SPRING Framework

**What Is Spring Framework?**

Spring is a powerful lightweight application development framework used for Java Enterprise Edition (JEE). In a way, it is 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 using which you can solve many technical problems. You can say that, the Spring Framework is a comprehensive tool for supporting applications using Java programming language.

**Features Of Spring Framework**

* **Lightweight:**Spring Framework is lightweight with respect to size and transparency.
* **Inversion Of Control (IOC):**In Spring Framework, loose coupling is achieved using Inversion of Control. The objects give their own dependencies instead of creating or looking for dependent objects.
* **Aspect Oriented Programming (AOP):**By separating application business logic from system services, Spring Framework supports Aspect Oriented Programming and enables cohesive development.
* **Container:**Spring Framework creates and manages the life cycle and configuration of application objects.
* **MVC Framework:**Spring Framework is a MVC web application framework. This framework is configurable via interfaces and accommodates multiple view technologies.
* **Transaction Management:**For transaction management,Spring framework provides a generic abstraction layer. It is not tied to J2EE environments and it can be used in container-less environments.
* **JDBC Exception Handling:**The JDBC abstraction layer of the Spring Framework offers an exception hierarchy, which simplifies the error handling strategy.

***How does Spring fits for Enterprise world?***

Spring is the most popular application development framework for enterprise Java and spring framework is an open source java platform. It provides high performing, easily testable and reusable code. Spring handles the infrastructure as the underlying framework so that you can focus on your application. Spring is modular in design, thereby making creation, handling and linking of individual components so much easier. Spring implements Model View Container(MVC) design pattern. Spring is Framework of the Framework, Loosely Couple, Spring is Type Compatible Means, It can work with any other Frameworks and Technology.

**Why Spring is Used?**

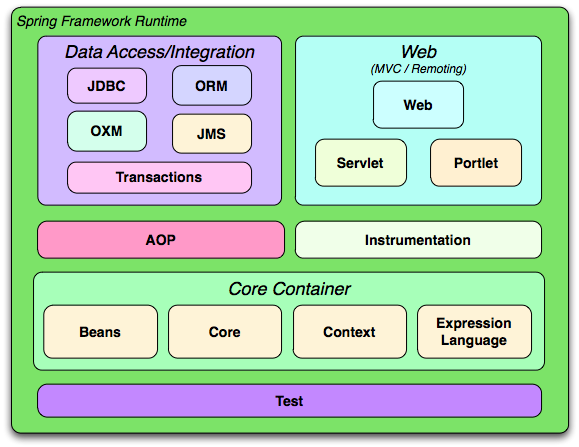
* Spring provides support to various frameworks like: Struts, Hibernate, Tapestry, EJB and JSF etc.
* In spring framework can be used in developing any java application, but mostly it use for building web.
* The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. on top of the java EE platform.
* Spring’s dependency injection approach encourages writing testable code.
* Easy to use but powerful database transaction management capabilities.
* Spring simplifies integration with other Java frameworks like JPA/Hibernate, ORM, Struts/JSF/etc. web frameworks.
* State of the art Web MVC framework for building web applications

**Features**

* **Spring Data:**Simplifies data access from relational and NoSQL data stores.
* **Spring Batch:**Provides powerful batch processing framework.
* **Spring Security:** Robust security framework to secure applications.
* **Spring Social:**Supports integration with social networking sites like Facebook, Twitter, LinkedIn, GitHub, etc.
* **Spring Integration:** An implementation of Enterprise Integration Patterns to facilitate integration with other enterprise applications using lightweight messaging and declarative adapters.

Spring Modules:

The Spring Framework contains a lot of features, which are well-organized in about twenty modules. These modules can be grouped together based on their primary features into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation and Test. These groups are shown in the diagram below.



Overview of the Spring Framework

### Core Container

* **Spring-beans-**This module provides the BeanFactory which is an implementation of the Factory design pattern. In fact the spring-beans module is what makes spring framework a container. It creates and manages the objects defined the spring bean configuration files.
* **Spring-core —**This module provides the dependency injection features.
* **Spring-context —**Based on the core and beans module the spring-context module provides further support for internationalization, event propagation and resource loading.
* **Spring-expression**— This module provides a powerful expression language to manipulate object graph at runtime

**AOP and Instrumentation**

AOP language is a **powerful** **tool** which allows developers to add enterprise functionality to the application such as transaction, security etc. It allows us to write less code and separate the code logic. AOP uses **cross-cutting concerns**.

**Instrumentation**

This module provides class instrumentation support and class loader implementations that are used in certain application servers.

