SZAKDOLGOZAT

CSEHI MIKLÓS

DEBRECEN 2009.

DEBRECENI EGYETEM

INFORMATIKA KAR

Témavezető:

Dr Adamkó Attila

egyetemi tanársegéd

Készítette:

Csehi Miklós

programtervező informatikus MsC

EJB3 ENTITÁSOK JAVASERVER FACES KERETRENDSZERBEN

Tartalomjegyzék

A dolgozat célja

Kevés olyan foglalkozás maradt az utóbbi években, amelyre a munkaerőpiac folyamatos kereslete lenne jellemző. Ezen kevesek egyike az egyetemi diplomás informatikus, melynek mesterszaki képzése 2007/8-as tanévben kezdődött az egész országban, így a Debreceni Egyetemen is. A képzés az informatika területeinek szerteágazó tudományaiba engedett bepillantást, hiszen maga az informatika is egy sokrétű tudomány. Engem ezek közül az irányvonalak közül mindig is a szoftverek fejlesztése vonzott leginkább. Az egyetemen ismerkedtem meg a Java nyelvvel és itt hallottam először olyan korszerű technikákról, amiket az informatikai cégek mindennapos fejlesztéseikhez használnak. Sajnos gyakorlatban ezek kipróbálására nem volt lehetőség, így az elsajátításuk önálló feladat maradt. Ezen dolgozatban a JavaServer Faces és az Enterprise Java Beans 3 entitások megismerésének tapasztalatait írom le. Célom az volt, hogy ezeket a korszerű technológiákat elsajátítva informatikai tudásom naprakészebb legyen.

Bevezetés

Az 1990-es évektől a számítógépes hálózatok technológiájának fejlődése forradalmasította az emberek közötti kommunikációt. Az internet évről évre nagyobb szerepet kap az üzleti világban csakúgy, mint a tömegtájékoztatásban vagy civil életben. A web, amit tizenöt évvel ezelőtt csak kevés kiváltságos használhatott, ma már olyan természetes a fejlett világban, mint a meleg víz vagy az elektromos áram. Az első weboldalak még csak statikus információk megjelenítésére voltak alkalmasak. Később azonban megjelentek olyan technológiák, melyekkel dinamikus tartalom generálása is lehetővé vált. Ezzel a webböngészőkben olyan oldalak jelentek meg, melyek tudása már vetekedett egy desktop alkalmazásával, így joggal nevezték őket webalkalmazásoknak. A webes alkalmazások megvalósítását több programnyelv is támogatta. Ezek egyike a Java, mely a nyelv születése óta szorosan kapcsolódik az internethez – a webböngészőbe letöltődő és ott önállóan működő java applet úttörő volt megjelenésekor. A java azóta is folyamatosan fejlődik, az appletek jelentősége is háttérbe szorult, ugyanakkor megjelentek olyan szerveroldali megoldások,

amiben a java szintén élen jár. A webes szolgáltatásokat támogató java technológiák összessége a Java Web Services. Ennek részei a

Java Servlet	**JavaServer Pages Standard Tag Library**
JavaServer Pages (JSP)	**XML technológiák (JAXB, JAXP, JAXR, JAX-RPC)**
JavaServer Faces (JSF)	**SOAP with Attachments API for Java (SAAJ)**

A szervlet technológia népszerűségét többek között objektumorientált szemléletének többszálúságának és könnyen skálázhatóságának köszönheti. Ugyanakkor a weben nagy szerepe van a felhasználónak megjelenítendő látványnak, amit a programtól célszerű elkülöníteni, erre fejlesztették ki a HTML alapú, mégis dinamikus JSP-t, mely futási idejű karakterisztikájában ugyanarra képes, mint a szervlet. A kód mennyiségének csökkentését elősegítette a Unified Expression Language. A Java Web Services nagyban épít már meglévő java technológiákra, mint például a JavaBeans-re, adatbáziskezelésben pedig az J2EE 5-ben megjelent entitásokra. A korábbi HTML és scriptlet kódokat vegyítő model1-es architektúrát felváltotta az MVC modellt webes környezetre illesztő model2. Ez nagyban megnövelte a kód áttekinthetőségét, ezáltal a fejlesztés hatékonyságát. Az alkalmazásfejlesztés ismétlődő feladatainak megoldására több népszerűbb és kevésbé népszerű keretrendszer is készült (Struts, Tapestry), mégis csak egy vált szabvánnyá a JavaServer Faces (JSF). Ebben a Sun a megjelent keretrendszerek legjobb tulajdonságát ötvözi és a Java Enterprise Edition 5-ben támogatását már kötelezővé tette. Dolgozatom egyik célja ennek a technológiának a bemutatása.

Elvárások webes keretrendszerekkel kapcsolatban

Egy korszerű webes keretrendszer elsődleges célja, hogy biztonságosan és kényelmesen kapcsolja össze a grafikus által tervezett felhasználói felületet az alatta futó üzleti logikával. A fejlesztés során több olyan probléma is felmerül, amelyekre egy korszerű webes keretrendszer előre kész megoldásokkal szolgál.

Konverzió: a nyílt végű adatbeviteli komponensek (HTML esetén tipikusan szövegmezők) input értékei minden esetben String típusúak, így ha azokkal numerikus műveleteket kívánunk végezni vagy adatbázismezők értékeinek akarjuk megfeleltetni akkor konverziót kell rajta végeznünk. Konverziókra olyan gyakran van szükség, hogy a keretrendszerünktől elvárhatjuk, hogy konvertereinket csak egyszer kelljen megírnunk és ezután azok újrafelhasználhatóak legyenek.

Validáció:A konvertált értéknek az üzleti logika által támasztott feltételeknek is meg kell feleljen. A validálandó adatokat öt csoportra oszthatjuk ellenőrzésük alapján:

- Nem ellenőrizendő adat
- Üzleti logika által korlátozott adat
- Fals adat kiszűrése
- Belső összefüggés által korlátozott adat
- Másik adat által korlátozott adat

  Ideális esetben a keretrendszer megoldást ajánl ezen adatok ellenőrzésére.

Oldalak közötti újrafelhasználhatóság: Egy weboldal gyakran több elemből tevődik össze s ezek közül több állandó elem is lehet. Ezeket az elemeket minden oldalon újraimplementálni nem lenne célszerű, egy kényelmes keretrendszer azonban megengedi, hogy az egyes elemekre hivatkozhassunk ezáltal felépíthessünk egy komplex oldalt. Ezt template jellegű újrafelhasználásnak hívják.

Komponensek támogatottsága: Egy keretrendszernek nem csak egy oldal vagy oldalelem újrafelhasználhatóságát kell biztosítania, az oldal egy elemeinek (komponenseinek) könnyű hivatkozhatóságát is. Ilyen komponensekből építhetjük majd fel a teljes oldalt.

Intelligens komponensek: Nem csak nézet, hanem viselkedés alapján is újrafelhasználhatóak a komponensek. Ezek az újnevezett ’data-aware’ komponensek képesek az input adataik konvertálására, validálására vagy akár adatbázisba mentésére is.

Navigáció: Egy felhasználói felület használatakor a user oldalak láncolatát járja be. Hasznos ha a keretrendszer az oldalak kapcsolódásának kialakítását valamilyen formában támogatja. Egyik oldalról a másikra való eljutás történhet deklaratív vagy programozott módon attól függően, hogy milyen feltételeknek kell teljesülniük az oldal megjelenítéséhez.

Nemzetköziesítés támogatása: Üzleti alkalmazásoknál gyakori követelmény, hogy a felület alkalmazkodni tudjon különböző felhasználók nyelvi és kulturális igényeihez. Ehhez nem elég csupán egy szöveget más-más nyelveken letárolni, majd később arra hivatkozni, különbözőek lehetnek a mértékegységek, dátumformátumok vagy akár színek jelentései is. Különböző országok adat validátorai eltérőek, míg egyes környezetekben bizonyos szövegmezőknek nincs értelmük.

Állapotmentés: A webes alkalmazások által használt HTTP protokoll önmagában állapotmentes. A felhasználó kényelmének érdekében azonban szükség van a komponensek állapotainak mentésére.

Fejlesztői szerepek szétválasztása: Egy összetett alkalmazás készítésekor az egyes fejlesztői szerepek különválnak, jellemzően üzletilogika-fejlesztő, komponensfejlesztő, weboldalfejlesztő és grafikusi szerepkörökre. A keretrendszernek is támogatnia kell, hogy az egyes feladatok a keretrendszerben se legyenek összemosva.

Szoftverbiztonság: A kész alkalmazással szemben követelmény, hogy ellenálló legyen a külső támadásokkal szemben. Ez általában az input és output adatok szűrését jelenti.

Más webes technológiák támogatása: együttműködés olyan elterjedt webes technológiákkal melyek kiegészítik a keretrendszert a fejlesztés során. Ilyen lehet például a CSS, a JavaScript vagy az AJAX.

Eszköztámogatottság: Támogassák a legelterjedtebb fejlesztői eszközök. A Java esetében az Eclipse, Netbeans vagy az IBM Rational Application Developer.

