Szakdolgozat

Péntek István

Debrecen

2010

Debreceni Egyetem
Informatika Kar

Webalkalmazásfejlesztés java technológiákkal -

Spring Web MVC

Témavezető:
Dr. Adamkó Attila
egyetemi adjunktus

Készítette:
Péntek István
programtervező informatikus

Debrecen

2010

1 Tartalomjegyzék

2 Történet

A Spring egy nyílt forrású keretrendszer. Jelenleg két támogatott platformra fejlesztik *Java* és *.Net*. Az első, nem hivatalos verziót *Rod Johnson* készítette, aki az *Expert One-on-One J2EE Design and Development* könyvében publikálta alkotását, 2002-ben. A keretrendszer létrehozását a piaci helyzet kényszerítette ki, ugyanis a Spring előtt nem volt olyan *J2EE*-re épülő keretrendszer, amely általános megoldást kínált volna a fejlesztés során felmerülő problémákra, a gyors és hatékony alkalmazásfejlesztésre, illetve a rétegelt alkalmazások készítésére. 2003 júniusában jelent meg *Appache 2.0* licenszel az első nagyközönségnek szánt változat, amely még nem végleges volt, de már tükrözte a Spring által kitűzött célt. Az első mérföldkő az 1.0 verzió volt, amely 2004 márciusában jelent meg. Az 1.2.6-os verzió több díjat is elnyert, ami már ekkor azt sugallta, hogy érdemes komolyan venni ezt a keretrendszert:

- Jolt productivity (2006)
- Jax Innovation (2006)

Jelenleg 3.0.1-es verzió a legfrissebb, amely elérhető a fejlesztői közösségnek.

A Spring rétegelt kialakítású, ami azt jelenti, hogy az egyes feladatkörök külön modulba tartoznak, a modulok kommunikálhatnak egymással. Ezt a keretrendszer, illetve a modulok konfigurálásával befolyásolhatjuk. Fontos megjegyezni, hogy a keretrendszer nem kényszeríti ki egyetlen egy tervezési minta, illetve szabvány követését sem, csak javaslatokat tesz. A fejlesztőn múlik, hogy melyiket preferálja.

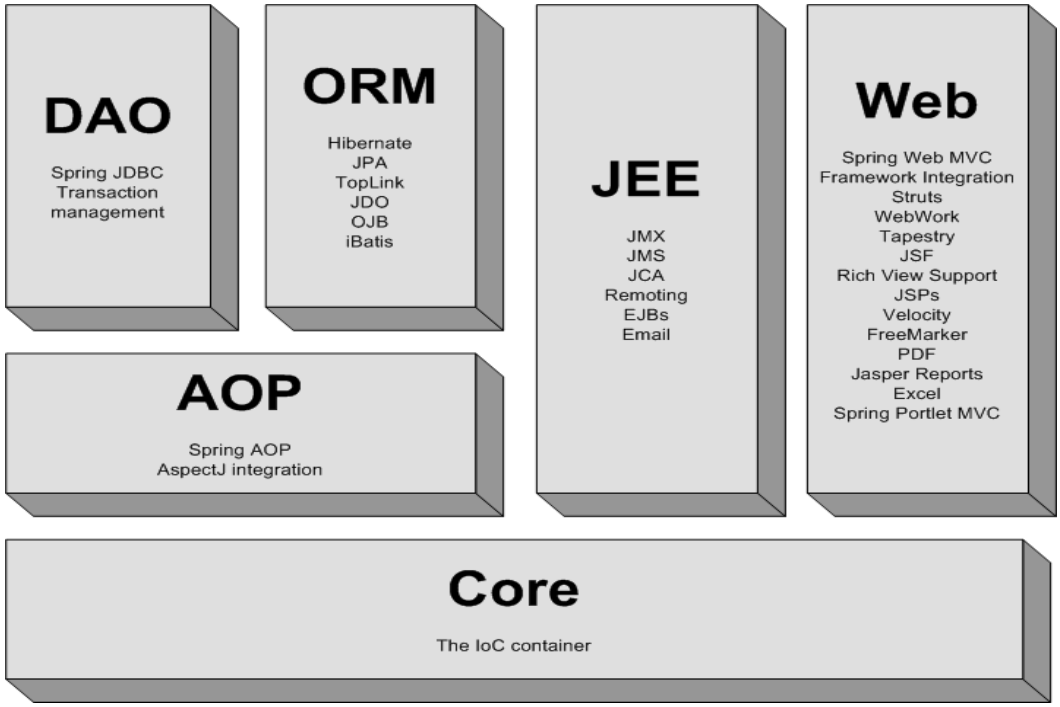
3 Célkitűzés

A szakdolgozat témája az *MVC* tervezési minta alapjainak bemutatása és a köré épülő lehetőségek megismertetése, illetve a Spring keretrendszer MVC mintához nyújtott támogatásának ismertetése. Ehhez előbb tisztázni kell a keretrendszer magját alkotó függőség injektálást és a függőség visszakeresést. Ezek segítségével képet kapunk majd az *Inversion of Control* működéséről. Részletesen kitérünk a kontrollerek működésére, elfogókat és nézetfeloldókat fogunk készíteni. Részletesen foglalkozunk a

nézettechnológiákkal és azzal, hogy a Spring milyen támogatást nyújt hozzájuk, legfőképp a *Java Server Pages(JSP)* technológiához. A nézeteknél kitérünk a Spring által nyújtott *form tag* komponensekre is, amelyek kiválthatják a JSP komponensek egy részét. Minden egyes főbb részhez készíteni fogunk egy kis alkalmazást, amely az itt tárgyalt témához készül, viszont nem egy komplex web alkalmazás, csak iránymutatás a megvalósításhoz.

4 Modulok

A keretrendszer több modulból áll, amelyek a jelenlegi piaci helyzetben tökéletesen elegendőek bármilyen üzleti alkalmazás fejlesztéséhez. Fontos megjegyezni, hogy nem kötelező mindet használni, illetve egyes modulok kiválthatóak egy, a piacon lévő másik modullal. Ebben az esetben biztosítani kell, hogy a Spring és a kiváltott modul kommunikálni tudjanak. A modulok több technológiát is támogatnak, ezzel is rugalmassá téve a fejlesztést. Külön modulok állnak rendelkezésre webes és *Java Enterprise Edition JEE* fejlesztésekhez, a fejlesztési területek sajátosságait figyelembe véve.



1. ábra: A Spring keretrendszer felépítése

Jelenleg az alábbi szolgáltatásokat nyújtja a keretrendszer, amelyekből kettőt részletesen megnézünk a szakdolgozatban:

- Inversion of Control konténer konfigurálja az alkalmazás kontroljait és Java objektumait, illetve menedzseli azokat életciklusuk során. Ez alkotja a Spring magját.
- Aspektusorientált programozás: az aspektus egy átmetsző követelmény reprezentációja. Az aspektus hasonlít a komponenshez abban, hogy a követelményt egy csomagban írja le, de az aspektus hatása a rendszerben elszórva jelentkezik. A Spring ehhez széleskörű támogatást nyújt, illetve rengeteg beépített eszközt biztosít. AspectJ programozási nyelv teljes körű támogatása.
- Menedzselt adathozzáférés: nemcsak adatbázis szinten, hanem komponens és bean szinten is nagyon erős a Spring adathozzáférési mechanizmusa. Nagyon átgondolt és hatékony adatkötés jellemzi.
- Tranzakciók kezelése: szabványos módon vezérelhetjük a tranzakciókat a keretrendszer támogatásával. Absztrakt módon, adatbázisrendszer típusától elvonatkoztatva lehet a tranzakciókat indítani, visszagörgetni, illetve véglegesíteni.
- Modell-View-Controller(MVC): egy tervezési minta. Lényege, hogy különválasztja a kódot modellre és nézetre, amelyek között a kapcsolatot a kontroller biztosítja, ő vezeti át a változásokat a két réteg között.
- Távoli hozzáférést biztosító keretrendszer: objektumok távoli hozzáférését biztosítja, ehhez szükséges a megfelelő szintű azonosítás. A `security` modullal összekötve hatékonyan alkalmazható.
- Kötegelt végrehajtás: a Spring támogatja az utasítások kötegelten történő végrehajtását, ami azt jelenti, hogy a beérkező kéréseket, utasításokat a keretrendszer bizonyos időközönként, illetve bizonyos számú végrehajtandó utasítás számának elérésével fogja végrehajtani.
- Hozzáférési jogosultságok kezelése: a Spring `security` egy előre implementált

felhasználói jogosultságkezelési mechanizmus. Több előre definiált szerepkört is tartalmaz, de olyan rugalmas, hogy akár saját szerepkörök is definiálhatóak.

- Üzenetek kezelése: a keretrendszer komponensei illetve moduljai kommunikálhatnak üzenetek, azaz delegáltak segítségével. Ehhez a Spring egy beépített kezelőt tartalmaz, ami akár felül is definiálható, illetve kiterjeszthető az aktuális feladat igényeinek megfelelően.
- Tesztelés: a legtöbb projekt során kevés a tesztelési lehetőség a szűk határidők, illetve a tesztelés bonyolultsága miatt. A keretrendszer erre kíván megoldást nyújtani annotációval, illetve beépített tesztelési mechanizmusokkal. Erősen támogatja a *JMock* eszközt.

5 Inversion of Control(IoC) konténer

Az *Inversion of Control* konténer konfigurálja az alkalmazás kontroljait és Java objektumait, illetve menedzseli azokat életciklusuk során. A java objektumok *Plan Old Java Object(POJO)*-k. *Martin Fowler*-től származik az *Inversion of Control* egy átfogalmazása, a *Dependecy Injection* vagyis a függőség injektálás, amely az *Inversion of Control* konténer egy változata. Az *Inversion of Control* konténer alkotja a Spring magját és az alábbiakért felelős:

- Objektumok létrehozása objektum, értékének másolása,
- Inicializáló metódusok hívása,
- Illetve objektumok konfigurálása
- Modulok között az objektumok megosztása

A konténer által létrehozott objektumok hívhatnak további menedzselt objektumokat vagy beaneket. A konténer konfigurálása egy XML betöltésével és értelmezésével történik. Ebben az XML fájlban vannak a beanek definíciói, ez alapján kell a beaneket létrehozni, illetve konfigurálni beállító metódusok, illetve konstruktorok segítségével. Az objektumok *Dependency Injection* vagy *Dependency Lookup* segítségével érhetőek el. Két különböző IoC van implementálva a Spring keretrendszerbe. A két implementációt az

alapötlet különbözteti meg egymástól, valamit a keretrendszer felxibilitása miatt szükségesek. A *Dependency Injection* megközelítés modern oldalról kezeli a problémákat. A *Dependency Lookup* egy régebbi szemléletet vall. Mindkét típusú implementációnak van két további implementációja. A további alkalmazott technológiától függően kell választani a kétféle *Inversion of Control* konténer implementáció közül.

5.1 Dependency Lookup

Ennél az implementációnál a komponensnek egy referenciát kell birtokolnia, minden egyes függőségéhez. A függőségek referenciái literálként jelennek meg a komponensben. Ezeken operál az *Inversion of Control*. A *Dependency Lookup*-nak két implementációja van a Spring keretrendszerben: *Dependency Pull*(függőség vonzás) és *Contextualized Dependency Lookup*(környezetei függőség visszakeresés).

5.2 Dependency Pull

Ez az implementáció teszi lehetővé, hogy a Spring ne csak Java EE környezetben tudjon, működni, hanem más platformokhoz igazodva is lehetővé tegye rétegelt alkalmazások készítését. Ennek köszönhetően használható a Spring *.NET* környezetben is. Természetesen ez a fajta *Inversion of Control* konténer implementáció is használható Java környezetben, viszont használata sokkal több plusz munkát igényel.

```
package szakdolgozat.ioc;

import szakdolgozat.spring.MessageService;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.
          support.BeanDefinitionReader;
import org.springframework.beans.factory.support.
          DefaultListableBeanFactory;
import org.springframework.beans.factory.
          support.PropertiesBeanDefinitionReader;
import org.springframework.core.io.ClassPathResource;
```

```
     public class DependencyPullDemo {
          public static void main(String[] args) {
               BeanFactory bf = getBeanFactory();
               MessageService service =
                    (MessageService) bf.getBean("service");
               service.execute();
          }

          private static BeanFactory getBeanFactory() {
               DefaultListableBeanFactory bf =
                    new DefaultListableBeanFactory();
               BeanDefinitionReader reader =
                    new PropertiesBeanDefinitionReader(bf);
               reader.loadBeanDefinitions(
                    new ClassPathResource("/META-INF/spring
                         /ioc.properties"));
               return bf;
     }
}
```

1. forráskód: A Dependency Pull használata forráskódból

Ahogyan az a kódból is látszik, ennél a módszernél a komponens egy központi helyen található, elérni a *BeanFactory* osztály használatával lehetséges. A regisztrációt általában a konstruktorban célszerű elvégezni. Aki a későbbiekben el szeretné érni, az a *BeanFactory* használatával teheti meg. A *BeanFactory* előkeresi a regisztrált beanek közül az aktuális névvel jelöltet és egy objektumot ad vissza.

