### Spring Core

Spring is most widely used Java EE framework.

Spring framework can be used in normal Java applications **to achieve loosely coupling** between different components **by implementing the dependency injection** and we can perform cross cutting tasks such as logging and authentication using spring support for AOP.

Spring MVC framework can be used to create web application as well as restful web services

Spring Context – Dependency Injection

Spring AOP – Aspect Oriented Programming

Spring DAO – for Database operations using DAO pattern

Spring JDBC – for JDBC and data source support

Spring ORM – for ORM tools support such as Hibernate

Spring web – for creating web applications.

Spring MVC – Model view controller implementations for creating web applications and web services.

**Inversion of control** is a principle of software engineering **by which control of objects is transferred to a container** or framework.

The container **is responsible to Inject those dependencies when it creates the bean**. This process is basically the **inverse of the bean** **itself controlling the instantiation or location of its dependencies**.

**Dependency Injection** is dependency of your objects are managed by container so that object can concentrate on functionality rather than searching for dependencies.

Dependency Injection **allows us to remove the hardcoded dependencies** and make our application **loosely coupled, extendable and maintainable**.

We can **implement dependency injection to move dependency resolution from compile time to run time**.

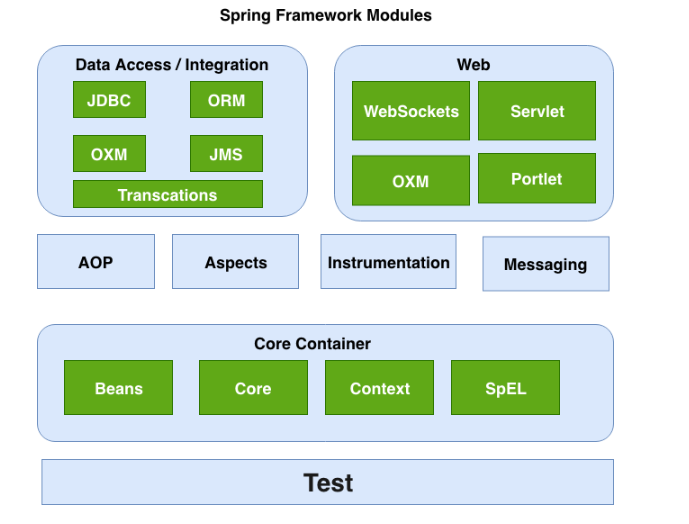
**Some of the benefits of using dependency injection** are

Separation of concerns, Boilerplate code reduction, configurable components and easy unit testing.

Dependency Injection is a process **whereby object define their dependencies through** constructor Injection, Setter Injection and Field level Injection.

**Autowiring**

* Autowiring **allows** the spring container to automatically resolve dependencies between collaborating objects.
* **We can use** @Autowired annotation **to mark the dependency** **which spring is going to resolve and inject it**.
* **We can use** @Autowired annotation with constructor, setter method, and field level.



**Constructor based Injection**

* **In Constructor based Injection, a constructor is used to inject a dependent object.**
* **Constructor based Injection is accomplished when the container invokes a constructor with a number of arguments.**
* **Constructor based Injection is used to resolve the dependencies of a dependent object.**
* Constructor based DI means that **required dependencies passed into the class at the time of instantiation**.

**Advantages**

* It’s suitable for **mandatory dependencies**. In a constructor-based DI **, you can be sure that the objects is ready to be used** **the moment is constructed**.
* The code structure is **very compact and clear to understand**.

**Disadvantages**

* It **may cause Circular dependency between objects**. Circular dependency means two objects depend on each other. For resolving that, we **should use** setter injection instead of a constructor injection.

|  |  |
| --- | --- |
| public class Foo{  @Autowired  public Foo(Bar bar, Baz baz){  } | <bean name="foo" class="C.Foo">  <constructor-arg>  <bean class="C.Bar"></bean></constructor-arg>  <constructor-arg>  <bean class="C.Baz"></bean></constructor-arg>  </bean> |

**Setter based Injection**

* **In a setter-based injection, the dependent object is provided by setter method in the dependent class.**
* **Setter-based injection is accomplished by calling the setter method on beans after invoking the no-args constructor through the container.**
* We can inject the dependency by setter methods. You pass the dependency objects in the setter method.
* **Use setter-based DI for optional dependencies** or if you need to reinject the dependencies later.
* Setter based DI is accomplished by the container will call the setter-based method of the class after invoking the no argument constructor or no-argument static factory method to instantiate the bean.

