Matematikai és Informatikai Intézet

Állatnyilvántartó rendszer fejlesztése

Készítette
Bíró Patrik
Programtervező informatikus BSc

Témavezető
Balla Tamás
tanársegéd

Eger, 2021

Tartalomjegyzék

Bevezetés

Szakdolgozatom célja a haszonállatok nyilvántartásának megvalósítása. A nyilvántartott állatok lekérdezésekor megkaphatjuk az állat alapadatait, a jelenlegi és múltbéli tartózkodási helyeit, az állathoz tartozó betegségek listáját. Az egyedek azonosítását elősegíti a tartási helyek rögzítése, mely tartalmazza a címet és helyrajzi számot. Továbbá szükséges tárolnom a tenyészeteket is, melyekhez az adott egyed lesz rendelve, attól függően, hogy jelenleg melyik tenyészetben tartózkodik. A nyilvántartó rendszer megvalósítása az Apache Netbeans fejlesztői környezet segítségével történik, Java nyelven. A webalkalmazás futtatását a WildFly alkalmazásszerverrel valósítottam meg.

1. fejezet

Alkalmazott technológiák bemutatása

1.1. Java EE

Java Platform, Enterprise Edition (Java EE) a Standard Edition (Java SE) tetejére épül és egy futásidejű környezetet biztosít nagyméretű, jól méretezhető, többrétegű, biztonságos alkalmazások fejlesztéséhez, futtatásához.

Java EE platform egyszerűsített programozási modellt alkalmaz, ezért az XML utasítások helyett használhatunk beépített annotációkat a forrás fájlokban, osztályokban melyekkel a Java EE szerver telepítés és futási időben konfigurálja az adott komponenst. Annotációk segítségével különböző speciális információkat fűzhetünk a forráskódunkhoz, melyeket a telepítési konfigurációs fájlban kellene feltüntetnünk.

A Java EE alkalmazásmodell alapját a hordozhatóság, biztonság és a fejlesztői produktivitás képezi. Úgy tervezték, hogy támogassa a vállalat szintű alkalmazások fejlesztését, implementálhassa a vállalati szolgáltatásokat. Az ilyen alkalmazások feladata, hogy képes legyen különböző forrásoktól adatot szerezni és különböző kliensekkel kommunikálni.

A vállalati funkciók, melyek különböző felhasználókat szolgálnak ki a középső rétegben kapnak helyet. A középső réteg általában dedikált szervereken futnak és az összes szolgáltatáshoz hozzáférnek.

Java EE többrétegű alkalmazás modellt használ a vállalat szintű alkalmazások esetében. Funkcióktól függően az alkalmazás logika több komponensre van felosztva, ezáltal a rétegeket különböző helyeken, eszközökön tudjuk futtatni. Ezek közül a kliensréteg komponensei a kliens eszközön futnak, a web-réteg és a vállalti-réteg komponensei a Java EE szerveren, míg az Enterprise information system (EIS) réteg szoftver az EIS szerveren fut.

Alapvetően a Java EE alkalmazás három rétegre van osztva, mely a kliens, szerver

és az adatbázis részt tartalmazza. Ez a három rész lehetőséget ad arra, hogy három különböző helyen és eszközön futtassuk ezeket.

Az alkalmazások több komponensből állnak, ezek a komponensek önállóak, de képesek kommunikálni más osztályokkal, összetevőkkel. Kliensen futó komponensek az alkalmazáskliensek és az Appletek. A JavaServlet, JavaServer Faces és a JavaServer Pages technológiák webkomponensek, melyek szerveren futnak. Továbbá a komponensek között megtalálhatóak vállalati komponensek is, mint például az Enterprise JavaBeans (EJB).

Minden komponens számára konténereken keresztül biztosít szolgáltatásokat a Java EE. A konténerek interfészt biztosítanak a komponensek és az alacsony szintű platformspecifikus funkcionalitások között, melyek támogatják a komponenst. Java EE szerver biztosítja az EJB és a webkonténereket. Az Enterprise JavaBeans (EJB) konténer a vállalati bean-eket kezeli, a webkonténer weboldalakat, servlet-eket, EJB komponenseket kezel. Ezek a konténerek szerveren futnak, míg a kliens komponenseit kezelő konténerek egy kliensen futnak, ahogy az alkalmazás kliens konténer, ami a kliens oldali komponenseket kezeli, és az Applet konténer, amely az appleteket kezeli.

1.2. Java Persistence API

A legtöbb vállalati alkalmazás hatalmas mennyiségű adatot tárol és kezel. Fejlesztők számára fontos, hogy az adatbázis műveletek végrehajtása hatékonyabb legyen. A JPA használatával az interakció terhe jelentősen csökken.

A Java Persistence API az osztályok és metódusok segítségével megkönnyíti a nagy mennyiségű adatok tárolását az adatbázisokban azáltal, hogy hidat képez a relációs adatmodellek és az objektummodellek között. Összekapcsolás során különböző eltéréseket figyelembe kell vennünk. A relációs adatmodellel szemben az objektum részletesebben leírható, lehetőség van az öröklésre, amit nem minden relációs adatbázis biztosít. Java esetében egy objektumot kétféleképpen lehet összehasonlítani egy másik objektummal, a memóriában található címük és a belső állapotuk alapján, míg a relációs adatmodell esetében az elsődleges kulcs segítségével tudjuk ezt megtenni. Továbbá létrehozhatunk kétirányú kapcsolatot azáltal, hogy kétszer definiáljuk az asszociációt az objektummodell esetében, viszont a relációs adatmodellek nem definiálhatnak több kapcsolatot miközben egy adott objektumot már vizsgálnak. Az adatnavigáció mindkét modell esetén eltérő. Java nyelven egy bizonyos érték eléréséhez elég ha az objektum gráfon keresztül meghívogatjuk a számunkra szükséges metódusokat, míg el nem érünk a megkapni kívánt mező értékéhez. Az SQL lekérdezések ezzel szemben nagyon megnőnének, ezért a lekérdezni kívánt táblákat JOIN kulcsszavakkal kapcsoljuk össze.

Az osztály szintű architektúrát tekintve, több egységet különböztetnünk meg. Az

EntityManagerFactory, ahogy a nevében is benne van, ez egy gyártó osztály, több EntityManager példányt kezel és hoz létre. Az EntityManager egy interfész, mely kezeli a perzisztencia műveleteket. Az Entity maga a perzisztenciaobjektum, az adatbázisban megjelenő rekord mása. EntityManager tranzakcióit az EntityTransaction interfész kezeli. Persistence osztály segítségével létrehozhatunk EntityManagerFactory példányokat. Query interfész irányítja a lekérdezéseket. Ezek az osztályok és interfészek segítenek az objektumokat rekordként eltárolni az adatbázisban.

Java Persistence API objektum-relációs leképezése 3 fő részből áll. Első az objektum fázis, ami magába foglalja az entitás osztályokat, szolgáltatásokat, CRUD műveleteket, egyszóval a teljes üzleti réteget. Második a perzisztencia fázis, ide tartoznak a JPA támogatások, azaz a különböző gyártók termékei, mint például Hibernate, Eclipselink. A leképezést segítő konfigurációs XML fájl is ide tartozik, ahol megadhatjuk hogyan történjen a leképezés az adatok között. A harmadik fázis tartalmazza a relációs adat és az üzleti réteg kapcsolatát. Amíg az üzleti réteg nem tölti fel az adatokat, addig a gyorsítótárban kapnak helyet, rácsszerkezetként kezelve.

Perzisztens osztály létrehozásához szükségünk van a javax.persistence csomag implementálására. Ezt követően már használhatjuk a csomaghoz tartozó annotációkat. Az osztály neve előtt megadott @Entity annotáció jelzi, hogy az adott osztály perzisztens lesz. A @Table(name = "") segítségével megadhatjuk, hogy az adatbázis mely tábláját akarjuk leképezni. A @NamedQuery-vel definiálunk egy-egy lekérdezést, melyet bizonyos elérési névvel ruházunk fel. Ezeket a lekérdezéseket a @NamedQueries listába helyezzük, szintén az osztály neve előtt.

program lista 1.1. Példa perzisztens osztályra

```
1  import javax.persistence;
2
3  @Entity
4  @Table(name = "animal")
5  @XmlRootElement
6  @NamedQueries({
7      @NamedQuery(name = "Animal.findAll", query = "
           ↪ SELECT a FROM Animal a"),
8      @NamedQuery(name = "Animal.findByEarTag", query =
           ↪ "SELECT a FROM Animal a WHERE a.earTag = :
           ↪ earTag"),
9            ...       })
10 public class Animal implements Serializable {
```

Az adatbázisban elsődleges kulcsként szereplő mező deklarációja előtt fontos megadnunk az @Id annotációt, jelezve ezt a tulajdonságot. @GeneratedValue segítségével megadhatjuk, hogy az adott mezőértéke hogyan generálódik az adatbázisban. @Column annotáció meghatározza az adatbázis egy oszlopát, a @NotNull jelzi, hogy nem

lehet null értéke, a @Size(min = 1, max = 250) pedig az attribútum lehetséges hosszát definiálja.

A NetBeans fejlesztői környezetet használva az adatbázisban található táblák leképezése, entitás osztályok létrehozásának ideje jelentősen lerövidül. Azt követően, hogy az alkalmazásszerveren kapcsolatot létesítettünk az adatbázissal, a NetBeans-ben létrehozott webalkalmazásunkhoz az úgynevezett "Entity Classes from Database" varázsló segítségével legenerálhatjuk akár az összes adattáblát a projektünkhöz. Az így létrejött perzisztens osztályok tartalmazni fogják az attribútumokhoz megfelelő annotációkat, a fontosabb adatbázis lekérdezéseket, és természetesen a getter-setter metódusokat, konstruktorokat.

1.3. Enterprise JavaBeans

Az Enterprise JavaBeans egy fejlesztési architektúra a nagy és skálázható üzleti szintű alkalmazásokhoz, emellett alapvető részét képzi a Java EE szerveroldali platformnak.

Előnye, hogy egyszerűsített fejlesztést biztosít a nagyvállalati alkalmazásokhoz. Az EJB konténer biztosítja a legtöbb rendszerszintű szolgáltatásokat, mint például a terhelés eloszlás, naplózás, tranzakció kezelés, kivételkezelés, ezáltal a fejlesztőknek elegendő az üzleti logikára fókuszálni. Ezenfelül a konténer kezeli az EJB példányok életciklusát is.

Enterprise JavaBeans három részre osztható, a Session Bean-re, Entity Bean-re és a Message Driven Bean-re. A Session Bean egy adott felhasználó munkamenetéről tárol adatokat, ez lehet állapot teljes, vagy állapotmentes is. Az entity bean-hez képest ez jóval kevesebb erőforrást igényel, és a munkamenet megszűnését követően a session bean is törlésre kerül. Az Entity Bean a tartós adattárolásban jelenik meg. A felhasználói adatokat segítségével eltárolhatjuk az adatbázisban, később pedig visszanyerhetjük azokat. A Message Driven Bean képes fogadni és kezelni a Java üzenő rendszer üzeneteit külső entitásoktól.

Enterpries Bean egyik típusa a Stateless Session Bean, azaz az állapotmentes munkamenet bean, melyet a független műveletek végrehajtására használják. Állapotmentes, mivel nem tárolja egyik kliens állapotát sem, bár a példány állapotát megőrizheti. Az EJB konténer eltárol néhány állapotmentes bean objektumot számukra létrehozott pool-ban, majd az objektumok segítségével a kliens kéréseit kitudja szolgálni. Viszont mivel több objektum van eltárolva, nem garantált, hogy egy adott kérésre, metódushívásra mindig ugyanaz az érték kerül visszaadásra.

Az Enterpries Bean másik típusa a Stateful Session Bean, az állapot teljes munkafolyamat bean, amely a kliens állapotát megőrzi a példányok változóiban. Az EJB konténer a kliensek kéréseinek teljesítésére biztosít egy stateful session bean-t, a kérések megszűnését követően ez a bean is törlésre kerül.

EJB 3.0 esetén az annotációk az EJB osztályok konfigurációs metaadatait írják le. A @Stateless, @Stateful és a @MessageDrivenBean annotációk az osztály előtt megadva meghatározzák annak típusát. Az attribútumokat tekintve mindhárom annotáció esetén megadhatunk egy tetszőleges nevet, leírást, továbbá a Java Naming and Directory Interface (JNDI) meghatározó nevét. A @MessageDrivenBean-nek attribútumai között még megadhatjuk az üzenetfigyelő interfészt, és a működési környezetében a konfigurációs részleteket. @Local annotációval megadhatjuk a helyi bean-eket, melyeket csak a helyi, azonos helyen szereplő kliensek érhetnek el, míg a @Remote annotációval ellátott bean-t különböző alkalmazásokból, alkalmazás szerverekről, JVM-ből is elérhetnek.