5.3 Contextualized Dependency Lookup

Hasonlóan működik, mint a *dependency pull*, viszont ebben az esetben a függőségek keresését a konténer végzi. Mindenképpen implementálnia kell a komponensnek a *ManagedComponent* interfészt, ami így néz ki:

```
public interface ManagedComponent {
     void lookup(BeanFactory container);
}
```

2. forráskód: ManagedComponent interfész felépítése

Egyetlen egy metódus implementálását követeli csak meg ez az interfész, ezzel jelezve a konténernek, hogy a komponens egy lehetséges függő-függőség kapcsolatban áll és lehetőség van visszakeresni azt. Amikor a konténer kész átadni egy komponensnek a függőségeit, akkor meghívja az interfész alapján definiált *lookup()* metódust az összes komponensen. Ekkor a komponensek felderíthetik az összes függőségüket a *BeanFactory* interfész segítségével.

```
public class ContextualizedDependencyLookupDemo {
     private static Set<ManagedComponent> components =
          new HashSet<ManagedComponent>();

     private static class MessageServiceComponent implements
        ManagedComponent {
          private MessageService service;

          public void lookup(BeanFactory container) {
              this.service =
                    (MessageService)container.getBean("service");
          }

          public void run() {
              this.service.execute();
          }
     }

     public static void main(String[] args) {
          BeanFactory bf = getBeanFactory();
          MessageServiceComponent msc = new
               MessageServiceComponent();
          registerComponent(msc);
          allowComponentsToLookup(bf);
          msc.run();
}
```

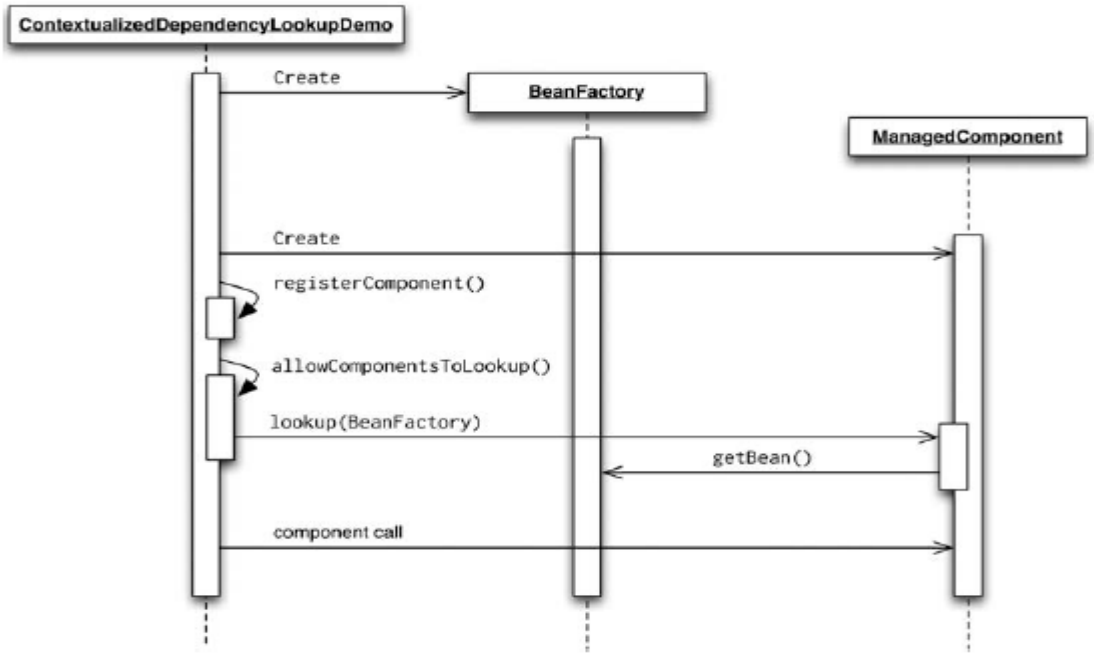
```
private static void allowComponentsToLookup(BeanFactory bf)
{
          for (ManagedComponent component : components) {
               component.lookup(bf);
          }
     }

     private static void registerComponent(ManagedComponent
     managedComponent) {
          components.add(managedComponent);
     }

     private static BeanFactory getBeanFactory() {
          DefaultListableBeanFactory bf =
               new DefaultListableBeanFactory();
          BeanDefinitionReader reader =
               new PropertiesBeanDefinitionReader(bf);
          reader.loadBeanDefinitions(new ClassPathResource(
               "/META-INF/spring/ioc.properties"));
          return bf;
     }
}
```

3. forráskód: Függőség visszakeresés használata forráskódból

A kódrészletből látszik, hogy a *MessageServiceComponent* osztály egy menedzselt komponens, őse a *ManagedComponent*. Egyetlen egy metódus implementálását követeli meg az interfész alapján a *lookup()* metódusét. Emiatt a ContextualizedDependencyLookupDemo osztály egy konténerként viselkedik. A konténer előállítja az összes függőséget, majd ha végzett meghívja a *lookup()* metódust az összes komponensen. Lehetőség van komponens regisztrálására is. A módszer hátránya, hogy minden komponensnek implementálnia kell az interfészt, hogy felderíthető legyen.



2. ábra: Contextualized dependency lookup UML szekvencia diagram

6 Dependency Injection(DI)

Dependency injection, avagy a függőség injektálás szintén két különböző implementációval rendelkezik, a konstruktort vagy beállító metódust használó implementáció. A függőség injektálás egy tervezési minta alapján működik, szorosan kapcsolódik az objektumorientált programozáshoz. A minta 3 részből tevődik össze: *dependent*(függő), *dependencies*(függőség) és *injector*(injektáló). Megvalósítása történhet például az *abstract factory* tervezési mintával.

A függőség injektálás alapja a *BeanFactory*. A *BeanFactory* interfész segítségével menedzseljük a komponenseket és azok függőségeit. Bean az az objektum, amit a keretrendszer a *DI*-n keresztül menedzsel. Bármelyik java osztály lehet bean. Az így elkészített alkalmazás a *BeanFactory* interfésszel kommunikál. Az interfész konfigurálható kódból és XML konfigurációs fájlok segítségével is. Kódból konfigurálni az interfészt csak kisebb alkalmazásoknál ajánlott, minden egyéb esetben ajánlott az XML-t választani. Ennek az az oka, hogy a kódból való konfigurálás átláthatatlanná teheti a kódot, míg XML esetén ez nem fordulhat elő, mivel az XML tiszta, átlátható és vizuálisan

is jól tagolható. Példányosításkor sem kell a konstruktorban elhelyezni azt a kódsort, amellyel az interfészt konfiguráljuk, ezzel tisztul a kód és koncentrálhatunk az üzleti logika implementálására az osztályainknál. A bean definíciója nem magáról a beanről tárol el információkat, hanem csak a függőségeiről. Minden beannek lennie kell legalább egy nevének, viszont egy beanhez több név is tartozhat bizonyos esetekben. Továbbá a beanhez meg kell adni azt az osztályt, amelyiket hivatkozza. A neveknek egyedinek kell lennie. A nevet később a függőségek felderítésére használja a keretrendszer, illetve névvel lehet hivatkozni a beanekre kódból, bár ez nem javasolt.

A *BeanFactory* interfész implementálja a *BeanDefinitionRegistry* interfészt, amely segítségével a konfigurációs fájlból kiolvashatjuk a *BeanDefinition* információt a *PropertiesBeanDefinitionReader*, illetve a *XMLBeanDefinitionReader* segítségével, attól függően, hogy a függőségi viszonyt XML-ből vagy property fájlból állítjuk be. Az interfészt az alábbi módon használhatjuk kódból, property fájlokat használva:

```
DefaultListableBeanFactory bf = new DefaultListableBeanFactory();

BeanDefinitionReader reader = new
    PropertiesBeanDefinitionReader(bf);

reader.loadBeanDefinitions(
    new ClassPathResource(
        "/META-INF/spring/beanfactorydemo.properties"));

Pelda pelda = (Pelda) bf.getBean("pelda");
```

4. forráskód: Bean definíciók olvasása forráskódból property fájl alapján

Miután betöltjük a property fájl tartalmát kiolvassuk a pelda beant a definíciók közül. A beanre a nevével hivatkozunk. Az XML konfigurációs fájl használata hasonló módon történik:

```
DefaultListableBeanFactory bf = new DefaultListableBeanFactory();

BeanDefinitionReader reader = new
    XMLBeanDefinitionReader(bf);
```

```
reader.loadBeanDefinitions(
    new ClassPathResource(
        "/META-INF/spring/beanfactorydemo.properties"));

Pelda pelda = (Pelda) bf.getBean("pelda");
```

5. forráskód: Bean definíciók olvasása forráskódból XML fájl alapján

Természetesen lehetőség van saját interfész implementálására is, viszont ezt a lehetőséget általában nem szükséges kihasználni a keretrendszer rugalmassága miatt. Azt érdemes tudni, hogy nem elegendő ez a két interfész implementálása, hogy ugyanazt a gazdag funkcionalitást megkapjuk, mint a Spring által nyújtott funkcionalitásnál megszokott.

A Spring keretrendszer és a java környezet az XML konfigurációs fájlt teljes mértékben ki tudja használni. Javasolt ennek a használata java környezetben történő fejlesztéseknél.

```
    <bean name="/fileUpload.htm" class="szakdolgozat.
                    service.FileUploadController">
        <property name="commandName" value="fileUpload" />
        <property name="commandClass" value="szakdolgozat.
                    service.FileUploadBean" />
        <property name="formView" value="uploadform" />
    </bean>
```

6. forráskód: XML konfigurációs fájl felépítése Spring esetén

Érdemes tisztázni pár fontosabb fogalmat, amelyek az alkalmazás fejlesztése során hasznosak lesznek:

Függő

A függő, akinek szüksége van valamilyen szolgáltatásra vagy információra, amit egy másik objektum birtokol.

Függőség

Olyan objektum, akitől a függő megkapja a kért szolgáltatást vagy információt, a függő függéségét fogja alkotni. Egy függőnek több függősége is lehet és viszont. A függőség egy gráfot alkot.

Injektáló

Az injektáló az, aki biztosítja a kapcsolatot a függő és függőségei között, illetve ő menedzseli az objektumokat is. Ha egy objektumnak szüksége van valamire, amit egy másik objektum birtokol, akkor szól a keretrendszernek, a keretrendszer pedig a kérést delegálja az objektumok felé. Azok az objektumok kapják meg a kérést, akik a függő függőségeit alkotják. Egy objektum addig létezik, amíg létezik függősége, valamint a függőségek addig léteznek, amíg a függőre szüksége van a keretrendszernek. A felesleges objektumokat a keretrendszer kisöpri a memóriából.

6.1 Constructor DI

A konstruktor általi függőség injektálás használatával egy komponens függőségei a konstruktorban határozódnak meg. A konstruktorban beállított függőségeken kívül más függőségekkel nem rendelkezhet. Viszont kihasználva a túlterhelést, miszerint egy objektumnak több alaphelyzetbe hozó metódusa is lehet, nem csak egy konstruktorban lehet meghatározni a függőségeket. A konstruktornak átadott paraméter(ek) alapján beállítódnak a függőségek. A konténer pedig átadja a függőségeket a komponensnek, amikor szükséges. A szükségességét ebben az esetben a *run()* metódus jelzi, ha ide kerül a vezérlés, akkor a konténer a komponens rendelkezésére bocsátja a függőségeket.

```
public class ConfigurableEncyclopedia implements Encyclopedia {
     private Map<String, Long> entries;

     public ConfigurableEncyclopedia(Map<String, Long> entries) {
          Assert.notNull(entries,
               "Az 'entries' argument nem lehet null értékű.");
          this.entries = entries;
     }
```

```
     public Long findLong(String entry) {
          return this.entries.get(entry);
     }
}
```

7. forráskód: Konstruktorban történő injektálás

Fontos, hogy a konstruktorban megtörténjen a paraméter értékének az ellenőrzése is, mivel futási időben *NullPointerException* kivétel keletkezhet. Ha az ellenőrzés elmarad, akkor célszerű a *run()* metódust, kivételkezeléssel ellátni, mivel itt váltódhat ki a kivétel.

Az ide tartozó konfigurációs fájl az alábbi lehet:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/
            spring-beans.xsd
            http://www.springframework.org/schema/util
            http://www.springframework.org/schema/util/
            spring-util.xsd">
     <bean id="encyclopedia"
          class="com.apress.prospring2.ch03.di.
          ConfigurableEncyclopedia">
          <constructor-arg>
               <util:map>
                <entry key="A_Ertek" value="13700000000"/>
                <entry key="B_Ertek" value="326190476"/>
               </util:map>
          </constructor-arg>
     </bean>
</beans>
```

8. forráskód: Konstruktorban történő injektálás esetén a konfigurációs fájl

Amit érdemes szemügyre venni, az a *<constructor-arg>* tag. Itt történik az információátadás a keretrendszernek, mégpedig, hogy az injektálás konstruktorban fog

történni. Aztán a *util* által hivatkozott névtérből a java környezet Map osztályát fogjuk használni, amely kulcs és érték párokból áll. Ennek megfelelően az átadott paraméterek definiálása is a *Map* osztály szerint történik.

Érdemes megjegyezni, hogy a keretrendszer néha képtelen megtalálni a megfelelő konstruktort. Ez általában akkor fordul elő, ha két nagyon hasonló konstruktor is létezik az osztályhoz. Hasonlóság alatt a paraméterek számát és/vagy típusát értjük. Például, ha van két konstruktor, mindkettő egy paramétert vár, az egyik egy *String* típusút a másik egy egész számot, a sorrendben pedig a *String* argumentumot váró konstruktor áll előrébb, akkor a keretrendszer ezt fogja választani. Ez annak köszönhető, hogy az XML-ben az átadott érték nem típusos, azt majd a keretrendszer fogja típussal ellátni. Ez könnyen kiküszöbölhető, ha az alábbi formában adjuk meg a konstruktort a konfigurációs fájlban:

```
<constructor-arg value="1" type="int"/>
```

9. forráskód: Konstruktoros injektálás esetén a típus meghatározása

Ekkor már a paraméter típusát is konkretizáljuk, így egyértelműen hozzárendelhető a megfelelő konstruktor. A típus mellett a konstruktorban elhelyezkedő paraméterek közül a sorszámot is lehet definiálni a *value* attribútummal.

6.2 Setter DI

Ennek a módszernek a működése nagyon hasonló a konstruktoros megvalósításhoz, viszont ebben az esetben külön beállító metódusokat hozunk létre a függőségi kapcsolatok beállítására. Lennie kell egy vagy több megfelelően felparaméterezett metódusnak, amely paraméterként megkapja a függőségeket. A beállító metódust a konténer fogja meghívni.

```
public static void main(String[] args) {
     XmlBeanFactory factory = new XmlBeanFactory(
          new ClassPathResource("/META-INF/spring/
                  injectdemo.xml"));
     InjectSimpleDemo simple =
          (InjectSimpleDemo)factory.getBean("injectSimpleDemo");
     System.out.println(simple);
     }

     public void setAgeInSeconds(Long ageInSeconds) {
          this.ageInSeconds = ageInSeconds;
     }

     public void setIsProgrammer(boolean isProgrammer) {
          this.isProgrammer = isProgrammer;
     }

     public void setAge(int age) {
          this.age = age;
     }

     public void setHeight(float height) {
          this.height = height;
     }

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

10. forráskód: Beállító metódusokkal történő injektálás esetén a forráskód tartalmazza a beállító metódusokat

Nagy előnye a konstruktorban történő injektálással szemben, hogy a program futása során többször is elvégezhetjük a beállítást, ami lehetővé teszi a függőségek beállítását és módosítását futási időben.

```
<bean id="injectSimpleDemo"
          class="com.apress.prospring2.ch03.
          beanfactory.InjectSimpleDemo">
     <property name="name" value="John Smith"/>
     <property name="age" value="35"/>
     <property name="height" value="1.79"/>
```

```
    <property name="isProgrammer" value="true"/>
    <property name="ageInSeconds" value="1103760000"/>
</bean>
```

11. forráskód: Beállító metódusokon keresztül történő injektálás esetén a konfigurációs fájl

Ebben az esetben a beanhez kötött osztályon operál a keretrendszer. Meghívja az összes felsorolt beállító metódust a megfelelő értékekkel. Ez a módszer kihasználja a java sajátosságait mégpedig, hogy minden felsorolt csupa kisbetűs névhez tartoznia kell egy *setMetódusnév* szintaktikájú beállító metódusnak. A metódus nevét, illetve annak hivatkozását a *name* attribútum segítségével adhatjuk meg, míg a paraméter értékét a *value* attribútum segítségével.