|  |  |
| --- | --- |
| public class Foo{  Bar bar;  Baz baz;  @Autowired  public void setBar(Bar bar){  this.bar = bar;  }  @Autowired  public void setBaz(Baz baz){  this.baz=baz;  } | <bean name="foo" class="C.Foo">  <property name="bar" class="C.Bar"></property>  <property name="baz" ref="baz"> <property>  </bean>  <bean id="baz" class="C.Baz"></bean> |

**Advantages**

* **It is more readable than the constructor injection**.
* **This is useful for non-mandatory dependencies**.
* **It solves the circular dependency problem in the application**.
* It **help us to inject the dependency only when it required.**
* **It is possible to reinject the dependencies, but it is not possible in constructor injection.**

**Disadvantage**

* There is no guarantee in a setter-based injection that the dependency will be injected.
* One can use setter-based DI to override another dependency. This can cause the security issues in a spring application.

**Field based Injection**

* Field based injection is **easy to use**, and it **has clean code compared to the other two types of dependency** **injection**.
* This type of DI has **the benefit of removing the clutter code over setter or constructor-based dependencies**, but **it has many drawbacks such as dependencies are invisible from the outside.**
* In a constructor-based and setter-based dependencies, **classes clearly exposes those dependencies using constructor and public interface method or setter method.**
* In a field-based DI, **the class is inherently hiding the dependencies from the outside the world**. Another difficulty is that **field injection cannot be used to assign dependencies to final/immutable fields**, as these **fields must be instantiated at class instantiation**.

Public class Company {

@Autowired

Products products;

}

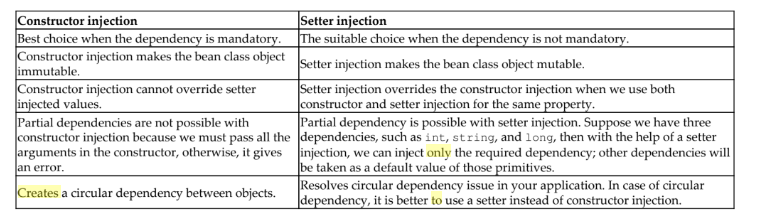
**When to use constructor-based DI and Setter based DI**

- Constructor Injection should be **used** **only on required bean properties**.

Using constructor injection on optional bean properties require large number of constructors to handle different combination of properties.

- Use Setter based DI, it **accommodates a large number of optional properties** and **make the bean more flexible** by allowing for re-injection on dependencies.

The generally accepted best practices is to use constructor injection for mandatory dependencies and setter injection for optional dependencies.



**Spring Scope**

We use @Scope annotation **to define the scope of the bean** **in the** **@Component class or @Bean definition**.

It can be singleton, prototype, request, session, application.

**Singleton scope**

* When you define the scope as Singleton, (the **Spring IOC Container** ) It **creates exactly one instance of object**.
* **Same instances will be returned all subsequent requests**.
* **Spring stores all singleton bean instances in a cache, all the subsequent request of the bean returns from cached object**.
* **Default scope is singleton**.
* **ApplicationContext instantiates singleton bean** **when the container is started**, **it doesn’t wait for getBean() method to be called**.

|  |  |
| --- | --- |
| @Component  @Scope(value= ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***)  **public** **class** SecondBean {  } | @Configuration  Public class AppConfig{  @Bean  @Scope(“singleton”)  Public SecondBean secondBean(){  Return new SecondBean();  }  } |

If bean scope is singleton, then IoC container creates the bean class object and keeps in the HashMap element as value by having bean ‘id’ as key and uses that object across the multiple “factory.getBean()” method.

**Prototype scope**

* When you define the scope as Prototype, the Spring IOC **Container creates new bean instance every time a bean is requested.**
* You request it through getBean() method.
* Prototype scoped beans **are mostly used for stateful beans**.

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

Country c1 = ctx.getBean(“country”); //creates a new bean instance if the scope is prototype.

Country c2 = ctx.getBean(“country”); //creates a new bean instance if the scope is prototype.

