Учимся готовить: Spring 3 MVC + Spring Security + Hibernate

Добрый день! Меня зовут Антон Щастный.

Это моя очередная статья, посвящённая разработке веб приложений на Java. Хочу предложить вам сделать небольшую систему учёта клиентов, написанную с использованием фреймворка Spring и библиотеки Hibernate.

Что будет в приложении:

Будет простой менеджер контактов, позволяющий добавлять в базу новые записи, просматривать имеющиеся, удалять ненужные. Сведения будут храниться в базе данных. Доступ к приложению – через веб, с аутентификацией и авторизацией пользователей.

Что будем использовать:

- веб фреймворк Spring MVC,
- фреймворк Spring Security,
- ORM библиотеку Hibernate,
- MySQL в качестве СУБД.

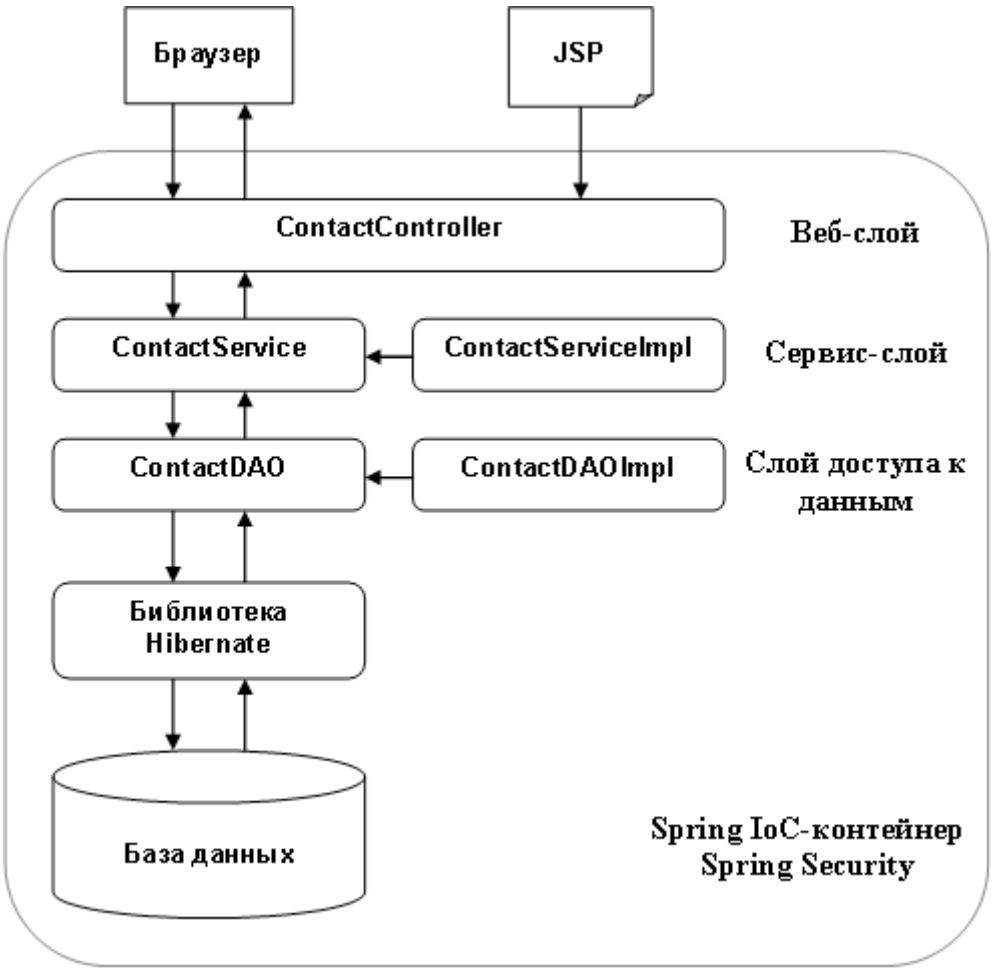
Инфраструктура:

- редактор SpringSource Tool Suite,
- сборщик проектов Maven,
- система логгирования Log4j,
- постоянное подключение к интернету.