1.4. JAX-RS

JAX-RS egy Java programozási nyelven alapuló API, mely segítségével REST architektúrában íródó alkalmazások fejlesztését segíti.

A REST (Representational State Transfer) a szerver-kliens architektúrát támogatja, ahol a kliensek kéréseket küldenek, a szerverek pedig válaszolnak egy-egy kérés feldolgozását követően. Architekturális stílus, ami különböző megkötéseket definiál, mint például a teljesítmény, méretezhetőség, módosíthatóság, annak érdekében, hogy a webszolgáltatás a leghatékonyabban működjön. Az alapelvek biztosítják a RESTful alkalmazásokat, hogy azok egyszerűek, gyorsak és könnyűek legyenek. Az URI-n keresztüli erőforrás azonosítás globális címzést biztosít az erőforrások, és a szolgáltatások számára. Az erőforrások négy alap művelettel kezelhetőek, ezek a létrehozás, olvasás, módosítás, törlés műveletek. A reprezentáció biztosítja, hogy az erőforrások különböző formátumokban jelenhessenek meg, mint XML, JSON, JPEG, szöveg formátumban.

Az erőforrás osztály megjelölése a @Path, vagy egy művelet végzési annotációval történik. A @Path annotáció attribútumaként megadhatjuk, hogy hol található az adott erőforrásosztály a szerveren, amely egy relatív URI útvonalat biztosít, emellett a különböző metódusok útvonalát is definiálhatjuk vele. @GET annotációval ellátott metódus feldolgozza a HTTP GET kéréseket, majd ennek megfelelően válaszol. A @POST, @PUT, @DELETE @HEAD annotációk hasonlóan határozzák meg a szerveren található metódusokat, melyek az annotációnak megfelelően fogadják a kéréseket. @Produces annotációval meghatározhatjuk, hogy a kliens milyen formátumban kérheti az adott erőforrást. Általában az XML és a JSON formátumot adjuk meg. A @Consumes annotációval a klienstől érkező erőforrások formátumait definiálhatjuk. Egy metódust paraméterátadással is definiálhatunk, ekkor az útvonal definíciójában kapcsos zárójelek között tüntetjük fel a megkapni kívánt paramétert, majd a paraméterlistában @PathParam annotációval határozzuk meg.

A @Path annotáció meghatározza az útvonalat melyen elérjük az erőforrást, és annak válaszát. Értéke egy részleges URI elérési utat biztosít a telepített szerver alap

program lista 1.2. Példa szolgáltatásra

```
1 @PUT
2 @Path("{id}")
3 @Consumes({MediaType.APPLICATION_JSON})
4 public void edit(@PathParam("id") Integer id, User
    ↪ entity) {
5         super.edit(entity);
6 }
```

útvonalához. Az URI útvonalak változókat is tartalmazhatnak, melyek futási időben kerülnek behelyettesítésre, annak érdekében, hogy egy adott metódus dolgozni tudjon velük. Ezeket a változókat fontos, hogy kapcsos zárójelek között adjuk meg: @Path("login/{name}"). Az URI változót megszabhatjuk regex definiálásával, ami alapból a következőnek felel meg: "[^/]+?". Ezáltal a paraméterben átadott érték karakterei között szereplő "/" karakter figyelmen kívül lesz hagyva. Saját regex megadását a következő szintaxis alapján tehetjük meg: @Path("login/{változó: regex}"), abban az esetben, ha a kérésben érkező paraméter nem egyezik a megadott formátummal, akkor 404-es hibakód választ fogja kapni a kliens. Több változó megadása esetén minden egyes változónak kapcsos zárójelek között kell, hogy szerepeljen, például: @Path("login/{name}/{password}").

NetBeans fejlesztői környezet segítségével gyorsan generálhatunk REST szolgáltatásokat. A webprojektünk nevére jobb egérgombbal kattintva válasszuk ki a "New", azon belül a "RESTful Web Services from Entity Classes" varázslóját. Itt kiválaszthatjuk mely perzisztens osztályokból szeretnénk szolgáltatást generálni, majd a célhely megadását követően legenerálja számunkra ezeket az osztályokat. Első alkalommal létrejön egy absztrakt ősosztály is, amit minden szolgáltatásosztály örököl. Az ősosztály eltárolja a gyerekosztályhoz tartozó entitás osztályát, ezáltal képes kezelni a perzisztencia műveleteket. Az alapművelet metódusokat öröklik meg a szolgáltatásosztályok, melyeket kiegészítenek az elérési útvonallal, az elfogadott médiatípussal, a megfelelő REST műveleti operátorral. Emellett létrejön egy alkalmazáskonfigurációs osztály részleges URI elérési útvonallal, ebben az osztályban minden szolgáltatásosztály egy listában eltárolásra kerül.

Miután elkészült a szolgáltatásunk, generálhatunk kliens osztályt, akár egy másik projekten belül. A varázslót megnyitva megadhatjuk az osztályunk nevét, célhelyét, a hozzákapcsolódó REST szolgáltatásosztályt. Az így létrejött osztály tartalmazza az összes hívható metódust a megfelelő médiatípussal, eltárolja az alap elérési útvonalat, később ehhez lesz hozzáfűzve az elérni kívánt osztály és az osztályhoz tartozó metódus részútvonala. Ezt követően már hívhatjuk az adott kliens metódusait, hogy kapcsolatba lépjen a kívánt szolgáltatással.

1.5. JavaServer Faces

JavaServer Faces technológia a webalkalmazások felületeinek fejlesztésére szolgáló keretrendszer. Tartalmaz egy API-t a komponensek megjelenítéshez, állapotuk és események kezeléséhez, szerveroldali érvényesítéshez, oldalak közötti navigáció definiálásához. Továbbá címkekönyvtárakat, melyek komponenseket biztosítanak a weblapokhoz és kapcsolatot a szerveroldali objektumok eléréséhez.

JavaServer Faces amellett, hogy jól definiált programozási modellt biztosít, különböző címkézési könyvtárak segítségével egyszerűsíti le egy weblap fejlesztését. Weboldal komponenseinek hozzáadása címkék definiálásával történik, lehetőségünk van az oldalon szereplő adatokat összekötnünk a szerveroldali adatokkal, komponensek által generált események összeköthetőek a szerveroldali alkalmazásokkal, szolgáltatásokkal. Testre szabott komponensek újrafelhasználhatóak és kiterjeszthetőek, emellett az alkalmazás állapota menthető majd visszaállítható a szerverkérések élettartamának megszűnését követően.

Egy JavaServer Faces alkalmazás magába foglalja azokat a weblapokat melyeken komponensek vannak elhelyezve, címkéket melyeket a weblapokhoz rendelünk hozzá, a kezelt bean-eket, azaz a "managed beans"-eket, amik az alap szolgáltatásokat tartalmazzák, mint például az erőforrás injektálás, életciklus visszahívás, elfogadás. Az alkalmazás tartalmaz egy web.xml fájlt a telepítési leírások számára, továbbá tartalmazhat több erőforrás konfigurációs fájt is, melyek meghatározhatják az oldalak közötti navigációs szabályokat, konfigurációs bean-eket, egyéb objektumokat, komponenseket. Egyedi objektumok között szerepelhetnek olyan fejlesztő által létrehozott eszközök, mint az érvényesítők, konvertálók, figyelők. A kliens kérését követően a webkonténer legenerálja a JavaServer Faces által megvalósított oldalt válaszként, mely HTML vagy XHTML formában jelenik meg.

Technológia képes webalkalmazások esetében kettéválasztani a viselkedést a nézetettől. Az alkalmazás képes összekötni a HTTP kéréseket a komponens specifikus eseménykezelővel és a komponenseket úgy kezelni, mint az állapot teljes objektumokat a szerveren. Logikai réteg és a megjelenítési réteg kettéválasztásának hatására a fejlesztők párhuzamosan tudnak dolgozni két különböző komponensen, mint ahogy a weboldalak szerkesztői akár programozói tudás nélkül képesek létrehozni egy-egy weblapot komponensek segítségével, míg a szkripteket pedig elegendő behivatkozni szerveroldalról.

JavaServer Faces API rétegezését tekintve a Java Servlet API tetejére került, ez lehetővé tesz néhány fontos tulajdonságot, mint például a különböző megjelenítési technológiák használatát, saját összetevők létrehozását az összetevőosztályokból, különböző kliens eszközök számára kimenet generálást. JavaServer Faces 2.0 már tartalmazza a Facelets technológiát, ami a JavaServer Faces technológián alapuló webalkalmazások preferált megjelenítési technológiája lett. Facelets technológia sablon és az összetett

komponensek által a kód újrahasználható és kiterjeszthető a komponensek számára, a JavaServer Faces annotáció fejlesztését használva automatikusan hozzákötjük a kezelt bean-t, mint egy erőforrást az alkalmazáshoz. Ezenfelül az oldal navigációját tekintve, a gyors konfigurációt az implicit navigációs szabályok teszik lehetővé. Ezek a fejlesztések az alkalmazás gyors konfigurálhatóságát biztosítja. Emellett a JavaServer Faces sokoldalú architektúrát tesz elérhetővé számunkra, mely képes kezelni a komponensek állapotát, feldolgozza a komponens adatokat, felhasználói bemeneteket érvényesíti, kezeli az eseményeket.

Egy web alkalmazás életciklusa alatt több feladat van kezelve, beleértve a bejövő kéréseket, paraméterek dekódolását, állapot mentését és módosítását, böngésző számára az oldal generálását. A keretrendszerek egy csoportja elrejti a fejlesztő elől az életciklusok részleteit, míg egy másik csoportjának fontos, hogy a fejlesztő manuálisan kezelje őket. A JavaServer Faces tekintve az életciklusok nagy részét automatikusan kezeli, bár a kéréséletciklusának különböző szakaszait feltárja, ezáltal a fejlesztők kezelhetnek, vagy végrehajthatnak különböző műveleteket attól függően, hogy az alkalmazás hogyan követeli azt meg. Egy életciklus a kliens kérésével indul és a szerver által adott válasszal fejeződik be. Az életciklus két fő szakasza van, a végrehajtás és megjelenítés.

Végrehajtási fázisban számos művelet történhet, ilyen például az alkalmazás nézet felépítése vagy helyreállítása, kérési paraméterérték alkalmazása, konverzió és érvényesítés végrehajtása összetevő értékek esetében, kezelt bean-ben található értékek frissítése, alkalmazás logika meghívása. Az első kéréshez még csak a nézet épül fel, a későbbi kérések valósíthatják meg a többi feladatot.

Megjelenítési szakaszban a kért nézet megjelenik a kliens számára válaszként. A kimenet generálása általában HTML vagy XHTML formában történik amit egy böngésző kliens értelmezni tud.

1.5.1. Facelets

Facelets hatékony és könnyűsúlyú oldal deklarációs nyelv, melyet JavaServer Faces nézetek létrehozására használnak a HTML stílus sablonok és komponensfák létesítésének alkalmazásával.

Facelets XHTML-t használ a weblapok létrehozására, támogatja a Facelets címke könyvtárak mellett a JavaServer Faces és JSTL címke könyvtárakat is, továbbá a kifejezőnyelvet, azaz az "Expression Language"-et. Sablont biztosít a komponensek és az oldalak számára. Nagyméretű projektek esetén támogatja a kód újrafelhasználását sablonok és összetett komponenseken keresztül, funkcionális bővíthetőséget biztosít komponenseknek és más szerveroldali objektumoknak, emellett gyorsabb fordítási időt, fordítási időben végzett kifejezőnyelv ellenőrzést, magas minőségű megjelenítést biztosít. A Facelets használatával csökken a fejlesztésre és telepítésre szánt idő.

JavaServer Faces technológia különböző címke könyvtárakat támogat a weblaphoz felhasználható komponensek számára. Facelets azáltal támogatja ezeket a címke könyvtárakat, hogy XML névtér deklarációkat használ. Támogatott könyvtárak között megtaláljuk a JavaServer Faces HTML címke könyvtárát, melynek előtagja a "h:" és UIComponent objektumok számára tárol komponens címkéket, ilyen például a HTML dokumentum fejrészének meghatározására szolgáló <h:head> címke, vagy a törzs meghatározása a <h:body> címke. Szöveges tartalom kiíratására a <h:outputText>, míg bevitelre a <h:inputText> használható. A Core könyvtár előtagja az "f:", egyedi műveletek számára tárol címkéket, melyek függetlenek bármilyen megjelenítési készlettől, ide tartozik az <f:actionListener>, mely egy komponens számára biztosít egy figyelőt, és az <f:attribute> ami lehetőséget biztosít arra, hogy egy paramétert, vagy attribútumot átadjunk egy komponensnek a megfigyelőn keresztül. Ezek mellet lehetőségünk van saját előtagokkal ellátott címkéket létrehoznunk, az összetett komponensek támogatása által. Abból kifolyólag, hogy a JavaServer Faces támogatja a kifejezőnyelvet, lehetőségünk van összekötnünk a komponens objektumokat vagy értékeket a kezelt bean metódusival és a "getter", "setter" függvényeivel.

