**Spring**

I want to develop one enterprise application with all required middle level services. What are the recommended technologies and framework for implementing all the services?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Services** | **Technology and Framework (Without Spring)** | **Introduced in Spring** | **To Integrate with**  **Spring** |
|  | Web Support | Servlet, JSP, Struts, JSF, Struts2, Flex | Spring MVC | All |
|  | Persistence Service | JDBC, Hibernate, JPA, Entity Bean (EJB) | X | All |
|  | Transaction Service | JDBC Transaction, Hibernate Transaction, JPA Transaction, EJB Container Transaction | Spring Transaction |  |
|  | Security Service | JAAS, Third Party | Spring Security | All |
|  | Remoting Service | RMI, EJB2/3, Web Service | X | All |
|  | Timer Service | JDK, EJB Container | √ | √ |
|  | Messaging Service | JMS, MDB | X | All |
|  | Resource Management | Web Container, EJB Container | Spring Container |  |
|  | Life Cycle Management | Web Container, EJB Container | Spring Container |  |
|  | Mailing Service | Java Mail API | X | All |
|  | Registry Service | RMI Registry, JNDI Registry, UDDI | X | All |
|  | Logging Service | Log4J | X |  |
|  | Unit Testing | JUnit, Mock Object, Easy Mock | √ | All |
|  | IOC | EJB3 Container | √ |  |
|  | AOP | EJB3 Container | √ |  |

**Inversion Of Control (IOC)**

IOC can be implemented in two ways.

1. Dependency Look Up
2. Dependency Injection
3. Dependency Look Up:

If you want any objects to use you can create those objects or you can look up and get it from registry. This is the normal flow of control.

1. Dependency Injection:

Dependency Injection mechanism injects the required object automatically this simplifies the programming.

Note: Now onward we’ll use the term Bean instead of instance or object or components.

The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

We can implement dependency Injection in two ways:

1. Setter Injection
2. Constructor Injection

Example of above two injections

**Inner Beans**

You can define one bean definition in other bean. This type of configuration is called as inner beans and cannot be reused.

Following is the sample xml:

Spring Core Container has implemented based on two popular GOF design pattern.

1. Factory Pattern
2. Singleton Pattern

**Bean Scope**

Bean instance created by the Spring Container can be available in 5 Scopes.

1. Singleton
2. Prototype
3. Request
4. Session
5. Global-session
6. Singleton:

When bean scope is Singleton than only one instance will be created for that bean and the same will be used to inject in other beans and same bean instance will be return whenever you will be call “getBean()” method.

By default all beans are singleton beans.

1. Prototype:

When bean scope is prototype then a new instance will be create every time whenever you call “getBean()” method.

Following is the way to specify the prototype scope.

<bean id = “x” class = “….” scope = “prototype” />

1. Request:
2. Session:

These two are equivalent to **HttpSerevletRequest** and **HttpSession** respectively in the case of Web base application. These two scopes must be used for web application only.

1. Goblal-session:

This scope is equivalent to HttpSession in case of Portal base Web Application.

**Bean Loading Styles**

Spring container loads the bean in two different styles.

1. Eager Loading or Aggressive Loading
2. Lazy Loading

* In case of Eager Loading, beans will be created by the Spring Container at container start up.
* In case of Lazy Loading, beans will be created by the Spring Container, when you will use it first time.
* The default loading approach is Aggressive Loading.
* To specify the Lazy Loading you can use “lazy-init” attribute as follows:

<bean id =”x” class = “…….” lazy-init = “true” />

**Injecting various type of values through DI (Dependency Injection)**

Using DI you can inject the following types:

1. Primitives
2. String
3. Wrappers
4. Collection (Map, Set, List, Properties)
5. Other Beans

**public** **class** LockerConstants {

**private** **static** Integer *paletteSize*;

**private** **static** String *ccPush*;

**private** **static** String *mnicTerminationPush*;

**private** **static** Map<Class<? **extends** CsdXml>, String> *csdXmlAbbrevMap*;

**private** **static** List<Integer> *defaultRoleIds*;

**private** **static** Map<String, String> *tables*;

**private** **static** String *nicNumberRegex*;

**private** **static** DateFormat *dateConverterDateFormat*;

**public** **static** Integer getPaletteSize() {

**return** *paletteSize*;

}

**public** **static** **void** setPaletteSize(**final** Integer paletteSize) {

ScreenConstants.*paletteSize* = paletteSize;

}

**public** **static** String getCcPush() {

**return** *ccPush*;

}

**public** **static** **void** setCcPush(**final** String ccPush) {

LockerConstants.*ccPush* = ccPush;