В моей предыдущей статье о Spring MVC был упущен ряд моментов по использованию аннотаций в Java-коде и применению Maven для сборки проекта. В данной статье я попытался исправить упущение.

Цель статьи – показать начинающим веб разработчикам совместное использование различных технологий платформы Java.

Содержание



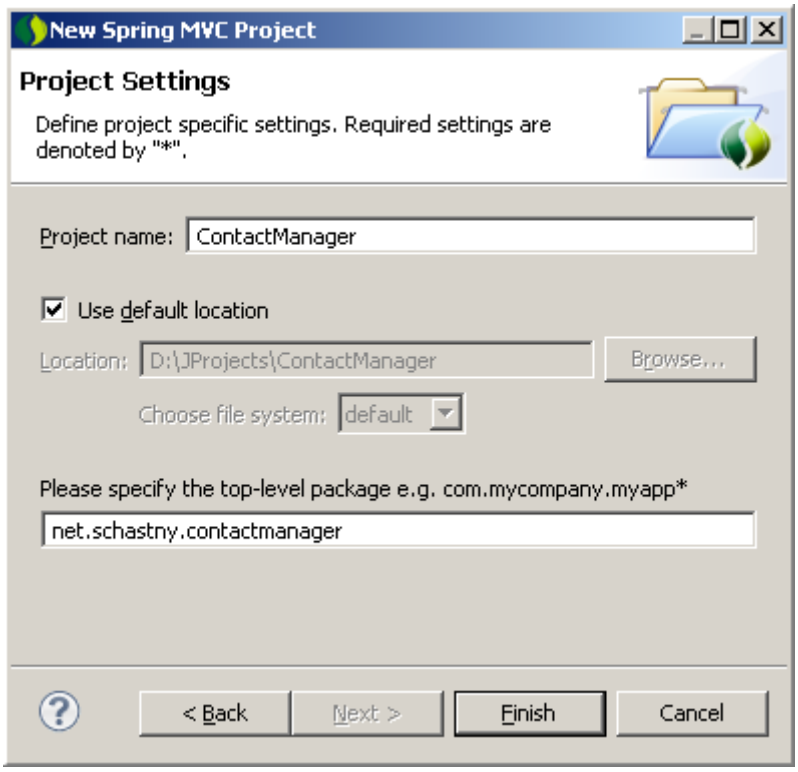
Архитектура разрабатываемого приложения.

1. Создание нового проекта в IDE

Проект будем разрабатывать в SpringSource Tool Suite (STS) – удобном редакторе, основанном на Eclipse IDE.

После установки STS создадим шаблонный проект:
File > New > Spring Template Project > Spring MVC Project.

Project name: *ContactManager*
Top level package: *net.schastny.contactmanager*



Редактор создаст шаблонный веб проект с базовыми настройками, имеющий следующую структуру:

src/main/java	Java-классы приложения.
src/main/resources	Все остальные файлы (ресурсы), необходимые для работы: например файл настроек логгера *log4j.xml*, папка *META-INF*.
src/test/java	Папка для тестов JUnit.
src/test/resources	Папка ресурсов юнит-тестов.
src/main/webapp	Папка, в которой размещаются файлы веб составляющей менеджера контактов: дескриптор развёртывания приложения web.xml, xml-файлы с настройками фреймворка Spring, jsp-страницы. *Внимание, статические ресурсы приложения: картинки, css, js-скрипты лежат в папке resources.*
pom.xml	Файл настроек проекта для Maven'а, содержит наименование проекта и перечень зависимостей проекта от сторонних библиотек.

Поменяем в настройках проекта (pom.xml) версию Spring с *3.0.4.RELEASE* на *3.0.5.RELEASE*.

Примечание: как создавать Maven-проекты вне IDE читайте здесь.

