**Spring, Spring Security, Spring AOP, ORM , IOC Container, Dependency Injection,**

**Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc.**

# Topic: Inversion Of Control (IOC) , Dependency Injection in Spring, Spring with ORM Frameworks

<https://data-flair.training/blogs/spring-annotation/>

**importants** <https://springframework.guru/spring-framework-annotations/>

<https://www.youtube.com/watch?v=NHmlH-QKfRA> (Spring Boot Logs)

<https://www.baeldung.com/tag/spring-annotations/>

# Spring Tutorial

It was **developed by Rod Johnson in 2003**. Spring framework makes the easy development of JavaEE application.

## Spring Framework: Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

### Spring Bean

Spring Bean is nothing special, any object in the Spring framework that we initialize through Spring container is called Spring Bean. Any normal Java POJO class can be a Spring Bean if it’s configured to be initialized via container by providing configuration metadata information.

### 

### Spring Bean Scopes: There are five scopes defined for Spring Beans.

1. [**singleton**](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) – Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
2. **prototype** – A new instance will be created every time the bean is requested.
3. **request** – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
4. **session** – A new bean will be created for each HTTP session by the container.
5. **global-session** – This is used to create global session beans for Portlet applications.

**///////////////////////////////////////////////////////////////////////////////////**

### Advantages of Spring Framework

1) **Predefined Templates** Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

In JDBC Example ,You don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) **Loose Coupling** The Spring applications are loosely coupled because of dependency injection.

#### 3) Easy to test The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

#### 4) Lightweight Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

#### 5) Fast Development The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

#### 6) Powerful abstraction It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

#### 7) Declarative support: It provides declarative support for caching, validation, transactions and formatting.

**Inversion Of Control (IOC) & Dependency Injection**

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain.

Thus, IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

#### Advantage of Dependency Injection

* makes the code loosely coupled so easy to maintain
* makes the code easy to test

**Spring - IOC Containers**

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.

The Spring container uses DI to manage the components that make up an application.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* To instantiate the application class
* To configure the object
* To assemble the dependencies between the objects

**There are two types of IoC containers.**

**BeanFactory Container:** is defined by the org.springframework.beans.factory.BeanFactory interface. The root interface for accessing the Spring container. Spring’s Dependency Injection functionality using this BeanFactory interface and its subinterfaces.

[**ApplicationContext Container**](https://www.tutorialspoint.com/spring/spring_applicationcontext_container.htm)**:** container is defined by the org.springframework.context.ApplicationContext interface.

The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over BeanFactory.

BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

So it is better to use ApplicationContext than BeanFactory.

**Using BeanFactory:** The XmlBeanFactory is the implementation class for the BeanFactory interface.

To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below: Resource resource=**new** ClassPathResource("applicationContext.xml");

BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Features:Bean instantiation/wiring

**Using ApplicationContext:** The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

The ApplicationContext is the central interface within a Spring application for providing configuration information to the application.

It implements the BeanFactory interface. Hence ApplicationContext includes all functionality of the BeanFactory and much more!

Its main function is to support the creation of big business applications.

**Features:**

Bean instantiation/wiring

Automatic BeanPostProcessor registration

Automatic BeanFactoryPostProcessor registration

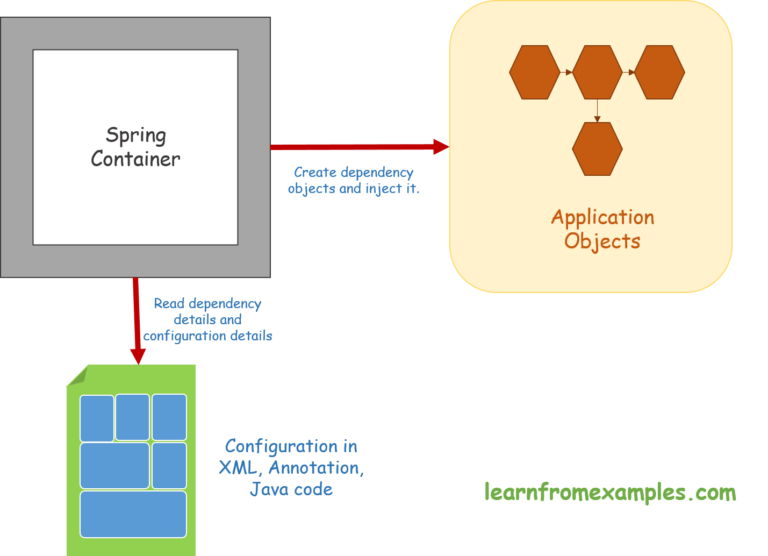
Convenient MessageSource access (for i18n)

ApplicationEvent publication

**Note :**

* IOC container used to read xml document and pass values to pojo class.
* Pojo classes can asscociate other class obj.
* Pojo class loosely cooped to other classess.

Q: How does a spring container work? IOC container (ApllicationContext and beanfacatory)



====================================================================================

**Dependency Injection in Spring: (JAVATPOINT)**

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing.

**Two ways to perform Dependency Injection in Spring framework**

By Constructor , By Setter method

**Note:** We are providing the information into the bean by applicationContext.xml file. The constructor-arg element invokes the constructor.

**Dependency Injection By Constructor:**

**Exam-1: Dependency Injection of primitive and string value by constructor(Here Address is dependent on Employee object)**

<bean id="e" **class**="com.javatpoint.Employee">

<constructor-arg value="12" type="int"></constructor-arg>

<constructor-arg value="Sonoo"></constructor-arg>

</bean>

**Exam-2: Dependency Injection of dependent object by constructor (Here Address is dependent on Employee object)**

<beans>

<bean id="a1" **class**="com.javatpoint.Address">

<constructor-arg value="ghaziabad"></constructor-arg>

<constructor-arg value="UP"></constructor-arg>

<constructor-arg value="India"></constructor-arg>

</bean>

<bean id="e" **class**="com.javatpoint.Employee">

<constructor-arg value="12" type="int"></constructor-arg>

<constructor-arg value="Sonoo"></constructor-arg>

<constructor-arg>

<ref bean="a1"/>

</constructor-arg>

<beans>

**Exam-3: Dependency Injection with Collection by constructor Injection**

**Set,list,map**

**More Examples see in javatpoint (**<https://www.javatpoint.com/spring-tutorial-dependency-injection-by-constructor>**)**

**Dependency Injection By Setter Method:**

The **<property>** sub element of  **<bean>** is used for setter injection.

Examples Using:

* primitive and String-based values
* Dependent object (contained object)
* Collection values etc.

Example1: Appicationcontext.xml file

<bean id="obj" **class**="com.javatpoint.Employee">

<property name="id">

<value>20</value>

</property>

<property name="name">

<value>Arun</value>

</property>

<property name="city">

<value>ghaziabad</value>  A

</property>

</bean>

</beans>

===========================================================================================

**` Autowiring in Spring**

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

**Advantage of Autowiring**

It requires the less code because we don't need to write the code to inject the dependency explicitly.

**Disadvantage of Autowiring**

* No control of programmer.
* It can't be used for primitive and string values.

Autowiring mode: No, byType, byName, constructor, autodetect (deprecated since spring 3)

Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

**tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.

**Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

**Spring Bean Life Cycle**

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there are lists of the activities that take place behind the scene between the time of bean Instantiation and its destruction,

This chapter will discuss only two important bean life cycle callback methods, which are required at the time of bean initialization and its destruction.

To define setup and teardown for a bean, we simply declare the <bean> with init-method and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy method specifies a method that is called just before a bean is removed from the container.

Following is the content of the MainApp.java file. Here you need to register a shutdown hook registerShutdownHook() method that is declared on the AbstractApplicationContext class. This will ensure a graceful shutdown and call the relevant destroy methods.

