszerű, például egy **login**.htm-hez mindenképpen egy **login**Controller-t fog keresni, vagyis fogja az URL-t levágja a .htm részt és hozzáragasztja a Controller végződést. A fenti hosszú konfigurációs bejegyzés helyett így elegendő ennyit beírnunk:

3.2.2. forráskód. *dispatcher-servlet.xml*

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.mvc.
           ControllerClassNameHandlerMapping"/>
```

CommonsPathMapHandlerMapping: A `SimpleUrlHandlerMapping` használatától abban különbözik, hogy az URL hozzárendelést a vezérlő osztály kódjában kell megadni a következő módon:

3.2.3. forráskód. *HomePageController.java*

```
/**
  * @@org.springframework.web.servlet.handler.
      commonsattributes.PathMap("/home.htm")
  */
public class HomePageController
     extends AbstractController {
...
}
```

Vagyis minden egyes vezérlő osztály kódjában szerepeltetni kell a `PathMap` attribútumot és az így elkészített kód fordításához szükség van a Jakarta Commons Attributes fordítóra, amelyről bővebben annak weboldaláról[12] szerezhetünk információkat.

BeanNameUrlHandlerMapping: Szintén rendkívül egyszerűen használható. Az alapelv az, hogy mivel egyébként is mindegyik vezérlő osztályhoz kell készíteni egy konfigurációs bejegyzést az URL hozzárendelést ebben a bejegyzésben adhatjuk meg:

3.2.4. forráskód. *dispatcher-servlet.xml*

```
<bean name="/home.htm"
    class="com.roadrantz.mvc.HomePageController">
</bean>
```

Az első feladatunk tehát az, hogy kiválasszuk a megfelelő `HandlerMapping` implementációt, de ha nem tudunk választani, akkor nem kell sokat fáradoznunk. Mindegyik `HandlerMapping` osztály implementálja az `Ordered` interfészt, így mindegyik rendelkezik egy `order` példányváltozóval, amely egy egész számot vár értékül. Ez lehetőséget ad arra, hogy több `HandlerMapping` implementációt is használjunk egyszerre. A szervlet először azzal a handlerrel próbálkozik, amelyiknél az order változó értéke a legkisebb, ha az nem ad választ az URL controller összerendelésre, akkor tovább lép és megpróbálkozik a következő handlerrel. Ennek megfelelően a konfigurációs bejegyzések a következőképpen alakulhatnak:

3.2.5. forráskód. *dispatcher-servlet.xml*

```
<bean id="beanNameUrlMapping" class="org.springframework.web.
          servlet.handler.BeanNameUrlHandlerMapping">
  <property name="order"><value>1</value></property>
</bean>
<bean id="simpleUrlMapping" class="org.springframework.web.
          servlet.handler.SimpleUrlHandlerMapping">
  <property name="order"><value>0</value></property>
  <property name="mappings">
  ...
  </property>
</bean>
```

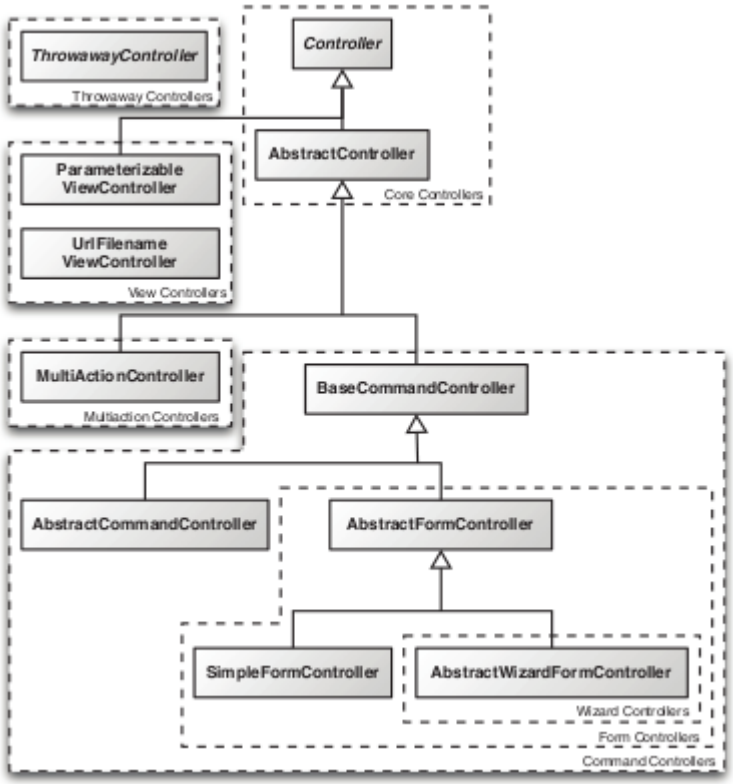
Ebben az esetben a szervlet először a `SimpleUrlHandlerMapping`-el próbálkozik, és ha nem jár sikerrel, akkor a `BeanNameUrlHandlerMapping`-hez fordul.

A vezérlők elkészítése

A következő lépés a különböző vezérlők elkészítése. A feladattól függően többféle vezérlőt készíthetünk, és ebben a Spring számos absztrakt osztállyal segít. Amint a 5.4 ábra mutatja ezen osztályok hierarchiája meglehetősen kiterjedt, ezért minden elemének bemutatására most nem is kerül sor, e helyett nézzük meg a leggyakrabban használtakat.

Leggyakoribb eset, ha egy linkre kattintás nyomán előálló HTTP GET kérést akarunk feldolgozni. Általában ilyen esetben nincs szükségünk validátor használatára, sem semmiféle komolyabb támogatásra, ezért megfelelő választás, ha a vezérlő osztályunkat az `AbstractController` leszármazottjaként hozzuk létre. Az `AbstractController` absztrakt osztály a `handleRequestInternal` metódust biztosítja számunkra és egyben garantálja, hogy a HTTP GET kérés esetén az osztályunk ezen metódusa lesz meghívva. A metódus bemenő paraméterei egy `HttpServletRequest` és egy `HttpServletResponse` objektum és a visszatérési értéke egy `ModelAndView` objektum. Használata azonban csak akkor célszerű ha a GET kérés során nem, vagy csak egy - két `request` paramétert kell feldolgozni. Amennyiben több kérési paraméter feldolgozására is szükség van, célszerűbb a `AbstractCommandController` absztrakt osztály kiterjesztése. Ez az osztály a `handle` metódust biztosítja számunkra, amelynek az előbbiekhez képest `Object` és egy `BindException` bemenő paramétere is van. Az új bemenő paraméterek lehetőséget adnak arra, hogy a kérési paramétereket egy objektumba csomagoljuk, így az adatok ellenőrzése és feldolgozása lényegesen egyszerűbb lesz. A kötés létrehozásához természetesen a konfigurációs bejegyzésben, vagy magában a vezérlő osztály kódjában meg kell adni, hogy mi lesz annak az objektumnak a típusa, amely a kérési paramétereket fogja tartalmazni. Az adatok validálására a konfigurációs bejegyzésben egy validátor osztály is megadható, ennek elkészítésére később részletesebben is kitérek.

Mivel az űrlapok feldolgozására használt `FormController` osztályok alkalmazása nagyon hasonló a `AbstractCommandController` használatához, ezért most csak az `AbstractController` kiterjesztésére nézzünk példát.



3.1. ábra. Spring MVC vezérlő hierarchia

A levélküldő webalkalmazás során néhány esetben nekem is szükségem volt olyan linkek elkészítésére, amelyekre kattintva az alkalmazásnak különböző szerveroldali műveleteket kellet végrehajtania ahhoz, hogy a következő oldalon a megfelelő információk jelenjenek meg. Így például az alkalmazásban létezik egy információs oldal, amely az Információk linkre kattintással érhető el és amely a végrehajtott műveletekről (folyamatban lévő, vagy már befejezett levélküldési folyamatokról) tartalmaz különböző hasznos információkat. A lap betöltéséhez a felhasználó azonosítóján kívül nincs más bemenő információra szükség ezért bátran választhattam a `AbstractController` osztály kiterjesztését, a kérést feldolgozó `InfoController` osztály elkészítésekor.

3.2.6. forráskód. *InfoController.java*

