

---

# Table of Contents

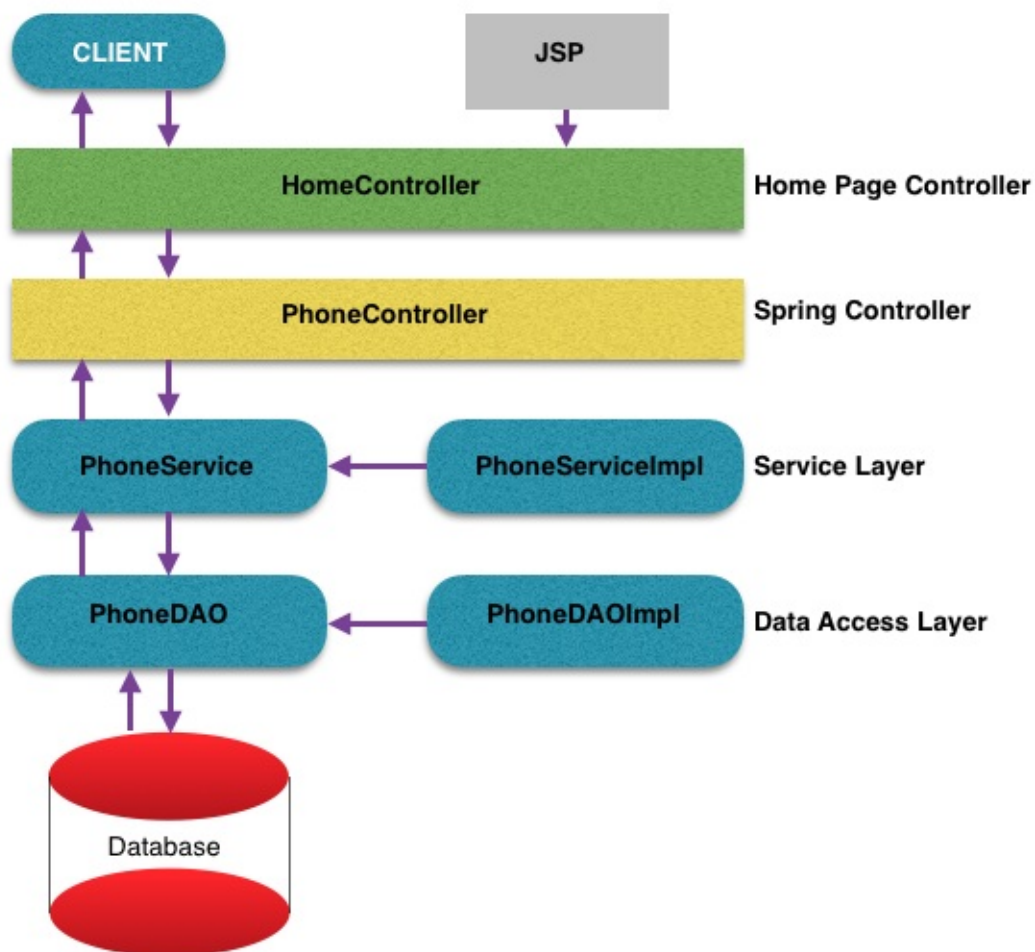
Introduction	1.1
Architecture	1.1.1
Graph	1.1.1.1
Dependencies	1.2
PostgreSQL Database Setup	1.3
Structure in Eclipse	1.4
Entity Class	1.5
DAO Layer	1.6
Service Layer	1.7
Presentation Layer	1.8
PhoneController	1.9
JSP	1.10
Web Configuration	1.11
servlet-context.xml	1.12
Results	1.13

# Objectives

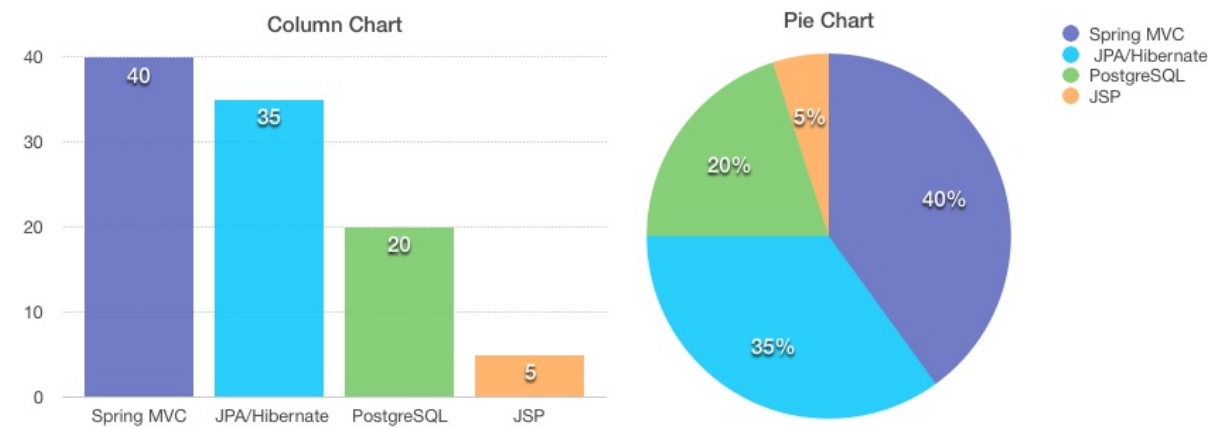
- 1) How to integrate Spring MVC with Hibernate using Maven
- 2) The dependencies which are required to integrate
- 3) How to perform CRUD operations on a Spring MVC project with Hibernate.
- 4) How to configure SessionFactory and Spring configurations.

