IoC Containers

Inversion of Control (IoC) is a software design principle that describes inverting the flow of control in a system, so execution flow is not controlled by a central piece of code. This means that components should only depend on abstractions of other components and are not be responsible for handling the creation of dependent objects. Instead, object instances are supplied at runtime by an IoC container through Dependency Injection (DI).

IoC enables better software design that facilitates reuse, loose coupling, and easy testing of software components.

At first IoC might seem complicated, but it’s actually a very simple concept. An IoC container is essentially a registry of abstract types and their concrete implementations. You request an abstract type from the container and it gives you back an instance of the concrete type. It’s a bit like an object factory, but the real benefits come when you use an IoC container in conjunction with dependency injection.

An IoC container is useful on all types of projects, both large and small. It’s true that large, complex applications benefit most from reduced coupling, but I think it’s still a good practice to adopt, even on a small project. Most small applications don’t stay small for long

Spring - Bean Life Cycle

https://www.tutorialspoint.com/spring/spring_bean_life_cycle.htm

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there are lists of the activities that take place behind the scene between the time of bean Instantiation and its destruction, this chapter will discuss only two important bean life cycle callback methods, which are required at the time of bean initialization and its destruction.

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

Initialization callbacks

The org.springframework.beans.factory.InitializingBean interface specifies a single method −

```
void afterPropertiesSet() throws Exception;
```

Thus, you can simply implement the above interface and initialization work can be done inside afterPropertiesSet method as follows −

```
public class ExampleBean implements InitializingBean {
   public void afterPropertiesSet() {
      // do some initialization work
   }
}
```

In the case of XML-based configuration metadata, you can use the **init-method** attribute to specify the name of the method that has a void no-argument signature. For example −

```
<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>
```

Following is the class definition −

```
public class ExampleBean {
   public void init() {
      // do some initialization work
   }
}
```

Destruction callbacks

The *org.springframework.beans.factory.DisposableBean* interface specifies a single method −

```
void destroy() throws Exception;
```

Thus, you can simply implement the above interface and finalization work can be done inside destroy method as follows −

```
public class ExampleBean implements DisposableBean {
   public void destroy() {
      // do some destruction work
   }
}
```

In the case of XML-based configuration metadata, you can use the **destroy-method** attribute to specify the name of the method that has a void no-argument signature. For example −

```
<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>
```

Following is the class definition −

```
public class ExampleBean {
   public void destroy() {
      // do some destruction work
   }
}
```

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment, you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

Steps	Description
1	Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.
3	Create Java classes *HelloWorld* and *MainApp* under the *com.tutorialspoint* package.
4	Create Beans configuration file *Beans.xml* under the **src** folder.
5	The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Here is the content of **HelloWorld.java** file −

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
   public void init(){
      System.out.println("Bean is going through init.");
   }
   public void destroy() {
      System.out.println("Bean will destroy now.");
   }
}
```

Following is the content of the **MainApp.java** file. Here you need to register a shutdown hook **registerShutdownHook** method that is declared on the AbstractApplicationContext class. This will ensure a graceful shutdown and call the relevant destroy methods.

```
package com.tutorialspoint;

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

public class MainApp {
   public static void main(String[] args) {
      AbstractApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

Following is the configuration file **Beans.xml** required for init and destroy methods −

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

<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" init-method = "init"
      destroy-method = "destroy">
      <property name = "message" value = "Hello World!"/>
   </bean>

</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

```
Bean is going through init.
Your Message : Hello World!
Bean will destroy now.
```

Default initialization and destroy methods

If you have too many beans having initialization and/or destroy methods with the same name, you don't need to declare **init-method** and **destroy-method** on each individual bean. Instead, the framework provides the flexibility to configure such situation using **default-init-method** and **default-destroy-method** attributes on the <beans> element as follows −

```
<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"
   default-init-method = "init"
   default-destroy-method = "destroy">

   <bean id = "..." class = "...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

</beans>
```