When we declare a bean scope as a prototype then IoC container doesn’t keep the created bean class object in HashMap, so it returns new object for every factory.getBean().

Prototype bean **incurs hit on performance during creation**, so when a prototype **uses resource such as network and database connection. It should be avoided completely**.

**As a rule, use singleton as bean scope for stateless session bean, Use prototype bean scope for stateful beans.**

**Request scope**

* When you define the scope as request, Spring container creates a single bean instance as per HttpRequest.
* The request bean scope **is only available in web aware application context**.
* The bean is discarded as soon as the request processing is done.

**Session scope**

* When you define the scope as session, Spring container create a single bean instance as per HttpSession.
* The session bean scope **is only available in web aware application context**.

**Application scope**

* The application bean scope is only available in web aware application context.
* When you define the scope as Application, Spring container creates a single bean instance as per web application.

**Singleton beans with prototype bean dependencies**

* When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time.

**prototype beans with singleton bean dependencies**

* When you use prototype-scoped beans with dependencies on singleton beans, singleton scoped bean dependencies are resolved at instantiation time and not prototyped beans.

**Difference between Singleton class and singleton bean scope**

**Singleton class** is scoped **by per Java class**

**Singleton bean scope** is scoped **by per Spring container**.

Spring Singleton bean referred to as “**a single instance per bean and per container**”. To understand it more, let’s look at the below example

<!-- Singleton is the default scope, you can skip scope attribute in below declaration--!>

<bean id="bean1" class="com.javadevjournal.SingletonBean" scope="singleton"/>

<bean id="bean2" class="com.javadevjournal.SingletonBean" scope="singleton"/>

In the above example, Spring Container will create 2 instances of our SingletonBean class.

In Spring singleton, scope creates one object of that bean as per spring container, meaning if there are multiple spring bean container in single JVM then multiple instances of that bean will be created .

Here container means spring container which is nothing but application context. This internally reads the configuration file and load the classes based on configuration.

Ideally you require only one spring container, so you will create only one application context, but you can create any number of application context.

ApplicationContext context1 = new ClassPathXmlApplicationContext("Beans.xml");

ApplicationContext context2 = new ClassPathXmlApplicationContext("Beans.xml");

Above example, we create two application context and it means that we have two spring containers.

Ideally you require only one spring container.

**Lazy loading** ensures that **beans are loaded on the fly when requested**.

**Preloading** ensures the **beans are loaded before they are used**. The Spring container uses preloading by default. So, loading all classes at the start.

**Spring IOC Container** **is built** as the **core module of the spring architecture**. IOC is also known as Dependency Injection(DI).

The interface org.sprinframework.context.**ApplicationContext** **is represented as Spring IoC Container**, and it is **complete control of bean’s life cycle** and **responsible for instantiating, configuring and assembling beans**.

**The container gets all the instructions to instantiate, configure, and assemble**, **by scanning bean configuration meta data**. **The configuration meta data can be represented using the following methods**:

* XML – based configuration - explicit configuration
* Java – based configuration – explicit configuration
* Annotation based configuration – implicit configuration

**For creating bean instances**, **we first need to instantiate a spring IoC container** **by reading the configuration meta data**. **After initialization of an IoC container**, **we can get the bean instances using the bean name or id**.

Spring provides two types of IoC container implementation.

* Bean Factory
* Application Context

**Bean Factory**

* Bean Factory **is basic container**, it can only manage a bean life cycle. but it can not provide services like Transaction, security, etc.,
* If we develop small scale application, then we use bean factory.
* **Bean factory is lazy initializer. Bean factory container will not create a bean object up to the request time.**
* Bean factory container supports only two scopes (Singleton, prototype).
* Bean factory does not support internalization, event handling, event processing.
* Bean Factory is responsible to source, configure or assemble the dependency between objects.

**Application Context**

* Application Context **is an advanced container**, it manage a bean life cycle and also provide services like Transaction, Security.
* If we are developing enterprise application like web application, distributed application, then Application Context is recommended to use.
* **Application context container creates bean object of singleton bean at the time of loading only. It is eager initializer.**
* Application Context container supports all the bean scope (Singleton, prototype, request, session).
* Application context supports internalization, event handling, event processing.

