Spring MVC  
[x](http://www.j2eebrain.com/spring-tutorial)

Content:

Spring Core

Aop,ioc,di

Spring JDBC

Spring MVC

**Difference between Model 1 and Model 2 architecture:**

**Features of MVC1:**

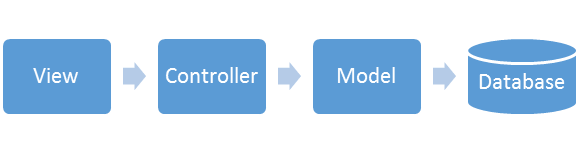
1. Html or jsp files are used to code the presentation. To retrieve the data JavaBean can be used.
2. In mvc1 archictecture all the view, control elements are implemented using Servlets or Jsp.
3. In MVC1 there is tight coupling between page and model as data access is usually done using Custom tag or through java bean call.

**Features of MVC2:**

1. The MVC2  architecture removes the page centric property of MVC1 architecture by separating Presentation, control logic and the application state.
2. In MVC2 architecture there is only one controller which receives all the request for the application and is responsible for taking appropriate action in response to each request.

**Architectueres**

The diagram is represented below:

[](https://www.guru99.com/images/jsp/022916_0452_JSPMVC1.png)

JSP Model Architeure.

MVC1: Model –Java Bean, View-JSP, FrontController –JSP

MVC2: Model –Java Bean, View-JSP, FrontController –Servlet

MVC3: Model –Java Bean, View-JSP, FrontController –Filter

MVC4: Model –Java Bean, View-JSP, ForntController –TagHandler

MVC2 Framework-struts 1.x,JSF 1.x,JSF2.x,Spring web MVC

MVC3 Framework-struts2.x

**J2EE arch:**

Presentation—Service Layer—Data Access Layer—datalayer

Jsp/html/jsf—Java Bean ,EJB,Spring Bean—JDBC,HB,Spring JDBC template –any DB

Data Access Layer—text,excel,dabs,collection,jdbc

JDBCTemplate

Java.util,Collection

Java.io.

Java.sql

Java Bean

SpringBean—webserver tomcat

EJB-enter Java Bean appli server

EJB—robust but complex to design, heavy weight, come with many services.

Spring modular in arch,we can inculde only need ed srvices so light weight in nature,no need of appl server

Servers:

Web Server(Spring) Application Server(EJB)

TomCat(75MB) GlassFish ,Jboss,Oracle logic Server(2GB)

Spring Web MVC templates :

* Jdbc template
* Rest template
* HB template
* JMS template
* OXM template

We can develop both standalone and web applications

**One Stop Shop Application**

**Advantages:**

* Supports various built in templates
* Provides a way to manage BL based in DI
* Both comprehensive & modular
* Test driven projects

Spring Architechture:

IOC and DI:

**IOC(Inversion Of Controller):**   Giving control to the container to get instance of object is called Inversion of Control., means instead of you are creating object using new operator, let the container do that for you.  
  
**DI(Dependency Injection):**  Way of injecting properties to an object is called Dependency injection.  
  
    We have three types of Dependency injection  
        1)  Constructor Injection  
        2)  Setter/Getter Injection  
        3)  Interface Injection  
Spring will support only Constructor Injection and Setter/Getter Injection.Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

4 major collection types are supported :

* List – <list/>
* Set – <set/>
* Map – <map/>
* Properties – <props/>

https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Demo: D:\ManishaSpring\CampusSpringDemo\IOCDI\src\com\syntel\p1

Demo:D:\ManishaSpring\CampusSpringDemo\IOCDI\src\com\syntel\p2

Demo:D:\ManishaSpring\CampusSpringDemo\IOCDI\src\com\syntel\Collection

IOC Container:

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. These objects are called Spring Beans



Following are the three important methods to provide configuration metadata to the Spring Container −

* XML based configuration file.
* Annotation-based configuration
* Java-based configuration

2 IOC container :

BeanFactory(I)

ApplicationContext(I) XMLBeanFactory(c)

WebApplicationContext(I) ClassPathXmlApplicationContext (c )AnnotaionConfigAppCon (c )

Spring provides following two distinct types of containers.