2. Создание структуры пакетов

net.schastny.contactmanager.dao	**Слой доступа к данным.** В нём будем размещать Data Access Objects – объекты доступа к данным.
net.schastny.contactmanager.domain	**Доменный слой.** Именно здесь лежат POJO-классы, описывающие объекты-сущности системы. В нашем случае это класс Contact.
net.schastny.contactmanager.service	**Сервис-слой приложения.** Содержит интерфейсы, в которых описан функционал приложения. Также содержит одну или несколько практических реализаций этих интерфейсов.

net.schastny.contactmanager.web	**Веб-слой приложения.** Здесь лежат классы-контроллеры, описывающие порядок взаимодействия пользователя с системой через веб.

Примечание: класс-контроллер, созданный редактором, можно удалить.

3. Добавление класса сущности в модель домена

```
package net.schastny.contactmanager.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "CONTACTS")
public class Contact {

	@Id
	@Column(name = "ID")
	@GeneratedValue
	private Integer id;

	@Column(name = "FIRSTNAME")
	private String firstname;

	@Column(name = "LASTNAME")
	private String lastname;

	@Column(name = "EMAIL")
	private String email;

	@Column(name = "TELEPHONE")
	private String telephone;

	// Getters and setters
}
```

Не забудьте добавить методы геттеры и сеттеры для свойств бина.

Добавим зависимости проекта в файл *pom.xml*. Обратите внимание, что класс из модели домена не будет зависить ни от Спринга, ни от Хибернейта.

```
		<dependency>
			<groupId>javax.persistence</groupId>
			<artifactId>persistence-api</artifactId>
			<version>1.0</version>
		</dependency>
```

Пояснение к использованным аннотациям.

@Entity	Класс представляет объект, который нужно долговременно хранить.
@Table(name = «CONTACTS»)	Свойства класса будем хранить в таблице CONTACTS.
@Column(name = «FIRSTNAME»)	Это свойство будет храниться в столбце firstname.
@Id	Это поле уникальное для объектов, то есть по нему будем искать объекты.
@GeneratedValue	Значение этого поля будет назначаться не нами, а генерироваться автоматически.

4. Слой доступа к данным

Интерфейс и практическая реализация объекта доступа к данным.

```
package net.schastny.contactmanager.dao;

import java.util.List;
import net.schastny.contactmanager.domain.Contact;

public interface ContactDAO {

	public void addContact(Contact contact);

	public List<Contact> listContact();

	public void removeContact(Integer id);
}
```

```
package net.schastny.contactmanager.dao;

import java.util.List;

import net.schastny.contactmanager.domain.Contact;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class ContactDAOImpl implements ContactDAO {

	@Autowired
	private SessionFactory sessionFactory;

	public void addContact(Contact contact) {
		sessionFactory.getCurrentSession().save(contact);
	}

	@SuppressWarnings("unchecked")
	public List<Contact> listContact() {

		return sessionFactory.getCurrentSession().createQuery("from Contact")
				.list();
	}

	public void removeContact(Integer id) {
		Contact contact = (Contact) sessionFactory.getCurrentSession().load(
				Contact.class, id);
		if (null != contact) {
			sessionFactory.getCurrentSession().delete(contact);
		}

	}
}
```

Обновим зависимости проекта в файле *pom.xml*. Добавим Хибернейт.

```
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-core</artifactId>
			<version>3.3.2.GA</version>
		</dependency>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-annotations</artifactId>
			<version>3.3.1.GA</version>
		</dependency>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-commons-annotations</artifactId>
			<version>3.3.0.ga</version>
		</dependency>
```

Пояснение к использованным аннотациям.

@Repository	Аннотация показывает, что класс функционирует как репозиторий и требует наличия прозрачной трансляции исключений. Преимуществом трансляции исключений является то, что слой сервиса будет иметь дело с общей иерархией исключений от Спринга (DataAccessException) вне зависимости от используемых технологий доступа к данным в DAO слое.
@Autowired	Аннотация позволяет автоматически установить значение поля SessionFactory.