Example:

public static void main(String[] args) {

AbstractApplicationContext context = new classPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

context.registerShutdownHook();

}

**Default Initialization and Destroy Method**

If you have too many beans having initialization and/or destroy methods with the same name, you don't need to declare init-method and destroy-method on each individual bean. Instead, the framework provides the flexibility to configure such situation using default-init-method and default-destroy-method attributes on the <beans> element as follows −

<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.springframework.org/schema/beans/spring-beans-3.0.xsd](http://www.springframework.org/schema/beans%20%20%20http://www.springframework.org/schema/beans/spring-beans-3.0.xsd)

default-init-method = "init" default-destroy-method = "destroy">

<bean id = "..." class = "..."> <!-- collaborators and configuration for this bean go here --> </bean>

</beans>

**===========================================================================**

**Spring-Bean Post Processors:**

The BeanPostProcessor interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic, etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the order property provided the **BeanPostProcessor implements the** **Ordered** **interface**.

The BeanPostProcessors operate on bean (or object) instances, which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An ApplicationContext automatically detects any beans that are defined with the implementation of the BeanPostProcessor interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation.

**=====================================================================================**

**Spring - Annotation Based Configuration(Aurowiring)**

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.

Example:

<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.springframework.org/schema/beans/spring-beans-3.0.xsd http://www.springframework.org/schema/context

<http://www.springframework.org/schema/context/spring-context-3.0.xsd>">

<context:annotation-config/>

<!-- bean definitions go here -->

</beans>

**Annotation & Description**

* @Required: this annotation applies to bean property setter method.
* @Autowired: this annotation can apply to bean property setter method ,non setter method ,constructor and property.
* @Qualifier: Qualifier annotation along with @autowired can be used to remove confusion by specifying which exact bean will be wired.
* @JSR250 Annotation: Spring support JSR 250 based annotation which include @Resources, @PostConstruct and PreDestroy.

**Note:**  <!--change location of the root application context xml file -->

    <context-param>

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

        <param-value>/WEB-INF/mvc-dispatcher-servlet.xml</param-value>

    </context-param>

**Spring AOP**

AOP breaks the program logic into distinct parts (called concerns).It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

**Why use AOP?**

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

**class** A{

**public** **void** m1(){...}

**public** **void** m2(){...}

**public** **void** m3(){...}

**public** **void** m4(){...}

**public** **void** m5(){...}

**public** **void** n1(){...}

**public** **void** n2(){...}

**public** **void** p1(){...}

**public** **void** p2(){...}

**public** **void** p3(){...}

}

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

Understanding Scenario I have to maintain log and send notification after calling methods that starts from m.

Problem without AOP We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

Solution with AOP We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

**AOP is mostly used in following cases:**

* to provide declarative enterprise services such as declarative transaction management.
* It allows users to implement custom aspects.

**AOP Concepts and Terminology**

* AOP concepts and terminologies are as follows:
* Join point
* Advice
* Pointcut
* Introduction
* Target Object
* Aspect
* Interceptor
* AOP Proxy
* Weaving

**Join point**

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

**Advice**

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

* Before Advice: it executes before a join point.
* After Returning Advice: it executes after a joint point completes normally.
* After Throwing Advice: it executes if method exits by throwing an exception.
* After (finally) Advice: it executes after a join point regardless of join point exit whether normally or exceptional return.
* Around Advice: It executes before and after a join point.

**Pointcut**

It is an expression language of AOP that matches join points.

**Introduction**

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

**Target** **Object**

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

**Aspect**

It is a class that contains advices, joinpoints etc.

**Interceptor**

It is an aspect that contains only one advice.

**AOP Proxy**

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

**Weaving**

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

### Spring AOP

Spring AOP can be used by 3 ways given below. But the widely used approach is Spring AspectJ Annotation Style. The 3 ways to use spring AOP are given below:

1. [By Spring1.2 Old style (dtd based) (also supported in Spring3)](https://www.javatpoint.com/spring-aop-example)
2. [By AspectJ annotation-style](https://www.javatpoint.com/spring-aop-aspectj-annotation-example)
3. [By Spring XML configuration-style(schema based)](https://www.javatpoint.com/spring-aop-aspectj-xml-configuration-example)

**================================================================**

**Spring with ORM Frameworks**

Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA (Java Persistence API), JDO(Java Data Objects), Oracle Toplink and iBATIS.

**Advantage of ORM Frameworks with Spring**

There are a lot of advantage of Spring framework in respect to ORM frameworks. There are as follows:

* **Less coding is required**: By the help of Spring framework, you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.
* **Easy to test**: Spring's IoC approach makes it easy to test the application.
* **Better exception handling**: Spring framework provides its own API for exception handling with ORM framework.
* **Integrated transaction management**: By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

**Hibernate and Spring Integration**

We can simply integrate hibernate application with spring application.

In hibernate framework, we provide all the database information hibernate.cfg.xml file.

But if we are going to integrate the hibernate application with spring, we don't need to create the hibernate.cfg.xml file. We can provide all the information in the applicationContext.xml file.

### Advantage of Spring framework with hibernate

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

So **it saves a lot of code**.

**Methods of HibernateTemplate class**

Let's see a list of commonly used methods of HibernateTemplate class.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | void persist(Object entity) | persists the given object. |
| 2) | Serializable save(Object entity) | persists the given object and returns id. |
| 3) | void saveOrUpdate(Object entity) | persists or updates the given object. If id is found, it updates the record otherwise saves the record. |
| 4) | void update(Object entity) | updates the given object. |
| 5) | void delete(Object entity) | deletes the given object on the basis of id. |
| 6) | Object get(Class entityClass, Serializable id) | returns the persistent object on the basis of given id. |
| 7) | Object load(Class entityClass, Serializable id) | returns the persistent object on the basis of given id. |
| 8) | List loadAll(Class entityClass) | returns the all the persistent objects. |

**Steps**

Let's see what are the simple steps for hibernate and spring integration:

**create table in the database** It is optional.

**create applicationContext.xml file** It contains information of DataSource, SessionFactory etc.

**create Employee.java file** It is the persistent class

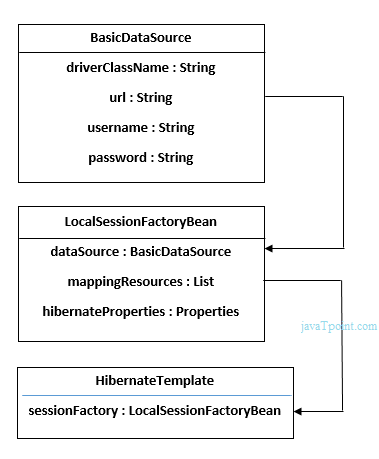
**create employee.hbm.xml file** It is the mapping file.

**create EmployeeDao.java file** It is the dao class that uses HibernateTemplate.

**create InsertTest.java file** It calls methods of EmployeeDao class.

**applicationContext.xml**

In this file, we are providing all the informations of the database in the **BasicDataSource** object. This object is used in the **LocalSessionFactoryBean** class object, containing some other informations such as mappingResources and hibernateProperties. The object of **LocalSessionFactoryBean** class is used in the HibernateTemplate class.



*applicationContext.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:p="http://www.springframework.org/schema/p"

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

        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dataSource" **class**="org.apache.commons.dbcp.BasicDataSource">

        <property name="driverClassName"  value="oracle.jdbc.driver.OracleDriver"></property>

        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>

        <property name="username" value="system"></property>

        <property name="password" value="oracle"></property>

    </bean>

    <bean id="mysessionFactory"  **class**="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

        <property name="mappingResources">

        <list>

        <value>employee.hbm.xml</value>

        </list>

        </property>

        <property name="hibernateProperties">

            <props>

                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>

                <prop key="hibernate.hbm2ddl.auto">update</prop>

                <prop key="hibernate.show\_sql">**true**</prop>

            </props>

        </property>

    </bean>

    <bean id="template" **class**="org.springframework.orm.hibernate3.HibernateTemplate">

    <property name="sessionFactory" ref="mysessionFactory"></property>

    </bean>

    <bean id="d" **class**="com.javatpoint.EmployeeDao">

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

    </bean>

    </beans>

6) InsertTest.java

This class uses the EmployeeDao class object and calls its saveEmployee method by passing the object of Employee class.

package com.javatpoint;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

public class InsertTest {

public static void main(String[] args) {

    Resource r=new ClassPathResource("applicationContext.xml");

    BeanFactory factory=new XmlBeanFactory(r);

    EmployeeDao dao=(EmployeeDao)factory.getBean("d");

    Employee e=new Employee();

    e.setId(114);

    e.setName("varun");

    e.setSalary(50000);

    dao.saveEmployee(e);

}  }

**Enabling automatic table creation, showing sql queries etc.**

You can enable many hibernate properties like automatic table creation by hbm2ddl.auto etc. in applicationContext.xml file. Let's see the code:

<property name="hibernateProperties">

            <props>

                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>

                <prop key="hibernate.hbm2ddl.auto">update</prop>

                <prop key="hibernate.show\_sql">**true**</prop>

            </props>

If you write this code, you don't need to create table because table will be created automatically.