1. BeanFactory container
2. ApplicationContext container

**D:\ManishaSpring\CampusSpringDemo\Demo1\src\com\app\simple**

**ApplicationContext** container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This 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 the *BeanFactory*. BeanFactory can still be used for lightweight applications like mobile devices or applet based applications where data volume and speed is significant.

The most commonly used ApplicationContext implementations are:

1. [**FileSystemXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html) – This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
2. [**ClassPathXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) – This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
3. [**WebXmlApplicationContext**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/context/support/XmlWebApplicationContext.html) – This container loads the XML file with definitions of all beans from within a web application.

Spring Bean Eager vs Lazy Initialization Configurations

By default, Spring “application context” eagerly creates and initializes all ‘[singleton scoped](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/)‘ beans during application startup itself

But sometimes, you may need to mark some or all beans to be lazy initialized due to different project requirements.

Spring provides two easy ways to configure lazy initialization of beans based on which kind of configuration you are employing i.e. [XML based configuration](https://howtodoinjava.com/spring5/core/applicationcontext-xml-config-example/) or [java based configuration](https://howtodoinjava.com/spring5/core/spring-bean-container-java-configuration-example/).

**Lazy initialized beans in XML based configuration**

**i) Lazy load specific beans only**

To enable lazy loading for specific beans, use lazy-init="true" attribute on bean definitions in bean configuration xml files.

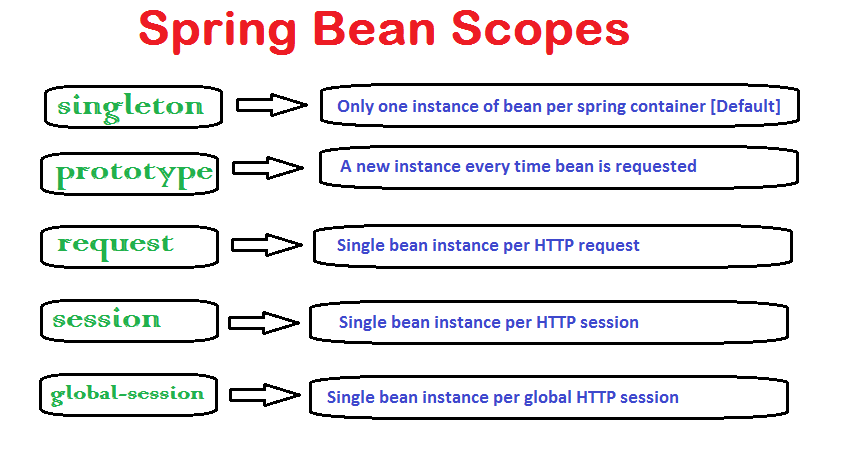
|  |
| --- |
| beans.xml |
| <beans>  <bean id="employeeManager" class="com.howtodoinjava.spring.service.impl.EmployeeManagerImpl"      lazy-init="true"/>    <beans> |

**ii) Lazy load all beans**

To enable lazy loading for all beans, use default-lazy-init="true" attribute on beans tag in bean configuration xml files.