```
public class InfoController extends AbstractController {
    private String successView;
    private InfoService infoService;
    private String logoutView;

    @Override
    protected ModelAndView handleRequestInternal(
```

```
                        HttpServletRequest request,
                        HttpServletResponse response) {
        HttpSession session = request.getSession();
        NmailUser nmailUser =
                   (NmailUser) session.getAttribute("nmailUser");
        Map<String, Object> model =
                  infoService.processCommand(nmailUser);

        if (model == null) {
            return new ModelAndView(getLogoutView());
        }

        return new ModelAndView(getSuccessView(), model);
    }

    ...
}
```

A vezérlő használatához a következő konfigurációs bejegyzést kellett elkészíteni:

3.2.7. forráskód. *dispatcher-servlet.xml*

```
<bean id="infoController"
    class="hu.nir.nmail.spring.controllers.InfoController">
    <property name="successView" value="informations"/>
    <property name="infoService" ref="infoService"/>
    <property name="logoutView" value="login"/>
</bean>
```

A példa megértéséhez szükség van a `ModelAndView` osztály bővebb ismeretére. Egy `ModelAndView` osztály konstruktorában, vagy egy view nevet, vagy egy view nevet és egy `Map` típusú objektumot, vagy pedig egy view nevet , egy modell nevet és egy modell objektumot vár bemenő paraméterként. A view név egy egyszerű String, amelynek ismeretében a `ViewResolver` előállítja a megfelelő View objektumot, ahogyan azt már a fejezet bevezető részében leírtam. Ha a betöltendő oldalon semmilyen dinamikus információ megjelenítésére nincs szükség, akkor a `ModelAndView` konstruktorában elegendő egy view nevet szerepeltetni. Ha azonban, mint ahogyan a jelenlegi példában is dinamikus információkkal kell megtölteni az oldalt, attól függően, hogy ezek az információk mennyire összetettek a view név mellett, vagy egy `Map`-et, vagy egy modellnevet és egy modell objektumot is szerepeltetni kell. A `Map`-ben egyébként model objektumok vannak letárolva, ahol a kulcsok a modell nevek. A model objektumok tetszőleges Java objektumok, amelyek a megjeleníteni kívánt információkat tárolják és amelyekre a weboldalon belül a modell név segítségével lehet hivatkozni. Ennek megfelelően ha a megjeleníteni kívánt modell objektum például `User` típusú, lenne, amelynek mondjuk lenne egy `String` típusú `vezetekNev` és `keresztNev` paramétere, akkor azokat a weboldalon a következőképpen tudnánk megjeleníteni:

3.2.8. forráskód. *dispatcher-servlet.xml*

```
...
<body>
${user.vezetekNev}   ${user.keresztNev}
</body>
...
```

feltéve, hogy a `user` objektumot user névvel tettük elérhetővé a `ModelAndView` objektumban, vagyis valahogy így adtuk meg: `ModelAndView(viewName,"user",user)`.

Webalkalmazás készítése során előbb vagy utóbb belefutunk abba a problémába, hogy űrlapokat kell feldolgoznunk. Már említettem, illetve a 5.4 ábrából is kikövetkeztethető, hogy erre a feladatra a Spring külön osztályokat biztosít. Ezek közül talán legcélszerűbb a `SimpleFormController` használata. Ez az osztály eltérően az előzőekben leírtaktól nem absztrakt, így önmagában is használható, igaz meglehetősen ritkán van arra szükségünk, hogy az űrlapon bekért adatokat egy az egyben visszaküldjük megjelenítésre, de tulajdonképpen ez az, amit ez az osztály csinál. Ha ennél értelmesebb feladatok elvégzésére vágyunk, akkor ugyanúgy, mint ahogy eddig csináltuk, ki kell terjesztenünk ezt az osztályt és valamelyik `onSubmit` metódusát újra kell implementálni. Az `onSubmit` metódusok csupán paraméterezésükben térnek el, legszerényebb verziója csak egy `Object` típusú objektumot vár bemenő paraméterében, míg legteljesebb verziója ezen felül egy `HttpServletRequest`, egy `HttpServletResponse` és egy `BindException` típusú objektumot is vár. A választásunknak csak attól kell függnie, hogy mire lesz szükségünk implementálás során, a választásnak más egyéb következménye nincs.

A `SimpleFormController` rendelkezik egy nagyon hasznos metódussal, amelynek újraimplementálása csak kivételes helyetekben szükséges és ez pedig nem más mint a `formBackingObject` metódus, amely csak egy `HttpServletRequest` típusú objektumot vár bemenő paraméterében, és a visszatérési értéke egy `Object` típusú objektum. A metódus feladata, hogy a kérési paraméterekből egy olyan objektumot állítson elő, amelynek típusa megegyezik a commandClass paraméterben megadottéval. A `commandName` és `commandClass` paraméterek beállítását a form feldolgozásához létrehozott vezérlőnk konfigurációs bejegyzésében végezhetjük el.

Lehetőség van validátor osztály megadására is, a `SimpleFormController` ezen osztály segítségével fogja validálni a beérkezett kérési paramétereket még mielőtt az `onSubmit` metódus meghívódna. Egy validátor osztály elkészítéséhez a Spring által biztosított `Validator` interfészt kell implementálni. Az interfész két metódust biztosít számunkra, amelyek közül a `validate` az érdekesebb, ugyanis a validálást végző osztály ezen metódusát hívja meg a vezérlő osztály.

Rövid példaként a levélküldő alkalmazás bejelentkező űrlapjának feldolgozását nézzük meg:

3.2.9. forráskód. *LoginController.java*

```
public class LoginController extends SimpleFormController {
    private LoginService loginService;
    private String errorView;
    private DownloadMailService downloadMailService;
    private RestrictionService restrictionService;

    @Override
    @SuppressWarnings("unchecked")
    protected ModelAndView onSubmit(HttpServletRequest request,
              HttpServletResponse response, Object command,
              BindException errors) {
        HttpSession session = request.getSession();

        //login
        Map<String,Object> loginData = loginService.processCommand(command);
        NmailUser nmailUser = (NmailUser) loginData.get("nmailUser");
        if (nmailUser == null) {
            return new ModelAndView(getFormView(),"loginData",command);
        }

        session.setAttribute("nmailUser", nmailUser);

        //download last 5 mail
        MailsData mailsData = (MailsData) session.getAttribute("mailsData");
        Map<String,Object> model =
               downloadMailService.processCommand(nmailUser, mailsData);
        if (model == null) {
            return new ModelAndView(getFormView(),"loginData",command);
        }
        session.setAttribute("mailsData", model.get("mailsData"));

        Map<String,Object> restrictionMap =
                    restrictionService.processCommand(nmailUser);
        session.setAttribute("restriction", restrictionMap.get("restriction"));
        return new ModelAndView(getSuccessView(),model);
    }

    ...
}
```

Az ehhez tartozó konfigurációs bejegyzések a következőképpen néznek ki:

3.2.10. forráskód. *dispatcher-servlet.xml*

```
<bean id="loginController" class="hu.nir.nmail.spring.controllers.LoginController">
    <property name="formView" value="login" />
    <property name="successView" value="base" />
```

```
    <property name="loginService" ref="loginService"/>
    <property name="downloadMailService" ref="downloadMailService"/>
    <property name="restrictionService" ref="restrictionService"/>
    <property name="commandName" value="loginData" />
    <property name="commandClass" value="hu.nir.nmail.spring.beans.LoginData" />
    <property name="validator" ref="loginValidator" />
</bean>

<bean id="loginValidator" class="hu.nir.nmail.validators.LoginValidator">
    <property name="dataManager" ref="dataManager"/>
</bean>
```

Végül a validátor kódja:

3.2.11. forráskód. *LoginValidator.java*