}

**public** **static** String getMnicTerminationPush() {

**return** *mnicTerminationPush*;

}

**public** **static** **void** setMnicTerminationPush(**final** String mnicTerminationPush) {

LockerConstants.*mnicTerminationPush* = mnicTerminationPush;

}

**public** **static** Map<Class<? **extends** CsdXml>, String> getCsdXmlAbbrevMap() {

**return** *csdXmlAbbrevMap*;

}

**public** **static** **void** setCsdXmlAbbrevMap(**final** Map<Class<? **extends** CsdXml>, String> csdXmlAbbrevMap) {

EventTypeConstants.*csdXmlAbbrevMap* = csdXmlAbbrevMap;

}

**public** **static** List<Integer> getDefaultRoleIds() {

**return** *defaultRoleIds*;

}

**public** **static** **void** setDefaultRoleIds(**final** List<Integer> defaultRoleIds) {

ScreenConstants.*defaultRoleIds* = defaultRoleIds;

}

**public** **static** Map<String, String> getTables() {

**return** CodeTableConstants.*tables*;

}

**public** **static** **void** setTables(**final** Map<String, String> tables) {

CodeTableConstants.*tables* = tables;

}

**public** **static** String getNicNumberRegex() {

**return** *nicNumberRegex*;

}

**public** **static** **void** setNicNumberRegex(**final** String nicNumberRegex) {

CsdHandlerConstants.*nicNumberRegex* = nicNumberRegex;

}

**public** **static** DateFormat getDateConverterDateFormat() {

**return** *dateConverterDateFormat*;

}

**public** **static** **void** setDateConverterDateFormat(**final** DateFormat dateConverterDateFormat) {

ScreenConstants.*dateConverterDateFormat* = dateConverterDateFormat;

}

}

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:jee=*"http://www.springframework.org/schema/jee"*

xmlns:util=*"http://www.springframework.org/schema/util"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd*

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

<bean class=*"com.crimsonlogic.mnis.cpdv2.constants.LockerConstants"*>

<property name=*"paletteSize"* value=*"5"* />

<property name=*"ccPush"* value=*"CC\_PUSH"*/>

<property name=*"mnicTerminationPush"* value=*"MNIC\_TERMINATE\_PUSH"*/>

<property name=*"csdXmlAbbrevMap"*>

<util:map key-type=*"java.lang.Class"*>

<entry key=*"com.crimsonlogic.mnis.cpdv2.model.csd.Births"* value=*"BTH"* />

<entry key=*"com.crimsonlogic.mnis.cpdv2.model.csd.BthDets"* value=*"BDT"* />

<entry key=*"com.crimsonlogic.mnis.cpdv2.model.csd.Deaths"* value=*"DTH"* />

<entry key=*"com.crimsonlogic.mnis.cpdv2.model.csd.Divorce"* value=*"DIV"* />

<entry key=*"com.crimsonlogic.mnis.cpdv2.model.csd.Marriages"* value=*"MAR"* />

<entry key=*"com.crimsonlogic.mnis.cpdv2.model.csd.ForeignReg"* value=*"FOR"* />

</util:map>

</property>

<property name=*"defaultRoleIds"*>

<util:list value-type=*"java.lang.Integer"*>

<value>1</value>

<value>2</value>

<value>3</value>

<value>4</value>

</util:list>

</property>

<property name=*"tables"*>

<util:map key-type=*"java.lang.String"* value-type=*"java.lang.String"*>

<entry key=*"COUNTRY"* value=*"Country"*/>

<entry key=*"DEATH\_TYPE"* value=*"Death Type"* />

<entry key=*"DISTRICT"* value=*"District"*/>

<entry key=*"MARITAL\_STATUS"* value=*"Marital Status"*/>

<entry key=*"SEX"* value=*"Sex"*/>

</util:map>

</property>

<property name=*"nicNumberRegex"* value=*"^[A-Z](?:\d{12})[A-G0-9]$"*/>

<property name=*"dateConverterDateFormat"*>

<bean class=*"java.text.SimpleDateFormat"*>

<constructor-arg index=*"0"* value=*"dd${date.separator}MM${date.separator}yyyy"* />

<property name=*"lenient"* value=*"false"* />

</bean>

</property>

</bean>

</beans>

**Using p namespace:**

* With the p namespace you can reduce the tag in the spring-context.xml.
* To use the p namespace add the below namespace.

xmlns:p = “http:www.springframework.org/schema/p”

* p namespace is used only for setter injection not for constructor injection.

