SPRING FRAMEWORK IN ECLIPSE

a. It needs to install plugins, **Spring tool suite (STS), Spring IDE** and **Maven integration plugin** for Eclipse.

b. Start a new Maven project in Eclipse.

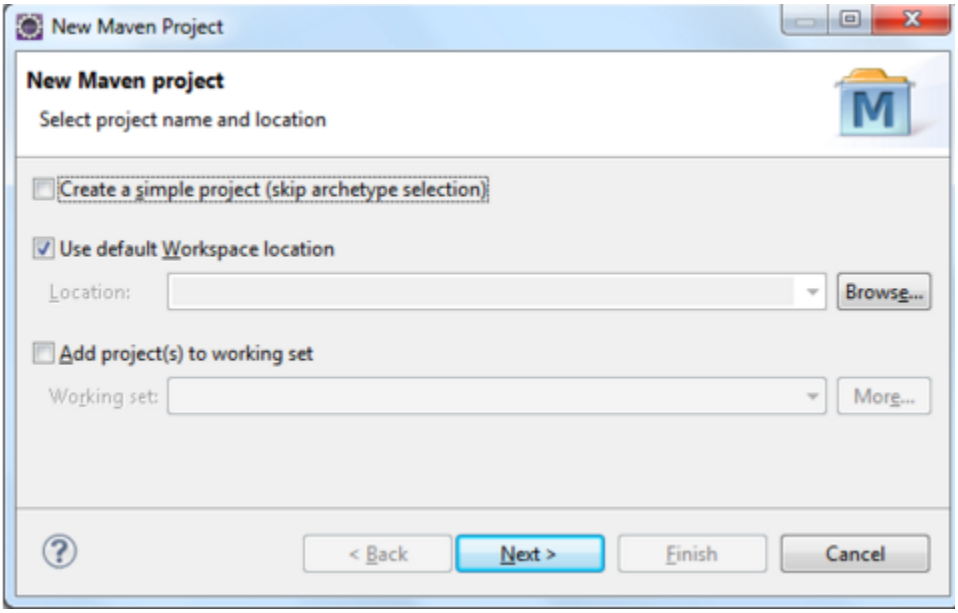


Figure (Step-b): Start a new Maven project

c. Define the Spring MVC archetype in the Maven project.

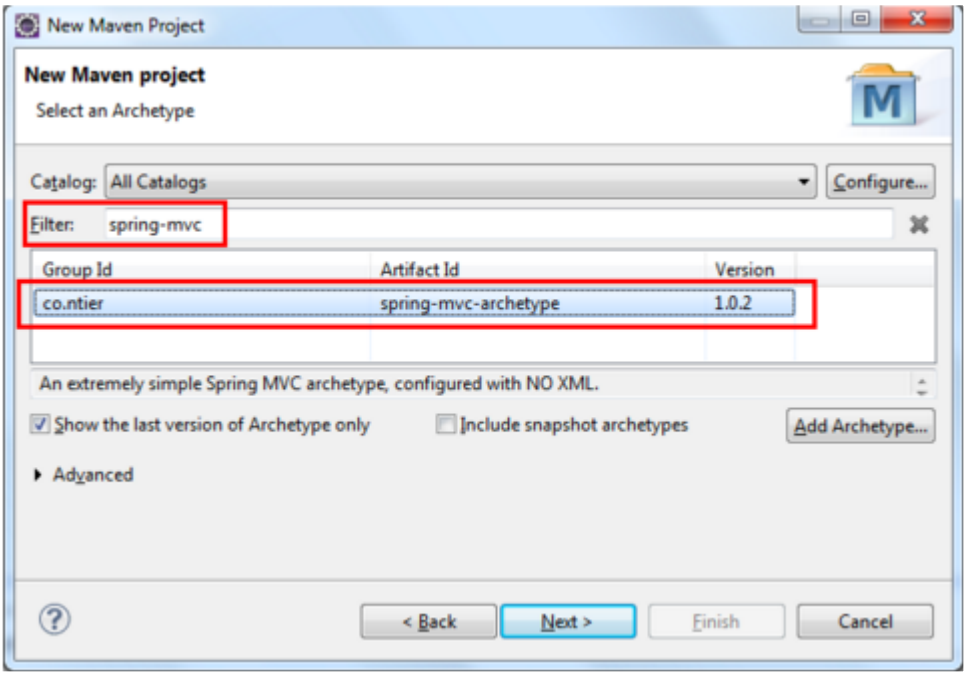


Figure (Step-c): Start a new Maven project

d. If it's not possible to do automatically, configure the archetype manually by proving the required parameters. The required information's are as following,

- Archetype Group Id: **co.ntier**
- Archetype Artifact Id: **spring-mvc-archetype**
- Archetype Version: **1.0.2**
- Repository URL: **<http://maven-repository.com/artifact/co.ntier/spring-mvc-archetype/1.0.2>**

It's also good idea to define maven as `maven-archetype-webapp` and insert the Spring dependencies in the project.

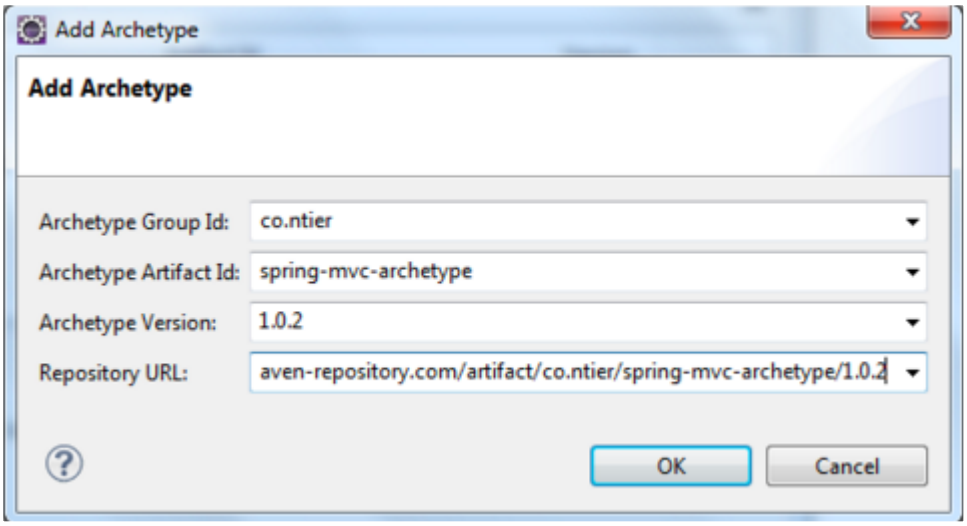


Figure (Step-d): Start a new Maven project

e. Complete generating new Maven project by proving Artifact Id and Group Id.

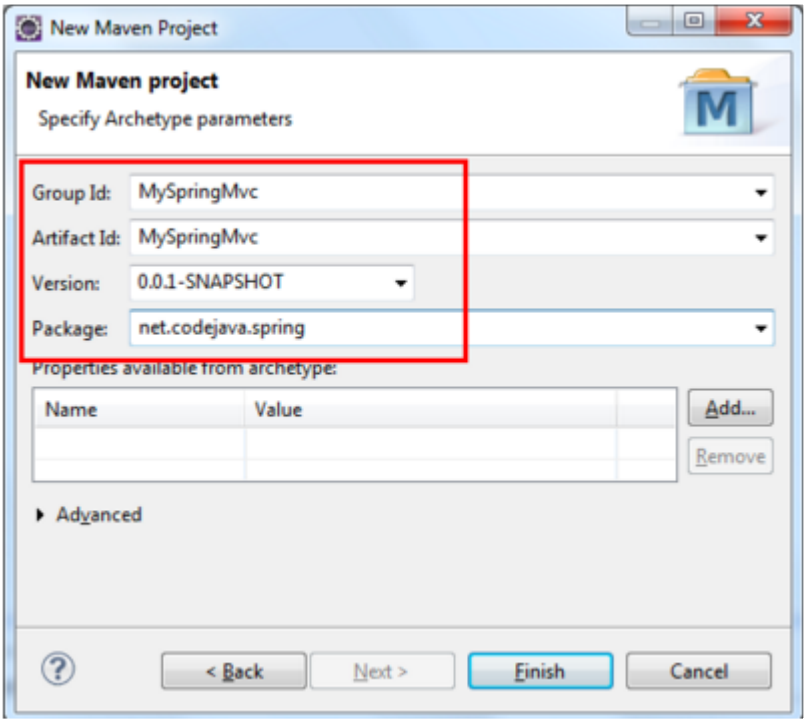


Figure (Step-e): Complete generating new Maven project

SPRING MVC PROJECT WITH STS

New file in STS->Spring project->Spring MVC project->Specify the top level project (say, com.spring.test). The last name of the top level project will be **<artifactId>test</artifactId>** in the **pom.xml** file. We won't need to provide spring-core, spring-beans and spring-webmvc in the **pom.xml** file.

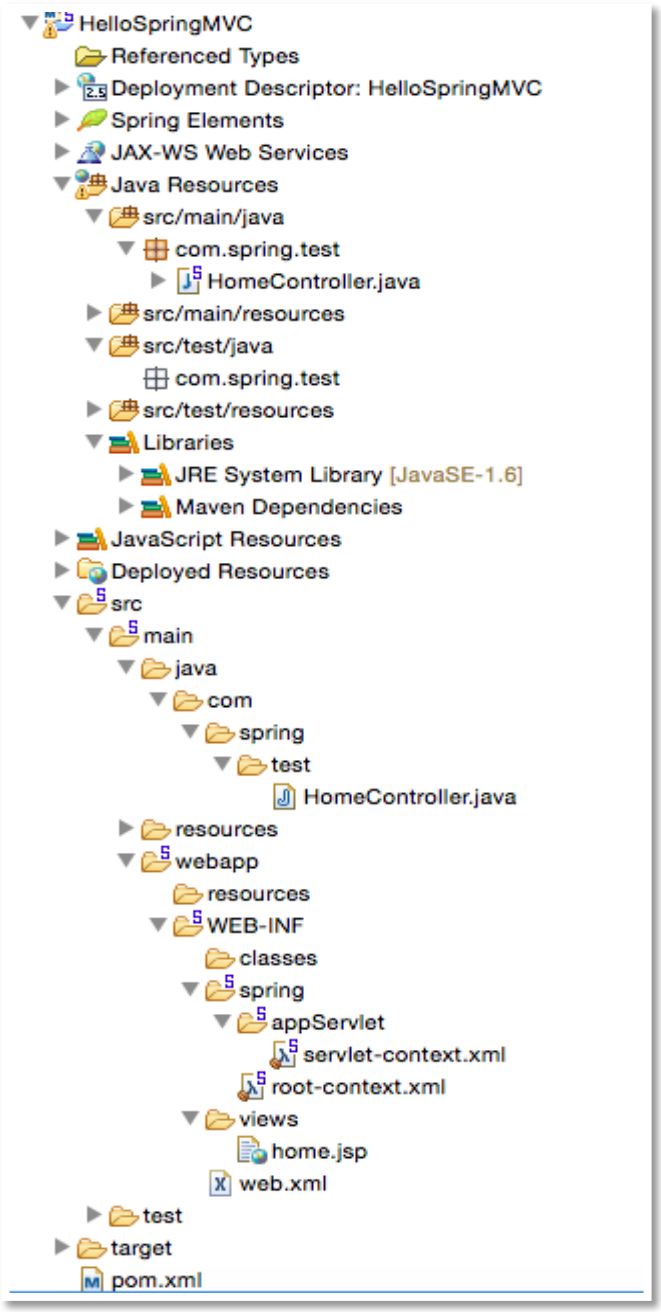


Figure: Spring MVC project with **STS**

web.xml

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/java

    <!-- The definition of the Root Spring Container shared by all Servlets and Filters -
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml</param-value>
    </context-param>

    <!-- Creates the Spring Container shared by all Servlets and Filters -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-c
    </listener>

    <!-- Processes application requests -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Figure: web.xml in STS

servlet-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springfra
        http://www.springframework.org/schema/beans http://www.springframework.org/sch
        http://www.springframework.org/schema/context http://www.springframework.org/s

    <!-- DispatcherServlet Context: defines this servlet's request-processing infrastr

    <!-- Enables the Spring MVC @Controller programming model -->
    <annotation-driven />

    <!-- Handles HTTP GET requests for /resources/** by efficiently serving up static
    <resources mapping="/resources/**" location="/resources/" />

    <!-- Resolves views selected for rendering by @Controllers to .jsp resources in th
    <beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolv
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>

    <context:component-scan base-package="com.spring.test" />

</beans:beans>
```

Figure: servlet-context.xml in STS

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
    <title>Home</title>
</head>
<body>
<h1>
    Hello world!
</h1>

<P>  The time on the server is ${serverTime}. </P>
</body>
</html>
```

Figure: home.jsp in STS

root-context.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="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.springfra

    <!-- Root Context: defines shared resources visible to all other web components -->

</beans>
```

Figure: root-context.xml in STS

THE JAVA SPRING TUTORIAL/ JOHN PURCELL

SEC-1: GETTIG STARTED

SEC-2: BASIC BEAN CONFIGURATION

SEC-3: AUTOWIRING

SEC-4: WIRING WITH ANNOTATIONS

SEC-5: SPRING EXPRESSION LANGUAGE (SPEL)

SEC-6: WORKING WITH DATABASES

SEC-7: WEB APP BASICS WITH SPRING MVC

50. A BASIC NON-SPRING WEB APP

a. File->New->Other->Dynamic web project. Take care to generate web.xml file with the project with checking the box.
b. Add **home.jsp** file in the project. This will be the entry point from the client side.

51. BRINGING IN MAVEN:

a. We need to Springify the project with Maven by, project (right click)->Configure->Convert to Maven project. Need to add group ID and artifact ID with the project. Group ID can be **com.spring.test** and artifact ID as **myTest**. Artifact ID will be the beginning of the URL mapping in the browser.

b. Go to the **pom.xml** file and add the dependencies in the file. The dependencies will be, **spring-web, spring-webmvc, spring-core, spring-context, spring-beans, spring-jdbc** and **mysql-connector-java**. We will also need additional dependencies such as **spring-security-core, spring-security-web, spring-security-config, validation-api** (javax.validation), **hibernate** (org.hibernate) and **hibernate-validator** (org.hibernate)in the project.

c. Update the project after adding the dependencies, Project (right click)-> Maven -> Update project

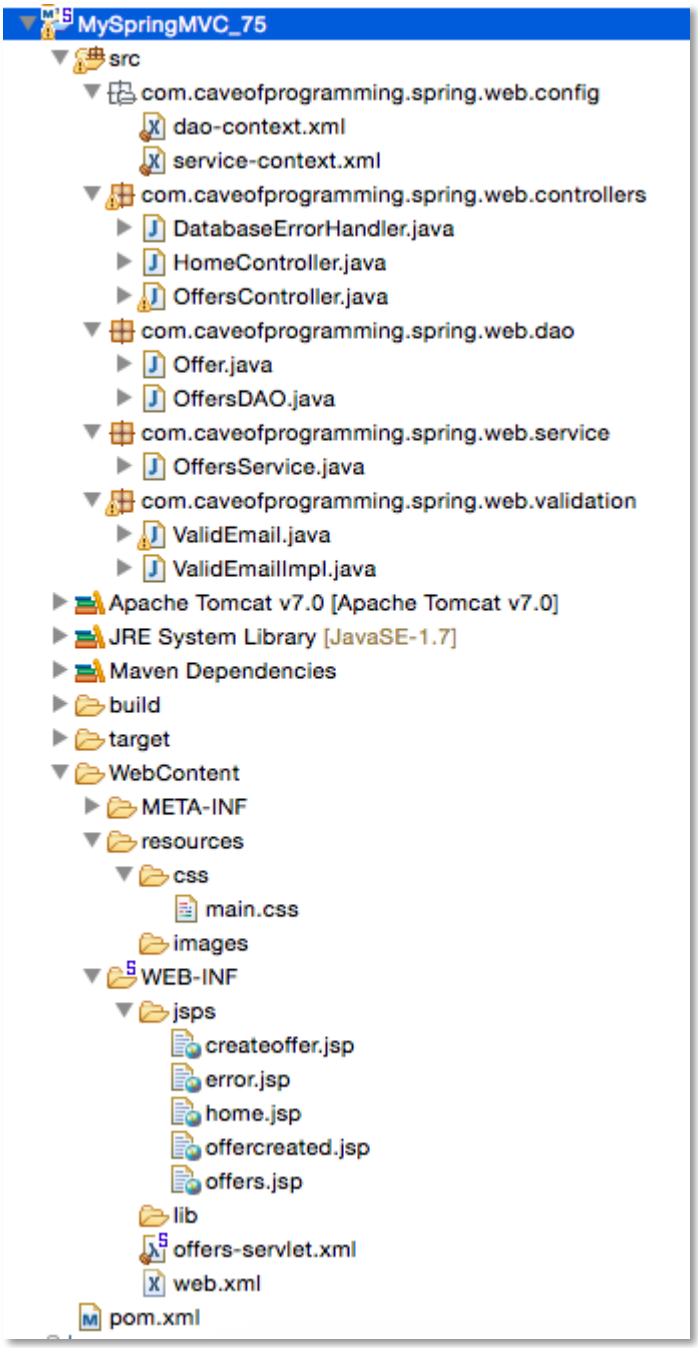


Figure: project structure in **Eclipse**

52. THE DISPATCHER SERVLET:

a. Add a Dispatcher servlet in the **web.xml** file by, Project (right_click)->Servlet->Select class **DispatcherServlet**. The class is available in the **Spring-webmvc** in location **org.springframework.web.servlet.DispatcherServlet.class**. This will add Servlet section in the **web.xml** file as follow,

