**Spring MVC example.**

В этом проекте показан базовый пример использования фреймворков Spring MVC 3+ и Hibernate.

**Запуск примера:**

Для запуска примера вам нужно иметь:

- Исходники проекта. Их можно скачать с Github репозитория(Lab 2): <https://github.com/Rudeg/Spring-MVC-Example>

**-** Одну из современных IDE для работы с Java. (IntelliJ IDEA, Eclipse, Netbeans).

- Tomcat версии 7.\*

- MySQL версии 5.\*

Вам нужно скачать исходники проекта с репозитория.

После этого, выполнить импорт проекта в IDE.

Следующим действием нужно указать настройки подключения к базе данных (мы используем MySQL). Они указываются в файле database.properties.

Далее настроить автоматический деплой на аппликейшн сервер(Tomcat). После этих действий проект должен заработать.

Видео запуска проекта в разных IDE:

* Idea: <http://screencast.com/t/x1ISPAaEx>
* Eclipse: <http://screencast.com/t/huO1WyaLCD>
* Netbeans: <http://screencast.com/t/LYXUpCwADX>

*Запуск проекта без IDE:*

Для запуска проекта без IDE вам потребуется:

- Apache Maven версии 3.\*

- Tomcat версии 7.\*

- MySQL версии 5.\*

Для начала нужно указать настройки подключения к базе данных (мы используем MySQL). Они указываются в файле database.properties.

После этого нужно собрать исходники проекта в war файл. Это выполняется с помощью Mavena(команда mvn package). Чтобы это выполнить, нужно перейти в папку с исходниками проектами (в ней находится .pom файл) и выполнить команду “mvn package”.

После этого должна появиться папка Target, в которой будет war файл SpringExample.war.

Теперь этот файл нужно скопировать в папку Tomcata - webapp. Это делается для того, чтоб при запуске томката он автоматически развернул war файл и запустил его. Tomcat запускается скриптом startup из папки bin.

После всех этих действий, проект будет доступен по адрессу <http://localhost:port/warname> , где

port – порт на котором запускается Tomcat; warname – имя War файла.

Видео запуска без IDE - <http://screencast.com/t/tEDioL6Y>

В этом примере были реализовано следующее:

- регистрация и логин пользователя;

- валидация (серверная и клиентская);

- просмотр и добавление постов;

- просмотр и добавление комментариев к постам.

Далее в этот пример можно добавить например редактирование\удаление постов и комментариев, администрирование и различные уровни доступа.

**Validation:**

В Spring присутствует возможность валидации бинов (Bean Validation) JSR-303. Для валидации мы будем использовать Hibernate Validation. Рассмотрим валидацию на примере класса User, который содержит сущностью аутентифицированного пользователя.

Для использования валидации необходимо создать POJO класс. **POJO** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***Plain Old Java Object***) — «простой Java-объект в старом стиле», простой [Java](http://ru.wikipedia.org/wiki/Java" \o "Java)-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели.

Итак, мы создали POJO класс для пользователя. У нас он также будет использоваться как доменная модель, которая будет сохраняться в базе данных и оттуда извлекаться. Т.к. этот класс реализует «бизнес-модель», то нам надо снабдить ее необходимыми и не избыточными свойствами. То есть добавить приватные поля, который будут отражать сущность модели, а также публичные геттеры и сеттеры для них. В данном случае нам в первую очередь понадобятся поля “username” (логин пользователя), “lastname” (фамилия), “firstname” (имя), “passwordHash” (хеш от пароля, т.к. хранить пароли в базе в открытом виде или даже зашифрованном – не секурно и они могут быть скомпроментированы. По правде говоря в таком виде их тоже лучше не хранить.), “sex” (пол пользователя мужчина/женщины/иное), “progLang” (языки программирования), “receiveNewsLetter” (хочет ли пользователь получать почтовую рассылку), “country” (страна проживания пользователя).

