Spring Register and Activation Bean

<context:annotation-config> vs <context:component-scan>

<context:annotation-config> is used to activate applied annotations in already registered beans in application context**.**

<context:component-scan>register the beans in context + it also scans the annotations inside beans and activate them

MyBeanA component class

```
package com.spring.beans.example;

import org.springframework.stereotype.Component;

@Component
public class MyBeanA {

	public MyBeanA(){
		System.out.println("Creating Bean MyBeanA");
	}
}
```

MyBeanB Component class

```
package com.spring.beans.example;

import org.springframework.stereotype.Component;

@Component
public class MyBeanB {

	public MyBeanB(){
		System.out.println("Creating Bean MyBeanB");
	}
}
```

MyBeanC Component class

```
package com.spring.beans.example;

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

@Component
public class MyBeanC {


	private MyBeanA beanA;
	private MyBeanB beanB;

	public MyBeanC(){
		System.out.println("Creating Bean MyBeanC");
	}

	public MyBeanA getBeanA() {
		return beanA;
	}

	@Autowired
	public void setBeanA(MyBeanA beanA) {
		System.out.println("Injecting MyBeanA refrence with MyBeanC");
		this.beanA = beanA;
	}
	public MyBeanB getBeanB() {
		return beanB;
	}

	@Autowired
	public void setBeanB(MyBeanB beanB) {
		System.out.println("Injecting MyBeanB refrence with MyBeanC");
		this.beanB = beanB;
	}
}
```

SpringBeanActivationDemo class

```
package com.spring.beans.example;

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

public class SpringBeansActivationDemo {

	public static void main(String[] args) {
		ApplicationContext appContext=new 
ClassPathXmlApplicationContext("spring.xml");
	}

}
```

Scenarios:

Define only bean tags:

```
<bean id="myBeanA" class="com.spring.beans.example.MyBeanA"/>

<bean id="myBeanB" class="com.spring.beans.example.MyBeanB"/>

<bean id="myBeanC" class="com.spring.beans.example.MyBeanC"/>
```

OUTPUT:

```
Creating Bean MyBeanA

Creating Bean MyBeanB

Creating Bean MyBeanC
```

Define bean tags and inject by property ref attributes

```
<bean id="myBeanA" class="com.spring.beans.example.MyBeanA"/>
<bean id="myBeanB" class="com.spring.beans.example.MyBeanB"/>
<bean id="myBeanC" class="com.spring.beans.example.MyBeanC">
        <property name="beanA" ref="myBeanA"/>
        <property name="beanB" ref="myBeanB"/>
</bean>
```

OUTPUT:

```
Creating Bean MyBeanA
Creating Bean MyBeanB
Creating Bean MyBeanC
Injecting MyBeanA refrence with MyBeanC
Injecting MyBeanB refrence with MyBeanC
```

Using only <context:annotation-config />

OUTPUT:

```
NO OUTPUT
```

Using <context:annotation-config /> with bean declarations

```
<context:annotation-config />
<bean id="myBeanA" class="com.spring.beans.example.MyBeanA"/>
<bean id="myBeanB" class="com.spring.beans.example.MyBeanB"/>
<bean id="myBeanC" class="com.spring.beans.example.MyBeanC"/>
```

OUTPUT:

```
Creating Bean MyBeanA

Creating Bean MyBeanB

Creating Bean MyBeanC

Injecting MyBeanA refrence with MyBeanC

Injecting MyBeanB refrence with MyBeanC
```

Using only <context:component-scan />

```
<context:component-scan base-package="com.spring.beans.example" />
```

OUTPUT:

```
Creating Bean MyBeanA

Creating Bean MyBeanB

Creating Bean MyBeanC

Injecting MyBeanA refrence with MyBeanC

Injecting MyBeanB refrence with MyBeanC
```

Using both <context:component-scan /> and <context:annotation-config />

```
<context:annotation-config />

<context:component-scan base-package="com.spring.beans.example" />

<bean id="myBeanA" class="com.spring.beans.example.MyBeanA"/>

<bean id="myBeanB" class="com.spring.beans.example.MyBeanB"/>

<bean id="myBeanC" class="com.spring.beans.example.MyBeanC"/>
```

OUTPUT:

```
Creating Bean MyBeanA

Creating Bean MyBeanB

Creating Bean MyBeanC

Injecting MyBeanA refrence with MyBeanC

Injecting MyBeanB refrence with MyBeanC
```

NOTE: With above configuration we are discovering beans two times and activating annotations two times as well. But output got printed one time only. Why? Because spring is intelligent enough to register any configuration processing only once if it is registered multiple tiles using same or different ways.