* **Aspect:**An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management.
* **Join Point:**A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc
* **Advice:** Advices are actions taken for a particular join point.
* **Pointcut:**Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not.
* **Target Object:** They are the object on which advices are applied.
* **AOP proxy:**Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes.
* **Weaving:**It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

### Data Access/ Integration

* The spring-jdbc module provides an abstraction layer on JDBC to simplify common CRUD operations.
* The spring-tx module provides transaction management support.
* The spring-orm module provides classes to simplify integrating popular ORM (object relational mapping) framework/APIs such as Hibernate and JPA.
* The spring-jms module provides provides features for producing and consuming JMS.
* The spring-oxm module was introduced in 3.0 . It provides an abstraction layer on top of common xml parsers such as JAXB, Castor, JiBX etc.

**Web**

* The spring-web module provides basic web features such as initializing the IOC container using servlet listener and creating a web application context.
* The spring-webmvc is also known as the Web Servlet module. This module contains the spring MVC classes and REST web services implementation.
* The spring-websocket module provides support for socket based two way communication.
* The spring-webmvc-portlet module mirrors the webmvc module and tailors it to support JSR-168 portal development.

### Test

The Test module contains the Test Framework that supports testing .

Spring Core Concepts:

### Core Container:

The [Core Container](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03.html#beans-introduction) consists of the Core, Beans, Context and Expression modules.

The [Core and Beans](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03.html#beans-introduction) modules :

-provide the most fundamental parts of the framework and provides the IoC and Dependency Injection features. The basic concept is the BeanFactory which provides a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from actual program logic.

The [Context](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s13.html) module:

-build on the solid base provided by the [Core and Beans](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03.html#beans-introduction)modules: it provides a way to access objects in a framework-style manner in a fashion somewhat reminiscent of a JNDI-registry. The Context module inherits its features from the Beans module and adds support for internationalization (I18N) (using for example resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also contains support for some Java EE features like EJB, JMX and basic remoting support. The ApplicationContext interface is the focal point of the Context module that provides these features.

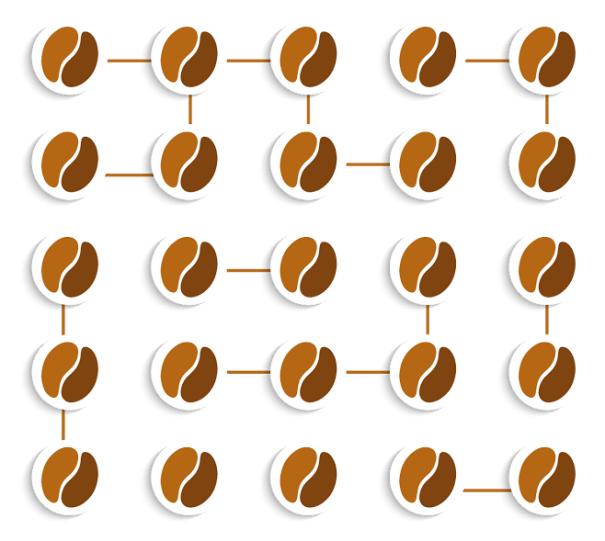
The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime. It can be seen as an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting of property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection, as well as common list aggregators.

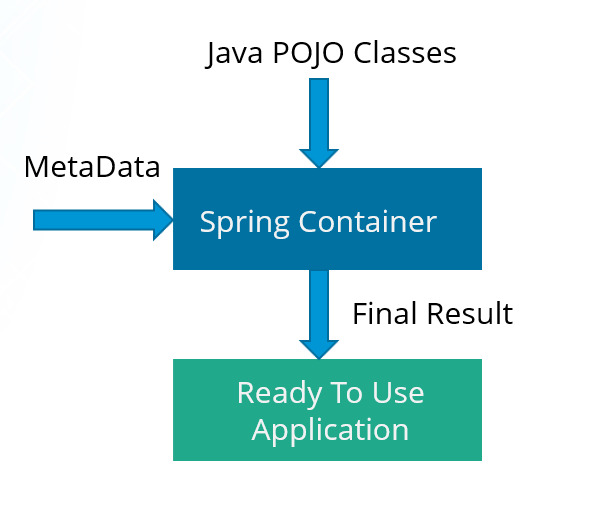
Introduction to IOC

IoC stands for Inversion of Control. It is the heart of the Spring Framework. The important tasks performed by the IoC container are:

1. Instantiating the bean
2. Wiring the beans together
3. Configuring the beans
4. Managing the bean’s entire life-cycle

 Here is a diagrammatic representation of how beans are wired together by IoC container.





The IoC container receives metadata from either an XML file, Java annotations, or Java code and works accordingly. IoC adds the flexibility and control of application, and provides a central place of configuration management for Plain Old Java Objects (POJO) of our application. This diagram represents an abstract view of the working of Spring Framework. It shows how Spring makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

There are two types of IoC containers:

1. BeanFactory
2. ApplicationContext

Lets discuss them in detail.