Итак, мы выбрали достаточную модель пользователя – теперь надо аннотировать необходимые поля ограничениями. Для этого нам понадобится пространство имен org.hibernate.validator.constraints.\*. Здесь присутствуют наиболее часто используемые валидаторы, среди них: ConstraintComposition, CreditCardNumber, Email, Length, ModCheck, NotBlank, NotEmpty, Range, SafeHtml, ScriptAssert, URL. Нам, к счастью, понадобятся пользо NotNull, NotBlank. NotNull явно указывает что в данном поле должно что-то присутствовать, а NotBlank указывает что оно не должно быть null, и после того как его тримнули (убрали пробелы в начале и в конце строки) его длина должна быть не нулевой. Пример использования вы можете посмотреть в папке src/main/java/model/entity/User.

В результате всех этих действий можно указывать в \*.jsp поля ошибок и они сами будут подставляться если поле не было провалидировано. Это выглядит так:

**…**

**<form:errors path="\*" cssClass="errorblock" element="div"/>**

**…**

**<form:errors path="firstname" cssClass="error"/>**

**….**

Полный код файла вы можете найти в репозитории. В данном случае это src/main/webapp/WEB-INF/pages/registration.jsp.

Также будет полезно знать что можно задавить локализацию и настраиваемое отображение ошибок в файле src/main/resources/messages.properties. Выглядит это так:  
**NotBlank.user.username = Username may not be empty!**

**NotBlank.user.firstname = First name may not be empty!**

**NotBlank.user.lastname = Last name may not be empty!**

Также можно создавать свои собственные валидаторы, но мы это рассматривать пока не будем.

**Hibernate:**

Первым действием мы создаем контекст хибернейта, в котором указываем все настройки. Они вынесены в отдельный файл **– database.properties**, в котором описаны следующие настройки:

jdbc.driver=com.mysql.jdbc.Driver - драйвер

jdbc.url=jdbc:mysql://localhost:3306/springexample – url коннекта к ДБ + указывается база данных

jdbc.user=root – имя пользователя

jdbc.password= - пароль.

После этого мы должны добавить этот контекст в web.xml конфиг:

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

classpath:context/dispatcher-servlet.xml

classpath:context/hibernate-context.xml

classpath:context/security-app-context.xml

</param-value>

</context-param>

Теперь мы можем создавать классы-сущности. Далее приведен пример сущности User:

package com.springexample.common.model.Entity;

import com.springexample.common.model.SexEnum;

import org.hibernate.validator.constraints.NotBlank;

import javax.persistence.\*;

import javax.validation.constraints.NotNull;

import java.util.Set;

@Entity

@Table(name="users")

public class User extends BaseEntity<Long> {

//textboxes

@Column(nullable = false)

@NotBlank

private String username;

@Column(nullable = false)

@NotBlank

private String firstname;

@Column(nullable = false)

@NotBlank

private String lastname;

@Column(nullable = false)

private String passwordHash;

@Column(name = "sexValue", nullable = false)

@Enumerated(EnumType.STRING)

//radio button

@NotNull

private SexEnum sex;

//checkboxes

private String [] progLang;

@Column(nullable = false)

private Boolean receiveNewsLetter;

@Column(nullable = false)

@NotBlank

private String country;

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})

@JoinTable(name = "ex\_user\_role", joinColumns = @JoinColumn(name = "user\_id"), inverseJoinColumns = @JoinColumn(name = "role\_id"))

private Set<Role> userRoles;

@OneToMany(mappedBy = "user", fetch = FetchType.EAGER)

private Set<Post> posts;

@OneToMany(mappedBy = "user", fetch = FetchType.EAGER)

private Set<Comment> comments;

Getters and Setters…

}

Далее мы создаем **DAO** (<http://ru.wikipedia.org/wiki/Data_Access_Object>) и **Service** уровни.