Az alkalmazás minden elérhető weboldala egy "Managed Bean"-hez van csatlakoztatva, itt találhatóak azok a metódusok, függvények melyek össze vannak kapcsolva a weboldal komponenseivel. Osztály neve előtt megadott @ManagedBean annotáció regisztrálja az adott osztályt kezelt bean-ként, majd az ezt követően megadott @SessionScoped annotáció munkamenetként regisztrálja az osztályt. Managed Bean alapértelmezetten paraméter nélküli konstruktor által jön létre és tartalmazhat olyan propertyket melyek hozzá vannak kötve egy komponens értékhez, példányhoz vagy konvertáló, figyelő, esetleg ellenőrző példányhoz. Általában komponens adatot érvényesít, komponens által kiváltott eseményeket kezeli és az alkalmazás által megnyitni kívánt oldalt hozza létre. Egy komponensnek értékül lehet adni primitív, numerikus, és bármilyen Java objektumtípust, melyhez biztosítva van egy konvertáló metódus, ami képes az adott típust String típussá konvertálni, majd vissza. Webalkalmazásomban ez főként a dátumok esetében jelent meg. A dátumokat long típusban tároltam az alkalmazáson belül, viszont megjeleníteni String típusként tudtam, egy dátum formátumot biztosítva az értéknek. Minden ilyen modell osztályban ahol a dátumot konvertálnom kellett, biztosítottam egy-egy "getter" metódust, ebben a SimpleDateFormat osztály segítségével a long típusú dátumot átformáztam, majd átkonvertáltam String típussá. Továbbá "setter" metódust is biztosítottam, ami a String típusként érkező paramétert a SimpleDateFormat osztállyal util.Date típussá konvertáltam, ezt követően meghívtam a Date osztály getTime metódusát, ami visszaadta ezredmásodpercekben a dátumot long típusként.

1.5.2. Expression Language

Expression Language (EL), azaz a kifejezésnyelv amely egyszerű szintaxist használva a webréteg felől dinamikusan hozzáfér az alkalmazáslogika által tárolt adatokhoz.

A kifejezésnyelv biztosítja a kifejezések késői és azonnali kiértékelését, "getter" "setter" metódusok használatát, emellett lehetőségünk van különböző függvények, metódusok meghívására is. Azonnali kiértékelés alatt azt értjük, hogy az oldal létrejöttekor értékelődik ki a kifejezés és ezzel együtt az értéket is megkapjuk, míg a késői kiértékelés az oldal életciklusa alatt bármikor megtörténhet mikor az aktuálissá válik. Azonnali kiértékelés szintaxis a következő: ${ }, a késői kiértékelés pedig a #{ } szintaxist használja.

Az azonnali kiértékelés egy olyan értéket ad vissza amely csak olvasható, ezáltal sablon szövegek vagy címkék attribútumainak értékei lehetnek. Futásidőben nincs lehetőség ezeknek az értékeknek a megváltoztatására, és megjelenítésére. Késői kiértékelés az oldal életciklusának bármelyik szakaszában kiértékelődhet, attól függően hogyan definiáltuk azt. Webalkalmazásom egyed keresési oldalán a <h:inputText value="#{animalController.searchedEarTag}"> bemeneti mező "value" attribútumának beállított kifejezés az animalController ManagedBean osztály searchedEarTag property-jét adjuk át. Ezáltal ha van "getter" property-je az oldal létrejöttekor, vagy újratöltésekor megjelenik a korábban beállított mezőértéke, az űrlap elküldésekor pedig a "setter" metóduson keresztül az input-ban megadott érték kerül beállításra.

Kifejezéseket tekintve megkülönböztethetünk érték és metódus szerinti kifejezéseket. Az érték szerinti megjeleníthet és beállíthat értékeket, míg a metódus szerinti referenciát adhat metódusra, amit később hívhatunk és akár visszatérési értéket is fogadhatunk tőle. Érték szerinti kifejezés adhat egy objektum attribútumára vagy property-jére referenciát, ehhez az objektum nevét kell használnunk. A webkonténer kiértékeli a kifejezést, majd megkeresi a feltüntetett objektumot az oldalon, a munkamenetben és az alkalmazás hatóköreiben, végül visszaadja a keresett értéket. Abban az esetben, ha nem talált névnek megfelelő objektumot "null"-al tér vissza. Egy enumeráció értékének vizsgálatát String literál segítségével tehetjük meg, kiértékeléskor enumeráció típusúvá fog konvertálódni. A . vagy [ ] jelölések segítségével hivatkozhatunk Bean vagy enumeráció property-jeire, tömbök, listák elemeire, webalkalmazásomban az egyed azonosítószámának property-jét meghívhatjuk #{animal.earTag} módon vagy #{animal["earTag"]} módon, ahol a zárójelek közötti rész egy String literál, mely a property-t azonosítja. Lista vagy tömb egy elemének eléréséhez olyan literált vagy metódust kell használnunk ami int típusúvá konvertálható. Egy tömb elemének lekérdezését megadhatjuk #{controller.array[1]} módon vagy #{controller.array.counter} módon, ahol a counter int visszatérési értéket biztosít. A "csak olvasható" kifejezésként megadhatunk literálokat, vagy egy property-vel végzet aritmetikai művelet is, mely az oldalon értékelődik

ki és jeleníti meg a kívánt eredményt. Literálok lehetnek logikai, Integer, lebegőpontos, String és null típusúak. Az érték szerinti megjelenítést statikus szövegekben, és olyan címkék attribútumaként használhatjuk melyek képesek feldolgozni egy kifejezést.

Metódus szerinti kifejezések hívhatnak olyan tetszőleges metódusokat melyek valamilyen értékkel térnek vissza. Komponens címkék a metódus kifejezések révén hívhatnak olyan metódusokat melyek segítségével bizonyos műveletek feldolgozása biztosítva lesz. Mivel egy metódus bármikor hívható az életciklus alatt, ezért ez a módszer a késői kiértékelési szintaxist részesíti előnyben. Ahogy az érték szerinti kifejezések esetében, itt is a . és [ ] jelölések által határozható meg egy-egy metódus. Kifejezés megadásakor figyelnünk kell arra, hogy egy címke attribútumában csak egy kifejezés állhat, továbbá pontosan definiálnia kell a JavaBean-t és a hívni kívánt metódust. Metódus hívásakor paraméterlistában is lehetőségünk van átadni az értéket, ekkor az átadni kívánt paramétereket ( ) zárójelek között tüntetjük fel, a kifejezésvégén.

Csak olvasható kifejezések írásakor használhatunk különböző operátorokat is. A támogatott operátorok között megtalálhatjuk az aritmetikai, logikai, relációs, "üres", azaz az empty és feltételes operátorokat. XML dokumentum formázását tekintve vannak karakterek melyeket ki kell váltanunk a vele megfelelő kulcsszóval, a dokumentum fordításakor ezek a karakterek valamilyen szerepet viselnek, ezáltal hiba léphet fel. A < karakter egy címke, tag kezdetét jelzi, ez miatt a < entitással helyettesítendő, ehhez hasonlóan a > karakter a címke végét jelöli, melyet a > entitással helyettesíthető. Logikai operátort tekintve az & egy entitás hivatkozás kezdetét jelölheti, ezáltal a & entitással váltható ki. A kifejezőnyelv biztosít kulcsszavakat a különböző operátorokra, ilyenek például az and, or, not, eq mely az "=="-et jelöli, ne ami az "!="-et azonosítja, < az lt, míg > a gt számára fenntartott kulcsszó. Fontos, hogy ezek a kulcsszavak azonosításra nem használhatóak, csak a megfelelő operátor műveletek elvégzésére alkalmasak.

1.6. WildFly

A WildFly alkalmazásszerver támogatja a Java EE és Jakarta EE platformokat, emellett a JBoss alkalmazásszerverek széria nyílt forráskódú kiadása.

Vállalati alkalmazások fejlesztését azáltal egyszerűsíti le, hogy a Java EE és Jakarta EE platformok technológiái közül több elérhető a konfigurációk között. Tartalmazza a Java Database Connectivity (JDBC) driver beállítását, mely azt teszi lehetővé, hogy a driver feltöltését követően lehetőségünk van hozzáférnünk az adatbázishoz. Miután beállítottunk legalább egy adatbázis-kapcsolatot biztosító driver-t, megadhatunk adatforrásokat az adatbázisok kiválasztását és az ahhoz szükséges kapcsolódási információk megadásával. Az adatforrásoknak biztosíthatunk neveket a JNDI által, melyekkel később a webalkalmazás fejlesztése közben elérhetjük ezeket a forrásokat. Továbbá olyan

alrendszerek tartalmaz, mint például az JAX-RS, ami a JAX-RS alkalmazások telepítését, funkcionalitását teszi lehetővé, ehhez a WildFly a RESTEasy-t használja. Az alrendszerek között megtalálható az EJB konténer, ezzel biztosítva azt, hogy távoli szolgáltatásokat telepítsünk az alkalmazásszerverre. Három típusa található meg az EJB-nek, a Session bean, az üzenet vezérelt bean és a entitás bean. JPA alrendszer kezeli a JPA konténer által kezelt követelményeket és lehetőséget ad arra, hogy perzisztencia egységek telepítésére szolgáló definíciókat, annotációkat és leírásokat futtassunk.

A WildFly alkalmazás szerver "managed domain" vagy "standalone server" módban indítható. Managed domain lehetővé teszi a többszerveres topológia futtatását, és kezelését, míg a standalone server mód egyetlen szerver kezelésére alkalmas.

Legtöbb esetben a "managed domain"-en keresztül elérhető központi irányítási lehetőség nem szükséges, ezekben az esetekben futtatható a WildFly példány "standalone" módban. A példány elindításához a "standalone.sh" vagy a "standalone.bat" szkripteket használhatjuk. Abban az esetben, ha több "standalone" szervert futtatunk, a felhasználónak maga kell gondoskodnia szerverek kezeléséről amit csak külön-külön tehet meg, egyszóval nincs lehetőség a több szerver együttes kezelésére. Azaz ha egy alkalmazást több szerveren szeretne futtatni, akkor egyesével minden szerver esetében szükséges telepítenie azt.

WildFly egyik legfontosabb fejlesztése, hogy egyetlen helyről képesek vagyunk irányítani az összes példányt. Domain Controller felelős azért, hogy minden egyes példány egy adott szabály szerint legyenek konfigurálva. Egy speciális Host Controller által az összes domain megosztható fizikai, vagy virtuális gépen. Host Controller példánya úgy van konfigurálva, hogy az központi Domain Controller-ként viselkedjen, emellett a Host Controller kapcsolat áll Domain Controller-el, hogy a kiszolgálón futó alkalmazás szerver példányok életciklusát kezelje. A "domain.sh" vagy "domain.bat" szkriptek futtatásával indul el a "managed domain" egy Host Controller-el és egy WildFly példánnyal együtt. Az egyik Host Controller-t úgy kell konfigurálni, hogy az Domain Controllerként viselkedjen.

A WildFly a szerverek konfigurációjához és kezeléséhez három módot biztosít, a szervereket kezelhetjük CLI-n keresztül, XML konfigurációs fájlok segítségével és webes felületen. Utóbbi a HTTP API végpont a kezelő kliens belépési pontja, ami a HTTP protokollra támaszkodik, ezáltal integrálódik a felügyeleti réteggel. Ez JSON kódolású protokollt és RPC stílusú API-t használ, hogy leírja és végrehajtsa a kezelési műveleteket "managed domain" vagy "standalone server" módú példányokon. A HTTP API végpont alapértelmezetten a 9990 porton fut, és kétféle kiszolgálásra ad lehetőséget. Egyik a kezelői felülethez való hozzáférés, amely a /console útvonalon érhető el, míg a másik a műveletek végrehajtásához biztosít útvonalat a /management helyen.

2. fejezet

Adatbázis bemutatása

Az állatnyilvántartó rendszer lényegi működéséhez szükséges adatokat a MySQL adatbázis-kezelő rendszer által tároltam és kezeltem. Az adatok tárolását megelőzően fontos volt megtervezni az adatbázist, biztosítva ezzel egy leírást az adatbázis felépítéséhez, az adatok közötti kapcsolatokhoz, és az adatredundancia elkerülése érdekében.

