Spring Life Cycle

**Bean Life Cycle**

1. The Bean Container finds the definition of the Spring Bean in the Configuration file.

2. The Bean Container creates an instance of the Bean using Java Reflection API.

3. **Using the dependency injection, spring populates all of the properties** as specified in the bean definition.

4. If the Bean class implements the **BeanNameAware** interface, then the **setBeanName**() method will be called by passing the name of the Bean.

5. If the Bean class implements the **BeanClassLoaderAware** interface, then the method **setBeanClassLoader**() method will be called by passing an instance of the ClassLoader object that loaded this bean.

6. If the Bean class implements the **BeanFactoryAware** interface, then the method **setBeanFactory**() will be called by passing an instance of BeanFactory object.

7. If there are any **BeanPostProcessors** object associated with the BeanFactory that loaded the Bean, then the method **postProcessBeforeInitialization**() will be called even before the properties for the Bean are set.

8. If the Bean class implements the **InitializingBean** interface, then the method **afterPropertiesSet**()will be called once all the Bean properties defined in the Configuration file are set.

9. If the Bean definition in the Configuration file contains a **'init-method'** attribute, then the value for the attribute will be resolved to a method name in the Bean class and that method will be called.

10. The **postProcessAfterInitialization**() method will be called if there are any Bean Post Processors attached for the Bean Factory object.

11. If the Bean class implements the **DisposableBean** interface, then the method **destroy**() will be called when the Application no longer needs the bean reference.

12. If the Bean definition in the Configuration file contains a **'destroy-method'** attribute, then the corresponding method definition in the Bean class will be called.

**Explain Bean lifecycle in Spring framework.**

<http://www.careerride.com/Spring-bean-lifecycle-in-spring-framework.aspx>

1. The spring container finds the bean’s definition from the XML file and instantiates the bean.  
2. Using the dependency injection, spring populates all of the properties as specified in the bean definition.  
3. If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean’s ID.  
4. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.  
5. If there are any BeanPostProcessors associated with the bean, their post- ProcessBeforeInitialization() methods will be called.  
6. If an init-method is specified for the bean, it will be called.  
7. Finally, if there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.

<http://www.javabeat.net/life-cycle-management-of-a-spring-bean/>

A ***Spring Bean*** represents a ***POJO component*** performing some useful operation. All ***Spring Beans*** reside within a ***Spring Container*** also known as ***IOC Container***. The Spring Framework is transparent and thereby hides most of the complex infrastructure and the communication that happens between the Spring Container and the Spring Beans. This section lists the sequence of activities that will take place between the time of Bean Instantiation and hand over of the Bean reference to the Client Application.

1. The Bean Container finds the definition of the Spring Bean in the Configuration file.
2. The Bean Container creates an instance of the Bean using Java Reflection API.
3. If any properties are mentioned, then they are also applied. If the property itself is a Bean, then it is resolved and set.
4. If the Bean class implements the BeanNameAware interface, then the setBeanName() method will be called by passing the name of the Bean.
5. If the Bean class implements the BeanClassLoaderAware interface, then the methodsetBeanClassLoader() method will be called by passing an instance of the ClassLoader object that loaded this bean.
6. If the Bean class implements the BeanFactoryAware interface, then the method setBeanFactory() will be called by passing an instance of BeanFactory object.
7. If there are any BeanPostProcessors object associated with the BeanFactory that loaded the Bean, then the method postProcessBeforeInitialization() will be called even before the properties for the Bean are set.
8. If the Bean class implements the InitializingBean interface, then the method afterPropertiesSet()will be called once all the Bean properties defined in the Configuration file are set.
9. If the Bean definition in the Configuration file contains a 'init-method' attribute, then the value for the attribute will be resolved to a method name in the Bean class and that method will be called.
10. The postProcessAfterInitialization() method will be called if there are any Bean Post Processors attached for the Bean Factory object.
11. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
12. If the Bean definition in the Configuration file contains a 'destroy-method' attribute, then the corresponding method definition in the Bean class will be called.

<http://mrbool.com/working-with-spring-bean-life-cycle/27173>

The following are the stages in a bean’s lifecycle.