Пример интерфейса и имплементации абстрактного DAO:

package com.springexample.common.dao;

import org.hibernate.criterion.Criterion;

import java.io.Serializable;

import java.util.List;

public interface AbstractDao<E, I extends Serializable> {

E findById(I id);

void saveOrUpdate(E e);

void delete(E e);

List<E> findByCriteria(Criterion criterion);

}

package com.springexample.common.dao.Impl;

import com.springexample.common.dao.AbstractDao;

import org.hibernate.Criteria;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.criterion.Criterion;

import org.springframework.beans.factory.annotation.Autowired;

import java.io.Serializable;

import java.util.List;

public abstract class AbstractDaoImpl<E, I extends Serializable> implements AbstractDao<E,I> {

private Class<E> entityClass;

protected AbstractDaoImpl(Class<E> entityClass) {

this.entityClass = entityClass;

}

@Autowired

private SessionFactory sessionFactory;

public Session getCurrentSession() {

return sessionFactory.getCurrentSession();

}

@Override

public E findById(I id) {

return (E) getCurrentSession().get(entityClass, id);

}

@Override

public void saveOrUpdate(E e) {

getCurrentSession().saveOrUpdate(e);

}

@Override

public void delete(E e) {

getCurrentSession().delete(e);

}

@Override

public List<E> findByCriteria(Criterion criterion) {

Criteria criteria = getCurrentSession().createCriteria(entityClass);

criteria.add(criterion);

return criteria.list();

}

}

Для получении сессии мы используем класс **SessionFactory**, который был описан ранее в контексте.

Далее пример реализации **UserDao**:

package com.springexample.common.dao.Impl;

import com.springexample.common.dao.UserDao;

import com.springexample.common.model.Entity.User;

import org.hibernate.criterion.MatchMode;

import org.hibernate.criterion.Restrictions;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public class UserDaoImpl extends AbstractDaoImpl<User, String> implements UserDao {

protected UserDaoImpl() {

super(User.class);

}

@Override

public void saveUser(User user) {

saveOrUpdate(user);

}

@Override

public User getByID(Long id) {

return (User)getCurrentSession().get(User.class, id);

}

@Override

public User getByUsername(String username) {

return findByCriteria(Restrictions.like("username", username, MatchMode.START)).get(0);

}

@Override

public List<User> findUsers(String userName) {

return findByCriteria(Restrictions.like("username", userName, MatchMode.START));

}

}

Следующим этапом будет реализация Service уровня. Пример реализации Service интерфейса для юзера:

package com.springexample.common.service;

import com.springexample.common.model.Entity.User;

import java.util.List;

public interface UserService {

User findByUserName(String userName);

void saveUser(User user);

User getById(Long id);

void deleteUser(String userName);

List<User> findUsers(String user);

}

**UserService implementation:**

package com.springexample.common.service.Impl;

import com.springexample.common.dao.UserDao;

import com.springexample.common.model.Entity.User;

import com.springexample.common.service.UserService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service("userService")

@Transactional(readOnly = true)

public class UserServiceImpl implements UserService {

@Autowired

private UserDao userDao;

@Override

public User findByUserName(String userName) {

return userDao.getByUsername(userName);

}

@Override

@Transactional(readOnly = false)

public void saveUser(User user) {

userDao.saveOrUpdate(user);

}

@Override

public User getById(Long id) {

return userDao.getByID(id);

}

@Override

@Transactional(readOnly = false)

public void deleteUser(String userName) {

User user = userDao.findById(userName);

if (user != null) {

userDao.delete(user);

}

}

@Override

public List<User> findUsers(String user) {

return userDao.findUsers(user);

}

}

После этого мы можем использовать наши Service классы в контроллерах. Пример метода, в котором реализована регистрация пользователя:

@Autowired

private UserService userService;

@Autowired

private RoleService roleService;

@RequestMapping(value = "/registration", method = RequestMethod.POST)

public String registration(@Valid @ModelAttribute("user") RegistrationUser user, BindingResult result, Model m) {

if (result.hasErrors()) {

//add objects to model

m.addAttribute("programmingLanguageList", initProgramLang());

m.addAttribute("countryMap", initCountry());

return "registration";

