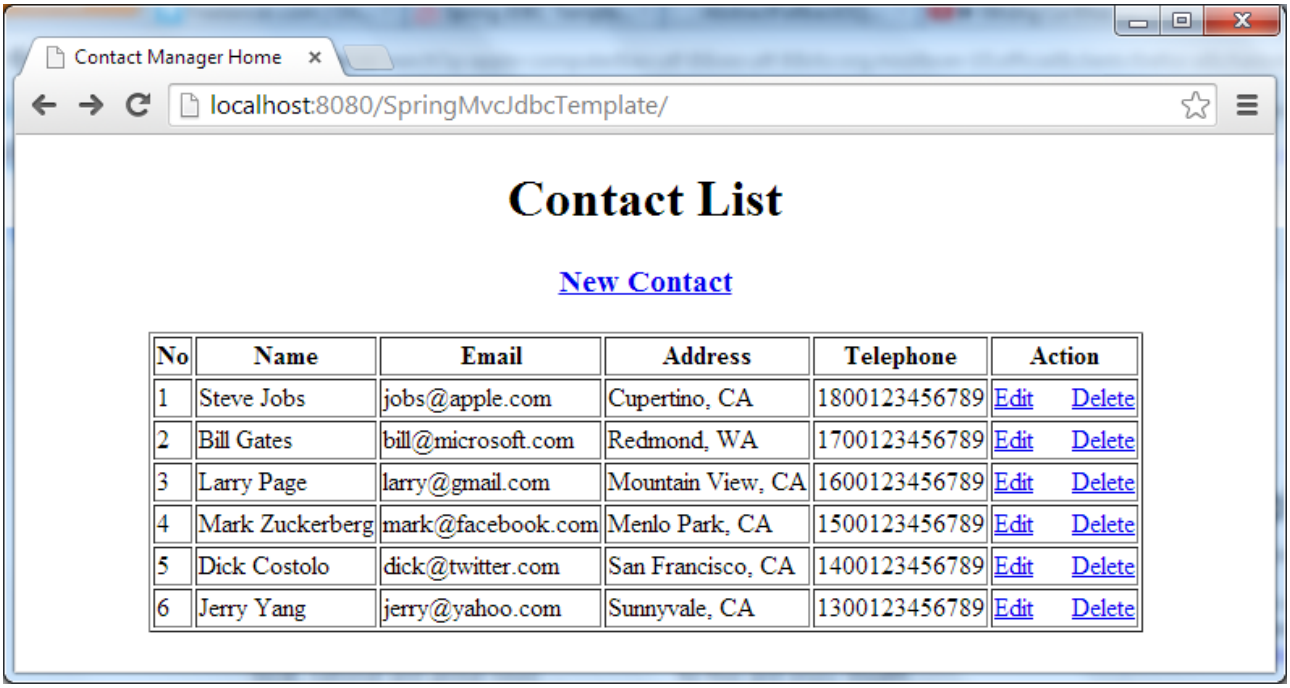
Spring MVC with JdbcTemplate Example

Last Updated on 20 January 2016 | Print Email

Spring makes it easy to work with JDBC through the use of **`JdbcTemplate`** and related classes in the `org.springframework.jdbc.core` and related packages. For an introductory tutorial for the basics of **`JdbcTemplate`**, see: Spring JDBC Template Simple Example. This tutorial goes further by demonstrating how to integrate **`JdbcTemplate`** in a Spring MVC application. The sample application in this tutorial manages a contact list that looks like this:



The sample application is developed using the following pieces of software/technologies:

- Java 7
- Eclipse Kepler
- Spring framework 4.0
- JSTL 1.2
- MySQL Database 5.5
- Maven 3

1. Creating MySQL database

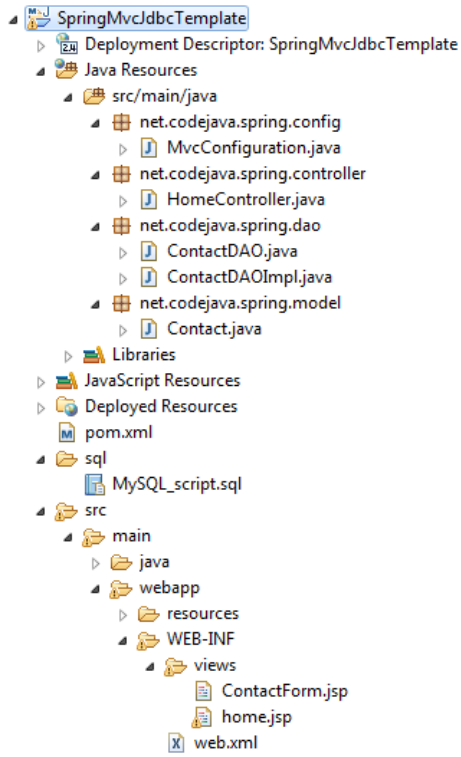
Execute the following MySQL script to create a database named **contactdb** and a table named **contact**:

```
create database contactdb;

CREATE TABLE `contact` (
  `contact_id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `email` varchar(45) NOT NULL,
  `address` varchar(45) NOT NULL,
  `telephone` varchar(45) NOT NULL,
  PRIMARY KEY (`contact_id`)
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT CHARSET=utf8
```