1. Instantiate - The Spring container instantiates the bean.
2. Populate properties- Spring IoC container injects the bean’s properties.
3. Set Bean Name- Spring container sets the bean name. If the bean implements BeanNameAware, spring container passes the bean’s id to setBeanName() method.
4. Set Bean Factory-If the bean implements BeanFactoryAware, Spring container passes theBeanFactory to setBeanFactory().
5. Pre Initialization-This stage is also called the bean postprocess . If there are anyBeanPostProcessors, theSpring container calls the postProcesserBeforeInitialization () method.
6. Initialize beans- If the bean implements IntializingBean,its afterPropertySet()method is called. If the bean has init method declaration, the specified initialization method is called.
7. Post Initialization- If BeanPostProcessors is implemented by the bean, the Spring container calls their postProcessAfterinitalization() method.
8. Ready to Use- Now the bean is ready to be used by the application.
9. Destroy- The bean is destroyed during this stage. If the bean implements DisposableBean, the Spring IoC container will call the destroy() method . If a custom destroy () method is defined, the container calls the specified method.

<http://geekabyte.blogspot.in/2014/07/hooking-into-container-and-bean-life.html>

This is briefly enumerated:  
  
1. Read and parse the Bean definitions (either as XML, JavaConfig)  
2. Process BeanFactoryPostProcessors  
3. Construct bean by calling its constructor  
4. Call setters, dependencies injected  
5. Inject the required beans as defined by the \*Aware Interfaces  
6. Call postProcessBeforeIntialization  
7. Call the initializaion callbacks.(like InitializingBean's afterPropertiesSet or a custom init-method)  
8. Call postProcessAfterInitialization

# **Spring Bean Life Cycle**

1. The Bean Container finds the definition of the Spring Bean in the Configuration file.
2. The Bean Container creates an instance of the Bean using Java Reflection API.
3. If any properties are mentioned, then they are also applied. If the property itself is a Bean, then it is resolved and set.
4. If the Bean class implements the **BeanNameAware** interface, then the **setBeanName()** method will be called by passing the name of the Bean.
5. If the Bean class implements the **BeanClassLoaderAware** interface, then the method **setBeanClassLoader()** method will be called by passing an instance of the **ClassLoader** object that loaded this bean.
6. If the Bean class implements the **BeanFactoryAware** interface, then the method **setBeanFactory()** will be called by passing an instance of **BeanFactory**object.
7. If there are any **BeanPostProcessors** object associated with the **BeanFactory** that loaded the Bean, then the method **postProcessBeforeInitialization()** will be called even before the properties for the Bean are set.
8. If the Bean class implements the **InitializingBean** interface, then the method **afterPropertiesSet()** will be called once all the Bean properties defined in the Configuration file are set.
9. If the Bean definition in the Configuration file contains a **'init-method'** attribute, then the value for the attribute will be resolved to a method name in the Bean class and that method will be called.
10. The **postProcessAfterInitialization()** method will be called if there are any Bean Post Processors attached for the Bean Factory object.
11. If the Bean class implements the **DisposableBean** interface, then the method **destroy()** will be called when the Application no longer needs the bean reference.
12. If the Bean definition in the Configuration file contains a **'destroy-method'**attribute, then the corresponding method definition in the Bean class will be called.

# **Difference between BeanNameAware and BeanFactoryAware**

Consider the two interfaces BeanNameAware and BeanFactoryAware. **The first one makes the object aware of their bean name in a bean factory. The second interface gives the bean access to the Bean Factory that created it.**

BeanNameAware makes the object aware of its bean name. It is best used in pre annotation config spring (2.x). You could reference the bean from a locator by its name then.   
BeanFactoryAware gives the bean access to the beanfactory that created it. For the usefulness of this, you should check the documentation:

Spring Bean Post Processor

As per Spring specifications, The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more BeanPostProcessor implementations.

There are two methods of BeanPostProcessor

* **postProcessBeforeInitialization**
* **postProcessAfterInitialization**

So in essence the method **postProcessBeforeInitialization** defined in the BeanPostProcessor gets called (as the name indicates) before the initialization of beans and likewise the **postProcessAfterInitialization** gets called after the initialization of the bean.

Note : **So basically the BeanPostProcessor can be used to do custom instantiation logic for several beans whereas the others are defined on a per bean basis.**

**BeanPostProcessor is used to customize the beans’ instantiation process.**

BeanPostProcessor class has two methods.

1) postProcessBeforeInitialization - as name clearly says that it's used to make sure required actions are taken before initialization. e.g. you want to load certain property file/read data from the remote source/service.

2) postProcessAfterInitialization - any thing that you want to do after initialization before bean reference is given to application.

Sequence of the questioned methods in life cycle as follows :

1) BeanPostProcessor.postProcessBeforeInitialization()

2) init()

3) BeanPostProcessor.postProcessAfterInitialization()

4) destroy()

An working example is given below.

