Универзитет у Београду

Математички факултет

Презентација могућности JEE технологије – апликација електронски школски дневник

Студент:

Милан Крстић

Ментор:

Владимир Филиповић

доцент Математичког факултета

Београд, Јун 2011.

Садржај

I Предговор

Овај мастер рад је замишљен као увод у Јаву, односно JEE технологију. Идеја је да се читаоцу пруже основне информације о технологији, принципи, методологије и слично. Подразумева се да читалац поседује основно информатичко знање.

Рад је организован у три главне целине:

- Увод – кратак опис коришћених технологија;
- Апликација – приказ и организација апликације и пратеће базе података;
- Кôд апликације – приказ делова кода из апликације, уз одговарајуће коментаре;

Циљ тезе је упознавање читаоца са JEE технологијом, као и оспособљавање за даљи рад и усавршавање у овој технологији.

II Увод

II.1 Јава

Џејмс Гослинг, Мајк Шеридан и Патрик Ноутон су започели рад на пројекту под именом „храст“ *(oak)* 1991, године. Планирано је да се тај програмски језик користи за ручне уређаје, али је јако брзо прерастао своју намену. Из тог пројекта ће касније настати Јава програмски језик. Сан Мајкросистемс *(Sun Microsystems*) је 1995. објавио прву верзију, под именом „Јава 1.0“. Занимљиво је да је име Јава изабрано из листе случајних речи.

Разликујемо три издања Јаве:

- Јава стандардно издање *(Java Standard Edition - JSE)* - служи за развој апликација за кућне рачунаре, из којег су се развила остала издања;
- Јава микро издање *(Java Micro Edition - JME)* - „осиромашено” основно издање - служи за прављење апликација за такозване уграђене *(embedded)* уређаје - уређаје са ограниченим ресурсима, као што су мобилни телефони, уређаји за ТВ, итд.;
- Јава издање за пословну примену *(Java Enterprise Edition - JEE)* - надоградња стандардног издања која омогућава развој пословних апликација;

Првобитно је у свим називима стајала двојка – нпр. *J2ME*, али је 2006. године избачена из маркетиншких разлога.

Да би корисник могао да користи Јаву на свом рачунару, мора да инсталира или Јава извршно окружење (*JRE – Java Runtime Enviroment*) – ствари потребне за извршавање Јава кода на рачунару или Јава алат за развој *(JDK – Java Development Kit),* који је надскуп извршног окружења, тј. садржи исте ствари плус компилатор, алате за развој, итд.

Током 2006. - 2007. године, Сан је објавио већину кода Јаве под ГПЛ *(GPL - GNU General Public License)* лиценцом, осим малог дела над којим ни сам Сан нема ауторска права.

II.1.1 Принципи

Постоји пет основних принципа који су праћени приликом прављења Јаве:

- Да буде једноставна, објектно-оријентисана и лака за учење;
- Да буде робусна и сигурна;
- Да буде портабилна и независна од архитектура рачунара;
- Да има добре перформансе;
- Да буде интерпретирана, вишенитна и динамична;

II.1.2 „Пиши једном, покрећи свуда“

Једна од главних предности Јаве је портабилност – што значи, једном написан кôд ће се извршавати на свим платформама. То се постиже уз помоћ Јавине виртуалне машине *(JVM)*. Када се кôд написан у Јави компајлира, он се не преводи директно у машински код, који је специфичан за сваку платформу, већ у такозвани бајт-код. Бајт-кôд се интерпретира на виртуелној машини, која је писана специјално за различите верзије система.

Иако овај начин превођења и извршавања програма има велику предност – портабилност, он такође има и великуману – спорост. Јаву је одувек пратио глас да ради доста спорије од језика чији се кôд преводи и извршава директно у машинском коду. Међутим, обзиром да се Јава стално развија, да се ти процеси стално унапређују и оптимизују, као и увођењем тачно-на-време *(Just-in-Time)* компилатора, та разлика се знатно смањила (чак има оних који тврде да разлике више ни нема).

II.1.3 „Сакупљање отпадака” (*Garbage collection*)

Јава је наследила доста особина језика Ц и Ц++, али има и неколико битних разлика у односу на њих. Једна од највећих је рад са меморијом. У горе поменутим језицима програмер је дужан да води комплетну бригу о меморији – и да је алоцира и да је ослободи. Самим тим је то било подложно грешкама, па су се често дешавала такозвана цурења меморије – када програмер алоцира меморију, али заборави да је избрише.

Стога је у Јави тај систем измењен, па програмер једино треба да размишља о алоцирању меморије, док сам систем брине о брисању. Систем који се тиме бави се зове аутоматски сакупљач отпадака (*automatic garbage collector*) - сам ослобађа меморију када је више ништа не користи. Програмер не може експлицитно да га позове. Идеално време за његову активацију би било када програм не ради ништа друго. Гарантовано је да ће се сакупљач отпадака активирати ако програму недостаје меморије, у покушају да је ослободи како би извршавање програма могло да се настави.

II.1.4 Синтакса

Синтакса је у великој мери наслеђена од Ц++-а, па се зато сматра да је прилично лако Ц++ програмеру да се пребаци на Јава програмирање.

У Јави је скоро све класа, функција или објекат - осим основних типова података (бројни, знаковни и логички тип) и операција (петље - *if, while*, итд.). Такође, због поједностављивања, али и умањивања грешака и лошег дизајнирања програма, избачени су и вишеструко наслеђивање и преклапање оператора.

Уведена су и правила по питању именовања датотека, па тако датотека мора да се зове исто као и класа која је декларисана као јавна (*public*) у њему, док за приватне (*private*) класе то правило не важи. Екстензија датотеке мора да буде *.java*. Након превођења кода у бајт-код, добијамо датотеку под истим именом, али са екстензијом *.class* и он може да се покреће и извршава.

Функција која се позива приликом покретања програма се назива главна (*main*). Програм може имати више главних функција (али у различитим класама), с тим што се мора тачно нагласити која треба да се позове приликом покретања програма. За разлику од Ц/Ц++-а, главна функција овде прима само један аргумент:

```
public static void main(String[])
```

Параметри који се прослеђују главној функцији се углавном задају преко командне линије и добијају се у као низ знакова.

Стандардни Здраво свете! програм би у Јави изгледао овако:

```
public class HelloWorld {
public static void main(String[] args) {
     System.out.println("Hello, world!");
     }
}
```

По горе поменутим правилима, ову датотеку бисмо морали да назовемо „*HelloWorld.java*“ и из њега бисмо након компајлирања добили „*HelloWorld.class*“, који затим можемо да покренемо.

System.out.println је стандардни начин за штампање садржаја на стандардни излаз.

Поновно коришћење кода је и у Јави битна ставка - охрабрује се коришћење већ написаних библиотека било од стране самог програмера, било од других програмера, организација, компанија, итд. Те библиотеке се укључују у програм коришћењем кључне речи *import:*

```
import java.applet.*;
```

Треба поменути и концепт Јава зрна *(Java bean)* - то су класе писане по одређеним конвенцијама - те конвенције омогућавају коришћење, поновно коришћење, мењање и повезивање зрна. Да би нека класа била зрно, она мора да има јавни конструктор без аргумената, за сваки од атрибута те класе морају да постоје методе - чланови класе који враћају и постављају вредност тог атрибута и мора да буде серијализована.

II.1.5 Аплети

Аплети су Јава програми који се најчешће користе у оквиру ХТМЛ страна, на које се постављају на сличан начин као слике.
„Здраво свете“ пример аплета:

```
import java.applet.*;
import java.awt.*;

public class HelloWorld extends Applet {
     public void paint( Graphics g ) {
          g.drawString("Hello World", 50, 25);
     }
}
```

II.1.6 Сервлети

Сервлети су серверске Јава апликације, које примају захтеве од клијената (најчешће коришћењем ХТТП протокола) и генеришу одговарајуће одговоре (најчешће ХТМЛ стране). Као што само име каже, ови програми се извршавају на серверу и свој излаз приказују клијенту.

Они имају неколико предности у односу на *CGI* – не креира се нови процес за сваки захтев (ова разлика је нарочито уочљива ако је посао који треба да се обави прилично мали), већ само посебна нит. То такође значи мању потрошњу меморије, обзиром да се не прави копија самих скриптова за сваки захтев посебно, пошто је кôд сервлета већ у меморији и нити које он покреће користе тај код.

II.1.7 JavaServer Pages

Јава Серверске Странице - JCC (*JavaServer Pages - JSP*) су Јава технологија која служи за динамичко генерисање интернет страница, коришћем Јава програмерског окружења. JCC је Санов одговор на ПХП и АСП.

JCC кôд се убацује у ХТМЛ страну, уоквирен делимитерима <% и %>. На пример:

```
<%@ page import="com.foo.bar" %>
<%! int serverInstanceVariable = 1; %>
```

II.1.8 JEE - Јава издање за пословну примену

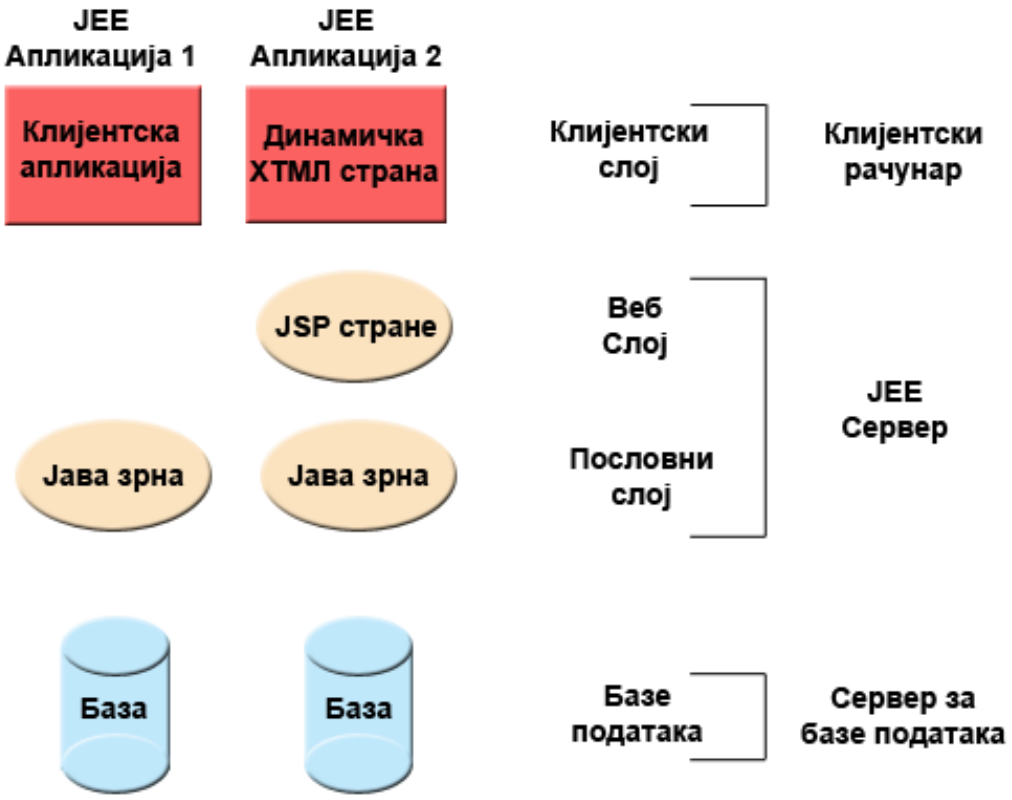
Јава је свој живот започела на десктоп рачунарима и тада је било велико питање да ли ће икада доспети на сервере. Данас, пословно издање Јаве је нешто што је чест избор приликом израде великих апликација.

По Сану, JEE чине сервиси, програмерски апликативни интерфејси(*API*) и протоколи - на пример повезивање на базу (*JDBC),* даљинско позивање метода (*RMI),* сервис за слање/примање порука (*JMS),* сервлети, портлети, итд.

JEE користи вишеслојни модел - JEE апликација се састоји из неколико слојева, па тако имамо, на пример, презентациони слој, пословни слој, слој за приступ подацима, итд. Овај приступ је бољи од стандардног двослојног (клијент-сервер) модела зато што је скалабилнији, чини апликацију лакшом за одржавање и даљи развој, итд.

JEE апликације се извршавају на апликационим серверима - то су сервери који су направљени тако да поштују JEE стандарде и који, самим тим, омогућавају да се на њима извршавају JEE апликације. Неки од најпознатијих апликационих сервера за JEE су: Глас

Фиш (*GlassFish),* Џеј Бос (*Jboss),* Апачи Џњронимо (*Apache Geronimo)*, итд. Тренутно актуелна верзија је Јава ЕЕ 6.