|  |
| --- |
| beans.xml |
| <beans default-lazy-init="true">  <bean id="employeeManager" class="com.howtodoinjava.spring.service.impl.EmployeeManagerImpl" />  <beans> Lazy initialized beans in Java based configurationi) Lazy load specific beans only To lazy load only specific beans, use [@Lazy](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Lazy.html) annotation along with @Bean annotation in java config.   |  | | --- | | AppConfig.java | | import org.springframework.context.annotation.Lazy;   @Configuration  public class AppConfig {        @Lazy      @Bean      public EmployeeManager employeeManager() {          return new EmployeeManagerImpl();      }  } |  ii) Lazy load all beans To lazy load all beans, use @Lazy annotation along with @Bean annotation in java config.   |  | | --- | | AppConfig.java | | import org.springframework.context.annotation.Lazy;  @Lazy  @Configuration  public class AppConfig {        @Bean      public EmployeeManager employeeManager() {          return new EmployeeManagerImpl();      }    }  Generic:used when only 1 instance of bean is required if we create multiple instance then it will throw an error Noqulifying Bean….if we are having more than 1 bean then generic is not prefered .  Message m=c.getBean(Message.class);  m.getMessage();  D:\ManishaSpring\CampusSpringDemo\Demo1\src\com\syntel\app1  Example  <?xml version = "1.0" encoding = "UTF-8"?>  <beans xmlns = "http://www.springframework.org/schema/beans"  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation = "http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  <!-- A simple bean definition -->  <bean id = "..." class = "...">  <!-- collaborators and configuration for this bean go here -->  </bean>  <!-- A bean definition with lazy init set on -->  <bean id = "..." class = "..." lazy-init = "true">  <!-- collaborators and configuration for this bean go here -->  </bean>  <!-- A bean definition with initialization method -->  <bean id = "..." class = "..." init-method = "...">  <!-- collaborators and configuration for this bean go here -->  </bean>  <!-- A bean definition with destruction method -->  <bean id = "..." class = "..." destroy-method = "...">  <!-- collaborators and configuration for this bean go here -->  </bean>  <!-- more bean definitions go here --> | |

Scope : D:\ManishaSpring\CampusSpringDemo\Demo1\src\com\syntel\scope



#### . singleton(stateless)

This bean scope is default and it enforces the container to have only one instance per spring container irrespective of how much time you request for its instance. This [singleton](https://howtodoinjava.com/design-patterns/singleton-design-pattern-in-java/) behavior is maintained by bean factory itself.

#### 1.2. prototype(stateful)

This bean scope just reverses the behavior of singleton scope and produces a new instance each and every time a bean is requested.

Remaining three bean scopes are web applications related. Essentially these are available through web aware application context (e.g. WebApplicationContext). Global-session is a little different in sense that it is used when application is portlet based. In portlets, there will be many applications inside a big application and a bean with scope of ‘global-session’ will have only one instance for a global user session.

#### 1.3. request(SpringMVC)

With this bean scope, a new bean instance will be created for each web request made by client. As soon as request completes, bean will be out of scope and garbage collected.

#### 1.4. session(SpringMVC)

Just like request scope, this ensures one instance of bean per user session. As soon as user ends its session, bean is out of scope.

#### 1.5. global-session

It is something which is connected to Portlet applications. When your application works in Portlet container it is built of some amount of portlets. Each portlet has its own session, but if your want to store variables global for all portlets in your application than you should store them in global-session. This scope doesn’t have any special effect different from session scope in Servlet based applications.

singleton

This scopes the bean definition to a single instance per Spring IoC container (default).

prototype

This scopes a single bean definition to have any number of object instances.

request

This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.

session

This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

global-session

This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

## How to define spring bean scope

#### 2.1. XML bean configuration file

|  |
| --- |
| beans.xml |
| <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-2.5.xsd>">           <bean id="demoBean" class="com.howtodoinjava.application.web.DemoBean" scope="session" />    </beans> |