**Employee.java**

package com.ddlab.rnd.spring;

public class Employee {

private String name;

private int id;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void init() {

System.out.println("init() method called , Emp object is getting initialized ...");

}

public void destroy() {

System.out.println("destroy() method called , Emp object will be destroyed now ...");

}

}

**Project.java**

package com.ddlab.rnd.spring;  
  
public class Project {  
 private String name;  
 private int id;  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void init() {  
 System.out.println("init() method called , Project object is getting initialized ...");  
 }  
  
 public void destroy() {  
 System.out.println("destroy() method called , Project object will be destroyed now ...");  
 }  
}

**InitBeans.java**

package com.ddlab.rnd.spring;  
  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.beans.BeansException;  
  
public class InitBeans implements BeanPostProcessor {  
   
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 System.*out*.println("BeforeInitialization : " + beanName);  
 if( bean instanceof Employee ) {  
 Employee emp = (Employee) bean;  
 emp.setId(1);  
 }  
  
 if( bean instanceof Project ) {  
 Project project = (Project) bean;  
 project.setId(1);  
 }  
  
 return bean; // you can return any other object as well  
 }  
  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 System.*out*.println("AfterInitialization : " + beanName);  
 return bean; // you can return any other object as well  
 }  
  
}

**MainApp.java**

package com.ddlab.rnd.spring;  
  
import org.springframework.context.support.AbstractApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class MainApp {  
 public static void main(String[] args) {  
  
 AbstractApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  
  
 Employee obj = (Employee) context.getBean("emp");  
 System.*out*.println(obj.getName());  
 System.*out*.println(obj.getId());  
 context.registerShutdownHook();  
 }  
}

**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-3.0.xsd">  
  
 <bean id="emp" class="com.ddlab.rnd.spring.Employee"  
 init-method="init" destroy-method="destroy">  
 <property name="name" value="Hello World!"/>  
 </bean>  
  
 <bean id="project" class="com.ddlab.rnd.spring.Project"  
 init-method="init" destroy-method="destroy">  
 <property name="name" value="Digital marketing"/>  
 </bean>  
  
 <bean class="com.ddlab.rnd.spring.InitBeans" />  
  
</beans>

**Output is given below**

BeforeInitialization : emp

init() method called , Emp object is getting initialized ...

AfterInitialization : emp

BeforeInitialization : project

init() method called , Project object is getting initialized ...

AfterInitialization : project

Hello World!

1

destroy() method called , Project object will be destroyed now ...

destroy() method called , Emp object will be destroyed now ...

Some Spring Life Cycle Concepts

**org.springframework.beans.factory   
Interface BeanNameAware**

|  |  |
| --- | --- |
| void | [**setBeanName**](http://docs.spring.io/spring-framework/docs/2.5.x/api/org/springframework/beans/factory/BeanNameAware.html#setBeanName(java.lang.String))**(**[**String**](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true)**name)            Set the name of the bean in the bean factory that created this bean.** |

## org.springframework.beans.factory  Interface BeanFactoryAware

|  |  |
| --- | --- |
| void | [**setBeanFactory**](http://docs.spring.io/spring-framework/docs/2.5.x/api/org/springframework/beans/factory/BeanFactoryAware.html#setBeanFactory(org.springframework.beans.factory.BeanFactory))([BeanFactory](http://docs.spring.io/spring-framework/docs/2.5.x/api/org/springframework/beans/factory/BeanFactory.html) beanFactory)           **Callback that supplies the owning factory to a bean instance.** |

The usage of above interfaces is given below.

**ICard.java**

**public** **interface** ICard {

String getType();

}

**CreditCard.java**

**import** org.springframework.beans.BeansException;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.BeanFactoryAware;

**import** org.springframework.beans.factory.BeanNameAware;

**public** **class** CreditCard **implements** BeanNameAware , BeanFactoryAware , ICard {

**public** String getType() {

**return** "CreditCard";

}

**public** **void** setBeanName( String beanId ) {

//Part of BeanNameAware

System.***out***.println("Defined Bean Id :::"+beanId);

}

**public** **void** setBeanFactory(BeanFactory beanFactory) **throws** BeansException {

//Part of BeanFactoryAware

System.***out***.println("Bean Factory :::"+beanFactory);

}

}

**Spring bean configuration(beans.xml)**

<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-3.2.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd"*>

**<bean id=*"cardBean"* class=*"com.ddlab.rnd.CreditCard"* />**

</beans>

**Test Program (App.java)**

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");