**Spring Data JPA Tutorial**

Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.

JPA (Java Persistent API) is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.

The implementation of JPA specification are provided by many vendors such as:

Hibernate, Toplink, iBatis,OpenJPA etc.

**Advantage of Spring JpaTemplate:** You don't need to write the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance and closing EntityManager. So, it save a lot of code.

**--------------------------------------------------------------------------------------------------------------------**

**Spring Security**

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

It is a sub-project of Spring framework

which was started in 2003 by Ben Alex.

Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

This framework targets two major areas are:

* **Authentication** is the process of knowing and identifying the user that wants to access.
* **Authorization** is the process to allow authority to perform actions in the application.

**Note: We can apply authorization to authorize web request, methods and access to individual domain.**

**Spring Security framework supports wide range of authentication models:**

* HTTP BASIC authentication headers
* HTTP Digest authentication headers
* LDAP (Lighweight Directory Access Protocol)
* Form-based authentication
* Automatic remember-me authentication
* HTTP X.509 client certificate exchange
* OpenID authentication
* Kerberos
* JOSSO (Java Open Source Single Sign-On)
* AppFuse
* AndroMDA
* Mule ESB
* DWR(Direct Web Request)

**Spring Security Features**

* LDAP (Lightweight Directory Access Protocol)
* JAAS (Java Authentication and Authorization Service) Login Module
* Basic Access Authentication
* Digest Access Authentication
* Web Form Authentication
* Authorization
* HTTP Authorization
* Software Localization
* Remember-me
* Single sign-on

### LDAP (Lightweight Directory Access Protocol): It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.

### Single sign-on: This feature allows a user to access multiple applications with the help of single account(user name and password).

### JAAS (Java Authentication and Authorization Service) LoginModule: It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.

### Basic Access Authentication:Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.

### Digest Access Authentication: This feature allows us to make authentication process more secure than Basic Access Authentication. It asks to the browser to confirm the identity of the user before sending sensitive data over the network.

### Remember-me: Spring Security supports this feature with the help of HTTP Cookies. It remember to the user and avoid login again from the same machine until the user logout.

### Web Form Authentication: In this process, web form collect and authenticate user credentials from the web browser. Spring Security supports it while we want to implement web form authentication.

### Authorization: Spring Security provides the this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.

### Software Localization: This feature allows us to make application user interface in any language.

### HTTP Authorization: Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

## Features added in Spring Security 5.0

**OAuth 2.0 Login:** This feature provides the facility to the user to login into the application by using their existing account at GitHub or Google. This feature is implemented by using the Authorization Code Grant that is specified in the OAuth 2.0 Authorization Framework.

**Reactive Support:** From version Spring Security 5.0, it provides reactive programming and reactive web runtime support and even, we can integrate with Spring WebFlux.

**Modernized Password Encoding:** Spring Security 5.0 introduced new Password encoder.

**DelegatingPasswordEncoder:**  Which is more modernize and solve all the problems of previous encoder **NoOpPasswordEncoder**.

**Spring Project Modules**

In Spring Security 3.0, the Security module is divided into separate jar files. The purpose was to divide jar files based on their functionalities, so, the developer can integrate according to their requirement.

The following are the jar files that are included into Spring Security module.

* spring-security-core.jar
* spring-security-remoting.jar
* spring-security-web.jar
* spring-security-config.jar
* spring-security-ldap.jar
* spring-security-oauth2-core.jar
* spring-security-oauth2-client.jar
* spring-security-oauth2-jose.jar
* spring-security-acl.jar
* spring-security-cas.jar
* spring-security-openid.jar
* spring-security-test.jar

**Core - spring-security-core.jar**

This is core jar file and required for every application that wants to use Spring Security. This jar file includes core access-control and core authentication classes and interfaces. We can use it in standalone applications or remote clients applications.

It contains top level packages:

* org.springframework.security.core
* org.springframework.security.access
* org.springframework.security.authentication
* org.springframework.security.provisioning

**Remoting - spring-security-remoting.jar**

This jar is used to integrate security feature into the Spring remote application. We don't need it until or unless we are creating remote application. All the classes and interfaces are located into **org.springframework.security.remoting** package.

**Web - spring-security-web.jar**

This jar is useful for Spring Security web authentication and URL-based access control. It includes filters and web-security infrastructure.

All the classes and interfaces are located into the **org.springframework.security.web** package.

**Config - spring-security-config.jar**

This jar file is required for Spring Security configuration using XML and Java both. It includes Java configuration code and security namespace parsing code. All the classes and interfaces are stored in **org.springframework.security.config** package.

**LDAP - spring-security-ldap.jar**

This jar file is required only if we want to use LDAP (Lighweight Directory Access Protocol). It includes authentication and provisioning code. All the classes and interfaces are stored into **org.springframework.security.ldap** package.

**OAuth 2.0 Core - spring-security-oauth2-core.jar**

This jar is required to integrate Oauth 2.0 Authorization Framework and OpenID Connect Core 1.0 into the application. This jar file includes the core classes for OAuth 2.0 and classes are stored into the **org.springframework.security.oauth2.core** package.

**OAuth 2.0 Client - spring-security-oauth2-client.jar**

This jar file is required to get client support for OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0. This module provides OAuth login and OpenID client support. All the classes and interfaces are available from **org.springframework.security.oauth2.client** package.

**OAuth 2.0 JOSE - spring-security-oauth2-jose.jar**

It provides Spring Security's support for the JOSE (Javascript Object Signing and Encryption) framework. The JOSE framework provides methods to establish secure connection between clients. It contains following collection of specifications:

* JWT (JSON Web Token)
* JWS (JSON Web Signature)
* JWE (JSON Web Encryption)
* JWK (JSON Web Key)

All the classes and interfaces are available into these two packages:

**org.springframework.security.oauth2.jwt** and **org.springframework.security.oauth2.jose.**

ACL - spring-security-acl.jar

This jar is used to apply security to domain object in the application. We can access classes and code from the **org.springframework.security.acls** package.

**CAS - spring-security-cas.jar**

It is required for Spring Security?s CAS client integration. We can use it to integrate Spring Security web authentication with CAS single sign-on server. The source code is located into **org.springframework.security.cas** package.

**OpenID - spring-security-openid.jar**

This jar is used for OpenID web authentication support. We can use it to authenticate users against an external OpenID server. It requires OpenID4Java and top level package is **org.springframework.security.openid**.