**BeanFactory**

* It is an interface defined inorg.springframework.beans.factory.**BeanFactory**.
* Bean Factory provides the basic support for Dependency Injection.
* It is based on factory design pattern which creates the beans of any type.
* BeanFactory follows lazy-initialization technique which means beans are loaded as soon as bean factory instance is created but the beans are created only when *getBean()* method is called.
* The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, you need to create the instance of XmlBeanFactory class as shown below:

|  |  |
| --- | --- |
| 1 | BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("beans.xml")); |

**ApplicationContext**

* It is an interface defined inorg.springframework.context.**ApplicationContext**.
* It is the advanced Spring container and is built on top of the BeanFactory interface.
* ApplicationContext supports the features supported by Bean Factory but also provides some additional functionalities.
* ApplicationContext follows eager-initialization technique which means instance of beans are created as soon as you create the instance of Application context.
* The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. You need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as shown below:

|  |  |
| --- | --- |
| 1 | ApplicationContext context=new ClassPathXmlApplicationContext("beans.xml"); |

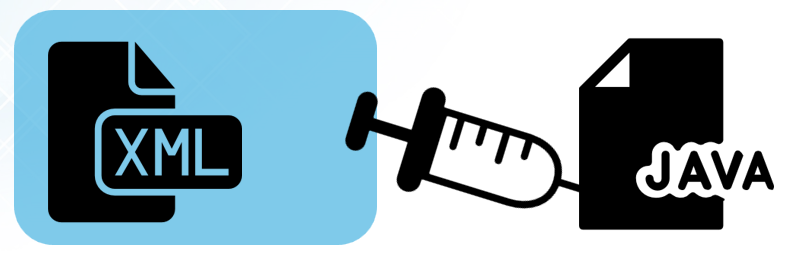
Inversion of Control (IoC) is a design principle (although, some people refer to it as a pattern). As the name suggests, it is used to invert different kinds of controls in object oriented design to achieve loose coupling. Here, the control means any additional responsibilities a class has other than its main responsibility, such as control over the flow of an application, control over the flow of an object creation or dependent object creation and binding.

IoC is all about inverting the control. To explain in layman's term, suppose you drive a car to your work place, it means you control the car. IoC principle suggests to invert the control, meaning instead of driving the car yourself, you hire a cab where another person will drive the car. Thus it is called inversion of the control from you to the cab driver. You don't have to drive a car yourself and let the driver do the driving so that you can focus on your main work.

IoC principle helps in designing loosely coupled classes which make them testable, maintainable and extensible.

**Dependency Injection**

Dependency Injection is also one of the **core concepts** of Spring Framework. It is a design pattern that removes the dependency from thecode. That is, the Spring Framework provides the dependencies of the class itself so that it can be easy to manage and test the application. You can provide information from external source such as XML file. Here, you do not create the objects instead you just define how they should be created and IoC container will create the objects for you.



In Spring, dependencies can be injected in two ways:

1. By constructor
2. By setter method

Lets see how they are done.

**By Constructor**

* The **<constructor-arg>** subelement of **<bean>** is used for constructor injection. e.g:

|  |  |
| --- | --- |
| 1 | <constructor-arg value="101" type="int"></constructor-arg> |

* By default when the Spring container loads the bean, it instantiates the bean with the default constructor. But you can also define a constructor argument in bean definition, using an argument constructor.

**By setter method**

* The **<property>** subelement of **<bean>** is used for setter injection.e.g:

|  |  |
| --- | --- |
| 1 | <property name="id" value="101"></property> |

* Setter-based Dependency Injection is accomplished by the container calling setter methods on your beans after invoking a no-argument.

# Difference between constructor and setter injection

1. [Difference between constructor and setter injection](https://www.javatpoint.com/difference-between-constructor-and-setter-injection)

There are many key differences between constructor injection and setter injection.

1. **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 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **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.

Spring core 3.0 annotations

The Java Programming language provided support for Annotations from Java 5.0. Leading Java frameworks were quick to adopt annotations and the Spring Framework started using annotations from the release 2.5. Due to the way they are defined, annotations provide a lot of context in their declaration.

Prior to annotations, the behavior of the Spring Framework was largely controlled through XML configuration. Today, the use of annotations provide us tremendous capabilities in how we configure the behaviours of the Spring Framework.

### @Autowired

This annotation is applied on fields, setter methods, and constructors. The@Autowired annotation injects object dependency implicitly.

When you use @Autowired on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

You can even use @Autowired  on private properties, as shown below. (This is a very poor practice though!)