JEE је замишљена тако да се већина рутинског посла, тј. посла који је код свих апликација исти или веома сличан (сесије, трансакције, ...) обавља аутоматски, како би програмеру остало што више времена за развој и имплементацију пословне логике саме апликације. Такође, још једна велика предност JEE-а је то што је независна од платформе.

Међутим, било је и проблема. Пре свега, коришћење је било превише комплексно. Обзиром да је идеја била да се рутински процеси аутоматизују, било је потребно доста времена за писање конфигурационих датотека који би описали како апликација треба да поступи у одређеним ситуацијама (на пример приликом руковања сесијама). Ово је било нарочито упадљиво пошто није било могуће искључити коришћење делова који нису били потребни, тако да су све компоненте морале да буду конфигурисане, што је био додатни и непотребан напор. Такође, тиме су се добијале гломазне апликације које су садржале и потребне и непотребне компоненте.

Постојао је и проблем зависности - ако компонента А захтева компоненту Б, онда је компонента А била дужна да се побрине да јој је компонента Б доступна. Ту се онда јављао и проблем везивања компоненте Б у коду компоненте А (тј. ако би у неком тренутку требало променити ту компоненту, морао би да се мења код, а самим тим би морала поново да се компајлира и цела апликација). Такође, углавном није било могуће вршити тестове, зато што су за тестирање били потребни неки елементи које иначе обезбеђује JEE сервер.

Поменути недостаци су створили добре предуслове за развој надоградњи, такозваних оквира за JEE. О једном од најпопуларнијих причамо у следећем поглављу.

II.2 Спринг (Spring)

Дакле, Спринг је један од најпопуларнијих оквира за Јаву који отклања многе битне недостатке поменуте раније - нпр. није потребно конфигурисати делове који се не користе, зависност између компоненти решава сам оквир, омогућено је тестирање, итд.

Језгро Спринга чине два модула - инверзија контроле *(Inversion of Control)* и аспектно-оријентисано програмирање.

Инверзија контроле је већ поменута - када постоји зависност између неких компоненти, оне морају ручно да се повежу. Мана тога је, као што је већ речено, што би онда морао да се мења и сам код, што је нарочито проблематично ако је, на пример, потребно то радити приликом тестирања апликације или ако се укључују различите компоненте у верзији за развој и верзији за продукцију. Код Спринга је то решено тако што се те зависности подешавају ван кода (у конфигурационим датотекама), тако да нема мењања кода приликом тих измена. Такође је могуће имати различите конфигурације за различите потребе (тест, развој, продукција), а само повезивање компоненти врши сам оквир и програмер о томе не мора да брине.

Ево како би део конфигурационе датотеке (неке апликације која користи Спринг) који брине о инверзији контроле могао да изгледа:

```
<beans>
    <bean id="laserPrinter" class="com.example.LaserPrinter"/>
    <bean id="inkjetPrinter" class="com.example.InkjetPrinter"/>
    <bean id="printerService" class="com.example.PrinterService">
        <property name="printer" ref="laserPrinter"/>
    </bean>
</beans>
```

Овим су дефинисана три зрна - `laserPrinter`, `inkjetPrinter` и `printerService`.

Такође, тиме што је у `printerService` зрну у поље за атрибут под именом `printer` стављена као референца `laserPrinter`, обезбеђено је све што је потребно за већ поменуту инверзију контроле - даље је све на Спрингу, који се брине да зрно које се референцира увек буде доступно у зрну које га референцира. Ако се некада укаже потреба да се користи неко друго зрно, све што је потребно урадити је референцирати се на неко друго зрно и то је то - нема мењања кода, поновног компајлирања, итд.

Један од важнијих модула Спринга је модул за аспектно оријентисано програмирање (АОП). АОП је у неку руку надоградња објектно-оријентисаног програмирања - он уводи појам аспекта, који је у АОП-у оно што је класа у ООП-у. Као што и само име каже, аспекти се односе на неке аспекте апликације - нпр. на логовање одређених догађаја из апликације у неку датотеку. Ако треба, нпр., логовати позиве метода у некој апликацији, без АОП-а би било неопходно ручно додати наредбу за штампање на почетку сваке функције. Међутим, са АОП-ом, потребно је описати шта треба да се деси у одређеној ситуацији. Тако би у овом примеру са логовањем било потребно дефинисати аспект који ће се активирати приликом позива функције и који ће штампати потребне податке.

Још једна згодна примена АОП-а је и приликом комуникације са базом - ако је потребно да се свака комуникација са базом обавља у оквиру трансакција, без АОП-а би око сваког упита било потребно урадити додатне послове везано за отварање/затварање трансакција. Са АОП-ом је то знатно лакше - дефинише се аспект који ће сваку комуникацију са базом уоквирити трансакцијом и тиме је програмер ослобођен било каквог даљег програмирања у вези са трансакцијама.

Спринг садржи још модула, на пример - *MVC - Model View Controller* модул (олакшава израду интерфејса ка кориснику), модул за рад са базама (за рад са релационим базама података коришћењем *JDBC*-ја и других објектно-релационих алата за мапирање), модул за рад са трансакцијама (олакшава рад са трансакцијама), модул за тестирање (омогућава тестирање тиме што учитава све потребне податке, поставља потребне променљиве и тиме ствара окружење који би требало да симулира реалну ситуацију), итд.

II.3 Хајбернејт (Hibernate)

Хајбернејт је алат који омогућава мапирање објектно-оријентисаног модела на традиционалну релациону базу података. Дистрибуира се под *GNU Lesser General Public License* лиценцом. Тренутно је актуелна верзија 3.6.0.

Главна улога Хајбернејта, и оно за шта се најчешће користи, је мапирање Јава класа на табеле у бази података. То практично значи да, након што корисник исправно мапира класе и табеле, велики део посла прелази са њега на Хајбернејт - корисник неће морати да пише *SQL* код, већ ће све операције над табелама обављати коришћењем тих Јава класа. Наравно, остављена је могућност писања стандардног *SQL* кода, ако то корисник жели, али то није препоручљиво, зато што би то онда онемогућило једну веома корисну ствар коју доноси Хајбернејт, а то је преносивост. Наиме, ако се приликом комуникације са базом користи само Хајбернејт, онда све што треба урадити приликом мењања базе која се користи је промена драјвера у конфигурационој датотеци. Ако корисник има потребу да пише неки свој *SQL* упит, а жели да задржи могућност преношења, онда може да користи такозвани *HQL - Hibernate Query Language* - Хајбернејтов језик за писање упита.

Да би апликација користила Хајбернејт, он се, наравно, мора конфигурисати. Део главне конфигурационе датотеке за Хајбернејт би могао да изгледа овако:

```
<hibernate-configuration>
  <session-factory>
    <property name="show_sql">true</property>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLMyISAMDialect</property>
    <property name="hibernate.connection.driver_class">org.gjt.mm.mysql.Driver</property>
    <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/applabs</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">password</property>

    <mapping resource="org/example/Foo.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

У овом примеру, апликација користи *MySQL* базу података, која је покренута на локалном рачунару и на коју се логује са корисничким именом *root* и шифром *password*. Такође се и референцира на једну конфигурациону датотеку која мапира класу `Foo` на истоимену табелу.

Мапирање класа на табеле се врши преко *XML* датотека или преко анотација.

Ево како би могла да изгледа горе поменута датотека за мапирање класе `Foo` на истоимену табелу:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="org.somepackage.eg">
  <class name="Foo" table="foo_table">
     <id name="id" type="java.lang.Long">
       <generator class="sequence" />
     </id>
    <property name="captionText" column="caption_text" />
  </class>
</hibernate-mapping>
```

Овим је мапирана табела из базе под именом `foo_table` на класу `Foo`. Та табела у бази има две колоне - `id` и `caption_text`. Јава кôд класе `Foo` на коју је мапирана ова табела би изгледала овако:

```
class Foo
{
     private int id;
     private String captionText;
     // getters and setters
};
```

Дакле, поменуте табела и класа су сада мапиране и повезане, тако да се од сад у коду може користити класа `Foo` и са њом може да се ради све што је потребно, нигде се не пише *SQL* код. На пример, прављење нове инстанце класе `Foo` и њено убацивање у табелу у бази би изгледало отприлике овако:

```
SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();
Foo foo = new Foo();
foo.setCaptionText(some caption);
Transaction tx = session.beginTransaction();
session.saveOrUpdate(foo);
tx.commit();
session.close();
```

Наравно, Хајбернејт омогућава међусобно повезивање табела, у било ком облику (један на један, један на много, итд.). Овде би требало поменути још једну јако занимљиву могућност коју нам доноси Хајбернејт - такозвано лењо довлачење (*lazy fetching*). У пракси, то изгледа овако - ако се приликом мапирања класа у конфигурационим датотекама подеси да су класа А и класа Б у релацији асоцијације, Хајбернејт ће, ако је лењо довлачење укључено, „довући“ све објекте класе Б који су повезани са одговарајућим објектом класе А. Ово може да буде јако корисно, да скрати писање кода, да учини рад програмеру лакшим и бржим, али исто тако може и битно да утиче на перформансе апликације, ако се не користи како треба (на пример ако се из базе узима велики број објеката класе А, од којих сваки садржи велики низ објеката класе Б).

II.4 Апачи Ант (Apache Ant)

Апачи Ант је софтвер за аутоматизовање процеса компајлирања софтвера. Сличан је Make-у, али има и неколико битних разлика. Прва је та да је Ант писан у Јави, и самим тим је најбољи за коришћење у Јава пројектима. Друга битна разлика је та што Make користи обичан текстуални формат за конфигурисање самог процеса аутоматизације, док Ант користи *XML* датотеке (које се по правилу зову *build.xml*).

Ант је настао као помоћни алат у раду на другом пројекту, од којег је касније настао *Apache Tomcat* (сервер који подржава све потребне стандарде за извршавање сервлета; не треба га мешати са „обичним“ Апач сервером, који је имплементација „обичног“ ХТТП сервера). Програмерима је био потребан алат који би им помогао у компајлирању софтвера на различитим платформама, па су зато развили Ант (*Another Neat Tool*), који је касније премашио популарност онога због чега је и настао. Било је покушаја и да се направи наследник, али ниједан није био успешан.

Ипак, и Ант има неке мане, на пример, почетнику организација *XML* датотека за конфигурисање може бити превише комплексна и тешка за учење. Такође, конфигурациона датотека расте са величином пројекта, тако да у великим пројектима може бити и непрегледан, нарочито ако се не води рачуна о начину писања.

Основна организација конфигурационе датотеке би изгледала овако:

```
<project name="esd" basedir="." default="build">
  <property name="appname" value="${ant.project.name}" />

  <target name="clean" description="Deletes files from WAR directory">
      <!--  -->
  </target>
</project>
```

На почетку датотеке потребно је рећи о којем се пројекту ради, где се тај пројекат налази и која је предефинисана акција. У овом случају, ради се о апликацији под именом `esd`, чији је основни директоријум онај у којем се *build.xml* налази и предефинисана акција је `build`.

У оквиру елемента `property` могу да се дефинишу променљиве које се касније користе у датотеци и ова опција се углавном користи за подешавање директоријума.

Оно где се права акција одвија су елементи `target` - у њима се дефинишу конкретне акције, као и оно што оне раде - брисање датотека, прављење директоријума, копирање нових датотека на потребне локације, компајлирање апликације, као и њено убацивање на сервер и, на крају, чишћење директоријума.

У оквиру Ант конфигурационе датотеке се могу укључити (обично на почетку) и друге датотеке, које садрже додатне информације, на следећи начин:

```
<property file="local.properties" />
```

Са:

```
<property name="src.dir" value="src" />
```

дефинисана је променљива `src.dir` која има вредност `src`. Она ће бити коришћена касније, у референцирању под-директоријума директоријума *src*:

```
<property name="conf.dir" value="${src.dir}/conf" />
```

Ово је јако практично, зато што, ако се укаже потреба да се преименује директоријум *src*, биће потребно то изменити само на једном месту и остатак апликације ће радити без проблема. На овом примеру је уједно представљено и како се дефинишу и користе променљиве у Анту.

У оквиру неке акције може да се дефинише и од чега она зависи, тј. шта је све потребно извршити да би се испунили услови да се она изврши - на пример, пре компајлирања је потребно иницијализовати све директоријуме. То се постиже додавањем опције `depends="init"` у оквиру `target` елемента. Може се дефинисати и више акција, а такође и свака од позваних акција може зависити од неких других акција - нпр. акција за пуштање апликације у рад зависи од две акције - контра-акције (брисања претходне итерације апликације) и позивања акције за дистрибуирање апликације. А акција за дистрибуирање позива акцију која поново гради целу апликацију, а та акција даље празни директоријуме, компајлира апликацију, итд.