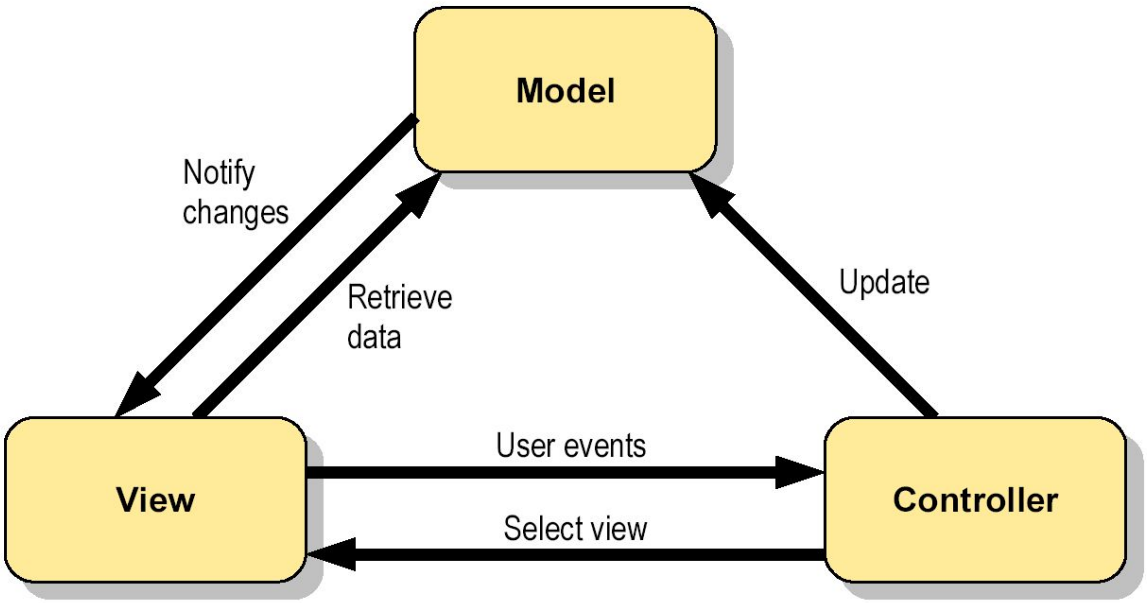
Bővíthetőség: A már meglévő keretrendszert új komponensekkel lehessen bővíteni. Ezeket a komponenseket a keretrendszert támogató gazdasági közösségek bocsátják a fejlesztők rendelkezésére.

A JavaServer Faces minden fentebb felsorolt követelménynek valamilyen szinten megfelelő komponensorientált webprogramozási keretrendszert specifikál. A specifikáció azt jelenti, hogy több implementáció is létezik a SUN referenciaimplementáción túl. Az 1.0 specifikációt (JSR-127) követte az 1.2-es (JSR-252) amely további hasznos eszközöket adott a fejlesztők kezébe.

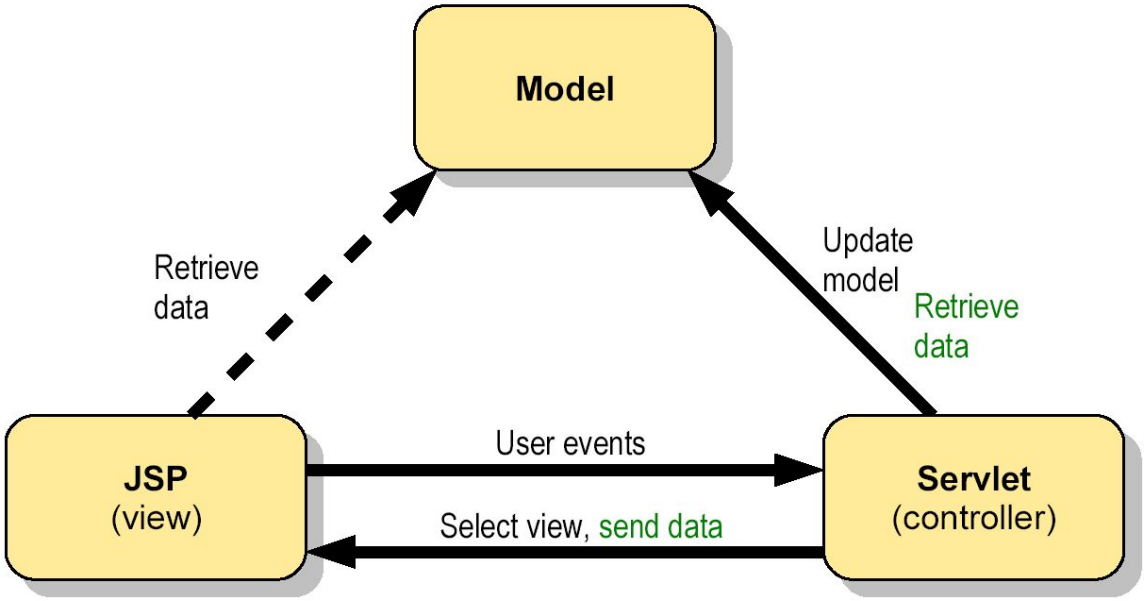
A Model View Controller architekturális minta megvalósítása JSF-ben

Az MVC minta egy webes alkalmazás mintája általános esetben. Elemei az esemény, a vezérlő, a nézetek és a modell. Modellen a valóság egy darabját leíró osztályokat értünk, a nézet pedig az a grafikus felhasználói felület, ami az adatokat rendezett formában megjeleníti. A kiszolgáló és felhasználó közötti párbeszédért a vezérlő a felelős. HTTP technológia esetén

a felhasználó böngészőjéből kérés érkezik a szerver felé, mely tartalmazza, hogy melyik oldalt milyen paraméterekkel kívánja megtekinteni. A kérést a vezérlő feldolgozza, majd legenerálja a modellt, vagy ha már létezik frissíti azt. A modell alapján a nézet összeállítja a felhasználónak küldendő felületet, majd továbbítja a felhasználó böngészője felé. A felhasználó a kapott adatok alapján új eseményt generálhat és a folyamat újraindul.



JSF technológia esetén a modell szerepét az entitásosztályok, a nézetét egy JSP oldal, a vezérlőét pedig egy szervlet végzi el.



A megfelelő mappában a szervletkonténer web.xml konfigurációs fájlját úgy kell módosítani, hogy minden olyan kérés, amely JSF specifikus elemeket használó JSP oldalra irányul a vezérlést a javax.faces.webapp package-ben található FacesServlet nevű szervletnek adja át. A FacesServlet ezután egy állapotgépet hoz létre, majd folytatja a feldolgozást. A modell és a vezérlő funkcióját az úgynevezett Managed JavaBeans –ek végzik el. Ezek a beanek nem Enterprise JavaBean-ek, menedzseltségüket a keretrendszer konfigurációs állományában regisztrálni kell, ezáltal a JSF képes lesz menedzselni ezen beanek életciklusát, vagyis létrehozni, és ha már nem kellenek megszüntetni őket. A menedzselt beaneknek két feladata van: a nézet felé modellként viselkednek - adatokkal látják el a nézeteket, a felhasználó által közölt adatokat pedig a feldolgozásuk után vissza kell írniuk a modellbe. Ezenkívül olyan publikus metódusokat nyújtanak a keretrendszer felé, melyek befolyásolják a rendszer viselkedését.

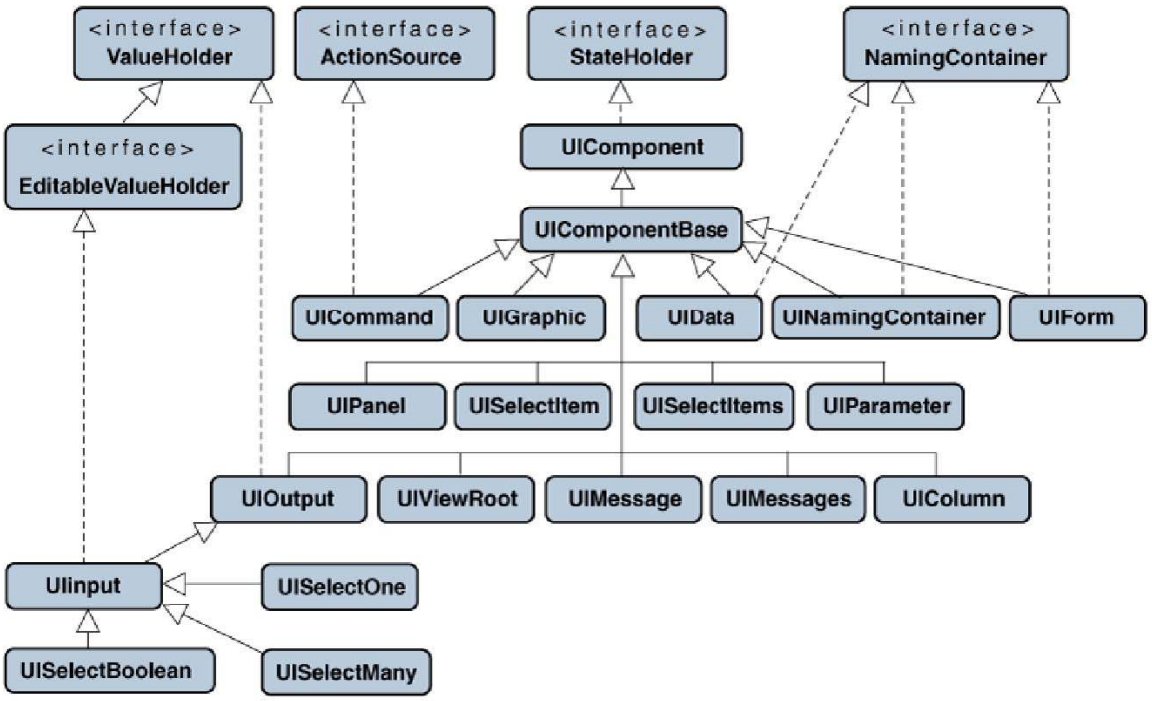
```
<servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <init-param>
            <param-name>javax.faces.LIFECYCLE_ID</param-name>
            <param-value>com.sun.faces.lifecycle.PARTIAL</param-value>
```

```
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
```