#### 2.2. Java configuration – @Scope annotation

|  |
| --- |
| DemoBean.java |
| @Service  @Scope("session")  public class DemoBean  {   //Some code  }  Bean Life Cycle:  **Types of bean life cycle events**  Spring bean factory is responsible for managing the life cycle of beans created through spring container. The life cycle of beans consist of **call back methods** which can be categorized broadly in two groups:   * **Post initialization** call back methods * **Pre destruction** call back methods   Spring Bean Life Cycle Spring Bean Life Cycle Callback Methods1.InitializingBean and DisposableBean callback interfaces The [org.springframework.beans.factory.InitializingBean](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/beans/factory/InitializingBean.html) interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container  D:\ManishaSpring\CampusSpringDemo\LifeCycleDemo\src\com\Syntel\Demo1  \***Aware interfaces for specific behavior** Custom init() and destroy() methods in bean configuration XML file The default init and destroy methods in bean configuration file can be defined in two ways:   * **Bean local definition** applicable to a single bean * **Global definition** applicable to all beans defined in beans context  2.3.1. Bean local definition Local definition is given as below.   |  | | --- | | beans.xml | | <beans>        <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"                      init-method="customInit"                      destroy-method="customDestroy"></bean>    </beans> |  2.3.2. Global definition Where as global definition is given as below. These methods will be invoked for all bean definitions given under <beans> tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.   |  | | --- | | <beans default-init-method="customInit" default-destroy-method="customDestroy">            <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"></bean>    </beans> |   Demo: D:\ManishaSpring\CampusSpringDemo\LifeCycleDemo\src\com\Syntel\Demo2 Spring Bean Life Cycle – @PostConstruct and @PreDestroy annotations Spring 2.5 onwards, you can use annotations also for specifying life cycle methods using @PostConstruct and @PreDestroy annotations.   * @PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before it’s instance is returned to requesting object. * @PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.   By default, Spring will not aware of the @PostConstruct and @PreDestroy annotation. To enable it, you have to either register ‘**CommonAnnotationBeanPostProcessor**‘ or specify the ‘**<context:annotation-config />**‘ in bean configuration file, 1. CommonAnnotationBeanPostProcessor <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-2.5.xsd">  <bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />  <bean id="customerService" class="com.mkyong.customer.services.CustomerService">  <property name="message" value="i'm property message" />  </bean>    </beans>  Copy 2. <context:annotation-config /> <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-2.5.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-2.5.xsd">  <context:annotation-config />  <bean id="customerService" class="com.mkyong.customer.services.CustomerService">  <property name="message" value="i'm property message" />  </bean>    </beans>  Demo: D:\ManishaSpring\CampusSpringDemo\LifeCycleDemo\src\com\Syntel\Demo2  Demo:D:\ManishaSpring\CampusSpringDemo\AnnotationDemo\src\com\Syntel\Demo3  Spring autowiring modes  <https://howtodoinjava.com/spring-core/spring-bean-life-cycle/>  <https://www.mkyong.com/spring/spring-postconstruct-and-predestroy-example/>  Spring - Beans Auto-Wiring:  The Spring container can **autowire** relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.  1) The default mode in traditional XML based configuration is no. 2) The default mode in java based @Autowired is byType.  Demo: D:\ManishaSpring\CampusSpringDemo\IOCDI\src\com\syntel\Autowiring(byname)  <https://howtodoinjava.com/spring-core/spring-bean-autowire-byname/>  <https://howtodoinjava.com/spring-core/spring-bean-autowire-bytype/>  <https://howtodoinjava.com/spring-core/spring-autowiring-by-constructor/>  https://howtodoinjava.com/spring-core/spring-autowire-by-autodetect/  Spring - Bean Definition Inheritance  A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.  A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.  Spring Bean definition inheritance has nothing to do with Java class inheritance but the inheritance concept is same. You can define a parent bean definition as a template and other child beans can inherit the required configuration from the parent bean.  When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent** attribute, specifying the parent bean as the value of this attribute.  