```
<!--  spring-webmvc from Dispatcher Servlet -->
    <servlet>
        <description></description>
        <display-name>offers</display-name>
        <servlet-name>offers</servlet-name>
<servlet-class> org.springframework.web.servlet.DispatcherServlet
</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>offers</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
<!--  END of spring-webmvc from Dispatcher Servlet -->
```

b. In the **WEB-INF** folder, add a bean configuration file **offers-servlet.xml** from the Spring section. The name should match with,

```
<servlet-name>offers</servlet-name>
```

The Section name previously added, say, **offers-servlet.xml**. No location information is needed as both of them are in the WEB-INF folder.

53. ADDING A CONTROLLER:

a. Make a new package **com.spring.test.controllers** for holding the Java Controller source files and add a new Java class **OfficeControllers.java** with @Controller annotation.

b. Write a method with @RequestMapping annotation and return the **home.jsp** file from the root level using String return value "home". The code should be as following,

```
@RequestMapping("/")
public String showHome() {

    return "home";
}
```

c. Go in the **offers-servlet.xml**-> namespaces-> add namespaces for **mvc** and **context**.

d. Go to the **context** tab in **offers-servlet.xml** file-> right click in the beans->Insert <context-component-scan> element -> put the controller package qualified name in the base-package* section. After that, add **<mvc:annotation-driven>** in the beans in same manner.

```
<context:component-scan base-package="com.spring.test.controllers">
</context:component-scan>
<mvc:annotation-driven></mvc:annotation-driven>
```

54. VIEW RESOLVERS:

a. Make a folder namely JSPs in the WEB-INF folder for containing the **.jsp** files. Put **home.jsp** inside the folder.

b. Add a new bean in the **offers-servlet.xml** file for recognizing the **home.jsp** file as follow, **offers-servlet.xml** -> beans tab-> new beans ->set class and parameters for the file location. The class name is **InternalResourceViewResolver** which is in the **spring-webmvc** JAR.

```
spring-webmvc -> org.springframework.web.servlet.view ->
org.springframework.web.servlet.view.ResourceBundleViewResolver.class
```

The **"prefix"** and **"suffix"** will provide information's about the file location.

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

55. USING THE SPRING DATA MODELS:

a. Write the OfficeController.java with HttpSession as following,

```
@RequestMapping("/")
public String showHome(HttpSession session){

    session.setAttribute( "name", "Boris" );
    return "home";
}
```

By using the HttpSession it's possible to retrieve data in the **home.jsp** file with the following syntax,

```
Session : <%=session.getAttribute("name")%>
```

Another way of writing the Java method,

```
@RequestMapping("/")
public ModelAndView showHome(){

    ModelAndView mv = new ModelAndView("home");
    Map<String,Object> model =  mv.getModel();
    model.put("name", "River");

    return mv;
}
```

Then, it will be possible to retrieve the data in the **.jsp** file as follow,

```
Request: <%=request.getAttribute("name")%>
```

Using Expression language, the syntax in the **.jsp** file will be as follow,

```
<p>Request using EL : ${name}</p>
```

Another way of writing the Java method,

```
@RequestMapping("/")
public String showHome( Model model){

    model.addAttribute( "name", "Natasha Bowels");
    return "home"
}
```

57. USING THE JSTL (JSP STANDARD TAG LIBRARY):

a. Search for JSTL core and get the right tag for JSP file.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Add this tag in the top of the **home.jsp** file. Also, add the JSTL dependency (**javax.servlet jslt**) in the **pom.xml** file. Inside the **home.jsp**, the code will be as follow,

```
<c:forEach var="row" items="${rs.rows}">
    Id : ${row.id}<br />
    Name : ${row.name}<br />
    Email : ${row.email}<br />
    Text : ${row.text}<br />
    <br/>
</c:forEach>
```

58. CONFIGURING THE JNDI DATA SOURCE:

a. Search in the Google for Tomcat 8 MySQL Datasource JNDI. In the page, look for MySQL DPCP example and inside find Context configuration section. This is the sample code for the server (Apache Tomcat) **context.xml** file.

```
<context>
    <Resource name="jdbc/spring" auth="Container" type="javax.sql.DataSource"
              maxTotal="100" maxIdle="30" maxWaitMillis="10000" username="student"
              password="student" driverClassName="com.mysql.jdbc.Driver"
              url="jdbc:mysql://localhost:3306/testDB" />
</context>
```

b. This will require correcting the Database name, user name and password to make it work.

c. Inside the web.xml between <web-app> tag, the following configuration information's need to be inserted. Look for the **web.xml configuration** section in the same article.

```
<description>MySQL Test App</description>
<resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/spring</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

The <res-ref-name> of the **web.xml** file needs to be matched with the <Resource name> as mentioned before.

d. Test the connection with the following SQL and the JSTL code as follow,

```
<sql:query var="rs" dataSource="jdbc/spring">
        select id, name, email, text from offers
</sql:query>

<c:forEach var="row" items="${rs.rows}">
    Id : ${row.id}<br />
    Name : ${row.name}<br />
    Email : ${row.email}<br />
    Text : ${row.text}<br />
    <br/>
</c:forEach>
```

59. BRINGING THE DAO CODE:

a. Create **com.spring.test.dao** package in the source (src) folder. Put Offer.java and OffersDAO.java (from tutorial-49) for interacting with the database.

```java
@Component("offersDao")
public class OffersDAO {

    private NamedParameterJdbcTemplate jdbc;

    @Autowired
    public void setDataSource(DataSource jdbc) {
        this.jdbc = new NamedParameterJdbcTemplate(jdbc);
    }

    public List<Offer> getOffers() {

        return jdbc.query("select * from offers", new RowMapper<Offer>() {

            public Offer mapRow(ResultSet rs, int rowNum) throws SQLException {
                Offer offer = new Offer();

                offer.setId(rs.getInt("id"));
                offer.setName(rs.getString("name"));
                offer.setText(rs.getString("text"));
                offer.setEmail(rs.getString("email"));

                return offer;
            }

        });
    }


    public boolean update(Offer offer) {

        BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(offer);



        return jdbc.update("update offers set name=:name, text=:text, email=:email where id=:id", params) == 1;
    }
    public boolean create(Offer offer) {

        BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(offer);

        return jdbc.update("insert into offers (name, text, email) values (:name, :text, :email)", params) == 1;
    }
```

```java
@Transactional
public int[] create(List<Offer> offers) {

    SqlParameterSource[] params = SqlParameterSourceUtils.createBatch(offers.toArray());

    return jdbc.batchUpdate("insert into offers (id, name, text, email) values (:id, :name, :text, :email)", params);
}

public boolean delete(int id) {
    MapSqlParameterSource params = new MapSqlParameterSource("id", id);

    return jdbc.update("delete from offers where id=:id", params) == 1;
}

public Offer getOffer(int id) {

    MapSqlParameterSource params = new MapSqlParameterSource();
    params.addValue("id", id);

    return jdbc.queryForObject("select * from offers where id=:id", params,
            new RowMapper<Offer>() {

                public Offer mapRow(ResultSet rs, int rowNum)
                        throws SQLException {
                    Offer offer = new Offer();

                    offer.setId(rs.getInt("id"));
                    offer.setName(rs.getString("name"));
                    offer.setText(rs.getString("text"));
                    offer.setEmail(rs.getString("email"));

                    return offer;
                }

            });
}
```

Figure: OffersDAO.java

```java
@Component("offersDAO")
public class OffersDAO {

    private NamedParameterJdbcTemplate jdbc;

    @Autowired
    private void setDataSource(DataSource jdbc ){

        this.jdbc = new NamedParameterJdbcTemplate(jdbc);
    }


    public List<Offer> getOffers(){

        String sql = "SELECT * FROM offers";

        return jdbc.query(sql, new RowMapper<Offer>() {

            @Override
            public Offer mapRow(ResultSet rs, int rowNum) throws SQLException {

                Offer offer = new Offer();

                offer.setId(rs.getInt("id"));
                offer.setName(rs.getString("name"));
                offer.setEmail(rs.getString("email"));
                offer.setText(rs.getString("text"));

                return offer;
            }
        });
    }
```

```java
public boolean create(Offer offer){

    BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(offer);

    String sql = "INSERT INTO offers(name, email, text) VALUES (name=:name, email=:email, text=:text)";
    return jdbc.update(sql, params) == 1;
}
```

```java
public boolean delete(int id ){

    MapSqlParameterSource params = new MapSqlParameterSource("id", id);

    String sql = "DELETE FROM offers WHERE id=:id";
    return jdbc.update(sql, params)==1;
}
```

```
@Transactional
public int[] create(List<Offer> offers){

    SqlParameterSource [] params = SqlParameterSourceUtils.createBatch(offers.toArray());
    String sql  = "INSERT INTO offers (id, name, email, text) VALUES (id=:id, name=:name, email:email, text:text)";
    return jdbc.batchUpdate(sql, params);
}
```

```
@Transactional
public boolean create(User user){

    MapSqlParameterSource params = new MapSqlParameterSource();

    params.addValue("username", user.getUsername());
    params.addValue("email", user.getEmail());
    params.addValue("password", passwordEncoder.encode(user.getPassword()));
    params.addValue("enabled", user.isEnabled());
    params.addValue("authority", user.getAuthority());

    String sql_one  = "insert into users(username, email, password, enabled) values(:username, :email, :password, :enabled)";
    String sql_two = "insert into authorities(username, authority) values (:username, :authority)";

    jdbc.update(sql_one, params);
    return jdbc.update(sql_two, params) ==1 ;
}
```

```
public List<User> getAllUsers(){

    String sql = "select * from users, authorities where users.username = authorities.username";

    return jdbc.query(sql, BeanPropertyRowMapper.newInstance(User.class));
}
```

```
public boolean exists(String username){

    String sql  = "select count(*) from users where username=:username";
    return jdbc.queryForObject(sql, new MapSqlParameterSource("username", username), Integer.class ) > 0;
}
```

b. Create **com.spring.test.config** package in the source folder. Add a Spring bean configuration file **dao-context.xml** in the package. Add a context namespace in the file and put the code to connect with the dao package as following,

```
<context:annotation-config></context:annotation-config>
<context:component-scan base-package="com.spring.test.dao">
</context:component-scan>
```

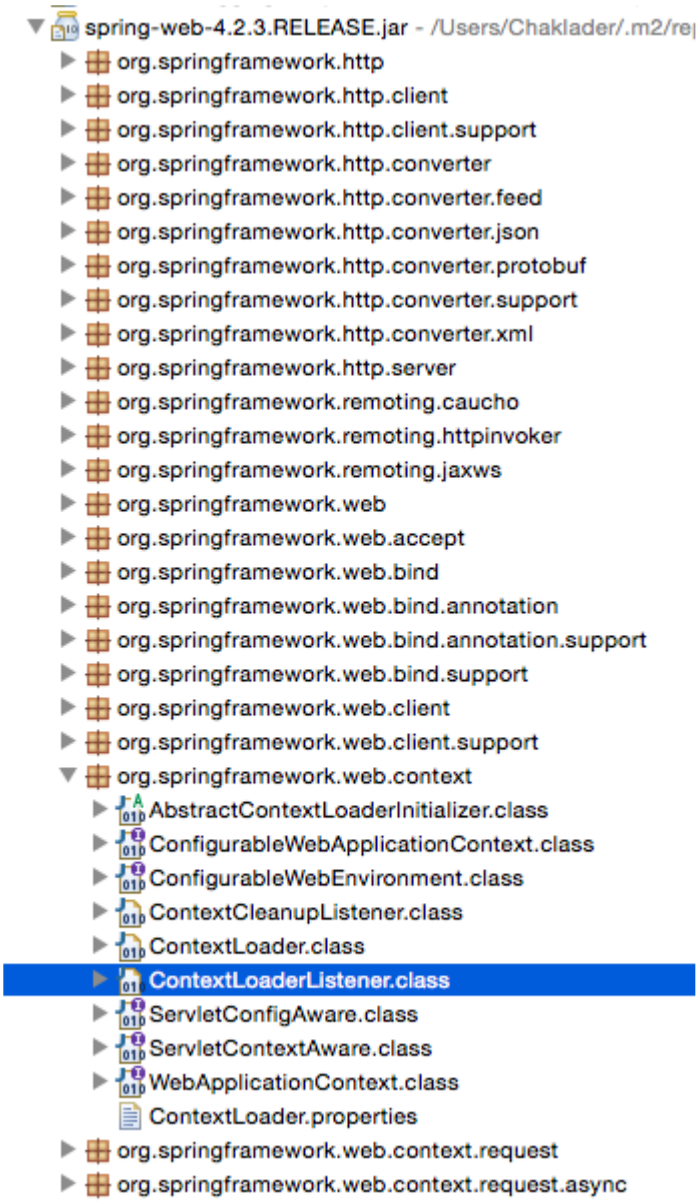
```
<context:annotation-config></context:annotation-config>
<context:component-scan base-package="com.spring.test.dao"></context:component-scan>

<jee:jndi-lookup jndi-name="jdbc/spring" id="dataSource"
    expected-type="javax.sql.DataSource">
</jee:jndi-lookup>

<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<tx:annotation-driven />
```

Figure: dao-context.xml

c. In the **web.xml** file, using **context-listener** to load the extra XML files which are being placed in the config package as follow,

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

The **ContextLoaderListener.class** is inside the **spring-web** JAR file in the Maven dependencies.