Az adatbázis alatt az integrált, logikailag összetartozó információk, adatok összességét és az adatok közötti kapcsolatok rendszerét értjük. Az adatbázisok kezeléséhez adatbázis-kezelő rendszert használunk, mely segítségével elvégezhetőek a fontosabb műveletek, az adatok rögzítése, tárolása, kezelése. Az adat, adatbázisok és az adatbázis-kezelő rendszerek, szoftverek együttes elnevezése az adatbázis rendszer. Az adatok lekérdezésének és feldolgozásának hatékonysága érdekében manapság relációs adatmodellt alkalmaznak, ami alatt azt érjük, hogy az adatok táblázatos formában, a táblázat oszlopaiban és soraiban jelennek meg.

A relációs adatmodell esetén egy táblázat és a táblázat soraiban tárolt adatok adják a relációt. A relációkat egyedi névvel határozzuk meg, az oszlopokban az adott típusnak megfelelő adatok jelenhetnek csak meg, továbbá az oszlopok, attribútum neveit a reláción belül egyediként kell meghatároznunk. A sorokban találhatóak a logikailag összetartozó adatok, viszont egy reláción belül két azonos sor nem kerülhet be. A sor és az oszlop metszéspontjában a táblázat egy mezője található, amely egy adatot tartalmaz. A relációkban található sorok adatszáma nem lehet se több se kevesebb, azaz minden sorában azonos mennyiségű adatnak kell szerepelnie.

Az adatbázis szoftverek segítik az adatbázis rekordok és fájlok kezelését, létrehozását, módosítását, lehetővé teszik a rekordok egyszerűbb létrehozását, adatok bevitelét, szerkesztését, frissítését és a jelentések készítését. A szoftver továbbá biztosítja az adatok tárolását, biztonsági mentéseket, jelentéseket, a többszörös hozzáférés-vezérlést és a biztonságot. Az erős adatbázis biztonság különösen fontos manapság, mivel gyakoribbá váltak az adatlopások. Az adatbázis szoftverekre gyakran hivatkoznak adatbázis-kezelő rendszerekként is. Ezek a szoftverek az által teszik egyszerűbbé az adatokhoz való hozzáférést és az adatok kezelését, hogy strukturált formában tárolják az adatokat.

Általában grafikus felülettel rendelkeznek, ezzel téve könnyebbé az adatok beszúrását, kezelését a felhasználók számára.

Az adatbázis-kezelő rendszerek interfészként szolgálnak az adatbázis és a végfelhasználók, vagy a programok között, mely lehetővé teszi az információk rendszerezésének, optimalizálásának lekérését, frissítését és kezelését. Továbbá megkönnyíti az adatbázisok felügyeletét és kezelését különböző adminisztratív műveletek biztosításával, mint például a teljesítményfigyeléssel, hangolással, biztonsági mentésekkel és a helyreállíthatósággal. Ilyen adatbázis-kezelő rendszer például a MySQL is.

A MySQL egy SQL (strukturált lekérdezőnyelv) alapú nyílt forráskódú relációs adatbázis-kezelő rendszer. Webalkalmazásokhoz tervezték és optimalizálták és bármilyen platformon futtatható. A MySQL számos előnyt nyújt, ezért több kis és nagy vállalat választja az adataik tárolására és kezeléséhez. Képes kezelni az erőteljes adatbázis csomagok nagy részhalmazát, a legtöbb operációs rendszeren futtatható különböző programozási nyelvek használata mellett, mint például PHP, C, C++, Java. Emellett a nagy adatmennyiségekkel gyorsan és az elvártnak megfelelően dolgozik. A tárolt adatok mennyisége hatalmas lehet, a beállításokat tekintve egy tábla maximális mérete kezdetben 4 Gb, amit megváltoztathatunk annak megfelelően, hogy az adott operációs rendszer és hardver képes-e kezelni, 8 terabyte-ra is akár. A szabad szoftver licenc lehetővé teszi, hogy a fejlesztők testre szabják a MySQL kódját annak érdekében, hogy az általuk használt környezethez igazítsák.

2.1. Táblák bemutatása

A táblák tervezése során fontos szempont volt az adatredundancia elkerülése, ezáltal figyelembe kellett vennem a normalizálást. A nem megfelelően megtervezett adatbázisokban ellentmondások és anomáliák jöhetnek létre. A normalizálás alkalmazásával oly módon rendszerezzük és határozzuk meg az adatbázisban lévő adatok helyét és kapcsolatait, hogy azok kisebb mértékben okozzanak redundanciát és inkonzisztens függőségeket. Redundáns adatok tárolásával memória területeket pazarlunk el, emellett bizonyos anomáliák adódhatnak. Ilyen anomália a módosítási anomália is, ami akkor következik be, mikor egy adott relációban bizonyos adatok több sorban is szerepelnek és mellettük csak néhány adat más-más soronként. Abban az esetben, ha több sorban ismétlődő információ értékét szeretnénk megváltoztatni, akkor figyelnünk kell arra, hogy ezt minden egyes sorban megtegyük, különben lekérdezéskor nem megegyező értékeket kaphatunk. Ehhez hasonló a törlési anomália, amikor arról beszélünk, hogy egy adott elem törlésével olyan adatok is elveszhetnek melyek alap esetben nem tartoznának bele a törlésbe, ilyen például az, mikor egy filmhez tartozó színész törlésével maga a film is törlésre kerül. A beszúrási anomália esetén azt vehetjük észre, hogy mikor egy új elemet kívánunk beszúrni egy bizonyos értékhez, akkor ennek az értéknek

a nevét, azonosítóját, azaz minden változatlan attribútumát helyesen kell felvinnünk, különben előfordulhat az, hogy egy azonosítóhoz más-más értékek tartoznak ami ellentmondáshoz vezethet.

A normálformákat tekintve egy reláció akkor van első normálformában, ha minden tulajdonsága elemi és nem tartalmaz adatcsoportokat. Ez alatt azt értjük, hogy egy mezőben nem tárolunk több értéket egyben, ehelyett kiszervezzük több mezőre ezeket az értékeket. Egy cím esetén úgy járhatunk el, hogy az irányítószámnak, a városnevének, az utca és a házszámnak külön-külön mezőket biztosítunk ahelyett, hogy ezeket ömlesztve egyetlen mezőben tárolnánk el, amelyből később nehezen tudnánk kinyerni bizonyos információkat. Abban az esetben, ha egy felhasználónak több telefonszáma van, akkor azokat nem tároljuk el egyetlen rekordban, helyette annyi rekordot viszünk fel ahány telefonszámot elszeretnénk tárolni. Továbbá fontos betartani a relációs adatmodell feltételeit, azaz egy rekord nem ismétlődhet a reláción belül, minden rekord esetén a mezők száma és sorrendje megegyezik.

A második normálforma akkor teljesülhet, ha a reláció első normálformában van, emellett a reláció minden másodlagos attribútuma teljesen függ a kulcstól, azaz teljes funkcionális függésben van. Az összetett kulcs több attribútumból áll, ezeket elsődleges attribútumoknak nevezzük, míg a többi attribútumot másodlagosnak. A teljes funkcionális függést úgy érhetjük el, ha a másodlagos attribútumok csak a kulcstól függnek, emellett az összetett kulcsot alkotó összes elsődleges attribútumtól függ minden másodlagos attribútum.

A harmadik normálforma akkor teljesül, ha az adatbázis minden relációja legalább második normálformában van és a relációkon belül nincs tranzitív függőség. Tranzitív függés alatt azt értjük, mikor bizonyos mezők nem csak a kulcstól, hanem más attribútumoktól is függenek. Ilyen a például a rendelések táblában található irányítószám és városneve mezők is, a rendelés azonosító meghatározza honnan rendeltek, de emellett az irányítószám szintén meghatározza a városnevét. A probléma megoldásához és a redundancia csökkentéséhez ezeket a mezőket külön táblába szervezzük, majd az eredeti táblában egy idegen kulcs formájában tüntetjük fel a kapcsolatot.

A normalizálást figyelembe véve készítettem el a webalkalmazáshoz tartozó adatbázist. Minden felhasználó és tartási hely esetén tárolnom kellett a pontos címüket is, amihez biztosítottam egy ország, megye és város táblát. Az ország táblát tekintve kettő attribútumot vettem fel, az első mező az elsődleges kulcsként funkcionáló ISO (International Organization for Standardization) kétbetűs kódot tartalmazza, mellyel egy-egy országot azonosíthatunk, míg a másik mező az országnevét határozza meg. Hasonlóan valósítottam meg a megye táblát, amiben szintén kettő mező található, a megyék nevét tartalmazó mező és az "id", azaz az azonosító mező, mely elsődleges kulcsként viselkedik. A városokat tartalmazó tábla abban különbözik, hogy a városokat meghatározó irányítószámokat nem nevezhettem ki elsődleges kulcsnak, mivel előfordulnak

olyan városok, községek, melyek esetén egy irányítószám több települést is meghatároz. Ezáltal ebben a táblában biztosítottam az irányítószám és a városnév mezők mellett egy azonosító mezőt is mely automatikusan növekszik és betölti az elsődleges kulcs szerepet.

A felhasználók a regisztrációt követően nem léphetnek be azonnal az alkalmazásba, minden egyes regisztrációt követően egy adminisztrátornak felül kell vizsgálnia az adott felhasználót, hogy meggyőződjön a megadott adatok és elérhetőségek helyességéről. Erre a célra biztosítottam egy elfogadási "tinyint(1)" típusú mezőt, mely értéke a regisztrációt követően 0, a hamis logikai érték reprezentációja, az elfogadást és a felülvizsgálatot követően pedig az 1 értéket fogja felvenni, ami az igaz logikai érték megfelelője. Amíg ez a mező érték 0, addig a rendszer bejelentkezéskor a "A felhasználó jelenleg még nincs felülvizsgálva" üzenetet fogja visszaadni. Továbbá a felhasználó bejelentkezést követően bármikor módosíthatja az adatait, viszont ezek helyességét is felül kell vizsgálnia egy adminisztrátornak, erre a célra egy hasonló mezőt hoztam létre ami az adatok módosítását követően fogja felvenni a hamis értéket. Az új mező létrehozását az indokolta, hogy a módosítást követően a felhasználó ne zárja ki magát a rendszerből. Egy felhasználóról tárolom a kereszt- és vezetéknevét, az elérhetőségét biztosítva egy telefonszámot és egy e-mail címet, mely az alkalmazásba való belépést is biztosítja a jelszó mezővel együtt, a lakhely meghatározásához idegenkulcsokat az ország, a megye és a város táblákhoz, és egy utca mezőt az utca és házszám tárolásához. Továbbá biztosítottam egy szerepkör mezőt a szerepkör meghatározásához.

A felhasználóhoz tartozó szerepkört, ahogy a város, a megye és az ország esetében egy külön táblát hoztam létre, amit idegenkulccsal kötöttem össze, ezzel biztosítva azt, hogy több szerepkör jöhessen létre az anomáliák elkerülésével. A regisztrációt követően minden személy a felhasználói szerepkört kapja meg, amit később egy adminisztrátor a regisztráció elfogadásakor, esetleg az elfogadást követően bármikor módosíthatja azt. Jelenleg az alkalmazásban három fő szerepkör van, a már előbb említett felhasználói szerepkör, ami az alap funkciók használatához biztosít hozzáférést, ilyen például a saját egyedek, tenyészetek, tartási helyek listázása, módosítása de emellett rá kereshetnek más egyedek adatlapjára is. Az állatorvosi szerepkör lehetővé teszi a felhasználó számára, a hozzá tartozó megyék általi listázást mind az egyedekre, tenyészetekre, tartási helyekre nézve. Emellett azt is biztosítja, hogy ezeket az adatokat első rögzítésükkor ők is elfogadhassák, és az esetleges hiba esetén értesíthessék a hozzátartozó felhasználót. Továbbá az egyedeket tekintve módosíthatják az egyedhez tartozó betegségek listáját beleértve a hozzáadást és a törlést is. A harmadik szerepkör az adminisztrátor, akik az egyedek, tenyészetek, tartási helyek mellett a felhasználókat is kezelhetik. Amellett, hogy meghatározzák egy-egy felhasználó szerepkörét, az állatorvosok esetén azt is megadhatják, hogy az adott állatorvoshoz mely megyék tartozzanak. Az állatorvosokhoz tartozó megyéket egy kapcsolótábla segítségével valósítottam meg, ezáltal

egy állatorvoshoz több megye is tartozhat. A kapcsolótáblában található egy azonosító mező, ami elsődleges kulcs és a kapcsolatok azonosítására szolgál. Ezenfelül maga az idegenkulcsként szereplő felhasználóazonosító és a megyeazonosító mezők, melyek együttesen megkapták az egyedi megkötést, ezáltal nem fordulhat elő az, hogy egy orvost ugyanahhoz a megyéhez többször hozzárendeljük.