Nézet

A JSF tagokat használó JSP fájlokat a konténer szervletté fordítja. Fordítási szempontból JSF tagek megegyeznek a JSP tagekkel, azonban amikor lefut a JSP fájlhoz tartozó servlet(…) metódus, ezen tagek egy fastruktúrát építenek fel a memóriában. Ezt a struktúrát View-nak (nézetnek) hívják a JSF terminológiában. Gyökéreleme a ViewRoot, amelyből kiindulva bejárható a struktúra.

A JSF tagek hierarchiájában minden taghez tartozik egy objektum melynek típusa függ attól, hogy felette milyen JSF elem áll. Ezt a hierarchiát mutatja az ábra:



A JSF komponensek ősosztálya a UIComponent, amely a StateHolder interfészt implementálja. Ezáltal a keretrendszer képes a komponens a komponensek állapotának elmentésére és az adatok feldolgozása után helyreállításukra.

A View memóriabeli fastruktúrájának gyökéreleme egy UIViewRoot típusú objektum, ilyen objektumból annyi van ahány különböző nézet (különböző ViewId).

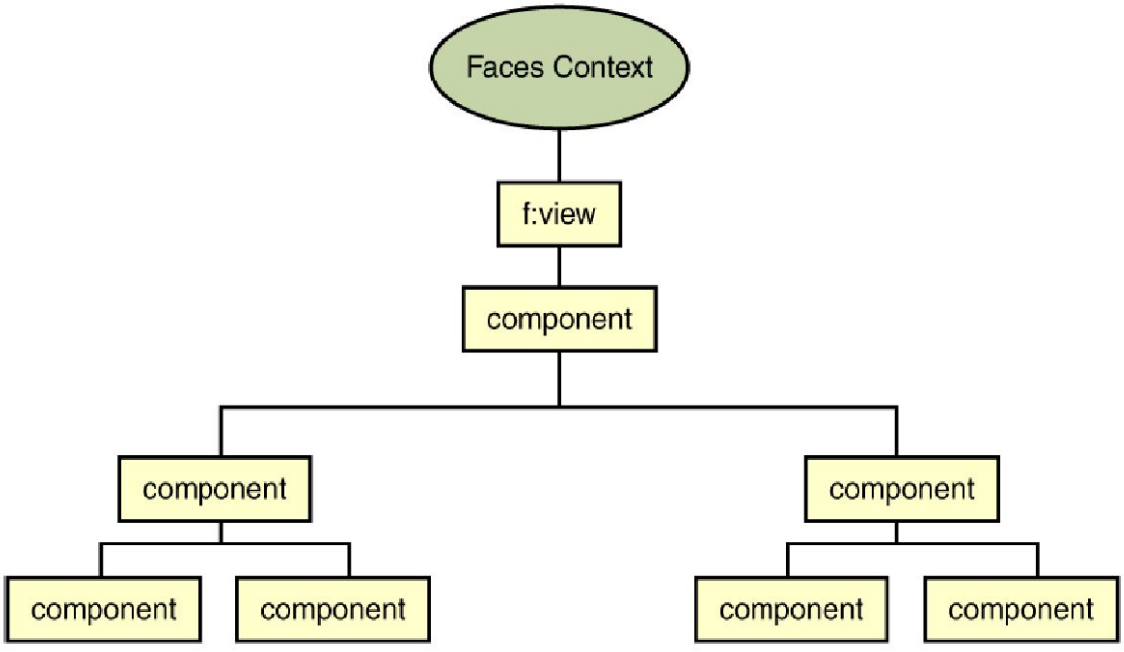
A kérés feldolgozásakor a vezérlő a UIForm submitted tagváltozóját ellenőrzi. Mivel egy nézetben több UIForm is lehet csak azt a formot dolgozza fel, ahol ezen logikai változó értéke igaz.

Az UICommand osztály komponensei valamilyen parancsot vagy reakciót váltanak ki. Ilyen lehet például egy submit input mező, egy link vagy egy javascript alapú komponens. Ha egy ilyen komponens aktiválódik a kérést delegáló logika meghívja az összes regisztrált ActionListenert interfészt implementáló osztályt.

A JSF kérés-feldolgozási ciklusában keletkezett üzenetek megjelenítésére használjuk az UIMessage osztályokat. Ilyen üzeneteket küldhet például egy érvénytelen adatot kapott validátor.

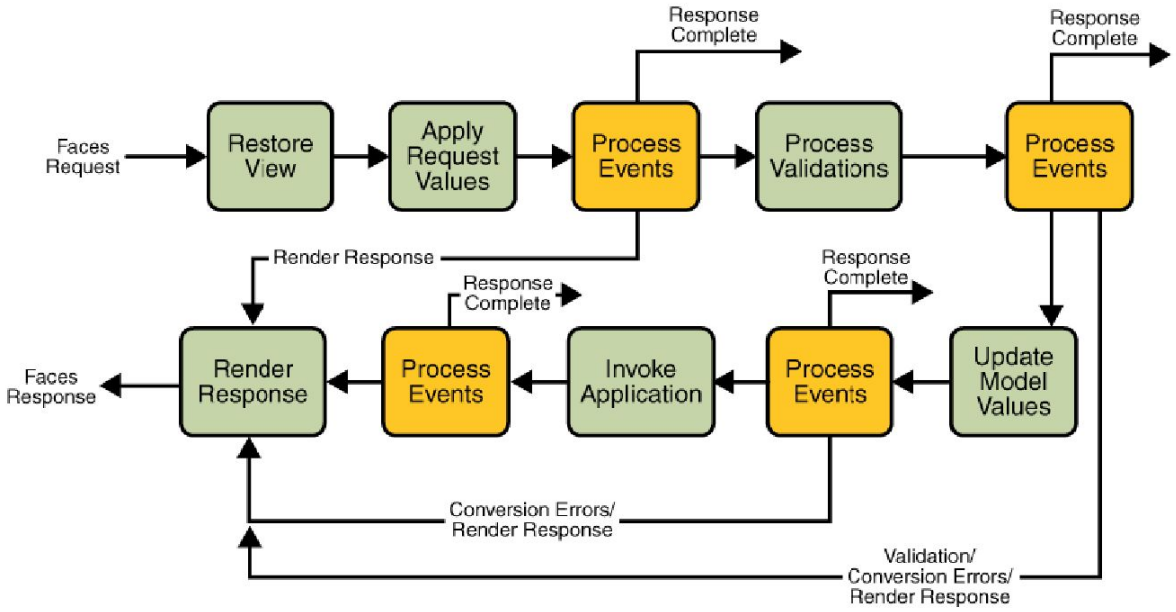
Értékeket kiírni az UIOutput és UIInput osztályokkal lehet. Mindkettőben a value attribútumban állíthatjuk be a megjelenítendő értéket, de a felhasználó csak az UIInput osztály értékeit módosíthatja. A value értéke mindig String típusú, így beíráskor és feldolgozásakor is el kell végezni a megfelelő konverziót ha más típusra lenne szükség.

Az alábbi ábra összeállított nézet memóriabeli fastruktúráját szemlélteti.



A JSF kérésfeldolgozása

A kérésfeldolgozást egy állapotgép végzi, amelynek hat fő állapota van. A kérésfeldolgozást a regisztrált FacesListener interfészt implementáló osztályok végzik, melyek egymásnak küldik a kérés paramétereit. A folyamat megszakadhat, nem szükségszerűen zajlik le a teljes feldolgozási kör.



A nézet visszaállítása (Restore View)

A keretrendszer először azt ellenőrzi, hogy postback kérés érkezett-e. A postback kérések olyan oldalra irányulnak amik állapota már mentésre került, leggyakrabban a kérés ugyan arra az oldalra irányul amire az azt megelőző is. Ha a kérés postback, akkor visszaállítja komponensek állapotát. Az eredeti állapotot a sessionből vagy a hidden űrlapváltozóból veszi. Ha nem postback a kérés, akkor létrehozza a View komponenfa gyökerét, majd a JSF kérésfeldolgozási állapotgépét a Nézet Generálása állapotba állítja.