D:\ManishaSpring\CampusSpringDemo\Demo1\src\com\syntel\Inheritance  Spring - Injecting Inner Beans  Java inner classes are defined within the scope of other classes, similarly, **inner beans** are beans that are defined within the scope of another bean. Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean  Demo: D:\ManishaSpring\CampusSpringDemo\IOCDI\src\com\syntel\InnerBean  Spring - Bean Post Processors  <https://www.tutorialspoint.com/spring/spring_bean_post_processors.htm>  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.  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.  DeDemo: D:\ManishaSpring\CampusSpringDemo\LifeCycleDemo\src\com\Syntel\BeanPostProcessor  D  Annotation: Starting from Spring 2.5 it became possible to configure the dependency injection using **annotations**. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.Annotation injection is performed before XML injection. Thus, the latter configuration will override the former for properties wired through both approaches.Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider the following configuration file in case you want to use any annotation in your Spring application.  **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.  **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context@Required  The @Required annotation applies to bean property setter methods.  @Autowired  The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.  @Qualifier  The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wire  The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context. To use this annotation, apply it over class as below: @Repository Annotation Although above use of @Component is good enough but we can use more suitable annotation that provides additional benefits specifically for DAOs i.e. @Repository annotation. The @Repository annotation is a specialization of the @Component annotation with similar use 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. 3. @Service Annotation The @Service annotation is also a specialization of the component annotation. It doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component in service-layer classes because **it specifies intent better**. Additionally, tool support and additional behavior might rely on it in the future. 4. @Controller Annotation @Controller annotation marks a class as a Spring Web MVC controller. It too is a @Component specialization, so beans marked with it are automatically imported into the DI container. When we add the @Controller annotation to a class, we can use another annotation i.e. @RequestMapping; to map URLs to instance methods of a class.  In **realtime usages**, we will face very rare situations where we will need to use @Component annotation. Most of the time, we will using @Repository, @Service and @Controller annotations. @Component should be used when the class does not fall into either of three categories i.e. controller, manager and dao.  D:\ManishaSpring\CampusSpringDemo\AnnotationDemo\src\com\Syntel\Demo1  D:\ManishaSpring\CampusSpringDemo\AnnotationDemo\src\com\Syntel\Demo2  D:\ManishaSpring\CampusSpringDemo\AnnotationDemo\src\com\Syntel\Demo3  D:\ManishaSpring\CampusSpringDemo\AnnotationDemo\src\com\Syntel\Demo4  https://www.journaldev.com/2623/spring-autowired-annotation  Spring - JDBC Framework  OverviewWhile working with the database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections, etc. However, Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.  JdbcTemplate Class  The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.  Instances of the JdbcTemplate class are threadsafe once configured. So you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs.  **AOP with Spring FrameworkPresentation Layer—**Service Layer—Data Access Layer—Data Layer  These **cross-cutting concerns** are conceptually separate from the application's business logic. There are various common good examples of aspects such as logging, auditing, declarative transactions, security, caching, etc.  Service layer (Business Logic)  Functional(core concern) Non –functional(cross cutting concern)  Withdraw,deposit login,security,transaction,scalibility  Before :non-fun are taken care by container.  AOP:to separate CCC form CC. we can use OOPs to do but it is static approach where as AOP inject the dependencies dynamically. AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations. It makes easy to maintain code in the present and future as well. You can add/remove concerns without recompiling complete sourcecode simply by changing configuration files (if you are applying aspects suing XML configuration).  The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect  **Aspect**  A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.  Target: piece of code that implements CC.  **Target object**  The object being advised by one or more aspects. This object will always be a proxied object. Also referred to as the advised object.  