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **public class Customer {**  **@Autowired**  **private Person person;**  **private int type;**  **}** |

When you use @Autowired on setter methods, Spring tries to perform the by Type autowiring on the method. You are instructing Spring that it should initiate this property using setter method where you can add your custom code, like initializing any other property with this property.



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **public class Customer {**  **private Person person;**  **@Autowired**  **public void setPerson (Person person) {**  **this.person=person;**  **}**  **}** |

Consider a scenario where you need instance of class A, but you do not store Ain the field of the class. You just use A to obtain instance of B, and you are storing B in this field. In this case setter method autowiring will better suite you. You will not have class level unused fields.

When you use @Autowired on a constructor, constructor injection happens at the time of object creation. It indicates the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the @Autowired annotation.



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8** | **@Component**  **public class Customer {**  **private Person person;**  **@Autowired**  **public Customer (Person person) {**  **this.person=person;**  **}**  **}** |

NOTE: As of Spring 4.3, @Autowired  became optional on classes with a single constructor. In the above example, Spring would still inject an instance of thePerson  class if you omitted the @Autowired  annotation.

## Spring Framework Stereotype Annotations

### @Component

This annotation is used on classes to indicate a Spring component. The@Component annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

### @Controller

The @Controller  annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

### @Service

This annotation is used on a class. The @Service marks a Java class that performs some service, such as execute business logic, perform calculations and call external APIs. This annotation is a specialized form of the @Component annotation intended to be used in the service layer.

### @Repository

This annotation is used on Java classes which directly access the database. The@Repository annotation works as marker for any class that fulfills the role of repository or Data Access Object.

This annotation has a automatic translation feature. For example, when an exception occurs in the @Repository there is a handler for that exception and there is no need to add a try catch block.

## Overview of new features

This is a list of new features for Spring 3.0. We will cover these features in more detail later in this section.

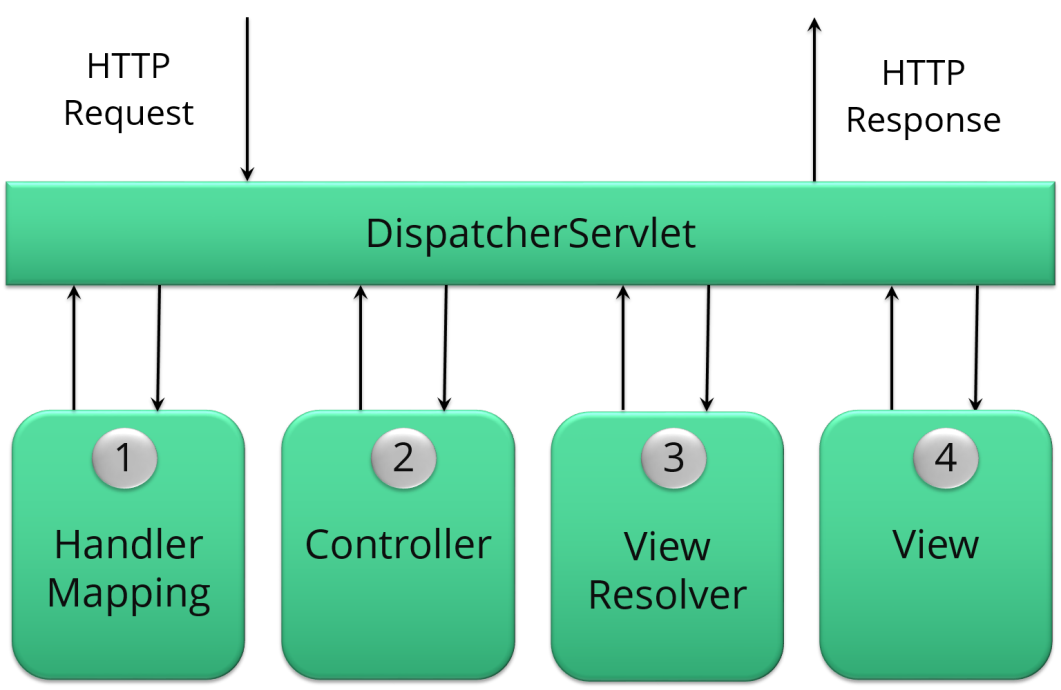
* Spring Expression Language
* IoC enhancements/Java based bean metadata
* General-purpose type conversion system and field formatting system
* Object to XML mapping functionality (OXM) moved from Spring Web Services project
* Comprehensive REST support
* @MVC additions
* Declarative model validation
* Early support for Java EE 6
* Embedded database support
* REST Webservices
* Java 5 support - Now Java 5 or above is needed to run Spring 3 based application
* Extended support for annotation-based components
* JSR 303 support has been added
* Java EE 6 support - JPA 2.0 and JSF 2.0 support
* JSR 330 support - Now the javax.inject annotations are supported
* Annotation-based formatting is now supported
* @MVC support
* Declarative model validation

### ****What do you mean by Spring MVC framework?****

The Spring web MVC framework provides model-view-controller architecture and ready to use components that are used to develop flexible and loosely coupled web applications. The MVC pattern helps in separating the different aspects of the application like input logic, business logic and UI logic, while providing a loose coupling between all these elements.