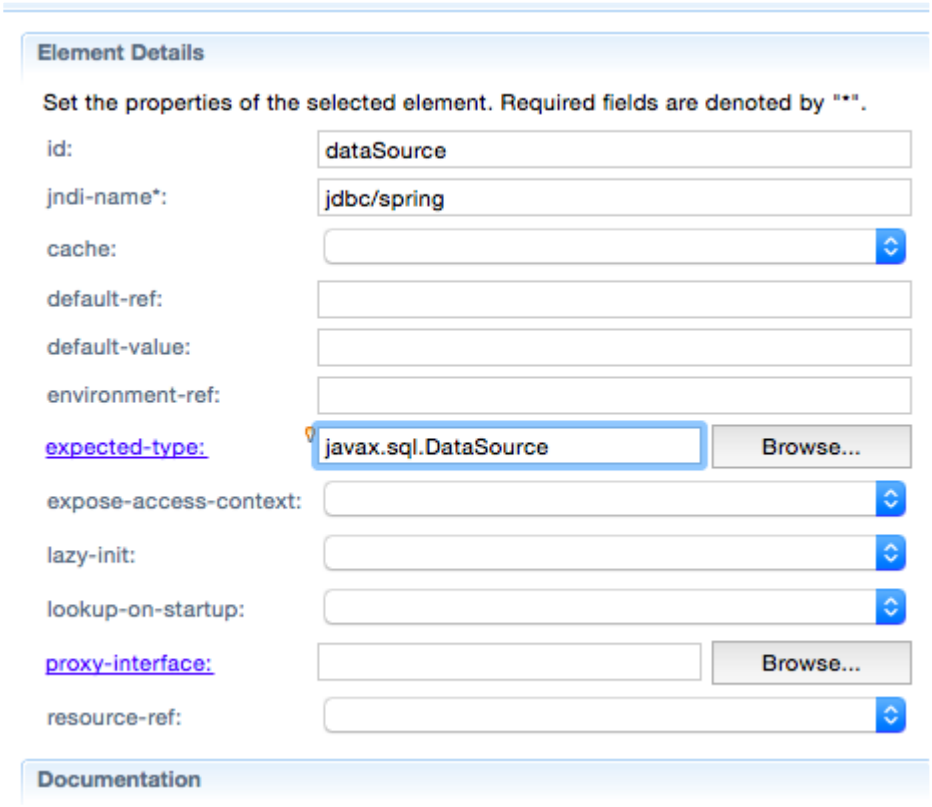
d. Now, load the XML files in the project using **web.xml** file providing the classpath of the respective files as follow,

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
    classpath:com/spring/test/config/dao-context.xml
    classpath:com/spring/test/config/service-context.xml
    </param-value>
</context-param>
```

61. CREATING A DATASOURCE BEAN:

a. In the **dao-context.xml** file, enable the **JEE** namespace, click the JEE tab, insert jndi-lookup and format according to the following,

```
<jee:jndi-lookup jndi-name="jdbc/spring" id="dataSource"
        expected-type="javax.sql.DataSource">
</jee:jndi-lookup>
```



The **jndi-name** needs to be matched with the **<Resource name>** in the **context.xml** file of the Apache Tomcat.

62. ADDING A SERVICE LAYER:

a. Create a new package **com.spring.test.service** and add a new class named OffersService.java there. Put @Service("OffersService") annotation top of the class.

b. Add a bean configuration file **service-context.xml** file in the config package. Enable context tab in from the namespace and add the following XMl there -

```
<context:annotation-config></context:annotation-config>
<context:component-scan base-package="com.spring.test.service">
</context:component-scan>
```

c. Add the classpath of the **service-context.xml** in the **web.xml** file.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
classpath:com/caveofprogramming/spring/web/config/dao-context.xml
classpath:com/caveofprogramming/spring/web/config/service-context.xml
</param-value>
</context-param>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframewor
        http://www.springframework.org/schema/context http://www.springframework.org/schema/c

    <context:annotation-config></context:annotation-config>
    <context:component-scan base-package="com.spring.test.service"></context:component-scan>
</beans>
```

Figure: service-context.xml

63. ADDING A NEW CONTROLLER:

a. Add a new source file **HomeController.java** in the controller package.
b. Add new **.jsp** file **offers.jsp** to link with the **home.jsp** as follow,

```
<p>
    <a href="${pageContext.request.contextPath}/offers">Show current offers.
    </a>
</p>
<p>
    <a href="${pageContext.request.contextPath}/createoffer">Add a new offer.
    </a>
</p>
```

64. GETTING THE URL PARAMETERS:

a. Add a get method in the source file as follow,

```
@RequestMapping(value="/test", method=RequestMethod.GET)
public String showTest(Model model, @RequestParam("id") String id) {

    System.out.println("Id is: " + id);
    return "home";
}
```

web.xml

```
<servlet>
  <description></description>
  <display-name>offers</display-name>
  <servlet-name>offers</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>offers</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
<description>Spring Database</description>
<resource-ref>
  <description>DB Connection</description>
  <res-ref-name>jdbc/spring</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
      classpath:com/caveofprogramming/spring/web/config/dao-context.xml
      classpath:com/caveofprogramming/spring/web/config/service-context.xml
      </param-value>
</context-param>

</web-app>
```

Figure: **web.xml** in the **WEB-INF** folder (ROOT level)

offers-servlet.xml

```
<?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:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.spring
        http://www.springframework.org/schema/beans http://www.springframework.org/
        http://www.springframework.org/schema/context http://www.springframework.or

    <context:component-scan base-package="com.caveofprogramming.spring.web.controll
    </context:component-scan>
    <mvc:annotation-driven></mvc:annotation-driven>

    <bean id="jspViewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsps/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

    <!-- for the CSS design files  -->
    <mvc:resources location="/resources/" mapping="/static/**" />

</beans>
```

Figure: **offers-servlet.xml** in the **WEB-INF** folder (ROOT level)

dao-context.xml

```xml
<?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:co
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/beans http://www.spr
        http://www.springframework.org/schema/context http://www.s

    <context:annotation-config></context:annotation-config>
    <context:component-scan
        base-package="com.caveofprogramming.spring.web.dao">
    </context:component-scan>

    <jee:jndi-lookup jndi-name="jdbc/spring" id="dataSource"
        expected-type="javax.sql.DataSource">
    </jee:jndi-lookup>
</beans>
```

Figure: **dao-context.xml** in **config** package

service-context.xml

```xml
<?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:context="http://www.sp
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springf
        http://www.springframework.org/schema/context http://www.springframework.org/s

    <context:annotation-config></context:annotation-config>
    <context:component-scan base-package="com.caveofprogramming.spring.web.service">
    </context:component-scan>
</beans>
```

Figure: **service-context.xml** in **config** package

home.jsp

directive.page | http://java.sun.com/jsp/jstl/core | Palette

```
1
2  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
3      pageEncoding="ISO-8859-1"%>
4  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
5
6  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/h
7  <html>
8  <head>
9  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10 <title>Insert title here</title>
11 </head>
12 <body>
13
14 <p><a href="${pageContext.request.contextPath}/offers">Show current offers.</a></p>
15 <p><a href="${pageContext.request.contextPath}/createoffer">Add a new offer.</a></p>
16
17 </body>
18 </html>
```

Figure: home.jsp in Eclipse

SEC - 8: WORKING WITH WEB FORMS

65. CREATING A FORM

a. Add new method in the **officeController.java** source file and also, create a new JSP file in the respective name. Started working with web forms in the **createoffer.jsp** file.

```
@RequestMapping("/createoffer")
public String createOffer(Model model) {

    model.addAttribute("offer", new Offer());
    return "createoffer";
}
```

66. GETTING FORM VALUES

a. Use the **<form>** tag in the **createoffer.jsp** files to get the form values.

```
<sf:form method="post"
    action="${pageContext.request.contextPath}/docreate"
    commandName="offer">
```

67. ADDING CSS STYLES

a. Put following code in the **offers-servlet.xml** file for locating the **.css** files to the project.

```
<!-- for the CSS design files -->
<mvc:resources location="/resources/" mapping="/static/**" />
```

b. Use the **main.css** file in the **.jsp** files (including **createoffer.jsp**) using the following import in the **<head>** of the file.

```
<head>
    <link href="${pageContext.request.contextPath}/static/css/main.css"
        rel="stylesheet" type="text/css" />
</head>
```

68. SERVING STATIC RESOURCES

a. Add a CSS file for designing the **.jsp** files as follow, WebContent->resources->css-> main.css
b. In the **offers-servlet.xml**, go in the mvc TAB and provide the information for location and mapping.

<mvc:resources location="/resources/" mapping="/static/**" />

** goes for the wild card and * goes for everything inside that particular folder.

c. Need to define the location of the CSS file in the top of the **createoffer.jsp** file.

<link href="${pageContext.request.contextPath}/static/css/main.css" rel="stylesheet" type="text/css" />

69. ADDING HIBERNATE FORM VALIDATION SUPPORT:

a. In the **OfficeController.java** source file, insert a @Valid annotation.

```
@RequestMapping(value="/docreate",method=RequestMethod.POST)
public String doCreate(Model model, @Valid Offer offer, BindingResult result ) {

    if(result.hasErrors()) {

            // return "createoffer";
        System.out.println(" The form is not validated ");
        List<ObjectError> errors = result.getAllErrors();

        for( ObjectError er: errors ){
            System.out.println("The errors : "+ er);
        }
    }

    else  {
        System.out.println("The form is validated");
    }

    // offersService.create(offer);
    System.out.println(offer);
    return "offercreated";
}
```

```
@RequestMapping(value = "/docreate", method = RequestMethod.POST)
public String doCreate(Model mode, BindingResult result, @Valid Offer offer) {

    if (result.hasErrors()){

        return "createoffer";
    }

    offersService.create(offer);

    return "offercreated";
}
```

b. Add necessary dependencies in the **pom.xml** file for validation and hibernate.

```xml
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.5.4-Final</version>
    <type>pom</type>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.1.Final</version>
</dependency>
```

70. MORE FORM VALIDATION TAGS

a. Add more validation tags in the **Offer.java** class (say, **@Size, @NotNull**) with necessary import.
b. Add a new package **com.caveofprogramming.spring.web.validation** and create 2 class file namely **ValidEmailImpl.java** and **ValidEmail.java**

there for writing custom validation feature.

c. Search for **javax.validation** constraints for learning about the tags and start writing custom validation feature.

```java
@Target({ METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = com.spring.test.validation.ValidEmailImpl.class)
public @interface ValidEmail {

    String message() default "This doesn't appear to be a valid email address";
    Class<?>[] groups() default{};
    Class<? extends Payload>[] payLoad() default{};
    int min() default 5;
}
```

figure: ValidEmail.java

```java
public class ValidEmailImpl implements ConstraintValidator<ValidEmail, String>{

    private int min;

    @Override
    public void initialize(ValidEmail constraintAnnotation) {

        min = constraintAnnotation.min();
    }

    @Override
    public boolean isValid(String email, ConstraintValidatorContext context) {

        if (email.length() < 5){
            return false;
        }

        if ( !EmailValidator.getInstance(false).isValid(email) ){

            return false;
        }

        return true;
    }
}
```

figure: ValidEmailImpl.java

71. MAKING THE FORM REMEMBER VALUES

a. If the submission is wrong, the form can't remember the previous values after refreshing and hence, we will need to write code for the **.jsp** files to remember the previous inserted values. Search Spring form tag for remembering values and get the following,

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

b. Add <sf:form> and path parameter as follow,

```
<sf:form method="post"
    action="${pageContext.request.contextPath}/docreate"
    commandName="offer">

    <table class="formtable">

        <tr>
            <td class="label">Name:</td>
            <td><sf:input class="control" path="name" name="name"
                    type="text" /> <sf:errors path="name" cssClass="error"></sf:errors></td>
        </tr>

        <tr>
            <td class="label">Email:</td>
            <td><sf:input path="email" class="control" name="email"
                    type="text" /> <sf:errors cssClass="error" path="email"></sf:errors>
            </td>
        </tr>

        <tr>
            <td class="label">Your offer:</td>
            <td><sf:textarea class="control" path="text" name="text"
                    rows="10" cols="10" /> <sf:errors cssClass="error" path="text"></sf:errors></td>
        </tr>

        <tr>
            <td class="label"></td>
            <td><input class="control" type="submit" value="Create Advert"></td>
        </tr>

    </table>

</sf:form>
```

c. commandName = **"offer"** in the **createoffer.jsp** needs to be matched with the respective attribute in the OfficeController.java file as follow,

```
@RequestMapping("/createoffer")
public String createOffer(Model model) {

    model.addAttribute("offer", new Offer());
    return "createoffer";
}
```

72. DISPLAYING THE FORM VALIDATION ERRORS

a. Add <sf:errors> tag in the **createoffer.jsp** file for displaying errors.
b. Add **.error** class in the main.css file as follow,

```
.error {

    font-size: small;
```

```
            color: red;
            margin-left: 10px;
        }
```

This will display the errors from **Offer.java** file if the submission is not valid. The message will come from the source method as follow,

```
@Size(min=5, max=100, message="Text must be between 20 and 255 characters.")
private String text;
```

74. CREATING A CUSTOM VALIDATION ANNOTATION:

a. Search custom validation constraints Java or right click on a validation annotation (say, @Size) and search for open declaration.
b. Create a package named **com.caveofprogramming.spring.web.validation** and put 2 source file **ValidEmailImpl.java** and **ValidEmail.java** there for creating customized annotation **@ValidEmail**

c. Add new dependency **commons-validator** in the **pom.xml** file,

```
<dependency>
    <groupId>commons-validator</groupId>
    <artifactId>commons-validator</artifactId>
    <version>1.4.0</version>
</dependency>
```

d. Import the class org.apache.commons.validator.routines.EmailValidator in the **ValidEmailImpl.java** source file. Respective method in the source file will as follow,

```java
@Override
public boolean isValid(String email, ConstraintValidatorContext context) {

    if(email.length() < min) {
        return false;
    }

    if(!EmailValidator.getInstance(false).isValid(email)){
        return false;
    }

    return true;
}
```

```java
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.apache.commons.validator.routines.EmailValidator;;

public class ValidEmailImpl implements ConstraintValidator<ValidEmail, String>{

    private int min;

    @Override
    public void initialize(ValidEmail constraintAnnotation) {

        min = constraintAnnotation.min();
    }

    @Override
    public boolean isValid(String email, ConstraintValidatorContext context) {

        if (email.length() < 5){
            return false;
        }

        if ( !EmailValidator.getInstance(false).isValid(email) ){

            return false;
        }

        return true;
    }
}
```

74. HOOKING UP THE CONTROLLER AND THE DATABASE CODE

a. Add a create method in the **OffersService.java** source file in the service package.
b. In the **offercreated.jsp** file, add the code as following,

```
Offer created:
<a href="${pageContext.request.contextPath}/offers">
    click here to view current offers.
</a>
```

75. EXCEPTION HANDLING IN SPRING MVC

a. Add **throwTestException** method in the service package and use inside the **OffersController.java** file. This should return **"error"** String that will return **error.jsp** file from the JSPs folder.

b. The better way to do that is to write a new class **DatabaseErrorHandler.java** in the controllers package and add the code as follow,

```
@ControllerAdvice
public class DatabaseErrorHandler {

    @ExceptionHandler(DataAccessException.class)
    public String handleDatabaseException(DataAccessException ex) {

        return "error";
    }
}
```

```
@ControllerAdvice
public class ErrorHandler {

    @ExceptionHandler(DataAccessException.class)
    public String handleDatabaseException( DataAccessException ex ){

        ex.printStackTrace();
        return "error";
    }

    @ExceptionHandler(AccessDeniedException.class)
    public String handleAccessException(AccessDeniedException ex){

        return "denied";
    }

}
```

SEC - 10: SPRING SECURITY AND MANAGING USERS

92. SERVLETS FILTERS: A REVIEW

a. Add a filter using **other->web->filter**, name the class **TestFiler,** make a **filer** package and put it inside.

b. The filter section will be updated inside the **web.xml** as following,

```
<filter>
    <display-name>TestFilter</display-name>
    <filter-name>TestFilter</filter-name>
    <filter-class>com.spring.web.filter.TestFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>TestFilter</filter-name>
    <url-pattern>/</url-pattern>
</filter-mapping>
```

Mark **<url-pattern>** as **/*** so it can access all the files inside the folder.

c. Print URL info inside the **doFilter** method as following,

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

    System.out.println(((HttpServletRequest) request).getRequestURL());

    // pass the request along the filter chain
    chain.doFilter(request, response);
}
```

93. ADDING A SPRING SECURITY FILTER

a. Delete the **TestFilter.java** class and associated portion inside the **web.xml** file.

b. Add 4 dependencies inside the **pom.xml** l) **spring-security-core** ll) **spring-security-web** lll) **spring-security-config** lV) **spring-security-taglib**

c. Add a new filter class. Go in the **new->other->web->filter->use existing filter class** -> **DelegatingFilterProxy** -> name the new class as **springSecurityFilerChain** -> finish

d. Edit inside the **web.xml** for **<url-pattern>** as following,