Az állatok nyilvántartása mellet fontos szempont volt egy belső üzenetküldő rendszer megalkotása is. Ezáltal az alkalmazást használó személyek üzenetet küldhetnek egymásnak, kérdezhetnek egymástól, állatorvosoktól, adminisztrátoroktól, míg az állatorvosok és az adminisztrátorok figyelmeztethetik, esetleg értesíthetik az adott felhasználót. A rendszer által küldött üzenetek is ennek segítségével jut el a felhasználókhoz. Egy üzenetet a megírást és az elküldést követően az "üzenetek" kapcsolótáblában tároljuk el. A kapcsolótáblában található egy idegenkulcs mező a feladó azonosítására, és egy a címzett azonosítására. Külön-külön mezőben eltároljuk a levél tárgyát és törzsét, emellett egy "bigint" típusú adatmezőben elmentjük azt a dátumot mikor az üzenet elküldésre került, ezáltal a címzett felhasználó információt kap az üzenet elkészülésének időpontjáról amit a többi üzenettel együtt időrendi sorrendben kerül listázásra. Az üzenetek megkülönböztetjük olvasott és olvasatlan jelző által, amit egy "tinyint(1)" típusú, "új" adatmezőben kerül eltárolásra.

Az állattartók tartási helyeken alakíthatnak ki tenyészeteket, ahol később állatokat tarthatnak. A tartási helyek felvételét szintén az állattartóknak kell kezdeményezniük, melyet később az illetékes állatorvos, esetleg egy adminisztrátor elfogadhat. Egy tartási hely rögzítésekor fontos meghatároznunk a címét, ami az ország, megye és a város idegenkulcs mezők által valósul meg, beleértve az utca mezőt is. Emellett fontos kritérium volt az is, hogy egy tartási helyhez tartozó tenyészetek típusának megegyezőnek kellett lenniük vagy éppen kivételbe kellett tartozniuk. Kivételeket tekintve hármat különböztetettem meg, az állattartó típusú tenyészet mellé keltető típusú tenyészet, a piac típusúhoz a rakodóhely típusú tenyészetek, míg az etető, itató hely típusú tenyészetekhez a pihentető típusú tenyészetek tartozhatnak. A tartási helyhez rendelhető tenyészet típusok meghatározásához a táblában egy "tenyészet típusa" mezőt biztosítottam és minden felvételkor a felvinni kívánt tenyészet típusa ezzel lesz összehasonlítva. Létrehoztam egy "kapcsolattartó felhasználó" idegenkulcs mezőt, mellyel lehetőségünk van azonosítani a tartási hely kérvényezőjét, és felmerülő probléma esetén ezt a felhasználót értesíthetik az állatorvosok, és az adminisztrátorok. A tartási helyet kezelő állatorvos számára is biztosítottam egy idegenkulcs mezőt.

A tartási hely pontos azonosítása érdekében a felhasználónak meg kell adnia a hozzátartozó helyrajzi számokat is, melyet egy kapcsolótábla segítségével rögzítek az adatbázisban. A tábla tartalmazza az adott tartási hely azonosítóját idegenkulcsként, magát a helyrajzi számot, és a helységet, mely a várostábla egy idegenkulcsa. A tartási helyhez továbbá kapacitások is tartoznak, azaz olyan területek, építmények melyek az

állatok lakhelyeként, mozgástereként funkcionál. Ezeket a kapacitásokat szintén egy kapcsolótábla által rögzítjük a tartási helyhez, amiben fontos eltárolnunk magát a tartási hely azonosítóját, a kapacitástípusának azonosítóját, a kapacitásnak megfelelő méretet, és az adott kapacitás létesítésének dátumát. A kapacitástípus táblában példaként a következő kapacitások találhatóak meg: az istálló hasznos, a kifutó / karám, és a ketrec hasznos melyeknek területét m^2 -ként adhatjuk meg, továbbá a legelő aminek a nagyságát hektárban rögzíthetjük. Egy tartási hely esetén meg kell adnunk az ott folyó tenyészetek tevékenységeit, azaz a tenyészetek által tartott, vágott állatokat. Az állatfajokat tartalmazó táblát és a tartási helyeket szintén kapcsolótáblával kötöttem össze, amiben rögzítésre kerül a tartási helyek és az állatfajok azonosítói, ezenfelül egy új állatfaj rögzítése esetén meg kell adnunk a tevékenység kezdetének dátumát, és az állatfajhoz tartozó hasznosítás típusát. A hasznosítás alatt a haszonállat tartásának célját értjük, ami lehet tej, hús, tojás, máj általi hasznosítás az állatfajtól függően.

A tartási helyet követően a következő fontos reláció a tenyészet. Későbbiekben a tenyészetekhez lesznek kapcsolva az állatok és ezáltal lesznek nyomon követhetőek tartózkodásuk, és az állatokhoz tartozó aktuális állattartók. A tenyészet táblában tároljuk egy tenyészet nevét, melyet nem kötelező megadni, a tenyészet típusát, ami lehet az állattartó (árutermelő / kereskedő), vágóhíd, karantén, keltető, mesterséges termékenyítő állomás, piac (állatvásár), rakodó / gyűjtőállomás, Állategészségügyi intézmény, kiállítás, megsemmisítő, legelő, etető- / itató hely, pihentetőhely. Tároljuk még a tenyészet minősítését, ami egy tenyészet esetén lehet EU-nak bejelentett gyűjtőállomás (rakodó), kereskedőtelep, kistermelői élelmiszer termelési (vágási) engedéllyel rendelkező árutermelő és méhanyanevelő. Továbbá meg kell adnunk a tenyészet besorolását is, ami az önálló tenyészet vagy a megyei körzetbe sorolt tenyészet lehet. Ahogy már korábban eljártam, egy tenyészethez rendelhető típusokat, minősítéseket és besorolásokat a tenyészet táblában idegenkulcsként adtam meg. Egy tenyészethez tartozó felhasználót a "felhasználó tenyészete", míg a tartási helyhez tartozó tenyészetet a "tartási hely tenyészete" kapcsolótábla segítségével kapcsoltam össze az idegenkulcsok tárolásával.

Az utolsó fő reláció a haszonállatok tárolására szolgáló állategyed tábla. Minden egyes állatot egy maximum tizenkét számjegyű azonosítóval szükséges ellátnunk, mely elsődleges kulcsként is funkcionál az adattáblában. Az egyed rögzítésekor a felhasználó megadhatja az állat nevét, becenevét ami tetszőleges lehet, de nem szükséges. Ezzel lehetőséget adva arra, hogy listázáskor könnyebben beazonosíthassa az adott egyedet. Minden egyed esetén viszont szükséges megadnunk az egyed születési dátumát, az egyed nemét, továbbá az egyedfaját, fajtáját, színét melyeket idegenkulcsként tárolunk. Az egyed születési adatainak meghatározását tekintve megadhatjuk az egyed anya azonosítóját, abban az esetben ha ismerjük az anyát, de szükséges megadnunk az ellésmódját, mely az "ellésmódok" táblából kiválasztva lehet alapértelmezetten a "nincs információ", továbbá a "könnyű, segítség nélkül", a "könnyű, segítséggel", a "nehéz, szaktechnikai be-

avatkozás nélkül", a "nehéz, szaktechnikai beavatkozással" és a "műtéti beavatkozással". A születés történhetett ikerelléssel, amihez "tinyint" típusú mezőt biztosítottam logikai érték szerinti meghatározáshoz. Végül az állat születési súlyát kell megadnunk kg-ban értve.

Az alkalmazás működését tekintve az ellés közeledtére figyelmeztető funkció is beépítésre került. A funkció működéséhez biztosítottam egy mezőt az inszemináció napjának eltárolásához az állat táblán belül, az állatfajok táblában pedig egy mezőt az adott állatfaj vemhességi idejének meghatározásához. Az állatfajokat és a vemhességi idejüket tekintve a táblában szerepel a szarvasmarha melynek vemhességi ideje 285 nap, a sertés 115 nap vemhességi idővel, a juh és a kecsek 150 nap vemhességi idővel és a ló melynek vemhességi ideje 335 nap. A baromfi esetén ez az idő nem volt releváns, ezáltal itt a "null" érték került rögzítésre. A funkció működését tekintve az alkalmazásba történő bejelentkezést követően minden egyes egyed esetén az inszeminációs dátum (abban az esetben, ha nem "null") és az adott bejelentkezési napnak dátumát összevetve kiszámolja az ellésig hátra lévő időt és ha ez az idő kevesebb mint 10 nap, akkor a rendszer az üzenet küldést alkalmazva figyelmezteti a felhasználót arra, hogy az egyed hamarosan megellik, feltéve, ha még nem küldött ilyen figyelmeztető üzenetet korábban.

A szín és a fajta táblákat tekintve hasonlóan épülnek fel, mint az ország, a megye, a tenyészet típusa és a tenyészet besorolás táblák. Rendelkeznek egy azonosító mezővel, és az adott tulajdonság megnevezésre szolgáló mezővel. A szarvasmarha színeit tekintve választhatunk az alap "nincs információ" opción kívül például a feketetarka, vöröstarka, fekete-vörös, zsemle tarka, egyéb tarka színek közül, míg a fajtákat tekintve választhatunk a magyar szürke, magyar tarka, holstein-fríz, jersey, limousin közül. Minden egyed esetén fontos rögzítenünk a múltbéli vagy az éppen tartó betegséget, ezáltal információt kaphatunk az állapotáról, lehetséges szövődményeiről, gyógyulási folyamatáról. Az állat felvételekor, módosításakor az állattartó személy is megadhatja minden egyede estén az elszenvedett betegségeket, de természetesen maga az állatorvos is rögzítheti az egyed módosítása révén. A betegségek számára biztosítottam egy relációt, majd egy kapcsoló tábla segítségével összekötöttem az állattáblával. Ebben a kapcsolótáblában biztosítottam mezőt az idegenkulcsok számára, továbbá minden egyes betegség esetén szükséges megadni a betegség kezdetének dátumát és abban az esetben, ha a betegség már végett ért, rögzíthetjük a vég dátumot is. De előfordulhat az, hogy a betegség egy életen át tart vagy éppen még tart, ekkor nem tudjuk rögzíteni ezt a dátumot, ezért ennek a mezőnek az értéke lehet "null" is. Minden betegség esetén az orvosok, felhasználók hozzáfűzhetnek egy maximum 300 karakteres megjegyzést is az adott betegséghez.

Az állatok tenyészetbe kerülésének rögzítésére szintén biztosítottam egy kapcsolótáblát, ami a tenyészetet és az állatokat köti össze. Az egyed egy tenyészethez való rendelésekor szükséges megadnunk a tenyészetkód és az állat azonosítója mellett a te-

nyészetbe kerülés dátumát is, továbbá az egyed kikerülésekor a kikerülésdátumát is rögzítenünk kell, ami eddig a "null" értéket vette fel. A kapcsolótábla biztosítja minden egyed esetén a nyomon követhetőséget azáltal, hogy információt kapunk az adott egyed két dátum közötti tartózkodás helyéről, az adott tenyészet lekérdezését követően pedig megkaphatjuk a tenyészethez tartozó állattartót, a tenyészet típusát, besorolását, továbbá a hozzákapcsolódó tartási helyet, ami biztosítja a pontos címet és helyrajzi számokat.

3. fejezet

Kezelői felület ismertetése

Az alkalmazást kliens-szerver architektúrában készítettem el, amiben a felhasználók a kliens grafikus felhasználói felületét használva kéréseket küldenek a szerveren található szolgáltatások igénybevételéhez. A szerver feldolgozza a kéréseket, meghatározza és meghívja a meghívni kívánt szolgáltatást, a legtöbb esetben az "EntityManager" osztály segítségével a szolgáltatás kapcsolatba lép az adatbázissal, onnan adatot kérhet le és oda adatot vihet fel, és abban az esetben ha az adott szolgáltatásnak van visszatérőértéke, akkor azt továbbítja a kliens felé.