2. Creating Maven Project in Eclipse

It's recommended to use **spring-mvc-archetype** to create the project (See: Creating a Spring MVC project using Maven and Eclipse in one minute). Here's the project's final structure:



The following XML section in `pom.xml`file is for adding dependencies configuration to the project:

```
<properties>
        <java.version>1.7</java.version>
        <spring.version>4.0.3.RELEASE</spring.version>
        <cglib.version>2.2.2</cglib.version>
</properties>

<dependencies>
        <!-- Spring core & mvc -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>${spring.version}</version>
        </dependency>

        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>${spring.version}</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-orm</artifactId>
                <version>${spring.version}</version>
                <type>jar</type>
                <scope>compile</scope>
        </dependency>

        <!-- CGLib for @Configuration -->
        <dependency>
                <groupId>cglib</groupId>
                <artifactId>cglib-nodep</artifactId>
                <version>${cglib.version}</version>
                <scope>runtime</scope>
        </dependency>


        <!-- Servlet Spec -->
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.1.0</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>javax.servlet.jsp</groupId>
                <artifactId>javax.servlet.jsp-api</artifactId>
                <version>2.3.1</version>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>jstl</groupId>
                <artifactId>jstl</artifactId>
                <version>1.2</version>
        </dependency>

</dependencies>
```

3. Coding Model Class

The model class - **`Contact.java`** - is pretty simple:

```
package net.codejava.spring.model;

public class Contact {
        private int id;
        private String name;
        private String email;
        private String address;
        private String telephone;

        public Contact() {
        }

        public Contact(String name, String email, String address, String telephone) {
                this.name = name;
                this.email = email;
                this.address = address;
                this.telephone = telephone;
        }

        // getters and setters
}
```


This class simply maps a row in the table **contact** to a plain old Java object (POJO) - `Contact`.

This book: **Getting started with Spring Framework** helps you master all major concepts like Spring core modules, dependency injection, Spring AOP, annotation-driven development, and more.

4. Coding DAO Classes

The **`ContactDAO`** interface defines methods for performing CRUD operations on the **contact** table:

```
package net.codejava.spring.dao;

import java.util.List;

import net.codejava.spring.model.Contact;

/**
 * Defines DAO operations for the contact model.
 * @author www.codejava.net
 *
 */
public interface ContactDAO {

        public void saveOrUpdate(Contact contact);

        public void delete(int contactId);

        public Contact get(int contactId);

        public List<Contact> list();
}
```


And here is an implementation - **`ContactDAOImpl.java`**:

```
package net.codejava.spring.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import net.codejava.spring.model.Contact;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;

/**
 * An implementation of the ContactDAO interface.
 * @author www.codejava.net
 *
 */
public class ContactDAOImpl implements ContactDAO {

        private JdbcTemplate jdbcTemplate;

        public ContactDAOImpl(DataSource dataSource) {
                jdbcTemplate = new JdbcTemplate(dataSource);
        }

        @Override
        public void saveOrUpdate(Contact contact) {
                // implementation details goes here...
        }

        @Override
        public void delete(int contactId) {
                // implementation details goes here...
        }

        @Override
        public List<Contact> list() {
                // implementation details goes here...
        }

        @Override
        public Contact get(int contactId) {
                // implementation details goes here...
        }

}
```

Pay attention to the beginning section that declares a **`JdbcTemplate`** and a **`DataSource`** object is injected via the constructor:

```
private JdbcTemplate jdbcTemplate;

public ContactDAOImpl(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
}
```

Now, let’s look at implementation details of each method.