### ****DispatcherServlet.****

The DispatcherServlet is the core of Spring Web MVC framework. It handles all the HTTP requests and responses. The DispatcherServlet receives the entry of handler mapping from the configuration file and forwards the request to the controller. The controller then returns an object of Model And View. The DispatcherServlet checks the entry of view resolver in the configuration file and calls the specified view component.



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* −

* After receiving an HTTP request, *DispatcherServlet*consults the *HandlerMapping* to call the appropriate *Controller*.
* The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
* The *DispatcherServlet* will take help from *ViewResolver*to pickup 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.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext*w which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INFdirectory. In this case, our file will be **HelloWebservlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as *[servlet-name]-servlet.xml* and default location as *WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener *ContextLoaderListener* in your web.xml file as follows −

<web-app...>

<!-------- *DispatcherServlet* definition goes here----->

....

<context-param>

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

<param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

</web-app>

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory −

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:context = "http://www.springframework.org/schema/context"

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/context

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

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

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

<property name = "prefix" value = "/WEB-INF/jsp/" />

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

</bean>

</beans>

Following are the important points about **HelloWeb-servlet.xml** file −

* The *[servlet-name]-servlet.xml* file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
* The *<context:component-scan...>* tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
* The *InternalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at */WEB-INF/jsp/hello.jsp* .

The following section will show you how to create your actual components, i.e., Controller, Model, and View.

Defining a Controller

The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller**annotation indicates that a particular class serves the role of a controller. The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

@Controller

@RequestMapping("/hello")

public class HelloController {

@RequestMapping(method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation**@RequestMapping(method = RequestMethod.GET)** is used to declare theprintHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write the above controller in another form where you can add additional attributes in *@RequestMapping* as follows −

@Controller

public class HelloController {

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above −

* You will define required business logic inside a service method. You can call another method inside this method as per requirement.
* Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
* A defined service method can return a String, which contains the name of the **view**to be used to render the model. This example returns "hello" as logical view name

# Spring MVC Handler Mapping

**HandlerMapping** is an Interface to be implemented by objects that define a mapping between requests and handler objects. By default **DispatcherServlet**uses **BeanNameUrlHandlerMapping** and **DefaultAnnotationHandlerMapping**. In Spring we majorly use the below handler mappings

1. BeanNameUrlHandlerMapping
2. ControllerClassNameHandlerMapping
3. SimpleUrlHandlerMapping

### [Spring MVC BeanNameUrlHandlerMapping Example](http://www.javainterviewpoint.com/spring-mvc-beannameurlhandlermapping-example/)

In**BeanNameUrlHandlerMapping** in this article. Here we will be mapping each request to a Bean directly.

### [Spring MVC ControllerClassNameHandlerMapping Example](http://www.javainterviewpoint.com/spring-mvc-controllerclassnamehandlermapping-example/)

**ControllerClassNameHandlerMapping** uses a convention to map the requested URL to the Controller. It will take the **Controller name** and **converts them to** **lower case** with a **leading “/”**

### [Spring MVC SimpleUrlHandlerMapping Example](http://www.javainterviewpoint.com/spring-mvc-simpleurlhandlermapping-example/)

**SimpleUrlHandlerMapping**is the simplest of all handler mappings which allows you specify URL pattern and handler explicity

Validation

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called DataBinder to do exactly that. The Validator and the DataBinder make up the validation package, which is primarily used in but not limited to the MVC framework.

**Validation using Spring’s Validator interface**

Spring features a Validator interface that you can use to validate objects. The Validator interface works using an Errors object so that while validating, validators can report validation failures to the Errors object.

Let’s consider a small data object:

**public** **class** Person {

**private** String name;

**private** **int** age;

*// the usual getters and setters...*

}

We’re going to provide validation behavior for the Person class by implementing the following two methods of the org.springframework.validation.Validator interface:

* supports(Class) - Can this Validator validate instances of the supplied Class?
* validate(Object, org.springframework.validation.Errors) - validates the given object and in case of validation errors, registers those with the given Errors object

Implementing a Validator is fairly straightforward, especially when you know of the ValidationUtils helper class that the Spring Framework also provides.

