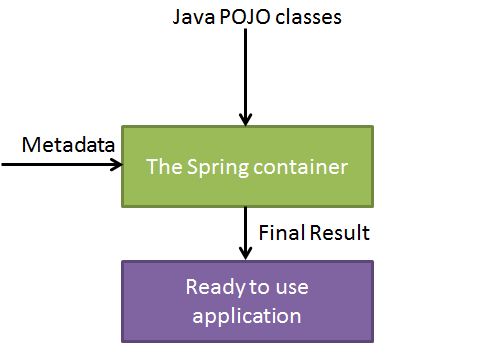
# Spring IoC Containers



Spring BeanFactory Container  
This is the simplest container providing basic support for DI and defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.

Spring ApplicationContext Container   
This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContext interface.

public class MainApp {

public static void main(String[] args) {

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

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

}

}

Why http are there in spring xsd defination?

Spring xml {should be placed in src folder}

-spring tag required xsd

<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">

xmlns:xsi , xsi:schemaLocation are name spaces.

http:// these are just string, only role to identifies xsd in local jars

but as starndard convention spring used http:// in these for user benefit if user wants to browse directly then he can do than.

**Spring Bean Scopes**

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext

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

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

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

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

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

**Spring bean Aware interface:**

All have unimplemented setter method having reference to particular object.

These are for beans. You need to implment this in bean class. No configuration required in spring.xml.

While loading application IOC container scan all beans class and call respective setter method.

ApplicationContextAware

ApplicationEventPublisherAware

BeanClassLoaderAware

BeanNameAware

# Spring Bean Life Cycle Callbacks

One Way:

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

*Below method name can be anyname.*

<bean id="exampleBean"

class="examples.ExampleBean" init-method="init"/>

<bean id="exampleBean"

class="examples.ExampleBean" destroy-method="destroy"/>

Second Way: If used both this one get priority over previous way.

**public** **class** BeanExample **implements** InitializingBean,DisposableBean{

@Override

**public** **void** afterPropertiesSet() **throws** Exception {

}

@Override

**public** **void** destroy() **throws** Exception {

}

}

Third Way: BeanPostProcessor runs for each and every bean.

**public** **class** DisplayNameBeanPostProcessor **implements** BeanPostProcessor {

@Override

**public** Object postProcessBeforeInitialization(Object bean, String beanName)

**throws** BeansException {

**return** **null**;

}

@Override

**public** Object postProcessAfterInitialization(Object bean, String beanName)

**throws** BeansException {

**return** **null**;

}

}

Provide bean defination of this class in spring class.

We can have as many as post processor we want. And even can configure order of execution

Other Way:

**public** **class** DisplayNameBeanPostProcessor **implements** BeanFactoryPostProcessor {

@Override

**public** **void** postProcessBeanFactory(

ConfigurableListableBeanFactory beanFactory) **throws** BeansException {

}

}

Provide bean definition of this class in spring class.

**Order of proccessing:**

# Spring Dependency Injection

Constructor-based dependency injection  
Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

Setter-based dependency injection  
Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

# Spring Bean Definition Inheritance

<bean id="helloWorld" class="com.tutorialspoint.HelloWorld">

<property name="message1" value="Hello World!"/>

<property name="message2" value="Hello Second World!"/>

</bean>

<bean id="helloIndia" class="com.tutorialspoint.HelloIndia"

parent="helloWorld">

<property name="message1" value="Hello India!"/>

<property name="message3" value="Namaste India!"/>

</bean>

If we re add the property in child bean its ovverrides the parent property entry.

If a Collection is filled to half in parent bean then how we proceed further in child bean to add more entry?

# Spring Injecting Inner Beans

Scope is limited to outer bean only.

<bean id="outerBean" class="...">

<property name="target">

<bean id="innerBean" class="..."/>

</property>

</bean>

# Spring Injecting Collection

<list> This helps in wiring ie injecting a list of values, allowing duplicates.