6.3 Injektálás vagy visszakeresés?

Ennyi féle lehetőség ismeretében, jogosan merül fel a kérdés, hogy melyiket is válasszuk? Általában az alkalmazott technológia függvénye a választás. Ha java környezetben maradunk és *Enterprise Java Bean(EJB)* technológiát használunk, akkor az *EJB 2.0*–hoz a visszakeresést kell választanunk. A technológia erre ad csak lehetőséget. *EJB 3.0* már az *@Inject* annotáció segítségével támogatja az injektálást is. Viszont ekkor legalább 1.5 java motort kell használnunk, mivel ettől a verziótól támogatott az annotáció. Ettől függetlenül *EJB 3.0*-val is alkalmazható a visszakeresés. Lehetőség van még *Java Persistence API(JPA)* használatára is függőség injektálással és annotációval kombinálva.

Eltekintve attól, hogy a visszakeresést is támogatja szinte az összes mai technológia, érdemes az injektálást választani. Ennek oka, hogy drasztikusan csökken a programozóra háruló feladat és felelősség. Szinte semmit sem kell implementálni, ahhoz hogy a függőségeket kezelni tudjuk. Míg a visszakeresésnél mindkét implementációban több sor kódot kell implementálni, a függőségek felderítésére vagy minden olyan komponenst, amelyiknek függősége lesz egy interfészből kell leszármaztatni, addig az injektálásnál elég annotációt használni. Így a megírt osztályok teljesen függetlenek

lesznek a konténertől. Egy esetleges interfészváltozásnál nem kell minden egyes legyártott osztálynál implementálni a változásokat. A visszakeresés a tesztelést is megnehezíti, mivel nem könnyű visszakeresni a függőséget. Injektálásnál ez leegyszerűsödik, mivel minden osztály tartalmaz egy referenciát, amit előzőleg konstruktorból vagy beállító metódusok segítségével a konténer beállított. Egy egyébként is komplex üzleti folyamatot tovább bonyolít, ha a függőségek implementálása visszakereséssel történik.

Egy jobb fejlesztőeszköz használatával az injektálás implementálása akár automatizálható is lehet. Teljesen passzív módon viselkedik az injektálás a visszakereséssel szemben, amíg a visszakeresésnél változhat, például a kulcs kinullázódhat, ami alapján vissza lehet keresni a függőséget vagy a függőség értéke nullázódhat, addig ez az injektálásnál nem fordulhat elő. Elkerülhetővé válik többek között egy nem kívánt `NullPointerException`. A kivétel visszakeresésnél csak futás időben derül ki, viszont injektálásnál, főképp annotációval, már fordítási időben kapunk információt az ilyen esetekről.

6.4 Beállító metódus vagy konstruktor?

Az aktuális problémától függ. A beállító metódus előnye, hogy többször is futtatható, konstruktorban viszont csak egyszer, a példányosításkor állítható be függőség. Attól függően, hogy fontos-e a függőségek futási időben történi regisztrálása és módosítása, kell eldöntenünk, hogy melyik injektálást válasszuk. A konstruktorral történő beállítás maga után vonja kissé megköti a fejlesztő kezét. A konstruktor használatánál az esetleges logikát a konstruktorban vagy konstruktorokban a függőségek beállításával együtt kell implementálni. Ha ezt is el szeretnénk kerülni, akkor legcélszerűbb a beállító metódusokon keresztül meghatározni a függőségeket. Ezáltal sokkal rugalmasabb lesz a kód, a függőségek regisztrálása és meghatározása, nem jár túl sok plusz munkával.

6.5 Függőségek visszakeresése

A keretrendszer normál esetben képes a függőségek visszakeresésére. Viszont előfordulhat olyan eset, amikor a függőségi hierarchiát nem tudja értelmezni. Ez akkor történhet meg, amikor a konfigurációs fájl rosszul van felépítve, általában programozói hiba, a feladat rossz megközelítése okozhatja. Alapesetben a függőségek a

konfigurációs fájlban felsorolt sorrendben fognak kialakulni. Ha van két osztály B és C, és a konfigurációs fájlban az alábbi sorrendben szerepelnek:

```
<bean id="c" class="szakdolgozat.beandependency.C"/>
<bean id="b" class="szakdolgozat.beandependency.B"/>
```

12. forráskód: Függőségek definiálása nélkül a konfigurációs fájl

akkor B függ C-től. Tegyük fel, hogy használnak egy közös osztályt, amelyiknek értékét beállítja B és aztán ezt használja fel C. Ha előbb C-re hivatkozunk, akkor a megszorítás sérül, mivel B nem tudta beállítani a megfelelő értéket. Ezt könnyen kiküszöbölhetjük a függőség logikus felépítésével és a depends-on attribútum használatával:

```
<bean id="c" class="szakdolgozat.beandependency.C" depends-
on="b"/>
```

13. forráskód: Függőségek definiálásával a konfigurációs fájl

Ettől függetlenül a Spring nem garantálja a függőségek kialakításánál, hogy feltétlen figyelembe veszi a definiálás sorrendjét. A sorrendet csak a *depends-on* attribútummal lehet befolyásolni. A fenti eset elkerülése végett érdemes használni ezt az attribútumot.

7 MVC

Az MVC betűszó a *model-view-controller* tervezési minta rövidítése. A tervezési minta lényege, hogy különválasztja a kódot modellre és nézetre. Ezek között a kontroller biztosítja a kapcsolatot. Ha a felhasználó a felhasználói felületen módosít valamilyen adatot, akkor azt a kontroller átvezeti a modellbe. A nézet elkészítésekor a kontroller adja át a megfelelő adatot a megjelenítéshez.

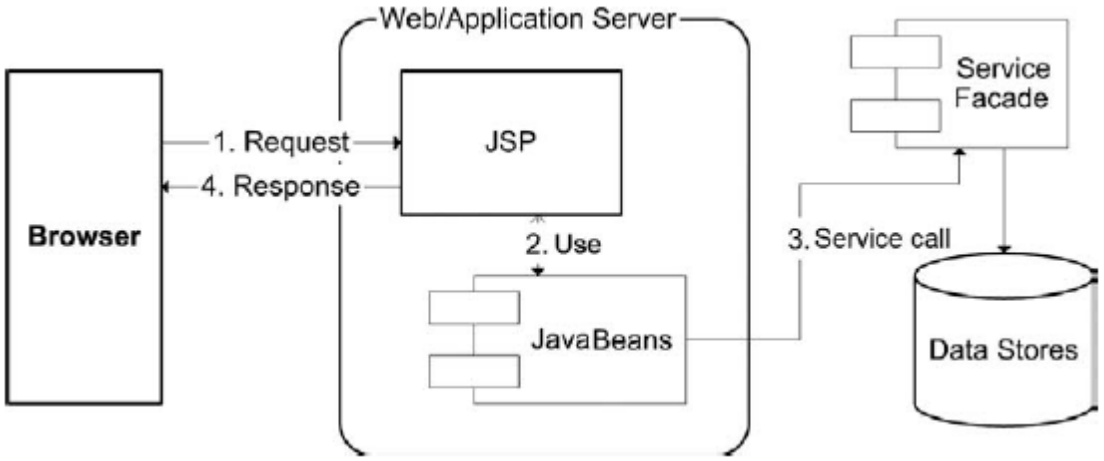
A modell azokat az adatokat reprezentálja, amit a felhasználói felületen meg kell jeleníteni, java környezetben ez általában egy java bean.

A nézet a modell által reprezentált adatokat jeleníti meg, megfelelően formázva. Ez a réteg biztosítja a felhasználói interakciót. Minden párbeszéd a felhasználóval a nézeten keresztül zajlik.

A kontroller reprezentálja az üzleti logikát és biztosítja, hogy a felhasználó által módosított értékek a felhasználói felületről átkerüljenek a modellbe. Java környezetben ez általában valamilyen servlet. A servlet kapja meg a GET illetve a POST kéréseket a kliensektől, amiket aztán a megfelelő réteg felé továbbít.

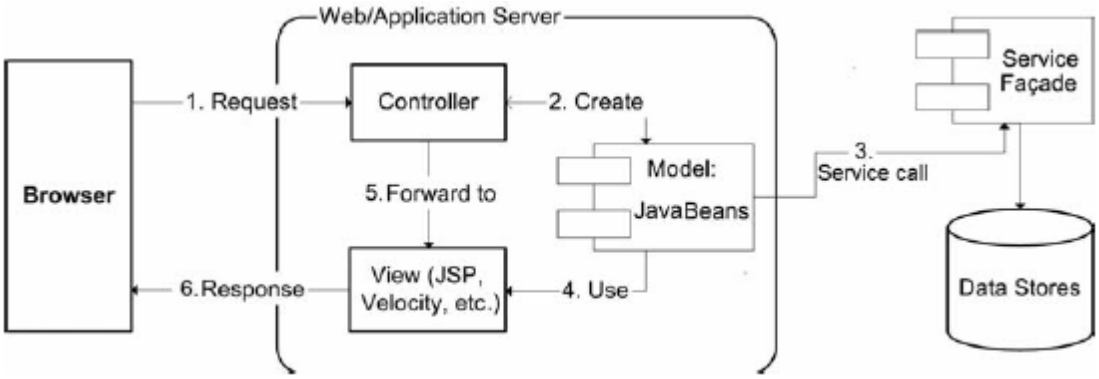
Spring esetében a servletek között legfelső szinten a `DispatcherServlet` helyezkedik el. Minden kérés hozzá érkezik be először és ő delegálja a kéréseket a többi servlet felé. Lehetőség van saját servlet definiálására is.

Az MVC-nek két implementációja használatos a mai fejlesztéseknél függetlenül a Springtől. Az egyik megközelítés, hogy a JSP oldal van a középpontban. Keveredik az MVC funkcionalitása, a kontroller és a nézet is a JSP oldalban van definiálva. Egyedül a modell különül el. A modell általában *POJO(Plain Old Java Object)* vagyis egyszerű java objektum. A nézet ezt az objektumot jeleníti meg, miközben felhasználja a nézetben implementált üzleti logikát is. Ez nagyobb alkalmazásoknál kevésbé javasolt, mivel átláthatatlan lesz az alkalmazás felépítése, illetve az alkalmazás módosítása rengeteg munkával jár. A kérést és a kérés kiszolgálását is a JSP oldal végzi, ami biztonsági kérdéseket is felvet. Nehézkesen lehet elkülöníteni az oldalakat jogosultság szerint.



3. ábra: Kevert MVC modell

A másik megközelítés szerint a modell a nézet és a kontroller teljesen elkülönül. A kontroller látja el a kérések fogadását, amit továbbít a nézetnek. A nézet pedig válaszol a kérésre. A nézet megjeleníti a kontroller által létrehozott modellt. Ebben az esetben a JSP oldal csak a megjelenítést végzi a kérések alapján. Az üzleti logika csak a kontrollerben van implementálva.



4. ábra: Tiszta MVC modell

8 Spring Web MVC

A Spring MVC lehetővé teszi számunkra, hogy rugalmas alkalmazásokat készítsünk. Támogatja a fent említett két különböző típusú MVC tervezési mintát. Az egész MVC megvalósítása a `DispatcherServlet` köré épül. A `DispatcherServlet` fogadja a kéréseket és továbbítja a megfelelő kezelő felé. Az alapértelmezett kezelő egy nagyon egyszerű, úgynevezett `Controller`, ami egy vezérlő interfész. Csak a modell és nézet kéréseit, illetve ezekre a válasz metódusokat tartalmazza. Természetesen ez nem elegendő egy nagyobb alkalmazás fejlesztéséhez, ezért ezt az interfészt, mint őst felhasználva építhetünk rá egy fát. A Spring is ezt teszi, így több kezelőt előre implementáltak a fejlesztők számára, mint például a `AbstractController`, `AbstractCommandController` és `SimpleFormController`. Saját vezérlő esetén tipikusan ezek használandók ősként, ezen interfészek kiterjesztésével írhatunk bármilyen saját kezelőt. Természetesen az őskontrollert célszerű jól megválasztani, mivel ha nem tartalmaz az alkalmazás formokat, akkor nem célszerű a `SimpleFormController`-ből leszármaztatni a saját vezérlőnket.

A keretrendszer nézet támogatása nagyon rugalmas. Szinte bármilyen nézetet kezelhetünk vele. Ha speciális nézetet akarunk használni, ami nincs beépítve a keretrendszerbe akkor csak meg kell írni hozzá a saját vezérlőnket. Normál esetben a `ModelAndView` példány a nézet nevét és a megfelelő objektumokat tartalmazza. A nézet névfeloldása rendkívüli mértékben konfigurálható, lehet például bean név vagy property fájl alapján, illetve implementálhatunk saját `ViewResolver` osztályt, amelyben saját szabályok alapján oldjuk fel a neveket.

A modell a java Map interfészére épül, amelynek köszönhetően teljesen absztrakt módon kezelhető a nézet. A fentebb említettek miatt a Spring MVC mintával készült alkalmazások esetében a nézet szabadon cserélhetővé válik, illetve egy alkalmazáson belül több nézettechnológiát is használhatunk.

8.1 Összekapcsolhatóság más MVC implementációkkal

Általában egy régebbi alkalmazást szeretnénk Springre átírni, illetve egy régebbi alkalmazáshoz szeretnénk hozzáírni, viszont nem akarjuk a régi technológiát használni, inkább haladva a technológiával más megoldásokat keresünk, de a meglévő munkánkat nem szeretnénk eldobni. Ebben az esetben előkerül a kompatibilitás kérdése, miszerint a régi rendszerrel képes lesz-e együttműködni az új?

Előfordulhat az is, hogy nem akarja a fejlesztő a Spring által implementált web MVC-t használni, de szeretné a Spring egyéb moduljait beépíteni az alkalmazásába. Ebben az esetben a Springet, mint komponenskönyvtárat használhatja az eredeti alkalmazásában, emiatt könnyedén integrálható a Spring a már meglévő rendszerbe.