5. Сервис-слой

Опишем интерфейс взаимодействия пользователя с системой.

```
package net.schastny.contactmanager.service;

import java.util.List;
import net.schastny.contactmanager.domain.Contact;

public interface ContactService {

        public void addContact(Contact contact);

        public List<Contact> listContact();

        public void removeContact(Integer id);
}
```

Практическая реализация интерфейса сервиса – класс ContactServiceImpl.java.

```
package net.schastny.contactmanager.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import net.schastny.contactmanager.dao.ContactDAO;
import net.schastny.contactmanager.domain.Contact;

@Service
public class ContactServiceImpl implements ContactService {

    @Autowired
    private ContactDAO contactDAO;

    @Transactional
    public void addContact(Contact contact) {
        contactDAO.addContact(contact);
    }

    @Transactional
    public List<Contact> listContact() {

        return contactDAO.listContact();
    }

    @Transactional
    public void removeContact(Integer id) {
        contactDAO.removeContact(id);
    }
}
```

Обновим зависимости проекта в файле *pom.xml*. Добавим поддержку транзакций.

```
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-tx</artifactId>
                        <version>${org.springframework-version}</version>
                </dependency>
```

```
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-orm</artifactId>
                        <version>${org.springframework-version}</version>
                </dependency>
```

Пояснение к использованным аннотациям.

@Service	Мы используем данную аннотацию, чтобы объявить, что этот класс представляет сервис – компонент сервис-слоя. Сервис является подтипом класса @Component. Использование данной аннотации позволит искать бины-сервисы автоматически (смотрите далее в root-context.xml).
@Transactional	Перед исполнением метода помеченного данной аннотацией начинается транзакция, после выполнения метода транзакция коммитится, при выбрасывании RuntimeException откатывается.

6. Добавление веб

Настройку веб приложения начнём с самого главного – с тьмы xml-файлов. В большом приложении очень удобно хранить различные настройки в разных файлах.

После xml-портянок напишем простенький контроллер.

Файлы настроек вы можете найти по ссылкам ниже.

web.xml	Это **дескриптор развертывания**. Файл, который описывает настройки развертывания приложения на сервере.
spring/root-context.xml	**Root Context**. Контекст всего приложения. Бины, описанные здесь, будут доступны всем сервлетам и фильтрам. Также здесь следует описывать бины безопасности и доступа к данным (в нашей конфигурации они вынесены в отдельные файлы).
spring/data.xml	Файл с настройками **ресурсов для работы с данными**.
spring/security.xml	Файл с **настройками безопасности**.
spring/appServlet/servlet-context.xml	**DispatcherServlet Context**. Определяет настройки одного сервлета и бины, которые доступны только этому сервлету.
spring/appServlet/controllers.xml	Файл с настройками **контроллеров** для данного сервлета.

spring/security.xml

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
                xmlns:beans="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.0.xsd">

</beans:beans>
```

Пояснение к используемым бинам.

messageSource	Бин для обеспечения интернациолизации приложения. Ниже мы создадим файлы **messages_en.properties** и **messages_ru.properties** с локализованными сообщениями на русском и английском.
propertyConfigurer	Для загрузки файла с настройками БД **jdbc.properties**.

dataSource	**Датасорс**, используется для подключения к БД. Мы предоставляем класс jdbc-драйвера, имя пользователя, пароль, другие настройки.
sessionFactory	Это бин конфигурации Хибернейта. В файле **hibernate.cfg.xml** будут содержаться маппинги на классы сущностей.
transactionManager	Бин настройки **менеджера транзакций**. Мы используем менеджер транзакций для управления транзакциями приложения.

Файлы:
/src/main/resources/messages_en.properties
/src/main/resources/messages_ru.properties
/src/main/resources/hibernate.cfg.xml
WEB-INF/jdbc.properties

7. Контроллер

```
package net.schastny.contactmanager.web;

import java.util.Map;

import net.schastny.contactmanager.domain.Contact;
import net.schastny.contactmanager.service.ContactService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class ContactController {

	@Autowired
	private ContactService contactService;