# Architecture

This architecture clearly explains our demo application. The database will be accessed by a Data Access layer popularly called as DAO Layer. This layer will use Hibernate API to interact with database. The DAO layer will be invoked by a service layer.



# The % of Framework in use



# Dependencies

Let us start by creating a simple web project with maven in Eclipse IDE, now let me tell how to setup the project in pom.xml, follow are the dependencies required for our program:

Important dependencies below are spring-context, spring-webmvc, spring-tx, hibernate-core, hibernate-entitymanager, spring-orm and postgresql driver.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.spring.demo</groupId>
    <artifactId>SpringMVC</artifactId>
    <name>SpringMVC</name>
    <packaging>war</packaging>
    <version>1.0.0-BUILD-SNAPSHOT</version>
    <properties>
        <java-version>1.6</java-version>
        <org.springframework-version>4.0.3.RELEASE</org.springframework-version>
        <org.aspectj-version>1.7.4</org.aspectj-version>
        <org.slf4j-version>1.7.5</org.slf4j-version>
        <hibernate.version>4.3.5.Final</hibernate.version>
    </properties>
    <dependencies>
        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${org.springframework-version}</version>
            <exclusions>
                <!-- Exclude Commons Logging in favor of SLF4j -->
                <exclusion>
```

```
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.4.1207.jre7</version>
</dependency>

<!-- Hibernate -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>

<!-- Apache Commons DBCP -->
<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.4</version>
</dependency>
<!-- Spring ORM -->
```

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

<!-- AspectJ -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>${org.aspectj-version}</version>
</dependency>

<!-- Logging -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${org.slf4j-version}</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
  <version>${org.slf4j-version}</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>${org.slf4j-version}</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.15</version>
  <exclusions>
    <exclusion>
      <groupId>javax.mail</groupId>
      <artifactId>mail</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

```
        <exclusion>
            <groupId>javax.jms</groupId>
            <artifactId>jms</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jdmk</groupId>
            <artifactId>jmxtools</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.sun.jmx</groupId>
            <artifactId>jmxri</artifactId>
        </exclusion>
    </exclusions>
    <scope>runtime</scope>
</dependency>

<!-- @Inject -->
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>

<!-- Servlet -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
```



```
</dependency>

<!-- Test -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.7</version>
  <scope>test</scope>
</dependency>

</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-eclipse-plugin</artifactId>
      <version>2.9</version>
      <configuration>
        <additionalProjectnatures>
          <projectnature>org.springframework.ide.e
clipse.core.springnature</projectnature>
        </additionalProjectnatures>
        <additionalBuildcommands>
          <buildcommand>org.springframework.ide.ec
lipse.core.springbuilder</buildcommand>
        </additionalBuildcommands>
        <downloadSources>true</downloadSources>
        <downloadJavadocs>true</downloadJavadocs>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.5.1</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <compilerArgument>-Xlint:all</compilerArgume
nt>
        <showWarnings>true</showWarnings>
        <showDeprecation>true</showDeprecation>
      </configuration>
    </plugin>
  </plugins>
</build>

```