A kérés paramétereinek rögzítése (Apply Request Values)

Az input komponensek ellenőrzik, hogy a kérésben van e hozzájuk rendelve valamilyen paraméter. Ha van, akkor azt az UIInput ősosztály submittedValue tagváltozójában eltárolásra kerül. Ez az érték azonban nem fog egyenesen a modellbe is kerülni ehhez először konvertálni és validálni kell egy második fázisban. Ez alól kivétel, ha a komponens immediate attribútuma be van kapcsolva, ekkor azonnal megtörténik az átvezetés.

Validációk futtatása (Process Validations)

A validáció a ViewRoot objektumtól kiindulva mélységi bejárással történik. A faszerkezet komponenseiből csak azokat validálja, melyek azon űrlapon vannak, amelynek a submitted metódusa igaz, a kérésben van hozzájuk paraméter, a rendered attribútuma igaz, az immediate pedig hamis. Konverterek közül először a komponenshez rendelt explicit konvertert próbálja, ha ilyen nincs az alkalmazásszintű konverterek között próbálkozik. Validálásnál nem fogad el üres karaktersorozatot ha a komponens required attribútuma igaz. Egy komponenshez több validátor is tartozhat, ha ezek közül bármelyik sikertelen a többi validátor már nem fog lefutni. A minden validátor sikeres volt az értéket validáltnak tekinti és beírja a komponens value attribútumába. Ha ebben a fázisban valahol validációs hiba történik, az állapotgép a ’Nézet Visszaállítása’ fázisba kerül.

A modell aktualizálása (Update Model Values)

A komponensek konvertált és validált value attribútumba írt értékei a modellbe íródnak. Ehhez a megfelelő managed bean setter metódusait kell használnia.

Az alkalmazás meghívása (Invoke Application)

Ebben a fázisban az UICommand leszármazott komponenseinek actionListener illetve action attribútumaiban található metódusok kerülnek meghívásra.

Nézet generálása (Render Response)

Itt a felépített komponensfa bejárásával a renderkit előállítja a kimenetet.

Managed Beanek használata

Általában minden JSP oldalhoz készítünk egy azonos nevű menedzselt osztályt is, ez nem kötelező de az esetek döntő többségében hasznos. A fejlesztés során ezek a menedzselt beanek fontos szerepet töltenek be. Komponenseknek szolgáltatnak adatokat. A saját tagváltozóiba adatokat tárolhatnak, melyekhez az alkalmazás felhasználhat. A felhasználó felületen történt változásokra reagálhatnak. A UI komponenseinek tulajdonságait befolyásolhatják.

Ha egy osztályt menedzselni szeretnénk létre kell hoznunk egy paraméterek nélküli konstruktorát. Példányosításukat a keretrendszer végzi és nem a new operátor segítségével jönnek létre. A WEB-INF/faces-config.xml fájlban regisztrálnunk kell a menedzselni kívánt osztályt. A <managed-bean-name> XML tagben az osztály teljesen kvalifikált nevét kell megadni. A <managed-bean-class> tagben pedig azt a nevet ahogy el szeretnénk érni a JSF-et használó JSP állományokban. A <managed-bean-scope> tartalma request, session vagy application lehet, attól függően, hogy a példány hol jöjjön majd létre. A session scope-ban elhelyezett menedzselt osztály példány egészen addig elérhető amíg a felhasználó sessionje él. Az application context menedzselt példányai az alkalmazás teljes futása alatt elérhetőek, nem kötődnek felhasználóhoz vagy kéréshez. A request contextben létrejött példányok a használatuk után közvetlenül a szemétgyűjtőbe kerülnek, míg a sessionben létrejöttek egy felhasználóhoz kötötten a session lezárásáig megmaradnak.

```
<faces-config version="1.2"

    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">

    <application>

        <el-resolver>minibank.JSF.util.JsfCrudELResolver</el-resolver>

    </application>

…

    <managed-bean>

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

```
        <managed-bean-class>minibank.JSF.FelhasznaloController</managed-
bean-class>

        <managed-bean-scope>session</managed-bean-scope>

    </managed-bean>

…..

</faces-config>
```

A JSP oldal forrásában a szabványos JSF komponensek találhatók. A value attribútummal lehet a komponens értékét a menedzselt osztály attribútumához rendelni. Az action attribútum az UICommand leszármazottainak esetén használható és action metódushoz rendelhető. A hozzárendelés szintaktikája a Unified Expression Language szabályait követi:

```
#{menedzselt_bean_nev.tulajdonsag}
```

A menedzselt bean nevének szerepelnie kell a faces.config.xml fájlban és tulajdonságaiban meg kell felelnie a JavaBean követelményeinek, vagyis a privát tagváltozóhoz getter és setter metódusokkal kell rendelkeznie.

Navigáció

Az oldalak közötti navigációt egy irányított gráf vezérli. A gráf csomópontjai az oldalak, az élek pedig a navigációs szabályok. Ilyen szabályt az UICommand osztály leszármazottai generálhatnak. Ilyen komponens például a HtmlCommandButton és a HtmlCommandLink, melyek action metódusa vezérli a navigációt. A navigáció lehet statikus vagy dinamikus, attól függően hogy az action értéke egy egyszerű String érték vagy egy metódus String visszatérési értéke.

A faces-config.xml –be kell felsorolni a navigációs szabályokat. Egy szabályt a <navigation-rule> elem közé kell zárni. A szabály leírásában használható további elemek még a <form-view-id>, amely azt a nézetazonosítót tartalmazza, amelynél a szabály érvénybe léphet. Itt megadható a * is mint az összes nézetet helyettesítő azonosító. Az egyes eseteket a <form-case> tartalmazza. A <form-action>-ben megadhatjuk azt a menedzselt bean metódust, amelynek visszatérési értékeként kell aktiválódni a szabálynak, míg a <form-outcome> nem köti metódushoz ezt az értéket.

```
    <navigation-rule>

        <navigation-case>
```

```
            <from-outcome>szamla_create</from-outcome>
            <to-view-id>/szamla/New.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <navigation-case>
            <from-outcome>szamla_list</from-outcome>
            <to-view-id>/szamla/List.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <navigation-case>
            <from-outcome>szamla_edit</from-outcome>
            <to-view-id>/szamla/Edit.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <navigation-case>
            <from-outcome>szamla_detail</from-outcome>
            <to-view-id>/szamla/Detail.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
```

A navigációs szabályok kiértékelése a következőképpen zajlik:

Ha null a végkimenetel, az oldalt újrarendereli, ha a bármi más akkor a <from-view-id> * nélküli értékei között keres illeszkedést. Ha nincs ilyen a * -ot tartalmazó elemek között keres és abból azt választja, amelyiknek a leghosszabb a prefixe. Ha nincs ilyen akkor a <form-view-id> nélküli szabályt választja. Az esetek közül azt választja, amelyikre illeszkedik mind a <form-outcome> mind a <from-action>, ha nincs ilyen, akkor először a <form-outcome> majd a <form-action> illeszkedését vizsgálja, ezután azt az estet keresi, ahol nincs megadva egyik sem, majd ha ilyen sincs, akkor új szabályt keres.

Ha a szabályválasztás után ugyanazon az oldalon maradtunk, ahol eddig voltunk a memóriába mentett view komponensfát újratölti, ha új oldalra irányított bennünket, akkor új nézetet kell felépíteni és a keretrendszer ezt állítja be viewRootnak.

JSF tagek

A JSF technológiát a JSP oldalak kiegészítésével lehet használni. Ez azt jelent hogy a JSP oldalakon belül JSF tageket használunk amiket a fordító feldolgoz. A tagek egyik csoportja az újnevezett Core Tag-ek. Használatukhoz a JSP oldalon belül az következő dikertívát kell beilleszteni:

```
<%@taglib prefix=”f” uri=”http://java.sun.com/jsf/core”%>
```

A Core Tageket a következő táblázat foglalja össze:

Tag	**Hatása**
f:view	Egy nézetet hoz létre, ennek a törzsén belül használhatunk más JSF komponenseket
f:subview	Egy nézet alnézetét hozza létre
f:facet	Egy facetet ad a komponenshez
f:attribute	Egy attribútumot állít be egy komponenshez.
f:param	Előállít egy paraméter komponenst
f:actionListener	Egy ActionListenert állít be egy komponenshez
f:valueChangeListener	Meghívja egy megadott metódust ha a figyelt érték megváltozik
f:setPropertyChangeListener (JSF 1.2)	Egy listenert ad egy komponenshez ami egy bean adott tulajdonságának értékét figyeli
f:converter	Egy Converter példányt ad a komponenshez
f:convertDateTime	Egy ConvertDateTime példányt ad a komponenshez
f:convertNumber	Egy NumberConverter példányt ad a komponenshez
f:validator	Egy validátort ad a komponenshez
f:validateDoubleRange	Double érték határait ellenőrző validátort ad egy komponenshez.
f:validateLength	Szöveghosszt ellenőrző validátort ad a komponenshez
f:validateLongRange	Long érték határait ellenőrző validátort ad egy komponenshez.
f:loadBundle	Betölt egy properties fájlt és elérhetővé tesz Map -ként
f:selectitems	A komponens választási lehetőségei közül többet is megad.
f:selectitem	A komponens választási lehetőségei közül egyet megad.
f:verbatim	Egy megadott tartalmat helyez el a nézetben.

Facettel egy szülő tag törzsén belül lehet elkülöníteni egy részt valamilyen logika szerint. Használni lehet saját komponensek írásánál vagy pl fejléceknél adattáblákban.

Ha a verbatim tag XML kifejezés, akkor nyitó és záró tagokat is kell tartalmaznia.

A core tagek elsősorban a keretrendszer működését befolyásolják, a valódi megjelenő komponensek a renderkittől függenek. A leggyakrabban a HTML 4.01-et generáló renderkitet alkalmazzák. Használatához a JSP oldalon a következő direktívát kell beilleszteni:

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

A HTML tagok attribútumainak legtöbbje használható egy speciális módban, amit pass-through módnak hívnak. Ezeket az attribútumokat a keretrendszer helyezi el azokban a HTML tagekben, amik majd a JSF komponenst fogják reprezentálni az oldalon. Ezek az attribútumok a következők

accesskey	readonly
accept	rel
accept-charset	rev
alt	rows
border	shape
charset	size
coords	style
dir	tabindex
disabled	target
hreflang	title
lang	type
maxlength	width

pass-through attribútumként használhatók még a javascript eseménykezelők.

A JSF-komponensek legfontosabb attribútumai

Value: a komponensek értékét lehet vele beállítani. Az input komponenseknél a getter és setter metódusokat hívja meg, míg az output komponenseknél csak a gettert. A value típusa függ az komponens típusától és a csatolt konvertertől.

Rendered: logikai értke lehet, amely azt szabályozza, hogy a komponens megjelenjen e a nézetben.

Disabled: a komponenst hozzáférhetőségét tilthatja a felhasználó számára a kimenetben. A HTML komponenseknek van hasonló nevű attribútuma, különbség viszont, hogy a JSF nyilvántartja az értékét és külső kérésből nem engedi megváltoztatni az értékét.

Binding: itt állíthatjuk be ha a JSF taget megvalósító JSF osztályt view-beli példányát el szeretnénk érni. Értéke egy Java EL kifejezés lehet, ami a menedzselt bean egy UIComponent típusú tulajdonságára mutat. A kérésfeldolgozás során a komponensfa felépítésekor a keretrendszer ellenőrzi, hogy egy komponens binding attribútuma be van-e állítva. Ha van beállított érték a menedzselt bean meghívásával megpróbálja elérni az UIComponent példányt, amit beilleszt a komponensfába. Binding attribútummal akkor látunk el komponenseket ha a komponensfába egy speciális UIComponent leszármazottat szeretnénk helyezni az alapértelmezett helyett, például programozott tulajdonságokkal bírót. Ugyanez a helyzet, ha szeretnénk hozzáférni futás közben a komponensfabeli példányhoz, hogy tulajdonságokat kérdezzünk le tőle.

JSF HTML TAGEK

Tag neve	Funkciója
column	Egy dataTable tagnél használatos oszlopot helyez ki. Zárótag is tartozik hozzá.
commandButton	Egy gombot helyez ki. Használható navigációs célra és formok input komponenseinek feldolgozásának indítására.
commandLink	Linket készít. Elsősorban külső oldalak elérésére való.
dataTable	Táblázatok készítéséhez használatos.
form	Nyitó és záró taggal is rendelkezik, szerepe megegyezik a HTML <form> tagéval, azonban hiddenként elküldi az űrlap nevét és a postback kérés esetén az oldal visszaállításához szükséges viewState azonosítót.
graphicImage	Képeket helyez fel az oldalra. A HTML<IMG>-vel szemben előnye hogy a value JSF EL kifejezés is lehet.
inputHidden	Rejtett szövegbeviteli mezőt helyez ki.
inputSecret	Egysoros jelszóbeviteli mezőt helyez ki.
inputText	Egysoros szövegbeviteli mezőt helyez ki.
inputTextarea	Többsoros szövegbeviteli mező. A cols és rows attribútumokkal beállítható a mérete.
message	Egy felhasználónak szánt üzenetet rendel egy komponenshez.
messages	Az aktuális View üzeneteit helyezi ki.
outputFormat	Tetszőleges szöveget ír ki, paramétereit menedzselt beanből is veheti.
outputLabel	Egy cimkét állít be, lehetővé teszi, hogy a felhasználók egyénileg állítsák be az oldal CSS –it.
outputLink	Egy ’href ’ attribútumú linket helyez ki.
outputText	Tetszőleges szöveget ír ki az oldalra. Value értékét paraméterből is veheti.
panelGrid	Egy tábla elemet helyez ki, zárótagje is van.

panelGroup	Egy panel konténerkomponenst helyez ki.
selectBooleanCheckbox	A felhasználótól egy logikai értéket kérhetünk be vele.
selectManyCheckbox	Szelekciós komponens, több elem is választható
selectManyListbox	Szelekciós komponens, több elem is választható
selectManyMenu	Szelekciós komponens, több elem is választható
selectOneListbox	Szelekciós komponens, elemeiből egy választható
selectOneMenu	Szelekciós komponens, elemeiből egy választható
selectOneRadio	Szelekciós komponens, elemeiből egy választható

A szelekciós komponenseknél az f:selectItems attribútumnak a menedzselt bean egy List típusú tulajdonságára kell mutasson, melyek SelectItem típusú elemeket kell tartalmazzon. A SelectItem az Objektum mellett a listában megjelenítendő cimkét is tartalmazza.

Konverterek, validátorok, üzenetkezelés

Az input komponensek közül a felhasználó által szabadon kitölthetők String Java típust adnak a modell felé. Ha más típusú értéket szeretnénk előállítani ebből akkor konverzióra van szükségünk. Konverzió történhet programozottan vagy konverterek segítségével. Konvertert elhelyezni a komponens tagjei közé írva lehet. Ilyen konverter például a convertNumber és a convertDateTime. A validátorok az értékek szemantikai helyességét vizsgálják. A validateLongRange ellenőrzi, hogy az érték egy megadott tartokmányban van-e integer típus esetén, míg a validateDoubleRange ugyanezt tesz double típus esetén. A validateLength String típus hosszát vizsgálja.

Ha a validációk futtatása fázisban érvénytelen értéket talál, az értéket tartalmazó komponens setValid metódusát false értékkel hívja meg és a komponens üzenetlistájában elhelyez egy hibaüzenetet. Ha ez megtörtént a keretrendszer nem lép tovább a ’modell aktualizálása’ fázisba, hanem a ’nézet generálása’ fázissal folytatja. A felhasználó számára ez azt jelenti, hogy ismét megjelenik az űrlap immár a hibaüzenetekkel kiegészítve. Hibaüzenet készítésére a h:message tagot használjuk ’for’ attribútummal, ami beazonosítja a validálandó

beviteli mezőt. Nem validátor de szemantikai ellenőrző szerepet tölt be az inputText komponensek egyik attribútuma a required, amely kötelezővé teszi a mező kitöltését. A JSF saját konverterein és validátorain kívül lehetőség van sajátok írására is, ehhez a javax.faces.convert.Converter interfészt, illetve a javax.faces.convert.Validator interfészt implementáló osztályokat kell írni.

```
<h:inputText id="Számlázás dátuma" value="#{szamla.szamla.szamlaDate}"
title="SzamlaDate" >

<f:convertDateTime pattern="MM/dd/yyyy HH:mm:ss" />
```

Adattáblák

Az adattáblák feladata az adatok áttekinthető táblázatos formában történő megjelenítése. Az adatok (entitások) jellemzően üzleti objektumok halmaza. Táblázatot a dataTable komponens beszúrásával kezdhetünk, melynek value attribútuma az entitásokat tartalmazó listára kell mutasson. Az egyes elemekre a var attribútumban megadott néven hivatkozhatunk. A törzsben az oszlopokat a column tagok alkotják, melyek adattagokból és fejlécből állnak. Megkülönböztetésükre a faceteket használjuk, amely lehetővé teszi, hogy XML tagek halmazát egyetlen komponensként használjuk. A facet name attribútuma lehet ’header’ vagy ’footer’ attól függően, hogy fej vagy lábléckénti akarjuk megjeleníteni. A column faceten kívüli törzsében az outputText segítségével írathatjuk ki az entitás megfelelő adattagjának értékét.

Ahhoz hogy az adattáblákat a gyakorlatban is könnyen használhassuk bizonyos funkciókkal kiegészíthetjük a dataTable komponenst. Használhatunk CSS stílusokat és a páros és páratlan sorokat különböző stílusokkal ellátva olvashatóbbá tehetjük a táblát. Lehetővé tehető a táblázat lapozhatósága, csökkentve ezzel a processzor terheltségét és a sávszélesség túlzott használatát. Gyakori igény lehet a táblázatból kiválasztott elem további feldolgozása, aminek kijelölése általában egy linkkel vagy egy rádiógombbal lehetséges. Több entitás kijelölése is hasonlóan történhet, leginkább egy jelölőnégyzettel. Táblázatokban, listákban megszokott, hogy valamilyen szempont szerint sorrendbe lehet rendezni az elemeit, amit a fejlécükre kattintva lehet elérni.