ICard card = (ICard) context.getBean("cardBean");

System.***out***.println(card.getType());

}

}

**Output**

Defined Bean Id :::cardBean

Bean Factory :::org.springframework.beans.factory.support.DefaultListableBeanFactory@18f47d52: defining beans [cardBean]; root of factory hierarchy

CreditCard

**BeanPostProcessor**

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.

import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

System.out.println("BeforeInitialization : " + beanName);

return bean; // you can return any other object as well

}

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

System.out.println("AfterInitialization : " + beanName);

return bean; // you can return any other object as well

}

}

Cross-Cutting Concerns

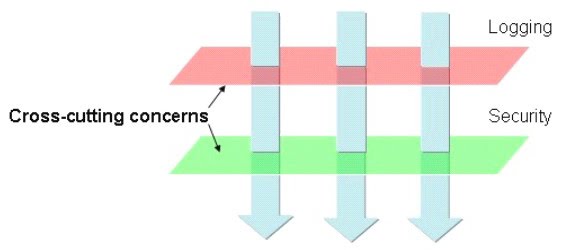
Before understanding the **Crosscutting Concern**, we have to understand the **Concern**.

A **Concern** is behavior we want to have in a module of an application.  
A **Concern** may be defined as a functionality we want to implement.

The **cross-cutting concern** is a concern which is applicable throughout the application and it affects the entire application.

**For example:** logging, security and data transfer are the concerns which are needed in almost every module of an application, hence they are cross-cutting concerns.

[Courtesy](http://www.javacodegeeks.com/2014/05/spring-interview-questions-and-answers.html#6)



This figure represents a typical application that is broken down into modules. Each module’s main concern is to provide services for its particular domain. However, each of these modules also requires similar ancillary functionalities, such as security logging and transaction management. An example of crosscutting concerns is "logging," which is frequently used in distributed applications to aid debugging by tracing method calls. Suppose we do logging at both the beginning and the end of each function body. This will result in crosscutting all classes that have at least one function.

<https://en.wikipedia.org/wiki/Cross-cutting_concern>

In [aspect-oriented software development](https://en.wikipedia.org/wiki/Aspect-oriented_software_development), **cross-cutting concerns** are aspects of a [program](https://en.wikipedia.org/wiki/Computer_program) that affect other [concerns](https://en.wikipedia.org/wiki/Concern_(computer_science)). These concerns often cannot be cleanly [decomposed](https://en.wikipedia.org/wiki/Modularity_(programming)) from the rest of the system in both the design and implementation, and can result in either *scattering* ([code](https://en.wikipedia.org/wiki/Source_code) duplication), *tangling* (significant dependencies between systems), or both.

Cross-cutting concerns are parts of a program that rely on or must affect many other parts of the system. They form the basis for the development of [aspects](https://en.wikipedia.org/wiki/Aspect_(computer_science)).[[1]](https://en.wikipedia.org/wiki/Cross-cutting_concern#cite_note-Patent_4-1) Such cross-cutting concerns do not fit cleanly into [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) or [procedural programming](https://en.wikipedia.org/wiki/Procedural_programming).[[2]](https://en.wikipedia.org/wiki/Cross-cutting_concern#cite_note-Seminal_1-2)

<https://en.wikipedia.org/wiki/Concern_(computer_science)>

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **concern** is a particular set of information that has an effect on the code of a [computer program](https://en.wikipedia.org/wiki/Computer_program). A concern can be as general as the details of [database](https://en.wikipedia.org/wiki/Database) interaction or as specific as performing a primitive calculation, depending on the level of conversation between developers and the program being discussed. IBM uses the term *concern space* to describe the sectioning of conceptual information.[[1]](https://en.wikipedia.org/wiki/Concern_(computer_science)#cite_note-1)