Insert or update a new contact:

```
public void saveOrUpdate(Contact contact) {
        if (contact.getId() > 0) {
                // update
                String sql = "UPDATE contact SET name=?, email=?, address=?, "
                                        + "telephone=? WHERE contact_id=?";
                jdbcTemplate.update(sql, contact.getName(), contact.getEmail(),
                                contact.getAddress(), contact.getTelephone(), contact.getId());
        } else {
                // insert
                String sql = "INSERT INTO contact (name, email, address, telephone)"
                                        + " VALUES (?, ?, ?, ?)";
                jdbcTemplate.update(sql, contact.getName(), contact.getEmail(),
                                contact.getAddress(), contact.getTelephone());
        }

}
```

Note that if the contact object having ID greater than zero, update it; otherwise that is an insert.

Delete a contact:

```
public void delete(int contactId) {
        String sql = "DELETE FROM contact WHERE contact_id=?";
        jdbcTemplate.update(sql, contactId);
}
```

List all contact:

```
public List<Contact> list() {
        String sql = "SELECT * FROM contact";
        List<Contact> listContact = jdbcTemplate.query(sql, new RowMapper<Contact>() {

                @Override
                public Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
                        Contact aContact = new Contact();

                        aContact.setId(rs.getInt("contact_id"));
                        aContact.setName(rs.getString("name"));
                        aContact.setEmail(rs.getString("email"));
                        aContact.setAddress(rs.getString("address"));
                        aContact.setTelephone(rs.getString("telephone"));

                        return aContact;
                }

        });

        return listContact;
}
```

Notice the use of **`RowMapper`** to map a row in the result set to a POJO object.

Get a particular contact:

```
public Contact get(int contactId) {
        String sql = "SELECT * FROM contact WHERE contact_id=" + contactId;
        return jdbcTemplate.query(sql, new ResultSetExtractor<Contact>() {

                @Override
                public Contact extractData(ResultSet rs) throws SQLException,
                                DataAccessException {
                        if (rs.next()) {
                                Contact contact = new Contact();
                                contact.setId(rs.getInt("contact_id"));
                                contact.setName(rs.getString("name"));
                                contact.setEmail(rs.getString("email"));
                                contact.setAddress(rs.getString("address"));
                                contact.setTelephone(rs.getString("telephone"));
                                return contact;
                        }

                        return null;
                }

        });
}
```

Notice the use of **`ResultSetExtractor`** to extract a single row as a POJO.

5. Coding MVC Configuration

Java-based classes and annotations are used to configure this Spring MVC application. Here's code of the **`MvcConfiguration`** class:

```java
package net.codejava.spring.config;

import javax.sql.DataSource;

import net.codejava.spring.dao.ContactDAO;
import net.codejava.spring.dao.ContactDAOImpl;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan(basePackages="net.codejava.spring")
@EnableWebMvc
public class MvcConfiguration extends WebMvcConfigurerAdapter{

        @Bean
        public ViewResolver getViewResolver(){
                InternalResourceViewResolver resolver = new InternalResourceViewResolver();
                resolver.setPrefix("/WEB-INF/views/");
                resolver.setSuffix(".jsp");
                return resolver;
        }

        @Override
        public void addResourceHandlers(ResourceHandlerRegistry registry) {
                registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");
        }

        @Bean
        public DataSource getDataSource() {
                DriverManagerDataSource dataSource = new DriverManagerDataSource();
                dataSource.setDriverClassName("com.mysql.jdbc.Driver");
                dataSource.setUrl("jdbc:mysql://localhost:3306/contactdb");
                dataSource.setUsername("root");
                dataSource.setPassword("P@ssw0rd");

                return dataSource;
        }

        @Bean
        public ContactDAO getContactDAO() {
                return new ContactDAOImpl(getDataSource());
        }
}
```

Notice the `getDataSource()` method returns a configured `DataSource` bean. You may have to change the database URL, username and password according to your environments.

The `getContactDAO()` method returns an implementation of the `ContactDAO` interface, which is the `ContactDAOImpl` class. This bean will be injected to the controller class, which is described below.

This book: **Spring in Practice** covers 66 Spring development techniques that help you solve practical issues you will encounter when using Spring framework.

6. Coding Controller Class

Skeleton of the **`HomeController`** class:

```
public class HomeController {

        @Autowired
        private ContactDAO contactDAO;

        // handler methods go here...
}
```

Notice we use the **`@Autowired`** annotation to let Spring inject an instance of the `ContactDAO`implementation into this controller automatically. Each handler method uses this `contactDAO` object to perform necessary CRUD operations. Let's see implementation details of each method.