	@RequestMapping("/index")
	public String listContacts(Map<String, Object> map) {

		map.put("contact", new Contact());
		map.put("contactList", contactService.listContact());

		return "contact";
	}

	@RequestMapping("/")
	public String home() {
		return "redirect:/index";
	}

	@RequestMapping(value = "/add", method = RequestMethod.POST)
	public String addContact(@ModelAttribute("contact") Contact contact,
			BindingResult result) {

		contactService.addContact(contact);

		return "redirect:/index";
	}

	@RequestMapping("/delete/{contactId}")
	public String deleteContact(@PathVariable("contactId") Integer contactId) {

		contactService.removeContact(contactId);

		return "redirect:/index";
	}
}
```

Пояснение к использованным аннотациям.

@Controller	Аннотация используется для магического превращения любого java класса в класс контроллера. Данный класс представляет собой компонент, похожий на *обычный сервлет (HttpServlet)* (работающий с объектами HttpServletRequest и HttpServletResponse), но с расширенными возможностями по применению в MVC-приложениях.
@RequestMapping	Аннотация используется для маппинга урл-адреса запроса на указанный метод или класс. Запрос можно маппить как на метод класса, так и на целый класс. Допускается указывать конкретный HTTP-метод, который будет обрабатываться (GET/POST), передавать параметры запроса.
@ModelAttribute	Аннотация, связывающая параметр метода или возвращаемое значение метода с атрибутом модели, которая будет использоваться при выводе jsp-страницы.
@PathVariable	Аннотация, которая показывает, что параметр метода должен быть связан с переменной из урл-адреса.

8. Вид

/src/main/webapp/WEB-INF/views/contact.jsp

9. Запуск приложения

До запуска системы нам необходимо сделать ещё две вещи:

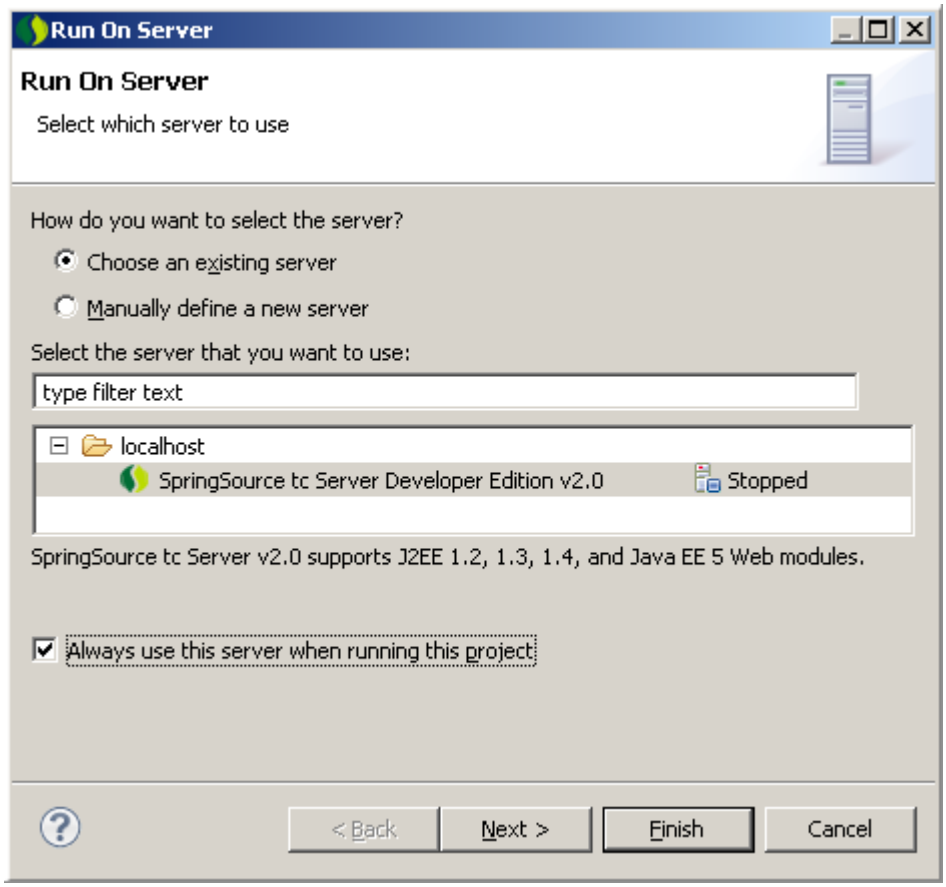
Во-первых, создать **базу данных для приложения** и пользователя под неё в соответствии с настройками файла *jdbc.properties*:

```
CREATE USER contactmanager@localhost identified BY '1234';
GRANT usage ON *.* TO contactmanager@localhost identified BY '1234';
CREATE DATABASE IF NOT EXISTS contactmanager;
GRANT ALL privileges ON contactmanager.* TO contactmanager@localhost;
USE contactmanager;
CREATE TABLE CONTACTS
  (
     id        INT PRIMARY KEY AUTO_INCREMENT,
     firstname VARCHAR(30),
     lastname  VARCHAR(30),
     telephone VARCHAR(15),
     email     VARCHAR(30),
     created   TIMESTAMP DEFAULT NOW()
  );
```

Во-вторых, обновить зависимости в *pom.xml*.

```
                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <version>5.1.6</version>
                </dependency>
```

Теперь запустим приложение на сервере: *ContactManager > Run As > Run on Server > SpringSource tc Server Developer Edition v2.0*.



Откройте адрес http://localhost:8080/ContactManager в браузере.

Примечание: Проверьте кодировку базы данных, таблиц в БД, соединения с БД, кодировку проекта в редакторе, так как вместо родных кириллических букв в браузере могут появиться кракозябры.

Для отладки работы Хибернейта можете в файл *src/main/resources/log4j.xml* добавить следующую строку:

```
        <!--org.hibernate.SQL logger-->
        <logger name="org.hibernate">
                <level value="info"/>
        </logger>
```

10. Безопасность

Обновим зависимости проекта в файле *pom.xml*. Добавим модули Spring Security.

```
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-web</artifactId>
                        <version>${org.springframework-version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-config</artifactId>
                        <version>${org.springframework-version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-core</artifactId>
                        <version>${org.springframework-version}</version>
                </dependency>
```

Добавим в *web.xml* фильтр безопасности, действующий на всё приложение.

```
        <filter>
                <filter-name>springSecurityFilterChain</filter-name>
                <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
        </filter>
        <filter-mapping>
                <filter-name>springSecurityFilterChain</filter-name>
                <url-pattern>/*</url-pattern>
        </filter-mapping>
```

В целях безопасности страницу логина *не следует помещать внутри папки WEB-INF*, так что разместим её в корне веб приложения – в папке webapp.
webapp/login.jsp

Там же разместим страницу ошибки 403: webapp/error403.jsp

Подредактируем файл security.xml.
security.xml

Пояснение к сделанным настройкам в файле security.xml

Выделим в приложении три группы пользователей (роли пользователей):

- анонимный пользователь (ROLE_ANONYMOUS),
- просто пользователь (ROLE_USER),
- администратор (ROLE_ADMIN).

Разграничим доступ к приложению:

- Сделаем возможность просматривать контакты доступной всем категориям пользователей.
- Добавлять новые контакты смогут только зарегистрированные пользователи.
- Удалять записи сможет только админ.

Где хранить данные о пользователях

Можно прямо в конфигурационном файле, если их немного. Можно в базе данных, можно в LDAP-хранилище.

Пример записи (*security.xml*) при использовании БД для хранения паролей.