Usually the code can be separated into logical sections, each addressing separate concerns, and so it hides the need for a given section to know particular information addressed by a different section. This leads to a [modular](https://en.wikipedia.org/wiki/Modularity_(programming)) program. [Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra) coined the term "[separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns)"[[2]](https://en.wikipedia.org/wiki/Concern_(computer_science)#cite_note-2) to describe the mentality behind this modularization, which allows the programmer to reduce the complexity of the system being [designed](https://en.wikipedia.org/wiki/Designed). Two different concerns that intermingle in the same section of code are called "[highly coupled](https://en.wikipedia.org/wiki/Coupling_(computer_science))". Sometimes the chosen module divisions do not allow for one concern to be completely separated from another, resulting in [cross-cutting concerns](https://en.wikipedia.org/wiki/Cross-cutting_concern).[[3]](https://en.wikipedia.org/wiki/Concern_(computer_science)#cite_note-3) The various [programming paradigms](https://en.wikipedia.org/wiki/Programming_paradigm) address the issue of[cross-cutting concerns](https://en.wikipedia.org/wiki/Cross-cutting_concern) to different degrees. [Data logging](https://en.wikipedia.org/wiki/Data_logging) is a common cross-cutting concern, being used in many other parts of the program other than the particular module(s) that actually log the data. Since changes to the logging code can affect other sections,[[*how?*](https://en.wikipedia.org/wiki/Wikipedia:Please_clarify)] it could introduce [bugs](https://en.wikipedia.org/wiki/Computer_bug) in the operation of the program.

Paradigms that specifically address the issue of concern separation:

* [Object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), describing concerns as objects
* [Functional programming](https://en.wikipedia.org/wiki/Functional_programming), describing concerns as functions
* [Aspect-oriented software development](https://en.wikipedia.org/wiki/Aspect-oriented_software_development), treating concerns and their interaction as constructs of their own standing

<https://en.wikipedia.org/wiki/Separation_of_concerns>

**Separation of concerns**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **separation of concerns** (**SoC**) is a design principle for separating a [computer program](https://en.wikipedia.org/wiki/Computer_program) into distinct sections, such that each section addresses a separate [concern](https://en.wikipedia.org/wiki/Concern_(computer_science)). A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate. A program that embodies SoC well is called a [modular](https://en.wikipedia.org/wiki/Modularity_(programming))[[1]](https://en.wikipedia.org/wiki/Separation_of_concerns#cite_note-laplante-1) program. Modularity, and hence separation of concerns, is achieved by [encapsulating](https://en.wikipedia.org/wiki/Encapsulation_(computer_science)) information inside a section of code that has a well-defined interface. Encapsulation is a means of [information hiding](https://en.wikipedia.org/wiki/Information_hiding).[[2]](https://en.wikipedia.org/wiki/Separation_of_concerns#cite_note-mitchell-2) Layered designs in information systems are another embodiment of separation of concerns (e.g., presentation layer, business logic layer, data access layer, persistence layer).[[3]](https://en.wikipedia.org/wiki/Separation_of_concerns#cite_note-microsoft-3)

The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well-separated, individual sections can be reused, as well as developed and updated independently. Of special value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections.

Spring 3 MVC Interceptor tutorial with example

Spring MVC provides a powerful mechanism to intercept an http request. Similar to [Servlet Filter concept](http://viralpatel.net/blogs/tutorial-java-servlet-filter-example-using-eclipse-apache-tomcat/), Spring MVC provides a way to define special classes called Interceptors that gets called before and after a request is served.

## Quick Overview

Each interceptor you define must implementorg.springframework.web.servlet.HandlerInterceptor interface. There are three methods that need to be implemented.

preHandle(..) is called before the actual handler is executed;

The preHandle(..) method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

postHandle(..) is called after the handler is executed;

afterCompletion(..) is called after the complete request has finished.

These three methods should provide enough flexibility to do all kinds of preprocessing and postprocessing.

You can define an interceptor class as follow:

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

**public class HelloWorldInterceptor implements HandlerInterceptor  {**

    @Override

    public boolean preHandle(HttpServletRequest request,

            HttpServletResponse response, Object handler) throws Exception {

        System.out.println("Pre-handle");

        return true;

    }

    //override postHandle() and afterCompletion()

}

Once the interceptor is defined, you can ask Spring MVC to configure it via <mvc:interceptors>tag within spring-servlet.xml file.

<mvc:interceptors>

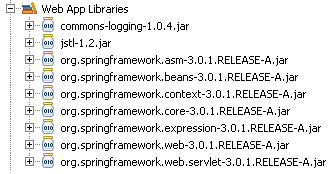
  <bean class="net.viralpatel.spring3.interceptor.HelloWorldInterceptor" />

</mvc:interceptors>

Let us start with the complete Demo application. We will create a demo Interceptor that logs each request.

Tools and technologies:

1. Java 5 (or above)
2. Spring MVC 3.0 (or above)
3. Eclipse 3.2 (or above)

We will need following JAR files in order to execute this project.  