```
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <version>1.2.1</version>
        <configuration>
            <mainClass>org.test.int1.Main</mainClass>
        </configuration>
    </plugin>
</plugins>
<finalName>${project.artifactId}</finalName>
</build>
</project>
```

# PostgreSQL Database Setup

Let us create a database called test.

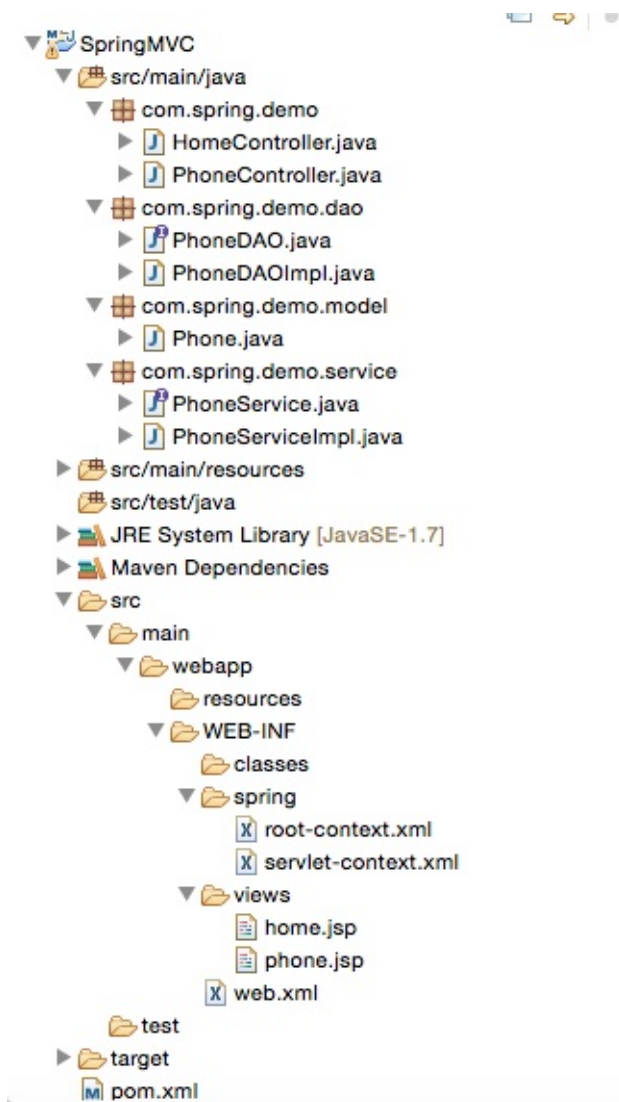
Structure of the Phone table:

```
CREATE TABLE Phone (  
  id bigserial NOT NULL PRIMARY KEY,  
  name varchar(20) NOT NULL,  
  review varchar(20) DEFAULT NULL  
);
```

# Structure

Create following packages under **src/main/java** folder.

- 1) com.demo.spring – This package will contain Spring Controller classes for Phone application.
- 2) com.demo.spring.dao – This is the DAO layer of Phone application. It consists of PhoneDAO interface and its corresponding implementation class. The DAO layer will use Hibernate API to interact with database.
- 3) com.demo.spring.model – This package will contain form object for Phone application. This will be a simple POJO class with different attributes such as id, name, review etc.
- 4) com.demo.spring.service – This package will contain code for service layer for our Phone application. The service layer will have one PhoneService interface and its corresponding implementation class



# Entity Class

First we will create an object or hibernate POJO class to store phone information. Also this class will be an Entity class and will be linked with Phone table in database.

Create a java class Phone.java under com.spring.demo.model package and copy following code into it.

## Note:

- First we've annotated the class with `@Entity` which tells Hibernate that this class represents an object that we can persist.
- The `@Table(name = "PHONE")` annotation tells Hibernate which table to map properties in this class to. The first property in this class is our object ID which will be unique for all events persisted. This is why we've annotated it with `@Id`.
- The `@GeneratedValue` annotation says that this value will be determined by the datasource, not by the code.
- The `@Column(name = "name")` annotation is used to map the property name column in the PHONE table.
- The 2nd `@Column(name = "review")` annotation is used to map the property review column in the PHONE table.
- `@Override toString()` is useful for logging, debugging, or any other circumstance where you need to be able to render any and every object you encounter as a string.
- Implement `equals()` and `hashCode()`

### ***Phone.java***

```
package com.spring.demo.model;
```

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

/**
 * Entity bean with JPA annotations Hibernate provides JPA implementation
 */
@Entity
@Table(name = "PHONE")
public class Phone {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

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

    @Column(name = "review")
    private String review;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
    }

    public String getReview() {
        return review;
    }

    public void setReview(String review) {
        this.review = review;
    }

    @Override
    public String toString() {
        return "Phone [id=" + id + ", name=" + name + ", review="
            + review + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        return result;
    }

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





## DAO Layer

The DAO class code PhoneDAOImpl implements the data access interface PhoneDAO which defines methods such as addPhone(), updatePhone(), listPhones(), getPhoneById() and removePhone() to access data from database.

**Note** that we are going to use `@Repository` Spring annotation.

The `@Repository` annotation is yet another stereotype that was introduced in Spring 2.0. This annotation is used to indicate that a class functions as a repository and needs to have exception translation applied transparently on it. The benefit of exception translation is that the service layer only has to deal with exceptions from Spring's `DataAccessException` hierarchy, even when using plain JPA in the DAO classes.

### *PhoneDAO.java*

```
package com.spring.demo.dao;

import java.util.List;
import com.spring.demo.model.Phone;

public interface PhoneDAO {

    public void addPhone(Phone p);
    public void updatePhone(Phone p);
    public List<Phone> listPhones();
    public Phone getPhoneById(int id);
    public void removePhone(int id);
}
```

### *PhoneDAOImpl.java*