**Difference between Bean Factory and Application Context**

**BeanFactory** is an interface representing a container that provides and manages bean instances. **The default implementation instantiates beans lazily when getBean() is called.**

**ApplicationContext** is an interface representing a container holding all information, metadata, and beans in the application. It also extends the BeanFactory interface, but **the default implementation instantiates beans eagerly when the application starts.**

The mostly used implementation of ApplicationContext are **FileSystemXmlApplicaitonContext**, **ClassPathXmlApplicationContext**, **AnnotationConfigApplicationContext**.

Spring also provides us with web-aware implementation of ApplicationContext interface, as shown

**XMLWebApplicationContext**, **AnnotationConfigWebApplicationContext**.

**We can use any one of these implementations to load beans into bean factory.**

If we want to load our **Configuration file Beans.xml from** the class path of our application, we can use ClassPathXmlApplicationContext class provided by Spring.

ApplicationContext context = new ClassPathXmlApplicationContext(“Beans.xml”);

If you are using a **Java configuration instead of XML configuration**, you can use AnnotationConfigApplicationContext.

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

After loading configuration file and getting the application context, we can fetch beans from spring container by calling getBean() method of the application context.

BankAccountService bankAccountService = context.getBean(BankAccountService.class);

**XML based Configuration**

The XML based configuration **has been primary configuration technique since spring started**.

We will see the example how the CustomerService object is injected in the BankingService class through an XML based configuration

**Constructor based**

<bean id=”bankingService” class=”com.example.BankingService”>

<constructor-arg ref=”customerService”/>

</bean>

<bean id=”customerService” class=”com.example.CustomerService”/>

**Setter based**

<bean id=”bankingService” class=”com.example.BankingService”>

<property name=”customerService” ref=”customerService”/>

</bean>

<bean id=”customerService” class=”com.example.CustomerService”/>

Context.getBean(“bankingService”); or

Context.getBean(“bankingService”,BankingService.class);

**Java based configuration**

In the Java based configuration, **we must annotate the class with @Configuration and the declaration of the bean can be achieved with @Bean annotation.**

**@Configuration**

@configuration annotation is used at **class level**.

@Configuration annotation is used on classes which defines beans.

Configuration class will have methods to instantiate and configure dependencies.

Its **primary purpose** is **source of bean definitions**.

We use @Configuration annotation in Java based configuration.

**@Bean**

@Bean annotation is used at **method level**.

@Configuration **will have methods to instantiate and configure dependencies**. **Such methods will be annotated with @Bean. It works as bean id and return the actual bean**.

The method is annotated with **@Bean annotation to describe that it is responsible for instantiating, configuring, and initializing new bean is to be managed by the Spring container**.

@Bean annotation supports initialization and destruction call back methods such as initMethod and destroyMethod.

@Bean(initMethod=”init”)

@Bean(destroyMethod=”destroy”)

**@ComponentScan**

* In Spring, **Component scanning is not enabled by default** in java-based configuration. We need to enable it with the @ComponentScan annotation.
* @Configuration annotation allows spring to know the packages to scan for annotated components and to create beans from them.
* ComponentScan is also used to specify base packages using basePackage attribute.

@ComponentScan(basePackage={”com.example”,”com.example1”}) in the AppConfig class.

* If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.
* @ComponentScan **tells the spring where to search for components, configuration and services**.

**Constructor Based Java Configuration**

@Configuration

Public class AppConfig{

@Bean(“bankingService)

Public BankingService getBankingService(CustomerService customerService){

Return new BankingService(customerService);

}

@Bean(name=”customerService”)

Public CustomerService getCustomerService(){

Return new CustomerService();

}

}

**OR**

@Configuration

Public class AppConfig{

@Bean

Public BankingService bankingService(){

Return new BankingService(customerService());

}

@Bean

Public CustomerService customerService(){

Return new CustomerService();

}

}

**Setter based Java Configuration**

@Configuration

Public class AppConfig{

@Bean

Public BankingService bankingService(){

Return new BankingService());

}

@Bean

Public CustomerService customerService(){

Return new CustomerService();

}

}

The method is annotated with **@Bean annotation to describe that it is responsible for instantiating, configuring, and initializing new bean is to be managed by the Spring container**.

By default, method name will be the bean id, however you can override the default behavior using the **name** attribute of the bean definition.