```
<h:dataTable value="#{ugyfel.ugyfel.szamlaCollection}"

 var="item" border="0" cellpadding="2" cellspacing="0"
rowClasses="jsfcrud_odd_row,jsfcrud_even_row" rules="all"
style="border:solid 1px"

rendered="#{not empty ugyfel.ugyfel.szamlaCollection}">

<h:column>

<f:facet name="header">

<h:outputText value="SzamlaId"/>

</f:facet>

<h:outputText value=" #{item.szamlaId}"/>

</h:column>

...

</h:dataTable>
```

Nemzetköziesítés

Weboldalakkal szemben elvárás lehet a többnyelvűség. A nemzetköziesítéshez ismernünk kell azt a nyelvet, amelyen a felhasználónk használni szeretné az oldalt. Ezt rábízhatjuk egy automatizmusra vagy a felhasználótól is lekérdezhetjük. A HTTP kérések fejlécében jelölve van, hogy a böngésző milyen nyelvű tartalmat részesít előnyben. Alapértelmezésként ez a nyelv megegyezik a böngésző felületének nyelvével. A JSF figyeli ezt a fejlécet és ha lehetősége van más a kért nyelvű tartalmat visszaadni akkor úgy tesz.

A többnyelvűséghez be kell állítanunk a változatokat tartalmazó fájlok elérését a faces-config.xml-ben. A <locale-config> tagek között kell felsorolni a támogatott nyelveket, erre a <default-locale> és a <supported-locale> tagekben jelölhetjük az alapértelmezett és a lehetséges nyelveket. A <message-bundle> tag tartalmazza, hogy a WEB-INF könyvtárból kiindulva hol vannak az alkalmazás properties fájljai. Minden nyelvhez külön fájl tartozik, a fájl nevét a keretrendszer a <locale-config>-ban felsorolt nyelvek kódjából és a <message-bundle>-ban megadott fájlnevek összeolvasztásából képzi. A nézetben az üzenetfájlokat a loadBundle taggel thetejük elérhetővé, az egyes címkékre pedig a var attribútummal hivatkozhatunk. A nemzetköziesítéshez még hozzátartozik, hogy az eltérő nyelvek különleges karaktereinek kiírására a keretrendszert fel kell készíteni, erre a weboldalakon általában az UTF-8 kódolást szokták használni.

```
<application>
<locale-config>
<default-locale>en</default-locale>
<supported-locale>hu</supported-locale>
</locale-config>
<message-bundle>minibank.Messages</message-bundle>
</application>
```

Entitások

Az entitások a J2EE technológia alapvető egységei. Szerepük hasonló az entity beankhez, azonban működésük eltér. Az entitások a Java Persistence API perzisztens objektumai, melyek egy adott objektum adatait perzisztens módon tárolják, és azok elérését biztosítják. Ez a gyakorlatban azt jelenti, hogy az adatokat az alkalmazások által használt relációs adatbázisban elérhetővé teszik. Ehhez az objektumrelációs leképezést használják (Object Relational Mapping) melynek lényege, hogy az egyes osztályokat adatbázistábláknak, az osztályok tulajdonságait pedig oszlopoknak feleltetik meg. Ezáltal az egyes példányok a tábla egyes soraként kerülnek eltárolásra a memóriában. Az ORM nagyban megkönnyíti a fejlesztést, hasonló leképezést megvalósító eszközök is léteznek (Hibernate, TopLink), melyek mind az EJB konténertől függetlenek és perzisztens objektumnak egyszerű Java osztályokat használnak (Plain Old Java Object). A SUN ezen rendszerek tapasztalatait felhasználva adta ki a Java Persistence API-t (JPA) az Enterprise Java Beans 3 specifikációban. Az entity fogalma csak a JPA-ban került bevezetésre ugyanakkor az entity bean technológia párhuzamosan megtalálható a J2EE-ben EJB 2.1-es formájában. Várható, hogy a közeljövőben a JPA entitás kiszorítja az entity bean technológiát.

Az EJB3 annotációi

Az objektumrelációs leképezésben az entitások az EJB3 annotációit használják. Az annotáció különböző osztályoknak és eszközöknek szóló metaadat, melyet a forráskódba helyeznek. A Java 5 előtt annotációkat speciális formátumú kommentekbe kellett helyezni. A Java5 meghatározza az annotációk használatának szintaxisát

```
@AnnotációNeve(paraméter1=érték1, paraméter2=érték2…)

//Ha csak egy paraméter van megenedett:

@AnnotációNeve(érték)

//Ha nem adunk át paramétert:

@AnnotációNeve, vagy @AnnotációNeve()
```

```
//Ha a paraméter tömb, akkor a tömök szintaxisát követi:

@AnnotációNeve(paraméter1={elem1, elem2,…}, paraméter2=érték…)
```

Az annotáció típusát egy interfész definiálja. Ez írja le, hogy mi a neve, milyen paramétereket milyen alapértelmezett értékekkel fogad el. Ezt az interfészt importálni kell az annotáció használatához. A perzisztenciával kapcsolatos annotációk interfészei a javax.pesistence csomagban találhatók, így ezt kell importálni.

A leképezéshez egy osztályt kell írni a megfelelő attribútumokkal és azok getter és setter metódusaival. A @Entity annotációval megjelöljük az osztályt, a @Id-t pedig az elsődleges kulcs jelölésére használjuk. A @Table jelzi, hogy az osztály milyen táblára, a @Column pedig, hogy melyik attribútum milyen oszlopra képződjön le. A JPA ezeket annotációk nélkül, automatikusan is megteszi az osztály és az attribútum nevét felhasználva, de a felhasználó ezt az alapértelmezést felüldefiniálhatja. Dátumtípusoknál @Temporal-lal kell ellátnunk azokat a típusokat, ahol nincs szükségünk a teljes milliszekundum pontosságú Timestamp típusra. A @SecondaryTable annotációval több relációban tárolhatjuk egy entitást. @Transient annotációval kell jelölni az entitás nem perzisztens attribútumait. A perzisztens attribútumok a következő típusok közül kerülhetnek ki:

String	java.sql.Timestamp
java.math.BigInteger	byte[]
java.math.BigDecimal	Byte[]
java.util.Date	char[]
java.util.Calendar	Character[]
java.sql.Date	enum
java.sql.Time	

Ezen felül bármely primitív típus és azoknak megfelelő objektumtípus, valamint más entitás, azok gyűjteménye illetve beágyazott osztály. Ez utóbbi olyan speciális entitás, amely egy másik perzisztens objektumhoz kapcsolódva létezik. Ezen típusmegszorítások biztosítják, hogy az adott entitás leképezhető legyen az SQL nyelvre.

Csakúgy, mint egy SQL táblának egy entitásnak is kell legyen egy kitüntetett attribútuma, ami alapján egyértelműen beazonosítható egy sora: ez az elsődleges kulcs. Alkalmazásból nem szabad megváltoztatni az értékét típusa pedig korlátozottabb, mint az

általános perzisztens attribútumoknak: primitív típusok a lebegőpontos típusokat kivéve, a csomagolóosztályaik, a String, a java.util.Date és a java.sql.Date lehetnek.

Az EJB3 entitások támogatják az elsődleges kulcsok automatikus generálását. Ennek hiánya az entity beanek egyik hátránya volt. Ha az elsődleges kulcs automatikus képzését a gépre szeretnénk bízni, akkor a @GeneratedValue annotációval kell ellátnunk az elsődleges kulcsot. Csak egész numerikus értéket lehet automatikusan generáltra állítani. A következő érték előállításának négy különböző stratégia alapján történhet. Ha a GenerationType enum paraméternek SEQUENCE értéket adunk, akkor az adatbázis által kezelt számláló adja az értéket. Ennek beállításait a @SequenceGenerator annotációban deklarálhatjuk. Az IDENTITY érték egy adatbázistábla egy autoinkrementálódó oszlopára fogja leképezni az elsődleges kulcs értékét. A TABLE az adatbázisbeli tábla meghatározott értékét adja az elsődleges kulcsnak, a kérdéses táblát a @TableGenerator paraméterrel állíthatjuk be. Az AUTO érték esetén a korábbi három stratégiából választ egyet az adatbázis-kezelő típusának függvényében. Összetett kulcs a @IdClass annotációval készíthető.

Ha az annotációkkal az attribútumokat jelöljük meg akkor az adatbázisból a megfelelő adatok közvetlenül a tagváltozókba töltődnek be vagyis nem a getter vagy setter metódusok segítségével. Ha azonban a gettereket, settereket jelöljük meg az annotációkkal, ekkor ezekkel fog megtörténni az adatok betöltése és visszamentése. Figyelni kell ugyanakkor arra, hogy egy entitáson belül ugyanazt az annotálást kövessük, tehát vagy csak attribútumot vagy csak metódust jelöljünk, ellenkező esetben az entitás működése nem lesz definiált.

Perzisztencia provider és az entitás menedzser

A Java Persistent API interfészeket definiál amiket újnevezett perzisztencia providerek használnak, hogy az entitások életciklusának menedzselését , adatbázis- és memóriabeli adatok szinkronizálását és minden egyéb perziszenciával kapcsolatos folyamatot biztosítsanak. Ilyen perzisztencia providerből több fajta is létezik, leggyakrabban használt a TopLink, a Hibernate vagy az Open JPA. A java standard edition kiadásához is használhatók megfelelő osztálykönyvtárak csatolásával, az Enterprise Editionben pedig a futtatási környezet tartalmaz valamilyen JPA implementációt.

Az entitások kezelése az EntityManager interfész segítségével lehetséges. Az EntityManager entitások halmazával dolgozik, melyet perziszenciakontextusnak hívunk. Ebben a halmazban minden egyedi kulccsal rendelkező perzisztens példányhoz egy hozzá tartozó memóriabeli példányt rendel. Ha egy tranzakcióban több komponenst is részt vesz a perziszenciakontextusnak azonosnak kell maradnia a tranzakció befejezéséig. Ez elérhető ha a tranzakció elején kapott EntityManager referenciát argumentumként továbbítja a komponensek között. Ezt a feladatot a konténer végzi, tehát amikor egy EJB komponens referenciát kér az egy EntityManagerre a konténer fogja az EntityManager példányhoz hozzárendelni az aktuális perzisztenciakontextust. Ezeket az EntityManagereket konténer által kezelt entitáskezelőknek hívják. Az is előfordulhat, hogy a nem akarjuk, hogy ugyanazon tranzakcióban szereplő EntityManager példányok hozzáférjenek egymás perzisztenciakontextusához. Ez az EntityManagerFactory interfész segítségével készített alkalmazás által kezelt EntityManagerrel történhet. Java SE környezetben – konténer híján- ez másképp nem megoldható.

Az EntityManagerFactory létrehozásakor megadhatunk egy paramétert, ami a perzisztenciaegység neve. A perzisztenciaegység egy halmaz, ami konfigurációs információkat, entitásokat, objektumrelációs leképezések módját definiáló annotációkat vagy xml metaadatokat és az EntityManagerFactory által létrehozott EntityManagereket tartalmazza. A perzisztenciaegység fizikai fájlok formájában is megjelenik: lefordított class fájlok mellett a persistence.xml fájlból áll, ami a META-INF könyvtárban található. Itt több perzisztenciaegység definiálására is lehetőség van. Az EntityManager nem szálbiztos, tehát a szinkronizációt a fejlesztőnek kell kezelnie – az EntityManagerFactory ugyanakkor szálbiztos osztály.

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

  <persistence-unit name="MinibankPU" transaction-type="JTA">

    <jta-data-source>jdbc/APP_ApacheDerby</jta-data-source>

    <properties/>

  </persistence-unit>

</persistence>
```