<set> This helps in wiring a set of values but without any duplicates.

<map> This can be used to inject a collection of name-value pairs where name and value can be of any type.

<props> This can be used to inject a collection of name-value pairs where the name and value are both Strings.

<property name="addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

<property name="addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

<property name="addressMap">

<map>

<entry key="1" value="INDIA"/>

<entry key="2" value="Pakistan"/>

<entry key="3" value="USA"/>

<entry key="4" value="USA"/>

</map>

</property>

<property name="addressProp">

<props>

<prop key="one">INDIA</prop>

<prop key="two">Pakistan</prop>

<prop key="three">USA</prop>

<prop key="four">USA</prop>

</props>

</property>

# Setting bean property value from properties file:

<bean

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="location">

<value>database.properties</value>

</property>

</bean>

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

Above configuration need to be done in Spring.xml

# Spring Annotations

# @Required, @Autowired, @Qualifier, other JSR annotation

# To use annotation we need to add relative beanPostProcessor entry into spring.xml

# As for @Reqiuired – RequiredAnnotationBeanPostProcessor

# As for @Autowired – AutowiredAnnotationBeanPostProcessor

# Spring Beans Auto-Wiring

# Its looks for the type

# Its looks for the name

# Then if for Qualifier

# For annotation bean post processor we have one short cut: <context:annotation-config>

Difference between <context:annotation-config> vs <context:component-scan>?

# <context:annotation-config> is used to activate annotations in beans already registered in the application context (no matter if they were defined with XML or by package scanning).

# <context:component-scan> can also do what <context:annotation-config> does but <context:component-scan> also scans packages to find and register beans within the application context.

# Event Handling in Spring

Core of Spring - ApplicationContext, which manages complete life cycle of the beans. ContextStartedEvent is published when the context is started and ContextStoppedEvent is published when the context is stopped.

## Listening to Context Events:

**public** **class** BeanExample **implements** ApplicationListener<ApplicationEvent>{

@Override

**public** **void** onApplicationEvent(ApplicationEvent event) {

}

}

ApplicationEvent - Class to be extended by all application events.

Example:

ContextRefreshedEvent, ContextStartedEvent, ContextStoppedEvent, ContextClosedEvent, RequestHandledEvent

**public** **class** BeanExample **implements** ApplicationListener< ContextRefreshedEvent >{

@Override

**public** **void** onApplicationEvent(ContextRefreshedEvent event) {

}

}

# Custom Events in Spring

# AOP with Spring Framework - as seprate file

# Spring JDBC Template – as seprate file

# Spring Transaction Management- as seprate file

# Spring Logging with Log4J

No configuration is required in spring xml for instance of logger.

## Ussing Log4J API

1. **Add this jar to classpath :** *log4j-x.y.z.jar*
2. Create log4J configuration file *log4j.properties* under the **src** folder.
3. static Logger log = Logger.getLogger(MainApp.class.getName());
4. **log.info("Ur logging message");**

## log4j.properties

# Define the root logger with appender file

log4j.rootLogger = DEBUG, FILE

# Define the file appender

log4j.appender.FILE=org.apache.log4j.FileAppender

# Set the name of the file

log4j.appender.FILE.File=C:\\log.out

# Set the immediate flush to true (default)

log4j.appender.FILE.ImmediateFlush=true

# Set the threshold to debug mode

log4j.appender.FILE.Threshold=debug

# Set the append to false, overwrite

log4j.appender.FILE.Append=false

# Define the layout for file appender

log4j.appender.FILE.layout=org.apache.log4j.PatternLayout

log4j.appender.FILE.layout.conversionPattern=%m%n

## Using Jakarta Commons Logging (JCL) API

1. **Add this jar to classpath :** *commons-logging-x.y.z.jar*
2. Create configuration file under the **src** folder same as above.
3. static Log log = LogFactory.getLog(MainApp.class.getName());
4. **log.info("Ur logging message");**

# Spring MVC Framework - as seprate file