**Test - spring-security-test.jar**

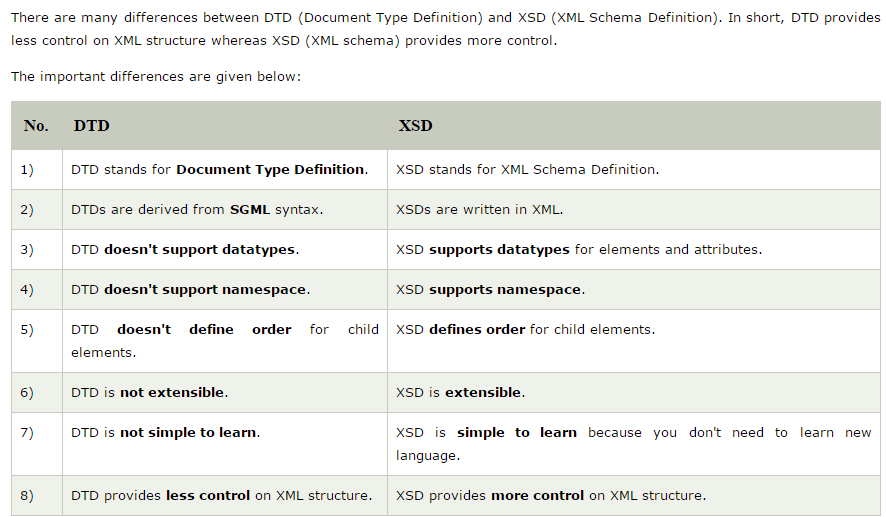
This jar provides support for testing Spring Security application.

==================================================

**Spring Question:**

**Topics Cover:** constructor and setter injection, Ioc container (beanFactory, ApplicationContext)

Q-1:



**Q-2: Difference between constructor and setter injection?**

There are many key differences between constructor injection and setter injection

**Partial dependency:** can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 argument constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.

Constructor Better for too many properties and Setter Better for few properties

**Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.

**Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

**Q: Why used Maven Project in Java?.**

* It is used for projects build, dependency and documentation.
* It simplifies the build process like ANT. But it is too much advanced than ANT.
* In short terms we can tell maven is a tool that can be used for building and managing any Java-based project.
* Maven is only a web deployment tool.

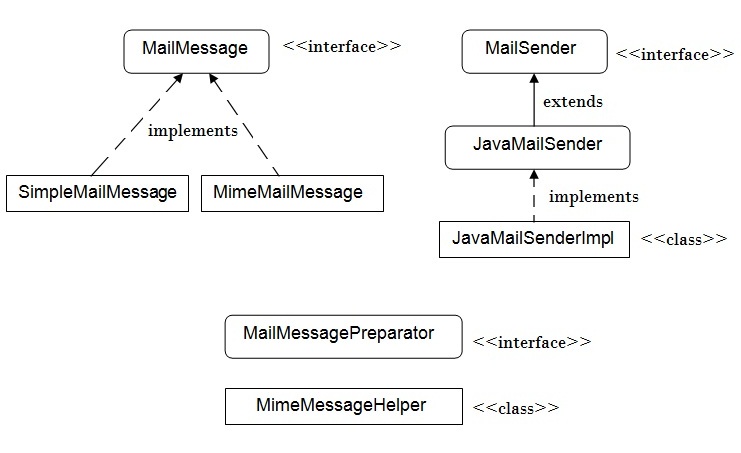
# Spring Java Mail Tutorial

Spring framework provides many useful interfaces and classes for sending and receiving mails.

The **org.springframework.mail** package is the root package that provides mail support in Spring framework.

## Spring Java Mail API

The interfaces and classes for java mail support in spring framework are as follows:



1. **MailSender interface**: It is the root interface. It provides basic functionality for sending simple mails.
2. **JavaMailSender interface**: It is the subinterface of MailSender. It supports MIME messages. It is mostly used with **MimeMessageHelper** class for the creation of JavaMail **MimeMessage**, with attachment etc. The spring framework recommends **MimeMessagePreparator**mechanism for using this interface.
3. **JavaMailSenderImpl class**: It provides the implementation of JavaMailSender interface. It supports JavaMail MimeMessages and Spring SimpleMailMessages.
4. **SimpleMailMessage class**: It is used to create a simple mail message including from, to, cc, subject and text messages.
5. **MimeMessagePreparator interface**: It is the callback interface for the preparation of JavaMail MIME messages.
6. **MimeMessageHelper class**: It is the helper class for creating a MIME message. It offers support for inline elements such as images, typical mail attachments and HTML text content.

Note:**MIME** (Multi-Purpose Internet Mail Extensions) is an extension of the original Internet e-mail protocol that lets people use the protocol to exchange different kinds of data files on the Internet: audio, video, images, application programs, and other kinds, as well as the ASCII text handled in the original protocol,

#### You need to load mail.jar and activation.jar files to run this example.

**Spring MVC with Hibernate**

**ContextLoaderListener:**

ContextLoaderListener is a Servlet listener that loads all the different configuration files (service layer configuration, persistence layer configuration etc) into single spring application context. This helps to split spring configurations across multiple XML files.

<listener>

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

</listener>

**RequestContextListener**

<listener-class> org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

==============================================================

**Note: If we want to remove xml configuration (web.xml & spring-servlet.xml) then we can create a class that extends** AbstractAnnotationConfigDispatcherServletInitializer class.

**Example:**

**package** com.ipssi.controller;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan({"com.ipssi"})

**public** **class** ConfigurationXml {

}

**package** com.ipssi.controller;

**import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

**public** **class** MyXmlConfigFile **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class<?>[] getRootConfigClasses() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**protected** Class<?>[] getServletConfigClasses() {

// **TODO** Auto-generated method stub

**return** **new** Class[] {ConfigurationXml.**class**};

}

@Override

**protected** String[] getServletMappings() {

// **TODO** Auto-generated method stub

**return** **new** String[] {"/"};

}

}

# Spring MVC Form Tag Library

The Spring MVC form tags are the configurable and reusable building blocks for a web page. These tags provide JSP, an easy way to develop, read and maintain.

The Spring MVC form tags can be seen as data binding-aware tags that can automatically set data to Java object/bean and also retrieve from it. Here, each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and easy to use.

**Configuration of Spring MVC Form Tag**

The form tag library comes under the spring-webmvc.jar. To enable the support for form tag library, it is required to reference some configuration. So, add the following directive at the beginning of the JSP page:

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

Pom.xml

**<dependency>**

**<groupId>**org.springframework**</groupId>**

**<artifactId>**spring-webmvc**</artifactId>**

**<version>**5.1.1.RELEASE**</version>**

**</dependency>**

**List of Spring MVC Form Tags**

|  |  |
| --- | --- |
| form:form | It is a container tag that contains all other form tags. |
| form:input | This tag is used to generate the text field. |
| form:radiobutton | This tag is used to generate the radio buttons. |
| form:checkbox | This tag is used to generate the checkboxes. |
| form:password | This tag is used to generate the password input field. |
| form:select | This tag is used to generate the drop-down list. |
| form:textarea | This tag is used to generate the multi-line text field. |
| form:hidden | This tag is used to generate the hidden input field. |

**The form tag**

The Spring MVC form tag is a container tag. It is a parent tag that contains all the other tags of the tag library. This tag generates an HTML form tag and exposes a binding path to the inner tags for binding.

**<form:form** action="nextFormPath" modelAttribute=?abc**?>**

**Spring MVC Form Text Field**

The Spring MVC form text field tag generates an HTML input tag using the bound value. By default, the type of the input tag is text.

**<form:input** path="name" **/>**

Here, **path** attribute binds the form field to the bean property.

The Spring MVC form tag library also provides other input types such as email, date, tel, etc.

===================================================================================================

=======================================================

**Spring Mvc Hibernate Configuration:**

**dataSource** bean is defined for org.apache.commons.dbcp.BasicDataSource class for basic connection pooling.

org.springframework.orm.hibernate4.LocalSessionFactoryBean bean is used for Hibernate 4 SessionFactory. For Hibernate 3, you will find similar classes as org.springframework.orm.hibernate3.LocalSessionFactoryBean and org.springframework.orm.hibernate3.AnnotationSessionFactoryBean.

One important point is that when we are depending on Spring framework for Hibernate Session management, we should not define hibernate.current\_session\_context\_class, otherwise, you will get a lot of session transaction-related issues.

**personDAO** and **personService** beans are self understood.

**transactionManager** bean definition for org.springframework.orm.hibernate4.HibernateTransactionManager is required for Spring ORM to support hibernate session transaction management. For Hibernate 3, you will find similar class as org.springframework.orm.hibernate3.HibernateTransactionManager. Spring uses AOP for transaction management, you can now relate it with @Transactional annotation.