Било која од поменутих акција се покреће крајње једноставно - из командне линије нпр., ако треба покренути нпр. поменуту init акцију, само је потребно откуцати "`ant init`" у директоријуму где је *build.xml* и то је то.

Ове акције су и највећа снага Анта, пошто могу прилично да се прилагоде, да позивају системске функције, да аутоматизују и знатно убрзају неке процесе који се стално понављају. Такође, могуће је правити и корисничке акције, али то већ превазилази овај кратки увод.

II.5 Постгрес (Postgres, PostgreSQL)

Постгрес је објектно-релациони систем за управљање базама података (*ORDBMS*). Настао је из пројекта *Ingres*, који је развијан на универзитету Беркли. Прва верзија је објављена 1995. године, а прва верзија отвореног кода је објављена првог Августа 1996. Данас је Постгрес бесплатан софтвер, отвореног кода, који се развија учешћем великог број програмера-волонтера чији се рад координира преко интернета. Нове верзије се издају отприлике једном годишње, док се исправке издају чешће. Тренутно

актуелна верзија је верзија 9.0.3.

Неке од важнијих особина су:

- Могућност коришћења процедуралних језика (*PL/Tcl, PL/Perl, PL/Python*, итд.) - дозвољава писање блокова команди у језицима који нису *SQL*;
- Индекси (изрази-индекси, парцијални индекси, итд.) - индексирање табела, индекси се могу креирати као резултати неког израза или функције (уместо само од вредности колоне), може се индексирати само део табеле, итд.;
- Тригери - догађаји који се покрећу приликом одређених акција - најчешће приликом уписивања или мењања података из табеле;
- Типови података - постојање разних типова података, на пример за чување ИП адреса, низова прменљивих дужина, итд.;

II.6 Остало

II.6.1 Развојно окружење – Еклипс (*Eclipse*)

Сам кôд је писан углавном у Еклипсу, (уз малу помоћ Ви едитора, за неке ситније и брзе интервенције), верзија за развој JEE апликација. Еклипс је веома моћно окружење, које се може надограђивати додацима којих има приличан број, а које развијају како појединци, тако и фирме. Неки од корисних додатака су нпр. *SVN* - додатак за верзирање софтвера, *HibernateTools* - помаже у раду са мапирањем датотека, могуће је из базе добити датотеке за мапирање, као и обратно, *Spring* додатак, итд.

II.6.2 Сервер - Апач Томкет (*Apache Tomcat*)

Сервер на којем је покретана и тестирана апликација је Апач Томкет. На тржишту постоји неколико верзија апликационих сервера за JEE, Томкет је у овом случају изабран због „сродности“ са већ навелико познатим „обичним“ Апач ХТТП сервером.

II.6.3 Оперативни систем – Линукс (*Linux*)

Оперативни систем на којем је апликација развијана је *Linux Slackware 13.1.*

III Апликација

Могућности неке технологије је најбоље показати на неком примеру, па је стога приступљено изради мини-апликације - сервлета - електронски школски дневник.

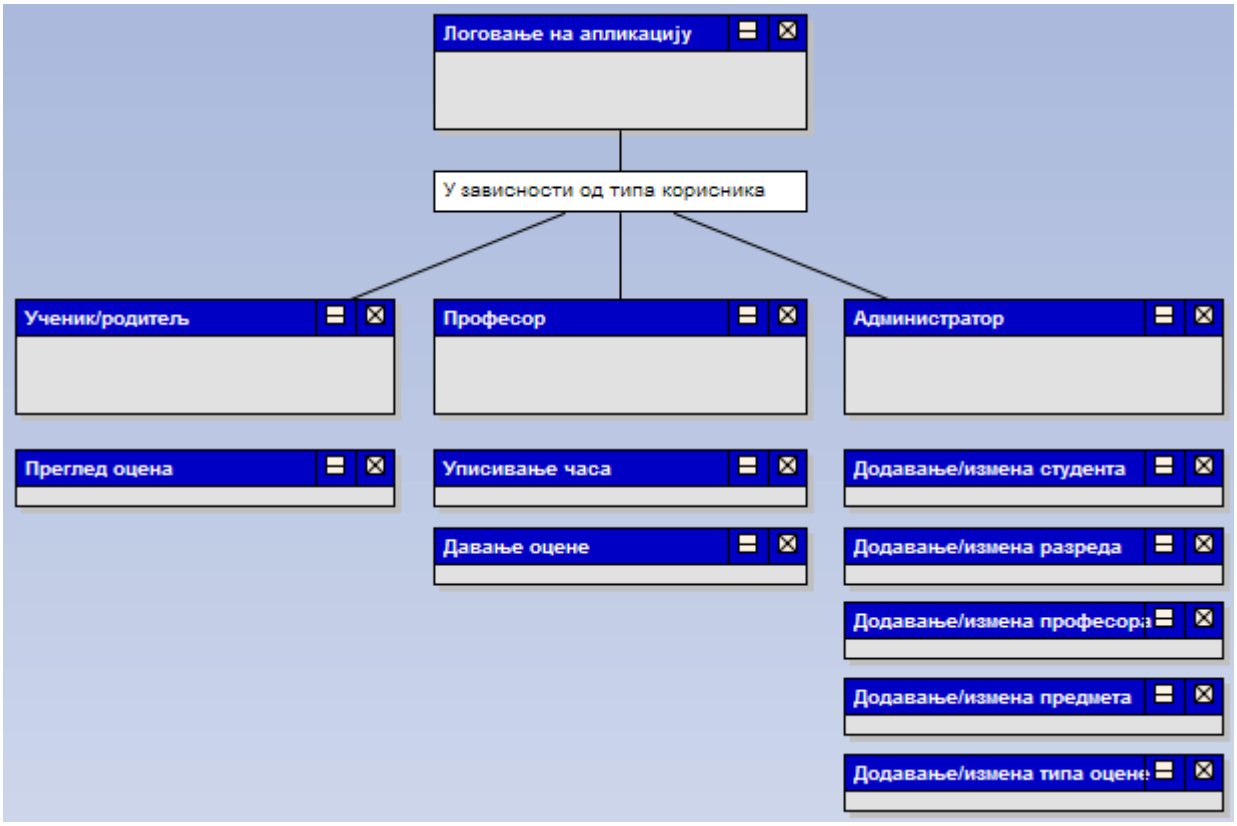
Класични дневници, какви се данас користе у школама, имају читав низ недостатака - ни ученици ни родитељи немају јасан увид у дневник (оцене, изостанке, итд.), практично свако може да да оцену било коме (на пример професор из српског језика може да да оцену из математике, ђак који на неки начин има приступ дневнику такође може да даје оцене, итд.), професори могу да мењају оцене, немају јасан увид зашто је нека оцена дата, на који начин је добијена (писмено, усмено, домаћи задатак, итд.), тешко је чувати архиву дневника и тако даље.

Сви поменути недостаци би били отклоњени коришћењем електронског дневника - и ученици и родитељи би имали стални приступ дневнику - у било које доба дана и ноћи, оцену може да да само професор, и то само за свој предмет, свака оцена носи своје податке, архива се без проблема чува у бази података, итд.

Приликом израде ове апликације, акценат је стављен на показивање могућности саме технологије, па су стога развијене само најосновније функционалности електронског дневника.

III.1 О апликацији

Апликација подржава три типа корисника - ученика, професора и администратора. Сваки корисник има своје корисничко име и шифру за логовање и може да приступи само оном делу апликације који је предвиђен за његов ниво приступа - тако, на пример, ученик или професор не могу да раде администрацију система.



Слика 1. - Странице доступне у апликацији

Када се улогује ученик (с тим што је замишљено да логовање на ученички налог користе и родитељи, ради контроле деце), он може да види своје оцене.

Када се улогује професор, има две опције на располагању - да упише час и да да оцену. У зависности од тога коју опцију изабере, отвара му се одговарајућа страница са одговарајућим формама, које треба да испуни подацима и сачува.

И, на крају, када се улогује администратор, он има највише опција да бира - чак пет. Он може да додаје/мења:

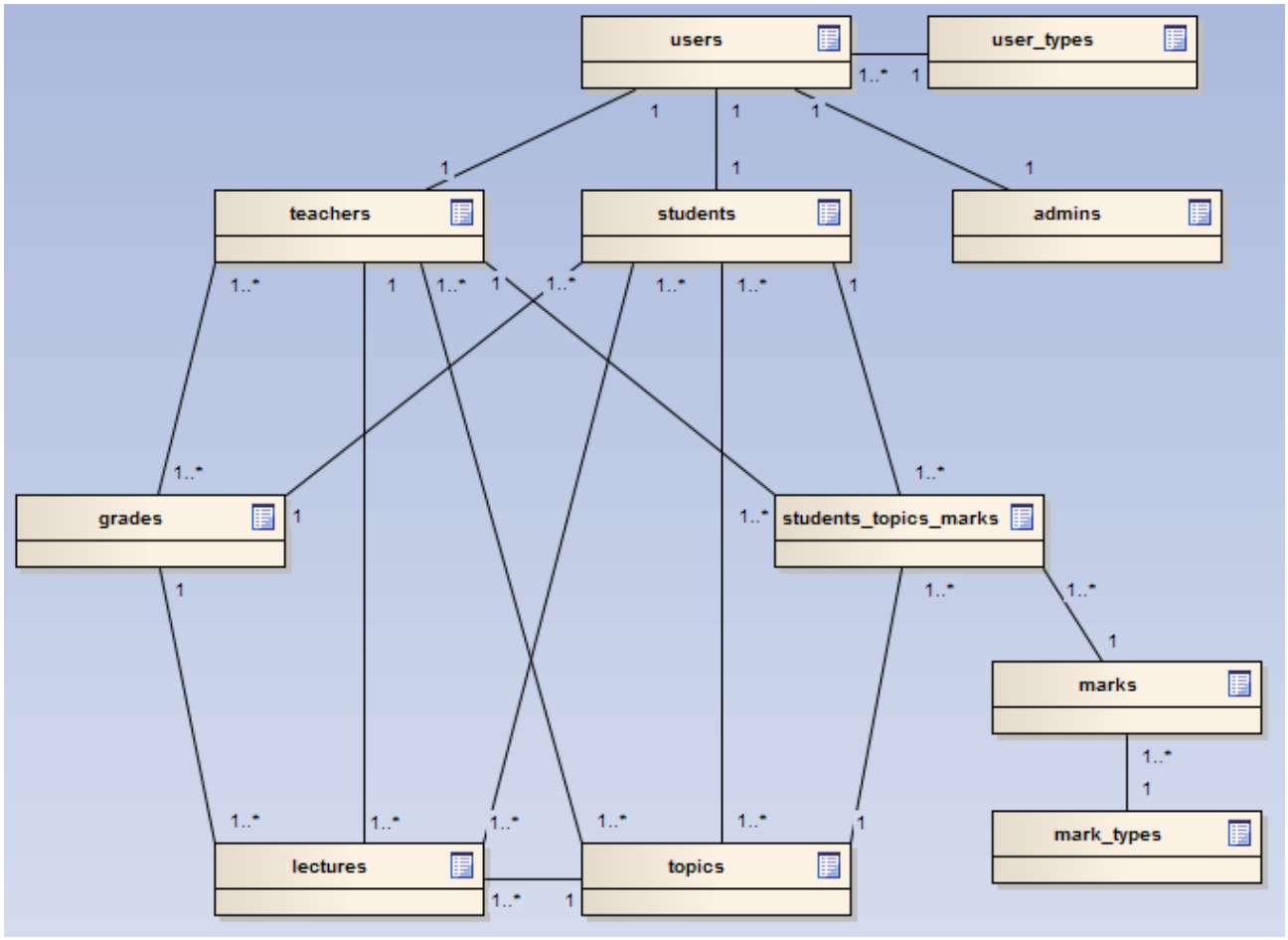
- професора;
- ђака;
- предмет;
- разред;
- тип оцене;

Избором неке од ових опција, отвара се одговарајућа форма за унос потребних података. Неке од тих форми су крајње једноставне - на пример, за додавање/мењање оцена, потребно је само унети име типа оцене. Док, са друге стране, за додавање/мењање професора, прво мора да му се додели корисничко име и шифра, са којима ће да се логује у апликацију, затим име и презиме, да се повеже са одељењима којима ће предавати, као и да се одреди које ће предмете предавати.

Апликација подржава вишејезичност - све поруке које се приказују кориснику се чувају у једној датотеци, тако да је за превод на неки други језик потребно превести само те поруке. Наравно, садржај који корисници уносе (професори, ђаци, имена предмета, итд.), не подлеже тим променама, пошто је он динамички.