Difference between @Component,@Service,@Controller,@Repository

@Component

The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context. To use this annotation, apply it over class as below:

@Repository

We can use more suitable annotation that provides additional benefits specifically for DAOs The @Repository annotation is a specialization of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the **unchecked exceptions** (thrown from DAO methods) eligible for translation into Spring **DataAccessException**.

@Service

The @Service annotation is also a specialization of the component annotation. It doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component in service-layer classes because it specifies intent better

@Controller

@Controller annotation marks a class as a Spring Web MVC controller. It too is a @Component specialization, so beans marked with it are automatically imported into the DI container. When you add the @Controller annotation to a class, you can use another annotation i.e. @RequestMapping; to map URLs to instance methods of a class.

How to enable component scanning

To enable this scanning, you will need to use “context:component-scan” tag in your applicationContext.xml file. e.g.

```
<context:component-scan base-package="com.spring.demo.service" />

<context:component-scan base-package="com.spring.demo.dao" />

<context:component-scan base-package="com.spring.demo.controller" />
```

The context:component-scan element requires a base-package attribute, which, as its name suggests, specifies a starting point for a recursive component search. You may not want to give your top package for scanning to spring, so you should declare three component-scan elements, each with a base-package attribute pointing to a different package.

How will you inject prototype bean into singleton bean scope to ensure that we will get different object of prototype bean?

We can use lookup-method tag to inject prototype bean into singleton bean:

1. Injecting prototype bean into singleton bean using ref keyword

```
<bean id="myEmp" class="com.spring.beans.example.MyEmployee">
        <property name="empname" value="Anil"/>
        <property name="empaddress" ref="empAdd"/>
</bean>
```

```
<bean id="empAdd" class="com.spring.beans.example.EmpAddress" scope="prototype">
        <property name="city" value="Bangalore"/>
        <property name="state" value="KA"/>
</bean>
```

```
ApplicationContext appContext=new ClassPathXmlApplicationContext("spring.xml");
MyEmployee emp=(MyEmployee) appContext.getBean("myEmp");
MyEmployee emp2=(MyEmployee) appContext.getBean("myEmp");
System.out.println(emp);
System.out.println(emp2);
```

Output:

```
MyEmployee created! hashcode:1860215686
EmpAddress Created! hashcode:282106579
MyEmployee [empname=Anil, empaddress=EmpAddress [city=Bangalore, state=KA]]
MyEmployee [empname=Anil, empaddress=EmpAddress [city=Bangalore, state=KA]]
```

2. Injecting prototype bean into singleton bean using lookup-method tag

```
<bean id="myEmp" class="com.spring.beans.example.MyEmployee">
        <property name="empname" value="Anil"/>
        <!-- <property name="empaddress" ref="empAdd"/> -->
        <lookup-method name="getEmpaddress" bean="empAdd"/>
</bean>
<bean id="empAdd" class="com.spring.beans.example.EmpAddress" scope="prototype">
        <property name="city" value="Bangalore"/>
        <property name="state" value="KA"/>
</bean>
```

```
ApplicationContext appContext=new ClassPathXmlApplicationContext("spring.xml");
```

MyEmployee emp=(MyEmployee) appContext.getBean("myEmp");

MyEmployee emp2=(MyEmployee) appContext.getBean("myEmp");

System.out.println(emp);

System.out.println(emp2);

OutPut:

MyEmployee created! hashcode:381312122

EmpAddress Created! hashcode:1068896286

MyEmployee [empname=Anil, empaddress=EmpAddress [city=Bangalore, state=KA]]