Advice: piece of code that implements CCC.  This is the actual action to be taken either before or after the method execution. This is the actual piece of code that is invoked during program execution by Spring AOP framework.  Before ,after ,after return,after throwing,around  This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.  Joint Point:it is well defined point CC where you want to inject CCC. Spring supports method call as jointpoint  Join point  This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.  This represents a point in your application where you can plug-in the AOP aspect.  PointCut: set of one or more jointpoints.  Aspect:Advice+Pointcut->What ,when , where  Spring FactoryBean  MVC Frame work  View -🡪Controller(POJO)🡪Model(Services,DAO,Business Methods ,all are POJO)  Controler(POJO)  Model  Business class  Services and DAO  View  MVC IOC container (WebApplicationContext)responsible to handle the Controller and Model  MVC provide support from view to controller, Model layer is executed by IOC container  Form  Features of MVC:  Bean  1.Form backup  Validation  2.MultiForm & MultiAction  3 built in validation support  4Internatinalization  5 Intercepters and tiles  6view Resolver  7 Exception Handling  MVC frame is designed by using  JSP model archi.  MVC1: Model –Java Bean, View-JSP,FrontController –JSP  MVC2: Model –Java Bean, View-JSP,FrontController –Servlet (Struts 1.x,Spring MVC,JSF FRAMEWORK)  MVC3: Model –Java Bean, View-JSP,FrontController –Filter (Struts2.x)  MVC4: Model –Java Bean, View-JSP,ForntController –TagHandler  Recommanded are MVC2,MVC3  MVC1,MVC4 are JSP not secure  Common functionalities such as validation,exception handling,tiles,internationalization are done by front controller.  Spring MVC the front controller is DispatherServlet.  JSF—faces servlet  MVC2—  Struts1.x—ActionServlet  Strust2.x—FilterDispather(StrutsPrepareAndExecuteFilter)  FrontControllerDesignPattern  MultipleInput map to single controller  In Spring Web MVC, **DispatcherServlet** class works as the front controller. It is responsible to manage the flow of the spring mvc application.  The Stero type annotation **@Controller** annotation is used to mark the class as the controller in Spring 3.or programmatically we can define controller by using ControllerClasses  1.ControlerInterface  2.AbstractController  3AbstractCommandController  4SimpleFormController  5 MultiActionController  6AbstractWizaedController  The **@RequestMapping** annotation is used to map the request url. It is applied on the method. Understanding the flow of Spring Web MVC As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller. The DispatcherServlet gets entry of handler mapping from the xml file and forwards the request to the controller. The controller returns an object of ModelAndView. The DispatcherServlet checks the entry of view resolver in the xml file and invokes the specified view component.  Tomcat supporturl pattrens-  complete charatcter sequence  \*.\* pattern(/\*)—for 1 frame use this  \*.extension pattern(for multiple controllers. For multiple MVC Framework for example Spring with struts )  MVC IOC Containers:  Spring WEB MVC:  Spring MVC is based on Model-View-Controller architecture. Below image shows Spring MVC architecture at a high level.  [spring mvc example, spring mvc architecture](https://cdn.journaldev.com/wp-content/uploads/2017/07/spring-mvc-architecture.png)  DispatcherServlet is the front controller class to take all requests and start processing them. We have to configure it in web.xml file. It’s job is to pass request to appropriate controller class and send the response back when view pages have rendered the response page.  HomeController.java will be the single controller class in our spring mvc example application.  home.jsp, user.jsp are the view pages in our spring mvc hello world example application.  User.java will be the only model class we will have in our spring mvc example web application.  Spring Web MVC and CRUD operation  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"suffix"* value=*".jsp"* />  <property name=*"prefix"* value=*"/WEB-INF/views/"*/>  </bean>    <bean id=*"ds"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"oracle.jdbc.driver.OracleDriver"*></property>  <property name=*"url"* value=*"jdbc:oracle:thin:@localhost:1521:xe"*></property>  <property name=*"username"* value=*"sys as sysdba"*></property>  <property name=*"password"* value=*"syntel123$"*></property>  </bean>    <bean id=*"jt"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"ds"*></property>  </bean>    <bean id=*"dao"* class=*"com.syntel.p1.EmployeeDao"*>  <property name=*"template"* ref=*"jt"*></property>  </bean>  <mvc:resources location=*"/WEB-INF/resources/"* mapping=*"/resourecs/\*\*"*/>  </beans>  Spring WITH Maven  D:\ManishaTesting\Testing\Maven\SpringDemo |
|  |
|  |
|  |