III.2 База података

Сви подаци о професорима, ученицима, оценама, одељењима, итд. се чувају у бази података. На дијаграму је дат поједностављен приказ табела из базе података - везне табеле су избачене због читљивости. Оне постоје између свих табела које су у односу много-на-много (професори - одељења, професори - предмети, итд.), као и код неких веза један-на-много (професори - предавања, одељења - предавања, итд.).

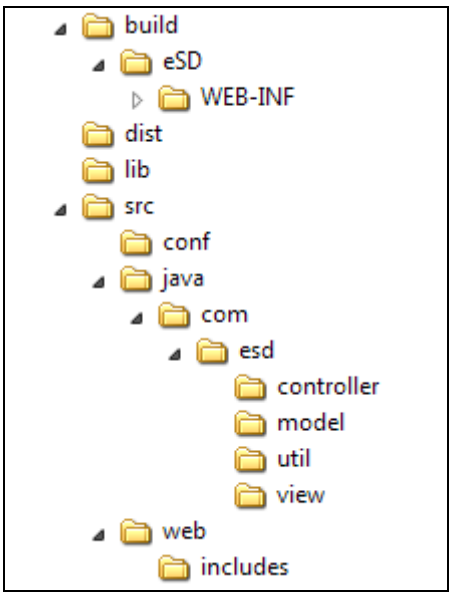


Слика 2. Упрошћени приказ базе података

III.3 Организација директоријума и датотека

Код прављења JEE (и не само JEE) апликација, јако је битно добро организовати хијерархију директоријума и датотека, односно пратити уобичајена и опште прихваћена правила. Сан је издао препоруку како би то требало да изгледа. Пре свега, ако се увек користе иста правила, олакшава се рад, пошто се избегава период навикавања на нову организацију, лакше се убацују нови програмери на пројекат, итд.

У изради ове апликације су поштована горе поменута правила. Следи приказ те организације.



Слика 3. - Приказ организације директоријума

У кореном директоријуму апликације, постоје четири директоријума:

- *build* - садржи искомпајлирану верзију апликације;
- *dist* - садржи .war датотеку - то је уствари цела апликација, спремна да се пусти на серверу и која садржи све што је потребно за њен рад;
- *lib* - садржи библиотеке потребне за рад апликације - нпр. библиотеке за Спринг, Хајбернејт, итд., а које се већ не налазе у *lib* директоријуму сервера;
- *src* - садржи изворни кôд и конфигурационе датотеке;

Поред ових директоријума, ту су још и две датотеке:

- *build.xml* - конфигурација за Ант;
- *local.properties* - генерална подешавања за апликацију;

У директоријуму src постоје три под-директоријума:

- *conf* - садржи конфигурационе датотеке;
- *java* - срж апликације, главни кôд се налази у под-директоријумима овог директоријума;
- *web* - садржи *index.jsp* као и *css* датотеке;

Следи детаљнији опис поменутих ставки.

III.3.1 *build* директоријум

Организација директоријума и датотека у овом директоријуму отприлике одговара организацији у фолдеру на самом серверу (након што се апликација распакује), из којег ће се апликација и сервирати корисницима.

III.3.2 *dist* директоријум

Као што је већ поменуто, овај директоријум садржи .war датотеку која у ствари представља саму апликацију, заједно са комплетним кодом, конфигурационим датотекама, библиотекама из lib директоријума, итд., једном речју - комплетну апликацију. Ова датотека

би требало да буде једина ствар коју је потребно пренети када је потребно пренети апликацију са једног сервера на други (не рачунајући базе података) и та апликација би требала да ради и на другом серверу, без проблема.

III.3.3 *lib* директоријум

Овде се стављају библиотеке потребне за рад саме апликације. У теорији, већина потребних библиотека ће већ бити укључена преко *lib* фолдера самог сервера, пошто их вероватно већ користе друге апликације на серверу. Али, ако апликација користи неку библиотеку која није много распрострањена, онда би свакако најбоља варијанта било убацити је у *lib* директоријум саме апликације. Треба још наппоменути да би са тим требало бити опрезан, пошто превише додатних библиотека може прилично увећати величину саме апликације (.*war* датотеке), што може бити проблематично приликом пребацивања, тестирања, итд.

III.3.4 build.xml

Као што је већ објашњено у уводном делу, ово је конфигурациона датотека за Ант.

III.3.5 local.properties

Овај директоријум служи да се у њега из *build.xml*-а издвоје ствари које ће се вероватно чешће мењати од остатка датотеке, али и које чине посебну целину.

У случају ове апликације, издвојен је директоријум у који се поставља .*war* датотека, односно локација са које ће сервер да покупи ту датотеку и ставити је у употребу:

```
deploy.dir=/M/java/apache-tomcat-6.0.26/webapps
```

Такође, издвојени су и параметри за приступ бази:

```
db.driver=org.postgresql.Driver
db.url=jdbc:postgresql://localhost/master
db.username=milan
db.password=milan
```

III.3.6 *src* директоријум

У оквиру *src* директоријума, постоје три под-директоријума:

- *conf;*
- *web;*
- *java;*

III.3.7 *conf* директоријум

Као што и само име каже, овде се чувају конфигурационе датотеке:

- *esd-servlet.xml* - ово је кључна датотека, она служи да повеже све написане класе, да убаци Спринг у апликацију, Хајбернејт, преводе, итд.;
- *messages.properties (messages_rs.properties)* - ове датотеке служе за превод. Оне су простог садржаја, са леве стране су предефинисане вредности, а са десне преводи за одговарајући језик. Нпр.:

```
professor=професор
```

а на страници се пише:

```
<fmt:message key="professor" />
```

Ако се у овом тагу напише реч за коју не постоји превод, исписаће се та реч, али окружена знаковима питања, како би било лако уочљиво где недостаје превод;

- *web.xml* - служи за подешавања самог сервлета, како ће да ради, које стране да прихвата, итд.;

На пример, следећи део кода:

```
<servlet-mapping>
   <servlet-name>esd</servlet-name>
   <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

дефинише да се сервлет зове `esd` и одређује да ће овај сервлет да узима у обзир само странице које се завршавају са `.html`.

Такође, могуће је дефинисати и заглавља која ће бити укључена у сваку страницу која се буде приказивала, што је јако згодно за ствари као што су банер или основни подаци на дну странице:

```
<include-prelude>
   /WEB-INF/jsp/aux/header.jsp
</include-prelude>
<include-coda>
   /WEB-INF/jsp/aux/footer.jsp
</include-coda>
```

III.3.8 *web* директоријум

Овај директоријум и његов садржај се копирају у базни директоријум апликације, када се она инсталира на сервер. У њему се налази прва страница апликације - *index.jsp*, као и стилови за сам сајт, у под-директоријуму *includes*.

III.3.9 *java* директоријум

У овом директоријуму се налази сама срж апликације - њен код. Директоријум *com/esd* садржи четири директоријума – *controller, model, util* и *view.* Постојање *controller, model u view* директоријума говори да је овде коришћена такозвана *MVC* архитектура. То је архитектура која је сачињена из три слоја – први је слој који чува податке – модел, други је слој који ради са њима – контролер и трећи слој је онај који их приказује – презентациони слој.

Кратак опис директоријума:

- *controller* - као што и само име каже - у овом директоријуму су смештени контролери - класе које покрећу акције. То је уствари пословни-ниво, где је смештена пословна логика апликације;
- *model* - овде се чувају мапирања базе података (у виду *XML* датотека), класе на које се табеле из базе мапирају, као и такозване класе-менаџери, које служе за добијање података из базе. На пример, за професора ће постојати *Teacher.java* - класа(зрно) на коју се мапира табела из базе, *Teacher.hbm.xml* - датотека која мапира табелу из базе на зрно и *TeacherManager.java* - класа која "дохвата" податке из базе;
- *util* - овде се налазе помоћне класе, нпр., класа за дефиницију корисничких тагова;
- *view* - у овом директоријуму се налазе датотеке које чине такозвани презентациони ниво апликације - *jsp* стране, кориснички тагови, итд.;

III.4 Организација директоријума и датотека на серверу

Када се апликација распакује и стартује на серверу, добија се слична организација директоријума и датотека. Садржај базног директоријума:

- *META-INF;*
- *WEB-INF;*
- *includes;*
- *index.jsp;*

META-INF директоријум није битан, он би требало да садржи податке о *jar* датотекама (*war* је посебна врста *jar* датотеке). Апликација би требало да настави да ради без проблема и ако би овај директоријум био обрисан.

У *WEB-INF* директоријум се смешта срж апликације. Овако изгледа његов садржај у случају *esd* апликације:

- *classes* - у њему се практично налази све што је потребно за рад апликације, осим презентационог нивоа - датотеке за мапирање, преводи, подешавања, а у његовом под-директоријуму *com/esd/* се налазе компајлиране класе (одговарајуће *.class* датотеке), у одговарајућим директоријумима (*controller, model, util*);
- *jsp* - ово је прекопиран *view* директоријум;
- *lib* - прекопирани *lib* директоријум;
- *src* - донекле прекопирани *src* директоријум - прекопиране су само класе из директоријума *controller*, *model*, *util;*
- *esd-servlet.xml* – прекопирано;
- *web.xml* – прекопирано;

includes і *index.jsp* су већ појашњени и они су само прекопирани.

Овде би требало поменути да је сав садржај *WEB-INF* директоријума недоступан кориснику који користи апликацију - он директно може да приступи само садржају који је ван њега. На пример, у случају *esd* апликације, ако корисник унесе адресу:

```
http://localhost:8080/esd/index.jsp
```

све ће радити нормално. Али, ако покуша са на пример:

```
http://localhost:8080/esd/WEB-INF/jsp/signin.jsp
```

добиће поруку о грешци. Датотеке у *WEB-INF* директоријуму су доступне само апликацији. Ово је урађено због сигурности. Препорука је да се све датотеке стављају у *WEB-INF* и да им се приступа само преко апликације, како би се спречило директно позивање страница од стране корисника, као и други видови злоупотреба.

IV Кôд апликације

Приказ кода ће пратити структуру *esd-servlet.xml* датотеке - за сваки њен део и оно што он дефинише биће приказивани делови кода из других датотека који су повезани са њим.

У овој датотеци се дефинишу разна Јава зрна – нпр. зрно за приступ бази, за трансакције, за контролере, итд.

Основни изглед дефиниције једног зрна:

```
<bean id="someId"  class="org.springframework.beans.factory.config.SomeClass">
    <property name="someName" value="someValue" />
</bean>
```

Из овог примера се види да је потребно дати име, идентификацију сваком зрну, и то име мора бити јединствено, пошто ће се оно користити за референцирање на ово зрно. Такође је потребно то зрно везати за неку класу, која одређује чему ће оно да служи, како и шта ће да ради, итд.

У телу зрна се дефинишу потребне ствари, у оквиру *property* елемента. У овом примеру се дефинише најосновнији атрибут и додељује му се вредност. Касније ће бити приказане и компликованије конструкције, где постоји листа или референца не неко друго зрно.

IV.1 Базе података

Прва три зрна која се дефинишу су везана за конфигурисање приступа бази података:

- *dataSource* зрно - дефинише параметре за приступ бази података, користи Спрингову класу *DriverManagerDataSource;*

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.postgresql.Driver" />
    <property name="url" value="jdbc:postgresql://localhost/master" />
    ...
</bean>
```

- *sessionFactory* - користи такође Спрингову класу *LocalSessionFactoryBean*, служи за прављење такозваних Хајбернејт сесија, преко којих се комуницира са базом и које ће бити обилато коришћене у апликацији. У оквиру њега се врши и подешавање Хајбернејта, навођење датотека за мапирање и слично, а користи и претходно зрно, користећи команду *ref="dataSource";*

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="useTransactionAwareDataSource" value="true" />
   <property name="mappingResources">
      <list>
         <value>User.hbm.xml</value>
         ...
      </list>
   </property>
   <property name="hibernateProperties">
      <props>
         <prop key="hibernate.dialect">
            org.hibernate.dialect.PostgreSQLDialect
         </prop>
         ...
      </props>
   </property>
</bean>
```

- *transactionManager* - служи за дефинисање трансакција, а користи Спрингову класу *HibernateTransactionManager*. Референцира се на *sessionFactory*, што значи да укључује и *dataSource*, тј. садржи у себи све што се тиче комуникације са базом у апликацији;

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

IV.2 URL мапирање

Следећа ствар коју је потребно урадити је мапирати *URL*-ове, тј. "спровести" корисникове захтеве на праву адресу, тј., на одговарајући контролер. Овај посао обављају два зрна:

- `urlMap` - служи за захтеве када се приступа страницама за које није потребна ауторизација;

- `urlMapAuthenticate` - служи за захтеве када се приступа страницама за које је потребна ауторизација;

Што се првог случаја тиче, ситуација је крајње једноставна - једине две стране којима је могуће приступити су страна за логовање на систем и страна која се приказује када се корисник излогује.

Када је потребна ауторизација, користи се следеће зрно:

```
<bean id="urlMapAuthenticate"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
            <list>
                    <ref bean="loginInterceptor" />
                    <ref bean="localeChangeInterceptor"/>
            </list>
    </property>
    <property name="urlMap">
            <props>
                    <prop key="/topic.html">topicListController</prop>
                    <prop key="/start.html">startingController</prop>
                    <prop key="/teacherAction.html">teacherActionController</prop>
                    <prop key="/studentAction.html">studentActionController</prop>
                    <prop key="/adminPage.html">adminActionController</prop>
            </props>
    </property>
</bean>
```

Очигледно, ово зрно користи Спрингову класу `SimpleUrlHandlerMapping`.

Први атрибут је ствар која се први пут појављује - *interceptors*. *Interceptors*, или пресретачи, служе да, као што и само име каже, “пресретну” захтев пре него што се испоручи одговарајућем контролеру и уради нешто са њим, нпр. да га измене, па проследе даље или, као у овом случају, изврше неке провере и, у зависности од резултата, проследе га даље или одбију.

У овом зрну имамо дефинисана два пресретача - први врши проверу корисника и предузима акције у складу са тим, док је други задужен за преводе.

Као што се из примера види, он се референцира на зрно `loginInterceptor`, чија дефиниција следи:

```
<bean id="loginInterceptor" class="com.esd.controller.HttpRequestInterceptor">
    <property name="signInPage">
            <value>signin.html</value>
    </property>
    <property name="applicationSecurityManager">
            <ref bean="applicationSecurityManager" />
    </property>
</bean>
```

Ово је прво зрно које се ослања на класу развијену за посматрану апликацију - `HttpRequestInterceptor`. Из кода те класе се може видети да је она изведена из Спрингове класе `HandlerInterceptorAdapter`. Такође, атрибути који су дефинисани за ово зрно постоје и у класи, као њени атрибути. За њих је обавезно направити функције, такозване гетере и сетере - функције које постављају и враћају њихову вредност, како би Спринг могао да ради са њима.

Како би захтев био преконтролисан према потреби апликације, преклопљена је функција `preHandle`:

```
public boolean preHandle(
        HttpServletRequest request,
        HttpServletResponse response,
        Object handler) throws Exception
{
   User user = (User) applicationSecurityManager
              .getUser(request);

   if (user == null)
   {
        response.sendRedirect(this.signInPage);
        return false;
   }

   return true;
}
```

која врши једноставну проверу - ако је корисник дефинисан (тј. улогован), враћа се *true*, тј. захтев се прослеђује даље. Ако није, корисник се преусмерава на страну за логовање.

Овде се користи и зрно `ApplicationSecurityManager` - то је једноставна класа која чува тренутног корисника у захтеву и по потреби проверава да ли постоји корисник у истом, тј. да ли је неко улогован.

Ако корисник прође аутентикацију или је већ улогован, пресретач ће дозволити да се захтев проследи на обраду. Тада на снагу ступа други део дефиниције зрна *urlMapAuthenticate* - прослеђивање захтева одговарајућем контролеру. Обзиром да је у *web.xml* датотеци подешавање такво да се очекују адресе захтева које се завршавају са *.html* (не рачунајући додатне параметре), овде се управо хватају такве адресе - па ће тако захтев за страницом која се завршава на */teahcerAction.html* да се проследи контролеру који ради са професорима - `teacherActionController`. Овај ред заправо значи да ће контролу у том случају преузети зрно под задатим именом, које, наравно, негде мора бити дефинисано.

У *urlMap* делу је дефинисан још један пресретач - пресретач за превођење - `localeChangeInterceptor` који служи за локализацију апликације. Он у захтеву тражи параметар `locale` и у зависности од његове вредности учитава одговарајући превод. Подразумевана датотека из које се читају преводи је `messages.properties`. У посматраној апликацији постоји и превод на српски, када ће у захтеву параметар `locale` бити једнак `rs` (`locale=rs`) и у том случају ће се преводи узимати из датотеке `messages_rs.properties`. Наравно, и за остале језике би важила иста правила (`locale=fr`, `messages_fr.properties`, `<value>messages_fr</value>`).

Имена датотека из којих се узимају преводи су дефинисана у зрну `messageSource`, које користи Спрингову класу `ResourceBundleMessageSource`:

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
            <list>
                    <value>messages</value>
                    <value>messages_rs</value>
            </list>
    </property>
</bean>
```

IV.3 Зрна апликације

Следи мало детаљнији преглед зрна која одређују понашање саме апликације.

IV.3.1 SignInController

Као што и само име каже, овај контролер служи за контролу и извођење логовања у апликацију.

Слика 4. - Прозор за логовање

Код:

```
<bean name="signInController" class="com.esd.controller.SignInController">
    <property name="sessionForm">
            <value>true</value>
    </property>
    <property name="formView">
            <value>/signin</value>
    </property>
    <property name="successView">
            <value>redirect:start.html</value>
    </property>
    <property name="commandClass">
            <value>com.esd.model.User</value>
    </property>
    <property name="validator">
            <ref bean="signinValidator" />
    </property>
    <property name="userManager">
            <ref bean="userManagerProxy" />
    </property>
    <property name="applicationSecurityManager">
            <ref bean="applicationSecurityManager" />
    </property>
</bean>
```

Први атрибут - `sessionForm` одређује да ли ће се објекат форме чувати или ће се сваки пут (приликом сваког отварања форме) правити нови. Ако је постављен на `true`, онда ће контролор увек прво покушати да нађе већ постојећи објекат повезан са актуелном сесијом. Тек ако то не успе, покушаће да направи нови објекат форме.

Следећа два атрибута - `formView` и `successView` - садрже податке о томе које ће се странице приказати, у зависности од резултата обраде захтева - као што само име каже - `formView` је приказ саме форме, а `successView` је оно што треба да се прикаже када је форма успешно послата.

Пример кода:

```
if(applicationSecurityManager.getUser(request) != null)
   return new ModelAndView(getSuccessView());
```

- ако је корисник већ дефинисан, то значи да је неко већ улогован, па је само потребно проследити га на страну која се приказује приликом успешног логовања.

Преко атрибута `commandClass` се ХТМЛ форма повезује директно са неком класом апликације - у овом случају са User класом. Тиме је Спрингу назначено да ће класу `User` попуњавати директно подацима из захтева.

Пример кода:

```
public void onBindAndValidate(     HttpServletRequest request,
                                    Object command,
                                    BindException errors) throws
Exception
{
   User formUser = (User) command;
   User dbUser = (User) command;

   if((dbUser = userManager.getUser(formUser.getUserId())) == null)
         errors.reject("error.login.invalid");
   else
         applicationSecurityManager.setUser(request, dbUser);
}
```

- `onBindAndValidate` је функција која се позива након повезивања и валидације и омогућава да се уради још неке специфичне ствари пре одговора на захтев. У овом примеру се на почетку види како се пуни командна класа (у овом случају то је `User`) из параметра `command`.

`signinValidator` атрибут дефинише валидатора за захтев - у овом примеру, врши се валидација корисника који покушава да се улогује у апликацију. Следи пример кода функције `validate` која чини већину класе `SignInValidator`:

```
public void validate(Object command, Errors errors)
{
   User user = (User) command;
   if(user == null)
         return;

   String userName = user.getUserName();
   String password = user.getPassword();

   if(userName.length() < 1)
         errors.reject("error.login.invalidUsername");
   else
         if(password == null || password.trim().length() < 1 ||
password.trim().length() > 10)
               errors.reject("error.login.invalidPassword");
}
```

Као што се може видети, овде се врши основна валидација - да ли је унето корисничко име, шифра, да ли је дужина одговарајућа, и слично, по потреби.

Следећи атрибут је `userManager`. Он служи за комуникацију са базом, а везано за `User` класу, па тако у примеру пре претходног, где је приказ `onBindAndValidate` функције, инстанца корисника добијена из *command* аргумента се пореди са инстанцом добијеном из базе помоћу `userManager`-а, и у зависности од тога се предузимају даљи кораци (приказ поруке о грешци или памћење улогованог корисника у захтев).

Класа `userManager` је дефинисана посебно, у директоријуму *model*, заједно са датотекама за мапирање и осталим *Manager* класама - `teacherManager`, `studentManager`, итд.

Заједничко за све *Manager* класе је да све наслеђују класу `HibernateDaoSupport`.

Пре наставка приче о добијању података из базе, следи кратак опис комуникације посматране апликације са базом.

Позната је чињеница да је комуникацију са базом, тј. добијање података из базе пожељно радити преко трансакција, зато што је увек могуће без проблема вратити податке на почетно стање у случају грешке.

Као што је већ поменуто у уводу, Спринг омогућава да се сва комуникација са базом уоквири трансакцијама и тиме знатно олакша рад програмерима - они више не морају да брину да ли су започели трансакцију, да ли су је прекинули, да ли су је сачували, и слично - они сада треба да брину само о логици саме апликације, остало Спринг ради за њих.

Наравно, све то је потребно дефинисати и подесити:

```
<bean id="baseProxy"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
abstract="true">
   <property name="transactionManager" ref="transactionManager" />
   <property name="proxyTargetClass" value="true"/>
   <property name="transactionAttributes">
      <props>
         <prop key="save*">PROPAGATION_REQUIRED</prop>
         <prop key="delete*">PROPAGATION_REQUIRED</prop>
         <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
   </property>
</bean>

<bean id="userManagerProxy" parent="baseProxy">
   <property name="target" ref="userManager" />
</bean>

<bean id="topicManagerProxy" parent="baseProxy">
   <property name="target" ref="topicManager" />
</bean>
```

Прво је дефинисано основно зрно - `baseProxy`, које користи Спрингову класу `TransactionProxyFactoryBean` и које је апстрактно - то значи да оно може да служи као шаблон - што и јесте његова сврха у апликацији - оно служи као шаблон за остала зрна и у њега су издвојени атрибути који су исти за сва зрна која се из њега изводе, а то су уствари подешавања самих трансакција (како не би свако зрно морало да буде писано из почетка).

Могуће је бирати понашање трансакција приликом чувања, брисања или добијања података из базе - у случају посматране апликације, PROPAGATION_REQUIRED значи да ће сваки нови упит ка бази прво проверити да ли већ постоји отворена трансакција, и ако постоји, упит ће ићи преко ње, а ако не постоји, тек онда се прави нова трансакција. `readOnly` опција у `get` атрибуту је постављена како би убрзала процес, пошто се, као што и само име каже, у оквиру те дефиниције подаци само добијају из базе, па није потребно да постоји могућност писања.

Овде би требало поменути и да је могуће подесити и ниво изолације сваке трансакције.

Након што је дефинисано базно зрно, тј. шаблон, зрна за конкретну употребу се једноставно референцирају на шаблон преко опције `parent="baseProxy"`, чиме су она наследила особине базног зрна. Наравно, одговарајуће зрно се преко атрибута повезује на одговарајући менаџер.

Овим је укратко описана комуникација са базом и подешавање те комуникације.

Као што је већ речено, UserManager класа служи да се из базе података доставе подаци о датом кориснику.

Ево како то изгледа на делу:

```
public User getUser(String userName)
{
     User user = null;
     user =
(User)getSessionFactory().getCurrentSession().createQuery("from User
where userName= ?").setString(0, userName).uniqueResult();

     return user;
}
```

Овде је у упиту ка бази коришћен *HQL* - Хајбернејтов *SQL* језик. Он је јако сличан обичном *SQL*, али је његова велика предност (а уједно и једна од највећих предности самог Хајбернејта) - преносивост. Ако су сви упити написани коришћењем *HQL-a*, ти упити ће радити на свим базама (које Хајбернејт подржава) - само је потребно променити дијалект у подешавањима.