**Q: Difference b/w <context:annotation-config/> and <mvc:annotation-driven>?**

Ans <https://javabeat.net/spring-mvc-component-scan-annotations/>

<https://howtodoinjava.com/spring-mvc/spring-mvc-difference-between-contextannotation-config-vs-contextcomponent-scan/>

**Maven Dependency Description:**

1. **<dependency>common-dbcp</dependency> For database Connection pool**
2. **<dependency>Spring-orm </dependency>**
3. **<dependency>jstl</dependency> jsp standard tag library**

**Q. What is Stereotype annotations**  and **Stereotype annotations**  in a Spring

Spring @Component, @Repository, @Service and @Controller **Stereotype Annotations**. ... **Stereotype annotations** are markers for any class that fulfills a role within an application. This helps remove, or at least greatly reduce, the Spring XML configuration required for these components

### -Spring Annotations Descriptions (<https://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial>)

Moreover, *@GetMapping*, *@PostMapping*, *@PutMapping*, *@DeleteMapping*, and *@PatchMapping* are different variants of *@RequestMapping*with the HTTP method already set to GET, POST, PUT, DELETE, and PATCH respectively.

These are available since Spring 4.3 release.

1. [**Spring @Bean**](https://www.journaldev.com/21577/spring-bean-annotation): Spring @Bean Annotation is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods.
2. [**Spring @Service**](https://www.journaldev.com/21435/spring-service-annotation):Spring @Service annotation is a specialization of @Component annotation. Spring Service annotation can be applied only to classes. It is used to mark the class as a service provider.
3. [**Spring @Component**](https://www.journaldev.com/21429/spring-component): Spring Component annotation is used to denote a class as Component. It means that the Spring framework will autodetect these classes for dependency injection when annotation-based configuration and classpath scanning is used.
4. [**Spring @RestController**](https://www.journaldev.com/21536/spring-restcontroller): Spring RestController annotation is a convenience annotation that is itself annotated with @Controller and @ResponseBody. This annotation is applied to a class to mark it as a request handler.
5. [**Spring @Controller**](https://www.journaldev.com/21515/spring-controller-spring-mvc-controller): Spring Controller annotation is a specialization of @Component annotation. Spring Controller annotation is typically used in combination with annotated handler methods based on the RequestMapping annotation.
6. [**Spring @Repository**](https://www.journaldev.com/21460/spring-repository-annotation): Spring @Repository annotation is used to indicate that the class provides the mechanism for storage, retrieval, search, update and delete operation on objects.
7. [**Spring @Configuration**](https://www.journaldev.com/21033/spring-configuration-annotation): Spring @Configuration annotation is part of the spring core framework. Spring Configuration annotation indicates that the class has @Bean definition methods. So Spring container can process the class and generate Spring Beans to be used in the application.
8. [**Spring @Value**](https://www.journaldev.com/21448/spring-value-annotation): Spring @Value annotation is used to assign default values to variables and method arguments. We can read spring environment variables as well as system variables using @Value annotation.
9. [**Spring @PropertySource**](https://www.journaldev.com/21440/spring-propertysource): Spring @PropertySource annotation is used to provide properties file to Spring Environment. This annotation is used with @Configuration classes.
10. [**Spring @PostConstruct and @PreDestroy**](https://www.journaldev.com/21206/spring-postconstruct-predestroy): When we annotate a method in Spring Bean with @PostConstruct annotation, it gets executed after the spring bean is initialized.

When we annotate a Spring Bean method with PreDestroy annotation, it gets called when bean instance is getting removed from the context.

1. [**Spring @Async**](https://www.journaldev.com/20457/spring-async-annotation): Spring @Async annotation allows us to create asynchronous methods in spring. Let’s explore @Async in this tutorial on spring framework.

**(Again)**

@**Repository :** Spring @Repository Annotation. Spring @Repository annotation is used to indicate that the class provides the mechanism for storage, retrieval, search, update and delete operation on objects.

## @Component: This annotation serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.

**@ComponentScan(basePackages = "com.websystique.springmvc")** ComponentScan referes to package locations to find the associated beans.

**@Service** annotation is used in your service layer and annotates classes that perform service tasks, often you don't use it but in many case you use this annotation to represent a best practice. For example, you could directly call a DAO class to persist an object to your database but this is horrible. It is pretty good to call a service class that calls a DAO. This is a good thing to perform the separation of concerns pattern.

**@Controller** annotation is an annotation used in Spring MVC framework (the component of Spring Framework used to implement Web Application). The @Controller annotation indicates that a particular class serves the role of a controller. The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations.

So looking at the Spring MVC architecture you have a DispatcherServlet class (that you declare in your XML configuration) that represent a front controller that dispatch all the HTTP Request towards the appropriate controller classes (annotated by @Controller). This class perform the business logic (and can call the services) by its method. These classes (or its methods) are typically annotated also with **@RequestMapping** annotation that specify what HTTP Request is handled by the controller and by its method.

@RequestBody: the @RequestBody annotation maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object.

@PostMapping("/request")

public ResponseEntity postController(

  @RequestBody LoginForm loginForm) {

    exampleService.fakeAuthenticate(loginForm);

    return ResponseEntity.ok(HttpStatus.OK);

}

@ResponseBody: The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

@Controller

@RequestMapping("/post")

public class ExamplePostController {

    @Autowired

    ExampleService exampleService;

    @PostMapping("/response")

    @ResponseBody

    public ResponseTransfer postResponseController(

      @RequestBody LoginForm loginForm) {

        return new ResponseTransfer("Thanks For Posting!!!");

     }

}

{"text":"Thanks For Posting!!!"}

===================

**The @ModelAttribute**

1. @ModelAttribute can be used either as a method parameter or at the method level.
2. When the annotation is used at the method level it indicates the purpose of that method is to add one or more model attributes. Such methods support the same argument types as [@RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) methods but that cannot be mapped directly to requests.

Let's have a look at a quick example here to start understanding how this works:

|  |  |
| --- | --- |
|  | @ModelAttribute  public void addAttributes(Model model) {      model.addAttribute("msg", "Welcome to the Netherlands!"); |

}

1. **@ModelAttribute methods are invoked before the controller methods annotated with @RequestMapping are invoked.**
2. **As a Method Argument:** When used as a method argument, it indicates the argument should be retrieved from the model. When not present, it should be first instantiated and then added to the model and once present in the model, the arguments fields should be populated from all request parameters that have matching names.

|  |  |
| --- | --- |
|  | @RequestMapping(value = "/addEmployee", method = RequestMethod.POST)  public String submit(@ModelAttribute("employee") Employee employee) {      // Code that uses the employee object        return "employeeView";  } |

==============================

**@Qualifier annotation(**<https://www.baeldung.com/spring-qualifier-annotation>**)**

## Autowire Need for Disambiguation

The [@Autowired](https://www.baeldung.com/spring-autowire) annotation is a great way of making the need to inject a dependency in Spring explicit. And although it's useful, there are use cases for which this annotation alone isn't enough for Spring to understand which bean to inject.

By default, Spring resolves autowired entries by type.

**If more than one bean of the same type is available in the container, the framework will throw NoUniqueBeanDefinitionException**, indicating that more than one bean is available for autowiring.

Let's imagine a situation in which two possible candidates exist for Spring to inject as bean collaborators in a given instance:

|  |  |
| --- | --- |
|  | @Component("fooFormatter")  public class FooFormatter implements Formatter {        public String format() {          return "foo";      }  }    @Component("barFormatter")  public class BarFormatter implements Formatter {        public String format() {          return "bar";      }  }    @Component  public class FooService {      @Autowired      private Formatter formatter;  } |

If we try to load FooService into our context, the Spring framework will throw a NoUniqueBeanDefinitionException. This is because **Spring doesn't know which bean to inject**. To avoid this problem, there are several solutions. The @Qualifier annotation is one of them.