```
<filter>
  <display-name>springSecurityFilterChain</display-name>
  <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>
```

e. Inside the **config** package, add a new **bean configuration** file namely **security-context.xml** and **activate** the **security namespace (sec)** there. If the security namespace is not there, search in the google for spring security namespace.

f. Add the **security-context.xml** inside the **web.xml** file as following,

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:com/spring/web/config/dao-context.xml
        classpath:com/spring/web/config/service-context.xml
        classpath:com/spring/web/config/security-context.xml
    </param-value>
</context-param>
```

94. ADDING A SPRING LOGIN FORM

a. In the **security-context.xml** file, go in the security namespace, **sec** and add **<security: authentication-manager>**
b. Inside the security authentication manager, add **<security: authentication:provider>**
c. Inside authentication provider, insert **<security:user-service>** element.
d. Inside the user-service, insert **<security:user>** element.
e. Fill the generated form with the informations with authorities, name and password.
f. Go in the **bean** again and insert **<security:http>** element
g. Inside the **http** section, insert **<security:intercept-url>** element
h. Click on the **http** and make the **use-expressions** as **true.**
i. Click on the **intercept-url** and define **pattern*** as **/**** and **access** as **denyAll**
j. Right click on the **http** and insert **<security:form-login>** element

In last the whole section is like as following,

```
<security:authentication-manager>
    <security:authentication-provider>
        <security:user-service>
            <security:user name="john" authorities="admin"
                password="seattle" />
        </security:user-service>
    </security:authentication-provider>
</security:authentication-manager>
<security:http use-expressions="true">
    <security:intercept-url pattern="/**" access="denyAll" />
    <security:form-login />
</security:http>
```

95. SERVING STATIC RESOURCES: ACCESS RULES

a. Provide the static resources information as following,

```
<mvc:resources location="/resources/" mapping="/static/**" />
```

96. CUSTOMIZING THE LOGIN FORM

a. Create login form of my own by taking the custom login form in browser and see the page source. Copy the page source.

b. Create a new **login.jsp** file inside the JSP folder and paste the source there.

c. In the **login.jsp** file, change the form value and put,

```
<form name='f'
    action='${pageContext.request.contextPath}/j_spring_security_check'
    method='POST'>
```

d. Add the stylesheet top of the page,

```
<head>
    <title>Login Page</title>
    <link href="${pageContext.request.contextPath}/static/css/main.css"
        rel="stylesheet" type="text/css" />
</head>
```

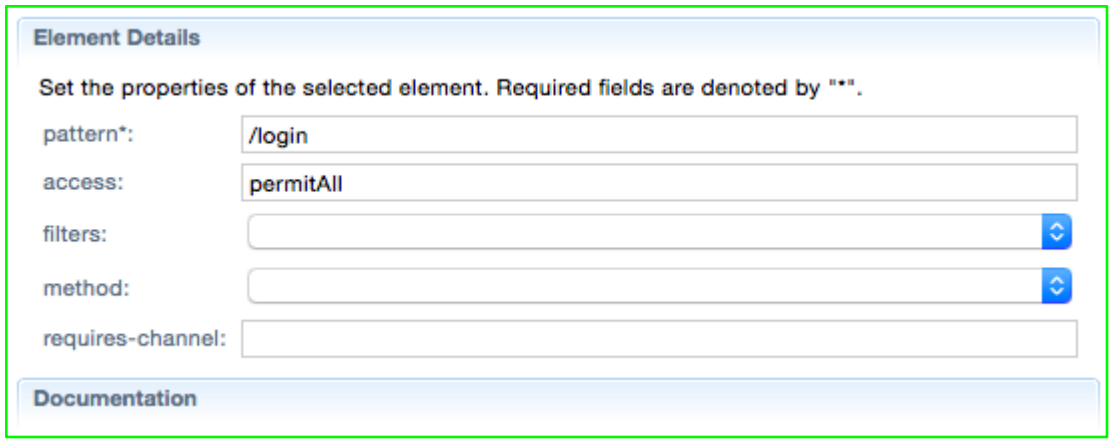
e. Table starts with class **<table class="formtable">**

f. Add **LoginController.class** in the Controller package. Annotate the class with **@Controller** and start to write the methods.

g. Write a new method **showLogin()** that maps with the **login.jsp** file as following,

```
@RequestMapping("/login")
public String showLogin() {

    return "login";
}
```

h. Add **<security:intercept-url pattern="/login" access="permitAll" />** inside the **<security:http>** tag of **security-context.xml** file.



i. Inside the **sec** section of the **security-context.xml,** go in the **form-login** and provide the values for **login-page:** as **/login**

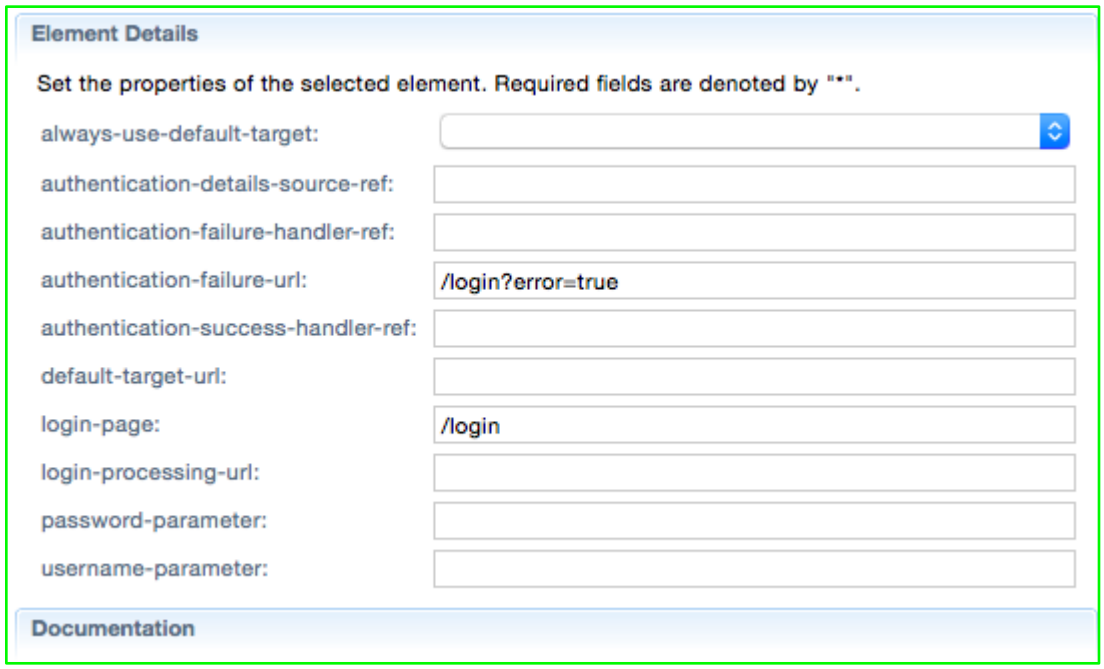


Figure:

j. The final version of the **<security:http>** is as following,

```
<security:http use-expressions="true">
    <security:intercept-url pattern="/admin" access="hasRole('ROLE_ADMIN')" />
    <security:intercept-url pattern="/createoffer" access="isAuthenticated()" />
    <security:intercept-url pattern="/docreate" access="isAuthenticated()" />
    <security:intercept-url pattern="/offercreated" access="isAuthenticated()" />
    <security:intercept-url pattern="/" access="permitAll" />
    <security:intercept-url pattern="/denied" access="permitAll" />
    <security:intercept-url pattern="/loggedout" access="permitAll" />
    <security:intercept-url pattern="/newaccount" access="permitAll" />
    <security:intercept-url pattern="/createaccount" access="permitAll" />
    <security:intercept-url pattern="/accountcreated" access="permitAll" />
    <security:intercept-url pattern="/static/**" access="permitAll" />
    <security:intercept-url pattern="/login" access="permitAll" />
    <security:intercept-url pattern="/offers" access="permitAll" />
    <security:intercept-url pattern="/**" access="denyAll" />
    <security:form-login login-page="/login"
        authentication-failure-url="/login?error=true" />
    <security:logout logout-success-url="/loggedout" />
    <security:access-denied-handler error-page="/denied" />
    <security:remember-me key="offersAppKey"
        user-service-ref="jdbcUserService" />
</security:http>
```

97. DISPLAYING THE LOGIN ERRORS

a. Add **authentication-failure-url:** as **/login?error=true** in the **form-login** of the **security-context.xml** In total it looks like the following,

```
<security:form-login login-page="/login"
    authentication-failure-url="/login?error=true" />
```

b. Use **JSLT** to show the error message. Core of the JSTl needs to be added on the top of the **login.jsp** as following,

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:if test="${param.error != null}">
    <p class="error">Login failed. Check that your username and password are correct.</p>
</c:if>
```

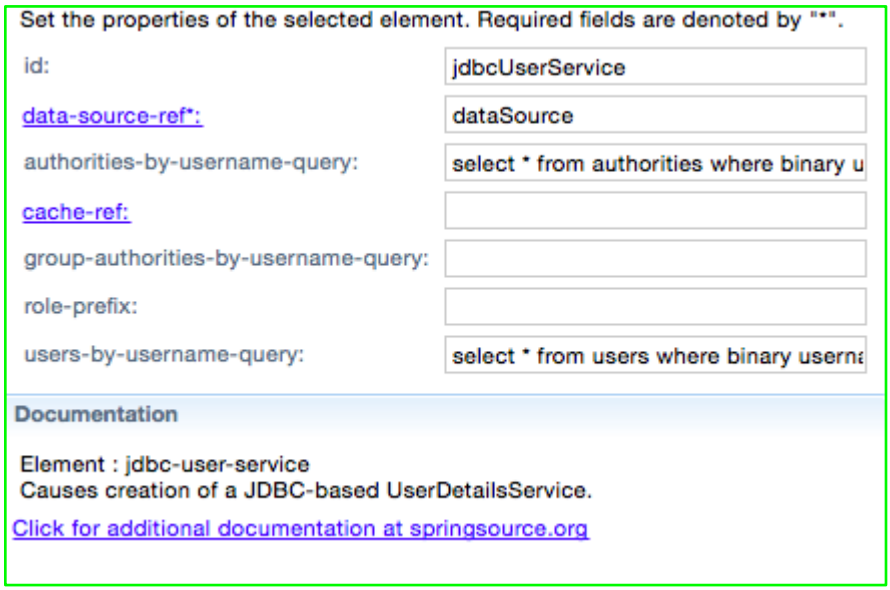
```
<c:if test="${param.error != null}">
    <p class="error">The login is failed. The USERNAME // PASSWORD
        provided is not correct.</p>
</c:if>
```

Error is defined in the **security-context.xml** file as following,

```
authentication-failure-url="/login?error=true"
```

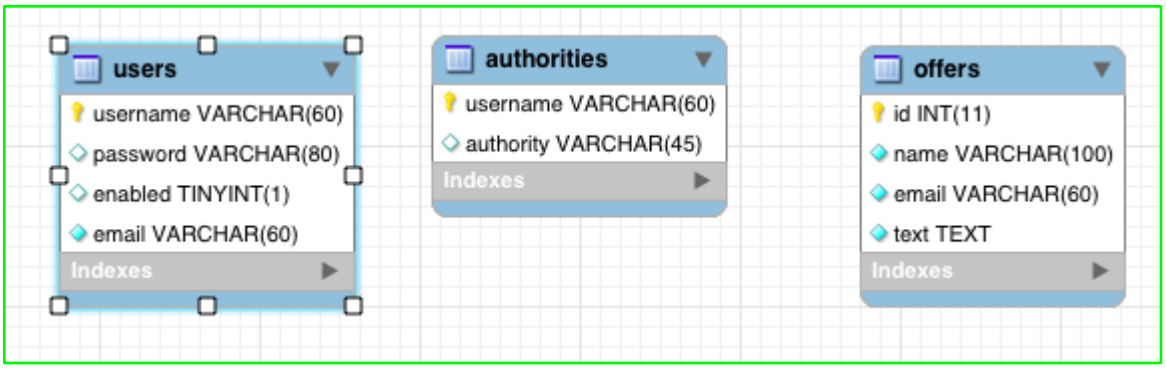
98. AUTHORIZING THE USERS FROM A DATABASE

a. **security-context.xml -> beans -> authentication-manager -> <security:authentication-provider>**
b. Right click on the **2nd** authentication provider -> **<security:jdbc-user-service>** element and fill it as following,




```
<security:jdbc-user-service data-source-ref="dataSource"
    authorities-by-username-query='select * from authorities where binary username = ?'
    users-by-username-query='select * from users where binary username = ?'
    id="jdbcUserService" />
```

c. Inside the **testDB** database, create 2 new tables namely **users** and **authorities**.



99. ADDING A CREATE ACCOUNT FORM

a. Add **newaccount.jsp** file in the JSP folder and put the table there.

```
<sf:form id="details" method="post"
    action="${pageContext.request.contextPath}/createaccount"
    commandName="user">

    <table class="formtable">
        <tr>
            <td class="label">User Name :</td>
            <td><sf:input class="control" path="username" name="username"
                    type="text" /></br>
                <div class="error">
                    <sf:errors path="username"></sf:errors>
                </div></td>
        </tr>

        <tr>
            <td class="label">Email :</td>
            <td><sf:input class="control" path="email" name="email"
                    type="text" /></br>
                <div class="error">
                    <sf:errors path="email"></sf:errors>
                </div></td>
        </tr>
        <tr>
            <td class="label">Password :</td>
            <td><sf:input id="password" class="control" path="password"
                    name="password" type="password" /></br>
                <div class="error">
                    <sf:errors path="password"></sf:errors>
                </div></td>
        </tr>
        <tr>
            <td class="label">Confirm password :</td>
            <td><input id="confirmpass" class="control" name="confirmpass"
                type="password">
                <div id="matchpass"></div></td>
        </tr>
        <tr>
            <td class="label"></td>
            <td><input class="control" value="Create Account" type="submit">
            </td>
        </tr>
    </table>
</sf:form>
```

c. Go inside the **LoginController.java** class and write the method **showNewAccount** there as following,

```
@RequestMapping("/newaccount")
public String showNewAccount(Model model) {

    model.addAttribute("user", new User());
    return "newaccount";
}
```

d. Write another method **createAccount** in the same class as following,

```
@RequestMapping( value = "/createaccount", method = RequestMethod.POST )
public String createAccount(@Valid User user, BindingResult result){

    if (result.hasErrors()){

        return "newaccount";
    }

    user.setAuthority("ROLE_USER");
    user.setEnabled(true);

    if ( usersService.exists( user.getUsername()) ){

        result.rejectValue("username", "DuplicateKey.user.username");
        return "newaccount";
    }

    try{

        usersService.create(user);
    }

    catch(DuplicateKeyException ex){

        result.rejectValue("username", "DuplicateKey.user.username");
        return "newaccount";
    }

    return "accountcreated";
}
```

e. Create file **accountcreated.jsp** inside the JSP folder.

f. Create **User.java** class inside the dao package with the follwoing initials,

@NotBlank

```
    @Size(min=8, max=15)
    @Pattern(regexp="^\\w{8,}$")
    private String username;

    @NotBlank
    @Pattern(regexp="^\\S+$")
    @Size(min=8, max=15)
    private String password;

    @ValidEmail
    private String email;

    private boolean enabled = false;
    private String authority;

    public User() {

    }
```

```
@NotBlank
@Size(min = 8, max = 15 )
@Pattern(regexp = "^\\w{8,}$")
private String username;

@NotBlank
@Pattern(regexp = "^\\S+$")
@Size(min = 8, max = 15 )
private String password;

// @ValidEmail
private String email;

private String authority;
private boolean enabled =  false;

public User(){

}
```

figure:

g. Allow the new JSP's accessible in the **security-context.xml** by defining inside the **<security:http>** as following,