Handler method for listing all contacts (also served as home page):

```
@RequestMapping(value="/")
public ModelAndView listContact(ModelAndView model) throws IOException{
        List<Contact> listContact = contactDAO.list();
        model.addObject("listContact", listContact);
        model.setViewName("home");

        return model;
}
```

Handler method for displaying new contact form:

```
@RequestMapping(value = "/newContact", method = RequestMethod.GET)
public ModelAndView newContact(ModelAndView model) {
        Contact newContact = new Contact();
        model.addObject("contact", newContact);
        model.setViewName("ContactForm");
        return model;
}
```

Handler method for inserting/updating a contact:

```
@RequestMapping(value = "/saveContact", method = RequestMethod.POST)
public ModelAndView saveContact(@ModelAttribute Contact contact) {
        contactDAO.saveOrUpdate(contact);
        return new ModelAndView("redirect:/");
}
```

Handler method for deleting a contact:

```
@RequestMapping(value = "/deleteContact", method = RequestMethod.GET)
public ModelAndView deleteContact(HttpServletRequest request) {
        int contactId = Integer.parseInt(request.getParameter("id"));
        contactDAO.delete(contactId);
        return new ModelAndView("redirect:/");
}
```

Handler method for retrieving details of a particular contact for editing:

```
@RequestMapping(value = "/editContact", method = RequestMethod.GET)
public ModelAndView editContact(HttpServletRequest request) {
        int contactId = Integer.parseInt(request.getParameter("id"));
        Contact contact = contactDAO.get(contactId);
        ModelAndView model = new ModelAndView("ContactForm");
        model.addObject("contact", contact);

        return model;
}
```


► Template Form ► Spring MVC ► Java Source Code ► Spring Framework

7. Coding Contact Listing Page (Home Page)

Here's source code of the **`home.jsp`** page that displays the contact list as well as action links for creating new, editing and deleting a contact.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Contact Manager Home</title>
    </head>
    <body>
        <div align="center">
                <h1>Contact List</h1>
                <h3><a href="/newContact">New Contact</a></h3>
                <table border="1">
                        <th>No</th>
                        <th>Name</th>
                        <th>Email</th>
                        <th>Address</th>
                        <th>Telephone</th>
                        <th>Action</th>

                                <c:forEach var="contact" items="${listContact}" varStatus="status">
                        <tr>
                                <td>${status.index + 1}</td>
                                        <td>${contact.name}</td>
                                        <td>${contact.email}</td>
                                        <td>${contact.address}</td>
                                        <td>${contact.telephone}</td>
                                        <td>
                                                <a href="/editContact?id=${contact.id}">Edit</a>
                                                    
                                                <a href="/deleteContact?id=${contact.id}">Delete</a>
                                        </td>

                        </tr>
                                </c:forEach>
                        </table>
        </div>
    </body>
</html>
```

Notice this JSP page uses JSTL and EL expressions.

8. Coding Contact Form Page

The contact form page (**`ContactForm.jsp`**) displays details of a contact for creating new or updating old one. Here's its full source code:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>New/Edit Contact</title>
</head>
<body>
        <div align="center">
                <h1>New/Edit Contact</h1>
                <form:form action="saveContact" method="post" modelAttribute="contact">
                <table>
                        <form:hidden path="id"/>
                        <tr>
                                <td>Name:</td>
                                <td><form:input path="name" /></td>
                        </tr>
                        <tr>
                                <td>Email:</td>
                                <td><form:input path="email" /></td>
                        </tr>
                        <tr>
                                <td>Address:</td>
                                <td><form:input path="address" /></td>
                        </tr>
                        <tr>
                                <td>Telephone:</td>
                                <td><form:input path="telephone" /></td>
                        </tr>
                        <tr>
                                <td colspan="2" align="center"><input type="submit" value="Save"></td>
                        </tr>
                </table>
                </form:form>
        </div>
</body>
</html>
```

Notice that this JSP page uses Spring form tags to bind the values of the form to a model object.

To test out the application, you can download the Eclipse project or deploy the attached WAR file at your convenience.

This video tutorial: **The Java Spring Tutorial** help you discover how to master the Spring framework instantly.

You may be also interested in:

- Understanding the core of Spring framework
- Spring MVC beginner tutorial with Spring Tool Suite IDE
- Spring JDBC Template Simple Example
- JDBC Tutorial: SQL Insert, Select, Update, and Delete Examples
- How to list records in a database table using JSP and JSTL

Share this article:

Attachments:

SpringMvcJdbcTemplate.war	[Deployable WAR file]	5521 kB
SpringMvcJdbcTemplate.zip	[Eclipse-Maven project]	27 kB