## 3. @Qualifier Annotation

By using the @Qualifier annotation, we can **eliminate the issue of which bean needs to be injected**.

Let's revisit our previous example and see how we solve the problem by including the @Qualifier annotation to indicate which bean we want to use:

|  |  |
| --- | --- |
|  | public class FooService {        @Autowired      @Qualifier("fooFormatter")      private Formatter formatter;  } |

By including the @Qualifier annotation together with the name of the specific implementation we want to use – in this example, Foo – we can avoid ambiguity when Spring finds multiple beans of the same type.

We need to take into consideration that the qualifier name to be used is the one declared in the @Component annotation.

Note that we could've also used the @Qualifier annotation on the Formatter implementing classes, instead of specifying the names in their @Component annotations, to obtain the same effect:

|  |  |
| --- | --- |
|  | @Component  @Qualifier("fooFormatter")  public class FooFormatter implements Formatter {      //...  }    @Component  @Qualifier("barFormatter")  public class BarFormatter implements Formatter {      //...  } |

## 4. @Qualifier vs @Primary

There's another annotation called [@Primary](https://www.baeldung.com/spring-primary) that we can use to decide which bean to inject when ambiguity is present regarding dependency injection.

This annotation **defines a preference when multiple beans of the same type are present**. The bean associated with the @Primary annotation will be used unless otherwise indicated.

Let's see an example:

|  |  |
| --- | --- |
|  | @Configuration  public class Config {        @Bean      public Employee johnEmployee() {          return new Employee("John");      }        @Bean      @Primary      public Employee tonyEmployee() {          return new Employee("Tony");      }  } |

In this example, both methods return the same Employee type. The bean that Spring will inject is the one returned by the method tonyEmployee. This is because it contains the @Primary annotation. This annotation is useful when we want to **specify which bean of a certain type should be injected by default**.

And in case we require the other bean at some injection point, we would need to specifically indicate it. We can do that via the @Qualifier annotation. For instance, we could specify that we want to use the bean returned by the johnEmployee method by using the @Qualifier annotation.

It's worth noting that **if both the @Qualifier and @Primary annotations are present, then the @Qualifier annotation will have precedence.** Basically, @Primary defines a default, while @Qualifier is very specific.

Let's see another way of using the @Primary annotation, this time using the initial example:

|  |  |
| --- | --- |
|  | @Component  @Primary  public class FooFormatter implements Formatter {      //...  }    @Component  public class BarFormatter implements Formatter {      //...  } |

**In this case, the @Primary annotation is placed in one of the implementing classes** and will disambiguate the scenario.

## 5. @Qualifier vs Autowiring by Name

Another way to decide between multiple beans when autowiring is by using the name of the field to inject. **This is the default in case there are no other hints for Spring**. Let's see some code based on our initial example:

|  |  |
| --- | --- |
|  | public class FooService {        @Autowired      private Formatter fooFormatter;  } |

In this case, Spring will determine that the bean to inject is the FooFormatter one since the field name is matched to the value that we used in the @Component annotation for that bean.

**////////// CommandLineRunner / ApplicationRunner//////////**

CommandLineRunner/ **ApplicationRunner** run() will get execute, just after applicationcontext is created and before springboot application starts up.  
It accepts the argument, which are passed at time of server startup.

## @Order Annotation Spring Boot

Whenever you have more than one class implementing the **CommandLineRunner / ApplicationRunner**, then you can use the **@Order** annotation mention which **run()** method

package com.javainterviewpoint;

import org.springframework.boot.CommandLineRunner;

import org.springframework.core.annotation.Order;

@Order(1)

public class NewHelloCommandLineRunner implements CommandLineRunner

{

@Override

public void run(String... args) throws Exception

{

System.out.println("New Hello Command Line Runner called");

}

}

package com.javainterviewpoint;

import org.springframework.boot.CommandLineRunner;

import org.springframework.core.annotation.Order;

@Order(2)

public class HelloCommandLineRunner implements CommandLineRunner

{

@Override

public void run(String... args) throws Exception

{

System.out.println("Old Hello Command Line Runner called");

}

}

=====================================================

@Component

**public** **class** AppStartupRunner implements ApplicationRunner {

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

@Override

**public** **void** run(ApplicationArguments args) throws Exception {

logger.info("Your application started with option names : {}", args.getOptionNames());

}

}

**//////////////////////Spring File Upload////////////////////**

### Spring MVC Multipart Configuration

To utilize Apache Commons FileUpload for handling multipart requests, all we need to do is configure multipartResolver bean with class as org.springframework.web.multipart.commons.CommonsMultipartResolver

servlet-context.xml

**<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>**

**Spring Question:** <https://www.javapedia.net/Spring-MVC-Interview-questions#qanda848>

**Difference between Spring Web MVC and Spring WebFlux.**

Spring Web MVC is the original web framework included with the spring framework as Servlet API and Servlet containers.

On the other hand, Spring Webflux, added in Version 5.0, is a reactive-stack web framework that provides fully non-blocking, supports reactive streams backpressure for web applications. Webflux also runs on servers like Netty, Undertow, and Servlet 3.1 + containers.

**Why Java web apps has dot(.) do extensions?**

do extension implies "do something". This extension convention is followed from struts1.

With the web application upgrades developers might have thought to retain and continue to use the do extension for backward compatibility.

**How to enable browser caching of static resources (JS, CSS) with Spring MVC?**

Using mvc:resources tag we can specify the cache period.

<mvc:resources mapping="/resources/\*\*" location="/WEB-INF/resources/"

cache-period="31556926"/>

Also mvc:interceptor can be used to specify the cache settings.

<mvc:interceptor>

<mvc:mapping path="/WEB-INF/resources/\*"/>

<bean id="webContentInterceptor"

class="org.springframework.web.servlet.mvc.WebContentInterceptor">

<property name="cacheSeconds" value="31556926"/>

<property name="useExpiresHeader" value="true"/>

<property name="useCacheControlHeader" value="true"/>

<property name="useCacheControlNoStore" value="true"/>

</bean>

</mvc:interceptor>

Q; JNDI:**(**apache-tomcat/conf Directory**)**

JNDI stands for **Java Naming and Directory Interface**. It comes standard with J2EE. What is its basic use? With this API, you can access many types of data, like objects, devices, files of naming and directory services, eg. it is used by EJB to find remote objects. Define in:

**Server.xml and context.xml And Application context.xml**

**How JNDI lookup differs between Tomcat and weblogic server?**

In Weblogic, the JNDI lookup for "dsJDBC" is just "dsJDBC" whilst in Tomcat, it accepts only the the formal format "java:comp/env/jdbc/dsJDBC".

In Tomcat, JndiLocatorSupport has a property resourceRef. When setting this true, "java:comp/env/" prefix will be prepended automatically.

**What is resource-ref in web.xml used for?**

It resource reference to an object factory for resources such as a JDBC DataSource, a JavaMail Session, or custom object factories configured into Tomcat or any other application server

**Use of BindingResult interface in Spring MVC.**

Use a BindingResult object as an argument to the validate method of a Validator inside a Controller and the BindingResult object will hold the validation errors.

validator.validate(modelObject, bindingResult);

if (bindingResult.hasErrors()) {

// Handle error

}

**Difference between Spring MVC and Spring Web Flow.**

Spring MVC is the standard solution for MVC by Spring for your web applications and it comprises of a view(your JSP page) , a controller class (annotated with @Controller ) and a model which represents your data.

Spring Webflow is built on top of Spring MVC and differs in purpose although it inherits most of the features of Spring MVC. Spring webflow is more suited for wizard/workflow style applications where you enter data on a form and that has to pass through a series of validations and capture number of additional data on different pages before you complete your business process

**Is transaction managed at DAO or Service layer in Spring?**

At service layer.

**How do I configure Spring MVC view in @Configuration class without spring XML?**

Create a bean in @Configuration class for UrlBasedViewResolver. This bean has different methods like setPrefix, setSuffix and setViewClass.