**Annotation based configuration**

Annotation based configuration **is a way of creating a bean**, **where we can move the bean configuration into the component class itself using annotations** on the relevant class, method, or field declaration.

**Annotation based configuration is turned off by default in Spring**. So first, **you have to turn it on by entering the <context:annotation-config/> element into spring XML file.**

Once you configure <context:annotation-config/> element, it indicates start of annotating your code.

Spring should automatically scan the package defined in <context:component-scan base-package=”com.example”/> and identify beans and wire them based on the pattern.

XML based configuration will override annotations because an XML based configuration will be injected after annotations.

**@Autowired annotation** **injects object dependency** **implicitly**.

We can use the @Autowired annotation on a constructor-setter-field-based dependency pattern.

**Constructor based injection**

@Component

Public class BankingService {

Public CustomerService customerService;

@Autowired

Public BankingService(CustomerService customerService){

This.customerService = customerService;

}

}

**Setter based Injection**

@Component

Public class BankingService {

Public CustomerService customerService;

@Autowired

Public void setCustomerService(CustomerService customerService){

This.customerService = customerService;

}

}

**Field based Injection**

@Component

Public class BankingService {

@Autowired

Public CustomerService customerService;

}

**@Autowired require false**

By default, the @Autowired annotation implies that the dependency is required. This means an exception will be thrown when a dependency is not resolved.

You can override the default behavior using the required=false with @Autowired annotation.

@Autowired(required=false)

**@Component**

@Component annotation is class level annotation.

**@Component annotation marks the Java class as bean so that the component scanning of spring can add into application context.**

**@Component annotation** is used to indicate that class is a component. These classes are used for auto detection and configured as bean when annotation-based configuration are used.

@Component annotation are used for automatic bean detection using classpath in spring framework.

During the ComponentScan, spring container automatically detect the classes annotated with @component.

**you can implement InitializingBean and DisposableBean interface in the component class**.

Spring **container calls afterPropertiesSet() for spring initializing bean and destroy() method for spring disposable bean**.

@Component annotation class - we can use @PostConstruct and @PreConstruct in the method.

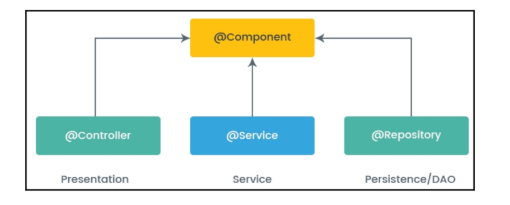
@PostConstruct is used for initialization of the method.

@PreDestroy is used for destroying of the method.

Spring framework provides us with some special annotations. **These annotations are used to create spring beans automatically in the application context**. So, there is no need to configure the bean explicitly using XML based or Java based configuration.

Special Annotations : @Component, @Service, @Repository, @Controller

In Spring, @Service, @Repository, @Controller are meta annotation for the @Component annotation.



@Controller is a **stereo type for the presentation layer** (spring-mvc) **to define controller**.

@Service is a **stereo type for the service layer**. Annotated class **which executes business logic, perform calculations and call external API in the service layer**.

@Repository is a **stereo type for persistence layer**. **It is used in Data Access Layer**. It works as marker for any class that fulfils the role of repository.

CustomerService class is a service layer class, so we should use @Service annotation instead of generic annotation @Component.

CustomerRepository interface is working at Data Access Object(DAO) layer of the application.

The class is annotated with @Repository annotation

**@Primary annotation**

**When there are multiple beans of same type, the NoUniqueBeanDefinitionException will be thrown.**

It **indicates that the spring container is unable to select a bean for DI because more than one eligible candidate**.

**In that case, we use the @primary annotation and take the control of the selection process.**

Public interface customerService {

Public void customerService();

}

@Component

Public class AccountService implements customerService {

Public void customerService() { System.out.println(“Prints Account Service”);}

}

@Component

@primary

Public class BankingService implements customerService {

Public void customerService() { System.out.println(“Prints Bank Service”);}

}

customerService cs = ctx.getBean(customerService.class);

cs.customerService(); //prints Bank Service

**@Qualifier annotation**

* @Qualifier annotation is used along with @Autowired annotation to avoid the confusion when the multiple instances of bean type is present.
* When you need to more control of dependency injection, @Qualifier can be used.
* This annotation is used to avoid the confusion occurs When you create more than one bean of same type and want to wire only one of them with property.
* @Qualifier is used to resolve ambiguous dependencies i.e., it helps to choose one of the dependencies.
* If there are multiple implementation for single interface, then we can use the *@Qualifier annotation* to choose the correct bean for the dependency injection.