Example: name of the field

<bean id = “hello” class = “com.app.spring.ioc.Hello” p:a = “99” p:str = “hello guys” p:obj-ref = “ao” />

**Wiring**

* Wiring is the process of injecting the dependencies into the required bean.
* You can perform wiring in two ways:

1. Explicit Wiring
2. Implicit Wiring or Autowiring
3. Explicit Wiring:

In the case of explicit wiring you need to configure the dependencies of the bean explicitly inside “**spring-context.xml**”

|  |  |  |
| --- | --- | --- |
| class Hai {  ……  ……  } | class Hello{  Hai hai;  //setters & getters  } | class Hello{  Hai hai;  Hello(Hai hai){  this.hai = hai;  }  } |

<bean id = “h” class = “com.app.Hai” />

<bean id = “hello” class = “com.app.Hello” >

|  |
| --- |
| <property name = “hai” ref = “h” /> |

</bean>

|  |
| --- |
| <bean id = “hello” class = “com.app.Hello” >  <property name = “hai”>  <ref local = “h” />  </property>  </bean> |

|  |
| --- |
| <bean id = “hello” class = “com.app.Hello” p:hai-ref = “h” /> |

|  |
| --- |
| <bean id = “hello” class = “com.app.Hello” >  <constructor-arg>  < ref local = “h”/>  </ constructor-arg>  </bean> |

|  |
| --- |
| <bean id = “hello” class = “com.app.Hello” >  <constructor-arg ref = “h” />  </bean> |

1. Implicit Wiring or Autowiring:
   * + - In case of autowiring Spring Container detects the bean dependencies and injects them automatically.
       - To provide autowiring you have to use “autowire” attribute of bean tag.
       - Following are the possible values of autowire attributes:
2. none
3. byName
4. byType
5. constructor
6. autodetect

|  |  |
| --- | --- |
| Class Hai{  -------  ------  } | Class Hello{  Hai hai;  ------  -----  } |

1. none
   * + when autowire attribute value is “none” then container won’t do the wiring automatically. This is the default value.
2. byName
   * + need to write the setter methods.
     + When autowire attribute value is “byName” then spring container will try to detect a bean id whose name is same as property name(variable name).
     + If matching bean is found then that will be injected by setter injection.
     + If matching bean is not found then property remains uninjected.
     + Bean name will be same as bean id.

|  |  |  |
| --- | --- | --- |
| class Hai{  -------  ------  } | class A{  ------  ------  } | class Hello{  Hai hai;  A aboj;  ------  -----  } |

<bean id = “hai1” class = “com.app.Hai” />

<bean id = “aboj” class = “com.app.A” />

<bean id = “hello” class = “com.app.Hello” autowire = “byName”/>

In the above example Hello bean has two dependencies:

1. Hai bean
2. A bean

But because of “byName” autowiring process only “A” bean reference will be injected. “Hai” bean reference will have null only because no matching bean found.

1. byType
   * + Need to write setter methods
     + When autowire attribute value is byType then container will try to detect the beans whose type is same as property type.

<bean id = “hello” class = “com.app.Hello” autowire = “byType” />

In above example both dependencies of Hello bean will be injected.

* + - When container finds zero matching beans for the property type then property remains uninjected.
    - When container finds exactly one matching bean then that will be injected by setter injection.
    - When container finds two or more matching beans then exception will be thrown.

**UnsatisfiedDependencyException**

1. Constructor
   * + When autowire attribute value is “constructor” then container will try to detect beans which are matching with arguments of the constructor.
     + If no matching beans found than properties remains uninjected.
     + If matching beans founds than properties will be injected through constructor injection.

If multiple matching beans found then exception will be thrown called “**UnsatisfiedDependencyException**”.

|  |  |  |
| --- | --- | --- |
| class Hai{  -------  ------  } | class A{  ------  ------  } | class Hello{  Hai hai;  A aboj;  Hello(Hai hai, A aboj){  this.hai = hai;  this.aboj = aboj;  }  ------  -----  } |

<bean id = “hai1” class = “com.app.Hai” />

<bean id = “aboj” class = “com.app.A” />

<bean id = “hello” class = “com.app.Hello” autowire = “constructor”/>

1. Autodetect
   * + When autowire attribute value is “autodetect” then container will try to resolve the autowiring process first with “constructor” autowiring then with “byType”.

**Autowiring through annotation**

* From spring 2.5 onwards following annotations have been provided to inject the bean dependencies automatically.

1. @Autowired
2. @Resource
3. @Required

To use the above annotation in your bean you need to do the following

1. Enable the context namespace
2. Write the following tag in the “spring-context.xml”.

**<context: annotation-config />**

* Injection will happen through setter injection not from the constructor injection.