If you are using [Apache Maven](http://viralpatel.net/blogs/introduction-apache-maven-build-framework-build-automation-tool/) as dependency management, add following dependencies to pom.xml.

|  |
| --- |
| <dependencies>    <dependency>        <groupId>org.springframework</groupId>        <artifactId>spring-webmvc</artifactId>        <version>3.0.1.RELEASE</version>    </dependency>    <dependency>        <groupId>jstl</groupId>        <artifactId>jstl</artifactId>        <version>1.2</version>    </dependency>  </dependencies> |

Let us create a simple Spring MVC Interceptor class that logs each request.

## Step 1: The Controller – Create new Spring MVC Controller

We create a simple Spring MVC controller that displays a plain JSP view.

HelloWorldController.java

|  |
| --- |
| package net.viralpatel.spring3.controller;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;    @Controller  public class HelloWorldController {      @RequestMapping(value = "/hello", method = RequestMethod.GET)      public String sayHello() {          return "hello";      }  } |

The controller has one method sayHello() which is mapped to URL **/hello** using@RequestMapping. This simply paints the hello.jsp view.

## Step 2: The View – Create new JSPs

Create two JSPs, hello.jsp that displays hello message and index.jsp that simply redirects first request to /hello.html.

/WEB-INF/jsp/hello.jsp

|  |
| --- |
| <html>  <head>      <title>Spring MVC Interceptor example</title>  </head>  <body>      <h1>Hello!!</h1>  </body>  </html> |

WebContent/index.jsp

|  |
| --- |
| <jsp:forward page="hello.html"></jsp:forward> |

The index.jsp simply redirects to hello.html which calls the HelloWorldController.

**Step 3: The Interceptor – Spring MVC HandlerInterceptor**

Let’s create the Spring based Handler Interceptor which will intercept the request and print a message.

HelloWorldInterceptor.java

|  |
| --- |
| package net.viralpatel.spring3.interceptor;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import org.springframework.web.servlet.HandlerInterceptor;  import org.springframework.web.servlet.ModelAndView;    **public class HelloWorldInterceptor implements HandlerInterceptor  {**      @Override      public boolean preHandle(HttpServletRequest request,              HttpServletResponse response, Object handler) throws Exception {            System.out.println("Pre-handle");          return true;      }        @Override      public void postHandle(HttpServletRequest request,              HttpServletResponse response, Object handler,              ModelAndView modelAndView) throws Exception {          System.out.println("Post-handle");      }        @Override      public void afterCompletion(HttpServletRequest request,              HttpServletResponse response, Object handler, Exception ex)              throws Exception {          System.out.println("After completion handle");      }  } |

Thus in each of the method preHandle, postHandle and afterCompletion we print a message on console.

**Step 4: Spring Configuration**

Now lets glue up the source code and configure Spring. Note how we declared the interceptor in spring-servlet.xml using <mvc:interceptors> tag.

spring-servlet.xml

|  |
| --- |
| <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"      xmlns:mvc="http://www.springframework.org/schema/mvc"      xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd">        <context:annotation-config />      <context:component-scan base-package="net.viralpatel.spring3.controller"/>  **<mvc:interceptors>**  **<bean class="net.viralpatel.spring3.interceptor.HelloWorldInterceptor" />**  **</mvc:interceptors>**      <bean id="jspViewResolver"      class="org.springframework.web.servlet.view.InternalResourceViewResolver">          <property name="viewClass"              value="org.springframework.web.servlet.view.JstlView" />          <property name="prefix" value="/WEB-INF/jsp/" />          <property name="suffix" value=".jsp" />    </bean>  </beans> |

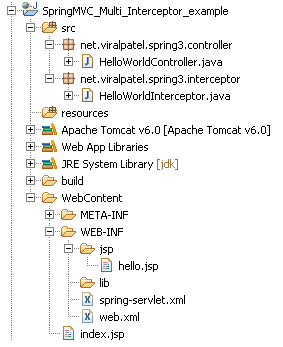
Also configure the spring’s DispatcherServlet in web.xml file.

web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"      xmlns="http://java.sun.com/xml/ns/javaee"      xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee          http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"      id="WebApp\_ID" version="2.5">        <display-name>Spring3MVC-Interceptor example</display-name>      <servlet>          <servlet-name>spring</servlet-name>          <servlet-class>              org.springframework.web.servlet.DispatcherServlet          </servlet-class>          <load-on-startup>1</load-on-startup>      </servlet>      <servlet-mapping>          <servlet-name>spring</servlet-name>          <url-pattern>\*.html</url-pattern>      </servlet-mapping>  </web-app> |

## Final Project Structure

Once all the source files and configuration files are in place, your project should look like below:



**Demo**

Compile and execute the project in Eclipse. Open below URL in browser.