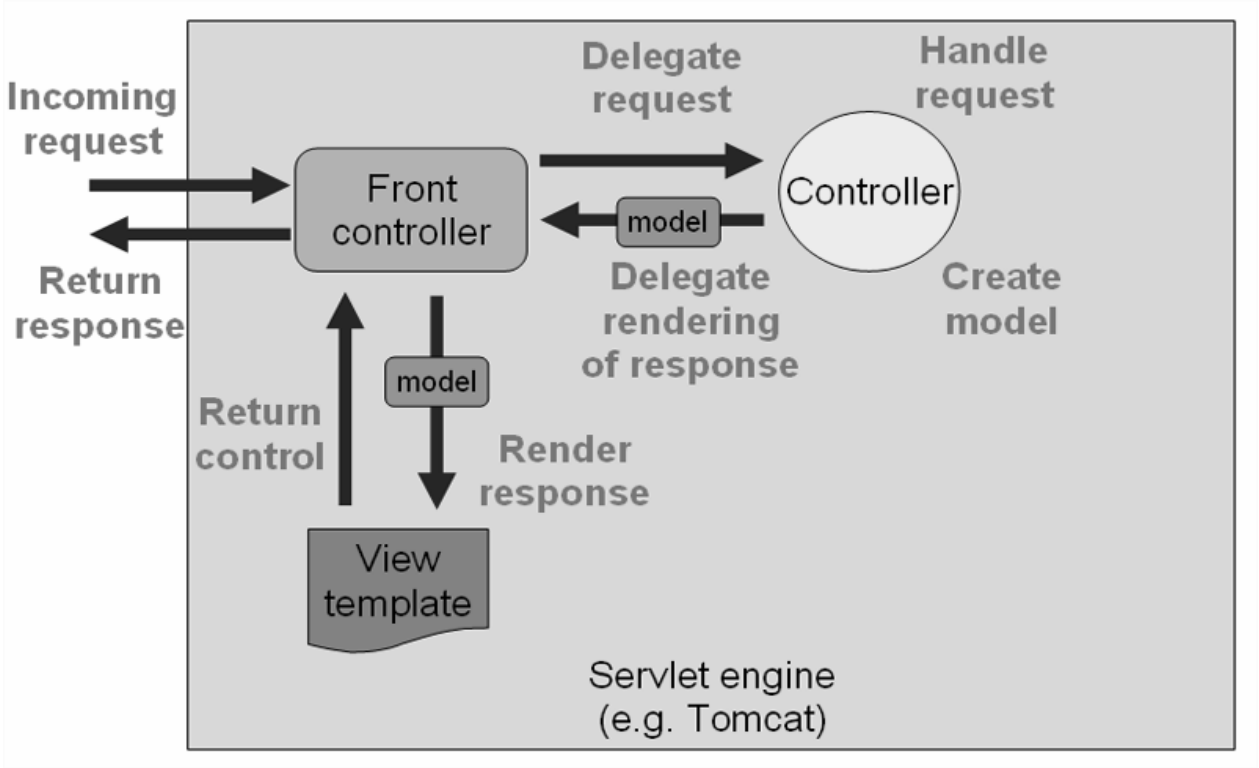
Spring Web MVC képességei:

- Szabályok és szerepkörök szétválasztása
- Rugalmas és visszafelé kompatibilis konfigurálhatóság
- Adoptálható
- Újrahasználható üzleti logika

- Testreszabható adatelérés és validáció
- Rugalmas nézettípus támogatás
- Flexibilis modell átalakíthatóság
- Lokalizálhatóság és téma választás
- Saját JSP komponenskönyvtár(Spring 2.0)
- Beanek élettartamának menedzselése

8.2 DispatcherServlet

A Spring Web MVC keretrendszer egy kérésvezérelt keretrendszer. Egy központi servlet köré épül, amely fogadja a kéréseket, majd a megfelelő kontroller felé továbbítja. A `DispatcherServlet` teljes mértékben integrálva van az `Inverison of Control` konténerrel, ezáltal a Spring összes funkciója kihasználható általa.



5. ábra: DispatcherServlet működése

A *DispatcherServlet* a *web.xml*-ben deklarálandó. A *web.xml* fájl egy konfigurációs fájl, amely tartalmazza a webes alkalmazás beállításait. Itt történik többek között a servletek és a kezdőoldal definiálása.

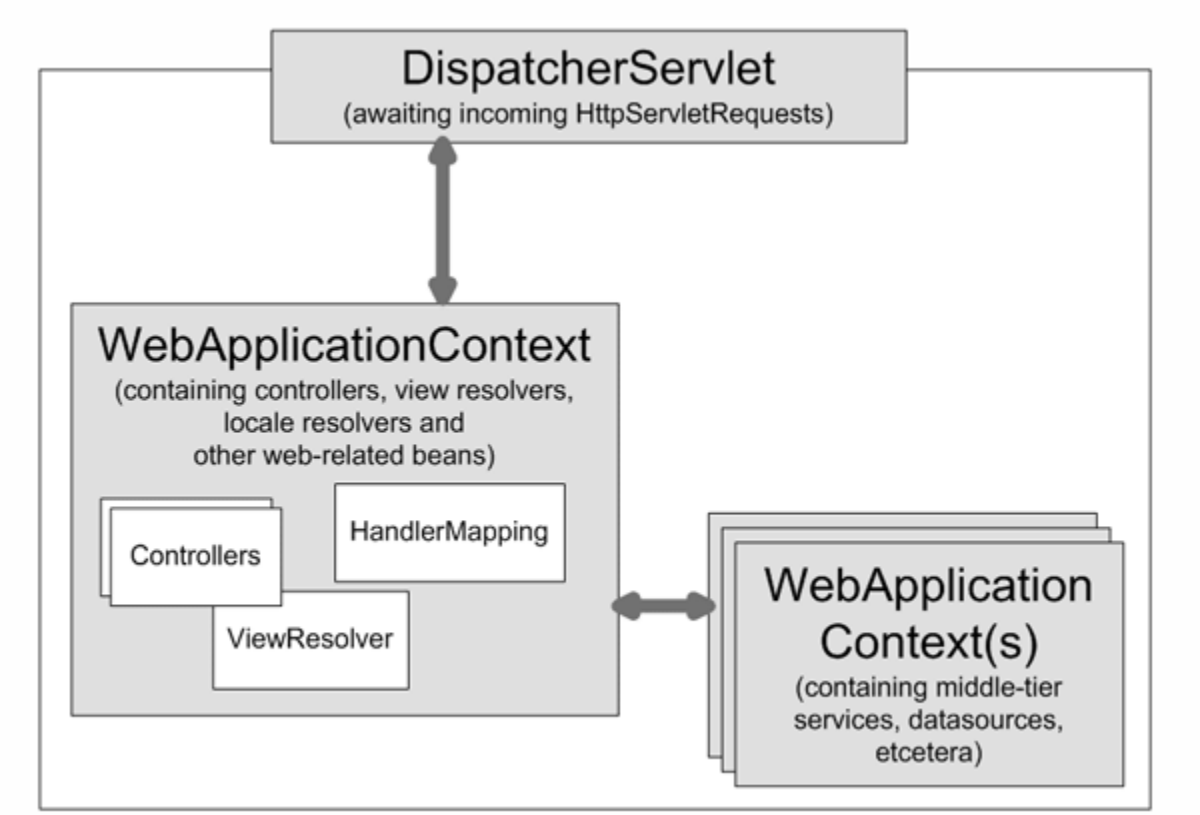
```
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.
                        DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>

</web-app>
```

14. forráskód: DispatcherServlet definiálása

Meg kell adni, hogy melyik servletet használjuk és, hogy azt hanyaggyára szeretnénk betölteni. A sorrendnek akkor van jelentősége, ha több servletet is használunk. Továbbá a servlet működéséhez szükséges még a servlet neve és az url minta, igény esetén több minta is megadható. Így lehetőség nyílik hogy a kéréseket a minta alapján a megfelelő nézetfeloldóhoz delegáljuk. Név alapján lesznek a servletek megkülönböztetve. A minta alapján kapja meg a kéréseket a servlet. Ez a servlet definíció nem Spring specifikus, szabványos J2EE servlet konfiguráció. Látható, hogy a példa servlet csak a *.form mintának megfelelő kéréseket fogja megkapni a *DispatcherServlet*-től. Spring-ben a *DispatcherServlet*-nek van egy saját *WebApplicationContext* példánya. A *WebApplicationContext* példánynak van saját hatásköre. A hatáskör szabályozza az objektumok elérését. A *WebApplicationContext* az *ApplicationContext* osztály kiterjesztése, rendelkezik olyan extra metódusokkal, amelyek például a témák és lokalizáció használatát teszik lehetővé a web alkalmazásban.



6. ábra: Kontextus hierarchia a Spring Web MVC-ben

A keretrendszer inicializálásnál megkeresi az összes olyan XML-t, amelynek a neve a definiált servlet nevével kezdődik és illeszkedik a *„-servlet.xml”* mintára. A servlet névkonvenciója a következő: *[servletnév]-servlet.xml*. Az XML fájlt a *WEB-INF* könyvtárban kell elhelyezni a projecten belül. Ahhoz hogy a példa servletet elérjük lennie kell egy */WEB-INF/pelda-servlet.xml* fájlnak a projecten belül, amely tartalmazza a servlet által elérhető beaneket.

A Spring *DispatcherServlet* sok speciális beant tartalmaz, amelyek feldolgozzák a kéréseket, illetve létrehozzák a kéréshez tartozó nézetet. Ezeket a beaneket a keretrendszer már tartalmazza, konfigurálásukat a *WebApplicationConext*-ben lehet elvégezni. Alapértelmezetten nem kell őket konfigurálni.

8.3 Controllers(vezérlők)

A vezérlőt olyan komponensek alkotják amelyek vezérlik az alkalmazás viselkedését. Tipikusan interfészek gyűjteménye. Minden olyan feladatot ellátnak, amely a kontroller hatáskörébe tartozik, mint például kérések fogadása és feldolgozása. A kontroller értelmezi a felhasználótól kapott adatokat és formázza azt a modellnek megfelelően. Majd a modell a nézeten keresztül reprezentálódik a felhasználó számára. A Spring a kontrollert nagyon absztrakt módon implementálja. Ezzel lehetőséget biztosít a fejlesztő számára, hogy saját kontrollereket hozzon létre. Számtalan típusú kontrollert tartalmaz a keretrendszer, többek között form specifikus és parancs alapú kontrollereket. Ennek megfelelően létezik egy ős interfész, amelyből a többit származtatva kialakul a keretrendszerben implementált hierarchia. Az ős interfész a *org.springframework.web.servlet.mvc.Controller*, amely a következőképpen néz ki:

```
public interface Controller {
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

15. forráskód: A Controller interfész felépítése

Egyetlen metódus implementálását követeli meg, amely fogadja a kéréseket, feldolgozza, majd egy *ModellAndView* objektummal tér vissza, amit a *DispatcherServelt* fog átadni megjelenítésre. Látható, hogy elég általános módon lett implementálva a *Controller* interfész, viszont ezen kívül számos más kontroller van implementálva a keretrendszerben, rengeteg funkcióval. Viszont abban mindegyik kontroller implementációja megegyezik, hogy kérést kell fogadnia, azt fel kell dolgoznia, végül egy *ModelAndView* objektummal kell visszatérnie.

8.3.1AbstractController és WebContentGenerator

A Spring által implementált összes kontroller az *AbstractController* osztályból származik, ennek köszönhetően lehet gyorsító tárazni és beállítani a *mimetype* attribútumot. Több szolgáltatást is nyújt ez az osztály:

- *supportedMethods*: ennek segítségével tudja eldönteni a vezérlő, hogy mely metódusokat kell elfogadnia. A két metódus általában a *GET* és a *POST*, de ez a fejlesztő által felülbírálható. Ha egy olyan kérés érkezik, amelyhez nincs metódus engedélyezve, akkor arról a kliens informálva lesz, kap egy *ServletException* kivételt.
- *requiresSession*: Azt mondja meg, hogy a kontrollernek munkája során szüksége van-e HTTP session-re vagy sem. Szintén a *ServletException* kivétel kerülhet át a kliens oldalra, ha olyan hívás következik be, amelyhez session szükséges.
- *synchronizeSession*: Ez akkor használatos, ha a kontrollert szinkronizálni szeretnénk a felhasználó aktuális HTTP session-jével.
- *cacheSeconds*: Ha a kontrollert gyorsító tárazni szeretnénk, akkor egy pozitív egész számot kell megadni. Alapértelmezetten a beállított érték -1, ebben az esetben nem történik gyorsító tárazás. Az itt megadott szám alapján őrzi meg a szerver a kliens számára a kontroller állapotát.
- *useExpiresHeader*: Akkor kell beállítani, ha a HTTP 1.0 szabvánnyal szeretnénk az alkalmazásunkat kompatibilissé tenni. Ekkor a generált válasz fejlécében egy *Expires* attribútum lesz. Alapértelmezetten az értéke *true*, minden kontroller visszafelé kompatibilis a HTTP szabvánnyal.
- *useCacheHeader*: Szintén kompatibilitási okból van rá szükség. A generált válasz fej részében a *Cache-Control* lesz. Alapértelmezett értéke ennek is *true*.

Amikor saját kontrollert készítünk felül kell bírálnunk a *handleRequestInternal(HttpServletRequest, HttpServletResponse)* metódust és implementálni a saját üzleti logikánkat, amelyekhez további metódusokat írhatunk, ha szükség van rá. Arra figyelni kell, hogy a kérést fel kell dolgozni, illetve egy *ModellAndView* objektumot kell visszaadni.

```
package szakdolgozat;

public class HelloVilagController extends AbstractController {

    public ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        ModelAndView mav = new ModelAndView("hello");
        mav.addObject("message", "Helló világ!");
        return mav;
    }
}
```

16. forráskód: Saját kontroller definiálása

A *PeldaController* számára beállítható tulajdonságok beállítása pedig a bean definiálásakor történik:

```
<bean id="peldaController" class="szakdolgozat.PeldaController">
    <property name="requiresSession" value="true"/>
</bean>
```

17. forráskód: Saját kontroller hozzáadása a Spring beállításaihoz

8.3.2MultiActionController

A Spring tartalmaz egy több akciót kezelő vezérlőt, amely segítségével több féle esemény kezelhető egyetlen kontrollerrel. Ez a kontroller a *org.springframework.web.servlet.mvc.multiaction* csomagban található. A kéréseket továbbítja a megfelelő kezelő felé. Megvalósítása delegált alapú. Akkor van rá szükség, amikor egy kontrollerben van implementálva sok egyszerű funkció, viszont

szeretnénk, hogy a komponens sok belépési ponttal rendelkezzen. Összesen kettő darab szolgáltatást nyújt:

- *delegate*: Kétféleképpen használhatjuk ezt a kontrollert. Az egyik módja, leszármazunk a *MultiActionController* osztályból és létrehozzuk a metódusokat, majd a *MethodNameResolver* segítségével feloldjuk a meghívandó metódus nevét. A másik lehetőség, hogy delegáltakat hozunk létre. A delegált egy olyan mutató, amely minden esetben függvényre mutat. Ezt a delegáltat adjuk oda a metódus neve helyett a *MethodNameResolver* számára, ami így megtalálja a meghívandó metódust.
- *methodNameResolver*: A kontroller számára szükséges, hogy fel tudja oldani a meghívandó metódus nevét. Alapja a beérkező kérés. A kérésben van specializálva a metódus. Három darab implementációja van jelenleg a keretrendszerben:
- *ParameterMethodNameResolver*: a kérésben kap egy paramétert, amelyet metódusnévként értelmez és így keresi meg a meghívandó metódust. Például: Ha a kérés a *http://www.inf.unideb.hu/index.view?testParameter=teszteltLe*, akkor az alábbi metódust fogja meghívni feltéve, hogy az létezik *testzteldLe(HttpServletRequest, HttpServletResponse).*
- *InternalPathMethodNameResolver*: a kérésben kapott fájlnevet fogja felhasználni, mint metódusnevet. Ha a kérés a *http://www.inf.unideb.hu/TeszteldLe.view* akkor a metódus, amit meghív, ha létezik a *testzteldLe(HttpServletRequest, HttpServletResponse).*
- *PropertiesMethodNameResolver*: egy property fájlt használ a metódusnevek felderítésére. Ha a fájl tartalmazza a */index/index.html=welcome* bejegyzést akkor, ha a kérés tartalmazza ezt a bejegyzést, a meghívandó metódus a *welcome(HttpServletRequest, HttpServletResponse)* lesz. Ez a

módszer a *PathMatcher*-el dolgozik, ha a fájl tartalmaz egy ilyen bejegyzést: */**/inde?.html*, akkor is működik a névfeloldás, mivel illeszkedik a mintára.

Az alábbi szignatúrával kell rendelkeznie egy *MultiActionController* osztályban definiált metódusnak:

```
public [ModelAndView | Map | void] metodusNev(HttpServletRequest,
HttpServletResponse [, Exception | AnyObject]);
```

18. forráskód: MultiActionController szignatúrája

Fontos megjegyezni, hogy ebben az esetben nem lehetséges a metódusok túlterhelése. Lehetőség van saját kivételkezelő írására is, amely a saját metódusaink kivételeit lekezelik. Bár a kivétel egy opcionális paraméter mindenképpen a *java.lang.Exception* vagy a *java.lang.RuntimeException* osztály leszármazottja kell, hogy legyen. Az *AnyObject* is egy opcionális paraméter, ez bármilyen osztály lehet. A visszatérési érték a felsorolt három típustól nem térhet el, nincs mód saját típusok alkalmazására sem.

8.3.3Command kontrollerek

A *command controller*, vagy parancs vezérlő lehetővé teszi, hogy az adat objektumok kommunikálhassanak egymással és dinamikusan bányásszanak adatokat a *HttpServletRequest*-ből, amelyekre az objektumoknak szüksége van. Nagy előnye a többi hasonló keretrendszerrel szemben, hogy az objektumoknak nem kell implementálniuk semmilyen keretrendszer specifikus interfészt sem. Számtalan beépített parancs vezérlő van implementálva a Spring keretrendszerben:

- *AbstractCommandController*: ez a kontroller az őse az összes parancsvezérlőnek. Ebből származtatva hozhatunk létre saját parancs vezérlőt. Ez az osztály semmilyen speciális funkcionalitást nem tartalmaz, csak validációs lehetőségeket kínál és lehetővé teszi, hogy a kérésből elérhetőek legyenek a paraméterek.

- *AbstractFormController*: Ezzel a kontrollerel formokat modellezhetünk, illetve kinyerhetjük a parancs objektumokat a kontroller számára. Miután a felhasználó a formot teljes egészében kitöltötte és kiváltódik egy submit esemény az *AbstractFormController* kinyeri a parancs objektumokat, validálja azokat és sikeres validálás esetén odaadja a kontrollernek. Akkor használandó, ha formokat szeretnénk használni, de nem akarjuk specifikálni, hogy melyik nézetet használjuk hozzá.
- *SimpleFormController*: egy olyan form kontroller, amelyik több támogatást ad, ha a formot a megfelelő parancs objektummal hozzuk létre. Létrehozhatunk egy parancs objektumot, egy nézet nevet a formhoz.
- *AbstractWizardFormController*: Saját varázsló kontroller létrehozására szolgál. Megköveteli a következő három metódus implementálását:
  - *validatePage():* az oldal validálásakor fog lefutni, mégpedig a validálás megkezdése előtt.
  - *processFinish():* a validálás után fog lefutni.
  - *processCancel():* a validálás megszakításakor vagy kivételek esetén fut le.

8.4 Handler mappings

A Spring alkalmazás környezet fájljában össze lehet rendelni az internet címeket a meghívni kívánt vezérlőkkel. A Spring *HandlerMapping* implementációkkal azonosítja a meghívandó vezérlőket. Három beépített implementációja van: *BeanNameUrlHandlerMapping*, *SimpleUrlHandlerMapping* és *ControllerClassNameHandlerMapping*.

8.4.1BeanNameUrlHandlerMapping

A bean nevét az URL alapján azonosítja. Ha az URL */editaccount.form*, akkor a bean neve a */editaccount.form* kell, hogy legyen. Ez a megoldás kisebb

alkalmazásoknál használható, mivel nem tartalmazhat helyettesítő karaktereket és szolgáltatáskérést.

```
<beans>
  <bean id="handlerMapping" class="org.springframework.web.
      servlet.handler.BeanNameUrlHandlerMapping"/>
  <bean name="/editaccount.form" class="org.springframework.web.
      servlet.mvc.SimpleFormController">
    <property name="formView" value="account"/>
    <property name="successView" value="account-created"/>
    <property name="commandName" value="account"/>
    <property name="commandClass" value="samples.Account"/>
  </bean>
<beans>
```

19. forráskód: SimpleFormController konfigurálása

Ebben az estben az összes */editaccount.form* kérést a *SimpleFormController* osztály fogja megkapni és feldolgozni. Természetes ez csak akkor fog működni, ha a *web.xml* fájlban közöljük az alkalmazással, hogy melyik kezelőt használja, ennek definiálása a következő:

```
<web-app>
    <servlet>
        <servlet-name>pelda</servlet-name>
        <servlet-class>
               org.springframework.web.
               servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>pelda</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

20. forráskód: web.xml fájl felépítése url minta megadásával

Definiálni kell egy servletet, amely az összes olyan kérést megkapja, amely **.form* alakú URL-el érkezett. Ezzel már a keretrendszer meg tudja mondani, hogy az ilyen

kéréseket melyik feldolgozó felé továbbítsa. Alapértelmezetten, ha nincs egyetlen kezelő sem definiálva a *web.xml* fájlban, akkor a rendszer létrehoz egy *BeanNameUrlHandlerMapping* kezelőt.

8.4.2SimpleUrlHandlerMapping

A *SimpleUrlHandlerMapping* osztály lehetővé teszi a kérésben meghatározni, teljes névvel és helyettesítő karakterekkel, melyik vezérlő fogja kezelni a kérést.

```
<web-app>
    ...
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.
              servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

21. forráskód: Több url minta definiálása a konfigurációs fájlban

Ha ezt a konfigurációt alkalmazzuk, akkor a **.from* és **.htm* mintára illeszkedő összes kérést egy kezelő fogja kezelni. A regisztrálandó bean pedig a következő:

```
    <bean class="org.springframework.web.servlet.handler.
           SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /*/account.form=editAccountFormController
```

```
                /*/editaccount.form=editAccountFormController
                /ex/view*.html=helpController
                /**/help.html=helpController
            </value>
        </property>
    </bean>
```

22. forráskód: SimpleUrlHandlerMapping bean konfigurálása

Nem szükséges az *id* attribútum megadása, a *DispatcherServlet* meg fogja találni a kezelőt. A *value* tagben felsorolt mintákra illeszkedő kéréseket az a kezelő fogja lekezelni.

8.4.3ControllerClassNameHandlerMapping

Automatikusan generál URL útvonalakat a vezérlők osztályainak neveiből. Mindhárom *HandlerMapping* implementáció az *AbstractHandlerMapping* kiterjesztése így megosztják az alábbi tulajdonságokat:

- *interceptorok*: Az alkalmazott elfogók listáját tartalmazza.
- *defaultHandler*: Ha nem talál feldolgozót, akkor az itt megadottat hívja. Kivétel esetén nem az alapértelmezett feldolgozó fog meghívódni, hanem a kivétel fog a kliens oldalra válaszként generálódni, 404-es kivétel formájában.
- *order*: Ez a tulajdonság fogja sorba rendezni a feldolgozó leképezéseket. A rendezés a *org.springframework.core.Ordered* interfész alapján történik.
- *alwaysUseFullPath*: Ha ez a tulajdonság *true*, akkor Spring a teljes elérési utat fogja használni a feldolgozó megkeresésére. Ha *false*, ami egyébként az alapértelmezett, akkor az aktuális servlet megfeleltetésben használt elérési utat használja. Például, ha egy servlet a */testing/** kifejezéssel lett összekötve a kezelővel és az *alwaysUseFullPath* tulajdonság értéke *true*, */testing/viewPage.html* lesz használva, ha a tulajdonság értéke *false*, akkor pedig *a /viewPage.html*.

- *urlDecode*: A *HttpServletRequest* olyan URL-ekkel és URI-kal tér vissza, amelyek nem kódoltak. Ha nem akarjuk, hogy kódoltak legyenek mielőtt egy *HandlerMapping* felhasználja a megfelelő feldolgozó megtalálására állítsuk át *true*-ra az értékét.
- *lazyInitHandlers*: Megengedi a lusta inicializálását a *singleton* tervezési mintájú feldolgozóknak.

BeanNameUrlHandlerMapping-et használja alapból a Spring keretrendszer, ha nincs más definiálva a Spring definíciós fájl(ok)ban.

8.5 HandlerInterceptor interfész

Az elfogók hasznosak, ha valamilyen műveletet szeretnénk bizonyos kérésekkor elvégezni. *HandlerInterceptor* interfész három metódust kínál. Egyet, ami az aktuális feldolgozó előtt fog végrehajtódni, egyet ami a feldolgozó után, és egyet ami a teljes kérés befejezésekor.

- A *preHandle* metódus: *boolean* értéket ad vissza alapértelmezetten és a feldolgozás előtt fog lefutni. Felhasználhatjuk ezt a metódust arra, hogy megszakítsuk vagy folytassuk a feldolgozási folyamatot. Amikor ez a metódus igaz értéket ad vissza a feldolgozási lánc folytatódik. Ha hamissal tér vissza, a *DispatcherServlet* feltételezi, hogy az elfogó maga kezeli a kérést, és nem folytatja a végrehajtását további elfogónak és az aktuális feldolgozónak sem.
- A *postHandle* metódus az amelyik minden feldolgozó futása után lefut. Ekkor van lehetőség a feldolgozás megszakítására, vagy a feldolgozás eredményének manipulálására, illetve annak felhasználására egy másik folyamat számára. Visszatérési értéke *void*.
- Az *afterCompletion* metódus visszatérési értéke is *void*. Ez a metódus akkor fut le, amikor az összes feldolgozó befejezte a munkáját, vagy megszakítottuk a folyamatot. Ekkor már a folyamat nem befolyásolható, az esetleges utómunkákat lehet itt elvégezni.

Annyi darab *HandlerMapping* és *HandlerInterceptor* beant definiálhatunk, amennyire szükségünk van, azt az egyet kell szem előtt tartanunk, hogy ezek ne ütközzenek egymással, egy URL-hez egy kezelőt rendeljünk hozzá.

8.6 Nézet feloldás

A Spring is, ahogyan a többi MVC keretrendszer is, implementálja a saját nézetfeloldó algoritmusát. Fontos, hogy a nézetet egy címhez rendeljük, amely címet a kérésben elküldünk a szervernek és a nézetfeloldó a cím alapján megtalálja a megfelelő nézetet, amit a kliensnek válaszban továbbít. A Spring ezt a kérdést is absztrakt módon közelíti meg. Nem kívánja a fejlesztőre erőltetni egyik nézetet sem, ennek érdekében több nézetfeloldó is implementálva van a rendszerben. Bármely típusú jelenleg divatos nézetre tartalmaz saját beépített nézetfeloldót, legyen az JSP, JSF vagy akár PDF. Ezek a nézetfeloldók akár egymás mellett is képesek dolgozni. Egy web alkalmazáson belül akár több típusú nézetet is használhatunk. Ha a beépített megoldások között nem találjuk a megfelelőt, akkor akár saját feloldót is írhatunk. Saját feloldó írása viszont körülményes és általában nincs is rá szükség.

8.6.1ViewResolver interfész

Minden nézetfeloldó a Spring Web MVC keretrendszerben egy *ModelAndView* példányt ad vissza. Nézeteket Springben névvel címezzük és a nézet meghatározók fogják őket azonosítani. Több beépített nézethatározó is van előre implementálva a rendszerben:

- *AbstractCachingViewResolver*: Egy absztrakt nézet meghatározó, ami a nézetek gyorsító tárazásáért felel.
- *XmlViewResolver*: A *ViewResolver* egy olyan implementációja, ami elfogad egy ugyanazon *DTD*-vel készült XML konfigurációs fájlt, amit a Spring bean gyár használ. Az alapértelmezett konfigurációs fájl a */WEB-INF/views.xml*. Nem támogatja a nemzetköziesítést.
- *ResourceBundleViewResolver*: Egy olyan *ViewResolver* implementáció,

ami bean definíciókat használ egy erőforráscsomagban (*ResourceBundle*). Az erőforráscsomag általában egy tulajdonság fájlban van meghatározva, ami a az alkalmazás gyökerében található. Az alapértelmezett fájlnév *views.properties*. Támogatja a nemzetköziesítést.

- *UrlBasedViewResolver*: Egy olyan *ViewResolver* implementáció, ami bean definíciókat használ egy erőforráscsomagban (*ResourceBundle*). Az erőforráscsomag általában egy tulajdonság fájlban van meghatározva, ami a classpath-ban található. Az alapértelmezett fájlnév *views.properties*.
- *InternalResourceViewResolver*: Egy kényelmesen használható alosztálya az *UrlBasedViewResolver* osztálynak, ami támogatja az *InternalResourceView*-t, azaz Servletek-et és JSP-ket. A nézethez tartozó osztályok minden ezen feloldó által generált nézethez megadhatók a *setViewClass(..)* metódussal.
- *VelocityViewResolver* / *FreeMarkerViewResolver*: Egy alosztálya az *UrlBasedViewResolver* osztálynak ami támogatja a *VelocityView*-t vagy *FreeMarkerView*-t.

Ha *Java Server Pages(JSP)* nézet technológiát használunk, akkor használható a *UrlBasedViewResolver* nézetfeloldó. Ez a nézet meghatározó lefordít egy nézet nevet egy URL-re és átadja a kérést a *RequestDispatcher*-nek a kérés teljesítésére.

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

23. forráskód: UrlBasedViewResolver nézetfeloldó definiálása

Ha a nézet neve *teszt*, a nézet meghatározó át fogja adni a kérést a *RequestDispatcher*-nek, ami továbbküldi a */WEB-INF/jsp/teszt.jsp* –nek. Ha

különböző nézet technológiákat keverünk egy webes alkalmazásban, akkor használhatjuk a *ResourceBundleViewResolver*-t, ami a megfelelő nézetet fogja biztosítani a kérés számára.

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.
               view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

24. forráskód: ResourceBundleViewResolver nézetfeloldó definiálása

A *ResourceBundleViewResolver* megvizsgálja az alapnevet, ami könyvtár nélkül megadott állománynév. Az eredmény egy *ResourceBundle* lesz, ami egy erőforrásfájl. Minden a *ResourceBundleViewResolver* által kezelt nézethez, a *[nézet név].class* tulajdonság értékét használja nézet osztálynak és a *[nézet név].url* értékét nézet internetcímeként.

8.6.2Nézet átirányítás

A kontroller alapesetben visszatér a logikai nézet nevével, amelyhez a nézetfeloldó megtalálja a megfelelő fizikai nézetet, a nézettechnológiát is definiálva ezzel. JSP nézettechnológia esetén az átirányítást a *InternalResourceViewResolver* vagy *InternalResourceView* osztály végzi. A servlet API-t használva a *RequestDispatcher.forward(..)* vagy *RequestDispatcher.include()* metódusok segítségével lehet átirányítást végezni.

Szükség lehet átirányításra, még mielőtt a nézet megjelenítésre kerülne. Általában akkor fordul elő, ha egy kontrolt *POST* metódussal hívunk meg, illetve egy formot *POST*-olunk. Legtöbbször a helyzet fokozódik és nemcsak átirányítást kell végezni, de a kérést delegálni kell másik kontrolnak, illetve formnak, amely akár adatbázist is igénybe vehet a feladatához, illetve hozzátehet a már meglévő adatokhoz újakat. Előfordulhat az az eset is, amikor olyan szituációba kerül a nézet, hogy nem jut el a megjelenítésig, mert a

felhasználó az előző form állapotát szeretné megtekinteni. Ekkor nemcsak egy POST, hanem egy GET is érkezik a kliens oldalról. Ez az eset még kezelhető is, mivel az oldal, amelyről visszanavigál a user nem rendelkezik állapottal. Ha viszont egy kitöltött formról navigálunk az előző oldalra, akkor két dolog történhet: a form megőrzi az állapotát, illetve nem őrzi meg. Ez beállítható konfigurációs fájlok segítségével. Rendszerint a felhasználó frissíti is az oldalt, ezt egy GET parancs kiadásával teszi, ami hatására a formot alaphelyzetbe kell hozni kivéve, ha egy olyan folyamat közepén vagyunk, ahol ez adatvesztéssel jár.

Az átirányítás nézettechnológia függő része az alkalmazásnak. Használható a nézet által támogatott eszközkészlet, ekkor a technológia szabályainak megfelelően jár el a keretrendszer, illetve használható a Servlet API is, amely segítségével általánosan közelíthető meg ez a probléma. A Spring keretrendszer több megoldást is kínál a problémára, amelyek a fentieket kibővítve alkalmasak bármilyen üzleti logika implementálására.

8.6.3RedirectView

Egyik módja a nézetátirányításnak, hogy a `RedirectView` osztályt használjuk a Spring keretrendszerből. Ebben az esetben a `DispatcherServlet` nem fogja használni a normál névfeloldó mechanizmust. A `RedirectView` egyszerűen meghívja a `HttpServletResponse.sendRedirect()` metódust, ami a kliens oldalon egy szimpla HTTP átirányításként fog megjelenni. A modellben szereplő összes attribútum HTTP kérés paramétereként fog viselkedni, átadódik a kliens oldalra. Ez azt jelenti, hogy a modellben minden attribútumnak olyan objektumnak kell lennie, amely `String` típusra konvertálható. Ennek oka, hogy a válasz a kliens számára egy típus nélküli `String`.

Ha ezt a megközelítést választjuk és a nézet kontrollerek által készül, akkor érdemes az URL-t, amelyre átirányítunk, beinjektálni a kontrollerbe, mégpedig a nézet nevek alkalmazásával.

8.6.4redirect: prefix használata

A fent részletezett *RedirectView* tökéletesen működik, ha a kontroller saját maga kezeli a nézetet, viszont a kontroller nem kap semmilyen visszajelzést arról, hogy az átirányítás megtörtént-e vagy sem. Könnyen beismerhető, hogy ez a megközelítés egy olyan alkalmazásnál, ahol a kliensekkel a kapcsolatot folyamatosan szeretnénk biztosítani, illetve meg szeretnénk győződni arról, hogy a kliens oldalra sikeresen megérkezett a válasz, akkor nem megfelelő. Ráadásul a kontrollerbe kell injektálni a nézetek nevét is, ami sok plusz munkával jár.

Erre a keretrendszer biztosít egy olyan megoldást, ahol konkrétan megmondható, hogy átirányítás szükséges és hová. Erre használatos a *redirect: prefix*. Szintaktikája a következő:

```
redirect:/my/response/controller.html
```

25. forráskód: Átirányítás relatív url segítségével

vagy

```
redirect:http://myhost.com/some/arbitrary/path.html
```

26. forráskód: Átirányítás abszolút url segítségével

Az eltérés csak a címzésben van. Az első egy relatív címzés, amely azt mondja, meg hogy az aktuális pozícióhoz képest hol van az a nézet, ahová átirányítást kell végezni.

A második egy abszolút címzés, a teljes URL szerepel a nézet nevében. Itt is a kontrollerbe kell injektálni a nézetek neveit abszolút vagy relatív címzéssel. Ezzel a módszerrel sem kapunk választ arra, hogy az átirányítás kliens oldalon mit váltott ki. Az ilyen prefixel ellátott kérést a *UrlBasedViewResolver* fogja feldolgozni és megtalálni a nézetet.

8.6.5forward: prefix használata

Szintén a *UrlBasedViewResolver* osztály fogja kezelni a kérést és megtalálni a hozzá tartozó nézetet. Akkor használatos, ha több nézettechnológiát vegyítünk. Készül egy *InternalResourceView a* nézethez , amely tartalmaz egy URL-t. Az URL alapján lehet azonosítani a nézeteket, ez azért hasznos mert több nézettípus estén az URL információval szolgál a kontrollernek, hogy melyik nézettechnológiát kell használnia. Szintén injektálni kell a kontrollerbe a nézeteket.

8.7 Lokalizáció

A Spring Web MVC támogatja a nemzetköziesítést más néven a lokalizációt. Mindezt a *DispatcherServlet* segítségével hajtja végre, amely a *LocalResolver* objektumot használja fel a megvalósításhoz. Az üzenetek így a kliens oldalon az adott felhasználó nyelvén érkezhetnek. Amikor egy kérés érkezik a *DispatcherServlet* megvizsgálja a kérés szövege alapján az aktuális nyelvet és ez alapján történik a kiszolgálás. Mindezt a *RequestContext.getLocale()* metódus segítségével éri el. Az ehhez szükséges elfogók és feloldók az *org.springframework.web.servlet.i18n* csomagban találhatóak és a megszokott módon XML-ek segítségével konfigurálhatóak.

8.7.1AcceptHeaderLocaleResolver

Ez az osztály a lokalizáció meghatározására a böngésző által küldött kérés fejlécét használja fel, ebből szedi ki a kliens gép nyelvét. A fejlécbe általában bekerül az operációs rendszer típusa, amelyből a nyelv könnyedén meghatározható. Erre a kérés *Accept-Language* tulajdonsága alkalmas.

```
GET /index.html HTTP/1.1
Host: www.example.com
Accept-Language: en
```

27. forráskód: HTML fejléc nyelvre vonatkozó információt hordoz

8.7.2CookieLocalResolver

A kliens oldalon tárolt süti is alkalmas lokalizáció felderítésére. Egy *CookieLocalResolver* definiálása az alábbi módon történik a szerver oldalon:

```
<bean id="localeResolver" class="szakdolgozat.
          CookieLocaleResolver">
    <property name="cookieName" value="clientlanguage"/>
    <property name="cookieMaxAge" value="100000">
</bean>
```

28. forráskód: Sütiket kezelő bean definiálása

Három tulajdonságot lehet beállítani a süti definiálásánál, ha csak lokalizációra használjuk:

- *cookieName*: a süti nevét adja vissza.
- *cookieMaxAge*: Az az idő, amíg a süti a kliens oldalon megtalálható, illetve amíg érvényes. Ha ez az érték -1, akkor a böngésző bezárásával a süti automatikusan törlődni fog.
- *cookiePath*: A süti láthatóságát befolyásolja. Az adott útvonaltól a fában lefelé elhelyezkedő útvonalak láthatják csak a sütit.

8.7.3SessionLocalResolver

A session felhasználásával a felhasználó kérései alapján lehet megállapítani az aktuális nyelvet. A szükséges információt a session alatt érkező kérésekből nyerhetjük ki.

8.7.4FixedLocaleResolver

A *LocalResolver* egy nagyon egyszerű implementációja, mindig egy előre konfigurált lokalizációval tér vissza.

8.7.5LocalChangeInterceptor

Definiálható úgy is a lokalizáció, hogy a felhasználó választhassa meg a nyelvet. A fent említett módszerek tökéletesen működnek, viszont abban az esetben, amikor a felhasználó nem a saját nyelvű operációs rendszer vagy böngészőt használ, akkor olyan nyelvet kénytelen használni, amit a rendszerünk rákényszerít. Ehhez egy elfogót kell deklarálni a Spring konfigurációs fájlban, a *LocalChangeInterceptor*-t. Minden nyelvváltáskor a *setLocal()* metódus fog meghívódni.

```
<bean id="localeChangeInterceptor" class="org.
springframework.web.servlet.i18n.
LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver" class="org.springframework.
           web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping" class="org.springframework.
           web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
          <list>
               <ref bean="localeChangeInterceptor" />
               <ref bean="themeChangeInterceptor" />
          </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

29. forráskód: A nyelvi beállításokat kezelő bean konfigurálása

Az összes *.view mintára illeszkedő kérés esetén a kérés stringjében benne lesz a nyelv is az alábbi módon:

```
http://www.sf.net/home.view?siteLanguage=nl
```

30. forráskód: Nyelvi információt tartalmazó url

Ebben az esetben a kérés alapján a válaszként megkapott kérés nyelve német lesz.

Normál esetben a felhasználó által választható nyelv van implementálva a rendszerekben, amelyre egy felhasználó által beállított értéket használnak. Így minden felhasználó a neki megfelelő nyelvet választhatja. Például egy nagy, többnyelvű közösségi oldalon regisztráció során a kérés-, süti- vagy session által szolgáltatott információk alapján határozzák meg a nyelvet, amelyet regisztráció után a felhasználó saját maga átdefiniálhat.

8.8 Témák használata

Egy web alkalmazás esetén alap igény a felhasználók részéről, hogy mindenki egyéni stílust tudjon használni. Ahhoz, hogy ez teljesüljön Springben a *org.springframework.ui.context.ThemeSource* osztályt kell alkalmazni. A *WebApplicationContext* interfész terjeszti ki ezt az osztályt, de delegálja a kéréseket az aktuálisan implementált példánynak. A delegált alapértelmezetten egy *org.springframework.ui.context.support.ResourceBundleThemeSource* osztály, amely az alkalmazás gyökerében található property fájlokból tölti be az aktuális témát. Természetesen olyan rugalmas a keretrendszer, hogy lehetőség van saját *ThemeSource* írására is. Ebben az esetben létre kell hozni egy beant, amelyet egy fenntartott névvel beregisztrálunk az alkalmazásba, a fenntartott név a *themeSource*. Az alkalmazás környezete detektálja ezt a beant és használatba veszi.

A property fájlban elhelyezhető bármilyen az alkalmazásban használ, témára vonatkozó bejegyzés, mint például:

```
css=themes/theme_en_US.css
flag_image=images/english_animated.gif
```

31. forráskód: Témák definiálása

A kulcsok, amelyekkel az elemeket regisztráljuk a nézetben hivatkozásként jelennek meg. JSP oldalnál a hivatkozás az alábbi módon néz ki:

```
<%@ taglib prefix="spring"
            uri="http://www.springframework.org/tags"%>
<html>
```

```
   <head>
      <link rel="stylesheet" href="<spring:theme
                code="styleSheet"/>" type="text/css"/>
   </head>
   <body background="<spring:theme code="background"/>">
      ...
   </body>
</html>
```

32. forráskód: Témák használatának módja JSP oldalon

Alapértelmezetten a keretrendszer üres prefixet használ. Üres prefix esetén a keretrendszer az alkalmazás gyökeréből próbálja meg betölteni a property fájlt. A témák használata is nemzetközesíthető.

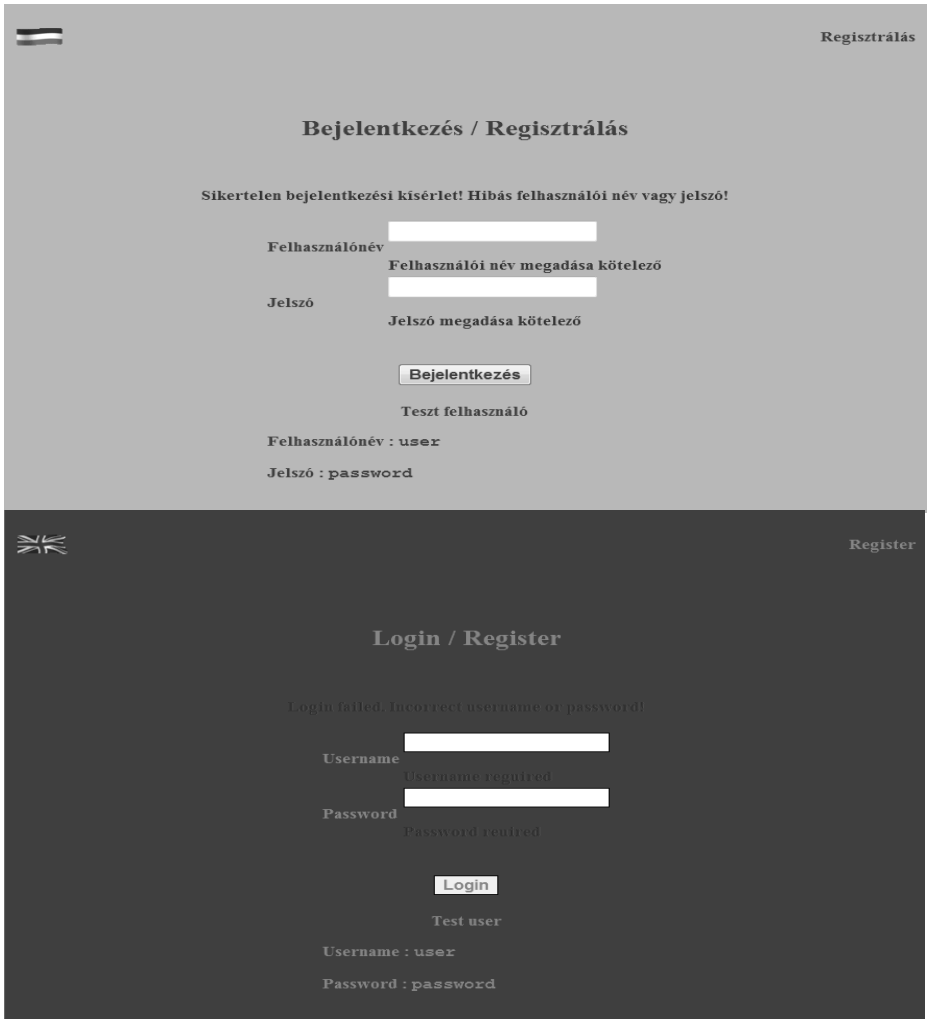
8.8.1Téma feloldó

Több téma esetén el kell dönteni, hogy melyik témát használjuk. A döntés történhet lokalizáció alapján, a felhasználó döntése alapján, a rendelkezésre álló sávszélesség alapján vagy bármelyik más tényezőt figyelembe véve. A *DispatcherServlet* megkeresi a *themeResolver* beant és eldönti, hogy melyik *Themeresolver* implementációt kell használnia. Az alábbi témafeloldó mechanizmusokat támogatja a Spring:

- *FixedThemeResolver*: Az alapértelmezett témát állítja be, ehhez a *defaultThemeName* propertyt kell használni.
- *SessionThemeResolver*: A felhasználó által használt session alapján állítja be a témát. Az egész session-re érvényes, viszont új session kezdésével érvényét veszti.
- *CookieThemeResolver*: A kiválasztott téma a sütiben tárolódik kliens oldalon. Amíg a süti létezik addig a téma a sütiben definiált lesz.