Public interface customerService {

Public void customerService();

}

@Component

Public class AccountService implements customerService {

Public void customerService() { System.out.println(“Prints Account Service”);}

}

@Component

Public class BankingService implements customerService {

Public void customerService() { System.out.println(“Prints Bank Service”);}

}

@Component

public class CustomerServiceFacade {

customerService customerService;

@Autowired

**@Qualifier("accountService")**

public void setCustomerService**(**customerService customerService) {

this.customerService = customerService;

}

public void fixQualifier() {

**customerService.customerService();**

}

}

ServiceSetterBasedInjection cbi = context.getBean(ServiceSetterBasedInjection.class);

cbi.fixQualifier();

**@Qualifier annotation works in the way of the strategy pattern.**

**@Lazy**

By default, **spring container creates and initializes all singleton bean at the time of application start-up**;

If you want to initialize a bean lazily, you can use @lazy annotation over class.

@Lazy annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.

When we use @Lazy annotation, that means the bean will be created and initialized only when it is first requested for.

We can control the pre-initialization of beans at the start-up by using @Lazy annotation.

@lazy annotation can be used in the class level annotation in the component class.

@lazy annotation can be used in the method level annotation on methods annotated with @Bean in the configuration class.

|  |  |
| --- | --- |
| @Component  @Lazy  **public** **class** A {  **public** A() {  System.***out***.println("A");  }  } | @Configuration  **public** **class** AppConfig {  @Bean("A")  @Lazy  **public** A a() {  **return** **new** A();  }  } |

**@Required annotation**

The @Required annotation applies to bean property setter methods. This annotation shows that the affected bean property must be populated at configuration time.

In case the required dependency is not available, the container will throw *BeanInitializationException* exception.

public class ShoppingCart {

private CartService cartService;

@Required

public void setCartService(CartService cartService) {

this.cartService = cartService;

}

// ...

}

### Spring profiles?

Spring profiles allow us to configure Spring Beans based on the profile of dev, QA, staging, production, etc. With Spring profile, we can make the same Spring app which points to different databases or message broker instances with a difference of just a flag.

**Multiple Spring Configuration**

Having multiple Spring configurations is recommended to increase maintainability and modularity.

You can load multiple Java-based configuration files:

|  |  |
| --- | --- |
|  | @Configuration  @Import({MainConfig.class, SchedulerConfig.class})  public class AppConfig {} |

**Spring Bean**

Any normal Java class is initialized by Spring IoC container is called Spring Bean.

We use application context to get the Spring Bean instance.

Spring Beans are initialized by spring container and all the dependencies are also injected.

When context is destroyed, it also destroys all the initialized beans.

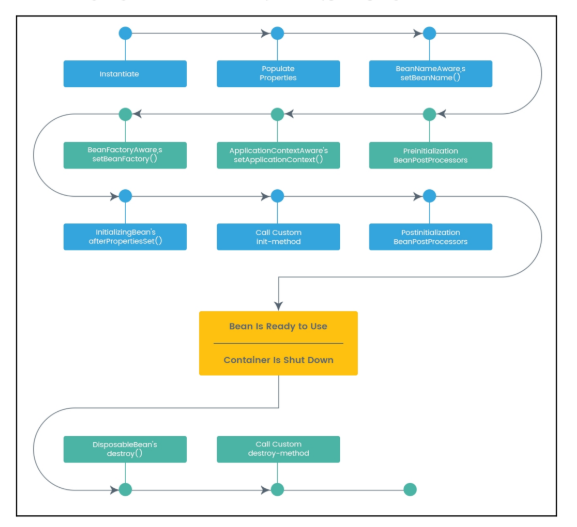
Sometimes we want to initialize other resources or do some validation before making our beans ready to use.

Spring framework provides support for Post-initialization and pre-destroy methods in Spring Beans.