**URL:** <http://localhost:8080/SpringMVC_Interceptor_example/>

Check the server console logs. You must see following:

|  |
| --- |
| Pre-handle  Post-handle  After completion handle |

XML or Annotations in Spring

Annotations have their use, but they are not the one silver bullet to kill XML configuration. I recommend mixing the two!

For instance, if using Spring, it is entirely intuitive to use XML for the dependency injection portion of your application. This gets the code's dependencies away from the code which will be using it, by contrast, using some sort of annotation in the code that needs the dependencies makes the code aware of this automatic configuration.

However, instead of using XML for transactional management, marking a method as transactional with an annotation makes perfect sense, since this is information a programmer would probably wish to know. But that an interface is going to be injected as a SubtypeY instead of a SubtypeX should not be included in the class, because if now you wish to inject SubtypeX, you have to change your code, whereas you had an interface contract before anyways, so with XML, you would just need to change the XML mappings and it is fairly quick and painless to do so.

I haven't used JPA annotations, so I don't know how good they are, but I would argue that leaving the mapping of beans to the database in XML is also good, as the object shouldn't care where its information came from, it should just care what it can do with its information. But if you like JPA (I don't have any expirience with it), by all means, go for it.

In general: If an annotation provides functionality and acts as a comment in and of itself, and doesn't tie the code down to some specific process in order to function normally without this annotation, then go for annotations. For example, a transactional method marked as being transactional does not kill its operating logic, and serves as a good code-level comment as well. Otherwise, this information is probably best expressed as XML, because although it will eventually affect how the code operates, it won't change the main functionality of the code, and hence doesn't belong in the source files.

There is a wider issue here, that of externalised vs inlined meta-data. If your object model is only ever going to persisted in one way, then inlined meta-data (i.e. annotations) are more compact and readable.

If, however, your object model was reused in different applications in such a way that each application wanted to persist the model in different ways, then externalising the meta-data (i.e. XML descriptors) becomes more appropriate.

Neither one is better, and so both are supported, although annotations are more fashionable. As a result, new hair-on-fire frameworks like JPA tend to put more emphasis on them. More mature APIs like native Hibernate offer both, because it's known that neither one is enough.

I always think about annotations as some kind of indicator of what a class is capable of, or how it interacts with others.

Spring XML configuration on the other hand to me is just that, configuration

For instance, information about the ip and port of a proxy, is definetly going into an XML file, it is the runtime configuration.

Using @Autowire, @Element to indicate the framework what to do with the class is good use of annotations.

Putting the URL into the @Webservice annotation is bad style.

But this is just my oppinion. The line between interaction and configuration is not allways clear

I've been using Spring for a few years now and the amount of XML that was required was definitely getting tedious. Between the new XML schemas and annotation support in Spring 2.5 I usually do these things:

1. Using "component-scan" to autoload classes which use @Repository, @Service or @Component. I usually give every bean a name and then wire them together using @Resource. I find that this plumbing doesn't change very often so annotations make sense.
2. Using the "aop" namespace for all AOP. This really works great. I still use it for transactions too because putting @Transactional all over the place is kind of a drag. You can create named pointcuts for methods on any service or repository and very quickly apply the advice.
3. I use LocalContainerEntityManagerFactoryBean along with HibernateJpaVendorAdapter to configure Hibernate. This lets Hibernate easily auto-discover @Entity classes on the classpath. Then I create a named SessionFactory bean using "factory-bean" and "factory-method" referring to the LCEMFB.
4. I think that visibility is a big win with an XML based approach. I find that the XML isn't really that bad, given the various tools out there for navigating XML documents (i.e. Visual Studio + ReSharper's File Structure window).
5. You can certainly take a mixed approach, but that seems dangerous to me if only because, potentially, it would make it difficult for new developers on a project to figure out where different objects are configured or mapped.
6. I don't know; in the end XML Hell doesn't seem all that bad to me.

It depends on what everything you want to configure, because there are some options that cannot be configured with anotations. If we see it from the side of annotations:

* plus: annotations are less talky
* minus: annotations are less visible

It's up to you what is more important...

In general I would recommend to choose one way and use it all over some closed part of product...

(with some exceptions: eg if you choose XML based configurations, it's ok to use @Autowire annotation. It's mixing, but this one helps both readability and maintainability)

I might be wrong, but I thought Annotations (as in Java's @Tag and C#'s [Attribute]) were a compile-time option, and XML was a run-time option. That to me says the are not equivalent and have different pros and cons.