```
public class LoginValidator implements Validator {

    private DataManager dataManager;

    public boolean supports(Class clazz) {
        return clazz.equals(LoginData.class);
    }

    public void validate(Object target, Errors errors) {
        LoginData loginData = (LoginData) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
                                    "loginName", "required.loginName");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
                                    "password", "required.password");
        NmailUserLoginManager nmailUserLoginManager =
                            dataManager.getNmailUserLoginManager();
        Long userId = nmailUserLoginManager.getUserId(loginData.getLoginName(),
                                                      loginData.getPassword());
        if (userId == null) {
            errors.reject("error.login");
        }
    }

    ...
}
```

A validátor minden esetben hibát jelez, ha az űrlapmezők nincsenek kitöltve, vagy az azonosító és jelszó páros nem megfelelő. Az üres űrlapmezőket a `ValidationUtils rejectIfEmptyOrWhitespace` metódusával lehet hibaként visszajelezni.

Megjelenítés

Az adatok feldolgozásával készen vagyunk, már csak egy dolog maradt hátra, a megjelenítés. A fejezet bevezető részében már leírtam, hogy a `ModelAndView` objektum a vezérlő osztálytól a szervlethez kerül és innen pedig a `ViewResolver`-hez. A feladatunk tehát az, hogy megtudjuk, mi is az a `ViewResolver` és hogyan is működik. A `ViewResolver` valójában egy interfész, így elvben ha egy osztály implementálja ezt az interfészt, akkor az alkalmassá válik arra, hogy meghatározza, hogy a `ModelAndView` objektumban visszaadott view név hatására mi jelenjen meg a böngészőben. A `ViewResolver`-ek valójában egy `View` objektumot állítanak elő, amelynek megfelelő megjelenítéséről már a szervlet gondoskodik. A Spring azonban nem csak a `ViewResolver` interfészt biztosítja a számunkra, hanem számos előre implementált osztállyal is a segítségünkre van, ezek közül néhánnyal érdemes közelebbről is megismerkedni.

Nézzük elsőként az `InternalResourceViewResolver` osztályt, amely a legegyszerűbb megoldás. Használatához egyszerűen csak fel kell vennünk a konfigurációs állományba és be kell állítanunk két fontos példányváltozóját. Mindkét példányváltozó String típusú, az egyik neve prefix, ennek az értéke határozza meg a megjelenített oldal URL-jének előtagját, míg a másik a suffix névre hallgat és értéke az URL suffixét határozza meg. Az `InternalResourceViewResolver` a beállított prefix és suffix értékek alapján végrehajtja a következő konkatenációt:

```
URL = prefix + viewNev + suffix
```

Az így előállt URL információt beállítja egy `View` típusú objektum `path` példányváltozójának értékeként, majd ezt az objektumot adja át a szervletnek megjelenítésre. Nézzük meg végül, hogy mit kell tennünk annak érdekében, hogy munkába állítsuk a webalkalmazásunkban az `InternalResourceViewResolver`-t:

3.2.12. forráskód. *Az InternalResourceViewResolver konfigurációja*

```
...
<bean id="viewResolver" class=
    "org.springframework.web.servlet.view.
          InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
...
```

Az így konfigurált `InternalResourceViewResolver` tehát ha azzal a view névvel találkozik, hogy home, akkor egyszerűen összeállítja a következő URL-t: `/WEB-INF/jsp/home.jsp` vagyis ez lesz az az oldal, amelyik végül a böngészőbe be fog töltődni.

A `InternalResourceViewResolver` nyilvánvalóan mindig azonos típusú view objektumokat fog gyártani, hogy pontosan milyet, az beállítható a `viewClass` példányváltozóján keresztül. Előfordulhat azonban, hogy az alkalmazásnak egy bizonyos ponton nem csak HTML

kimenetet kell tudni előállítani, hanem XML-t, PDF-et, vagy bármi mást. Ilyen feladat esetén jól jön egy olyan `ViewResolver` osztály, amely tetszőlegesen sokféle view-t tud előállítani. A `BeanNameViewResolver` pontosan ilyen feladatok ellátására képes. A `ModelAndView` objektumban megkapott view nevet nem egy URL összeállítására használja, hanem a konfigurációs állományban olyan bean definíciót keres, amely bean a view névvel azonos néven lett létrehozva. Ha tehát a view név mondjuk *info*, akkor a következő bean-t fogja megkeresni a konfigurációs fájlban:

3.2.13. forráskód. *dispatcher-servlet.xml*

```
<bean id="info" class="pelda.InfoView"/>
```

és végül ezt a beant továbbítja a szervletnek megjelenítésre.

Gyakorlatilag ugyanilyen megfontolásból lehet hasznos a `XmlFileViewResolver`, amely egy külön megadott XML fájlból olvassa ki a megjelenítendő view objektumot, illetve a `ResourceBundleViewResolver`, amely ugyanerre egy properties fájlt használ.

Nyilvánvalóan ezeken kívül létezik még számos más előregyártott `ViewResolver` osztály, de mi magunk is készíthetünk, mint ahogyan azt már korábban említettünk, ezért most nem lenne sok értelme ha mindegyiken végigrágnánk magunkat. Ami még lényeges, hogy a `ViewResolver` interfész is biztosít egy `order` változót, amely lehetővé teszi azt, hogy több `ViewResolver`-t is használjunk egyidejűleg rangsorolva ugyanúgy, ahogyan a különböző `HandlerMapping` megoldások közül is választhattunk.

A megjelenítés kérdéséhez tartozik, hogy milyen template-et választhatunk a weboldalak elkészítéséhez. A Spring ebben a kérdésben rendkívül megengedő, gyakorlatilag bármit használhatunk. Én az alkalmazás elkészítéséhez a JSP-t választottam, de a jövőben mindenképpen a JSF felé fogok orientálódni, mivel lényegesen gazdagabb eszközkészlettel rendelkezik.

3.3. Perzisztencia kezelés

Általában az egyes keretrendszerek valamilyen speciális feladat megoldására koncentrálva adnak eszközöket a fejlesztők kezébe. A Spring mindenképpen eltér ezektől, és afféle svájci bicskaként viselkedik, így a perzisztenciakezeléshez is igyekszik némi segítséget adni. A némi segítség azt jelenti, hogy nem teljesen önerejéből oldja meg a perzisztencia kezelést, hanem nagymértékben a Hibernate-re támaszkodik és különböző wrapper osztályokat biztosít a Hibernate használatához.

A Hibernate-tel való kapcsolattartás legfontosabb eleme a `org.hibernate.Session` interfész. Ezen interfész segítségével érhetjük el az olyan alap adatkezelő műveleteket, mint a mentés, törlés, adatok beolvasása, stb. A Spring a `HibernateTemplate` osztályt biztosítja számunkra ahhoz, hogy igénybe vehessük a `Session` interfész által nyújtott szolgáltatásokat. A perzisztencia kezelés megvalósításához tehát először mindenképpen referenciát kell szereznünk egy `Session` objektumra, ezt a `SessionFactory` interfészen keresztül tehetjük meg. Nekünk

azonban elegendő egy megfelelően konfigurált `SessionFactory` objektumot előállítani, és ezt átadni a `HibernateTemplate`-nek, amely innentől kezdve minden szükséges lépésről gondoskodik.

A `SessionFactory` konfigurálása előtt mindenképpen meg kell említeni, hogy a Hibernate pontosan mit is csinál. A Hibernate lehetővé teszi, hogy az adatbázisban tárolt adatokhoz objektum orientált módon férjünk hozzá. Ezt úgy teszi lehetővé, hogy az adatbázisban lévő táblákat Java osztályoknak felelteti meg, így a táblában lévő adatokat objektumok listájaként fogjuk megkapni. A `SessionFactory` konfigurálásakor tehát először is meg kell adnunk azokat a konfigurációs fájlokat, amelyek leírják, hogy a Hibernate melyik táblát milyen osztálynak feleltessen meg. Hogy a gyakorlatban is érzékelhető legyen miről van szó, lássunk erre egy egyszerű példát. Tételezzük fel, hogy a következőképpen néz ki az adatbázis táblánk:

3.3.1. forráskód. *A USER tábla*