@Bean

public UrlBasedViewResolver setupViewResolver() {

UrlBasedViewResolver resolver = new UrlBasedViewResolver();

resolver.setPrefix("/views/");

resolver.setSuffix(".jsp");

resolver.setViewClass(JstlView.class);

return resolver;

}

**How do I configure DispatcherServlet without using web.xml in Spring MVC?**

Create a class implementing WebApplicationInitializer interface and implement onStartup() method. In this method we can register all the annotation based application configuration classes, servlet and its mappings, listener etc including DispatcherServlet.

public class WebAppInitializer implements WebApplicationInitializer {

public void onStartup(ServletContext servletContext) throws ServletException {

AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();

ctx.register(ApplicationConfig.class);

ctx.setServletContext(servletContext);

Dynamic dynamic = servletContext.addServlet("dispatcher", new DispatcherServlet(ctx));

dynamic.addMapping("/");

dynamic.setLoadOnStartup(1);

}

}

**How to start Spring MVC using spring boot?**

Spring provides spring-boot-starter-web that can resolve all Spring MVC required JAR. In our project, we can include it using maven as dependency.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<version>1.2.2.RELEASE</version>

</dependency>

**Advantage of using HandlerInterceptorAdapter in Spring MVC framework.**

Implementing HandlerInterceptor interface forces to implement all the three methods preHandle(), postHandle() and afterCompletion() irrespective of whether it is needed or not. To avoid that, you can use HandlerInterceptorAdapter class that implements HandlerInterceptor and provides default empty implementations. Extending HandlerInterceptorAdapter class allows you to override only the method that are required.

**Explain Spring MVC Interceptor.**

HandlerInterceptor interface can be used in spring mvc application to pre-handle and post-handle web requests that are handled by Spring MVC controllers. These handlers are mostly used to manipulate the model attributes returned/submitted before it is passed to the views/controllers.

A handler interceptor can be registered for particular URL mappings so that it only intercepts requests mapped to certain URLs. Each handler interceptor must implement the HandlerInterceptor interface, which contains three callback methods for you to implement: preHandle(), postHandle() and afterCompletion().

**What is the front controller class of Spring MVC?**

A front controller is defined as a controller that handles all requests for a Web Application. DispatcherServlet servlet is the front controller in Spring MVC that intercepts every request and then dispatches requests to an appropriate controller. When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring?s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.

Explain MultipartResolver in Spring framework?

Spring provides MultipartResolver to handle the file upload process in web application. There are two concrete implementations included in Spring.

**CommonsMultipartResolver for Jakarta Commons FileUpload.**

StandardServletMultipartResolver for Servlet 3.0 Part API.

To define an implementation, create a bean with the id 'multipartResolver' in DispatcherServlet application context. Such a resolver gets applied to all requests handled by that DispatcherServlet. If a DispatcherServlet detects a multipart request, it will resolve it via the configured MultipartResolver and pass a wrapped HttpServletRequest. Controllers can then cast their given request to the MultipartHttpServletRequest interface, which permits access to any MultipartFiles.

**How does Spring MVC provide validation support?**

Spring supports validations primarily into two ways.

Using JSR-303 Annotations and any reference implementation, for example, Hibernate Validator.

Using custom implementation of org.springframework.validation. Validator interface.

**What is Spring MVC framework?**

The Spring web MVC framework facilitates model-view-controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.

The MVC pattern separates the different aspects of an application (for example, input logic, business logic, and UI logic) and also enables loose coupling between model, view and controller within the application.

**Advantages of Spring MVC framework over other MVC framework.**

Clear separation of roles. - Model, controller, validator, command object, form object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled/extended by a specialized object.

* Powerful and straightforward configuration of framework and application API as JavaBeans.
* Reusable business code. You can use existing business objects as command or form objects instead of mirroring them in order to extend a particular framework base class.
* Customizable binding and validation.
* Customizable handler mapping and view resolution.
* Convenient locale and theme resolution.
* A JSP form tag library, introduced in Spring 2.0, makes it easier to write forms in JSP pages.
* support for different response types; generate XML, JSON, Atom etc.
* Flexible data binding: on a type mismatch, it is displayed as a validation error on the screen. Any plain Business POJO can be directly configured as form-backing object.

**Explain the request flow and its lifecycle in Spring MVC.**

The request will be received by DispatcherServlet as the first step.

DispatcherServlet gets the help of HandlerMapping and maps the @Controller class associated with the given request to delegate the request.

So request gets transferred to the @Controller, and then @Controller will process the request by executing appropriate methods and returns ModeAndView object (contains both Model data and View name) back to the DispatcherServlet.

Now DispatcherServlet send the model object to the ViewResolver to resolve and retrieve the actual view page.

DispatcherServlet will pass the Model object to the View page to display the result and create the Response.

Finally DispatcherServlet sends the response back to the browser.

**What is DispatcherServlet in Spring MVC Framework?**

Spring web MVC framework is request-driven, flows through the central Servlet, DispatcherServlet that handles all the HTTP requests and responses. Spring's DispatcherServlet is completely integrated with the Spring IoC container so it allows you to use every feature that Spring has along with Request handling.

After receiving an HTTP request, DispatcherServlet gets helps from the HandlerMapping (configuration files) to resolve and call the appropriate Controller. The Controller takes the request and calls the appropriate service methods and set model data and then returns view name to the DispatcherServlet. The DispatcherServlet will take help from ViewResolver to resolve the defined view for the request. Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

<web-app>

<display-name>SpringMVC Demo App</display-name>

<servlet>

<servlet-name>springMVC</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springMVC</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

By default, DispatcherServlet resolves its configuration file using <servlet\_name>-servlet.xml. For example, with the above web.xml file, DispatcherServlet will look for springMVC-servlet.xml file in classpath.

DispatcherServlet is the front controller in Spring MVC that intercepts every requests and then dispatches/forwards it to an appropriate controller.

**Spring MVC Framework: What is ContextLoaderListener?**

ContextLoaderListener reads the spring configuration file specified using 'contextConfigLocation' in web.xml, parses it and loads the beans defined in that config file to the ApplicationContext.

<servlet>

<servlet-name>springMVC</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

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

<param-value>/WEB-INF/applicationContext.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

The ContextLoaderListener has purposes like,

binding the lifecycle of the ApplicationContext with that of lifecycle of the ServletContext.

creates the ApplicationContext and WebApplicationContext.

**What is the difference between ApplicationContext and WebApplicationContext in Spring MVC?**

Web Application context extends Application Context which is designed to work with the standard javax.servlet.ServletContext to communicate with the container.

Beans that are instantiated in WebApplicationContext will also be able to use ServletContext if they implement ServletContextAware interface.

**Can we have multiple Spring configuration files in Spring MVC?**

Yes, we can have more than one spring context files. The following are the 2 ways to configure multiple context files.

1. Specify the list of files in web.xml file using contextConfigLocation init parameter.

<servlet-name>springMVC</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

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

<param-value>

WEB-INF/spring-core-context.xml,

WEB-INF/spring-DAO.xml,

WEB-INF/spring-Services.xml

</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springMVC</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

2. Import the configuration files into a existing configuration file that is already configured.

<beans>

<import resource="spring-core-context.xml"/>

<import resource="spring-DAO.xml"/>

<import resource="spring-services.xml"/>

... //other configurations

</beans>

**What are the Spring MVC Annotations?**

The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring loads it into the application context.

The @Repository annotation is a specialization of the @Component with similar behavior and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from DAO methods) eligible for translation into Spring DataAccessException.

**What is a Controller in Spring MVC framework?**

A Controller class is responsible to handle different kind of client requests based on the request mappings. We can create a controller class by using @Controller annotation. Usually it?s used with @RequestMapping annotation to define handler methods for specific URI mapping.

**What is the default scope of Spring MVC controllers?**

Spring MVC controllers are singleton by default and any controller object variable/field will be shared across all the requests and sessions.

If the object variable should not be shared across requests, one can use @Scope("request") annotation above your controller class definition to create instance per request.

Sping MVC - pass model between controllers.

avoid redirect instead use forward.