**public** **class** PersonValidator **implements** Validator {

**/\*\***

**\* This Validator validates \*just\* Person instances**

**\*/**

**public** **boolean** supports(Class clazz) {

**return** Person.**class**.equals(clazz);

}

**public** **void** validate(Object obj, Errors e) {

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

Person p = (Person) obj;

**if** (p.getAge() < 0) {

e.rejectValue("age", "negativevalue");

} **else** **if** (p.getAge() > 110) {

e.rejectValue("age", "too.darn.old");

}

}

}

As you can see, the static rejectIfEmpty(..) method on the ValidationUtils class is used to reject the 'name' property if it is null or the empty string.

Handler interceptors

All HandlerMapping implementations supports handler interceptors that are useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Interceptors must implement HandlerInterceptor from theorg.springframework.web.servlet package with three methods that should provide enough flexibility to do all kinds of pre-processing and post-processing:

* preHandle(..) — before the actual handler is executed
* postHandle(..) — after the handler is executed
* afterCompletion(..) — after the complete request has finished

Spring **HandlerInterceptor** declares three methods based on where we want to intercept the HTTP request.

1. **boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**: This method is used to intercept the request before it’s handed over to the handler method. This method should return ‘true’ to let Spring know to process the request through another spring interceptor or to send it to handler method if there are no further spring interceptors.

If this method returns ‘false’ [Spring framework](https://www.journaldev.com/16922/spring-framework) assumes that request has been handled by the spring interceptor itself and no further processing is needed. We should use response object to send response to the client request in this case.

Object *handler* is the chosen handler object to handle the request. This method can throw Exception also, in that case [Spring MVC Exception Handling](https://www.journaldev.com/2651/spring-mvc-exception-handling-controlleradvice-exceptionhandler-handlerexceptionresolver) should be useful to send error page as response.

1. **void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)**: This HandlerInterceptor interceptor method is called when HandlerAdapter has invoked the handler but DispatcherServlet is yet to render the view. This method can be used to add additional attribute to the ModelAndView object to be used in the view pages. We can use this spring interceptor method to determine the time taken by handler method to process the client request.
2. **void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)**: This is a HandlerInterceptor callback method that is called once the handler is executed and view is rendered.

If there are multiple spring interceptors configured, *preHandle()* method is executed in the order of configuration whereas *postHandle()* and *afterCompletion()* methods are invoked in the reverse order.

Views

In Spring MVC, view resolvers enable you to render models in a browser without tying you to a specific view technology like JSP, Velocity, XML…etc.

There are two interfaces that are important to the way Spring handles views are **ViewResolver** and **View**. The ViewResolver provides a mapping between view names and actual views. The View interface addresses the preparation of the request and hands the request over to one of the view technologies.

Below are the important viewresolvers provided by spring framework:

1. **AbstractCachingViewResolver :** Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
2. **XmlViewResolver :**Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.
3. **ResourceBundleViewResolver :**Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.
4. **UrlBasedViewResolver :**Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
5. **InternalResourceViewResolver :**Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
6. **VelocityViewResolver/FreeMarkerViewResolver :**Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) or FreeMarkerView ,respectively, and custom subclasses of them.
7. **ContentNegotiatingViewResolver :**Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

### Spring's form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

Let's go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

#### Configuration

The form tag library comes bundled in spring-webmvc.jar. The library descriptor is called spring-form.tld.

To use the tags from this library, add the following directive to the top of your JSP page:

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

... where form is the tag name prefix you want to use for the tags from this library.

#### The form tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in thePageContext so that the command object can be accessed by inner tags. All the other tags in this library are nested tags of the *form* tag.

Let's assume we have a domain object called User. It is a JavaBean with properties such as firstName and lastName. We will use it as the form backing object of our form controller which returns form.jsp. Below is an example of what form.jspwould look like:

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName" /></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName" /></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form:form>

The firstName and lastName values are retrieved from the command object placed in the PageContext by the page controller. Keep reading to see more complex examples of how inner tags are used with the form tag.

The generated HTML looks like a standard form:

<form method="POST">

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value="Harry"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value="Potter"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form>

The preceding JSP assumes that the variable name of the form backing object is 'command'. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

<form:form commandName="user">

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName" /></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName" /></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form:form>

#### The input tag

This tag renders an HTML 'input' tag using the bound value and type='text' by default. For an example of this tag, see [the section called “The form tag”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/view.html#view-jsp-formtaglib-formtag). Starting with Spring 3.1 you can use other types such HTML5-specific types like 'email', 'tel', 'date', and others.

#### The checkbox tag

This tag renders an HTML 'input' tag with type 'checkbox'.