```
CREATE TABLE USER (
id serial PRIMARY KEY,
user_name text NOT NULL,
password text NOT NULL
);
```

és ennek a táblának a következő Java osztályt akarjuk megfeleltetni:

3.3.2. forráskód. *User.java*

```
package pelda;

public class User {
    private int id;
    private String userName;
    private String password;

    ...
}
```

akkor a konfigurációs fájl, amely az adatbázis táblához hozzárendeli a `User` osztályt így kell, hogy kinézzen:

3.3.3. forráskód. *User.hbm.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  '-//Hibernate/Hibernate Mapping DTD 3.0//EN'
  'http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd'>
```

```
<hibernate-mapping auto-import="true" package="pelda">
    <class name="User" table="USER">
        <id name="id" column="ID" unsaved-value="null">
            <generator class="native"/>
        </id>
        <property name="userName" column="USER_NAME"/>
        <property name="password" column="PASSWORD"/>
    </class>
</hibernate-mapping>
```

vagyis egyszerűen megmondjuk, hogy melyik osztály melyik táblának lesz a megfelelője és, hogy az egyes oszlopnevek az adott osztály melyik példányváltozójához fog tartozni. Ezekben a konfigurációs fájlokban természetesen a táblák közötti különböző kapcsolatok is leírhatóak. Azonban nem kell feltétlenül XML fájlokban megadni ezeket az összerendeléseket, mivel az osztályokban annotációkkal is megadhatjuk ezeket az információkat, ebben az esetben a `SessionFactory` konfigurációs beállításaiban fel kell sorolni, az annotált osztályokat. Az annotációk segítségével egyszerűen rendelhetünk a kódhoz különböző metaadatokat, használatára példát is láthatunk az EJB-ről szóló fejezetben.

A Hibernate mapping beállításokon kívül természetesen szükség van az adatbázis elérhetőségének megadására is illetve a Hibernate dialektus megadására (ez arról ad információt, hogy milyen adatbázist használunk, pl.: MySQL, Oracle, PostgreSQL, stb.), amely beállítási információkat szintén a `SessionFactory` osztálynak kell átadni annak konfigurációja során. Végezetül nézzük meg, hogy a lehető legegyszerűbb esetben hogyan jutunk el konfigurációk során egy használható `HibernateTemplate` típusú objektumhoz:

3.3.4. forráskód. *dispatcher-servlet.xml*

```
<beans>
    ...

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
             <value>jdbc:postgresql://localhost:5432/peldaadatbazis</value>
        </property>
        <property name="url">
             <value>org.postgresql.Driver</value>
        </property>
        <property name="username">
             <value>felhasznalo</value>
        </property>
        <property name="password">
             <value>titkosjelszo</value>
        </property>
    </bean>
```

```
    <bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="mappingResources">
            <list>
                <value>
                        pelda/User.hbm.xml
                </value>
            </list>
        </property>

        <property name="hibernateProperties">
            <props>
                    <prop key="hibernate.dialect">
                        org.hibernate.dialect.PostgreSQLDialect
                    </prop>
            </props>
        </property>

        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.HibernateTemplate">
       <property name="sessionFactory" ref="sessionFactory" />
   </bean>

    ...
</beans>
```

Természetesen ahhoz, hogy a `HibernateTemplate` objektumunkat használni is tudjuk be kell injektálni az egyik saját osztályunkba:

3.3.5. forráskód. *UserDao.java*

```
package pelda;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import pelda.User;
import java.sql.SQLException;
import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.springframework.orm.hibernate3.HibernateCallback;

public class UserDao extends HibernateDaoSupport {

    public UserDao() {
```

```
        super();
    }

    public void update(User user) {
        this.getHibernateTemplate().update(user);
    }

    public void save(User user) {
        this.getHibernateTemplate().save(user);
    }

    public void delete(User user) {
        this.getHibernateTemplate().delete(user);
    }

    public List getUser(String userName, String password) {
        final String name = userName;
        final String pw = password;

        List list =  getHibernateTemplate().executeFind(new HibernateCallback() {
                public Object doInHibernate(Session session)
                        throws HibernateException, SQLException {

                        StringBuffer sb = new StringBuffer(100);
                        sb.append("select distinct user ");
                        sb.append("from User user ");
                        sb.append("where user.userName = '"+name+"' ");
                        sb.append("and user.password = '"+pw+"'");

                        Query query = session.createQuery(sb.toString());
                        List list = query.list();

                        return list;
                }
        });

        return list;
    }
}
```

Az osztályunkat (`UserDao`) ahogyan a példában is láttuk a `HibernatDaoSupport` osztályból kell származtatnunk, így könnyedén hozzáférhetünk a beinjektált `HibernateTemplate` objektumhoz, amely segítségével a példában látható módon hajthatunk végre minden szükséges adatbázis műveletet. A `UserDao` bean konfigurációja a következő egyszerű módon adható meg:

3.3.6. forráskód. *dispatcher-servlet.xml*

```
...
<bean id="userDao" class="pelda.UserDao">
   <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
...
```

A levélküldő webalkalmazás elkészítése során végül nem ragaszkodtam a Spring által nyújtott perzisztenciakezelő megoldáshoz, mert bár kétségtelenül egyszerű módot ad az adatbázis eléréséhez, gondolnom kellett arra is, hogy a későbbiek során más alkalmazások számára is könnyen hozzáférhetővé kell tennem az adatbázisban tárolt adatokat, akár olyan alkalmazások számára is, amelyek nem a Spring keretrendszer segítségével lettek létrehozva. Ha pedig mindezt a lehető legegyszerűbb módon szeretném elérni, akkor kétségtelenül akad erre lényegesen jobb megoldás is, ahogyan azt a következő fejezetben látni is fogjuk.

Azt azonban leszűrhetjük, hogy a Spring egy kitűnő választás lehet webalkalmazások készítésére, hiszen amint láthattuk beépített eszközeivel világos, jól strukturált webalkalmazások hozhatók létre akkor is, ha kezdő programozóként kevés tapasztalattal rendelkezünk ahhoz, hogy mások számára is jól érthető, megfelelően szervezett kódot készítsünk. Mindezek mellet kiterjedt eszközkészlete azt is lehetővé teszi, hogy úgy hozzunk létre alkalmazásokat, hogy nem kell feltétlenül kilépnünk a Spring világából.

4. fejezet

Perzisztenciakezelés EJB segítségével

A Sun hivatalos definíciója alapján az Enterprise JavaBeans (EJB) komponens alapú, elosztott üzleti alkalmazások fejlesztését és telepítését támogató architektúra specifikációja. Az Enterprise JavaBeans architektúrában készített alkalmazások skálázhatóak, tranzakciósak és több felhasználót kezelve biztonságosak. Egy ilyen alkalmazást elég megírni egyszer, és ezután bármely olyan szerver platformon telepíthető lesz, amely támogatja az Enterprise JavaBeans specifikációt.

Az EJB komponensek a JavaBean-ektől eltérően nem fejlesztői komponensek, hanem önállóan telepíthetők egy konténerbe, amely futásidőben különböző szolgáltatásokat biztosít a számára, és mint ilyen, rendkívül jól alkalmazható elosztott, többrétegű alkalmazások fejlesztésére.

A jelenlegi projektben az EJB-t a perzisztenciakezelés megoldására alkalmaztam, és ehhez a jelenleg elérhető 3.0-ás verziót használtam. Ezt a feladatot elvégezhettem volna a Spring által nyújtott lehetőségekkel is, viszont a levélküldő alkalmazáshoz egy adminisztrációs felület is tartozik, amely egy teljesen különálló nem webes alkalmazás, mivel adminisztrátori jogokkal rendelkező külső felhasználókat egyelőre nem tervezünk. Az EJB használata mellett nem kell a két különböző alkalmazásban a Hibernate konfigurációjával bajlódnunk, elegendő egyszerűen megadni az alkalmazás számára szükséges enterprise bean-ek elérhetőségét.