A keretrendszer támogatja a *ThemeChangeInterceptor* használatát is, amely kérésenként változtatja a témát, ha a kérés tartalmazza a megfelelő paramétert.

A szakdolgozat során elkészített alkalmazásban a témát és a nyelvet együtt kezelem. A lokalizáció során meghatározódik a property fájl alapján a használt stílus is.



7. ábra: A téma és a nyelv kezelése az alkalmazásban

A nyelv meghatározása a beérkezett kérés alapján történik. Minden kérés egy *GET* parancs elküldésével kezdődik a kliens oldalon. Ez a HTML kérés tartalmaz egy fejlécet, amelyben benne van a kliens oldalon használt nyelv. A nyelv alapján betöltődik a megfelelő property fájl, amely tartalmazza az oldalon elhelyezendő statikus szövegek nyelvi megfelelőjét. Ez a fájl tartalmazza a stílust leíró fájl elérési útvonalát is, illetve egyéb nyelvi információkat.

```
css=themes/theme_en_US.css
flag_image=images/english_animated.gif
```

33. forráskód: Téma meghatározása property fájl alapján

8.9 MultipartResolver

A Spring rendelkezik beépített fájl feltöltés támogatással. Ennek megvalósítása a MultipartResolver objektummal történik, amely az *org.springframework.web.multipart* csomagban található. Továbbá támogatja a MultipartResolver használatát. Alapértelmezetten nincs lekezelve ez a képessége a keretrendszernek. Ha használni szeretnénk, akkor engedélyezni kell. Az engedélyezés úgy történik, hogy létrehozunk egy *MultipartResolver* példányt, amelyet az alkalmazás majd használni fog. Ezután minden kérést megvizsgál a Spring és ha nem tartalmaz *Multipart* kérést, akkor nem történik semmi, viszont ha a kérést tartalmazza azt, akkor a definiált *MultipartResolver* példányhoz kerül a kérés.

A feloldó regisztrálása az alábbi módon történik:

```
<bean id="multipartResolver" class="org.springframework.web.
          multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize" value="100000"/>
</bean>
```

34. forráskód: MultipartResolver bean definiálása

vagy:

```
<bean id="multipartResolver" class="org.springframework.web.
        multipart.cos.CosMultipartResolver">
    <property name="maxUploadSize" value="100000"/>
</bean>
```

35. forráskód: MultipartResolver bean definiálása

Természetesen ez csak akkor fog működni, ha a megfelelő jar-t átadjuk az alkalmazásnak. A két féle definiálás kétféle jar implementálását követeli meg. Az

CommonMultipartResolver definiálása esetén a *common-fileupload.jar*, míg a *CosMultipartResolver* esetén a *cos.jar* fájlt kell használni.

Ha a *DispatcherServlet* detektálja, hogy a kérésben *multi-part* attribútum található, akkor megkeresi hozzá a feloldót, amelyet implementáltunk és továbbítja neki a kérést. A feloldó a *HttpServletRequest* objektumot becsomagolja egy *MultipartHttpServletRequest* objektummá, amely támogatja a fájlok feltöltését.

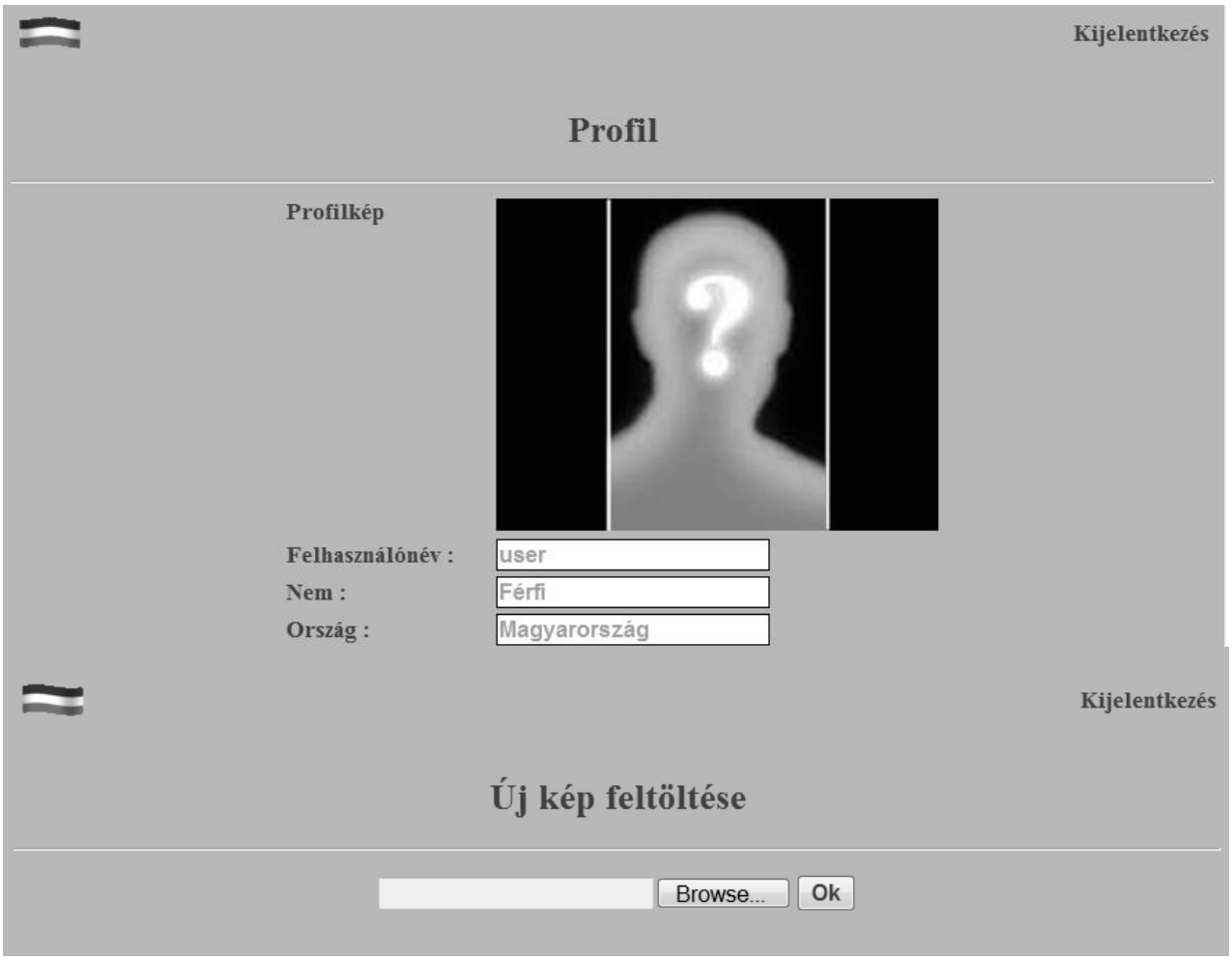
Először létre kell hozni egy formot, amelyen lehetővé tesszük a fájlok feltöltését.

```
<html>
    <head>
        <title>Fájl feltöltése</title>
    </head>
    <body>
        <h1>Fájl feltöltése</h1>
        <form:form commandName="fileUpload" enctype
                    ="multipart/form-data">
        <p align="center"><input type="file" name="file"
          value=<spring:message code="browse"/>><input type
                    ="submit"
          value=<spring:message code="ok"/>></p>
        </form:form>
     </body>
</html>
```

36. forráskód: Fájl feltöltése formon keresztül

A formon található egy *enctype* nevű attribútum, amely jelzi, hogy egy *multipart* kérés fog érkezni a szerverhez. Az *enctype* mező egy titkosított mező, a böngésző feladata, hogy dekódolja a mező értékét. A feltöltést a *Jakarta Commons' FileUpload* és a *Jason Hunter's COS* implementációkkal valósítja meg a Spring. Mindkét implementáció elismert és évek óta használatos a java alapú webalkalmazások készítése során.

Az alkalmazásban a *Jakarta Commons' FileUpload* megoldást választottam.



8. ábra: Fájl feltöltése az alkalmazásban

8.10 Spring komponens könyvtár használata

Ahogyan azt már fentebb láthattuk a Spring keretrendszer nagyszerűen képes együttműködni a különböző nézetekkel. Ezt a képességét erősíti az is, hogy a *spring.jar* tartalmaz nagyon sok olyan implementációt, ahol a keretrendszer használata sokkal egyszerűbb les. A 2.0-ás verziótól kezdve a Spring tartalmaz olyan mechanizmusokat, amelyek segítségével könnyebb a JSP szintű komponenseket összekötni a Spring Web MVC keretrendszerrel. Összekötés alatt elsősorban az adatkötést kell érteni. A Spring implementálta saját komponens könyvtárát. Egyetlen egy hátránya van, hogy csak a Spring Web MVC keretrendszerrel együtt használható. Előnye

viszont, hogy nagyon sok kódtól menti meg a fejlesztőt. Minden komponensnek van attribútuma, amely a HTML attribútumokkal ekvivalens. A komponensek megjelenítése HTML4.01/XHTML szabvány szerint történik.

8.10.1 Konfiguráció

Ahhoz, hogy a Spring komponenseit használni tudjuk a nézet oldalon hivatkozni kell a Springet az alábbi módon:

```
<%@ taglib prefix="form" uri="http://www.springframework.
            org/tags/form" %>
```

37. forráskód: JSP oldalon elhelyezendő beállítások

A könyvtár leíró a *spring-form.tld* fájlban található, ahol a komponensek képességei részletezve vannak.

8.10.2 form tag

Ez a komponens a HTML *form* komponenseként jelenik meg, azzal az extra képességgel, hogy a belső komponensek elérik egymás attribútumait. Ha a formon belülről egy komponens szeretné elérni a form egy másik komponens attribútumát, akkor ezt a formon keresztül könnyedén megteheti. Egy parancs objektum kerül a *PageContext* objektumba, amelyen keresztül a belső komponensek operálnak.

8.10.3 Input tag

Egyszerű HTML inputként jelenik meg. Egyetlen egy plusz attribútuma van a text, amelyen keresztül beállítható illetve lekérhető a szöveges tulajdonsága.

8.10.4 Checkbox tag

Szintén HTML inputként jelenik meg kliens oldalon, típusa a HTML *checkbox* típusa. A kinyerhető értéke igaz vagy hamis, attól függően, hogy ki van-e jelölve vagy sem.

8.10.5 Radiobutton tag

Ez a komponens is HTML inputként fog megjelenni, a típusa pedig a HTML *radiobutton* típusa lesz. A *radiobutton* komponenseket csoportokba lehet szervezni ezzel biztosítva, hogy egyszerre csak egy legyen kijelölve.

8.10.6 Password tag

Típusa a HTML *password* típusa és HTML inputként fog a kliens oldalon renderelődni. A beírt értéket elfedi, viszont beállítható az is, hogy ne legyen a jelszó elfedve. Alappértelmezetten a jelszó a külvilág számára azonos karaktekként jelenik meg, például csillag vagy pötty formájában.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq"
                    showPassword="true" />
    </td>
</tr>
```

38. forráskód: Jelszó kezelésére használható Spring komponens

8.10.7 Select tag

A HTML *select* típusaként renderelődik. Tipikusan olyan helyen alkalmazzák, ahol több lehetőség közül kell választani nem feltétlenül egy darabot. Alapértelmezetten egy elem kiválasztását engedélyezi, viszont ez a *multiple* attribútummal felülbírálható.

```
<tr>
    <td>Skills:</td>
    <td>
         <select name="skills" multiple="true">
                 <option value="Nulla">Nulla</option>
                 <option value="Egy" selected="true">
                                Egy</option>
                 <option value="Ketto">Kettő</option>
         </select></td>
    <td></td>
```

```
</tr>
```

39. forráskód: Select tag használata

8.10.8 Option tag

A *radiobutton* tag-hez hasonló. A HTML *option* típusaként fog megjelenítődni. Több lehetőség közül egy választását teszi lehetővé.

```
<tr>
    <td>House:</td>
    <td>
        <select name="szamok">
            <option value="Nulla"
                    selected="true">Nulla</option>
            <option value="Egy">Egy</option>
            <option value="Ketto">Kettő</option>
            <option value="Harom">Három</option>
        </select>
    </td>
</tr>
```

40. forráskód: Option tag használata

8.10.9 Options tag

Az *option* taghez hasonló, azzal az egy különbséggel, hogy több elem kiválasztását is engedélyezi. Az *items* attribútumnak akár lista is megadható.

8.10.10 Textarea tag

A HTML *textarea* komponenseként fog renderelődni. Szöveges megjelenítésre alkalmas.

8.10.11 Hidden tag

HTML *input* tag-jeként renderelődik hidden attribútummal.

8.10.12 Errors tag

HTML *span* tag-jeként renderelődik. Hibaüzenetek közlésére használatos.

8.10.13 Tag-ek használata az alkalmazásban

A lehető legtöbb komponens felhasználására törekedtem az alkalmazás elkészítése során. A komponensek felhasználásánál törekedtem arra, hogy ne statikusan kerüljenek bele adatok, hanem megvalósuljon a dinamikus adatkötés, így tetszőleges mennyiségű adatot lehet kötni egyetlen komponenshez. A dinamikusság miatt például nem a *radiobutton* komponenst, hanem a *radiobuttons* komponenst használtam, aminek meg lehet adni egy listát, amellyel dolgozik.

9. ábra: Komponensek használata az alkalmazásban

8.11 Kivételek kezelése

A Spring biztosít egy *HandlerExceptionResolver* objektumot, amelyet a nem kezelt kivételek lekezelésére használhatunk, tipikusan kontrollerek által kiváltott kivételek kezelései kerülnek ide. Ehhez a *web.xml*-ben regisztrálni kell a kivételkezelőt. A keretrendszer tartalmaz egy előre megírt általános, minden kivételt elkapó kivételkezelőt amely az *org.springframework.web.servlet.handler.SimpleMappingExceptionResolver* csomagban található. Az egyetlen egy megkötés, hogy az elkapandó kivételeket definiálni kell. Lehet nézethez is kötni egy-egy kivételt, ekkor ha a kivétel bekövetkezik a kivételhez kötött nézetre fogja átirányítani a keretrendszer a kérést.