**Difference Between @RequestParam and @PathVariable in Spring MVC.**

@RequestParam and @PathVariable annotations are used for accessing the values from the request.

The primary difference between @RequestParam and @PathVariable is that @RequestParam used for accessing the values of the query parameters where as @PathVariable used for accessing the values from the URI template.

**Explain Interceptors in Spring MVC framework.**

Handler interceptors are required when specific functionality need to be applied to certain requests. Handler Interceptors should implement the interface HandlerInterceptor and override the below methods.

**preHandle** is called before the actual handler is executed;

**postHandle** is called after the handler is executed;

**afterCompletion** is called after the request is complete.

**What is view Resolver pattern and how does it work in Spring MVC?**

View Resolver pattern is a J2EE pattern that enables a web application to dynamically select its view technology. For example, HTML, JSP, Tapestry, JSF, XSLT etc. In this pattern, View resolver holds mapping of different views, controller return name of the view, which is then delegated to the View Resolver for selecting an appropriate view. Spring MVC framework supplies inbuilt view resolver for selecting views.

**Advantages of Spring MVC over Struts.**

Spring provides an integrated framework for all tiers of your application.

Spring provides a very clean separation between controllers, JavaBean models, and views.

Spring Controllers are configured using IoC like any other objects. This makes them easy to test and integrated with other objects managed by Spring.

Spring MVC web tiers are typically easier to test than Struts web tiers, due to the elimination forced concrete inheritance and explicit dependence of controllers on the dispatcher servlet.

Spring MVC is entirely based on interfaces that makes it very flexible unlike Struts, which forces your Action and Form objects into concrete inheritance.

Spring MVC is truly view-agnostic. you may choose view as JSP or Velocity, XLST or other view technologies.Developers can easily implement the Spring View interface to create a custom view mechanism.

Spring provides interceptors as well as controllers, making it easy to factor behaviors common to the handling of many requests.

**What is ModelAndView in Spring MVC Framework?**

ModelAndView is an object that holds both the model and view. The handler returns the ModelAndView object and DispatcherServlet resolves the view using View Resolvers and View.

The View is an object which contains view name in the form of the String and model is a map to add multiple objects.

In the below example 'employeeDetails' is the view name.

ModelAndView model = new ModelAndView("employeeDetails");

model.addObject("employeeObj", new EmployeeBean(123));

model.addObject("msg", "Employee information.");

return model;

**What are the different Controller implementations in Spring MVC framework?**

Controller,

AbstractCommandController,

SimpleFormController,

WizardFormController,

and MultiActionController.

**What is a MultipartResolver in Spring MVC framework?**

MultipartResolver interface is used for uploading files; CommonsMultipartResolver and StandardServletMultipartResolver are 2 implementations provided by spring framework to facilitate file uploading. By default there are no multipart resolvers configured, to enable it define a bean named 'multipartResolver' with type as MultipartResolver in spring bean configurations.

Once configured, any multipart request will be resolved by the configured MultipartResolver and pass on a wrapped HttpServletRequest. Then it is used in a controller class to get the file and process it.

**How do I handle exceptions in Spring MVC Framework?**

Spring MVC Framework provides the following methods to achieve robust exception handling.

Controller Based- define the exception handler methods in controller classes and annotate these methods with @ExceptionHandler annotation.

Global Exception Handler- Exception Handling is a cross-cutting concern and Spring provides @ControllerAdvice annotation that we can use with any class to define our global exception handler.

HandlerExceptionResolver implementation- Spring Framework provides HandlerExceptionResolver interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

**How do I handle views in Spring MVC using XML?**

To handle views in Spring MVC, we need to configure InternalResourceViewResolver bean in spring XML where we need to define prefix and suffix of the views name.

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/pages/"/>

<property name="suffix" value=".jsp"/>

</bean>

How to create Controller in Spring MVC?

To create a Controller in Spring MVC, create a class and annotate it with @Controller and @RequestMapping annotations. @Controller declares this class as a controller and @RequestMapping defines the path mapping of request url to the controller.

**Spring MVC:**

**How to access values from Model in JSP?**

Use JSTL, to retrieve values from Model as ,

${userId}.

**How to configure DispatcherServlet without web.xml in Spring MVC?**

Create a class implementing WebApplicationInitializer interface and override onStartup() method. In the method, we can register annotation based application configuration class, servlet and mappings, listener etc.

**What is the role of @EnableWebMvc in Spring MVC?**

Using @EnableWebMvc, spring enables the MVC related configuration.

@EnableWebMvc annotation is applied on configuration class with @Configuration annotation.

**How do I create a Spring MVC controller without a view?**

Set the controller method return type as void and mark the method with @ResponseBody annotation.

**How do I return a string from the Spring MVC controller without a view?**

Set the return type of the method as String and mark the method with @ResponseBody annotation.

@RequestMapping(value="/returnHelloWorld", method=GET)

@ResponseBody

public String returnHelloMethod() {

return "Hello world!";

}

**Explain @ResponseBody annotation in Spring MVC.**

When a controller method is marked with @ResponseBody, spring deserializes the returned value of the method and writes it directly to the Http Response automatically.

The return value of the method constitute the body of the HTTP response and not placed in a Model, or interpreted as a view name.

**Q: Explain @RequestBody annotation in Spring MVC.**

Spring automatically converts the content of the incoming request body to the parameter object when annotated with the @RequestBody annotation.

@ResponseBody @RequestMapping("/getUserInfo")

public String getUserInformation(@RequestBody UserDetails user){

return user.getFirstName() + " " + user.getLastName();

}

**Q: How do I quickly resolve MVC RequestMapping calling wrong controller method?**

Construct the base URI and then use path variables. If the issue persists, change the method from GET to POST or vice versa for one of the methods.

**Q: What does request.getParameter return when the parameter does not exist in Spring MVC/Servlet?**

The return type of the getParamter is String and it returns null if the parameter does not exist.

**Q: RestTemplate: How do you resolve SSLHandShakeException?**

Installing Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files in your java 7/8 installation is one of the ways to resolve SSL handshake failure.

**Q: How do I map the JSON field name to a different Java class property name?**

Using @JsonProperty("JSON propery name") we can configure and map POJO class property name with the JSON property.

**Q: How do I configure JNDI DataSource in Spring Web Application?**

//How to configure JNDI datasource for MySQL in Apache Tomcat with Spring Boot web application

To use the servlet container configured JNDI DataSource, we need to wire it in the spring bean configuration file and then inject it to spring beans as dependency. Once done, we can use it with JdbcTemplate to perform database operations.

< bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

< property name="jndiName" value="java:comp/env/jdbc/MySQLDB"/>

< /bean>

**Q: How to implement localization in Spring MVC applications?**

Spring framework provides LocaleResolver to support the internationalization and localization. To have your Spring MVC application support the internationalization, you will need to register two beans.

SessionLocaleResolver resolves locales by inspecting a predefined attribute in the user's session. If the session attribute does not exist, this locale resolver determines the default locale from the accept-language HTTP header.

LocaleChangeInterceptor interceptor detects if a special parameter is present in the current HTTP request. The parameter name can be customized with the paramName property of this interceptor. If such a parameter is present in the current request, this interceptor changes the user? locale according to the parameter value.

# Q:[What are the differences between Model, ModelMap, and ModelAndView?](https://stackoverflow.com/questions/18486660/what-are-the-differences-between-model-modelmap-and-modelandview)

**Model**: is an interface it contains four addAttribute and one mergeAttribute method.

**ModelMap**: implements Map interface. while ModelMap is a class .It also contains Map method.

**ModelAndView**: It allows a controller return both as a single value.ModelAndView is just a container for both a ModelMap and a view object. It allows a controller to return both as a single value.

**Spring Project Work**

1. **Spring MVC Project WebChat**

* **Create maven project**
* **Add jar dependencies (**<spring.version>4.3.9.RELEASE</spring.version>**)**
* **Spring Transaction jar (Spring context, Spring-tx, Spring-jdbc)**