```
<security:http>
    <security:intercept-url pattern="/newaccount" access="permitAll" />
</security:http>
```

h. In the end of the **login.jsp** page, put the code as following,

```
<p>
    <a href="<c:url value="/newaccount"/>">Create new account</a>
</p>
```

100. MAKING THE CREATE ACCOUNT FORM WORK

a. Create **UserDao.java** inside the dao package and set the data source insde the file as following,

```
private NamedParameterJdbcTemplate jdbc;

@Autowired
public void setDataSource(DataSource jdbc) {

    this.jdbc = new NamedParameterJdbcTemplate(jdbc);
}
```

b. Add a **create** method and put the following code,

```
@Transactional
public boolean create(User user){

    MapSqlParameterSource params = new MapSqlParameterSource();

    params.addValue("username", user.getUsername());
    params.addValue("email", user.getEmail());
    params.addValue("password", passwordEncoder.encode(user.getPassword()));
    params.addValue("enabled", user.isEnabled());
    params.addValue("authority", user.getAuthority());

    String sql_one  = "insert into users(username, email, password, enabled) values(:username, :email, :password, :enabled)";
    String sql_two = "insert into authorities(username, authority) values (:username, :authority)";

    jdbc.update(sql_one, params);
    return jdbc.update(sql_two, params) ==1 ;
}
```

c. Create **UsersService.java** class inside the service package and put the code for connecting the **service** package with **dao** package as following,

```java
@Service("usersService")
public class UsersService {

    private UsersDao usersDao;

    @Autowired
    public void setOffersDao(UsersDao usersDao) {
        this.usersDao = usersDao;
    }

    public void create(User user) {
        usersDao.create(user);
    }

    public boolean exists(String username) {
        return usersDao.exists(username);
    }

    @Secured("ROLE_ADMIN")
    public List<User> getAllUsers() {
        return usersDao.getAllUsers();
    }
}
```

101. ADDING VALIDATION TO THE USER FORM

A. Add some validation in the top of the **User.java** class as following,

```
@NotBlank ( message = "the username can't be blank")
@Size(min=8, max=15)
@Pattern(regexp="^\\w{8,}$", message = "some msg will be showed.")
private String username;

@NotBlank ( message = "some msg")
@Pattern(regexp="^\\S+$")
@Size(min=8, max=15)
private String password;

@ValidEmail
private String email;
```

```
        private boolean enabled = false;
        private String authority;
```

We also need to do proper import,

```
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotBlank;
import com.caveofprogramming.spring.web.validation.ValidEmail;
```

102. DEALING WITH DUPLICATE USERNAMES

a. To handle the duplicate usernames, use **try/catch** block inside the **LoginController.java** file as following,

```
@RequestMapping( value = "/createaccount", method = RequestMethod.POST )
public String createAccount(@Valid User user, BindingResult result){

    if (result.hasErrors()){

        return "newaccount";
    }

    user.setAuthority("ROLE_USER");
    user.setEnabled(true);

    if ( usersService.exists( user.getUsername()) ){

        result.rejectValue("username", "DuplicateKey.user.username");
        return "newaccount";
    }

    try{

        usersService.create(user);
    }

    catch(DuplicateKeyException ex){

        result.rejectValue("username", "DuplicateKey.user.username");
        return "newaccount";
    }

    return "accountcreated";
}
```

figure:

b. Create method exists inside the **UsersService.java** as following,

```
public boolean exists(String username) {
    return usersDao.exists(username);
}
```

c. Create a method **exists** inside the **UsersDao.java** as following,

```
public boolean exists(String username){

    String sql  = "select count(*) from users where username=:username";
    return jdbc.queryForObject(sql, new MapSqlParameterSource("username", username), Integer.class ) > 0;
}
```

figure:

This method will do the query inside the database and will see if the user is already existed.

103. STORING VALIDATION MESSAGES IN A PROPERTY FILE

a. Create new package messages and put a property file namely **messages.properties** and put the following informations inside,

```
Size.user.username = Username must be between 8 and 15 characters long.
NotBlank.user.username = Username cannot be blank.
Pattern.user.username = Username can only consist of numbers, letters and the underscore character.
DuplicateKey.user.username = This username already exists!!!!
NotBlank.user.password = Password cannot be blank.
Pattern.user.password = Password cannot contain spaces.
Size.user.password = Password must be between 8 and 15 characters long.
ValidEmail.user.email = This does not appear to be a valid email address.
UnmatchedPasswords.user.password = Passwords do not match.
MatchedPasswords.user.password = Passwords match.

Size.offer.name = Name must be between 5 and 100 characters.
ValidEmail.offer.email = This email address is not valid.
Size.offer.text = Text must be between 20 and 255 characters.
```

b. Inside the **offers-servlet.jsp** file, add a new bean after going on the **beans** tab. Class will be **ResourceBundleMessageSource** and new it needs to insert property on the **messageSource**. The entire definition will be as following,

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value=" com.spring.test.messages.messages"></property>
</bean>
```

104. USING JQUERY TO VERIFY THE PASSWORD

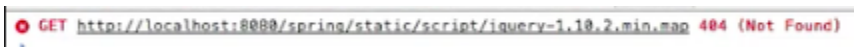
a. Create a new folder namely **script** inside the resources folder. Put the minified **jQuery** inside the script folder. Fix the path inside the project by **Project properties -> JavaScript -> Include path -> exclude by filling the form (**/static/**)**

b. **Project properties -> validation -> enable project specific settings -> Client-side JS validator -> settings -> Add exclude group -> Exclude group -> Add rule -> Add folder/ file name -> next ->** browse and select **jQuery** file.

c. Take the **/newaccount** in the Chrome and open **Setting-> Tools -> JS console**

d. Like the jQuery file in the **newaccount.jsp** file with the **<script>** tag.

e. Delete the top of the **jquery.js** and the following error will go away from the JS console.



f. Put JS inside the **<script>** tag as following,

```
function onLoad() {

    $("#password").keyup(checkPasswordsMatch);
    $("#confirmpass").keyup(checkPasswordsMatch);
    $("#details").submit(canSubmit);
}

function canSubmit() {

    var password = $("#password").val();
    var confirmpass = $("#confirmpass").val();

    if(password != confirmpass) {

        alert("<fmt:message key='UnmatchedPasswords.user.password' />")
        return false;
    }

    else {

        return true;
    }
}
```

```
function checkPasswordsMatch() {

    var password = $("#password").val();
    var confirmpass = $("#confirmpass").val();

    if (password.length > 3 || confirmpass.length > 3) {

        if (password == confirmpass) {

            $("#matchpass").text("<fmt:message key='MatchedPasswords.user.password' />");
            $("#matchpass").addClass("valid");
            $("#matchpass").removeClass("error");
        }

        else {

            $("#matchpass").text("<fmt:message key='UnmatchedPasswords.user.password' />");
            $("#matchpass").addClass("error");
            $("#matchpass").removeClass("valid");
        }
    }
}
$(document).ready(onLoad);
```

105. USING PROPERTY FILE VALUES IN JSP's

a. Update the **messages.properties** file with the following code,

UnmatchedPasswords.user.password = Passwords do not match.
MatchedPasswords.user.password = Passwords match.

b. Search **fmt** taglib in the Google and put in top of the **newaccount.jsp** file,

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

Then, we can get the alerts as following,

alert("<fmt:message key='UnmatchedPasswords.user.password' />")
alert("<fmt:message Key="MatchedPasswords.user.password">")

106. ADDING A LOGOUT LINK

a. Create a new **logout.jsp** file in the JSP folder.

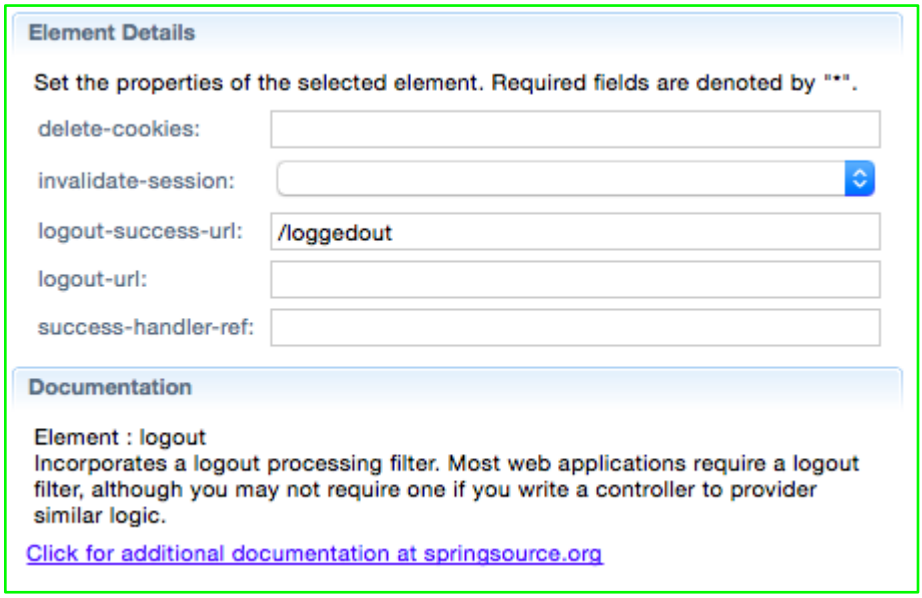
b. Go in the **LoginController.java** class and write method **showLoggedOut** inside the class.

```
 @RequestMapping("/loggedout")
public String showLoggedOut() {

     return "loggedout";
}
```

c. In the **security-context.xml** file, put the following code,

```
<security:http>
     <security:intercept-url                pattern="/loggedout"
access="permitAll" />
</security:http>
```

d. Put a security rule for the log out. Go in the **security-context.xml** and security **sec** namespace. In the **http** tab, insert **<security:logout>** element. Put parameters as the following,



e. In the **home.jsp** file, put a logout link. The **home.jsp** will be finally like the following,

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

<p><a href="${pageContext.request.contextPath}/offers">Show current offers.</a></p>
<p><a href="${pageContext.request.contextPath}/createoffer">Add a new offer.</a></p>

<sec:authorize access="!isAuthenticated()">
<p><a href="<c:url value='/login'/>">Log in</a></p>
</sec:authorize>

<sec:authorize access="isAuthenticated()">
<p><a href="<c:url value='/j_spring_security_logout'/>">Log out</a></p>
</sec:authorize>

<sec:authorize access="hasRole('ROLE_ADMIN')">
<p><a href="<c:url value='/admin'/>">Admin</a></p>
</sec:authorize>

</body>
</html>
```

figure: home.jsp

107. WORKING WITH ROLES

a. Create an **admin.jsp** file inside the JSP folder only for the authorized users.
b. In the **HomeController.java**, create a method **showAdmin** with the following code,

```
@RequestMapping("/admin")
public String showAdmin() {

    return "admin";
}
```

c. Create **admin** user in the authorities table. Put the code inside the JSP file as following,

```
<security:intercept-url pattern="/admin"
access="hasRole('ROLE_ADMIN')" />
```

108. OUTPUTTING TEXT BASED ON AUTHENTICATION STATUS

a. It will discuss how the text will be showed in the JSP pages based on the level of authorization. Here is a great page on the subject matter : <https://docs.spring.io/spring-security/site/docs/3.0.x/reference/el-access.html> . Search for the Spring security taglib and put the following in top of the page,

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

Put that link in the top of the **home.jsp** file for adding security in the page. It will need to add **spring-security-taglib** JAR in the **pom.xml** file.

b. In the **home.jsp** file, show texts based on the authentication of the page as following,

```
<sec:authorize access="!isAuthenticated()">
     <p><a href="<c:url value='/login'/>">Log in</a></p>
</sec:authorize>

<sec:authorize access="isAuthenticated()">
     <p><a href="<c:url value='/j_spring_security_logout'/>">Log out</a></p>
</sec:authorize>

<sec:authorize access="hasRole('ROLE_ADMIN')">
     <p><a href="<c:url value='/admin'/>">Admin</a></p>
</sec:authorize>
```

109. ROW MAPPING WITH BEAN-PROPERTY-ROW-MAPPER

a. Put the **showAdmin** method inside the **LoginController.java** and edit it as following,

```
@RequestMapping("/admin")
```

```
public String showAdmin(Model model) {

      List<User> users = usersService.getAllUsers();
      model.addAttribute("users", users);
      return "admin";
}
```

b. Add getAllUsers method inside the **UsersService.java** class as following,

```
@Secured("ROLE_ADMIN")
public List<User> getAllUsers() {

      return usersDao.getAllUsers();
}
```

c. Write **getAllUsers** method inside the **UsersDao.java** class as following,

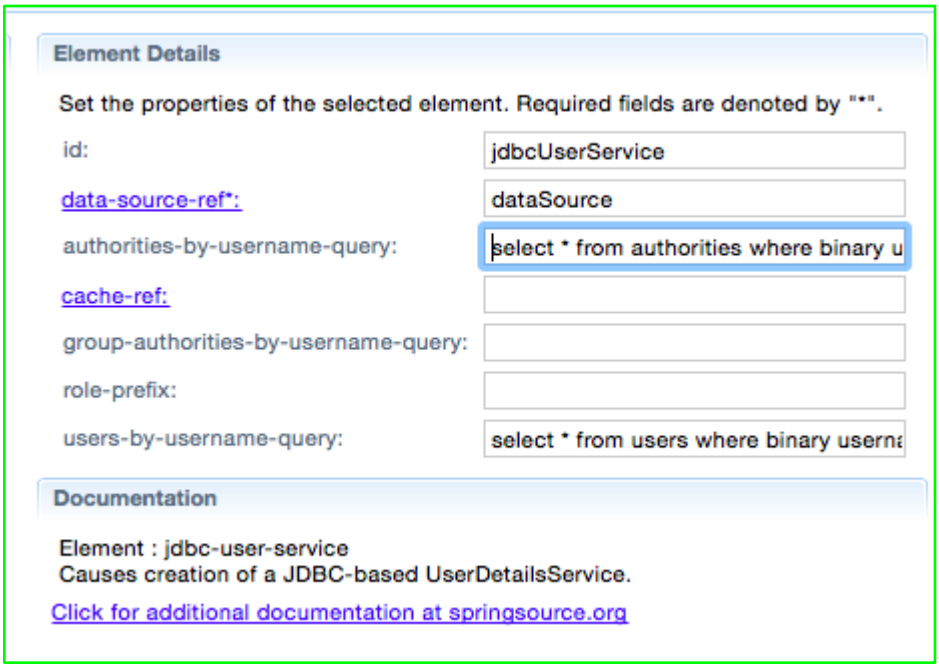
```
public List<User> getAllUsers() {

      return jdbc.query("select * from users,  authorities
where                    users.username=authorities.username",
BeanPropertyRowMapper.newInstance(User.class));

}
```

d. Go inside the **admin.jsp** file and put the following code inside,

```
<body>
<h3>Authorised Users Only!</h3>

<table class="formtable">
<tr><td>Username</td><td>Email</td><td>Role</td><td>Enabled</td></tr>

<c:forEach var="user" items="${users}">

<tr><td><c:out value="${user.username}"></c:out></td>
<td><c:out value="${user.email}"></c:out></td>
<td><c:out value="${user.authority}"></c:out></td>
<td><c:out value="${user.enabled}"></c:out></td>

</tr>

</c:forEach>

</table>
</body>
```

110. USING CUSTOM AUTHENTICATION QUERIES: CASE SENSITIVE USER NAMES

a. In the **security-context.xml** file, go inside the **authentication-provider -> jdbc-user-service** and fill the form using the following parameters,



The query will be as following,

select * from authorities where binary username = ?
select * from users where binary username = ?

So, we will see inside the **security-context.xml** code as following,

```
<security:authentication-manager>
    <security:authentication-provider>
        <security:jdbc-user-service data-source-ref="dataSource"
            authorities-by-username-query='select * from authorities where binary username = ?'
            users-by-username-query='select * from users where binary username = ?'
            id="jdbcUserService" />
        <security:password-encoder ref="passwordEncoder"></security:password-encoder>
    </security:authentication-provider>
</security:authentication-manager>
```

111. METHOD LEVEL ACCESS CONTROL

a. We will define access level based on the authentication provided on the certain methods. In the **security-context.xml** file, put **<security:global-method-security>** elements on the beans as following,

<security:global-method-security secured-annotations="**enabled**"></security:global-method-security>

b. For using this, we will need to use **ROLE** in the database. So, change authorities table with **USER_ROLE** and **USER_ADMIN**