I also think a mix is the best thing, but it also depends on the type of configuration parameters. I'm working on a Seam project which also uses Spring and I usually deploy it to different development and test servers. So I have split:

* Server specific configuration (Like absolute paths to resources on server): Spring XML file
* Injecting beans as members of other beans (or reusing a Spring XML defined value in many beans): Annotations

The key difference is that you don't have to recompile the code for all changing server-specific configurations, just edit the xml file. There's also the advantage that some configuration changes can be done by team members who don't understand all the code involved.

An important part in using an annotation-only approach is that the concept of a "bean name" more or less goes away (becomes insignificant).

The "bean names" in Spring form an additional level of abstraction over the implementing classes. With XML beans are defined and referenced relative to their bean name. With annotations they are referenced by their class/interface. (Although the bean name exists, you do not need to know it)

I strongly believe that getting rid of superfluous abstractions simplifies systems and improves productivity. For *large* projects I think the gains by getting rid of XML can be substantial.

In the scope of DI container, I consider annotation based DI is abusing the use of Java annotation. By saying that, I don't recommend to use it widely in your project. If your project does really needs the power of DI container, I would recommend to use Spring IoC with Xml based configuration option.

If it is just for a sake of Unit-test, developers should apply Dependency Inject pattern in their coding and take advantages from mocking tools such as EasyMock or JMock to circumvent dependencies.

You should try to avoid using DI container in its wrong context.

This is the classic 'Configuration versus Convention' question. Personal taste dictates the answer in most cases. However, personally I prefer Configuration (i.e. XML based) over Convention. IMO IDE's are sufficiently robust enough to overcome some of the XML hell people often associate w/ the building and maintaining an XML based approach. In the end, I find the benefits of Configuration (such as building utilities to build, maintain and deploy the XML config file) outweighs Convention in the long run.

I use both. Mostly XML, but when I have a bunch of beans that inherit from a common class and have common properties, I use annotations for those, in the superclass, so I don't have to set the same properties for each bean. Because I'm a bit of a control freak, I use @Resource(name="referredBean") instead of just autowiring stuff (and save myself a lot of trouble if I ever need another bean of the same class as the original referredBean).

**Advantages of the annotation:**

1) All the information is in a single file (no need to open two files to configure a given behavior)   
2) When the class changes, no need to modify the xml file

**Advantages of xml file:**

1) Clear separation between the POJO and its behavior   
2) When you do not know which POJO is responsible for the behavior, it is easier to find that POJO (searching in a subset of files rather than all the source code)

Spring Core with No XML Configuration - Spring with Annotations only

# Java Code

## SpringNoXMLConfig.java

package com.ddlab.spring;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages="com.ddlab.spring")

public class SpringNoXMLConfig {

}

## ICustomer.java

**package** com.ddlab.spring;

**public** **interface** ICustomer

{

**public** String getName();

**public** String getType();

}

## RetailCustomerImpl.java

**package** com.ddlab.spring;

**import** org.springframework.stereotype.Component;

@Component(value="retail")

**public** **class** RetailCustomerImpl **implements** ICustomer {

**public** String getName() {

**return** "John Abraham";

}

**public** String getType() {

**return** "Retail";

}

}

## CorporateCustomerImpl.java

**package** com.ddlab.spring;

**import** org.springframework.stereotype.Component;

@Component(value="corporate")

**public** **class** CorporateCustomerImpl **implements** ICustomer {

**public** String getName() {

**return** "JDA Software Inc.";

}

**public** String getType() {

**return** "Corporate";

}

}

## Bank.java

package com.ddlab.spring;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component(value="bank1")

public class Bank

{

@Autowired @Qualifier(value="retail") //@Qualifier(value="corporate")

private ICustomer customer;

public void display()

{

System.out.println("Customer Name : "+customer.getName());

System.out.println("Customer Type : "+customer.getType());

}

}

## TestBank.java

package com.ddlab.spring;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TestBank {

@SuppressWarnings("resource")

public static void main(String[] args) {

ApplicationContext context = null;

context = new AnnotationConfigApplicationContext(SpringNoXMLConfig.class);

Bank bank = (Bank) context.getBean("bank1");

bank.display();

}

}

**What about normal bean ?**

To get only Bean

@Bean(value="retail")

public ICustomer getRetailCustomer() {

return new RetailCustomers();

}  
  
There is no need to write **@Component** in the RetailCustomer class.