A persistece.xml-ben a <persistence unit> -ben deklaráljuk a perzisztenciakontextus nevét és a tranzakció típusát. Ez utóbbi két lehetséges értéket vehet fel: ha JTA értéket állítunk be a tranzakciók elejét és végét nem az EntityManager végzi, hanem a Java Transaction API közvetlenül az erőforrásmenedzsert fogja meghívni. Ha az EntityManager konténer által kezelt, akkor ez a beállítás kötelező, de Java EE környezetben amúgy is ez az általános választás. Java SE környezetben a másik választás a tipikus, ez a RESOURCE_LOCAL. Ekkor a tranzakciók határait az EntityManager.getTransaction() hívással kapott EntityTransaction interfészen keresztül jelöli ki az alkalmazás, amelyek így csak lokális tranzakciók lehetnek. A <provider> tagben a perzisztenciaprovidert adatjuk meg. Az alkalmazásszerveren regisztrált adatforrásokat, melyekbe a provider az adatokat menti és visszatölti a <jta-data-source> tagben állíthatjuk be. A <properties> elemben a perzisztenciaprovider saját beállításait tehetjük meg.

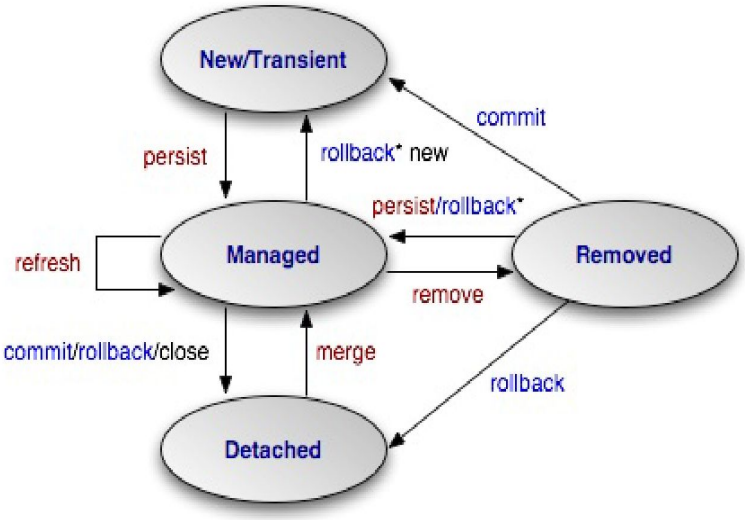
Entitások életciklusa

Új: a new operátorral példányosítva az entitás ebbe az állapotba kerül. Ekkor még nem létezik hozzá kapcsolt perzisztens adat, ezért a rajta végzett változtatások nem is kerülnek be az adatbázisba.

Menedzselt: az EntityManager.persist() hívásával kerül ebbe az állapotba. Az entitás a perzisztenciakontextusba kerül, a tranzakciók végén a bennük lévő adat szinkronizálódik az adatbázissal. Ha valamilyen oknál fogva a tranzakció vége előtt szeretnénk még a tranzakció vége előtt visszaírni az adatbázisba a változásokat, akkor az EntityManager.flush() metódussal megtehetjük. Ennek fordítottja vagyis az entitás frissítése az adatbázisból az EntityManager.refresh() paranccsal történik.

Lecsatolt: az entitások akkor kerülnek ebbe az állapotba, ha megszűnik a perzisztenciakontextus. Ekkor a rajtuk végzett változtatások nem kerülnek be az adatbázisba, viszont ilyenkor használhatók Data Transfer Objectként a különböző szoftverrétegek között. Ha a lecsatolt entitáson változás történt, amit az adatbázisba szeretnénk menteni, akkor az EntityManager.merge()-el egy menedzselt entitásba menthetjük át a tartalmát.

Törölt: Csak menedzselt állapotból kerülhet törlésre kijelölt állapotba objektum az EntityManager.remove() hívással. Végleg megszűnni akkor fog, ha a tranzakció jóváhagyásra került, előtte még visszaállítható menedzselt állapotba az EntityManager.persist() hívással.



Entitások keresése

A Java Persistence API változatos lehetőségeket kínál az entitások keresésére. Lehetőség van elsődleges kulcs alapján keresni az EntityManager interfész segítségével. A keresési metódus az elsődleges kulcs mellett egy valamilyen típussal paraméterezett generikus osztálytípust vár. Visszatérési értéke a korábban megadott típus lesz, ez akár valamilyen entitás is lehet.

```
Ugyfel aktUgyfel = em.find(Ugyfel.class, 3);
```

Összetettebb lekérdezések elvégzésére az EntityManagernek vannak további metódusai melyek visszatérési értéke Query interfészt implementáló objektumok. A bennük szereplő lekérdezésekben használhatunk native SQL vagy EJB-QL lekérdezőnyelvet is. A JPA további lehetőségekkel egészíti ki EJB-QLt, együttműködésüket Java Persistence query language-nek nevezik. Ezen bővítményekkel lehetővé válik egyszerre több entitás módosítása vagy törlése, a JOIN, HAVING és GROUP BY allekérdezések támogatása. A JPQLben a bemenő paraméterek helyét ?1, ?2 helyett :paraméterNév1, :paraméterNév2 alakban definiálhatjuk a lekérdezésekben és nyelv támogatja a projekciót is. Megtehetjük, hogy a lekérdezés eredményeként kapott oszlopokat valamilyen korábban definiált objektumként kapjuk vissza. JPQLben lekérdezéseket futási időben is hozhatunk létre.

```
//EJB-SQL-lekérdezés dinamikus összeállítása

Public Query createQuery (string ejbqlString)

// natív SQL-törlést vagy –módosítást végző lekérdezés létrehozása

public Query createNativeQuery (String sqlString)

// natív SQL-lekérdezés létrehozása, a visszatérési

// típus definiálásával

public Query createNativeQuery (String sqlString, Class resultClass)

// natív SQL-lekérdezés létrehozása a visszakapott

// oszlopok leképezése entitásokra a resultSetMapping
```

```
// paraméterrel azonosított

//@SqlResultSetMapping annotációval történik

public Query createNativeQuery (String sqlString String resultSetMapping)

//Query létrehozása statkusan előre definiált, névvel ellátott,

// EJB-QL vagy natív SQL nyelven megírt lekérdezésből, amit

//@NamedQuery vagy @NamedNativeQuery annotáció tartalmaz

public Query createNamedQuery (String nameOfQuery)
```