Let's assume our User has preferences such as newsletter subscription and a list of hobbies. Below is an example of the Preferences class:

**public** **class** Preferences {

**private** **boolean** receiveNewsletter;

**private** String[] interests;

**private** String favouriteWord;

**public** **boolean** isReceiveNewsletter() {

**return** receiveNewsletter;

}

**public** **void** setReceiveNewsletter(**boolean** receiveNewsletter) {

**this**.receiveNewsletter = receiveNewsletter;

}

**public** String[] getInterests() {

**return** interests;

}

**public** **void** setInterests(String[] interests) {

**this**.interests = interests;

}

**public** String getFavouriteWord() {

**return** favouriteWord;

}

**public** **void** setFavouriteWord(String favouriteWord) {

**this**.favouriteWord = favouriteWord;

}

}

The form.jsp would look like:

<form:form>

<table>

<tr>

<td>Subscribe to newsletter?:</td>

*<%-- Approach 1: Property is of type java.lang.Boolean --%>*

<td><form:checkbox path="preferences.receiveNewsletter"/></td>

</tr>

<tr>

<td>Interests:</td>

<td>

*<%-- Approach 2: Property is of an array or of type java.util.Collection --%>*

Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>

Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>

Defence Against the Dark Arts: <form:checkbox path="preferences.interests"

value="Defence Against the Dark Arts"/>

</td>

</tr>

<tr>

<td>Favourite Word:</td>

<td>

*<%-- Approach 3: Property is of type java.lang.Object --%>*

Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>

</td>

</tr>

</table>

</form:form>

There are 3 approaches to the checkbox tag which should meet all your checkbox needs.

* Approach One - When the bound value is of type java.lang.Boolean, the input(checkbox) is marked as 'checked' if the bound value is true. The value attribute corresponds to the resolved value of the setValue(Object) value property.
* Approach Two - When the bound value is of type array or java.util.Collection, the input(checkbox) is marked as 'checked' if the configured setValue(Object) value is present in the bound Collection.
* Approach Three - For any other bound value type, the input(checkbox) is marked as 'checked' if the configured setValue(Object) is equal to the bound value.

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

<tr>

<td>Interests:</td>

<td>

Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox"

value="Defence Against the Dark Arts"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

</td>

</tr>

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is notchecked, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for Spring form data binding to work. The checkbox tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("\_") for each checkbox. By doing this, you are effectively telling Spring that “ the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what ”.

#### The checkboxes tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

Building on the example from the previous checkbox tag section. Sometimes you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the checkboxes tag. You pass in an Array, a List or a Map containing the available options in the "items" property. Typically the bound property is a collection so it can hold multiple values selected by the user. Below is an example of the JSP using this tag:

<form:form>

<table>

<tr>

<td>Interests:</td>

<td>

*<%-- Property is of an array or of type java.util.Collection --%>*

<form:checkboxes path="preferences.interests" items="${interestList}"/>

</td>

</tr>

</table>

</form:form>

This example assumes that the "interestList" is a List available as a model attribute containing strings of the values to be selected from. In the case where you use a Map, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

#### The radiobutton tag

This tag renders an HTML 'input' tag with type 'radio'.

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

<tr>

<td>Sex:</td>

<td>Male: <form:radiobutton path="sex" value="M"/> <br/>

Female: <form:radiobutton path="sex" value="F"/> </td>

</tr>

#### The radiobuttons tag

This tag renders multiple HTML 'input' tags with type 'radio'.

Just like the checkboxes tag above, you might want to pass in the available options as a runtime variable. For this usage you would use the radiobuttons tag. You pass in an Array, a List or a Map containing the available options in the "items" property. In the case where you use a Map, the map entry key will be used as the value and the map entry's value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

<tr>

<td>Sex:</td>

<td><form:radiobuttons path="sex" items="${sexOptions}"/></td>

</tr>

#### The password tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

<tr>

<td>Password:</td>

<td>

<form:password path="password" />

</td>

</tr>

Please note that by default, the password value is not shown. If you do want the password value to be shown, then set the value of the 'showPassword' attribute to true, like so.

<tr>

<td>Password:</td>

<td>

<form:password path="password" value="^76525bvHGq" showPassword="true" />

</td>

</tr>

#### The select tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested optionand options tags.

Let's assume a User has a list of skills.