EmpAddress Created! hashcode:1237174744

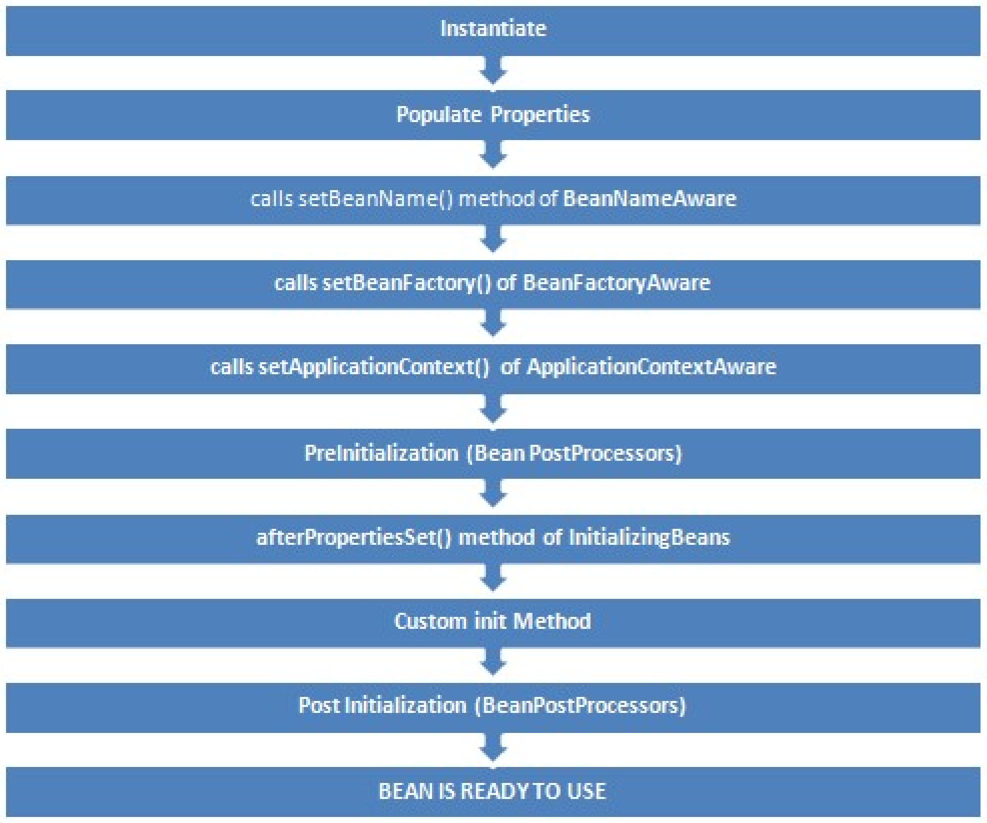
MyEmployee [empname=Anil, empaddress=EmpAddress [city=Bangalore, state=KA]]

Collection Wiring:

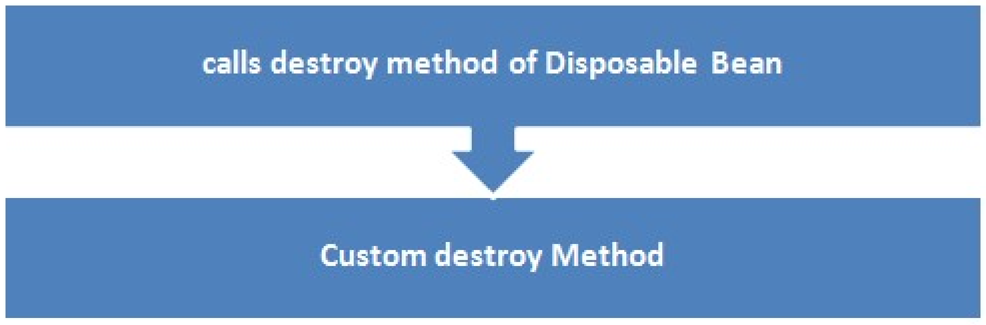
- list
- set
- map
- props

```
<property name="testListValue">
        <list value-type="int">
                <value>5</value>
                <value>15</value>
                <value>5</value>
                <value>20</value>
        </list>
</property>

<property name="emails">
        <props>
                <prop key="administrator">admin@mymail.com</prop>
                <prop key="support">support@mymail.com</prop>
        </props>
</property>

<property name="govtIds">
        <map>
                <entry key="PANCARD" value="BAKSK7836H"></entry>
                <entry key="ELECTIONCARD" value="A123Z004"></entry>
                <entry key="DRIVINGLICENSE" value="DL1020167"></entry>
        </map>
</property>
```

Spring Bean Life cycle


Instantiate
Populate Properties
calls setBeanName() method of BeanNameAware
calls setBeanFactory() of BeanFactoryAware
calls setApplicationContext() of ApplicationContextAware
PreInitialization (Bean PostProcessors)
afterPropertiesSet() method of InitializingBeans
Custom init Method
Post Initialization (BeanPostProcessors)
BEAN IS READY TO USE

Following diagram shows the method calling at the time of destruction.