1. @Autowired
   * + When a bean property is marked with @Autowired annotation then that property will be injected through “byType” autowiring process.
     + No need to write setter methods.
2. @Resource
   * + To use this annotation you must have “javaee.jar” file.
     + No need to write setter methods.
     + @Resource annotation allows you to inject the properties in two ways.
3. byName autowiring
4. byType autowiring

* byType

@Resource

Hai hai;

* byName

@Resource(name = “hai1”)

Hai hai2;

@Required

When a method is mearked with @Required annotation then that method will be called by container automatically.

Summary:

1. Using xml based explicit wiring
2. Using xml based auto wiring
3. Using annotation based autowiring (as of Spring 2.5)

**Spring Container Callbacks (Life Cycle methods)**

Spring container provides following container callbacks.

1. Initialization callbacks
2. Destruction callbacks
3. Knowing who you are
4. Knowing where you are
5. Extending Spring Container functionality
6. Initialization callbacks:
   * + When you are developing the bean you may need to initialize the bean with the required resources.
     + You can do this using wiring process or dependency injection.
     + You can also initialized the resource using various callbacks methods provided in the bean life.
     + There are three ways to initialize the Bean Resource explicitly
7. Using @PostConstruct annotation
8. Using “afterPropertiesSet()” of “InitializingBean” interface.
9. Using custom “init()” method.
10. Destruction callbacks:
    * + When Spring container shut down, it will try to destroy all the bean instance running inside the container.
      + You may need to release some resources associated with the bean at container shout down time. You can do this in three ways:
11. Using @PreDestroy annotation.
12. Using “destroy()” method of “DisposableBean” interface.
13. Using custom destroy method.

Example:

applicationContext.xml

<context:annotation-config />

<bean id = “hello” class = “com.app.Hello” init-method = “init” destroy-method = “cleanup” />

Hello.java

package com.app;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

import org.springframeowrk.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializatingBean;

public class Hello implements InitializingBean,DisposableBean{

@PostConstruct

public void myInit(){

System.out.println(“myInit()”);

}

public void init(){

System.out.println(“init()”);

}

public void afterProperties(){

System.out.println(“afterProperties()”);

}

@PreDestroy

public void myDestoy(){

System.out.println(“myDestory()”);

}

public void destroy(){

System.out.println(“destroy()”);

}

public void cleanup(){

System.out.println(“cleanup()”);

}

}

**Test.java**

package com.app;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframewrok.context.support.ClassPathXmlApplicationContext;

public class Test{

public static void main(String [] args){

AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(“applicationContext.xml”);

System.out.println(“Spring Container ready….”);

ctx.registerShutDownHook();

System.out.println(“spring Container down…..”);

}

}

1. **Knowing who you are**
   * + If bean wants to know its name then bean has to implement “BeanNameAware” interface and has to override the following method.

public void setBeanName(String Name)

* + - Bean name is same as the Bean ID
    - If Bean class is implementing “BeanNameAware” interface then container call “setBeanName(String name)” method automatically in Bean life.

1. **Knowing where you are**
   * + If bean wants to get the reference to the Spring Container where it is running then bean class can implement one of the following two interfaces.
2. BeanFactoryAware
3. ApplciationContextAware
   * + When Bean class is implementing “BeanFactroyAware” interface then container calls the following method automatically.

public void setBeanFactory(BeanFactory bf)

* + - When your bean class is implementing ApplicationContextAware interface then container calls the following method automatically

public void setApplicationContext(ApplicationContext ctx)

1. **Extending Spring Container Functionality**
   * + If you want to extend the container functionalities you have to develop and register “BeanPostProcessor” interface.
     + Steps to write custom BeanPostProcessor
2. Write your bean class by implementing BeanPostProcessor interface.
3. Override the following two methods available in BeanPostProcessor Interface.

* Object postProcessorAfterInitialization(Object bean, String beanName)
* Object postProcessorBeforeInitialization(Object bean, String beanName)

1. Register the BeanPostProcessor class with Spring Container

**Lifecycle of a Bean in the ApplicationContext Container**

When Spring Container is creating Bean instance then container will do the following tasks:

1. Container loads the Bean class into main memory.
2. Container creates the Bean instance with appropriate constructor as per the constructor injection. If no constructor injection is defined then default constructor will be called.
3. Container will try to inject the Bean dependencies using the following ways in the same order:
4. Properties marked with @Autowired and @Resource annotation will inject.
5. Properties which are matching with xml based based autowiring will inject.
6. Properties which are matching with explicit wiring will inject.
7. If bean class is implementing BeanNameAware Interface then container calls setBeanName() method.
8. If bean class is implementing BeanFactoryAware interface then container calls setBeanFactory() method.
9. If bean class is implementing ApplciationContextAware interface then container calls setApplicationContext() method.
10. If any method of the bean is marked with @PostConstruct annotation then that method will be called by the container.
11. If any BeanPostProcessor is registered with the container then container calls

postProcessorBeforeInitialization()