Треба приметити да се овде не ради директно са табелама у бази, већ са зрнима која су дефинисана у апликацији - у овом случају - `User` и која су повезана са табелама у бази података - у овом случају - `users` табела, а детаљи тог повезивања су дати у *User.hbm.xml* датотеци:

```
<hibernate-mapping>
   <class name="com.esd.model.User" table="users">
          <id name="userId" column="userid">
                 <generator class="sequence">
                        <param name="sequence">users_userid_seq</param>
                 </generator>
          </id>

     <property name="userName" />
     <property name="password" />
          <property name="userTypeId" />
   </class>
</hibernate-mapping>
```

Поље `userid` у бази је јединствено, идентификационо поље, и зато је у овој конфигурацији потребно рећи Хајбернејту да би требало вредности за нове уносе у табелу да тражи у секвенци `users_userid_seq` у бази података. Остала поља се просто наводе, с тим што је битно да се зову исто као и атрибути зрна на које се табела мапира. То зрно изгледа отприлике овако:

```
public class User implements Serializable
{
        private int userId;
        private int userTypeId;
        private String userName;
        private String password;

        public int getUserId();
        public void setUserId(int userId);
        public int getUserTypeId();
        public void setUserTypeId(int userTypeId);
        public String getUserName();
        public void setUserName(String userName);
        public String getPassword();
        public void setPassword(String password);

        ...
}
```

Као што се види, потребне су и функције које постављају и враћају вредности атрибута.

Што се самог упита тиче, треба рећи још пар ствари - са `getSessionFactory().getCurrentSession()` се узима активна (или отвара нова, ако активна не постоји) конекција ка бази (ту се користе зрна за повезивање на базу која су већ описана). Затим следи и сам упит, који је тривијалан. Постоји и верзија `getUser` функције која за параметар прима `ID` корисника и која ради на исти начин. Она је написана због удобности.

И, на крају, треба поменути и *jsp* страницу која је "везана" на овај контролер.

```
<fmt:message key="basic.username" />
```

Ово је кључна ствар у *jsp* странама, а везано за преводе - овај израз се аутоматски преводи у стринг који одговара кључу `basic.username` у одговарајућој датотеци са преводима.

```
<%@ include    file="/WEB-INF/jsp/aux/includemessages.jsp"%>
```

Ова линија укључује додатну *jsp* страну - *includemessages.jsp*, која служи за исписивање порука о грешци, али и обавештења кориснику (за ово друго су касније у апликацији коришћена још два механизма - "физички" метод и кориснички дефинисани тагови).

includemessages.jsp:

```
<spring:bind path="command.*">
  <c:if test="${not empty status.errorMessages}">
    <c:forEach var="error" items="${status.errorMessages}">
      <font color="red"><c:out value="${error}" escapeXml="false" />
      </font>
      <br />
    </c:forEach>
  </c:if>
</spring:bind>

<!-- status messages -->
<c:if
  test="${not empty message}">
  <font color="green"><c:out value="${message}" /></font>
  <c:set var="message" value="" scope="session" />
</c:if>
```

Као што се може видети, у првом делу се проверава да ли постоје грешке и, ако постоје, приказује се порука о грешци. Слично важи и за обавештење.

Повратак на *jsp* страну за логовање:

```
<spring:bind path="command.userName">
    <input name='userName' />
</spring:bind>
```

И овде и у претходном примеру се по први пут среће `spring:bind` таг. Он служи да се повеже *jsp* страну са контролером иза, тј. са командном класом. Када се одговарајућа поља у форми повежу са одговарајућим атрибутима класе, Спринг преузима потпуну контролу - аутоматски попуњава зрна, форме на страници, итд., чиме знатно олакшава рад и омогућава да се програмер више посвети самој пословној-логици апликације.

IV.3.2 StartingController

Као што је већ приказано у претходном примеру, `successView` у зрну за логовање је постављен на: “`redirect:start.html`”, па, када се погледа на шта је тај *URL* мапиран, лако је доћи до закључка да се након логовања активира зрно `startingController`.

Ово зрно има улогу скретничара - приказује страницу у зависности од тога који тип корисника се управо улоговао - ђак, професор или администратор.

```
<bean name="startingController" class="com.esd.controller.StartingController">
   <property name="teacherManager">
      <ref bean="teacherManagerProxy" />
   </property>
   <property name="studentManager">
      <ref bean="studentManagerProxy" />
   </property>
   <property name="adminManager">
      <ref bean="adminManagerProxy" />
   </property>
   <property name="applicationSecurityManager">
      <ref bean="applicationSecurityManager" />
   </property>
   <property name="teacherView">
      <value>teacher</value>
   </property>
   <property name="studentView">
      <value>student</value>
   </property>
   <property name="adminView">
      <value>admin</value>
   </property>
   <property name="successView">
      <value>teacher</value>
   </property>
</bean>
```

Укључују се и потребна менаџер-зрна, како би било могуће коришћење одговарајућих података.

Одабир стране се врши у функцији `handleRequest`, у зависности од типа корисника:

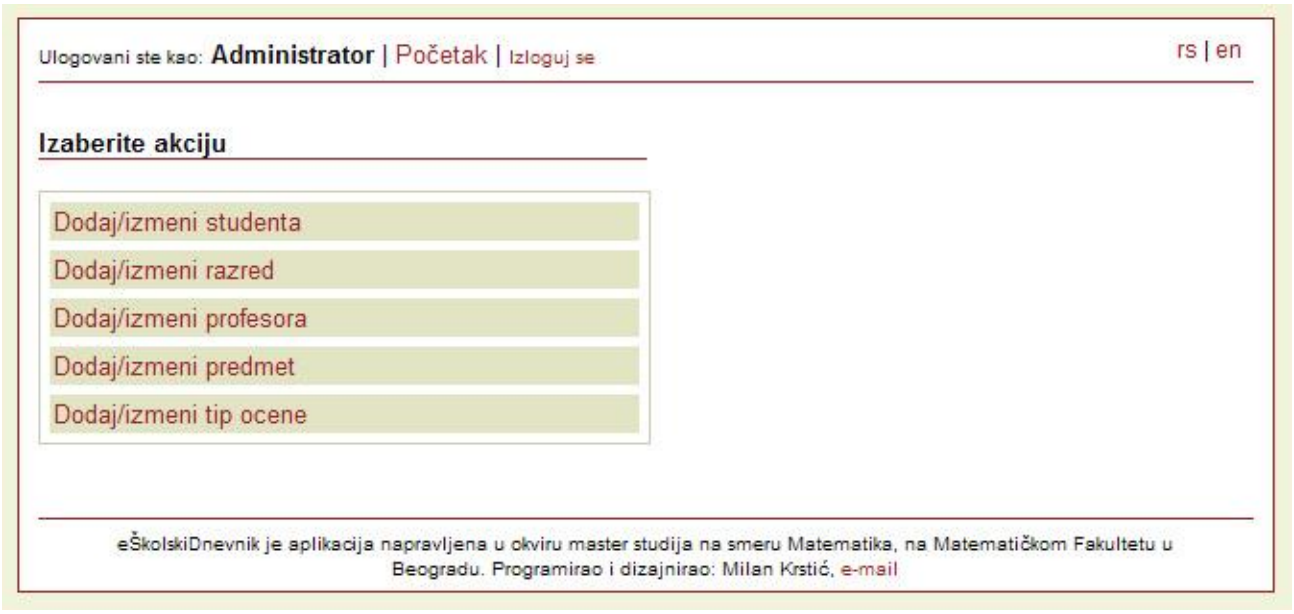
```
User user = (User) applicationSecurityManager.getUser(request);

if(user.getUserTypeId() == TEACHER) {
     Teacher teacher = teacherManager.getTeacher(user.getUserId());
     return new ModelAndView(getTeacherView(), "teacher", teacher);
}
else if(user.getUserTypeId() == STUDENT) {
     Student student = studentManager.getStudent(user.getUserId());
     return new ModelAndView(getStudentView(), "student", student);
}
else if(user.getUserTypeId() == ADMIN) {
     Admin admin = adminManager.getAdmin(user.getUserId());
     return new ModelAndView(getAdminView(), "admin", admin);
}
```

Примери страна:

Слика 5. - Почетна страна за професора

Слика 6. - Почетна страна за ђаке



Слика 7. - Почетна страна за администратора

Следи део кода из `StudentManager`-а, пошто се ђак из базе добија на другачији начин од оног који је коришћен у `UserManager`-у у приликом логовања:

```
public Student getStudent(int studentId)
{
     Student student = null;

     student = (Student)getHibernateTemplate().get(Student.class,
studentId);
     student.getMarks().size();
     student.getTopics().size();

     return student;
}
```

Као што се може видети, овде се користи функцију `get`, којој се прослеђује `ID` ђака чији подаци се траже.

Оно што је занимљиво у овој функцији су наредне две линије, у којима се позива `size()` функција над колекцијама за оцене и предмете. Хајбернејт ове колекције попуњава примарним кључевима одговарајућих редова, али потпуно попуњавање се дешава тек када неки кôд из апликације покуша да приступи неком елементу ових колекција. Ту могу настати два проблема - први је да је сесија можда истекла, па више не постоји конекција на базу, што ће довести до грешке у апликацији. Други је такозвани "*n+1*" проблем - како петља иде кроз колекцију, Хајбернејт попуњава један по један објекат, па тако за сваког члана колекције ради по један упит ка бази, што је јако споро и прилично заузима ресурсе.

Зато се користи `size()` функција, пошто она практично натера Хајбернејт да попуни колекције.

Наравно, да би Хајбернејт знао како да повеже нпр. ђака и предмете, те везе морају бити представљене у конфигурационој датотеци - у овом случају - *Student.hbm.xml:*

```
<hibernate-mapping>
   <class name="com.esd.model.Student" table="students">
            ...
            <set name="topics" table="students_topics" order-by="topicId">
                     <key column="studentId"/>
                     <many-to-many column="topicId" class="com.esd.model.Topic"/>
            </set>
            ...
   </class>
</hibernate-mapping>
```

Student.java:

```
public class Student {
   ...
   private Set<Topic> topics;
   ...
   public Set<Topic> getTopics();
   public void setTopics(Set<Topic> topics);
   ...
}
```

Ово је само део који се односи на предмете. У конфигурационој датотеци је наглашено да ће контејнер у који ће се смештати подаци бити типа `set`, да ће се та променљива звати `topics`, да је табела која повезује тренутну табелу (`students`) са циљном (која је представљена као зрно, са пуном путањом - `com.esd.model.Topic`) `student_topics`, да се подаци сортирају по `topicId`-ју, као и по којим колонама да се врши повезивање - `studentId` и `topicId`. И то је то.

Дакле, страна која ће се приказати након успешног логовања зависи од типа корисника који се улоговао - професор, ђак или администратор, тј. приказаће се *teacher.jsp*, *student.jsp* или *admin.jsp*. Опције које ће сваки од тих типова корисника имати пред собом су већ описане на почетку.

Следи део кода из *teacher.jsp*:

```
<form:form commandName="teacher" action="teacherAction.html?action=lectureSignin">
   <fmt:message key="story.lectureSignIn" />:
   <form:select path="topicId" multiple="false">
      <form:option  value="0" label="---" selected="selected" />
      <form:options items="${teacher.topics}" itemLabel="topicName" itemValue="topicId" />
   </form:select>
   <form:select path="grade" multiple="false">
      <form:option  value="0" label="---" selected="selected"/>
      <form:options items="${teacher.grades}" itemLabel="gradeName" itemValue="gradeId" />
   </form:select>
   <input type="submit" value="<fmt:message key="story.submit" />" />
</form:form>
```

Овде се први пут користи Спрингов *form* таг. Он олакшава прављење форми и њихово повезивање са зрнима.