A továbbiakban minden egyéb lehetőségtől eltekintve az EJB-t kizárólag abból a szempontból vizsgálom, hogy hogyan használható a perzisztenciakezelés megoldására.

4.1. Entity bean-ek létrehozása

Egy Entity bean az EJB 3.0-ás verziójától kezdve egy egyszerű JavaBean. Korábban ezek is enterprise bean-ek voltak, akár csak a későbbiekben részletezett Session beanek. Segítségükkel képesek lehetünk a valós világ objektumainak reprezentálására és azok perzisztens tárolására. Például a jelenlegi levélküldő alkalmazás kifejlesztésénél ilyen módon reprezentált objektumok a felhasználó, vagy például a felhasználó jogai. Az EJB ezeket az objektumokat le tudja képezni a megfelelő adatbázis táblákba, így az alkalmazáson belül nem kell foglalkoznunk azzal, hogy

ezen objektumok hogyan vannak leképezve a relációs modellben, a fejlesztés során mindvégig objektum orientált módon gondolkozhatunk.

Példaként először nézzünk meg egy egyszerű esetet.

4.1.1. forráskód. *Felhasznalo.java*

```
import javax.persistence.* ;
@Entity
@Table(name="FELHASZNALO")
public class Felhasznalo {
   private int id;
   private String account;
   private String password;

   @Id
   @Column(name="USER_ID")
   public int getId( ) { return id; }

   public void setId(int pk) { this.id = pk; }

   @Column(name="ACCOUNT")
   public String getAccount( ) { return account; }

   public void setAccount(String account) {
        this.account = account;
   }

   @Column(name="PASSWORD")
   public int getPassword( ) { return password; }

   public void setPassword(String password) {
        this.password = password;
   }
}
```

Az így megadott `Felhasznalo` osztály objektumai a következő adatbázis táblában fognak leképződni:

FELHASZNALO	
ID	INT PRIMARY KEY
ACCOUNT	VARCHAR
PASSWORD	VARCHAR

Az EJB konténer számára különböző annotációkkal jelezzük, hogy hogyan szeretnénk leképezni az objektumainkat. A `@Entity` annotáció jelzi, hogy a megadott osztály egy Entity bean és ezen osztály objektumait objektum relációs megfeleltetéssel (object to relational mapping (ORM)) egy relációs adatbázisba szeretnénk leképezni. A `@Table` és `@Column` annotációk

nem kötelezőek, segítségükkel a tábla neve és az oszlopok nevei adhatóak meg, hiányukban a konténer kikövetkezteti ezeket. Fontosabb még az elsődleges kulcs megadása, amelyet a `@Id` annotációval jelölhetünk ki.

Természetesen az így létrehozott táblák nem egymástól teljesen függetlenül léteznek, hiszen számos eset van amikor a táblák között valamilyen kapcsolatot kell biztosítani. Az EJB hétféle kapcsolatot tud kezelni.

Egyirányú egy-egy kapcsolat

Jó példa lehet erre a kapcsolattípusra a felhasználókat és a címeket tároló táblák közötti kapcsolat. Minden felhasználóhoz egy cím tartozik és minden címhez is egy felhasználó fog tartozni ebben a kapcsolati modellben, valamint a címeket tároló táblából nem lehet visszakeresni, hogy az adott címhez melyik felhasználó tartozik. Ennek megfelelően csak a Felhasznalo bean-ben kell jelezni a kapcsolatot:

4.1.2. forráskód. *Felhasznalo.java*

```
package com.titan.domain;
@Entity
public class Felhasznalo implements java.io.Serializable {
    ...
    private Cim cim;
    ...

    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn (name="CIM_ID")
    public Cim getCim( ) {
        return cim;
    }

    public void setCim(Cim cim) {
        this.cim = cim;
    }
}
```

A forráskód részlet alapján látható, hogy a kapcsolat megadásához a `@OneToOne` annotációt kell használni, a `@JoinColumn` annotációval pedig a külső kulcsot adhatjuk meg, amely alapján a felhasználóhoz tartozó címinformációk lekérdezhetőek.

Kétirányú egy-egy kapcsolat

Kétirányú kapcsolat esetén azt szeretnénk elérni, hogy a kapcsolatban álló táblák bármelyikéből kiindulva elérhetők legyenek a másik táblában tárolt adatok. Például, ha egy szolgáltatást a felhasználók csak fizetés ellenében vehetnek igénybe, akkor szükség lehet egy olyan táblára, ahol bankkártya adataikat tároljuk. Ilyen esetben jól jöhet, hogy a felhasználó kilétét a bankkártya

száma alapján is megtudjuk határozni és fordítva. A kapcsolatokat mindkét bean-ben jelezni kell.

4.1.3. forráskód. *BankKartya.java*

```
@Entity
public class BankKartya implements java.io.Serializable {
    private int id;
    private Date ervenyessegiIdo;
    private String szam;
    private String nev;
    private String szervezet;
    private Felhasznalo felhasznalo;

    ...

    @OneToOne(mappedBy="bankKartya")
    public Felhasznalo getFelhasznalo( ) {
        return this.felhasznalo;
    }

    public void setFelhasznalo(Felhasznalo felhasznalo) {
        this.felhasznalo = felhasznalo;
    }

    ...
}
```

Ami újdonság az egyirányú kapcsolat leírásához képest az az, hogy a `@OneToOne` annotációnak a `mappedBy` attribútumát is beállítottuk. Az itt megadott érték jelzi, hogy a `Felhasznalo` bean `bankKartya` tulajdonsága alapján kétirányú kapcsolatot szeretnénk a két bean között. Ennek megfelelően a `Felhasznalo` bean a következőképpen fog kinézni:

4.1.4. forráskód. *Felhasznalo.java*

```
@Entity
public class Felhasznalo implements java.io.Serializable {
    private BankKartya bankKartya;
    ...
    @OneToOne (cascade={CascadeType.ALL})
    @JoinColumn(name="BANK_KARTYA_ID")
    public BankKartya getBankKartya( ) {
        return bankKartya;
    }

    public void setBankKartya(BankKartya bankKartya) {
```

```
        this.bankKartya = bankKartya;
    }
    ...
}
```

Egyirányú egy-sok kapcsolat

A levélküldő alkalmazásból merítve egy példát, azt láthatjuk, hogy egy felhasználó rengeteg email címre küldhet levelet és az tulajdonképpen nem érdekes, hogy az emailcímeket tároló táblából is meg tudjuk határozni, hogy az adott cím melyik felhasználóhoz tartozik, ekkor tehát egyirányú egy-sok kapcsolatot kell létesítenünk a `Felhasznalo` és az `EmailCim` beanek között.

4.1.5. forráskód. *Felhasznalo.java*

```
@Entity
public class Felhasznalo implements java.io.Serializable {
    ...
    private List<EmailCim> emailCimek = new ArrayList<EmailCim>( );
    ...
    @OneToMany(cascade={CascadeType.ALL})
    public List<EmailCim> getEmailCimek( ) {
        return emailCimek;
    }

    public void setEmailCimek(List<EmailCim> emailCimek) {
        this.emailCimek = emailCimek;
    }
}
```

A `Felhasznalo` bean-ből tehát az `EmailCim` bean-ek egy listája érhető el, ezzel és a `@OneToMany` annotációval jeleztük, hogy a kapcsolat egy-sok jellegű. Az `EmailCim` bean-ben nem kell semmilyen módon jeleznünk a kapcsolatot, így az egyirányú lesz.

Egyirányú sok-egy kapcsolat

Egy képzeletbeli utazási iroda esetében előfordulhat, hogy egy busszal sok utat szerveznek, de a busz adataiból nézve nem érdekes, hogy milyen utakon vett részt, az azonban fontos, hogy az utazás melyik busszal volt lebonyolítva, így ebben az esetben egy egyirányú sok-egy kapcsolatra lesz szükségünk a modellezéshez.