```
package com.spring.demo.dao;

import java.util.List;
```

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Repository;

import com.spring.demo.model.Phone;

@Repository
public class PhoneDAOImpl implements PhoneDAO {

    private static final Logger logger = LoggerFactory.getLogger(
        PhoneDAOImpl.class);

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sf) {
        this.sessionFactory = sf;
    }

    @Override
    public void addPhone(Phone p) {
        Session session = this.sessionFactory.getCurrentSession(
        );
        session.persist(p);
        logger.info("Phone saved successfully, Phone Details=" +
        p);
    }

    @Override
    public void updatePhone(Phone p) {
        Session session = this.sessionFactory.getCurrentSession(
        );
        session.update(p);
        logger.info("Phone updated successfully, Phone Details="
        + p);
    }

    @SuppressWarnings("unchecked")
```

```
@Override
public List<Phone> listPhones() {
    Session session = this.sessionFactory.getCurrentSession(
);
    List<Phone> phonesList = session.createQuery("from Phone
").list();
    for (Phone p : phonesList) {
        logger.info("Phone List::" + p);
    }
    return phonesList;
}

@Override
public Phone getPhoneById(int id) {
    Session session = this.sessionFactory.getCurrentSession(
);
    Phone p = (Phone) session.load(Phone.class, new Integer(
id));
    logger.info("Phone loaded successfully, Phone details="
+ p);
    return p;
}

@Override
public void removePhone(int id) {
    Session session = this.sessionFactory.getCurrentSession(
);
    Phone p = (Phone) session.load(Phone.class, new Integer(
id));
    if (null != p) {
        session.delete(p);
    }
    logger.info("Phone deleted successfully, phone details="
+ p);
}
}
```



# Service Layer

This Service Layer act as a bridge between the DAO (Persistence) layer and the Presentation (Web) layer. Even in service layer similar to DAO layer we have the interface and its implementation.

In the ServiceImpl class, we are using mainly **three** Spring annotations: @Service, @Transactional and @Autowired

## @Service:

- Indicates that the annotated class PhoneServiceImpl is a "Service".
- A stereotype of @Component for persistence layer

## @Transactional:

- Enables Spring's transactional behaviour.
- Can be applied to a class, an interface or a method.
- This annotation is enabled by setting in the context configuration file.
- The attribute readOnly = true which sets the transaction to read only mode so that it cannot modify data in any case.

## @Autowired\*

- Marks the field as to be autowired by Spring's dependency injection facilities.
- The field is injected right after construction of the bean, before any config methods are invoked.
- Autowires the bean by matching the data type in the configuration metadata.

## *PhoneService.java*

```
package com.spring.demo.service;

import java.util.List;

import com.spring.demo.model.Phone;

public interface PhoneService {

    public void addPhone(Phone p);
    public void updatePhone(Phone p);
    public List<Phone> listPhones();
    public Phone getPhoneById(int id);
    public void removePhone(int id);

}
```

### ***PhoneServiceImpl.java***

```
package com.spring.demo.service;

import java.util.List;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.spring.demo.dao.PhoneDAO;
import com.spring.demo.model.Phone;

@Service
public class PhoneServiceImpl implements PhoneService {

    @Autowired
    private PhoneDAO phoneDAO;

    public void setPhoneDAO(PhoneDAO phoneDAO) {
        this.phoneDAO = phoneDAO;
    }

    @Override
```

```
@Transactional
public void addPhone(PHONE p) {
    this.phoneDAO.addPhone(p);
}

@Override
@Transactional
public void updatePhone(PHONE p) {
    this.phoneDAO.updatePhone(p);
}

@Override
@Transactional
public List<PHONE> listPhones() {
    return this.phoneDAO.listPhones();
}

@Override
@Transactional
public PHONE getPhoneById(int id) {
    return this.phoneDAO.getPhoneById(id);
}

@Override
@Transactional
public void removePhone(int id) {
    this.phoneDAO.removePhone(id);
}

}
```



# Presentation Layer

Now we create a controller that handles the web requests dispatched via. DispatcherServlet. Let the name of the controller be PhoneController and put it under the package com.spring.demo.

## @Controller:

- Indicates that the annotated PhoneController class serves the role of a "Controller".
- A stereotype of @Component for web layer.
- To enable autodetection of such annotated controllers, you add component scanning to your configuration at servlet-context.xml.

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

## @Autowired:

- A Constructor or a setter method as to be autowired by Spring's dependency injection facilities.
- The field is injected right after construction of the bean, before any config methods are invoked.
- Autowires the bean by matching the data type in the configuration metadata

## @RequestMapping

- DispatcherServlet uses this annotation to dispatch the requests to the correct controller and the handler method.
- Maps URLs or web requests onto specific handler classes like **@RequestMapping("/phones")** and handler methods like example **@RequestMapping("/edit/{id}")**
- RequestMapping element value is a String array ( String[] ) so it can accept multiple URL paths.
- The handler methods which are annotated with this annotation are allowed to have very flexible return types and method signatures. Can find the detail [here](#).