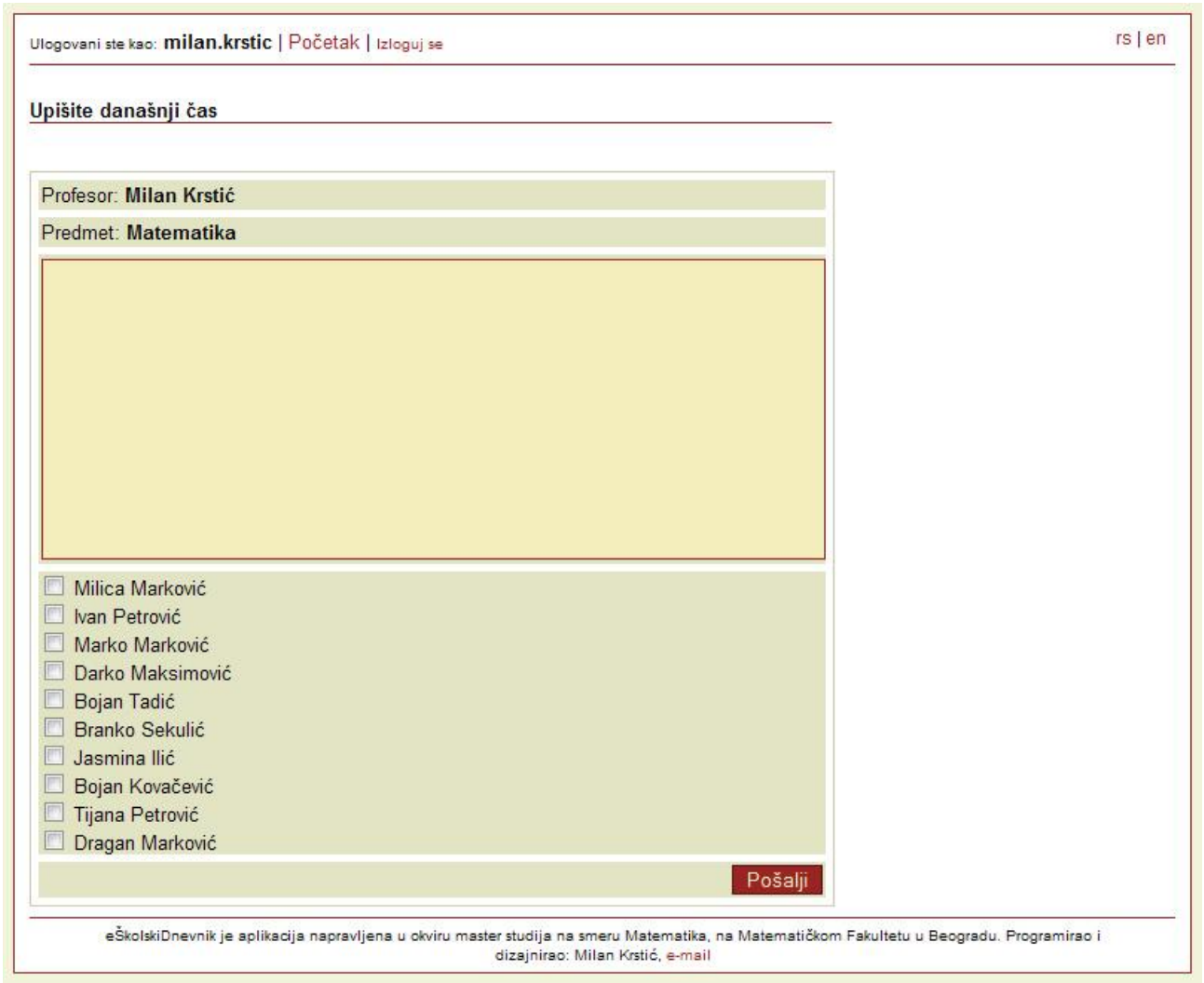
Одмах на почетку следи још нешто ново - `commandName="teacher"` - овом командом се Спрингу ставља до знања да би подаци са ове форме требало да се сместе у зрно `teacher`. Самим тим, сва улазна поља која се користе у овој форми би требало да постоје и у зрну за професора.

`<form:select>` таг скраћује посао - није потребно писати петље, штампати податке ручно и слично - све што је потребно је `<form:options>` тагу проследити одакле да "вуче" податке (опцијом `items="${teacher.grades}"`), остало ће он сам урадити.

IV.3.3 TeacherActionController

Део из конфигурационе датотеке који је у вези са овим контролером је веома сличан претходном примеру, тако да овде неће бити приказан.

Као што је већ речено на почетку, професор може да ради две ствари - да упише час и да да оцену. У оба случаја, професор на *teacher.jsp* страни мора да изабере на који предмет (од оних које он предаје) и на које одељење (од оних којима он предаје) се то односи. Након тога, приказује му се следећа страна:



Слика 8. - Упис часа

У поље за текст се уписује оно што је планирано да се ради на том часу, а одсутни ученици се бележе тако што се штиклира поље поред њиховог имена.

Следи кôд из *jsp* стране:

```
<c:if test="${! empty param.saveLecture}">
  <br />
  <div class="notification">
     <fmt:message key="story.lectureSignedIn" />
  </div>
  <br />
  <br />
</c:if>
```

Ово је “физички” метод приказивања нотификација који је поменут раније - једноставно, тагом `c:if` се испитује да ли је параметар `saveLecture` дефинисан и, ако јесте, то значи да је професор управо уписао час, па је потребно и написати одговарајуће обавештење о томе.

У самом контролеру, параметром `action` контролеру се каже шта треба да ради - да ли да прикаже страницу за упис часа или давање оцене:

```
String action = request.getParameter("action");
if(action.equals(getGiveMarkView()))
   return new ModelAndView(getGiveMarkView(), "teacher", teacher);
else if(action.equals(getLectureSigninView()))
   return new ModelAndView(getLectureSigninView(), "lecture", lecture);
```

Подаци о неком часу се чувају у табели `lectures`, па самим тим постоји и зрно `Lecture`, који је потребно формирати и напунити подацима пре него што се сачува у бази:

```
Lecture lecture = new Lecture();
lecture.setGrade(grade);
lecture.setTopic(topic);
lecture.setTeacher(teacher);
lecture.setStudents(students);
lecture.setDate(new Date());
```

Наравно, сви ови подаци се добијају из корисничког уноса или из базе, а на основу корисничког уноса.

Након тога, следи чување података:

```
lectureManager.saveLecture(lecture);
```

Наравно, кôд из `lectureManager`-а:

```
public void saveLecture(Lecture lecture)
{
    getHibernateTemplate().saveOrUpdate(lecture);
}
```

Крајње једноставно!

Као што је већ поменуто, о повезивању података из бази се брине Хајбернејт, али тек након што су подаци исправно мапирани. Ево и дела конфигурационе датотеке *Lecture.hbm.xml:*

```
<set name="students" table="lectures_students" order-by="lectureId"
lazy="false">
    <key column="lectureId"/>
    <many-to-many column="studentId" class="com.esd.model.Student"/>
</set>

<property name="description" column="description" />
<property name="date" column="date" />

<join table="lectures_teachers" optional="true">
    <key column="lectureId"/>
    <many-to-one name="teacher" column="teacherid" not-null="true"/>
</join>
<join table="lectures_topics" optional="true">
    <key column="lectureId"/>
    <many-to-one name="topic" column="topicid" not-null="true"/>
</join>
<join table="lectures_grades" optional="true">
    <key column="lectureId"/>
    <many-to-one name="grade" column="gradeid" not-null="true"/>
</join>
```

И, наравно, *Lecture.java*:

```
public class Lecture implements Serializable
{
     private Set<Student> students;
     private Teacher teacher;
     private Topic topic;
     private Grade grade;

     ...

     public Set<Student> getStudents();
     public void setStudents(Set<Student> students);
     public Teacher getTeacher();
     public void setTeacher(Teacher teacher);
     public Topic getTopic();
     public void setTopic(Topic topic);
     public Grade getGrade();
     public void setGrade(Grade grade);
}
```

Као што се види, овде се опет користи `set` као контејнер за ђаке, пошто на један час може да се веже више ђака. Али, са друге стране, за један час може да се веже само по један професор, предмет и одељење, па тако постоји само по једна променљива тог типа, која је опет повезана преко *join* табеле у бази - `lectures_teachers`, `lectures_topics` или `lectures_grades`.

`saveOrUpdate` функција прави нови унос у бази ако већ не постоји или освежава већ постојећи.

На сличан начин се врши припрема података за давање оцене, али се само давање оцене одвија на мало другачији начин. У контролеру за професора се позива функција из `teacherManager-a`:

```
teacherManager.giveMark(Integer.parseInt(request.getParameter("studentId"
)), Integer.parseInt(request.getParameter("topicId")),
Integer.parseInt(request.getParameter("enteredMark")),
Integer.parseInt(request.getParameter("marktypeId")),
teacher.getTeacherId());
```

која изгледа овако:

```
public void giveMark(int studentId, int topicId, int enteredMark, int
marktypeId, int teacherId)
{
     Mark mark = new Mark();
     mark.setMarkType(marktypeId);
     mark.setValue(enteredMark);

     getHibernateTemplate().saveOrUpdate(mark);

     StudentTopicMark stm = new StudentTopicMark();
     stm.setStudentId(studentId);
     stm.setMarkId(mark.getMarkId());
     stm.setTopicId(topicId);
     stm.setTeacherId(teacherId);

     getHibernateTemplate().saveOrUpdate(stm);
}
```

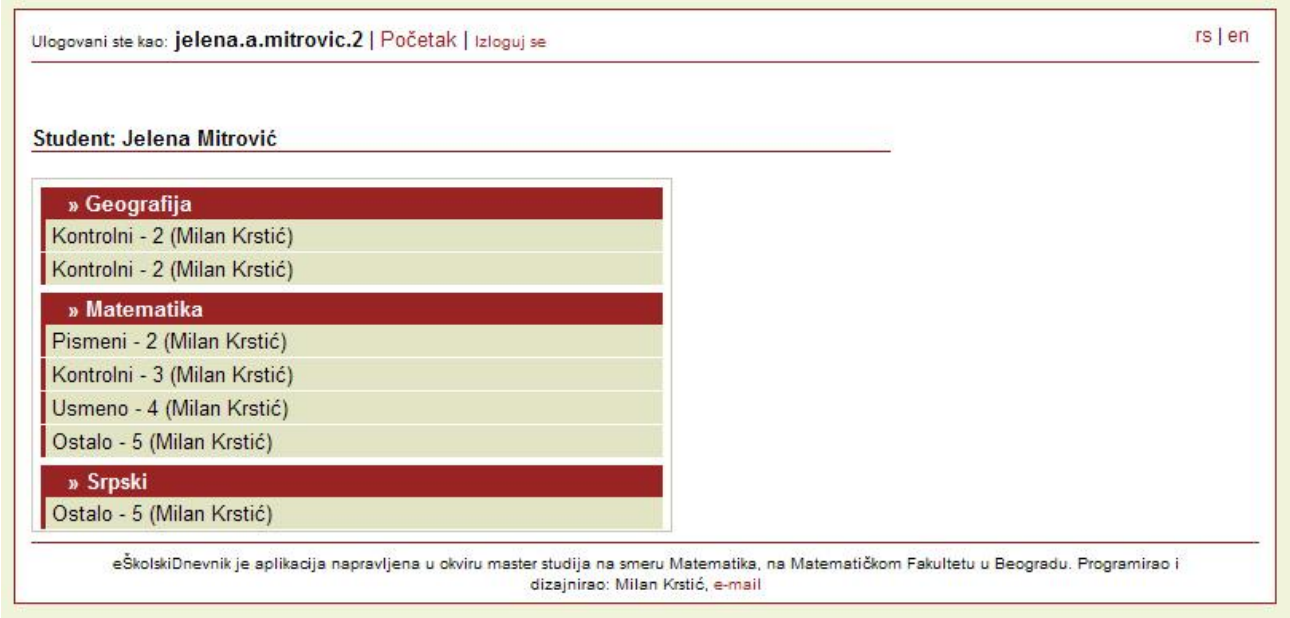
У првом делу се прави нова оцена и, пошто је то новонастали податак, чува се у бази. У другом делу је потребно ту оцену повезати са одговарајућим ђаком, професором и предметом, тако да се сви ти подаци заједно уписују у табелу намењену за њихово спајање. Наравно, као што је већ поменуто, уписивање података у одговарајуће табеле у бази обавља Хајбернејт, а у зависности од тога како су мапирани.

Овде још треба поменути други начин за приказ менија за одабир. У претходном примеру су коришћени Спрингови тагови `form`, `form:select` и `form:options`, док се овде користи петљу `c:forEach` која иде кроз контејнер са подацима и исписује их све редом.

```
<select name="studentId">
   <option value="0">---</option>
   <c:forEach items="${teacher.grade.students}" var="student">
      <option value="${student.studentId}">${student.lastName}
${student.firstName}</option>
   </c:forEach>
</select>
```

IV.3.4 StudentActionController

Овај контролер је по питању одабира стране, добијања података из базе и приказа исти као и претходни, зато нећемо бити примера кода из тог дела. Оно што не постоји у другим контролерима је сређивање података пре приказа. Зрно `Student` са собом доноси податке о оценама и предметима, али они су разбацани, па, ако би они били само тако штампали, десило би се да оцене буду измешане, тј. да се прикаже нпр. оцена из Математике, па из Географије, па Историје, па онда опет из Математике, па две из Географије, итд.



Слика 9. - Преглед оцена

Стога је у зрно `Student` додат атрибут:

```
Map<String, List<StudentsMarks> > topicsMarksOrdered;
```

Ово ће бити *HashMap*-а чији ће кључ бити име предмета, а вредност ће бити повезана листа са оценама. Ова мапа се попуњава приликом позива методе `getTopicsMarksOrdered`, позивом `formatTopicsMarksOrdered()` методе:

```
private Map<String, List<StudentsMarks>> formatTopicsMarksOrdered() {
     List<StudentsMarks> m = new ArrayList<StudentsMarks>(marks);
     topicsMarksOrdered = new HashMap<String, List<StudentsMarks> >();
     for(int i = 0; i < m.size(); i ++) {
          String s = m.get(i).getTopicName();
          if(topicsMarksOrdered.containsKey(s))
               topicsMarksOrdered.get(s).add(m.get(i));
          else {
               List<StudentsMarks> l = LinkedList<StudentsMarks>();
               l.add(m.get(i));
               topicsMarksOrdered.put(s, l);
          }
     }
     return topicsMarksOrdered;
}
```

Да би ђаку били приказани сви релевантни подаци о оценама, потребно је спојити неколико табела. Због тога је у овој ситуацији коришћен *view* - олакшава посао, а у исто време је ово и прилика да се демонстрира како се ради са њиме у Хајбернејту.