}

Role userRole = roleService.getById(2L); //ROLE\_USER

Set<Role> roles = new HashSet<Role>();

roles.add(userRole);

User u = new User();

u.setPasswordHash(databasePasswordSecurerBean.secureUser(user, user.getPassword()));

u.setComments(user.getComments());

u.setCountry(user.getCountry());

u.setFirstname(HtmlUtils.htmlEscape(user.getFirstname()));

u.setLastname(HtmlUtils.htmlEscape(user.getLastname()));

u.setPosts(user.getPosts());

u.setProgLang(user.getProgLang());

u.setReceiveNewsLetter(user.getReceiveNewsLetter());

u.setSex(user.getSex());

u.setUsername(HtmlUtils.htmlEscape(user.getUsername()));

u.setId(user.getId());

u.setUserRoles(roles);

userService.saveUser(u);

m.addAttribute("username", user.getUsername());

return "redirect:login";

}

**Spring Security:**

Для Spring Security(<http://docs.spring.io/spring-security/site/index.html>) мы тоже создаем свой контекст файл и также указываем его в web.xml (см пример выше). Ниже представлен пример контекста:

<?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:security="http://www.springframework.org/schema/security"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/security

http://www.springframework.org/schema/security/spring-security.xsd">

<bean id="passwordEncoder" class="org.springframework.security.authentication.encoding.ShaPasswordEncoder"/>

<bean id="bean" class="com.springexample.common.constraits.DatabasePasswordSecurerBean"

depends-on="dataSource">

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

</bean>

<security:http auto-config="true">

<security:intercept-url pattern="/main" access="ROLE\_ADMIN,ROLE\_USER" />

<security:form-login login-page="/login" default-target-url="/main"

authentication-failure-url="/loginfailed"/>

<security:logout logout-success-url="/logout"/>

</security:http>

<security:authentication-manager>

<security:authentication-provider>

<security:password-encoder ref="passwordEncoder">

<security:salt-source user-property="username"/>

</security:password-encoder>

<security:jdbc-user-service data-source-ref="dataSource"

users-by-username-query="select username,passwordHash,'true' from users where username=?"

authorities-by-username-query="

select u.username, ur.authority from users u, user\_roles ur

where u.id = ur.id and u.username =?"

/>

</security:authentication-provider>

</security:authentication-manager>

</beans>

Для шифрования пароля использовался класс DatabasePasswordSecurerBean.

package com.springexample.common.constraits;

import com.springexample.common.model.Entity.User;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.RowCallbackHandler;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

import org.springframework.security.authentication.encoding.PasswordEncoder;

import java.sql.ResultSet;

import java.sql.SQLException;

public class DatabasePasswordSecurerBean extends JdbcDaoSupport {

@Autowired

private PasswordEncoder passwordEncoder;

public void secureDatabase() {

getJdbcTemplate().query("select username, password from users",

new RowCallbackHandler(){

@Override

public void processRow(ResultSet rs) throws SQLException {

String username = rs.getString(0);

String password = rs.getString(1);

String encodedPassword =

passwordEncoder.encodePassword(password, username);

getJdbcTemplate().update("update users set password = ? where username = ?", encodedPassword, username);

System.out.println("Updating password for username:"+username+" to: "+encodedPassword);

logger.debug("Updating password for username:" + username + " to: " + encodedPassword);

}

});

}

//Test Secure password method for new user

public String secureUser(User user, String password) {

return passwordEncoder.encodePassword(password, user.getUsername());

}

}

В контексте мы указываем конфиги для URL. Также мы использовали кастомную login форму. Далее были написаны 2 запроса, которые служат для авторизации пользователя.

users-by-username-query="select username,passwordHash,'true' from users where username=?"

authorities-by-username-query="select u.username, ur.authority from users u, user\_roles ur where u.id = ur.id and u.username =?"