Kapcsolatok entitások között

A perzisztens objektumok létrehozásakor a közöttük lévő kapcsolatok jelölésére is szükség van. Ezt az entitásoknál annotációkkal tehetjük meg. Az egyes kardinalitásoknak megfelelően a @OneToOne, @OneToMany, @ManyToOne és a @ManyToMany annotációk állnak rendelkezésünkre. Ezeket a kapcsolat másik végén álló tagváltozóra vagy getter metódusra kell alkalmazni attól függően, hogy metódusalapú elérésről van-e szó. Ha a kapcsolat egyirányú, akkor csak az egyik oldalról lehet elérni az entitást. Kétirányú kapcsolatnál viszont meg kell határozni a tulajdonosi viszonyt. Egy-egy kapcsolatban az idegen kulcsot tartalmazó oldal a tulajdonos. Egy-több kapcsolatban az a tulajdonos amelyik a @ManyToOne annotációt tartalmazza (a több oldal). Több-több esetben kapcsolótábla közbeiktatatására van szükség, itt mindkét oldal lehet tulajdonos. A tulajdoni viszony meghatározására a kétirányú kapcsolatok összerendelésénél van szükség. A kétirányú kapcsolat inverz oldalán egy mappedBy attribútumban vissza kell hivatkozni, egyébként a perzisztencia provider nem fogja kétirányúnak tekinteni a kapcsolatot.

```
@Entity

@Table(name = "UGYFEL")

@NamedQueries({@NamedQuery(name = "Ugyfel.findAll", query = "SELECT u FROM Ugyfel u"),
@NamedQuery(name = "Ugyfel.findByUgyfelId", query = "SELECT u FROM Ugyfel u WHERE
u.ugyfelId = :ugyfelId")})

public class Ugyfel implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
@Id
@Basic(optional = false)
@Column(name = "UGYFEL_ID")
private Integer ugyfelId;
@Lob
@Column(name = "UGYFEL_NAME")
private String ugyfelName;
@OneToMany(mappedBy = "szamlaOwner", fetch = FetchType.EAGER)
private Collection<Szamla> szamlaCollection;
```

Egy-egy kapcsolatban bármely oldalra helyezhető az idegen kulcs, míg egy-több kapcsolat esetében a több oldalon kell lennie. A több – több kapcsolatban a két tábla közötti kapcsolótábla tartalmazza az idegen kulcsokat. Az idegen kulcsot tartalmazó oszlopot a @JoinColumn, a kapcsolótáblát pedig @JoinTable annotációval megjelölhetők. Használatuk nem kötelező: a perzisztencia provider az entitás- és attribútumneveket használja alapértelmezett nevekként.

A Java Persistence API-ban az adatbázis konzisztenciájának megtartása a fejlesztőre hárul. Ez azt jelenti, hogy egy kapcsolatmódosításban érintett összes entitást módosítani kell, egyébként nem lesz definiált a perzisztencia provider működése. Ilyen eset például a kaszkádolt törlés, melynél az egy-több kapcsolat esetében a tulajdonos törlésekor az inverz oldal objektumainak is törlődniük kell. A JPA kaszkádolást az entitások életciklusának másik három eseményére is lehet használni. Beállításuk a kapcsolat típusát jelölő annotáció paraméterében adható meg, értékük CascadeType.REMOVE, CascadeType.REFRESH, CascadeType.PERSIST, CascadeType.MERGE, CascadeType.ALL lehet.

Erőforrás-kezelés szempontjából fontos kérdés, hogy ha az entitás a memóriába kerül betöltődjenek-e vele a hozzá kapcsolódó entitások is. Ha az entitáshoz kapcsolódó perzisztens objektumok is a memóriába töltődnek akkor mohó betöltésről, ha viszont csak a kapcsolódó entitásokra történő hivatkozáskor töltődnek be, akkor lusta betöltésről beszélünk. Az egyik megoldás erőforrás-takarékosabb, a másik viszont gyorsabb működést eredményez. Az entitások memóriába való betöltését a fetch paraméterrel állíthatjuk be. Alapértelmezett értéke a FetchType.EAGER, ezt a FetchType.LAZY értékkel állíthatjuk lustára, ám perzisztencia

provider ezt felülbírálva mohó betöltőstratégiát választhat. Az entitások kezelésekor figyelni kell rá, hogy ha egy entitás lecsatolttá válik, akkor a kapcsolódó objektumokból csak azok lesznek elérhetők, amik már a memóriában vannak, ha ilyenkor a lusta betöltés miatt korábban nem hivatkozott értékre hivatkozunk, kivételt kapunk. Figyelni kell továbbá a lecsatolt entitás újramenedzselésénél, hogy a perzisztencia provider a nem memóriabeli kapcsolódó entitásokat törli. Ennek megoldására a merge() hívás előtt elsődleges kulcs alapján meg kell keresni az eredeti entitást, amelybe a módosított entitás megfelelő értékeit átmásoljuk.

A fetch stratégiák értelmezhetők az entitások attribútumaira is. Így a lusta fetch-el megakadályozhatjuk, hogy nagyméretű képek vagy kollekciók feleslegesen memóriát foglaljanak, de ez csak metódusalapú elérés esetén lehetséges.

Összefoglalás

A JSF a Sun által támogatott webes keretrendszer specifikáció. Az 1.2-változatot a korábban megjelent keretrendszerek használatából szerzett tapasztalatok felhasználásával írták meg. Célja a fejlesztés során ismétlődő feladatok leegyszerűsítése, fejlesztőbarátabbá tétele, valamint az üzleti logika és a felhasználói felület biztonságos összekapcsolása. Általánosságban olyan problémák megoldására ad lehetőséget, mint a validáció, az oldalak közötti újrafelhasználhatóság, a komponensek támogatottsága, az intelligens komponensek fejlesztésének lehetősége, az oldalak közötti navigáció, a nemzetköziesítés, a komponensek állapotának mentése, a fejlesztői szerepek elkülönítése, a szoftverbiztonság és a bővíthetőség. A JSF együttműködik más technológiákkal is, leggyakrabban az AJAX-al egészítik ki. Támogatják a legnépszerűbb fejlesztőeszközök, mint a NetBeans és az Eclipse. Komponensalapúságának köszönhetően a JSF moduláris keretrendszer, komponenseinek interfészei szabványosítottak, így az alapértelmezett mechanizmusok lecserélhetők. Az MVC architekturális mintát használja, melyben a model szerepét az entitásosztályok, a nézet szerepét egy JSP oldal, a vezérlőét pedig egy szervlet végzi. A nézet egy fastruktúrát épít fel, melynek gyökéreleme egy UIView típusú objektum, ágai és levelei pedig más objektumok. Ezen absztrakt objektumok mindegyikéhez tartozik egy renderer, amely a alkalmazói környezettől függően állít elő belőlük valós objektumokat. A JSF használja a JavaBeans technológiát, de a beanek használatához menedzseltté kell tenni őket. A menedzselt beanek betölthetnek session, application és request szerepeket. Az oldalak közötti navigációt szabályok által vezérelt irányított gráfban adhatjuk meg. A komponenseket a hagyományos JSP oldalakba illesztett JSF tagekkel adhatunk a nézethez, tulajdonságaikat az attribútumaikkal változtathatjuk. Sok kódolástól kíméli meg a fejlesztőt a JSF egyszerű validátor és konverterkezelési technikája, és az adatokat rendezett formában, gyakran bővített funkciókkal is rendelkező adattábla. Az adatbázisban tárolt adatok objektumrelációs leképezéssel entitásokba kerülnek. Az entitás fogalmát a Java Persistence APIban vezették be, megvalósításában az Enterprise Java Beans 3-ban alkalmazott annotációkat használja. Az SQL nyelv és a relációs adatbázisok minden szabálya és technikája megvalósítható általa. A JPA interfészdefiniáltságát kihasználva több úgynevezett perzisztencia provider is létezik, melyek a fizikai adatbázis és a persistence API között tartják a kapcsolatot. Az entitások

kezelése a perziszenciakontextussal dolgozó entitásmenedzserrel lehetséges, keresésükre és lekérdezésükre több lehetőség is létezik.

Irodalomjegyzék:

Imre Gergely: Szoftverfejlesztés Java EE platformon SZAK Kiadó 2007

Sun Webkonferencia 2007 előadásai *(http://web.conf.hu/2007/program/i)*

Netbeans tutorialok *(http://www.netbeans.org/kb/index.html)*

People who have Passion for Java Technology *(www.javapassion.com)*