We can do this by two ways

1. By implementing the InitializingBean and DisposableBean interface.
2. Using init-method and destroy-method attribute in Spring bean configurations.

**Spring Bean Life Cycle**



Spring container is responsible for managing the life cycle of the bean, from creation to destruction.

First, Spring bean needs to be instantiated based on Java or XML Bean definitions.

After that, when the bean is no longer required, it will be removed from the Spring container.

A bean is required to be instantiated by spring container before its usage. Similarly, the bean is removed by the spring container when it is no longer in use.

Spring bean factory is responsible for managing the life cycle of beans created through spring container. The life cycle of beans consist of callback methods which can be categorized broadly in two groups:

* Post initialization callback methods
* Pre destruction callback methods

Spring Container

@PostConstruct

1. Set bean name - setBeanName(String name)
2. Set bean class loader aware - setBeanClassLoader(ClassLoader classLoader)
3. Set bean factory aware - setBeanFactory(BeanFactory beanFactory)
4. Set Resource loader aware - setResourceLoader(ResourceLoader resourceLoader)
5. Set application event publisher aware - setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher)
6. Set message source aware - setMessageSource(MessageSource messageSource)
7. Set application context aware - setApplicationContext(ApplicationContext applicationContext)
8. Post process before initialization – BeanPostProcessor - postProcessBeforeInitialization(Object bean, String beanName)
9. After properties set - afterPropertiesSet()
10. Custom init method – init()
11. Post process after initialization - BeanPostProcessor - postProcessAfterInitialization(Object bean, String beanName)
12. Destroy - destroy ()

@PreDestroy

public class SpringBeanLifeCycleExamples implements InitializingBean,DisposableBean,ApplicationContextAware,ApplicationEventPublisherAware,BeanNameAware,BeanFactoryAware,BeanClassLoaderAware,MessageSourceAware,NotificationPublisherAware,LoadTimeWeaverAware,ResourceLoaderAware{ }

|  |  |
| --- | --- |
| InitializingBean | afterPropertiesSet() |
| DisposableBean | Destroy() |
| BeanNameAware | setBeanName(String name) |
| BeanClassLoaderAware | setBeanClassLoader(ClassLoader classLoader) |
| BeanFactoryAware | setBeanFactory(BeanFactory beanFactory) |
| ResourceLoaderAware | setResourceLoader(ResourceLoader resourceLoader) |
| ApplicationEventPublisherAware | setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) |
| MessageSourceAware | setMessageSource(MessageSource messageSource) |
| ApplicationContextAware | setApplicationContext(ApplicationContext applicationContext) |
| BeanPostProcessor | 1. postProcessBeforeInitialization(Object bean, String beanName) 2. postProcessAfterInitialization(Object bean, String beanName) |
| @PostConstruct | public void customInit() {} |
| @PreDestroy | public void customDestroy() {} |

**Spring framework provides four ways for controlling life cycle events of beans:**

* InitializingBean and DisposableBean call back interfaces
* Other Aware interfaces for specific behavior
* Custom init() and destroy() methods in bean configuration file
* @PostConstruct and @PreDestroy annotations

For example, beanInit() and beanDestroy() methods are example of life cycle method:

<beans>

<bean id="demoBean" class="com.roytuts.beans.InitDestroyDemoBean"

init-method="beanInit" destroy-method="beanDestroy"></bean>

</beans>

**Spring Transaction Management**

* Spring framework provides Transaction management supports through Declarative Transaction management as well as Programmatic Transaction Management.
* We use @Transactional annotation for Declarative Transaction management.
* We need to configure transaction manager for the data source in the spring bean configuration file.

<bean id=”transactionManager” class=”org.springframework.jdbc.datasource.DataSourceTransactionManager”>

< property name=”datasource” ref=”datasource”/>

</bean>

**What Design patterns are used in Spring framework?**

1. Singleton pattern in Singleton scoped beans
2. Prototype pattern in prototype scoped beans
3. Model View Controller in Spring MVC
4. Data Access Object in Spring DAO support
5. Front Controller in Spring Dispatcher Servlet
6. Template Method Pattern in Spring JDBC support
7. Adapter Pattern in Spring MVC and Spring Web
8. Proxy Pattern in Spring AOP support
9. Factory Pattern in Bean Factory classes