4.1.6. forráskód. *Utazas.java*

```
@Entity
public class Utazas implements java.io.Serializable {
    private int id;
    private String nev;
    private Busz busz;

    public Utazas( ) {}
    public Utazas(String nev, Busz busz) {
        this.nev = nev;
        this.busz = busz;
    }

    @Id @GeneratedValue
    public int getId( ) { return id; }

    public void setId(int id) { this.id = id; }

    public String getNev( ) { return nev; }
    public void setNev(String nev) { this.nev = nev; }

    @ManyToOne
    @JoinColumn (name="BUSZ_ID")
    public Ship getBusz( ) { return busz; }

    public void setShip(Busz busz) { this.busz = busz; }
}
```

Ebben az esetben a sok-egy kapcsolatot a `@ManyToOne` annotációval lehet megadni, míg a `Busz` entity-ben semmilyen módon nem kell jeleznünk a kapcsolatot.

Kétirányú egy-sok, vagy sok-egy kapcsolat

Az utazási irodás példán továbbhaladva, ki tudunk találni olyan esetet, amikor egy-sok kapcsolatra van szükség, de azt szeretnénk, ha ez kétirányú lenne. Ilyen eset lehet, ha a helyfoglalásokat szeretnénk nyilvántartani, amikor is egy utazásra több helyfoglalás is érkezik és nyílván való módon a helyfoglalás adataiból szeretnénk látni, hogy melyik utazásra szól, illetve az utazás adataiból is lekérdezhetőnek kell lennie, hogy mely foglalások szólnak rá. Így tulajdonképpen nézőpont kérdése, hogy a kapcsolatunk egy-sok, vagy sok-egy, ezért kétirányú esetben nem is különböztetjük meg a kettőt.

4.1.7. forráskód. *Utazas.java*

```
@Entity
public class Utazas implements java.io.Serializable {
    ...
    private List<Helyfoglalas> helyfoglalasok = new ArrayList<Helyfoglalas>( );
    ...

    @OneToMany(mappedBy="utazas")
    public List<Helyfoglalas> getHelyfoglalasok( ) { return helyfoglalasok; }

    public void setHelyfoglalasok(List<Helyfoglalas> helyfoglalasok) {
        this.helyfoglalasok = helyfoglalasok;
    }
}
```

Az utazás szempontjából nézve egy-sok kapcsolatunk van ennek megfelelően `@OneToMany` annotációt használunk.

4.1.8. forráskód. *Helyfoglalas.java*

```
@Entity
public class Helyfoglalas implements java.io.Serializable {
    ...
    private Utazas utazas;
    ...

    @ManyToOne
    @JoinColumn(name="UTAZAS_ID")
    public Utazas getUtazas( ) { return utazas; }

    public void setCruise(Utazas utazas) { this.utazas = utazas; }
}
```

Az helyfoglalás szemszögéből nézve pedig sok-egy kapcsolatról van szó. Azáltal, hogy mindkét entity-ben megadtuk a kapcsolatot, a kapcsolat kétirányú lesz.

Kétirányú sok-sok kapcsolat

Sok-sok kapcsolat esetében már mindenképpen szükség van kapcsolótáblára a relációs modellben, míg az objektum orientált modellben ez szükségtelen, ennek megfelelően entity bean-t sem kell létrehoznunk a kapcsolótábla reprezentálására, az EJB gondoskodik annak relációs modellbeli létrehozásáról.

A következő példában felhasználókat és számítógépeket tároló táblákat szeretnénk összekötni. Mivel egy falhasználónak több számítógéphez is lehet hozzáférése és egy számítógéphez is több felhasználó férhet hozzá egy időben, a táblák közti kapcsolat mindenképpen sok-sok lesz.

4.1.9. forráskód. *Felhasznalo.java, Szamitogep.java*

```
public class Felhasznalo implements java.io.Serializable {
    ...
    @ManyToMany
    @JoinTable(name="HOZZAFERES",
    joinColumns={@JoinColumn(name="FELHASZNALO_ID")},
    inverseJoinColumns={@JoinColumn(name="SZAMITOGEP_ID")})
    public List<Szamitogep> getSzamitogepek( ) { return szamitogepek; }

    public void setSzamitogepek(List<Szamitog> szamitogepek) {
        this.szamitogepek = szamitogepek;
    }
    ...
}

@Entity
public class Szamitogep implements java.io.Serializable {
    ...
    private List<Felhasznalo> felhasznalok = new ArrayList<Felhasznalo>( );
    ...

    @ManyToMany(mappedBy="szamitogepek")
    public List<Felhasznalo> getFelhasznalok( ) {
        return felhasznalok;
    }

    public void setFelhasznalok(List<Felhasznalo> felhasznalok) {
        this.felhasznalok = felhasznalok;
    }
    ...
}
```

A kapcsolat megadásához a `@ManyToMany` annotációt kell használni és az egyik entity-ben meg kell adni a kapcsolótábla adatait is ezt a `@JoinTable` annotáció segítségével tehetjük meg.

Egyirányú sok-sok kapcsolat

Ha a kapcsolat egyirányú, akkor csak az egyik entity-ben kell jelezni a kapcsolatot a `@ManyToMany` annotációval és ebben az entityben kell megadni a kapcsolótáblát is. Az adatbázisban a táblák szerkezete ugyanaz lesz mint kétirányú esetben, így az egyirányúság csak az objektum orientált modellben lesz értelmezhető.

4.2. Entity bean-ek menedzselése

Az eddigiekben nagyon rövid ízelítőt kaphattunk az entity bean-ek létrehozásáról, de mindez idáig semmit sem mondtam, hogy végül hogyan használhatjuk őket. A használatukhoz mindenképpen szükség lesz egy enterprise bean-re, amelyen keresztül elérhetjük ezeket az entity-ket.

Az EJB többféle enterprise bean típust is ismer, ennek ellenére most csak egyetlen egyel a Statless Session bean-nel ismerkedünk meg.

A Statless Session bean-ek amint a nevük is mutatja állapotmentesek, ami azt jelenti, hogy a bean bármely metódusát is hívjuk meg annak lefutása nem fog függni attól, hogy korábban ezt a metódust, vagy egy másikat meghívtunk e már, vagyis semmilyen állapotot nem tárolunk a bean-ben. Egy Session bean-nek természetesen semmi köze sincs ahhoz, hogy enity beaneket akarunk e kezelni vele vagy sem, bármilyen más üzleti logika lekódolható benne, azonban az esetek túlnyomó részében ez a fajta bean megfelelő választás lesz az entity beanek menedzselésére.

Egy Statless Session bean létrehozásához elegendő egy osztályt a `@Statless` annotációval ellátni illetve egy, vagy két interfészt megadni hozzá. Az interfészek megadásával biztosítjuk a Session bean szolgáltatásainak elérését és azért kell esetenként két interfészt is megadnunk, mert külön vezérelhetjük, hogy a bean mely metódusai érhetőek el lokálisan és melyek távoli kapcsolaton keresztül. Miután egy enterprise bean-t telepítettünk, az alkalmazásszerver JNDI bejegyzést készít arról, hogy hogyan érhetőek el. Ha mindkét interfészt megadtuk akkor ugyanahhoz a bean-hez két bejegyzés is készül, így értelemszerűen a megfelelőt kell választani használatukkor.

Áttérve az entity bean-ek menedzselésének kérdésére, egy Session bean-en kívül egy `EntityManager` objektumra is szükségünk lesz. Az `EntityManager` interfész a `javax.persistence` csomag része akárcsak az entity bean-eknél használt annotációk. Gyakorlatilag minden szükséges műveletet biztosít, amire csak szükségünk lehet az entity bean-ek perzisztens módon való kezeléséhez. Első feladatunk tehát, hogy referenciát szerezzünk egy ilyen objektumra, ehhez használhatjuk az `EntityManagerFactory` osztályt, vagy ennél kényelmesebb mód, ha a `@PersistenceContext` annotációval megkérjük az alkalmazásszervert, hogy injektálja azt be nekünk, a használni kívánt enterprise bean-be.