### **@Qualifier:**

- Used along with @Autowired to remove the conflict by specifying which exact bean will be wired.

Created the following HomeController that will be used to return the default view for our application.

### **HomeController.java**

```
package com.spring.demo;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HomeController {

    @RequestMapping(value="/")
    public ModelAndView mainPage() {
        return new ModelAndView("home");
    }

    @RequestMapping(value="/index")
    public ModelAndView indexPage() {
        return new ModelAndView("home");
    }
}
```

# PhoneContoller

## *PhoneController.java*

```
package com.spring.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.spring.demo.model.Phone;
import com.spring.demo.service.PhoneService;

@Controller
public class PhoneController {

    private PhoneService phoneService;

    @Autowired(required = true)
    @Qualifier(value = "phoneService")
    public void setPhoneService(PhoneService ps) {
        this.phoneService = ps;
    }

    @RequestMapping(value = "/phones", method = RequestMethod.GET)
    public String listPhones(Model model) {
        model.addAttribute("phone", new Phone());
        model.addAttribute("listPhones", this.phoneService.listPhones());
        return "phone";
    }
}
```

```
// For add and update phone both
@RequestMapping(value = "/phone/add", method = RequestMethod
.POST)
public String addPhone(@ModelAttribute("phone") Phone p) {

    if (p.getId() == 0) {
        // new phone, add it
        this.phoneService.addPhone(p);
    } else {
        // existing phone, call update
        this.phoneService.updatePhone(p);
    }

    return "redirect:/phones";

}

@RequestMapping("/remove/{id}")
public String removePhone(@PathVariable("id") int id) {

    this.phoneService.removePhone(id);
    return "redirect:/phones";
}

@RequestMapping("/edit/{id}")
public String editPhone(@PathVariable("id") int id, Model mo
del) {
    model.addAttribute("phone", this.phoneService.getPhoneBy
Id(id));
    model.addAttribute("listPhones", this.phoneService.listP
hones());
    return "phone";
}

}
```

### **@ModelAttribute:**

- This annotation is used for preparing the model data and also to define the

command object that would be bound with the HTTP request data.

- This annotation can be applied on the methods as well as on the method arguments.
- In the above controller example, we used it on the argument of the `addPhone()` method.

An *@ModelAttribute* on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

### **@PathVariable:**

It binds the method parameter to a URI template variable. URI template is a parameterized URI

# JSP

Even though we can create multiple pages but here, we are going to create one single page which will call add, edit, remove and list the data.

***src/webapp/WEB-INF/views/home.jsp*** for default page

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ page session="false" %>
<html>
<head>
    <title>Phone Page</title>
    <style type="text/css">
        .tg {border-collapse:collapse;border-spacing:0;border-color:#ccc;}
        .tg td{font-family:Arial, sans-serif;font-size:14px;padding:10px 5px;border-style:solid;border-width:1px;overflow:hidden;word-break:normal;border-color:#ccc;color:#333;background-color:#fff;}
        .tg th{font-family:Arial, sans-serif;font-size:14px;font-weight:normal;padding:10px 5px;border-style:solid;border-width:1px;overflow:hidden;word-break:normal;border-color:#ccc;color:#333;background-color:#f0f0f0;}
        .tg .tg-4eph{background-color:#f9f9f9}
    </style>
</head>
<body>
<h1>Welcome</h1>
<p>
    ${message}<br/>
    <a href="${pageContext.request.contextPath}/phones">Click here to generate the Phone List</a><br/>
</p>
</body>
</html>

```

### **src/webapp/WEB-INF/views/phone.jsp**

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