Az adatbázis minden táblájához biztosítottam egy perzisztencia osztályt a táblák leképezésének és hozzáférésének megvalósításához szerver oldalon. Ezen perzisztencia osztályokon belül a @NamedQuery annotáció által az osztályhoz tartozó névvel beazonosítható lekérdezéseket definiáltam, oly módon, hogy egy adott táblából mely adatokat kellett szükségszerűen kinyernünk, beleértve az összetett lekérdezéseket is. A perzisztencia osztályokhoz generáltam REST szolgáltatás osztályokat, amiben a már korábban létrehozott perzisztencia osztálybeli nevesített lekérdezéseket felhasználva deklaráltam a függvényeket. A függvények rendelkeznek egy elérési úttal, a kliensek ezen keresztül érhetik el az adott szolgáltatást, rendelkezhetnek elérési útvonalban átadott paraméterrel, és visszatérési értékkel.

A kliens oldalon a szerveren található szolgáltatás osztályokhoz generáltam kliens osztályokat. A kliens osztály tartalmaz egy alap elérési útvonalat a szerver eléréséhez, amit később a konstruktor által kibővít az adott osztálynak megfelelő elérési útvonallal, ehhez fűzi hozzá a kívánt függvény meghívásához szükséges elérési utat az adott függvény meghívásakor. A szervertől visszatérési értékként kapott osztályok és a szerver felé elküldött osztályok feldolgozásához és fogadásához a szerveren szereplő perzisztencia osztályoknak megfelelő modellosztályokat hoztam létre. Az adatbázisba új rekordként felvitt objektumot előbb a kliens oldalon példányosítom, majd a grafikus felhasználói felület által, esetleg kódból a példány tulajdonságait beállítom ami végül elküldésre kerül a szerver felé, ahol megtörténik az adatbázisban való rögzítés. A grafikus felhasználói felületet az xhtml weblapok alkotják melyeket a JavaServer Faces és a Facelets

technológiák segítségével hoztam létre. A weboldalakon szereplő adatok kezelése, megjelenítése, a különböző elemekhez kapcsolódó függvények, eljárások biztosítása a kontroller osztályok feladata. A kontrollerek kötik össze a weboldalt és a klienst egy-egy funkció megfelelő működése érdekében.

program lista 3.1. Példa kliens osztály használatára

```
1 AnimalClient client = new AnimalClient();
2 AnimalModel animal = client.find_JSON(AnimalModel.
    ↪ class, earTag);
3 client.close();
```

3.1. Bejelentkezés

A webalkalmazás oldalát felkeresve a bejelentkező oldalra kerülünk, ahol lehetőségünk van a regisztráció oldalára továbblépnünk ha esetleg még nem lenne felhasználói fiókunk. Regisztrációkor meg kell adnunk a vezeték- és keresztnevünket, egy telefonszámot, egy e-mail címet és egy jelszót a későbbi bejelentkezéshez, a lakhely megadásához szükséges kiválasztanunk a legördülőlistákból az országot, a megyét és az irányítószám beírását követően egy újabb legördülő listából választhatjuk ki a lakcímünkhöz tartozó települést, abban az esetben, ha az adott irányítószám több települést határoz meg. Az utolsó beviteli mezőben az utcát adhatjuk meg. A regisztráció oldalt tekintve, a beviteli mezők felett elhelyeztem egy-egy <h:message> címkét és a "for" attribútumának beállítottam az adott beviteli mező azonosítójának értékét, míg a "style" attribútumának a piros színt adtam meg a figyelem felkeltés érdekében. A címke segítségével visszajelzést küldhetünk esetleges hiányosságokról, helytelenül megadott adatokról a felhasználó számára. A legördülő listákat a <h:selectOneMenu> címke által implementáltam az "id", "styleClass", "value" attribútumok megadásával. Az "id" biztosít számára egy azonosítót, a "styleClass" egy stílusosztályt a beimportált "css" fájlból, a "value" pedig "@ManagedBean" annotációval ellátott kontroller osztály egy mezőjének "getter" és "setter" nevét, ezáltal megkapjuk a jelenleg beállított értékét, míg más opció választása esetén a kiválasztott értéket veszi fel. A <h:selectOneMenu> címkék között megadhatjuk a legördülő lista elemeit, akár egyesével a <f:selectItem> címke többszöri felhasználásával, de lehetőségünk van a kontroller osztály egy listát visszaadó függvényét behivatkoznunk <f:selectItems> címke "value" attribútumában. A különböző értékek betöltésére az előbb említett <f:selectItems> címkét implementáltam, amelyben a "value" értékeként az adott mezőértékének megfelelően adtam át, az országokat, a városokat és a településeket. A beállított listákhoz a "var" attribútummal egy-egy változó nevet kellett biztosítanom, ezzel meghatározhattam minden egyes lista érték esetén a felhasználó által látott értéket az "itemLabel" és a működéshez fontos

felvehető értéket az "itemValue" attribútumok által. Mind az országok, megyék és a városok esetén a látható értékük a nevük, míg a háttérben beállított értékük az azonosítójuk volt.

program lista 3.2. Példa legördülőlistára

```
1 <h:selectOneMenu id="cities" styleClass="selectmenu"
    ↪ value="#{registerController.cityId}">
2
3         <f:selectItems value="#{registerController.
              ↪ cities}" var="ci" itemLabel="#{ci.name}"
              ↪  itemValue="#{ci.id}" />
4
5 </h:selectOneMenu>
```

A regisztráció gomb megnyomását követően, a vele összekötött regisztráció eljárás kerül meghívásra a kontrolleren belül. A lap elküldését követően meghívódnak a beviteli mezőkhöz kötött kontrollerosztálybeli mezők beállítási függvényei. Ezek a függvények megvizsgálják, hogy a beállítani kívánt érték megfelel-e az elvárásnak. Hibás adat esetén lekérjük a FacesContext aktuális példányát és egy üzenetet csatolunk hozzá a következő módon: "FacesContext.getCurrentInstance().addMessage(clientId, new FacesMessage("Hiba üzenete"));", ahol a "form"-on szereplő beviteli mezőhöz tartozó "message" címke beazonosítása, azaz a példában szereplő "clientId" megadása "form_id:bevitelimező_id" formában történik. Helyesen és hiánytalanul megadott adatok esetén a regisztráció eljáráson belül létrejövő új UserModel példány számára beállítódnak ezek az értékek, továbbá beállítjuk az elfogadási értékét hamisra, ezt követően UserClient osztály create_JSON metódusát meghívva rögzítjük az új felhasználót az adatbázisban a szerveren keresztül.

A sikeres regisztrációt követően megkísérelhetünk bejelentkezni a rendszerbe. Az e-mail cím és a jelszó beírását követően a bejelentkezés gomb megnyomásával meghívjuk a hozzákapcsolódó ellenőrző metódust. A metóduson belül először megvizsgáljuk, hogy mindkét mező kitöltésre került-e, a hiányzó adatról figyelmeztetjük a felhasználót. Ezt követően meghívjuk a szerveren található bejelentkezési adatokat megvizsgáló függvényt, ami egy nevesített lekérdezés alapján lekéri azt a felhasználót amelyhez az adatok tartoznak. A függvénynek háromféle visszatérési értéke lehet, az első, ha a lekérdezés nem járt sikerrel, ekkor a felhasználót a helytelen adatok megadására figyelmeztetjük. A második, ha helyesen adtuk meg a bejelentkezési adatokat, de a felhasználó még nem aktív, erről szintén tájékoztatjuk a felhasználót. A helyesen megadott adatok és az aktivált felhasználó esetén először szerver oldalon meghívjuk azt az eljárást, mely üzenetet küld a felhasználónak a 10 napon belül megellő állategyedéről és ezt követően visszatérési értékben tájékoztatjuk a kliens oldali kódot a bejelentkezésről. A kliens oldalon a HttpSession interfész által létrejön egy példány, ezzel biztosítva a felhasználó

bejelentkezésének megtartását. A példányhoz a felhasználó azonosítója és a szerepkör azonosítója attribútumként hozzáfűzésre kerül, ezeket az adatokat a bejelentkezés alatt bármikor lekérdezhetjük. Kijelentkezéskor ezt a Session-t érvénytelenítjük.

Sikeres bejelentkezést követően a felhasználó az üzenetek oldalra kerül átirányításra, ami az alkalmazás kezdőoldala, míg kijelentkezéskor és az olyan oldalak megnyitásakor melyekhez a szerepköréből adódóan nincs jogosultsága megtekinteni és hozzáférni, a bejelentkezési oldalra kerül visszairányításra.

3.2. Navigáció

Az alkalmazáson belül történő navigáció megvalósítása a navigációs kontrollerben és a hozzá kapcsolódó navigációs sávot megjelenítő oldalon került definiálásra.

A JavaServer Faces egy új oldal megnyitására két módszert biztosít, egyik a továbbítás (forward) a másik az átirányítás (redirect). Egy paraméterérték feldolgozását tekintve a továbbítási kérés megtartja ezt az értéket, míg az átirányítás nem tartja meg, mivel a továbbiakban a kérelem objektuma más lesz. Az átirányítás használatakor a HttpSession objektumba szükséges mentenünk a megtartani kívánt értékeket. A továbbítás folyamata nem befolyásolja a klienst, azaz a böngészőben található URL nem változik, míg az átirányítás során az új oldal címe kerül az URL címbe és a megnyitni kívánt oldal kerül betöltésre. Továbbításkor elegendő az oldal nevének megadása, míg az átirányításkor szükséges kibővítenünk ezt a "?faces-redirect=true" kifejezéssel.

A navigációs kontrollerben elhelyeztem kettő mezőt, a megnyitni kívánt oldal nevének tárolására szolgáló mezőt, az "activePageName"-et és a jelenleg bejelentkezett felhasználó szerepkörének azonosítóját visszaadó mezőt, a "roleId"-t. A navigációs kontrollerben található navigációt megvalósító függvények működését tekintve, a függvény meghívását követően először beállításra kerül az oldal neve az "activePageName" mező értékének és ezt követően visszaadásra kerül a String típusú oldal megnyitásához szükséges kifejezés. A felhasználó adatlapjának megtekintésekor a "account.xhtml?faces-redirect=true" kifejezés kerül visszaadásra, ezzel átirányítva minket a felhasználói adatlap nézetre.

A bejelentkezés és a regisztrációs oldalakon kívül minden oldal tetején megjelenítésre kerül a navigációs sáv. A navigációs sáv definiálást egy külön xhtml oldalon valósítottam meg, majd a <ui:include src="navigationbar.xhtml" /> címke által beépítettem az oldalakba. A beépíthetőséget figyelembe véve ezen az oldalon a html dokumentumot meghatározó címkék helyett csak a listát meghatározó <ul> címkét és a listaelemeket meghatározó <li> címkéket használtam. A navigációs sáv sorrendbeli felépítését tekintve először az üzenetek menüpontot hoztam létre, mely alatt lenyíló menüpontként az "Új üzenet" menüpont található. A mellette lévő lista elem a "Saját állatok" menüpont, amire a kurzort húzva megjelenik szintén lenyíló módban a te-

nyészet és a tartási hely menüpontjai. Az ezt követő lista elem az állatok keresésére szolgáló menüpont. A további két fő menüpont az engedélykérések és a listázás, melyek megjelenítését tekintve csak az adminisztrátorok és az állatorvosok számára történik meg. A navigációs sáv jobb oldalán a felhasználói fiók és a kilépés gomb jelenik meg. A formázást tekintve az adott menüpontra kattintva vagy az adott menüpont alatt lévő menüpontok egyikére kattintva a fő menüpont kiemelésre kerül, az "activePageName" mező értékének vizsgálatát követően.

3.3. Üzenetek

Az üzenetek oldalának felépítését tekintve, a html fejlécében elhelyezett title címkék között az alap "Üzenetek" cím mellett a webalkalmazás funkcióit tekintve egy plusz funkcióként az új üzenetek darabszámának megjelenítése is bekerült. A darabszámot tekintve, ha nincs új üzenet, akkor a cím nem változik, ellenben az új üzenetek darabszámának megfelelően "[+1]" formátumban jelenik meg a darabszám az "Üzenetek" cím jobb oldalán. Viszont mikor az új üzenetek száma meghaladja a 99-et, akkor onnantól kezdve a "[+99]" formátum lesz érvényben.

A kifejező nyelvet alkalmazva határoztam meg az új levelek darabszámát. A userHasMail kontroller newMailCount, azaz az új levelek számának visszaadását biztosító függvény meghívását követően ternáris operátort használva megvizsgálom hogy a visszaadott érték nagyobb-e mint 0 a ">" kulcsszót használva és kisebb-e mint 99 a "<" kulcsszót alkalmazva. Ha a két érték között szerepel az új levelek száma, akkor felépítem a string-et konkatenációval, hogy a korábban már ismertetett formátumot kapjuk. Abban az esetben viszont, mikor nem teljesül ez a feltétel egy üres string-et adunk vissza. Továbbá megvizsgálom szintén a ternáris operátorral, hogy ez az érték nagyobb-e mint 99, ha ez teljesül megjelenítésre kerül a "[+99]".