```
<bean id="exceptionResolver" class="org.springframework.web.
              servlet.handler.SimpleMappingExceptionResolver">
     <property name="defaultErrorView" value=""/>
     <property name="exceptionMappings">
          <value>
          java.lang.NullPointerException=nullPointerErrorView
          javax.servlet.ServletException=servletErrorView
          </value>
     </property>
</bean>
```

41. forráskód: Kivételek kezelését végző bean konfigurálása

A *defaultErrorView* property értéke írja le azt a nézetet, amelyikhez az átirányítás történik kivétel esetén.

8.12 Spring keretrendszer és más nézettechnológiák

Előfordulhat, hogy bizonyos adatok megjelenítése már az adott nézettechnológiával bonyolultnak bizonyulhat. Ekkor kell más nézettechnológiát is alkalmazni a megjelenítésre. Ilyen például, ha JSP technológiával mester oldalakat szeretnénk kezelni. Rengeteg plusz munka és egy csomó hibalehetőség mellett sem tudjuk teljes mértékben kihasználni a JSP előnyeit. A későbbi módosítások pedig szinte lehetetlenek. Ekkor érdemes lehet például a Velocity nézettechnológiával is foglalkozni. A

szakdolgozatban JSP nézettechnológiával és a köré épülő lehetőségekkel fogunk foglalkozni.

8.12.1 Java Server Pages(JSP)

A JSP már bizonyított, mint sikeres és könnyen használható technológia. Minden nézethez tartozik egy java osztály, amelyben akár az üzleti logikát is leírhatjuk. Sajnos ez a lehetőség általában a projektek MVC tervezésére is káros hatással van. Minden a kontrollerre és modellre vonatkozó információ belekerül a JSP oldalhoz tartozó java osztályba. Ezzel viszont a kód átláthatatlan lesz. Sokszor kerülnek olyan kódok a java osztályokban amelyeket például javascript segítségével kellene megoldani. A Javascript viszont nem túl kedvelt a fejlesztők körében, ezért inkább szerver oldalon oldják meg a kliens oldali feladatokat is. Ebben az esetben szokott előkerülni a saját komponens írása. A Spring ezt megelőzendő biztosít a fejlesztők számára több olyan lehetőséget, amellyel az ilyen jellegű hibák elkerülhetőek. A JSP nagyszerűen támogatja az MVC tervezési mintát.

8.12.2 message

Üzenetek megjelenítésére használatos. Az üzenetet a kódjával azonosítja, amelyet a *messageSource* beanből nyer ki. Az üzeneteket egy property fájlban kell felsorolni az azonosítójukkal együtt. A bean definiíciója az alábbi:

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans">
     <bean id="messageSource"
     class="org.springframework.context.support.
          ResourceBundleMessageSource">
     <property name="basename"><value>messages</value></property>
     </bean>
</beans>
```

42. forráskód: Üzeneteket kezelő bean konfigurálása

A property fájl pedig az alábbi módon néz ki:

```
greeting=Helló <b>Spring</b> Framework
required=Ennek a mezőnek a kitöltése kötelező!
```

43. forráskód: Üzeneteket tartalmazó property fájl

Az üzenetek lokalizálhatóak is, ezt a property fájlban kell jelezni a keretrendszer számára. Az üzenetek megjelenítése kliens oldalon a következőképpen történik:

```
<%@taglib                                                    prefix="spring"
uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<html>
        <head>
        <title>
              Pro Spring
              </title>
        </head>
        <body>
               <h1>
                     <spring:message code="login_register"/>
               </h1>
        </body>
</html>
```

44. forráskód: Üzenetek felhasználásának módja JSP oldalakon

A *message* tag használata nagyon egyszerű a *code* attribútum jelöli a kiírandó üzenetet, amelyet a *messageSource* bean majd visszakeres a property fájlban. Az alábbi attribútumokat lehet még beállítani a *messge* tag esetén:

- *code*: a kód, ami alapján be lehet állítani a megjelenítendő üzenetet.
- *arguments*: vesszővel elválasztott lista, amely átadódik a *messageSource* bean *getMessage()* metódusának. Ez alapján fog történni az üzenet visszakeresése.
- *text*: az az üzenet ami akkor fog megjelenni, ha az üzenet azonosítóját nem találja meg a *messageSource* bean. Általában a *getMessage()* metódus megkapja, mint argumentumot.

8.12.3 theme

A *themeResolver* bean definiálása után készíthetünk témával ellátott oldalakat. A feloldó megkeresi az összes témát, majd a megfelelőt alkalmazza. A témák használatának részletei a szakdolgozat téma kezelésével foglalkozó részénél található.

8.12.4 hasBindErrors

Az adatok kinyerése és validálása közben is váltódhat ki le nem kezelt kivétel, ezek elkapására szolgál a *hasBindErrors* tag. Hiba esetén a tag törzsében definiált utasítások hajtódnak végre, például üzenet megjelenítése:

```
<spring:hasBindErrors name="command">
There were validation errors: <c:out value="${errors}"/>
</spring:hasBindErrors>
```

45. forráskód: hasBindErrors tag használata

8.13 nestedPath

Segítségével a nézet más beágyazott objektumokat tud rendelkezésre bocsátani a komponenseknek. Az objektumok értéke másolódik, így a komponens, amelyik használni szeretné csak egy másolattal tud dolgozni. Egyetlen egy attribútuma van a *path*, amellyel a másolandó objektum útvonalát tudjuk beállítani.

```
<spring:nestedPath name="user"/>
```

46. forráskód: nestedPath tag használata

8.13.1 bind

Tipikusan adatok elérésére használatos és validációra. A *path* attribútum alapján találja meg azt az objektumot a keretrendszer, amelynek az értékét szeretné megtudni, illetve validálni.

```
<spring:bind path="name">
     <input name="name" value="<c:out value="${status.value}"/>">
     <span class="error">
          <c:out value="${status.errorMessage}"/>
     </span>
</spring:bind>
```

47. forráskód: bind tag használata

A kód alapján a *command* objektumot szeretnénk elérni az aktuális hatáskörből, annak is a *name* property-jét.

8.14 Annotáció használata és a kontrollerek

Az XML alapú konfigurálás nagyszerű lehetőség és sokat egyszerűsít a kódón, viszont 1.5-ös verziójú java keretrendszertől lehetőség van az annotáció használatára. Erre a Spring keretrendszer is fel van készítve. Az MVC a Spring 2.5-ös verziójától támogatja a kontrollerek annotációval történő konfigurálását. Nem kell a meglévő osztályainkat kiterjeszteni vagy interfészt implementálni.

8.14.1 @Controller

Ezzel az annotációval jelezhetjük, hogy az osztály egy kontroller. Az előzőekben kifejtett deklarációhoz képest itt nincs szükség interfész implementálására. Egyetlen dolgot kell csupán tenni, jelölni kell az annotációt:

```
@Controller
public class IndexController{
     public ModelAndView displayIndex(
     // logika
     }
}
```

48. forráskód: Osztály szintű annotáció használata

Ahhoz, hogy ez működjön szükség van a konfigurációs fájl fel kell készíteni az annotációra.

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

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/
         spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/
        context/spring-context-2.5.xsd">
   <context:component-scan base-package="szakdolgozat.web" />
</beans>
```

49. forráskód: Annotáció használatához szükséges beállítások

8.14.2 @RequestMapping

Ezt a fajta annotációt az URL-ek kontrollerhez kötésére használjuk.

```
@Controller
@RequestMapping("/index.htm")
public class IndexController{
     public ModelAndView displayIndex(
     }

     @RequestMapping("/jsp/reg.htm")
     protected ModelAndView listProductsHandler(
     …
     }
}
```

50. forráskód: RequestMapping annotáció használata

Ha osztály szinten akarjuk használni az annotációt, akkor az *AnnotationMethodHandlerAdapter* beant kell konfigurálnunk a servlet számára. Ha metódus szinten akarjuk használni, akkor pedig a *DefaultAnnotationHandlerMapping* bent kell megfelelően konfigurálni. Saját kezelő írása esetén mindenképpen konfigurálni kell az *AnnotationMethodHandlerAdapter* beant. A *DefaultAnnotationHandlerMapping* beant mindenképpen a Spring konfigurációs fájljában kell definiálni.

```
<context:component-scan base-package="com.apress.
          prospring2.ch17.web" />
<bean class="org.springframework.web.servlet.mvc.annotation.
          AnnotationMethodHandlerAdapter"/>
<bean class="org.springframework.web.servlet.mvc.annotation.
          DefaultAnnotationHandlerMapping"/>
```

51. forráskód: RequestMapping annotáció használatához szükséges beállítások

8.14.3 @RequesParam

Kérés paramétereinek az elérésére használható a kontrollerben. Lehet kötelező vagy opcionális. Ha nem jelöljük, hogy opcionális, akkor kötelezően léteznie kell. Azt, hogy opcionális legyen a *required* attribútum segítségével lehet beállítani.

```
@RequestParam ("productId", required="false")
```

52. forráskód: RequesParam annotáció használata

8.14.4 @ModelAttribute

Ha egy metódus egy objektum típussal tér vissza, akkor ezt az objektumot lehet a modellben használni, ha ezzel az annotációval ellátjuk. Az annotált metódus értékét a modellből tetszőlegesen el lehet érni.

```
@ModelAttribute("products")
public Collection<Product> populateProducts() {
     return this.productManager.findAllProducts();
}
```

53. forráskód: ModelAttribute annotáció használata

Előnye, hogy akár metódus paramétereként is használható az annotáció.

```
public String processSubmit(@ModelAttribute("product") Product
            product, BindingResult result,SessionStatus status) {
     this.productManager.saveProduct(product);
     return getSuccessView();
}
```

54. forráskód: ModelAttribute annotáció használata paraméterként

9 Összegzés

A Spring Web MVC mellett említésre méltó még webes alkalmazásfejlesztéshez a keretrendszer WebFlow- és Security modul implementációja. Mindkettő tökéletesen konfigurálható és használható a Web MVC-vel. A fejlesztők támogatása és a kód komplexitásának csökkentése volt a cél a keretrendszer megalkotásakor, ami jelenleg sikeresnek tűnik. Flexibilitásának köszönhetően egyre több projektnek lesz az alapköve. A szakdolgozatban elkészült példaalkalmazásokból látszik, hogy milyen rugalmasan konfigurálhatóak a Spring által implementált eszközök.

Minden főbb részhez készült egy kisebb alkalmazás, amelyik csak azt a részt hivatott bemutatni és kiemelni az előnyeit a keretrendszernek. Ezek a kis alkalmazások nem az üzleti logikára fektetik a hangsúlyt, hanem a keretrendszer által kínált eszközök használatának sokszínűségét hivatottak bemutatni. Készült még egy nagyobb alkalmazás, amelyben már az üzleti logika implementálása is szerepet kapott. Az alkalmazás képzeletbeli levelező listákat menedzselő oldalt testesít meg. Az alkalmazásban Spring és JSP eszközök találhatóak. A hangsúlyt a Spring által nyújtott formokra fektettem. Ezeken a formokon történik az adatbevitel, a validáció, üzenetek közlése a felhasználó számára, illetve a navigálás is a formok segítségével lett implementálva. Az alkalmazás úgy készült el, hogy az angol és magyar nyelvű képzeletbeli felhasználókat szolgálja ki. Ehhez nyelvi fájlok és nyelvenként különböző stílusok készültek. Mindenhol előtérbe helyeztem a saját kontrollerek és validátorok írását, ezzel is hangsúlyozva a Spring rugalmasságát.

A kitűzött célt az alkalmazás elkészítésével elértem. Kihasználtam minden olyan eszközt, amit a keretrendszer nyújt az MVC témakörben. Természetesen a feladatot könnyebben is meg lehetett volna oldani más modulok bevonásával, mint például *WebFlow* és *Security*, de azt nem tartottam célszerűnek, mert háttérbe szorult volna nagyon sok olyan lehetőség és eszköz, amely szorosan kapcsolódik a Spring Web MVC modulhoz.

10 Szakirodalom

[1] Apress Pro Spring 2.5 (Jan Machacek, Aleksa Vukotic, Anirvan Chakraborty, és Jessica Ditt)

[2] Apress Spring Web Flow (Erwin Vervaet)

[3] Johnson, Rod, Juergen Hoeller, Colin Sampaleanu, Alef Arendsen, Rob Harrop, Thomas Risberg, Darren Davison, et al. 2003. Spring documentation. *The Spring Framework.* http://www.springframework.org/documentation

[4] Belapurkar, Abhijit. 2004. Use continuations to develop complex Web applications: A programming paradigm to simplify MVC for the Web. developerWorks. http://www-128.ibm.com/developerworks/java/library/j-contin.html

[5] DeBolt, Virgina.: HTML és CSS szerkesztés stílusosan. Kiskapu kiadó, 2005.

[6] The Internet Society. 1999. Hypertext transfer protocol—HTTP/1.1. *World Wide WebConsortium.* http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[7] Springsource: http://www.springsource.org/

[8] SpringSource.org: http://www.springsource.org/

[9] Springdeveloper: http://www.springdeveloper.com/oscon/mvc.pdf

[10] Spring Web MVC: http://cchweblog.wordpress.com/2009/06/16/spring-mvc-resolving-locale-and-externalizing-locale-sensitive-text-messages/

[11] Témák használata: http://blog.inflinx.com/2009/10/08/theming-websites-using-spring-mvc/

[12] Patterns of Enterprise Application Architecture: Front Controller

[13] Spring.NET Application Framework

[14] Jolt winners 2006

[15] JAX Innovation Award Gewinner 2006

[16] Spring 3.0.2 release announcement

[17] VMware completes acquisition

[18] http://www.andygibson.net/blog/index.php/2008/08/28/is-spring-between-the-devil-and-the-ejb Spring VS EJB3

[19] "Pitchfork FAQ". http://www.springsource.com/web/guest/pitchfork/pitchfork-faq. Retrieved 2006-06-06.

[20] http://houseofhaug.wordpress.com/2005/08/12/hibernate-hates-spring Hibernate VS Spring

[21] Introduction to the Spring Framework

[22] Johnson, Expert One-on-One J2EE Design and Development, Ch. 12. et al.

[23] Patterns of Enterprise Application Architecture: Front Controller

11 Köszönetnyílvánítás

Elsősorban köszönöm az útmutatást és a sok konzultációs időpontot a témavezetőmnek, Dr. Adamkó Attilának. Továbbá szeretném megköszönni az Orgware Kft közreműködését, ahol informatikusként megismerkedhettem ezzel a nagyszerű keretrendszerrel, sőt komolyan elmélyedhettem a Spring rejtelmeiben. Köszönöm a cég alkalmazottainak, akikkel megvitathattuk a keretrendszer képességeit, illetve képet kaphattam a céges környezetben történő fejlesztés szépségeiről. Köszönöm a Debreceni Egyetemnek, aki nélkül a fent említett céggel nem is találkozhattam volna. Köszönöm a projektben részt vevő hallgató társaimnak, akikkel komolyabb szakmai vitákat folytathattam akár napokon keresztül egy adott probléma megoldására vonatkozóan.

Köszönöm szüleimnek, akik tanulmányaim során segítettek, mostak és főztek rám. Elnézték, ha hétvégén nem tudtam hazamenni munkahelyi elfoglaltságom miatt.

Végül köszönöm a Microsoft-nak, hogy a Microsoft Office 2010 béta szoftver segítségével elkészíthettem ezt a szakdolgozatot és közben legalább tucatnyi hibáról küldhettem jelentést.