```

```
= "form" %>
<%@ page session="false" %>
<html>
<head>
    <title>Phone Page</title>
    <style type="text/css">
        .tg {border-collapse:collapse;border-spacing:0;border-c
olor:#ccc; cellpadding:10;
cellspacing:10;}
        .tg td{font-family:Arial, sans-serif;font-size:14px;padd
ing:10px 5px;border-style:solid;border-width:1px;overflow:hidden
;word-break:normal;border-color:#ccc;color:#333; cellpadding:10;
cellspacing:10;}
        .tg th{font-family:Arial, sans-serif;font-size:14px;font
-weight:normal;padding:10px 5px;border-style:solid;border-width:
1px;overflow:hidden;word-break:normal;border-color:#ccc;color:#3
33;background-color:#f0f0f0; cellpadding:10;
cellspacing:10;}
        .tg .tg-4eph{background-color:#f9f9f9}

a.button {
    background-color: #4CAF50; /* Green */
    border: none;
    color: white;
    padding: 10px 15px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 15px;
    margin: 10px 15px 10px 8px;
}
a.btn{
background-color: #f44336;
    border: none;
    color: white;
    padding: 10px 15px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
```



```
        font-size: 15px;
        margin: 10px 15px 10px 8px;
    }
    .table {
border: 1px;

    }
    .th{ width:10%;
    }
    .value{
margin: 10px 15px 10px 8px;
    }

        </style>
</head>
<body>
<h1>
    Add a Phone
</h1>

<c:url var="addAction" value="/phone/add" ></c:url>

<form:form action="${addAction}" commandName="phone">
<table>
    <c:if test="${!empty phone.name}">
    <tr>
        <td>
            <form:label path="id">
                <spring:message text="ID"/>
            </form:label>
        </td>
        <td>
            <form:input path="id" readonly="true" size="8" disabled="true" />
            <form:hidden path="id" />
        </td>
    </tr>
    </c:if>
    <tr>
        <td>
```

```

        <form:label path="name">
            <spring:message text="Name"/>
        </form:label>
    </td>
    <td>
        <form:input path="name" />
    </td>
</tr>
<tr>
    <td>
        <form:label path="review">
            <spring:message text="Review"/>
        </form:label>
    </td>
    <td>
        <form:input path="review" />
    </td>
</tr>
<tr>
    <td colspan="2">
        <c:if test="${!empty phone.name}">
            <input type="submit"
                value="<spring:message text="Edit Phone"/>"
        />
        </c:if>
        <c:if test="${empty phone.name}">
            <input type="submit"
                value="<spring:message text="Add Phone"/>" /
        >
        </c:if>
    </td>
</tr>
</table>
</form:form>
<br>
<h3>Phones List</h3>
<c:if test="${!empty listPhones}">
    <table class="tg">
        <tr>
            <th width="80">ID</th>

```

```
        <th width="120">Phone Name</th>
        <th width="120">Review</th>
        <th width="60">Action</th>

    </tr>
    <c:forEach items="${listPhones}" var="phone">
        <tr>
            <td>${phone.id}</td>
            <td>${phone.name}</td>
            <td>${phone.review}</td>
            <td><a href="<c:url value='/edit/${phone.id}' />" c
lass="button">Edit</a>
                <a href="<c:url value='/remove/${phone.id}' />" cla
ss="btn">Delete</a></td>
        </tr>
    </c:forEach>
</table>
</c:if>
</body>
</html>
```

# Web Configuration

Now going to web configuration we can see the two configuration xmls. This will be needed if you have multiple servlets and filters that shares common parameters. You can define only one servlet config xml and put all of contents from both xmls into the one. The tag `<context-param>` defines parameters that are shared by all servlets and filters whereas `<init-param>` defines parameters that are accessible by only that `<servlet>` inside which it is placed. We also defined the **DispatcherServlet** which is the default request handler for all the requests as defined by the pattern "/" in `<url-pattern>` .

## `<web-app>`:

- Defines the root element of the document called Deployment Descriptor

## `<context-param>`:

- This is an optional element which contains the declaration of a Web application's servlet context initialization parameters available to the entire scope of the web application.
- The methods for accessing context init parameter is as follows:

```
String paramName = getServletContext().getInitParameter("paramName")
```

## `<init-param>`:

- An optional element within the servlet.
- Also defines servlet configuration initialization parameters only available to the servlet inside which it's defined.
- The methods for accessing servlet init parameter is as follows:

```
String paramName = getServletConfig().getInitParameter("paramName")
```

## `<listener>`:

- Defines a listener class to start up and shut down Spring's root application

context.

- We might not use this because we don't define the element `<context-param>`

### `<servlet>:`

- Declares a servlet including a referring name, defining class and initialization parameters.
- We can define multiple servlets in the document the most interesting thing is you can even declare multiple servlets using the same class with different initialization parameters but, the name of each servlet must be unique.

### `<servlet-mapping>:`

- Maps a URL pattern to a servlet.
- In this case, all requests will be handled by the DispatcherServlet named dispatcherServlet