A html törzsében először meghívom a LoginController bejelentkezést ellenőrző függvényét, melynek paraméterként át kell adnom az adott oldal nevét és az adott oldal megtekintéséhez szükséges szerepkör azonosítóját. A függvény összehasonlítja az adott felhasználó és az elvárt szerepkör azonosítóját, abban az esetben ha a felhasználó nincs bejelentkezve a szerepkör azonosítója a Session-ben nem szerepel, vagy éppen ha be van jelentkezve, de a szerepköre nem elegendő a hozzáféréshez, akkor átirányítjuk őt a bejelentkezési oldalra. Megfelelő szerepkörrel viszont a paraméterben átadott oldalra irányítjuk vissza.

Az üzenetek röviden, egy-egy sorban listázom az üzenet küldőjével, tárgyával, dátumával és egy törlésre szolgáló gombbal. Az üzenet tartalmazó sorra kattintva a sor alatt megjelenítem az adott üzenet törzsét. A <c:forEach> címke "items" attribútumában megadom a userHasMail kontroller azon függvényét, mely a levélfogadója alapján visszaadja az üzeneteket, és a feldolgozáshoz a "var" attribútumban biztosítok egy

kulcsszót. A táblázat minden egyes sorában először lekérem az adott üzenethez tartozó küldő azonosítója alapján a küldő nevét, majd az üzenet tárgyát, és végül a long típusként szereplő dátumot a userHasMail kontroller getDateFormat függvény meghívásával átkonvertálom olvasható formátummá. Ha az adott üzenet olvasatlan, akkor ezt sort sötétszürke színnel és vastag betűtípussal kiemelem. A táblázat következő sorában megjelenítem az üzenet törzsét, viszont ezt a sort alapértelmezetten nem láthatóvá teszem. A JavaScript toggle() funkciójával biztosítom azt, hogy az adott sor kinyitható, esetleg összecsukható legyen.

Az "Új üzenet" menüpontra kattintva küldhetünk üzeneteket. Az oldal megnyitását követően leellenőrizzük, hogy az adott felhasználó hozzáférhet-e az adott oldalhoz, majd megjelenítjük számára az üzenet írásához szükséges űrlapot. A beviteli mezőket tekintve először szükséges megadnia a címzett e-mail címét, az üzenet tárgyát, és a textarea-ban magát az üzenetet. A küldés gombra kattintva meghívódik az üzenetet elküldő metódus, amely először megvizsgálja a megadott adatok helyességét, egy új üzenet modell példány számára beállítja ezeket az adatokat, és a UserHasMail kliens példánya segítségével átadja ezt a szerveren található create függvény számára. A sikeres küldést követően a kontrollerben található üzenethez tartozó mezőket alaphelyzetbe állítjuk és a felhasználót visszairányítjuk az üzenetek oldalra.

3.4. Felhasználói fiók

Minden felhasználónak lehetősége van megtekinteni a saját regisztrációja által bekerült adatokat az adatbázisba és szükség esetén ezeket az adatokat módosíthatja is.

Az oldal megnyitását követően az Account kontroller mezőibe betöltésre kerülnek az adott felhasználó adatai, melyet úgy kapunk meg, hogy a User kliens osztály egy példánya meghívja az adott felhasználó keresése szolgáltatást a Session-ből lekérdezett felhasználó azonosító átadásával. A lekérdezést követően a weboldalon kiíratásra kerül a felhasználó vezeték- és keresztneve, e-mail címe, telefonszáma, a lakcímét tekintve az ország azonosítója általi lekérdezést követően az ország neve, a megye azonosító segítségével a megye neve, a város azonosítóval pedig az irányítószám és a város neve. A táblázat utolsó két sorában az utcát és a felhasználóhoz tartozó szerepkört jelenítjük meg.

A módosítás gombra kattintva átirányítjuk a módosítást biztosító weboldalra, ahol az űrlapon szereplő adatok megváltoztatását követően a mentéssel rögzítheti a módosításokat. Az felhasználói adatok betöltése hasonló módon történik mint a megtekintési oldalon, annyi eltéréssel, hogy az adatok beviteli mezőkbe kerülnek, míg az országot, megyét és a várost tekintve pedig legördülő listába. Ezen az oldalon lehetősége van a felhasználónak a jelszavának megváltoztatására is, ehhez biztosítva van kettő jelszó beviteli mező. A mentés gomb megnyomását követően meghívásra kerül adatok mentését

biztosító metódus. A metódus megvizsgálja a beállítani kívánt adatokat, hiba esetén értesíti a felhasználót, viszont ha mindent rendben talált akkor meghívja a User kliens edit metódusát a módosított felhasználó és az azonosítójának paraméterben való átadásával, ezzel megtörténik a rögzítés.

3.5. Saját állatok

A nyilvántartási rendszer fontos eleme az állategyedek felvétele, módosítása és listázása. Minden felhasználó megtekintheti a hozzátartozó egyedeket a "saját állatok" menüpontra kattintva.

Az állatok listázása hasonlóan az üzenetek listázásához, az egyedek külön-külön egyetlen sort töltenek be, pár információt megjelenítve. Az információk balról jobbra haladva a következők: azonosító szám, megadásától függően az egyed neve, az egyed fajtájának neve, melyet a fajta azonosítójából kapunk és végül az egyed születési dátuma olvasható formátumban. Az egyed adatait a <c:forEach> címke segítségével listáztam, azáltal, hogy az "items" attribútumában megadtam az Animal kontrollernek azt a függvényét, amely az adott felhasználó azonosítóját felhasználva meghívja a szerveren található állatok lekérdezését. A szolgáltatás felépít egy statikus lekérdezést egy már megírt nevesített lekérdezésből, átadva a felhasználó azonosítóját. A felhasználó azonosítójának megadásával történő lekérdezés megírását tekintve azokat az állatokat kellett megkapnunk, melyek azonosítója megegyeznek a BreedingHasAnimal, azaz a tenyészetek és az állatok kapcsoló táblájában található állat azonosító számmal, továbbá ennek a rekordnak a tenyészet azonosítója meg kell, hogy egyezzen a UserHasBreeding, azaz a felhasználó és a tenyészet kapcsolótáblájában található tenyészet azonosítójával, és ennek a rekordnak a felhasználó azonosítója egyenlőnek kell lennie a paraméterben átadott felhasználó azonosítóval. A BreedingHasAnimal táblában egy állathoz több tenyészet is tartozhat, attól függően, hogy egy-egy tenyészetben meddig tartózkodott és hányszor váltott tenyészetet. Ebből adódóan a lekérdezés során figyelnünk kellett arra, hogy csak azokat az állatokat kérjük le, melyek az adott tenyészet esetén még nem rendelkeznek tenyészetből való kikerülés dátummal, mivel ezek az állatok jelenleg a keresett felhasználóhoz tartoznak, a múltbéli állatait pedig nem szeretnénk megkapni.

program lista 3.3. Példa NamedQuery-re

```
1 @NamedQuery(name = "Animal.findByUserId", query =
2         "SELECT a FROM Animal a, BreedingHasAnimal b,
              ↪ UserHasBreeding u "
3         + "WHERE a.earTag = b.animalEarTag "
4         + "AND b.breedingId = u.breedingId "
5         + "AND u.userId = :userId "
6         + "AND ( b.endDate < 1 "
7         + "OR b.endDate IS NULL)" )
```

Az állatok lekérdezését követően az állatról minden fontos adat megjelenítésre kerül kliens oldalon a korábban említett ciklus által. Az adatokat egy táblázat soraiba rendezve jelenítettem meg. A felhasználó az általa kiválasztott egyedet tartalmazó sorra kattintva meghívja az adott egyedre vonatkozóan az egyed összes adatának lekérdezését biztosító metódust. A weboldalon ezt követően megvizsgálom, hogy az általa választott egyed lekérdezése sikeresen megtörtént-e és ezt követően a <c:set> címke "var" attribútumaként beállított "animal" és a "value" attribútumában átadott lekérdezett egyeddel az adott állathoz biztosítok egy változót a könnyebb feldolgozás érdekében, ezáltal például az egyed azonosítójának megjelenítésére az "animalController.searchedAnimal.earTag" helyett hivatkozhatunk "animal.earTag"-ként is.

Az egyed adatlapján megjelenítésre kerül az egyed azonosítója, neve, születési dátuma, halál dátuma, neme, az egyed faja, fajtája és színe, melyekhez külön-külön biztosítottam egy függvényt, mely az egyedhez tartozó faj, fajta és szín azonosítók alapján vissza adja azok neveit. Továbbá megjelenítésre kerül az egyed adott tenyészet kódja, állattartó neve, tenyészet címe, tenyészetbe érkezés és a legutóbbi inszemináció dátuma. Az egyed születését tekintve megjelenítésre kerül az anya azonosítója, információ az ikerellésről, az ellésmódjának elnevezése és a borjú tömege. A következő sorokban az egyed előző tenyészetei kerülnek megjelenítésre szintén a <c:forEach> ciklus által. A kliens meghívja a BreedingHasAnimal szolgáltatás osztálynak azt a függvényét, mely az átadott egyed azonosító alapján vissza adja a hozzá tartozó tenyészeteket időrendi sorrendbe rendezve. A weboldalon sorban a tenyészet kódja, az egyed bekerülésének és kikerülésének dátuma jelenik meg. Az ezt követő sorokban az egyedhez tartozó betegségeket listáztam ciklus által. A folyamat a tenyészetekhez hasonló, azzal a különbséggel, hogy itt a AnimalHasDiseases szolgáltatás osztályon keresztül kérjük le a szükséges adatokat. Az egyed előző tenyészeteinek és betegségeinek megjelenítése abban az esetben releváns, ha korábban már tartózkodott más tenyészeteknél és átesett már valamilyen betegségen. Ezáltal a tenyészetek és a betegségek listázása előtt a <c:if> "test" attribútumában megvizsgáltam a megjelenítéshez megfelelő listában szereplő adatok számát és 0 esetén nem jelenítettem meg az adott listázáshoz szükséges cella fejléc elemeket.

Az adatlap alján az egyedhez tartozó módosítási gombot helyeztem el. A gomb megnyomásával betöltésre és megjelenítésre kerülnek az egyed adatai átirányítva a módosítási oldalra. Az oldalon lehetőségünk van az egyed azonosítóján kívül mindent módosítani. A dátumok megadására a <h:inputText> címkét az "a:type="date"" attribútum beállítással használtam, ezzel egy dátum típusú html beviteli elemet generáltam a weboldalon. Továbbá az egyed modellosztályán belül biztosítottam egy getter property-t, ami "yyyy-MM-dd" formátumban adja vissza az adott dátumot a naptár számára és egy setter property-t, ami a naptárban beállított időt ezredmásodpercé konvertálja és az értéket beállítja a hozzátartozó dátum mezőjének. A tenyészeteket és a betegsége-

ket külön-külön táblázatban helyeztem el, így a hozzájuk tartozó sorokat a felhasználó egyszerűbben módosíthatja és törölheti a sorvégén található törlés gombbal. A táblázatok alján egy hozzáadás gombot helyeztem el az új sorok beszúrását biztosítva. A már meglévő tenyészetek és betegségek a kontrolleren belül egy-egy listába kerülnek, minden módosítás az adott lista elem módosítását jelenti, egy lista elem törlésekor a törlést végző metódus paraméterlistájában bekérjük az adott lista elemet és elvégezzük a törlést. A hozzáadás hasonlóan működik, a meghívott metódus a listához egy új üres lista elemet fűz. A weboldalon a törlés és a hozzáadás gombok megnyomásával meghívásra kerül az <f:ajax> címkék között definiált ajax kérés, amivel frissül az űrlap, ezáltal láthatóvá válik az adott lista művelet, hozzáadás esetén az új üres sor, törlés esetén pedig a kiválasztott sor eltüntetése.

A kívánt módosítások elvégzése után a felhasználó a mentés gombra kattintva meghívja a módosítások mentését végző metódust. Az Animal, a BreedingHasAnimal és az AnimalHasDiseases modellosztályokban a naptár számára felvett dátumokat formázó property-k miatt már nem illeszkednek ezek az osztályok az elvárt JSON formátumra. Ebből adódóan az org.json.JSONObject osztály segítségével állítottam össze a módosításhoz szükséges JSON formátumú String típusú értéket. A JSONObject osztály működését tekintve, példányosításkor a ".put()" metódussal új értéket adhatunk az objektumunkhoz, ahol az első paraméter a kulcsot, míg a második paraméter az értéket definiálja.

program lista 3.4. Példa JSONObject osztályra

```
1 String jsonString = new JSONObject()
2         .put("earTag", this.newAnimal.getEarTag())
3         .put("name", this.newAnimal.getName())
4         .toString();
```

A String típusú json formátumú értékek felépítését követően, az adott modellosztálynak megfelelően meghívtam a hozzátartozó kliens módosító metódusát és átadtam paraméterben a már felépített String típusú json objektumot, ezzel megvalósítva az adatok rögzítését.