1. If bean class is implementing InitializingBean interface then container calls

afterPropertiesSet()

1. If any custom init() method is specified in the spring-context.xml using “init-method” attribute then that method will be called by the container.
2. If any BeanPostProcessor is registered in the container then container calls

postProcessAfterInitialization()

1. Now Bean instance is ready to use.

**When Spring Container is destroying the bean instance, container will do the following task:**

1. If any method of the bean is marked with @PreDestroy annotation then that method will be called by the container.
2. If bean class is implementing DisposableBean Interface then container calls “destroy()” method.
3. If any custom destroy method is specified using “destroy-method” attribute then that method will be called by the container.

**BeanFactory Container**

* **BeanFactory** is the basic spring container which is responsible to create and manage bean instances.
* You can create the BeanFactory container as follows:

1. Resource rs = newClassPathResource(“spring-context.xml”);

BeanFactory factory = new XmlBeanFactory(rs);

Hello h = (Hello) factory.getBean(“hello”);

1. Using File System resource:

Resource rs = new FileSystemResouce(“J:/myworkspace/SpringExample/src/spring-context.xml”);

**Lifecycle of a Bean in the BeanFactory container**

* When BeanFactory container creating a bean instance then container will do following tasks:

1. Container loads the Bean class into main memory
2. Container creates the Bean instance with appropriate constructor as per constructor injection.
3. Container will try to inject the Bean dependencies using following two ways:
4. Using XML based autowiring
5. Using explicit wiring
6. If Bean class is implementing BeanNameAware interface then container call setBeanName() method.
7. If Bean class is implementing BeanFactoryAware interface then container calls “setBeanFactory()” method.
8. If Bean class is implementing initializing Bean interface then container calls “afterPropertiesSet()” method.
9. If any custom init method is specified using “init-method” attribute then that will be called by the container.
10. Bean is ready to use.

**When Bean factory container is destroying the bean instance it will do the following task.**

1. If bean class is implementing DiposableBean interface then container call “destroy()” method.
2. If any custom destroy method is specified using “destroy-method” attribute then that method will be called by the container.

**Bean Definition Inheritance**

1. When you are configuring multiple beans inside the spring-context.xml you may find some common properties among the beans with the same values.
2. Configuring same properties repeatedly for multiple beans increases the length of the xml document and decreases readability.

|  |  |
| --- | --- |
| Class Hello{  int x;  int y;  String str;  } | Class Hai{  int x;  int a;  String str;  } |

Normal bean definition:

<bean id = “hello” class = “com.app.Hello” >

<property name = “x” value = “99” />

<property name = “y” value = “88” />

<property name = “str” value = “aaaa” />

</bean>

<bean id = “hai” class = “com.app.Hai”>

<property name = “x” value = “99” />

<property name = “a” value = “111” />

<property name = “str” value = “aaa” />

</bean>

**Inheritance Bean Definition:**

<bean id = “base” abstract = “true”>

<property name = “x” value = “99” />

<property name = “str” value = “aaa” />

<bean>

<bean id = “hello” class = “com.app.Hello” parent = “base”>

<property name = “x” value = “0” />

<property name = “y” value = “88” />

</bean>

<bean id = “hai” class = “com.app.Hai” parent = “base”>

<property name = “a” value = “111” />

</bean>

**Externalization Bean Properties**

1. Write the required number of properties fields and place them under src folder of the project.

For eg:

message1.properties

msg.x=99

msg.y=88

message2.properties

msg.str=aaa

1. Register the property place holder configure with spring container as follows:

<bean id = “x” class = “org.springframework.beanfactory.config.PropertyPlaceholderConfigurer”>

<property name = “locations”>

<list>

<value>message1.properties</value>

<value>message2.properties</value>

</list>

</property>

</bean>

1. Select the bean to externalize its properties.

|  |
| --- |
| Class Hello{  int x;  int y;  String str;  } |

1. Access the required keys from the properties files as follows:

<bean id = “hello” class = “com.app.Hello”>

<property name = “x” value = “${msg.x}” />

<property name = “y” value = “${msg.y}” />

<property name = “str” value = “${msg.str}” />

</bean>

**Accessing properties from Message Bundles:**

1. When you are developing any web based application you may get a requirement to implement internationalization(i18).
2. When you are supporting multiple languages for your application you need to write multiple message bundles with some name combination.