<tr>

<td>Skills:</td>

<td><form:select path="skills" items="${skills}"/></td>

</tr>

If the User's skill were in Herbology, the HTML source of the 'Skills' row would look like:

<tr>

<td>Skills:</td>

<td><select name="skills" multiple="true">

<option value="Potions">Potions</option>

<option value="Herbology" selected="selected">Herbology</option>

<option value="Quidditch">Quidditch</option></select>

</td>

</tr>

#### The option tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

<tr>

<td>House:</td>

<td>

<form:select path="house">

<form:option value="Gryffindor"/>

<form:option value="Hufflepuff"/>

<form:option value="Ravenclaw"/>

<form:option value="Slytherin"/>

</form:select>

</td>

</tr>

If the User's house was in Gryffindor, the HTML source of the 'House' row would look like:

<tr>

<td>House:</td>

<td>

<select name="house">

<option value="Gryffindor" selected="selected">Gryffindor</option>

<option value="Hufflepuff">Hufflepuff</option>

<option value="Ravenclaw">Ravenclaw</option>

<option value="Slytherin">Slytherin</option>

</select>

</td>

</tr>

#### The options tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

<tr>

<td>Country:</td>

<td>

<form:select path="country">

<form:option value="-" label="--Please Select"/>

<form:options items="${countryList}" itemValue="code" itemLabel="name"/>

</form:select>

</td>

</tr>

If the User lived in the UK, the HTML source of the 'Country' row would look like:

<tr>

<td>Country:</td>

<td>

<select name="country">

<option value="-">--Please Select</option>

<option value="AT">Austria</option>

<option value="UK" selected="selected">United Kingdom</option>

<option value="US">United States</option>

</select>

</td>

</tr>

As the example shows, the combined usage of an option tag with the options tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

The items attribute is typically populated with a collection or array of item objects. itemValue and itemLabel simply refer to bean properties of those item objects, if specified; otherwise, the item objects themselves will be stringified. Alternatively, you may specify a Map of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If itemValue and/or itemLabel happen to be specified as well, the item value property will apply to the map key and the item label property will apply to the map value.

#### The textarea tag

This tag renders an HTML 'textarea'.

<tr>

<td>Notes:</td>

<td><form:textarea path="notes" rows="3" cols="20" /></td>

<td><form:errors path="notes" /></td>

</tr>

#### The hidden tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML input tag with type 'hidden'.

<form:hidden path="house" />

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

<input name="house" type="hidden" value="Gryffindor"/>

#### The errors tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Let's assume we want to display all error messages for the firstName and lastName fields once we submit the form. We have a validator for instances of the User class called UserValidator.

**public** **class** UserValidator **implements** Validator {

**public** **boolean** supports(Class candidate) {

**return** User.**class**.isAssignableFrom(candidate);

}

**public** **void** validate(Object obj, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");

}

}

The form.jsp would look like:

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName" /></td>

*<%-- Show errors for firstName field --%>*

<td><form:errors path="firstName" /></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName" /></td>

*<%-- Show errors for lastName field --%>*

<td><form:errors path="lastName" /></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form:form>

If we submit a form with empty values in the firstName and lastName fields, this is what the HTML would look like:

<form method="POST">

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value=""/></td>

*<%-- Associated errors to firstName field displayed --%>*

<td><span name="firstName.errors">Field is required.</span></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value=""/></td>

*<%-- Associated errors to lastName field displayed --%>*

<td><span name="lastName.errors">Field is required.</span></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form>

What if we want to display the entire list of errors for a given page? The example below shows that the errors tag also supports some basic wildcarding functionality.

* path="\*" - displays all errors
* path="lastName" - displays all errors associated with the lastName field
* if path is omitted - object errors only are displayed

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

<form:form>

<form:errors path="\*" cssClass="errorBox" />

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName" /></td>

<td><form:errors path="firstName" /></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName" /></td>

<td><form:errors path="lastName" /></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form:form>

The HTML would look like:

<form method="POST">

<span name="\*.errors" class="errorBox">Field is required.<br/>Field is required.</span>

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value=""/></td>

<td><span name="firstName.errors">Field is required.</span></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value=""/></td>

<td><span name="lastName.errors">Field is required.</span></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes" />

</td>

</tr>

</form>

#### HTTP Method Conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that is has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring's HiddenHttpMethodFilter does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your web.xml, and a POST with a hidden \_method parameter will be converted into the corresponding HTTP method request.

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

<form:form method="delete">

<p class="submit"><input type="submit" value="Delete Pet"/></p>

</form:form>

This will actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the HiddenHttpMethodFilter, as defined in web.xml:

<filter>

<filter-name>httpMethodFilter</filter-name>

<filter-**class**>org.springframework.web.filter.HiddenHttpMethodFilter</filter-**class**>

</filter>

<filter-mapping>

<filter-name>httpMethodFilter</filter-name>

<servlet-name>petclinic</servlet-name>

</filter-mapping>

The corresponding @Controller method is shown below:

*@RequestMapping(method = RequestMethod.DELETE)*

**public** String deletePet(*@PathVariable* **int** ownerId, *@PathVariable* **int** petId) {

**this**.clinic.deletePet(petId);

**return** "redirect:/owners/" + ownerId;

}