Spring Bean Life Cycle – InitializingBean, DisposableBean

```
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

import com.spring.beans.Employee;

public class EmployeeService implements InitializingBean, DisposableBean{
	private Employee employee;

	public Employee getEmployee() {
		return employee;
	}

	public void setEmployee(Employee employee) {
		this.employee = employee;
	}

	public EmployeeService(){
		System.out.println("EmployeeService no-args constructor called");
	}

	@Override
	public void destroy() throws Exception {
		System.out.println("EmployeeService Closing resources");
	}

	@Override
	public void afterPropertiesSet() throws Exception {
		System.out.println("EmployeeService initializing to dummy 
value");
		if(employee.getName() == null){
			employee.setName("Anil");
		}
	}
}
```

Spring Bean Life Cycle – Custom post-init, pre-destroy

```
public class MyEmployeeService{

	private Employee employee;

	public MyEmployeeService(){
		System.out.println("MyEmployeeService no-args constructor 
called");
	}

	//pre-destroy method
	public void destroy() throws Exception {
		System.out.println("MyEmployeeService Closing resources");
	}

	//post-init method
	public void init() throws Exception {
		System.out.println("MyEmployeeService initializing to dummy 
value");
		if(employee.getName() == null){
			employee.setName("Anil");
		}
	}
}
```

Spring Aware Interfaces

Sometimes we need Spring Framework objects in our beans to perform some operations, for example reading ServletConfig and ServletContext parameters or to know the bean definitions loaded by the ApplicationContext. That's why spring framework provides a bunch of *Aware interfaces that we can implement in our bean classes.

Some of the important Aware interfaces are:

- **ApplicationContextAware** – to inject ApplicationContext object, example usage is to get the array of bean definition names.
- **BeanFactoryAware** – to inject BeanFactory object, example usage is to check scope of a bean.
- **BeanNameAware** – to know the bean name defined in the configuration file.
- **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
- **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

```
public class MyAwareService implements ApplicationContextAware, BeanClassLoaderAware,
BeanFactoryAware,BeanNameAware,   ResourceLoaderAware {

        @Override

        public void setApplicationContext(ApplicationContext ctx)throws BeansException {

                System.out.println("setApplicationContext called");

                System.out.println("setApplicationContext:: Bean Definition Names="+
Arrays.toString(ctx.getBeanDefinitionNames()));

        }

        @Override

        public void setBeanName(String beanName) {

                System.out.println("setBeanName called");

                System.out.println("setBeanName:: Bean Name defined in context="+ beanName);

        }

        @Override

        public void setBeanClassLoader(ClassLoader classLoader) {

                System.out.println("setBeanClassLoader called");
```

```
                        System.out.println("setBeanClassLoader:: ClassLoader Name="+
classLoader.getClass().getName());

            }

            @Override

            public void setResourceLoader(ResourceLoader resourceLoader) {

                        System.out.println("setResourceLoader called");

                        Resource resource = resourceLoader.getResource("classpath:spring.xml");

                        System.out.println("setResourceLoader:: Resource File Name="+ resource.getFilename());

            }

            @Override

            public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

                        System.out.println("setBeanFactory called");

                        System.out.println("setBeanFactory:: employee bean singleton="

                                                + beanFactory.isSingleton("employee")); }}
```

Spring Validation: **Spring's Validator interface**

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

Let's consider a small data object:

```
public class CustomerValidator implements Validator {

    public boolean supports(Class clazz) {
        return Customer.class.equals(clazz);
    }
    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Customer cust = (Customer) obj;
        if (cust.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (cust.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

```
@Controller

public class CustomerController {

	private static final Logger logger = LoggerFactory

			.getLogger(CustomerController.class);

	private Map<String, Customer> customers = null;

	public CustomerController(){

		customers = new HashMap<String, Customer>();

	}

	@RequestMapping(value = "/cust/save.do", method =
RequestMethod.POST)

	public String saveCustomerAction(

			@Valid Customer customer,

			BindingResult bindingResult, Model model) {

		if (bindingResult.hasErrors()) {

			logger.info("Returning custSave.jsp page");

			return "custSave";

		}

		logger.info("Returning custSaveSuccess.jsp page");

		model.addAttribute("customer", customer);

		customers.put(customer.getEmail(), customer);

		return "custSaveSuccess";

	}

}
```

Thank You!!