For eg:-

messages.properties

messages\_hi.properties

messages\_en.properties

1. If you want to get the value of required key from the message bundle you need to do the following:
   1. Write the required one or more message bundles and placed them under src folder.
   2. Register the “ResourceBundleMessageSource” Interface.

<bean id = “messageSource” class = “org.springframework.context.support.ResourceBundleMessageSource”>

<property name = “baseName”>

<value>messages</value>  
 </property>

</bean>

* 1. Access the values of the required keys using the methods of MessageSource interface through application context object.

ctx.getMessage(“login.un.required”, null, null);

ctx.getMessage(“login.em.required”, null, “Email is Required.”,null);

ctx.getMessage(“app.length”, new Object[]{“username”,”5”,10}, null);

**Properties Editors:**

1. Property editors are responsible to edit the property of the bean to convert from one format to another format.

For eg:-

When you are configuring the bean properties inside spring-context.xml you are supplying the value in similar way for primitives, wrappers and String. But these values will be converted to corresponding types.

1. Spring provides multiple built in property editors, which will be called by the spring container automatically whenever it is required.
2. If required you can write your own custom property editors as follows:
   1. Write your own property editor class by extending “PropertyEditorSupport” class which is available in “java.bean” package.
   2. Override the following method.

public void setAsText(String text);

* 1. Write the required conversion logic inside “setAsText(String)” method.
  2. After developing the required number of editors. Register all the editors with the spring container.

<bean class = “org.springframework.bean.factory.config.CustomEditorConfigurer”>

<property name = ”customEditors”>

<map>

<entry key = “com.app.spring.ioc.Fee” />

<entry key = ”com.app.spring.ioc.FeeEditors” />

</map>

</property>

**Publishing Application Events:**

Steps to write application event

1. Write your custom event class by extending “ApplicationEvent” class.
2. Declare the variable to store the information related to your event.
3. Write the two augmented constructor, one with the source of an event, second with information you have to pass along with event.

**Steps to write Application Listener**

1. Write your own listener class by implementing ApplicationListener interface.
2. Override the following method:

public void onApplicationEvent() of ApplicationEvent.

1. Implement the required logic after receiving the event inside “onApplicationEven()” method.
2. Register your application listener with the spring container.

**AOP (Aspect Oriented Programming)**

* AOP is a new kind of programming language whose main goal is to provide clean separation between cross cutting concerns and core business logic.

**AOP Terminology:**

1. Aspect
2. Advice
3. Join Point
4. Point Cut
5. Advicer
6. Introduction
7. Target
8. Proxy
9. Weaving
10. **Aspect:**

Aspect is a cross cutting concern which is commonly required for various requirement of a module and various modules of your enterprise application.

For eg:

Logging, Security and transaction or the aspects which are commonly required.

1. **Advice:**

Advice is a java class which has actual implementation of an aspect.

For eg:

Advice is a java class where logging related codes.

1. **Join Point:**

Join Point is a point in a program execution where exactly you want to apply advices.

Spring AOP supports various joint points related to method.

For eg:

1. Method before
2. Method returning successfully
3. Method throwing exception

Aspect J framework provides field level advices also.

1. **Point Cut:**

Point Cut is collection of joint points.

1. **Advicer**

Advicer is a combination of point cut and advice i.e. advices know where to apply (advice) and what to apply(advice).

1. **Introduction**

Introduction is a process of adding new variables and methods to the class dynamically at runtime.

1. **Target**

Target is an object on which you want to apply the advice.

1. **Proxy**

Proxy is an object which you will get after applying advice to the target object.

1. **Weaving:**

Weaving is the process of applying advicers to the target object at joins point to get the proxy objects.

* Weaving can be done in three ways:

1. Compile time weaving (Source Code level)
2. Load Time Weaving (Byte Code Level)
3. Run Time Weaving (Native code level)

* Following are various AOP providers available in market:

1. Spring AOP
2. AspectJ AOP
3. JBoss AOP

* Spring offers three different ways to implement AOP

1. Classical model (Using AOP API)
2. Annotation Based Model
3. Schema Based Model

b and c are available in Spring 2 onwards.

1. Implementing AOP using AOP API (Classical modal)

* Spring AOP API provides the following four types of advices.