```
<security:http use-expressions="true">
    <security:intercept-url pattern="/admin" access="hasRole('ROLE_ADMIN')" />
    <security:intercept-url pattern="/createoffer" access="isAuthenticated()" />
    <security:intercept-url pattern="/docreate" access="isAuthenticated()" />
    <security:intercept-url pattern="/offercreated" access="isAuthenticated()" />
    <security:intercept-url pattern="/" access="permitAll" />
    <security:intercept-url pattern="/denied" access="permitAll" />
    <security:intercept-url pattern="/loggedout" access="permitAll" />
    <security:intercept-url pattern="/newaccount" access="permitAll" />
    <security:intercept-url pattern="/createaccount" access="permitAll" />
    <security:intercept-url pattern="/accountcreated" access="permitAll" />
    <security:intercept-url pattern="/static/**" access="permitAll" />
    <security:intercept-url pattern="/login" access="permitAll" />
    <security:intercept-url pattern="/offers" access="permitAll" />
    <security:intercept-url pattern="/**" access="denyAll" />
    <security:form-login login-page="/login"
        authentication-failure-url="/login?error=true" />
    <security:logout logout-success-url="/loggedout" />
    <security:access-denied-handler error-page="/denied" />
    <security:remember-me key="offersAppKey"
        user-service-ref="jdbcUserService" />
</security:http>
```

c. Inside the **UsersService.java** file, update **getAllUsers** methods as following,

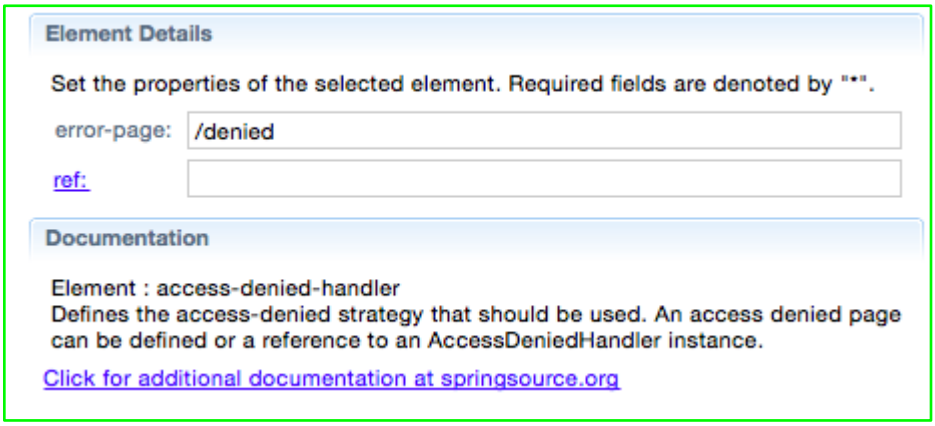
```
@Secured("ROLE_ADMIN")
public List<User> getAllUsers() {

    return usersDao.getAllUsers();
}
```

d. Update **create** method inside the the **OffersService.java** class as following by providing @Secure annotation,

```
@Secured({"ROLE_USER", "ROLE_ADMIN"})
public void create(Offer offer) {

    offersDao.create(offer);
}
```

112. CATCHING SECURE ANNOTATION VIOLATIONS

a. In the **security-context.xml** file and on the **http** tab, insert **<security: access-denied-handler>** element, enter **/denied** in the error-page as following,



Check in the using **showAdmin** method with the following code,

```
@RequestMapping("/admin")
public String showAdmin(Model model){

      throw new AccessDeniedException("Hello");
}
```

b. Use the **ErrorHandler.java** class for dealing with errors as following,

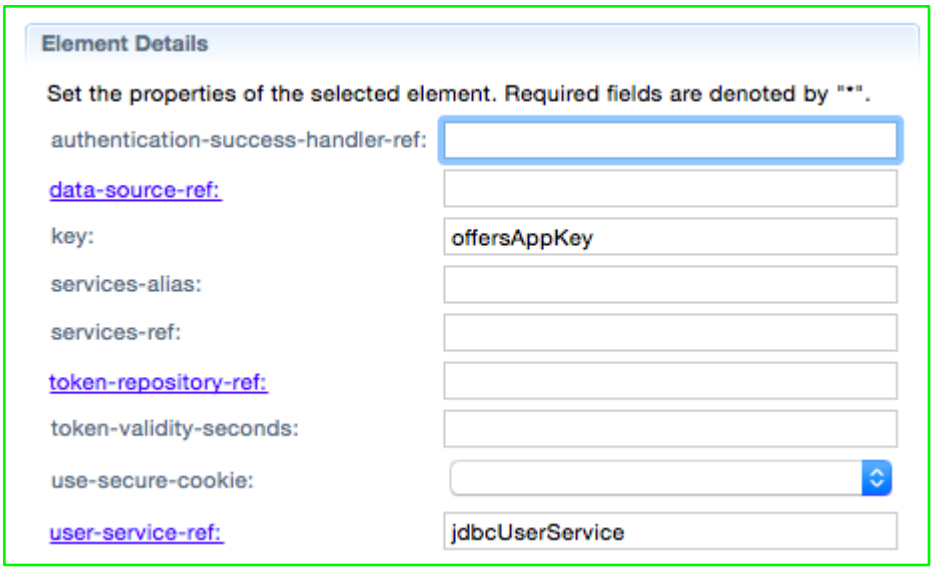
```java
package com.caveofprogramming.spring.web.controllers;

import org.springframework.dao.DataAccessException;

@ControllerAdvice
public class ErrorHandler {
    @ExceptionHandler(DataAccessException.class)
    public String handleDatabaseException(DataAccessException ex) {
        ex.printStackTrace();
        return "error";
    }

    @ExceptionHandler(AccessDeniedException.class)
    public String handleAccessException(AccessDeniedException ex) {
        return "denied";
    }
}
```

113. ADDING REMEMBER ME FUNCTIONALITY

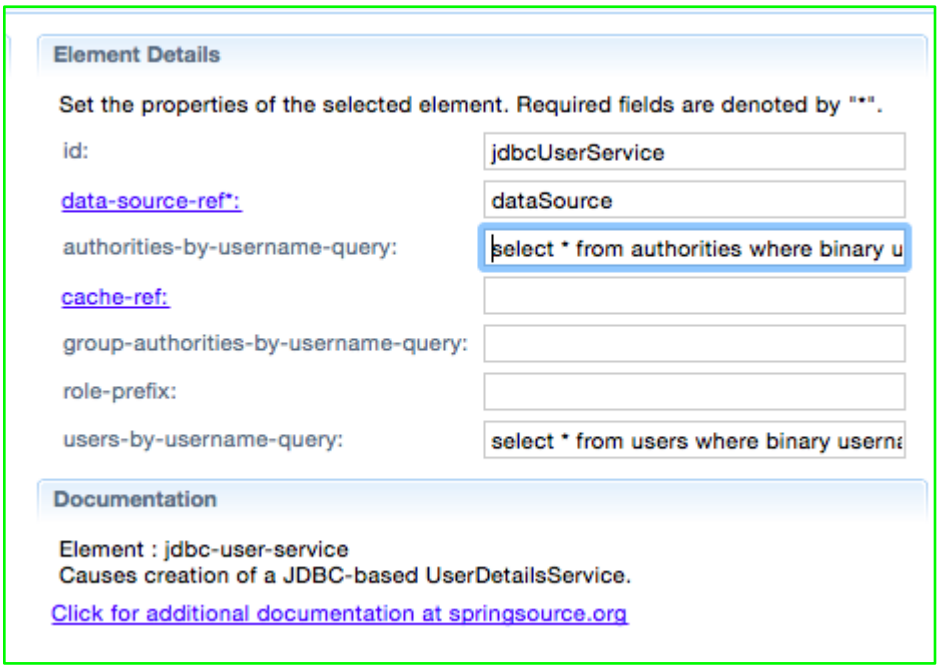
a. Inside the **web.xml** file, put the session out informations as following,

```
<session-config>
  <session-timeout>20</session-timeout>
</session-config>
```

b. In the **security-context.xml** file and over the **http** tab, insert **<security:remember-me>** element. On that tab and inside the key, put **offersAppKey** as value.



For JDBC, we can use **user-service-ref** as **jdbcUserService** . This **id** needs to be matched **jdbc-user-service** as following,



The code on the **security-context.xml** will be as following,

```
<security:remember-me key="offersAppKey"
                      user-service-ref="jdbcUserService" />
```

c. Implement the remember me section in the **login.jsp** file as following,

```
<tr>
    <td>Remember me:</td>
    <td><input type='checkbox'
name='_spring_security_remember_me' checked="checked"/></td>
</tr>
```

114. ENCRYPTING PASSWORDS

a. Add new bean in the **security-context.xml** as following,

```
<bean id="passwordEncoder"
    class="org.springframework.security.crypto.password.St
andardPasswordEncoder">
</bean>
```

b. In the **security-context.xml** file, provide the reference for password encoding as following,

authentication-manager->authentication-provider->password-encoder -> ref = "passwordEncoder"

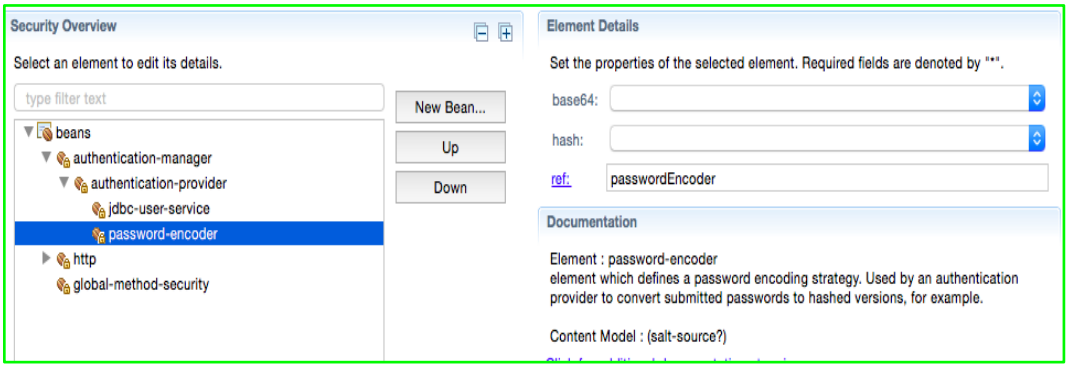


figure:

c. Inside the **UsersDao.java** insert the following code,

```
@Autowired
private PasswordEncoder passwordEncoder;

	@Transactional
	public boolean create(User user) {

		MapSqlParameterSource params = new
MapSqlParameterSource();

		params.addValue("username", user.getUsername());
		params.addValue("password",
passwordEncoder.encode(user.getPassword()));
		params.addValue("email", user.getEmail());
		params.addValue("enabled", user.isEnabled());
		params.addValue("authority", user.getAuthority());

		jdbc.update("insert into users (username, password,
email, enabled) values (:username, :password, :email, :enabled)",
params);

		return jdbc.update("insert into authorities (username,
authority) values (:username, :authority)", params) == 1;

	}
```

d. Inside the **dao-context.xml** file, insert a new bean as following with property of name and ref as **dataSource**,

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<tx:annotation-driven />
```

figure:

e. Inside the **newaccount.jsp** file, change the type from **text** to **password** as following,

```
		<tr>
```

```
                    <td class="label">Password:</td>
                    <td><sf:input    id="password"    class="control"
path="password"
                              name="password" type="password" />
                         <div class="error">
                              <sf:errors
path="password"></sf:errors>
                         </div></td>
          </tr>
```

SEC-11: APACHE TILES AND SPRING MVC

115. TILES DEPENDENCIES

116. HELLO WORLD APACHE TILES

a. add new bean **“tilesViewResolver”** in the **offers-servlet.xml** file as following,

```
<bean id="tilesViewResolver"
    class="org.springframework.web.servlet.view.tiles2.TilesViewResolver">
</bean>
```

b. add another bean **“tilesConfigure”** with class **“TilesConfigure”**. Insert property inside the bean as following,

```
<bean id="tilesConfigurer"
    class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/layouts/default.xml</value>
        </list>
    </property>
</bean>
```

c. Add new folder **layouts** in the **WEB-INF** folder. Add new **default.xml** file inside the folder.

d. Add the location of the **default.xml** inside the **offers-servlet.xml** file using bean ID **tilesConfigurer** as mentioned before.

e. Write the various page info inside the **default.xml** file using **<tiles-definition>** tag.

f. Add a new folder **templates** with the **WEB-INF** folder and add **default.jsp** file inside as following,

```
<definition name="offers.base" template="/WEB-INF/templates/default.jsp">
    <put-attribute name="includes" value=""></put-attribute>
    <put-attribute name="title" value="Offers Homepage"></put-attribute>
    <put-attribute name="header" value="/WEB-INF/tiles/header.jsp"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/content.jsp"></put-attribute>
    <put-attribute name="footer" value="/WEB-INF/tiles/footer.jsp"></put-attribute>
</definition>
```

figure:

117. ADDING HEADERS AND FOOTERS

a. Get the **taglib** for apache tiles and put inside the **default.jsp** file.

b. Create a folder called **tiles** inside the **WEB-INF** folder.

c. Put **content.jsp**, **header.jsp** and **footer.jsp** files inside the **tiles** folder.

d. In the **default.jsp** all the information's about **content.jsp**, **header.jsp** and **footer.jsp** will remain.

```
<tiles:insertAttribute name="includes"></tiles:insertAttribute>
</head>
<body>

    <div class="header">
        <tiles:insertAttribute name="header"></tiles:insertAttribute>
    </div>

    <div class="content">
        <tiles:insertAttribute name="content"></tiles:insertAttribute>
    </div>

    <hr />
    <div class="footer">
        <tiles:insertAttribute name="footer"></tiles:insertAttribute>
    </div>
</body>
```

f. In the **default.jsp** file, use the **<tiles:insertAttribute>** as tag.

e. In the **default.xml**, add the **<put-attribute>** for listing other JSP files.

```
<definition name="offers.base" template="/WEB-INF/templates/default.jsp">
    <put-attribute name="includes" value=""></put-attribute>
    <put-attribute name="title" value="Offers Homepage"></put-attribute>
    <put-attribute name="header" value="/WEB-INF/tiles/header.jsp"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/content.jsp"></put-attribute>
    <put-attribute name="footer" value="/WEB-INF/tiles/footer.jsp"></put-attribute>
</definition>

<definition name="home" extends="offers.base">
    <put-attribute name="title" value="Offers Homepage"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/home.jsp"></put-attribute>
</definition>

<definition name="offers" extends="offers.base">
    <put-attribute name="title" value="Current Offers"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/offers.jsp"></put-attribute>
</definition>

<definition name="login" extends="offers.base">
    <put-attribute name="title" value="Login"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/login.jsp"></put-attribute>
</definition>

<definition name="newaccount" extends="offers.base">
    <put-attribute name="includes"
        value="/WEB-INF/tiles/newaccountscript.jsp"></put-attribute>
    <put-attribute name="title" value="Create Account"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/newaccount.jsp"></put-attribute>
</definition>

<definition name="accountcreated" extends="offers.base">
    <put-attribute name="title" value="Account Created"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/accountcreated.jsp"></put-attribute>
</definition>
```

```
<definition name="admin" extends="offers.base">
    <put-attribute name="title" value="Authorised Users Only!"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/admin.jsp"></put-attribute>
</definition>


<definition name="createoffer" extends="offers.base">
    <put-attribute name="title" value="Create Offer"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/createoffer.jsp"></put-attribute>
</definition>


<definition name="denied" extends="offers.base">
    <put-attribute name="title" value="Access Denied"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/denied.jsp"></put-attribute>
</definition>


<definition name="error" extends="offers.base">
    <put-attribute name="title" value="Error"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/error.jsp"></put-attribute>
</definition>


<definition name="loggedout" extends="offers.base">
    <put-attribute name="title" value="Logged Out"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/loggedout.jsp"></put-attribute>
</definition>


<definition name="offercreated" extends="offers.base">
    <put-attribute name="title" value="Offer Created"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/offercreated.jsp"></put-attribute>
</definition>
```

```
<definition name="offerdeleted" extends="offers.base">
    <put-attribute name="title" value="Offer Deleted"></put-attribute>
    <put-attribute name="content" value="/WEB-INF/tiles/offerdeleted.jsp"></put-attribute>
</definition>
```

118. FORMATTING THE OFFER APPLICATION

a. In the **default.jsp** all the information's about **content.jsp**, **header.jsp** and **footer.jsp** will remain.

b. Add some code inside the **header.jsp** and **footer.jsp** files.

c. Update the **main.css** to accommodate the new JSP files.