Az állatok listázását megvalósító weblapon elhelyeztem egy új egyed felvételét biztosító gombot is. A gomb megnyomását követően a felhasználó átirányításra kerül a módosításhoz hasonló oldalra, az adatok betöltése nélkül, üres űrlapot megjelenítve. Az oldal kitöltését tekintve megegyező a módosítási oldaléhoz, azzal a különbséggel, hogy az állat azonosítóját is meg kell adnia a felhasználónak. Az új egyed mentését biztosító metódus annyiban tér el a módosítást végzőtől, hogy itt a kliensek "create_JSON" metódusát hívják meg, az új rekordok felvételének megvalósításához.

3.6. Tenyészetek

A felhasználóhoz tartozó tenyészetek listázásának megvalósítását a "Tenyészetek" oldalon biztosítottam, továbbá ezen az oldalon keresztül vehet fel új tenyészetet és módosíthat egy már meglévőt.

A tenyészeteket először egy-egy sorban jelenítem meg, pár információt feltüntetve, hasonlóan az állatok listázásához. Minden tenyészetről a felhasználó a tenyészetkódját, a tenyészet nevét, a tenyészettípusát és a tenyészet besorolását tekintheti meg. Egy tenyészet kiválasztását követően az adott tenyészet sorára kattintva a felhasználó megtekintheti a tenyészet adatlapját, ami a tartalmazza a tenyészetkódját, nevét, a tenyészethez tartozó állattartó nevét, tenyészet címét a hozzákapcsolódó tartási hely címének lekérését követően, továbbá a tenyészet típusát és minősítését. Az állategészség ügyi adatokon belül megjelenik a tenyészet besorolása, az illetékes állategészség ügyi hatóság azonosító száma és neve. Ahogy az állatok esetében, az adatlap alján megtalálható az adott tenyészethez tartozó módosítási gomb.

A módosítási oldalon az adott azonosítóhoz tartozó tenyészet adatainak módosítására van lehetőség. Ezen az oldalon egy legördülő listából kiválaszthatjuk a tényeszet típusát, minősítését, besorolását. Módosíthatjuk a tenyészethez tartozó állattartót, tenyészet nevét és kiválaszthatjuk mely tartási helyhez kapcsolódjon. A mentés gomb megnyomásával a Breeding kontrolleren belül található módosítások elmentését biztosító metódus kerül meghívásra. A metódus a tenyészet modellosztályának adott példányát a kliens egy példánya által meghívott edit_JSON metódus paraméterében átadva elvégzi a rögzítést. A következő lépésben a tenyészethez tartozó állattartó módosítása történik, oly módon, hogy az adott UserHasBreeding modellosztály kerül átadásra az ehhez tartozó kliens osztály edit_JSON metódusa számára. Utolsó lépésben a kapcsolódó tartási hely módosításának rögzítésének történik.

3.7. Tartási hely

A felhasználóhoz kapcsolódó tartási helyek megjelenítés, módosítása és felvétele a "Tartási hely" menüponton belül történik.

A listázást tekintve a szerveroldalon található nevesített lekérdezésnek a felhasználó azonosítója alapján kell visszaadnia azokat a tartási helyeket, melyek azonosítója megegyezik a HoldingPlaceHasBreeding kapcsolótáblában található tartási helyek azonosítójával, továbbá ezeknek a rekordoknak a tenyészet azonosítójuk a UserHasBreeding kapcsolótáblában található tenyészetekkel egyeznek meg, amellett, hogy a keresett felhasználóhoz tartoznak. Ezzel visszakapjuk azokat a tartási helyeket, melyekhez az adott felhasználó tenyészetei csatlakoznak. Az új tartási hely rögzítését követően előfordulhat olyan eshetőség, hogy még nem kapcsolódik hozzá tenyészet, ebből adódóan

nincs lehetőségünk egyetlen felhasználó számára sem hozzáférhetővé tenni ezt a helyet megjelenítés és módosítási szempontból sem. A probléma megoldására a tartási hely táblájában létrehoztam egy mezőt, mely a kapcsolattartó személy azonosítóját rögzíti, míg a lekérdezéshez hozzáfűztem egy "vagy" operátort és definiáltam azt az eshetőséget, hogy a keresett felhasználó azonosítója megegyezik a tartási hely kapcsolattartó mezőjében szereplő azonosítóval. A tartási hely adatlapján a tartási hely azonosítására megjelenik az azonosítója és a címe, az adott tartási helyhez kapcsolódó tenyészetek tevékenységeit a HoldingPlaceHasSpecies kapcsolótábla rekordjai közül kapjuk meg, majd soronként listázva megjelenítjük a kiválasztott fajt, a tevékenység kezdetének dátumát és a hasznosítást. A tartási hely helyrajzi számainak megjelenítésekor minden sorban megjelenítésre kerül az adott helyrajzi szám városa és maga a helyrajzi szám. A helyhez kapcsolt kapacitások listázásakor megjelenítésre kerül a kapacitás neve, mérete és a létesítés dátuma. Egy új tartási hely felvételekor és már egy meglévő módosításakor a tenyészetek tevékenységei, a helyrajzi számok és a tartási hely kapacitásai esetében fontos szempont volt, hogy a tartási helyhez ezekből a rekordokból több felvétele is biztosított legyen. Ezáltal, ahogy már korábban az állatok esetében az állathoz kapcsolódó tenyészetek és betegségek módosítása történt, itt is biztosítottam ezekhez a rekordokhoz egy-egy listát. Az adott lista elem módosításakor a listában szereplő rekord tulajdonságai kerülnek módosításra, törlés esetén a listából kiválasztott objektum kerül törlésre, míg hozzáadás esetén egy új, tulajdonság nélküli elem kerül rögzítésre, melyet a felhasználó tetszőlegesen módosíthat. A mentés gombra kattintva egy-egy ciklus által felépítem minden lista elemhez a megfelelő String típusú json objektumot és ezt követően szelekcióval eldöntöm, hogy az adott listaelem az adatbázisban már szereplő rekord vagy egy újonnan felvenni kívánt elem. Ha már szerepel az adatbázisban akkor a hozzátartozó kliens edit_JSON metódusával módosítom, viszont ha még nem szerepel akkor a kliens create_JSON metódusával rögzítem az új elemet. Egy listaelem törlése esetén a törölni kívánt elem azonosítóját eltárolom egy Integer osztályú objektumok tárolására alkalmas listában és a mentést végző metóduson belül minden azonosító átadásra kerül a neki megfelelő kliens által meghívott remove metódus paraméterlistájába.

3.8. Engedélykérések

Az engedélykérések menüpont alatt találhatóak az adminisztrátorok és az orvosok számára a nem elfogadott tartási helyek, tenyészetek, állatok listázását biztosító menüpontok, továbbá az adminisztrátorok itt találhatják a regisztrációkat listázó menüpontot is.

A regisztrációkat csak az adminisztrátorok tekinthetik meg, a weboldal hozzáférését tekintve adminisztrátori szerepkör szükséges. Ezen az oldalon kerülnek listázásra

az új felhasználók, akik regisztrációját még nem fogadtak el. Az adminisztrátorok itt ellenőrizhetik a regisztrálni kívánt felhasználók adatait az adott személy adatlapjának kiválasztását követően. Az elfogadást megelőzően lehetőségük van a felhasználó szerepkörét kiválasztani, állatorvosi szerepkör esetén megyéket rendelhetnek az adott felhasználóhoz.

A nem elfogadott tartási helyek, tenyészetek és állatok listázása kétféleképpen történhet. Az adminisztrátorok számára minden rekord listázásra kerül, viszont az állatorvosok számára csak azok, melyek az adott állatorvos megyéiben találhatóak. A tartási helyeket listázó weblapon az elfogadás előtt lehetőségünk van módosítani az adott tartási helyhez tartozó állatorvos azonosítóját, mely módosítását követően egy ajax hívással az azonosítóhoz tartozó név is megjelenítésre kerül. Esetleges hiba esetén értesíthetik, tájékoztathatják a kapcsolattartó felhasználót az üzenet gombra kattintva, ezzel betöltésre kerül az "Új üzenet" oldal, továbbá a kapcsolattartó felhasználó e-mail címe bekerül a címzett beviteli mezőbe. A nem elfogadott tenyészetek és állatok listázását biztosító weblapokon szintén minden rekordhoz tartozik üzenet küldési lehetőség. Az elfogadáshoz az elfogadást végrehajtó személynek szükséges megadnia a jelszavát és ezt követően az elfogadás gombra kattintva, helyesen megadott jelszó esetén az adott rekord elfogadott állapotba kerül a módosítás során és kikerül az adott oldal lista elemei közül.

Az adminisztrátoroknak és az állatorvosoknak a "Listázás" menüpont alatt lehetőségük van megtekinteni a kiválasztott kategóriának megfelelő rekordokat és azok adatlapját. Az állatorvosok csak a hozzájuk tartozó megyékben található rekordokat tekinthetik meg és a regisztrált felhasználókhoz csak az adminisztrátoroknak férhetnek hozzá. Minden rekord esetében lehetőségük van üzenetet küldeni a rekordhoz valamilyen módon kapcsolódó felhasználónak, emellett az állatok listázását megvalósító oldalon az állatorvosok minden egyed esetében módosíthatják az állathoz tartozó betegségek listáját. Az adminisztrátoroknak lehetőségük van a felhasználókat az elfogadási oldalhoz hasonlóan módosítani, azaz kiválaszthatják a felhasználó szerepkörét, és az állatorvosokhoz tartozó megyéket kezelhetik. A módosítást követően jelszóval tudják megerősíteni a rögzítést.

4. fejezet

Összegzés

A szakdolgozatom célja az állatok nyilvántartásához biztosított nyilvántartó rendszer fejlesztése volt, melyben minden állat esetén fontos rögzíteni az állatok faját, fajtáját, színét, nemét, születési dátumát, születésével kapcsolatos információkat, esetleges betegségeit és az egyed azonosítására alkalmas azonosítószámot. Az állatok mellett a tenyészetek és a tartási helyek tárolását és kezelését is biztosítottam, ezáltal az állatok nyomon követhetővé váltak, lekérdezhető az állathoz kapcsolódó tartózkodási helyek címei és típusai. A nyilvántartó rendszert használó személyek a rendszerben betöltött szerepüknek megfelelően férhetnek hozzá különböző funkciókhoz, az állattartók a saját állataikat, a hozzájuk tartozó tenyészeteket és tartási helyeket kezelhetik, míg az állatorvosok a hozzájuk rendelt megyék szerint tekinthetik át és kezelhetik a főbb rekordokat. Az adminisztrátorok hozzáférhetnek az összes felhasználóhoz, állathoz, tenyészethez és tartási helyhez.

A szakdolgozat elkészítése során megismerkedhettem új technológiákkal, tovább fejleszthettem a kliens-szerver architektúra ismereteimet a Java programozási nyelven. A webalkalmazás fejlesztése közben elmélyíthettem tudásomat a JAX-RS technológián belül, mellyel lehetőségem nyílt a Java Persistence API által leképezett perzisztencia osztályokhoz kapcsolódó szolgáltatásokat létrehoznom, amiket később a kliensek meghívhatnak és használhatnak. A JavaServer Faces és a hozzá kapcsolódó Facelets és az Expression Language technológiák megismerésével könnyen hozhattam létre weboldalakat, az oldalakon megjelenő tartalom kontrollerekből való kezelése egy-egy EL kifejezéssel egyszerűen megvalósítható, ellenőrizhető és feldolgozható. A kliens oldali adatfeldolgozást tekintve a @SessionScoped annotációval ellátott, MVC minta szerinti kontroller osztály segítségével könnyedén valósítottam meg a kapcsolatot a grafikus felhasználói felület és a szerver oldalon található szolgáltatások között.

Az új technológiák megismerésével és elsajátításával megalapozhattam ezen technológiákon belüli fejlődésemet, melyet a későbbiekben a továbbtanulás folyamán a további új ismeretek szerzésével és elmélyítésével valósíthatok meg, amit a későbbi munka folyamán nagymértékben hasznosíthatok.

Irodalomjegyzék

[1] Fazekas István: *Valószínűségszámítás*, Debreceni Egyetem, Debrecen, 2004.

[2] Tómács Tibor: *A valószínűségszámítás alapjai*, Líceum Kiadó, Eger, 2005.