1. Method Before Advice
2. After Returning Advice
3. Throws Advice
4. Method Interceptor

**Point Cuts and Advicers**

Spring provides two ways to define the point cuts:

1. jdk regular expression based point cut
2. Aspect expression based point cuts
3. jdk regular expression based point cut (reg exp):

<bean id = “appPointCut” class = “org.springframework.aop.support.JdkRegexpMethodPointCut”>

<property name = “pattern” value =”.add\*()” />

</bean>

1. AspectJ expression based point cuts:

<bean id = “org.springframework.aop.aspectj.AspectJExpressionPointCut”>

<property name = “expression” value = “execution(public \*Hello.m\*(..))”/>

</bean>

AspectJ expression syntax

execution( modifiers Patterns ? return type pattern declaring type pattern ? name-pattern(param-pattern) throws-pattern ?)

eg:-

execution( public \* Hello.m\*(..))

execution( int \*.add\*(..))

execution( \* \* service.\*(..))

execution( \* com.app.p1.Hello\*.\*(..))

execution( \* com.app.p1.Hello\*.\*(..) throws HelloException)

execution( \*service.\*(..)) //-> invalid expression

**Advicer**

* Advicer should know what to (advice) apply and where to apply(point cut)
* Configure the following to form the advicer

<bean id = “mba” class = “com.app.MBAdvice” />

<bean id = “mbAdvicer” class = “org.springframework.aop.support.DefaultPointCutAdvicer” />

<property name = “advice” ref = “mba” />

<property name = “pointcut” ref = “appPointCut”/>

</bean>

**Autoproxying**

1. If you want to create proxy objects for any given target object you need to configure your bean in the “spring-context.xml” with ProxyFactoryBean class.
2. If you have more business services to apply the advicers than you need to configure more beans with ProxyFactoryBean to create the proxy objects. This increases the complexity for the developer and also increases the size of your “spring-context.xml”.
3. To avoid this autoproxy concept is provided in spring2.0.
4. Autoproxying is responsible to detect the beans automatically to create the proxy objects without configuring ProxyFactoryBean.
5. You can implement autoproxy in two ways:
6. Annotation based autoproxying
7. Xml based autoproxying
8. To implement any of the above stated autoproxy way you need to do the following steps commonly.
9. Enable the AOP namespace.
10. Write the following tag to enable the autoproxy.

<aop : aspectj-autoproxy />

1. Schema based support (Xml based autoproxying)
   * + Write the program
2. Annotation based autoproxying
   * + Write the program

**Spring Data Access**

**Question:** What is the importance of DAO design pattern?

**Question:** How many ways are there to establish the connection with database server?

**Question:** What is a transaction and what is the importance of managing transaction?

**Question:** What are ACID properties?

**Question:** What are transaction concurrency problem?

**Question:** What are transaction isolation levels?

**Question:** Flat transaction vs Nested Transaction?

**Question:** Locale transaction vs distributed transaction?

**Question:** How to manage the transaction in JDBC, Hibernate, JPA and EJB?

**Question:** How to specify the required isolation levels in JDBC, Hibernate, JPA and EJB?

**Spring DAO Support:**

1. DAO design patterns main goal is to decouple persistence layer components to business layer components.
2. Spring supports DAOs in two ways:
3. Using Built in DAO classes
4. Using DataAccessExceptionAccessHieararchy
5. Using Built in DAO classes:

* DaoSupprots
* JdbcDaoSupports
* HibernateDaoSupports
* JpaDaoSupports
* IbatisDaoSupports

1. Using DataAccessExceptionAccessHieararchy
2. Spring provides specialized exception hierarchy which is to solve the problem associated with persistence provider implementation like Jdbc, Hibernate, JPA.
3. In Jdbc and Hibernate we will get always SQLException and HibernateException for any kind of underlying problem i.e. on categorization of exceptions.
4. The exception thrown by Jdbc API and HIberntate API are compile time exception so you need to report about these exceptions every time either by writing try catch repeatedly or by propagating.

* These problems are eliminated in spring with clear categorization of exceptions.
* For all exception DataAccessException is the root exception and is RuntimeException.

**SpringDataSourceSupprots**

You are established the connection with the database or data source in two ways.

1. Driver Manager Data Source:
2. JNDI Data Source:
3. Driver Manager Data Source:

To use Driver Manager Connection you need to configure the following bean in the spring-context.xml.

<bean id = “ds” class = “org.springframework.jdbc.datasourc.DriverManagerDataSource”>

<property name = “driverClassName” value = “…….”/>

<property name = “url” value = “…….”/>

<property name = “username” value = “…….”/>

<property name = “password” value = “…….”/>

</bean>

1. JNDI Data Source:

* Make sure that some application server is installed and running.
* Configure the data source in application server (assume that AppDataSourceJNDI is the name specified for the JNDI name of the Data Source).