View је креиран на следећи начин:

```
CREATE VIEW
   students_marks AS
      SELECT
         stm.studentid, t.topicid, stm.teacherid, stm.markid, t.topic_name,
         (((te.first_name)::text || ' '::text) || (te.last_name)::text) AS teacher_name,
         mt.mark_type_name AS mark_type, m.value AS mark
      FROM
         (((( students_topics_marks stm JOIN topics t ON ((stm.topicid = t.topicid)))
            JOIN marks m ON ((stm.markid = m.markid)))
            JOIN mark_types mt ON ((m.marktype = mt.mark_type_id)))
            JOIN teachers te ON ((stm.teacherid = te.teacherid))
         );
```

Што се мапирања у Хајбернејту тиче, *StudentsMarks.hbm.xml* изгледа отприлике овако:

```
<hibernate-mapping>
   <class name="com.esd.model.StudentsMarks" table="students_marks">

      <composite-id>
         <key-property name="studentId">
            <column name="studentid" />
         </key-property>
         <key-property name="topicId">
             <column name="topicid" />
         </key-property>
         <key-property name="markId">
            <column name="markid" />
         </key-property>
         <key-property name="teacherId">
            <column name="teacherid" />
         </key-property>
      </composite-id>

      <property name="topicName" column="topic_name" />
      <property name="teacherName" column="teacher_name" />
      <property name="markType" column="mark_type" />
      <property name="mark" column="mark" />
   </class>
</hibernate-mapping>
```

Дакле, мапирање *view*-а на зрно иде исто као и код мапирања обичних табела (`StudentsMarks` зрно изгледа исто као и остала раније виђена зрна). Главна разлика је око *ID* поља - користи се композитни кључ, састављен од *id*-ја ђака, предмета, оцене и професора. Хајбернејту је неопходан јединствени кључ за сваку од табела које се мапирају, како би приликом нпр. чувања могао да одреди који тачно ред да сачува.

IV.3.5 AdminActionController

И код овог контролера механизам је сличан као и код претходна два - у зависности од параметра *action* приказује се одговарајућа страница, с тим што је овде избор већи - додавање/едитовање ђака, додавање/едитовање разреда, додавање/едитовање професора, додавање/едитовање предмета и додавање/едитовање типа оцене. И овде се подаци

похрањују у базу коришћењем `saveOrUpdate` функције, опет захваљујући Хајбернејт мапирању.

Оно што је ново је део *jsp* странице - кориснички дефинисани тагови. Овде је написан један, ради демонстрације.

Да би кориснички таг радио, потребно је прво је прво "описати" га, користећи *XML* датотеку која има екстензију *.tld*. У овом примеру то је *SaveConfirmation.tld* који је сачуван у *com.esd.view.tags*. Део те датотеке који се директно односи на овај таг изгледа овако:

```
<taglib>
<tlib-version>1.0</tlib-version>
  <short-name>saved</short-name>
  <tag>
     <description>Displays notification</description>
     <name>saved</name>
     <tag-class>com.esd.util.SaveConfirmation</tag-class>
     <body-content>JSP</body-content>

     <attribute>
        <name>save</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
     </attribute>
  </tag>
</taglib>
```

`short-name` таг је опште име за колекцију тагова који би могли да буду дефинисани овде. У датом примеру постоји само један - `saved`.

`description` је, као што само име каже, опис самог тага. У оквиру `name` тага се даје име тагу, оно преко чега ће бити позван на *jsp* страници. У `tag-class`-у се даје тачна локација класе која ће да дефинише понашање самог тага. И, на крају овог дела, у оквиру тага `body-content` се каже шта може да садржи тело овог тага - могуће опције су `empty`, `tagdependent` и `JSP`.

Ако је потребно омогућити прослеђивање неког атрибута кориснички дефинисаном тагу, онда ти атрибути морају бити описани у овој датотеци. У датом тагу је дефинисан један атрибут, чије је име дефинисано у `name` тагу - `save`. `required` таг може бити `true` и `false` и означава да ли је овај атрибут обавезан. Опција `rtexprvalue` служи да се подеси да ли се као атрибут прихвата израз или не.

Што се класе која дефинише понашање овог тага тиче, у овом случају је најбитнија функција `doStartTag`, која се активира када се наиђе на почетак тага:

```
public int doStartTag() throws JspException {
     if (save.equals("yes"))
          return TagSupport.EVAL_BODY_INCLUDE;

     return TagSupport.SKIP_BODY;
}
```

Као што се може видети, ако атрибут тага има вредност `yes`, онда ова функција враћа `EVAL_BODY_INCLUDE`, што значи да се садржај тага приказује. У супротном враћа `SKIP_BODY`, што значи да се садржај не приказује. Могуће је дефинисати и `doEndTag` функцију, која би се позивала када се наиђе на крај тага и она враћа `SKIP_PAGE` ако је потребно прескочити приказ остатка странице или `EVAL_PAGE` у супротном случају.

Наравно, кориснички дефинисани тагови нуде још много могућности, али то превазилази домен овог рада.

И, на крају, како се овај таг користи - пре свега је потребно “учитати” тај таг на *jsp* страницу:

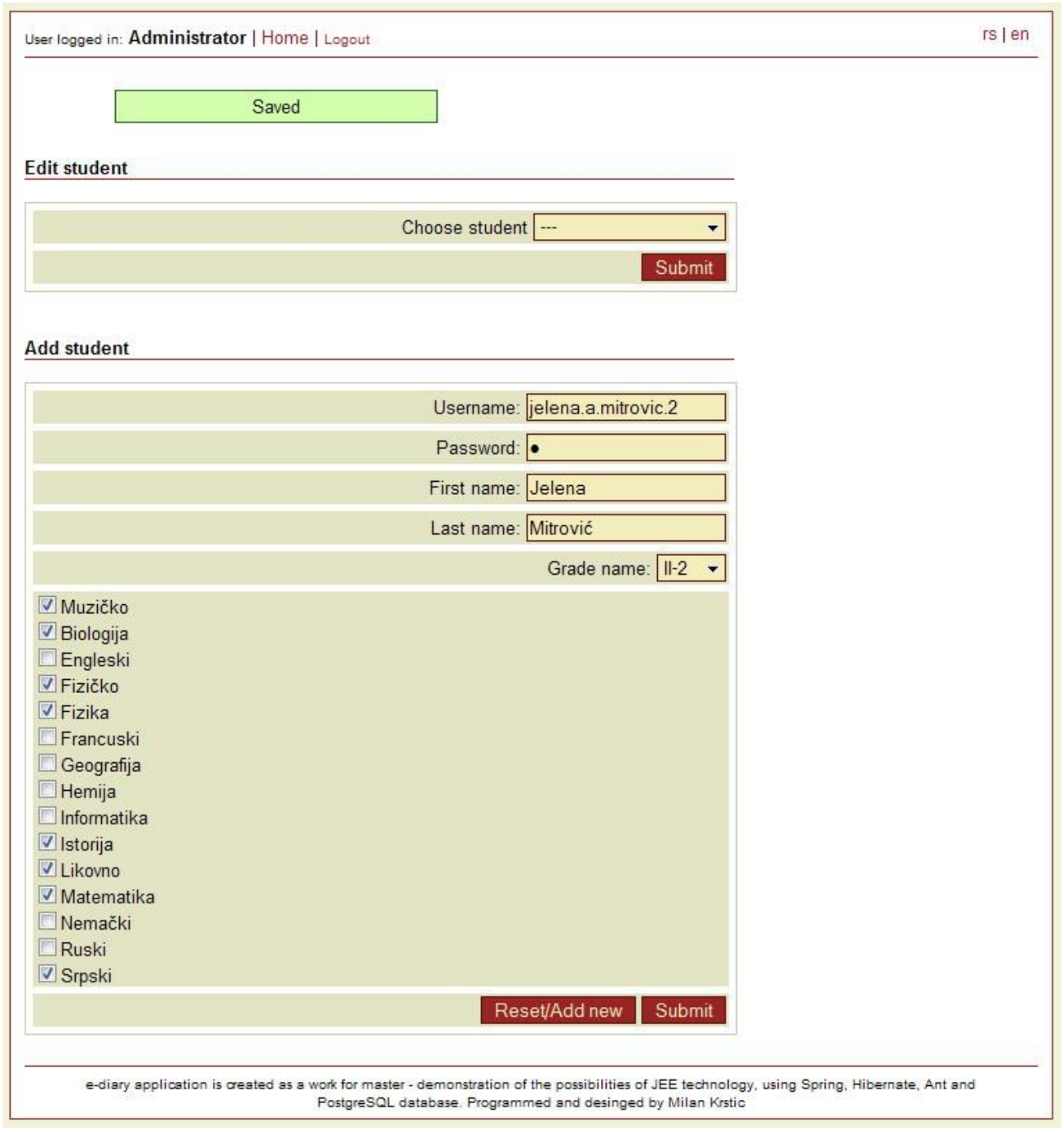
```
<%@ taglib uri="/WEB-INF/jsp/tags/SaveConfirmation.tld" prefix="notification" %>
```

Овим је речено где се налази дефиниција тог тага, као и који префикс ће се користити приликом позива.

Сам позив изгледа овако:

```
<notification:saved save="${param.save}">
  <br />
  <div class="notification">
     <fmt:message key="admin.saved" />
  </div>
  <br />
</notification:saved>
```

Као што је већ речено, ако је атрибут `save` различит од `yes`, тело тага се неће ни приказати. У супротном, исписаће поруку о успешном похрањивању података у базу.



Слика 10. - Изглед странице за уређивање података о ђаку, као и пример приказа резултата рада кориснички дефинисаног тага

IV.3.6 SignOutController

И, на крају имамо контолер који омогућава кориснику да се излогује из апликације. То се постиже једноставном акцијом у `handleRequest` функцији:

```
applicationSecurityManager.removeUser(request);
```

Да би ово било јасније, треба се присетити приче са почетка - `HttpRequestInterceptor` “пресреће” сваки захтев за страницама за које је потребна аутентикација и, ако не нађе активног корисника, приказује страницу за логовање. Самим тим, овом горе поменуто акцијом је активан корисник излогован.

`successView` овог контролера је подешен на “`redirect:signin.html`”, што значи да ће се кориснику, након што се успешно излогује са система, приказати прозор за логовање.

V Закључак

Као што је већ раније речено, циљ ове апликације је демонстрација могућности JEE технологије, тако да је функционалност саме апликације била у другом плану.

Идеја аутора овог пројекта је била развој електронског дневника (независно од мастер рада), са свим функционалностима које би један такав дневник требало да има, а који би био објављен као отворени код и дат свима на коришћење. Међутим, ту идеју су омеле животне околности, тако да је развијена само основна верзија.

Овај рад и апликација приказују главне могућности примењених технологија, и представљају добру базу за даље усавршавање ове тематике.

VI Литература

1. http://www.java.com/en/
2. http://en.wikipedia.org/wiki/Java_(programming_language)
3. http://blogs.sun.com/jonathan/entry/better_is_always_different
4. http://en.wikipedia.org/wiki/JEE
5. http://www.springsource.org/
6. http://en.wikipedia.org/wiki/Spring_Framework
7. http://www.hibernate.org/
8. http://ant.apache.org/
9. http://www.developer.com/java/j2me/article.php/989631/Building-with-Ant-Introduction.htm
10. http://www.postgresql.org/
11. http://java.sun.com/blueprints/code/projectconventions.html#23136
12. *Anil Hemrajani, "Agile Java Development With Spring, Hibernate and Eclipse"* – Писац даје преглед технологија, а затим развија апликацију за бележење радних сати, уводећи нове опције уз помоћ нових технологија које том приликом појашњава.
13. *Marty Hall, "Core Servlets and JavaServer Pages, Volume 1"* – Веома опширна и свеобухватна књига, са мноштвом примера. Пружа комплетан приказ ове технологије.
14. *Marty Hall, Larry Brown, Yaakov Chaikin, "Core Servlets and JavaServer Pages, Volume 2"* – Наставак претходне књиге, са још више детаља, уз додатак тема које нису обухваћене првом књигом, као и неких напредних техника и могућности