Az `EntityManager` által biztosított műveletek a következők:

4.2.1. forráskód. *EntityManager.java*

```
package javax.persistence;
public interface EntityManager {
    public void persist(Object entity);
    public <T> T find(Class <T> entityClass, Object primaryKey);
    public <T> T getReference(Class <T> entityClass, Object primaryKey);
    public <T> T merge(T entity);
    public void remove(Object entity);
    public void lock(Object entity, LockModeType lockMode);
    public void refresh(Object entity);
    public boolean contains(Object entity);
    public void clear( );
    public void joinTransaction( );
    public void flush( );
    public FlushModeType getFlushMode( );
    public void setFlushMode(FlushModeType type);
```

```
    public Query createQuery(String queryString);
    public Query createNamedQuery(String name);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString, String resultSetMapping);
    public Query createNativeQuery(String sqlString, Class resultClass);
    public Object getDelegate( );
    public void close( );
    public boolean isOpen( );
}
```

Ezek közül csak néhány leggyakrabban használt metódust emelnék ki. Ilyen például a `find(Class<T>entityClass,ObjectprimaryKey)` metódus, amely segítségével az elsődleges kulcs alapján a megfelelő tábla egy sorát kérdezhetjük le, azaz egy objektumot kapunk eredményül. A `persist(Objectentity)` metódus segítségével gyakorlatilag egy `INSERT` utasítást tudunk végrehajtani, vagyis hatására az `EntityManager` perzisztálja az entity bean-t. A `remove(Objectentity)` metódus törli az adatbázisból az entity-ben tárolt adatokat, vagyis a tábla egy sorát. A `merge(Tentity)` metódus segítségével módosíthatjuk a már letárolt adatokat. Különböző bonyolult lekérdezések futtatására is van lehetőség ezeket a `createQuery`, a `createNamedQuery`, illetve a `createNativeQuery` metódusok segítségével tehetjük meg.

Természetesen lehetőségünk van tranzakciókezelésre. Ennek legegyszerűbb megvalósítására használhatjuk a `UserTransaction` osztályt, amelynek egy példányát a `SessionContext`-en keresztül szerezhetjük meg. A tranzakcióban kezelendő műveleteket a `UserTransaction` `begin` és `comit` metódusai között kell elhelyeznünk.

4.2.2. forráskód. *HypotheticalBean.java*

```
@Stateless
@TransactionManagement(TransactionManagerType.BEAN)
public class HypotheticalBean implements HypotheticalLocal {
    @Resource SessionContext ejbContext;

    public void someMethod( ) {
        try {
            UserTransaction ut = ejbContext.getUserTransaction( );
            ut.begin( );
            // Ide kerülnek a tranzakcióban végzendő műveletek.
            ut.commit( );
        } catch(IllegalStateException ise) {...}
        catch(SystemException se) {...}
        catch(TransactionRolledbackException tre) {...}
        catch(HeuristicRollbackException hre) {...}
        catch(HeuristicMixedException hme) {...}
        ....
    }
}
```

Természetesen az EJB-vel megvalósított perzisztencia kezelés nem csupán annyiból áll, mint amit e rövid fejezetben leírtam, sőt a kihagyott lehetőségek száma annyira nagy, hogy túlságosan is hosszan lehetne azokat sorolni, nem beszélve az EJB egyéb lehetőségeiről, amelyek nem is kapcsolódnak a perzisztenciakezelés témaköréhez. Abban azonban bízom, hogy az itt leírtak elegendőek ahhoz, hogy belássuk, ha az alkalmazásunk már elér egy bizonyos komplexitást, akkor érdemes elgondolkodnunk ennek a technológiának a használatán.

5. fejezet

Az NMAIL levélküldő alkalmazás rövid bemutatása

Miután minden alkalmazott technológiát bemutattam, amely szükséges volt a webalkalmazás elkészítéséhez, most nézzük meg részletesebben, hogy tulajdonképpen milyen szolgáltatásokat is kellet megvalósítani és ezek hogyan lettek kivitelezve.

Amint a bevezetőből már kiderült az NMAIL levélküldő egy mini alkalmazás, amely a felhasználók számára lehetővé teszi, hogy nagy mennyiségű hírlevelet küldjenek ki azon ügyfeleik számára, akik igényt tartanak erre a tájékoztatási formára.

Az alkalmazás által nyújtott szolgáltatásokat csak regisztrált felhasználók vehetik igénybe, így a kezdőoldal tulajdonképpen egy bejelentkezési felület. A bejelentkezési információkat HTTPS protokollon továbbítjuk a szerver felé. A beérkező adatok feldolgozásáról a `LoginController`-nek kell gondoskodnia, de még mielőtt bármilyen további feldolgozási műveletre sor kerülne a kontrollerhez tartozó `LoginValidator` kapja meg a kérést, ahol az űrlapmezőkben megadott adatok egy `LoginData` objektumba csomagolva jelennek meg. A `LoginValidator` ellenőrzi, hogy a bejelentkezési űrlapmezők ki lettek e töltve (vagyis az adott `LoginData` objektumnak a tulajdonságai rendelkeznek e értékkel), illetve, hogy a megadott adatok bejelentkezési információkhoz tartozik e regisztrált felhasználó és aktív státuszban van-e az előfizetése. Amennyiben bármely feltételen megbukik a validálási folyamat, úgy a validátor a `LoginData` objektumhoz, illetve annak kifogásolt tulajdonságaihoz hozzákapcsolja a megfelelő hibakódo(ka)t és visszaküldi ezt az objektumot a bejelentkezési képernyőre, ahol a hibakód(ok)hoz tartozó hibaüzenetek megjelenítésre kerülnek.

Amennyiben a `LoginValidator` nem talál semmilyen hibát, és a jogosultságok is rendben vannak, a vezérlés a `LoginController onSubmit` metódusához kerül, ahol a külön service objektumok töltik be a felhasználóhoz rendelt POP3 fiókból a legutolsó 5 levelet, illetve a felhasználóhoz tartozó egyéb megszorításokat. A levelek egy `MailsData` objektumba kerülnek és ezt az objektum kerül megjelenítésre az alkalmazás levélküldő képernyőjén. A különböző megszorításokat tartalmazó `RestrictionBean` objektum meneti hatókörbe kerül, és a benne tárolt szabályok befolyásolják, hogy milyen további műveleteket hajthat végre a felhasználó.

5.1. ábra. Bejelentkezési képernyő

Levélküldő modul

A levélküldő felület két valamelyest elkülönülő részre van osztva. A felső részben található űrlapmezőkkel módosíthatjuk az elküldendő levél bizonyos fejlécadatait (feladó neve, címe, tárgy), valamint tesztelés céljából teszt címeket adhatunk meg vesszővel elválasztva.

5.2. ábra. Levélküldési képernyő

Az alsó részben egy táblázatban jelenítjük meg a postaládában lévő utolsó legfeljebb 5 üzenetet, ezek közül rádiógombokkal választhatjuk ki a továbbküldendőt. A levelek tartalmi módosítására egyelőre nem adunk lehetőséget, bár a későbbiekben, nem zárkózunk el egy ilyen lehetőség bevezetésétől sem, amennyiben a szolgáltatás igénybevevői részéről erre igény mutatkozik.

A leveket tartalmazó táblázat alatt kapott helyet a *Levélküldés* illetve a *Teszt üzenet küldés* gomb. Teszt üzenetek küldése bármikor lehetséges, így ez a gomb mindig aktív, viszont ilyen üzeneteket csak a fentebb megadható teszt címekre lehetséges küldeni. Az éles üzenetek küldésének gyakorisága minden felhasználó számára külön megadható időkorláttal van szabályozva, ezzel megpróbálunk gátat szabni a túl gyakori levélküldésnek, mivel egy felhasználó több tízezer emailcímre postázhat ki leveleket, így ha naponta többször is megismételné ezt a műveletet a jelenlegi szervereink valószínűleg nem sokáig bírnák a terhelést. A művelet elvégzése után a felületen letiltott állapotba kerül a *Levélküldés* gombm, és a szerver a továbbiakban minden nem teszt célú üzenetküldést megtagad, mindaddig, amíg az időkorlát le nem telik, illetve az előző küldési folyamat be nem fejeződik.