<bean id = “ds” class = “org.springframework.jndi.JndiTemplat” />

<property name = “environment” />

<props>

<prop key = “Context.INITIAL\_CONTEXT\_FACTORY”>weblogic.jndi.WLInitialContextFactory</prop>

<prop key = “Context.PROVIDER\_URL”>t3://localhost:7001</prop>

</props>

</property>

</bean>

<bean id = “ds” class = “org.springframework.jndi.JndiObjectFactoryBean”>

<property name = “jndiName” value = “AppDataSourceJNDI”>

<property name = “jndiTemplat” ref = “jndiTemp” />

</bean>

**Spring Transaction Support**

Spring provide a uniform way to perform the transaction with various persistence technologies and frameworks.

Spring provides special transaction manager called PlateformTransactionManager (which is root for all the transaction in Spring).

* Following are the concrete implementation of PlateformTransactionManagerInterface for various persistence providers.

1. DataSourceTransactionManager
2. HibernateTransactionManager
3. JPATransactionManager

You can implement the transaction in spring in the following ways:

1. Programmatic transaction
2. Declarative transaction
3. Using TransactionProxyFactory bean
4. Using Namespace support
5. Using Annotation support

**Spring with JDBC**

**Spring with Hibernate**

**Spring with JPA**

**Spring Transaction Management**

1. When you are managing the transaction in spring you need to indentifying required transaction attributes and required isolation levels.
2. There are six transactional attributes which are provided as constants in TransactionDefinition interface follows:

PROPAGATION\_REQUIRED

PROPAGATION\_REQUIRED\_NEW

PROPAGATION\_SUPPORTS

PROPAGATION\_NOT\_SUPPORTED

PROPAGATION\_MANDATORY

PROPAGATION\_NEVER

|  |  |  |
| --- | --- | --- |
|  | New transaction state | Parent transaction state |
| REQUIRED | New transaction will be started. Your method will run in new transaction. | Your method will run in parent transaction. |
| REQUIRED\_NEW | SAME | Parent transaction will be suspended, new transaction will be started. Your method will run on new transaction. Parent transaction will be resumed. |
| SUPPORTS | Your method will run in new transaction. | Your method will run in parent transaction. |
| NOT\_SUPPORTED | Your method will run in new transaction. | Your method will run in parent transaction. |
| MANDATORY | Exception will be thrown | Your method will run in parent transaction. |
| NEVER | Your method will run in new transaction. | Exception will be thrown. |

1. To avoid the transaction concurrency problems you need to select required transaction isolation levels.
2. In Spring all the isolation levels are defined in “TransactionDefinition” interface as follows:
3. ISOLATION\_READ\_UNCOMMITED
4. ISOLATION\_READ\_COMMITED
5. ISOLATION\_REPEATABLE\_READ
6. ISOLATION\_SERIALIZABLE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Dirty Read** | **Repeatable Read** | **Phantom Read** | **Locking** |
| **ISOLATION\_READ\_UNCOMMITED** | √ | √ | √ |  |
| **ISOLATION\_READ\_COMMITED** | X | √ | √ | Column Lock |
| **ISOLATION\_REPEATABLE\_READ** | X | X | √ | Row Lock |
| **ISOLATION\_SERIALIZABLE** | X | X | X | Table Lock |

* Spring offers four different ways to implement the transaction:

1. Programmatic transaction
2. Declarative transaction using TransactionProxyBean
3. Declarative transaction using Annotation
4. Declarative transaction using namespace based
5. Programmatic transaction with Spring:
6. **With JDBC**
7. Configure data source (either DriveManager or DataSource or JNDIDataSource).
8. Configure DataSorceTransactionManager, org.springframework.jdbc.datasource and inject data source.
9. Configure JdbcTemplat or SimpleJdbcTemplat and inject data source.
10. Configure the required DAO and inject both DataSourceTransactionManager and JdbcTemplat
11. Write the DAO implementation as follows:

public class JdbcHelloDao{

@Autowired

JdbcTemplat jtemp;

@Autowired

DataSourceTransactionManager txManager;

void add(){

DefaultTransactionDefinition txdef = new DefaultTransactionDefinition();

txdef.setIslationLevel(TransactionDefinition.ISOLATION\_READ\_COMMITED);

txdef.setPropagationBehaviour(TransactionDefinition.PROPAGATION\_REQUIRES\_NEW);

TransactionState ts = null;

try{

ts = txManager.getTransaction(txdef);

……..

txManager.commint(ts);

} catch(Exception e){

txManager.rollback(ts);

}

}

1. **With Hibernate**
2. **With JPA**