119. CREATING TILES FROM THE JSP FILES

a. **home.jsp** and **offers.jsp** will be also in the **tiles** folder

b. Create **newaccountscript.jsp** inside the tiles folder and add scripts as following,

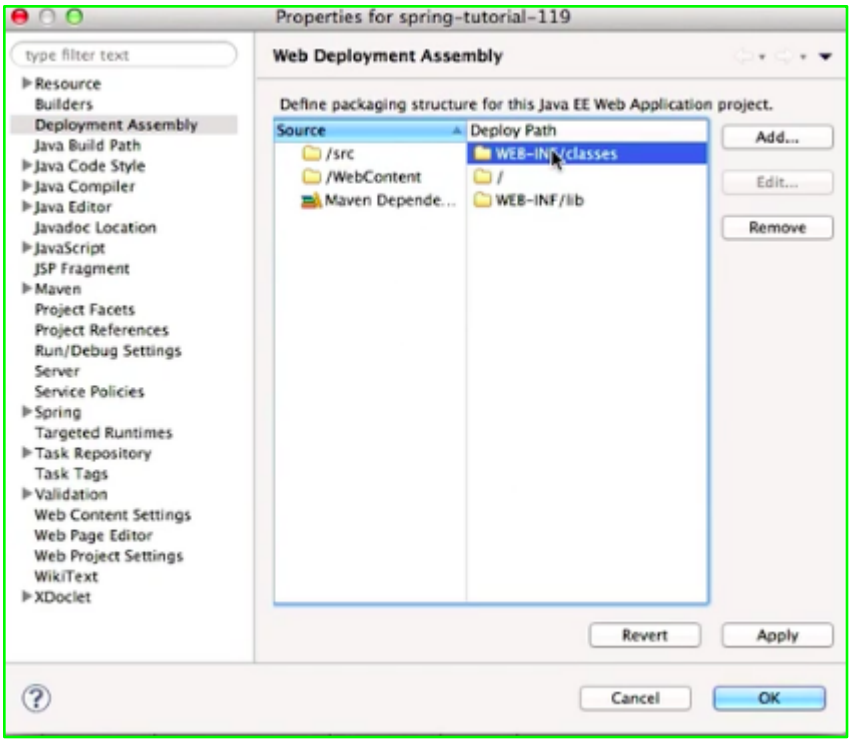
```
function onLoad() {

    $("#password").keyup(checkPasswordsMatch);
    $("#confirmpass").keyup(checkPasswordsMatch);
    $("#details").submit(canSubmit);
}

function canSubmit() {

    var password = $("#password").val();
    var confirmpass = $("#confirmpass").val();

    if (password != confirmpass) {
        alert("<fmt:message key='UnmatchedPasswords.user.password' />")
        return false;
    }

    else {

        return true;
    }
}
```

```
function checkPasswordsMatch() {

    var password = $("#password").val();
    var confirmpass = $("#confirmpass").val();

    if (password.length > 3 || confirmpass.length > 3) {

        if (password == confirmpass) {

            $("#matchpass").text(
                    "<fmt:message key='MatchedPasswords.user.password' />");
            $("#matchpass").addClass("valid");
            $("#matchpass").removeClass("error");
        }

        else {

            $("#matchpass")
                    .text(
                            "<fmt:message key='UnmatchedPasswords.user.password' />");
            $("#matchpass").addClass("error");
            $("#matchpass").removeClass("valid");
        }
    }
}
```

SEC-12: LOGGING AND TESTING

120. LOGGING LOG4J LOGGING

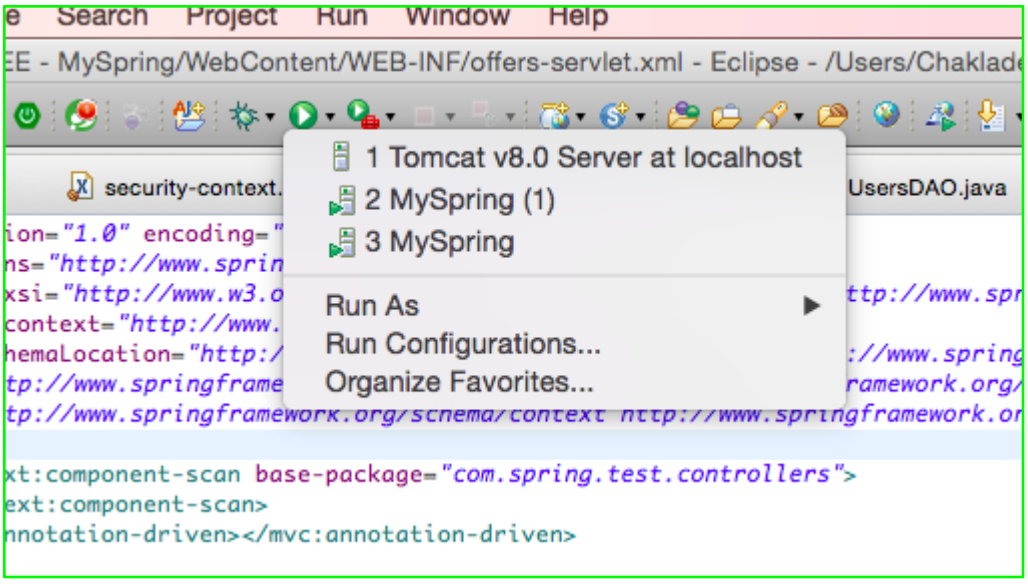
a. Add **log4j** inside the **pom.xml** file (preferably, any previous version).
b. Indie the **project** -> **project property**, we can see the **/src** is deployed inside the **WEB-INF/classes** as following,

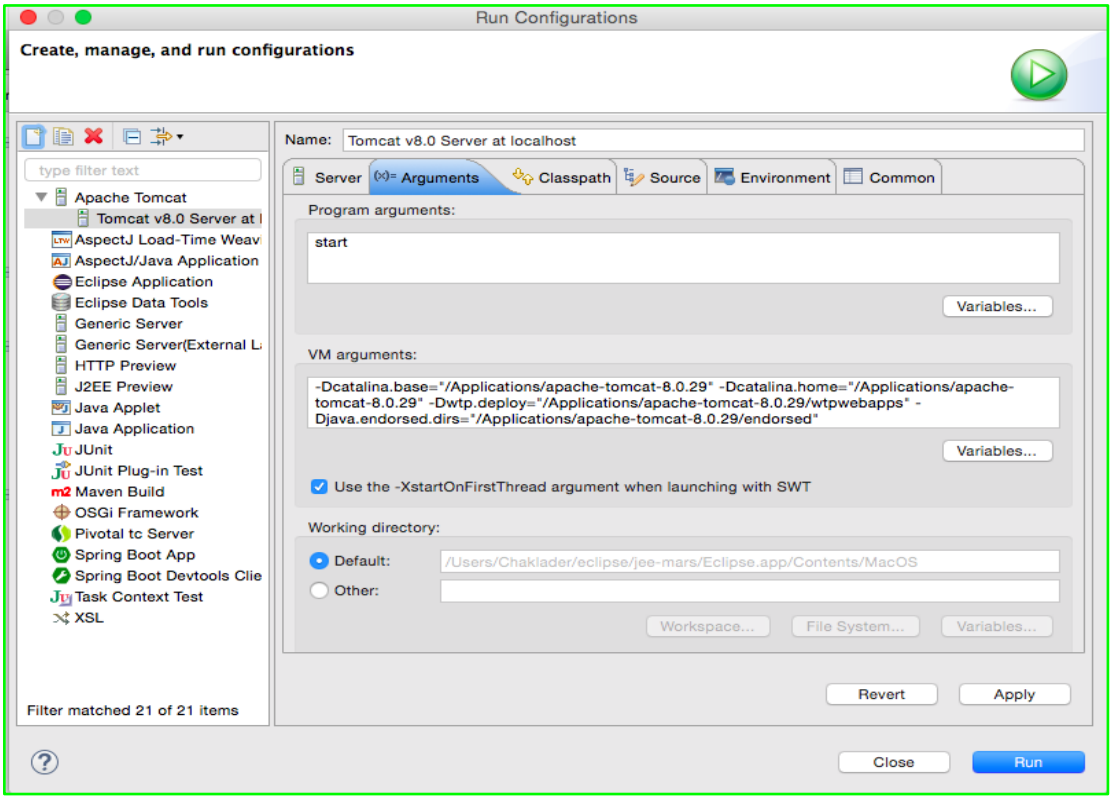


c. Add **log4j.properties** file inside the **src** folderas following,

```
log4j.rootLogger=INFO, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
```

d. In the **Run configuration** and argument tab, put the **VM arguments** as **-Dlog4j.debug**

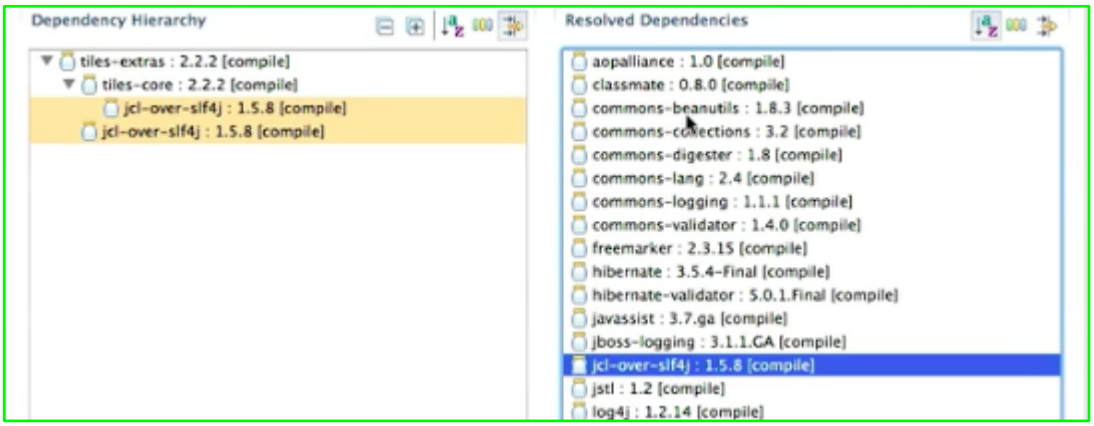




121. RESOLVING LOGGING CONFLICTS

a. Add **slf4j-api** and **slf4j-jcl** in the dependency of the **pom.xml** file.

b. In the dependency hierarchy, select **slf4j-over-jcl** and in the left side on the **tiles-extras: 2.2.2 right** click on the slf4j-over-jcl and **exclude** it as maven artifact from the list. Then, the **slf4j-over-jcl** will removed from the resolved dependencies list.



This include a **exclude** section of the **org.apache.tiles** dependency in the **pom.xml** file as following,

```
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-extras</artifactId>
    <version>2.2.2</version>
    <exclusions>
        <exclusion>
            <artifactId>jcl-over-slf4j</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jcl</artifactId>
    <version>1.7.5</version>
</dependency>
```

122. USING LOGGING

a. Write the logging info in the **HomeController.java** as follwoing,

```
private static Logger logger = Logger.getLogger(HomeController.class);

@Autowired
private OffersService offersService;

@RequestMapping("/")
public String showHome(Model model, Principal principal) {

    logger.debug("Show the home page ... ");
    List<Offer> offers = offersService.getCurrent();

    model.addAttribute("offers", offers);

    boolean hasOffer = false;

    if(principal != null) {
        hasOffer = offersService.hasOffer(principal.getName());
    }

    model.addAttribute("hasOffer", hasOffer);

    return "home";
}
```

We can also write as following,

```
logger.info("Show the home logging info");
```

b. We will also need to make respective changes in the **log4j.properties** file as following,

```
log4j.rootLogger=INFO, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
```

123. USING A MYSQL TEST DATABASE

a. Get the database info from the MySQl database for making test database.

124. USING SPRING PROFILES

a. Add **spring-test** dependency in the **pom.xml** file.
b. In the **dao-context.xml** file, add beans namely **production** as following,

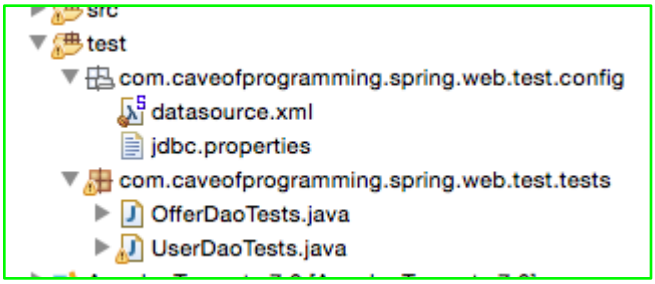
```
<beans profile="production">

    <jee:jndi-lookup jndi-name="jdbc/spring" id="dataSource"
        expected-type="javax.sql.DataSource">
    </jee:jndi-lookup>

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <tx:annotation-driven />

</beans>
```

c. In the **web.xml** file, write a **<context-param>** and put the beans name **production** for reference as following,

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
    classpath:com/caveofprogramming/spring/web/config/dao-context.xml
    classpath:com/caveofprogramming/spring/web/config/service-context.xml
    classpath:com/caveofprogramming/spring/web/config/security-context.xml
    </param-value>
</context-param>
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>production</param-value>
</context-param>
```

125. CREATING JUNIT TESTS

a. Add a new **source folder** namely **test** in the project and add the tests and config package inside the folder as following,



b. Add new dependencies **junit** and **commons-dbcp** (for connecting test database w/o apache tomcat) in the pom.xml file.

c. In the **spring.web.test.config** package, add new bean configuration file namely **datasource.xml** and provide the following informations,

```
    <context:component-scan base-package="com.caveofprogramming.spring.test">
    </context:component-scan>

    <beans profile="dev">
        <context:property-placeholder
            location="com/caveofprogramming/spring/web/test/config/jdbc.properties" />

        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close">

            <property name="driverClassName" value="${jdbc.driver}"></property>
            <property name="url" value="${jdbc.url}"></property>
            <property name="password" value="${jdbc.password}"></property>
            <property name="username" value="${jdbc.username}"></property>
        </bean>
    </beans>
</beans>
```

d. In the same **spring.web.test.config** package, add a new **jdbc.properties** file and provide the info as following,

```
jdbc.username = root
jdbc.password = letmein
jdbc.driver = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/springtest
```

e. In the **spring.web.test.tests** package, add 2 source file for test namely **UserDaoTests.java** and **OfferDaoTest.java** for performing the tetsing.

f. In the **datasource.xml,** add the beans profile as **"dev"**.

g. In the UserDAOTests.java file write the tests as following,

```
@ActiveProfiles("dev")
@ContextConfiguration(locations = {
        "classpath:com/caveofprogramming/spring/web/config/dao-context.xml",
        "classpath:com/caveofprogramming/spring/web/config/security-context.xml",
        "classpath:com/caveofprogramming/spring/web/test/config/datasource.xml" })

@RunWith(SpringJUnit4ClassRunner.class)
public class UserDaoTests {

    @Autowired
    private UsersDao usersDao;

    @Autowired
    private DataSource dataSource;