Technikailag bármely levélküldési folyamatot is választjuk az űrlap adatai ugyanazon `MailsData` objektumba kerülnek, amellyel a rendelkezésre álló levelek adatait is megjelenítettük, és ez az objektum kerül továbbításra a `SendMessageController`-hez amely a megfelelő levélküldő servicehez továbbítja. A levélküldő service előállítja a postázandó üzenetet a `MailsData` objektumban tárolt adatok alapján és minden egyes címre egyenként küldi el, így a címzettek nem szerezhetnek tudomást arról, hogy rajtuk kívül kik kapták meg ugyanazt az üzenetet.

5.3. ábra. Információs oldal

5.4. ábra. Beállítások, importálás

Információs modul

Amennyiben nem teszt üzenetek küldéséről van szó, úgy a levélküldési folyamat nagyon hosszú időt is igénybe vehet. A szerver terhelésének csökkentése érdekében a levélküldő szál a leveleket blokkokban továbbítja, amely blokkok mérete külön felhasználónként szabályozható, és ezen blokkok között szintén külön megadható várakozási idő is be van állítva. Tekintettel tehát a hosszú küldési folyamatra egy információs oldalt is létrehoztam az alkalmazásban, ahol nyomon követhető a folyamat állapota. A levélküldő szál, bizonyos időnként bejegyzést készít egy adatbázis táblába a folyamat aktuális állapotáról és ezeket az állapotinformációkat jelenítem meg az információs oldalon Ajax segítségével, vagyis egy JavaScript függvény HTTP GET kéréseket intéz a szerver felé, amelyet a `ProcesController` dolgoz fel. A `ProcesController` egy service objektum segítségével lekérdezi a folyamatokat tároló táblából a legutolsó 10 folyamat állapotát és `EmailProcesData` objektumok listájaként küldi vissza ezeket az információkat az információs felületre. A felületen a még be nem fejezett folyamatok aktuális állapotáról progressbar-ok segítségével is tájékoztatást adunk. A progressbar megjelenítéséhez szintén Ajaxos technológiát használtam. Ha már így szóba került ez a technológia, talán érdemes pár szót vesztegetni rá, annál is inkább, mivel az utóbbi időben egyre népszerűbb, illetve az eddigi leírásokból teljesen kihagytam.

Az Ajax olyan webes technológia, amely több más technológia együttes használatából épül föl. A kifejezés első használója Jesse James Garrett volt, aki előadásaiban leszögezte, hogy ez nem egy betűszó, ennek ellenére a szakirodalomban az Asynchronous JavaScript and XML rövidítéseként használják. Ennek oka, az hogy ez a kifejezés jól lefedi azt, hogy az Ajax mire is használható, vagyis JavaScript függvények segítségével aszinkron módon kérhetünk le a szer-

vertől információkat, amelyeket az éppen megjelenített weboldalba ágyazhatunk DOM segtségével anélkül, hogy a weboldalt frissítenünk kellene. A szervertől lekért információkat XML és plain text formátumban is megkaphatjuk. Mivel a megjelenített HTML fájl újratöltése nélkül frissíthető annak tartalma, kiváló lehetőséget nyújt olyan animációk elkészítéséhez, mint például a jelen programban is használt progressbar.

Tovább folytatva az információs oldal ismertetését, ezen az oldalon egyéb információk is láthatóak az adatbázisban letárolt emailcímekről. Minden levélküldési folyamat során megfigyeljük, hogy mely címekre nem volt sikeres a levélküldés, és mely címek voltak duplán letárolva. Ezekről a címekről egyelőre csak számadatokat közlünk.

Beállítások

Az alkalmazáshoz tartozó beállításokat nem szívesen bíztuk a felhasználókra, így a legtöbb megrendelő számára nem vagy csak korlátozottan érhetők el ezek a funkciók. Ezen menüpont alatt jelenleg csak a levelező kiszolgáló POP3 elérési adatai adhatók meg, illetve email címek adatbázisba való importálása érhető el. Email címek importálását, jelenleg csak csv fájlból tudja a program megoldani. Ezen funkciók mindegyike külön külön, vagy egyben is letiltható.

Összefoglalás

A bevezetőben azt a célt tűztem magam elé, hogy olyan keretrendszert találjak, amely segítségével könnyen elkészíthető egy egyszerű webalkalmazás és amelyre a későbbiekben is támaszkodhatom, ha nagyobb, komplexebb alkalmazások kifejlesztésében kell részt vennem. Ahhoz, hogy nagyméretű, bonyolult felépítésű szoftverek megírását is eltudjuk végezni mindenképpen szükséges, hogy a lehető leghatékonyabb módon szervezzük a kódunkat, hogy a kód növekedésével az áttekinthetőség ne romoljon, és a karbantartás ne váljék túlságosan is nehézkessé. Kezdő fejlesztők számára azonban pontosan az okozza a legnagyobb nehézséget, hogy nincsenek tapasztalataik abban, hogyan is érdemes egy programot felépíteni. Ebben segíthet a szakirodalomban fellelhető tervezési minták elsajátítása és azok az eszközök, amelyek ezen tervezési minták mentén segítik a fejlesztést. Néhány különböző keretrendszer szemügyrevétele után végül a Spring Framework bizonyult a legjobb választásnak a számomra. A Spring pontosan azt nyújtja, ami a leginkább szükséges egy kezdő fejlesztő számára, vagyis nem engedi meg, hogy kód a nem megfelelő szervezés miatt karbantarthatatlanná váljon, miközben szinte észrevehetetlen módon a fejlesztőbe nevel egyfajta igényt jó struktúrájú programok létrehozására.

Mindenképpen érdemes azonban megjegyezni, hogy a Spring sem csodafegyver, bizonyos helyzetekben gátló tényező lehet, ha egy keretrendszer mindent ennyire szigorúan előír, ezért a használni kívánt eszközök kiválasztásakor meg kell fontolni, hogy valóban a kiszemelt eszköz segítené a legjobban a munkát. Pontosan ilyen megfontolások miatt a perzisztenciakezelés megvalósításához az EJB-t választottam és nem a Spring által nyújtott lehetőségeket. Tapasztalatom szerint egy projekt befejezése után a legtöbb fejlesztő úgy érzi, hogy bizonyos dolgokat másként kellett volna megcsinálni, talán más eszközöket kellett volna használni. Most sincs ez másként, úgy érzem a megjelenéshez nem a JSP-t kellett volna használnom, hanem a JSF-et.

Úgy gondolom, hogy az információs oldal kialakításánál, ahol Ajaxot is használnom kellett lényegesen egyszerűbb lett volna a dolgom, mivel a JSF-hez meglehetősen jó Ajaxos tagkönyvtárak érhetőek el. Egy következő webes projektben, ha lesz rá lehetőségem mindenképpen törekedni fogok a használatára.

Irodalomjegyzék

[1] Jason Hunter, *Java Szervletek Programozása*, 2002

[2] Hans Bergsten, *JavaServer Pages*, 2000

[3] David Geary, Cay Horstmann, *Core JavaServer Faces*, 2007

[4] Craig Walls, Ryan Breidenbach, *Spring in Action*, 2008

[5] Nyékyné Gaizler Judit (szerk.), *J2EE útikalauz Java programozóknak*, 2002

[6] Bócz Péter, Szász Péter, *A világháló lehetőségei*, 2000

[7] Chris Bates, *XML elmélet és gyakorlat*, 2004

[8] *The JavaServer Faces Technology Tutorial*, 2003

[9] Bruce W. Perry, *Java Servlet & JSP Cookbook*, 2004

[10] Dave Crane, Bear Bebeault, Jord Sonneveld, *Ajax in Practice*, 2007

[11] `http://www.martinfowler.com/articles/injection.html`

[12] `http://jakarta.apache.org/commons/attributes`