```
        <authentication-manager>
                <authentication-provider>
                        <jdbc-user-service data-source-ref="dataSource" />
                </authentication-provider>
        </authentication-manager>
```

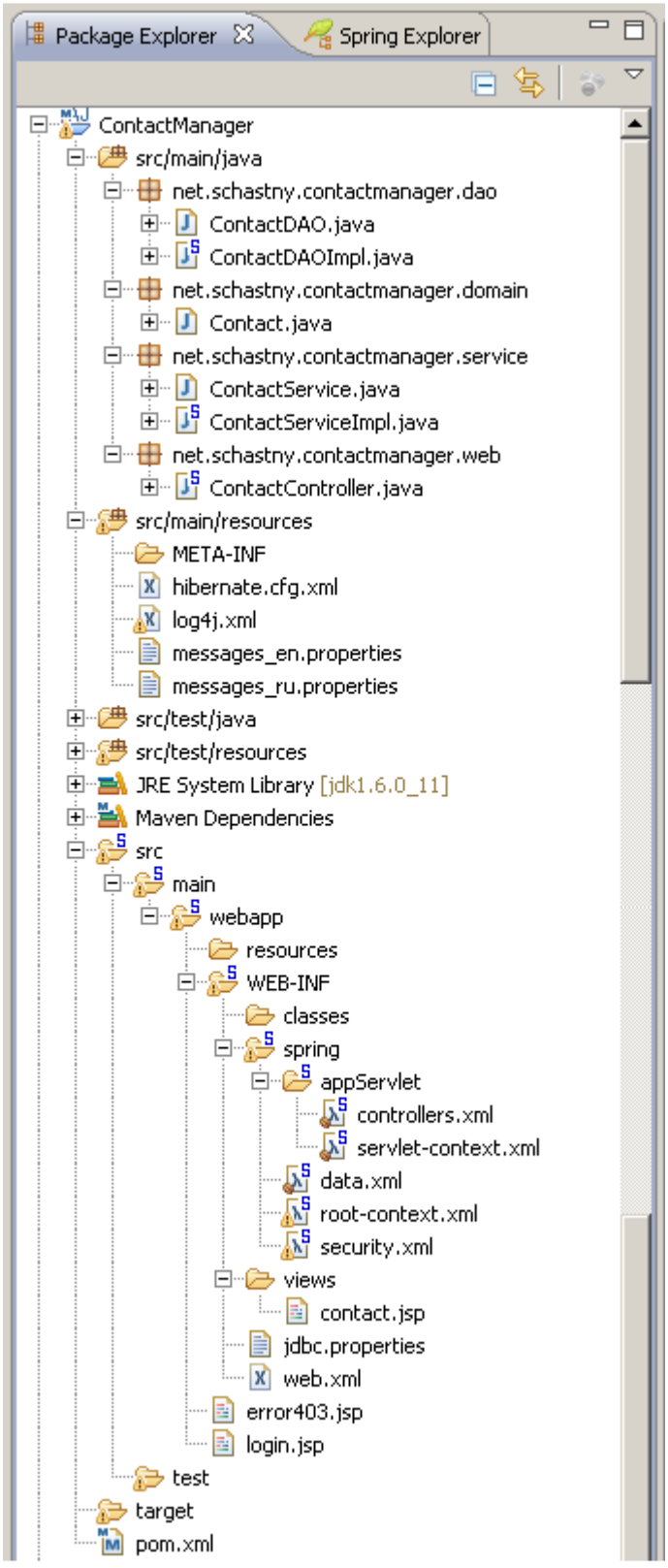
Пояснения к некоторым элементам.

`<remember-me />`	Поддержка мультисессии. После повторного запуска браузера не нужно заново логиниться.
`<anonymous username="guest" granted-authority="ROLE_ANONYMOUS" />`	Определение анонимного пользователя и наделение его ролями, так чтобы с ним можно было работать, как с обыкновенным пользователем приложения.
`<password-encoder hash="md5" />`	Хранение паролей в зашифрованном виде. В нашем приложении не используется.

11. Заключение

Вот и всё! Осталось прикрутить дизайн, рюшечки на jQuery (статья для начинающих) и продавать поделку за деньги. Но это, дорогие мои читатели, уже совсем другая история.

После выполнения всех манипуляций у вас должна получиться следующая структура проекта:



Буду признателен за указанные ошибки и неточности в статье.

Использованные ресурсы:

1. Spring Recipes: A Problem-Solution Approach, Second Edition, Gary Mak, Ken Sipe, Josh Long, Daniel Rubio.
2. Create Spring 3 MVC Hibernate 3 Example using Maven in Eclipse,
3. The Spring Framework Reference Documentation,
4. Mastering Spring MVC 3 by Keith Donald.

Скачать готовый проект можно здесь, а базу данных для проекта — здесь.

spring framework, spring mvc, spring security, hibernate, многабукаф