    @Before
    public void init() {
        JdbcTemplate jdbc = new JdbcTemplate(dataSource);

        jdbc.execute("delete from offers");
        jdbc.execute("delete from users");
    }

    @Test
    public void testUsers() {
        User user = new User("johnwpurcell", "John Purcell", "hellothere",
                "john@caveofprogramming.com", true, "user");

        assertTrue("User creation should return true", usersDao.create(user));
        List<User> users = usersDao.getAllUsers();
        assertEquals("Number of users should be 1.", 1, users.size());

        assertTrue("User should exist.", usersDao.exists(user.getUsername()));

        assertEquals("Created user should be identical to retrieved user",
                user, users.get(0));

    }
}
```

126. CODING THE JUNIT DAO TESTS

a. Inside the **OfferDaoTests.java** file, put the codes as following,

```
@ActiveProfiles("dev")
@ContextConfiguration(locations = {
        "classpath:com/caveofprogramming/spring/web/config/dao-context.xml",
        "classpath:com/caveofprogramming/spring/web/config/security-context.xml",
        "classpath:com/caveofprogramming/spring/web/test/config/datasource.xml" })
@RunWith(SpringJUnit4ClassRunner.class)
public class OfferDaoTests {

    @Autowired
    private OffersDao offersDao;

    @Autowired
    private UsersDao usersDao;

    @Autowired
    private DataSource dataSource;

    @Before
    public void init() {
        JdbcTemplate jdbc = new JdbcTemplate(dataSource);

        jdbc.execute("delete from offers");
        jdbc.execute("delete from users");
    }
```

```
@Test
public void testOffers() {

    User user = new User("johnwpurcell", "John Purcell", "hellothere",
            "john@caveofprogramming.com", true, "user");
    assertTrue("User creation should return true", usersDao.create(user));
    Offer offer = new Offer(user, "This is a test offer.");
    assertTrue("Offer creation should return true", offersDao.create(offer));
    List<Offer> offers = offersDao.getOffers();
    assertEquals("Should be one offer in database.", 1, offers.size());
    assertEquals("Retrieved offer should match created offer.", offer,
            offers.get(0));

    // Get the offer with ID filled in.
    offer = offers.get(0);
    offer.setText("Updated offer text.");
    assertTrue("Offer update should return true", offersDao.update(offer));

    Offer updated = offersDao.getOffer(offer.getId());
    assertEquals("Updated offer should match retrieved updated offer",
            offer, updated);

    // Test get by ID ///////
    Offer offer2 = new Offer(user, "This is a test offer.");
    assertTrue("Offer creation should return true", offersDao.create(offer2));
    List<Offer> userOffers = offersDao.getOffers(user.getUsername());
    assertEquals("Should be two offers for user.", 2, userOffers.size());
    List<Offer> secondList = offersDao.getOffers();

    for(Offer current: secondList) {
        Offer retrieved = offersDao.getOffer(current.getId());

        assertEquals("Offer by ID should match offer from list.", current, retrieved);
    }
    // Test deletion
    offersDao.delete(offer.getId());
    List<Offer> finalList = offersDao.getOffers();
    assertEquals("Offers lists should contain one offer.", 1, finalList.size());
}
```

b. In the User.java add an **equals** method for performing the tests as following,

```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
            + ((authority == null) ? 0 : authority.hashCode());
    result = prime * result + ((email == null) ? 0 : email.hashCode());
    result = prime * result + (enabled ? 1231 : 1237);
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    result = prime * result
            + ((username == null) ? 0 : username.hashCode());
    return result;
}
```

```java
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    User other = (User) obj;
    if (authority == null) {
        if (other.authority != null)
            return false;
    } else if (!authority.equals(other.authority))
        return false;
    if (email == null) {
        if (other.email != null)
            return false;
    } else if (!email.equals(other.email))
        return false;
    if (enabled != other.enabled)
        return false;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    if (username == null) {
        if (other.username != null)
            return false;
    } else if (!username.equals(other.username))
        return false;
    return true;
}
```

Use authority, email, enabled, name and username for equals method. We don’t need to include the password as that will be encrypted in the database.

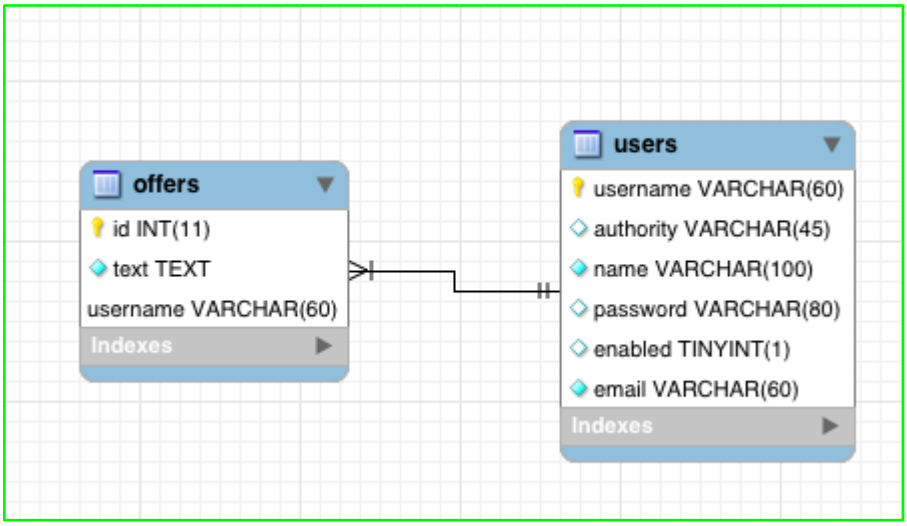
```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((text == null) ? 0 : text.hashCode());
    result = prime * result + ((user == null) ? 0 : user.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Offer other = (Offer) obj;
    if (text == null) {
        if (other.text != null)
            return false;
    } else if (!text.equals(other.text))
        return false;
    if (user == null) {
        if (other.user != null)
            return false;
    } else if (!user.equals(other.user))
        return false;
    return true;
}
```

figure: **Offer.java**

SEC-13: IMPROVING THE “OFFERS” WEB APPLICATION

127. NORMALIZING THE DATABASE

a. Modify the database to keep the shape as following,



128. QUERYING TABLES WITH FOREIGN KEYS AND REFACTORING THE DAO LAYER

a. Change the **User.java** and **Offer.java** class to match with the datbase.

b. Change the **UsersDAO.java** and **OffersDAO.java** matched with as before.

129. REFACTORING THE WEB LAYER

a. Change the **security-context.xml** file for making the queries,

```
<security:authentication-manager>
    <security:authentication-provider>
        <security:jdbc-user-service data-source-ref="dataSource"
            authorities-by-username-query='select username, authority from users where binary username = ?'
            users-by-username-query='select username, password, enabled from users where binary username = ?'
            id="jdbcUserService" />
        <security:password-encoder ref="passwordEncoder"></security:password-encoder>
    </security:authentication-provider>
</security:authentication-manager>
```

b. Change the **messages.properties** file to match with the changes.

c. Change the **newaccount.jsp** to match the change.

d. Change the **Offers.jsp** to meet the matching.

130. GETTING THE USERNAME OF THE LOGGED-IN USER

a. In the **OffersController.java**, impose Principle for adding various entities as following,

```
@RequestMapping("/createoffer")
public String createOffer(Model model, Principal principal) {

    Offer offer = null;

    if (principal != null) {
        String username = principal.getName();

        offer = offersService.getOffer(username);
    }

    if (offer == null) {
        offer = new Offer();
    }

    model.addAttribute("offer", offer);

    return "createoffer";
}
```

131. DELETING FROM TABLES WITH FOREIGN KEYS AND A LITTLE BUGFIX

132. CUSTOM ROWMAPPER

a. Create the **OfferRowMapper.java** as following,

```java
public class OfferRowMapper implements RowMapper<Offer> {

    @Override
    public Offer mapRow(ResultSet rs, int rowNum) throws SQLException {

        User user = new User();
        user.setAuthority(rs.getString("authority"));
        user.setEmail(rs.getString("email"));
        user.setEnabled(true);
        user.setName(rs.getString("name"));
        user.setUsername(rs.getString("username"));

        Offer offer = new Offer();
        offer.setId(rs.getInt("id"));
        offer.setText(rs.getString("text"));
        offer.setUser(user);

        return offer;
    }

}
```

133. CONDITIONAL DATABASE DEPENDENT TEXT IN JSPS

a. Change the **OffersService.java** class and **home.jsp, createoffer.jsp** file to match the requirement.

134. EDITING DATABASE OBJECTS WITH FORMS

a. Add a new method **SaveOrUpdate(Offer offer)** inside the **OffersService.java** class.

135. MULTIPLE FORM SUBMITS AND OPERATIONAL PARAMETERS

a. Add delete button in the **createoffer.jsp** file to delete the offers.

136. ADDING A CONFIRM DIALOGUE WITH JQUERY

a. Add some JS in the **createoffer.jsp** file so that delete button will popup an alert and only after confirming the atert, the offer will be deleted.

SEC -14: HIBERNATE

137. INTRODUCING HIBERNATE

a. Hibernate is an object relational mapping framework.

138. A SIMPLE HIBERNATE QUERY

a. Add spring-orm **org.springframework** , hibernate-core **org.hibernate** dependency in the **pom.xml** file.

b. In the **datasource.xml** file (tests source folder), add **sessionFactory** bean in the **<beans profile="dev">** as following and add 2 properties namely **dataSource** and **hibernateProperties.**

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
            <prop key="javax.persistence.validation.group.pre-persist">com.caveofprogramming.spring.web.dao.PersistenceValidationGroup
            </prop>
            <prop key="javax.persistence.validation.group.pre-update">com.caveofprogramming.spring.web.dao.PersistenceValidationGroup
            </prop>
            <prop key="javax.persistence.validation.group.pre-remove">com.caveofprogramming.spring.web.dao.PersistenceValidationGroup
            </prop>
        </props>
    </property>
    <property name="packagesToScan">
        <list>
            <value>com.caveofprogramming.spring.web.dao</value>
        </list>
    </property>
</bean>
```

c. Change the **UsersDAO.java** class using hibernate as folllowing,

```java
@Repository
@Transactional
@Component("usersDao")
public class UsersDao {

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private SessionFactory sessionFactory;

    public Session session() {
        return sessionFactory.getCurrentSession();
    }

    @Transactional
    public void create(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        session().save(user);
    }

    public boolean exists(String username) {
        Criteria crit = session().createCriteria(User.class);
        crit.add(Restrictions.idEq(username));
        User user = (User)crit.uniqueResult();
        return user != null;
    }

    @SuppressWarnings("unchecked")
    public List<User> getAllUsers() {
        return session().createQuery("from User").list();
    }

}
```

d. Put the transaction informations in the datasource.xml file as following,

```xml
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
<tx:annotation-driven />

<bean id="exceptionTranslator"
    class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor">
</bean>
```

139. SAVING OBJECTS USING HIBERNATE

a. Change the code of the **UsersDAO.java** and **OffersDAO.java** matching with hibernate.

140. VALIDATING GROUPS AND PASSWORD ENCRYPTION

a. Inside the **dao-context.xml** file, put the **sessionFactory** bean.

b. Add **PersistenceValidationGroup.java** and **FormValidationGroup .java** interfaces in the dao package.

c. Inside the **User.java,** provide the validation groups informations as following,

```
@NotBlank(groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Size(min=8, max=15, groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Pattern(regexp="^\\w{8,}$", groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Id
@Column(name="username")
private String username;

@NotBlank(groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Pattern(regexp="^\\S+$", groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Size(min=8, max=15, groups={FormValidationGroup.class})
private String password;

@ValidEmail(groups={PersistenceValidationGroup.class, FormValidationGroup.class})
private String email;

@NotBlank(groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Size(min=8, max=60, groups={PersistenceValidationGroup.class, FormValidationGroup.class})
private String name;

private boolean enabled = false;
private String authority;
```

d. In the **LoginController.java,** provide the group validation information as following,

```
@RequestMapping(value="/createaccount", method=RequestMethod.POST)
public String createAccount(@Validated(FormValidationGroup.class) User user, BindingResult result) {

    if(result.hasErrors()) {
        return "newaccount";
    }

    user.setAuthority("ROLE_USER");
    user.setEnabled(true);

    if(usersService.exists(user.getUsername())) {
        result.rejectValue("username", "DuplicateKey.user.username");
        return "newaccount";
    }

    try {
        usersService.create(user);
    } catch (DuplicateKeyException e) {
        result.rejectValue("username", "DuplicateKey.user.username");
        return "newaccount";
    }


    return "accountcreated";
}
```

e. Now, we will need to tell hibernate which validation group to use.

141. TRANSLATING HIBERNATE EXCEPTIONS TO SPRING EXCEPTION

a. In the datasource.xml, put the new bean **exceptionTranslator** as following,

```
<bean id="exceptionTranslator"
    class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor">
</bean>
```

b. In the **UsersDAO.java**, add the annotation **@Repository** in the top of the class.

142. QUERIES WITH CRITERIA

a. Change the **exists** method in the UsersDAO.java class to match with the hibernate as following,

```
public boolean exists(String username) {

    Criteria crit = session().createCriteria(User.class);
    crit.add(Restrictions.idEq(username));
    User user = (User)crit.uniqueResult();
    return user != null;
}
```

143. MAPPING ONE TO MANY RELATIONSHIPS

a. In the top of the user.java class, put **@Entity** and **@Table(name = "offers")** , in the top of int id, use annotation **@Id** and **@GeneratedValue**

b. User has many to one relationship in the table. So, in the top of **private User user,** put **@ManyTOOne** and **@JoinColumn(** name **="username")**

c. Above the private String text, out the annotation **@Column(** name = **"text").** In last everything will be as following,

```
@Id
@GeneratedValue
private int id;

@Size(min=5, max=255, groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Column(name="text")
private String text;

@ManyToOne
@JoinColumn(name="username")
private User user;
```

d. In the **OffersController.java** class, put the validation informations as follwoing,

```
@RequestMapping(value = "/docreate", method = RequestMethod.POST)
public String doCreate(Model model, @Validated(value=FormValidationGroup.class) Offer offer,
        BindingResult result, Principal principal,
        @RequestParam(value = "delete", required = false) String delete) {

    if (result.hasErrors()) {
        return "createoffer";
    }

    if(delete == null) {
        String username = principal.getName();
        offer.getUser().setUsername(username);
        offersService.saveOrUpdate(offer);
        return "offercreated";
    }
    else {
        offersService.delete(offer.getId());
        return "offerdeleted";
    }

}
```

144. RESTRICTIONS OF JOINED TABLES

a. Change the **OffersDAO.java** to match with the hibernate as following,

```java
@SuppressWarnings("unchecked")
public List<Offer> getOffers() {
    Criteria crit = session().createCriteria(Offer.class);

    crit.createAlias("user", "u").add(Restrictions.eq("u.enabled", true));

    return crit.list();
}

@SuppressWarnings("unchecked")
public List<Offer> getOffers(String username) {
    Criteria crit = session().createCriteria(Offer.class);

    crit.createAlias("user", "u");

    crit.add(Restrictions.eq("u.enabled", true));
    crit.add(Restrictions.eq("u.username", username));

    return crit.list();
}

public void saveOrUpdate(Offer offer) {
    session().saveOrUpdate(offer);
}

public boolean delete(int id) {
    Query query = session().createQuery("delete from Offer where id=:id");
    query.setLong("id", id);
    return query.executeUpdate() == 1;
}

public Offer getOffer(int id) {
    Criteria crit = session().createCriteria(Offer.class);
    crit.createAlias("user", "u");
    crit.add(Restrictions.eq("u.enabled", true));
    crit.add(Restrictions.idEq(id));
    return (Offer)crit.uniqueResult();
}
```

145. MULTIPLE CRITERIA

   a. Change the getOffers methods in the **OffersDAO.java** for hibernate and OfferDAOTests.java for performing the unit tests.

146. UPDATING OBJECTS

147. DELETING OBJECTS

148. COMPLETING THE OFFERS DAO