**web.xml:** /webapp/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <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>
```

# servlet-context.xml

The various components from the above xmls are described as follows:

## `<annotation-driven />:`

- Initializes components required for dispatching the requests to our Controllers.
- Register's a `RequestMappingHandlerMapping`, a `RequestMappingHandlerAdapter`, and an `ExceptionHandlerExceptionResolver` in support of processing requests with annotated controller methods using annotations such as `@RequestMapping`, `@ExceptionHandler` and others.
- Configures support for new Spring MVC features such as declarative validation with `@Valid`, HTTP message conversion with `@RequestBody/@ResponseBody`, formatting Number fields with `@NumberFormat` etc.

This element should be placed in your `DispatcherServlet` context (or in your root context if you have no `DispatcherServlet` context defined)

## `<context:component-scan>:`

- Scans for the classes annotated with `@Component`, `@Service`, `@Repository` and `@Controller` defined under the base-package and registers their corresponding beans.
- `@Component` serves as a generic stereotype for any Spring-managed component; whereas, `@Repository`, `@Service`, and `@Controller` serve as specializations of `@Component` for more specific use cases.

## `<resources>:`

- This tag allows static resource requests following a particular URL pattern to be served by a `ResourceHttpRequestHandler` from any of a list of Resource location.

- This provides a convenient way to serve static resources from location other than the web application root, including locations on the classpath e.g.  
`<mvc:resources mapping="/resources/**" location="/, classpath:/web-resources/" />`

### InternalResourceViewResolver:

- Spring provides different view resolvers to render models in a browser without tying you with a specific view technology.
- Spring enables you to use JSPs, Velocity templates and XSLT views.

InternalResourceViewResolver is one of them.

### src/main/webapp/WEB-INF/sping/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
" xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/mv
c http://www.springframework.org/schema/mvc/spring-mvc.xsd
    http://www.springframework.org/schema/beans http://www.s
pringframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context http://www
.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/tx http://www.spri
ngframework.org/schema/tx/spring-tx-4.0.xsd">

    <!-- DispatcherServlet Context: defines this servlet's reques
st-processing
        infrastructure -->

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

    <!-- Handles HTTP GET requests for /resources/** by efficien
tly serving
        up static resources in the ${webappRoot}/resources direc
```



```
tory -->
    <resources mapping="/resources/**" location="/resources/" />

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

    <beans:bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <beans:property name="driverClassName" value="org.postgresql.Driver" />
        <beans:property name="url"
            value="jdbc:postgresql://localhost:5432/phones" />
        <beans:property name="username" value="postgres" />
        <beans:property name="password" value="1234" />
    </beans:bean>

    <!-- Hibernate 4 SessionFactory Bean definition -->
    <beans:bean id="hibernate4AnnotatedSessionFactory"
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <beans:property name="dataSource" ref="dataSource" />
        <beans:property name="annotatedClasses">
            <beans:list>
                <beans:value>com.spring.demo.model.Phone</beans:value>
            </beans:list>
        </beans:property>
        <beans:property name="hibernateProperties">
            <beans:props>
                <beans:prop key="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect
            </beans:prop>
        </beans:property>
    </beans:bean>
```

```
        <beans:prop key="hibernate.show_sql">true</beans
:prop>
        </beans:props>
    </beans:property>
</beans:bean>

    <beans:bean id="phoneDAO" class="com.spring.demo.dao.PhoneDAOImpl">
        <beans:property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />
    </beans:bean>
    <beans:bean id="phoneService" class="com.spring.demo.service.PhoneServiceImpl">
        <beans:property name="phoneDAO" ref="phoneDAO"></beans:property>
    </beans:bean>
    <context:component-scan base-package="com.spring.demo" />

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <beans:bean id="transactionManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">
        <beans:property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />
    </beans:bean>

</beans:beans>
```

# Results

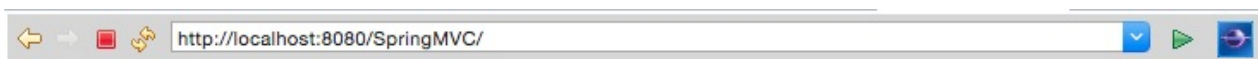
## Running the Application

Now, running the app is quite simple enough.

1. Right click on the project.
2. Go to the Run As-> Maven built
3. Run the mvn command to download all dependent JARs.
4. Run Tomcat server
5. Go to the browser and enter the following URL:  
`http://localhost:8080/SpringMVC/`
6. The port number might be different in your case. Please have a look at the tomcat log in console for that.

The ScreenShots below gives a clear information:

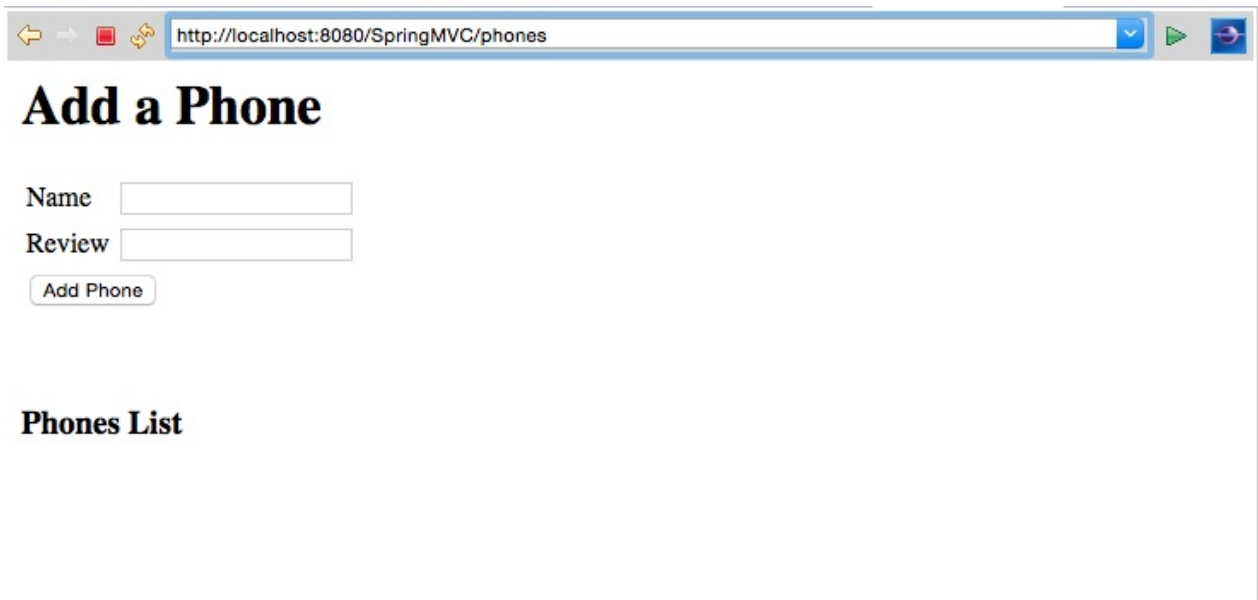
## Home Page



# Welcome

[Click here to generate the Phone List](#)

***When you click on the link***



http://localhost:8080/SpringMVC/phones

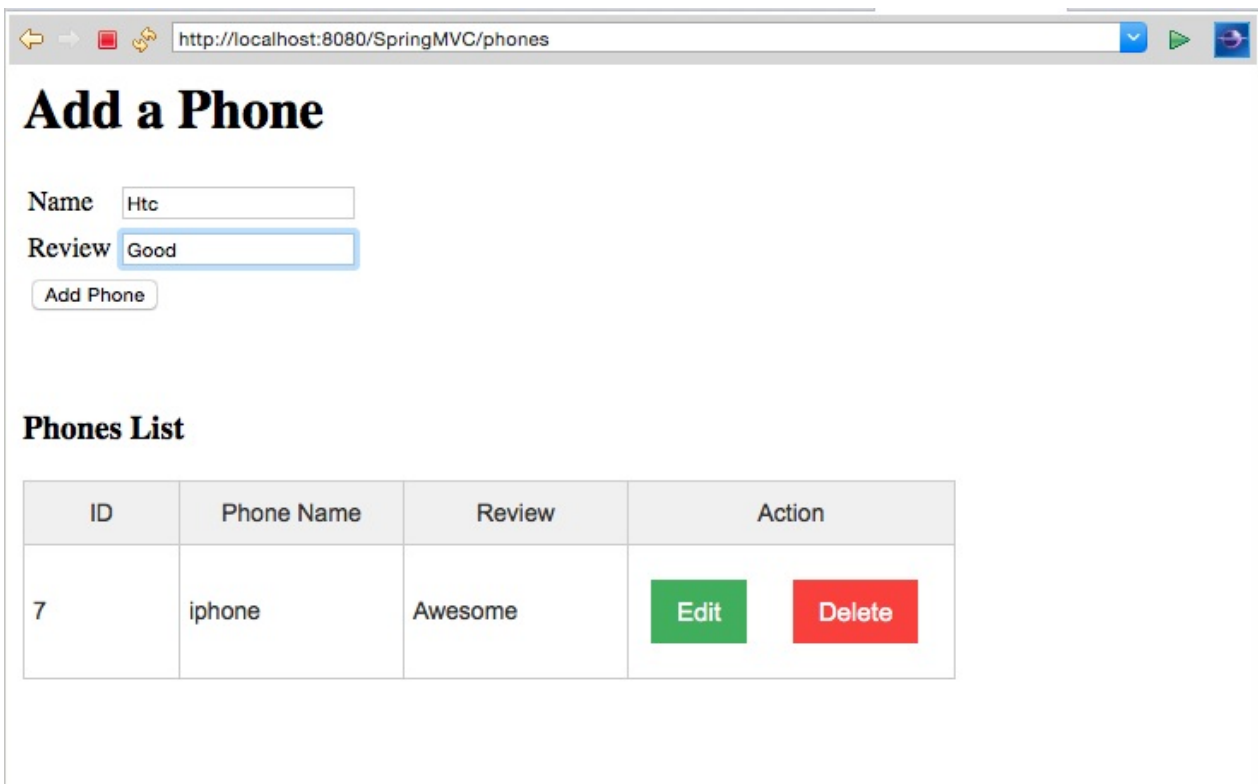
## Add a Phone

Name

Review

### Phones List

## *Add Phone details to the list*



http://localhost:8080/SpringMVC/phones

## Add a Phone

Name

Review

### Phones List

ID	Phone Name	Review	Action
7	iphone	Awesome	<input type="button" value="Edit"/> <input type="button" value="Delete"/>

## *Edit/Delete*

http://localhost:8080/SpringMVC/edit/7

# Add a Phone

ID

7

Name

iphone

Review

Awesome

Edit Phone