**Spring IoC Container**

Spring IoC is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, objects dependencies are injected by other assembler objects. Spring IoC container is the program that **injects** dependencies into an object and make it ready for our use.

Spring IoC container classes are part of org.springframework.beans and org.springframework.context packages. Spring IoC container provides us different ways to decouple the object dependencies.

BeanFactory is the root interface of Spring IoC container. ApplicationContext is the child interface of BeanFactory interface that provide Spring AOP features, i18n etc.

Some of the useful ApplicationContext implementations that we use are;

* **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
* **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
* **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
* **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

### Spring Bean

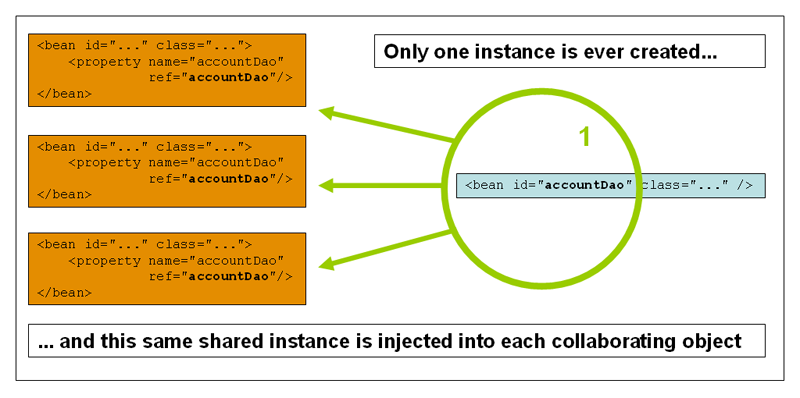
There are five scopes defined for Spring Beans.

1. [**singleton**](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) – Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
2. **prototype** – A new instance will be created every time the bean is requested.
3. **request** – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
4. **session** – A new bean will be created for each HTTP session by the container.
5. **global-session** – This is used to create global session beans for Portlet applications.

### The singleton scope

Only one shared instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.



Spring’s concept of a singleton bean differs from the Singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one *and only one* instance of a particular class is created *per ClassLoader*. The scope of the Spring singleton is best described as *per container and per bean*. This means that if you define one bean for a particular class in a single Spring container, then the Spring container creates one *and only one* instance of the class defined by that bean definition. *The singleton scope is the default scope in Spring*. To define a bean as a singleton in XML, you would write, for example:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

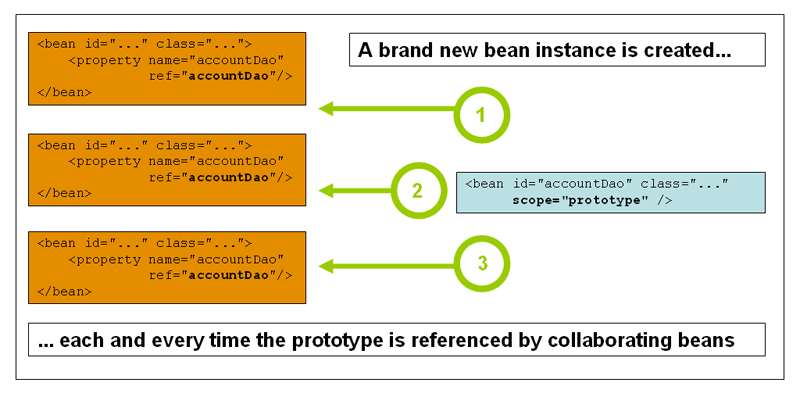
*<!-- the following is equivalent, though redundant (singleton scope is the default) -->*

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

### The prototype scope

### The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a getBean() method call on the container. As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

### The following diagram illustrates the Spring prototype scope. A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram*.*



The following example defines a bean as a prototype in XML:

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

**NOTE:**

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding. To get the Spring container to release resources held by prototype-scoped beans, try using a custom [bean post-processor](https://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#beans-factory-extension-bpp), which holds a reference to beans that need to be cleaned up.

### 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. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see [Section 7.4.6, “Method injection”](https://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#beans-factory-method-injection)

### Request, session, global session, application, and WebSocket scopes The request, session, globalSession, application, and websocket scopes are *only* available if you use a web-aware Spring ApplicationContextimplementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, an IllegalStateException will be thrown complaining about an unknown bean scope.

#### **Initial web configuration**

To support the scoping of beans at the request, session, globalSession, application, and websocket levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is *not* required for the standard scopes, singleton and prototype.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring DispatcherServlet or DispatcherPortlet, then no special setup is necessary: DispatcherServlet and DispatcherPortlet already expose all relevant state.

If you use a Servlet 2.5 web container, with requests processed outside of Spring’s DispatcherServlet (for example, when using JSF or Struts), you need to register the org.springframework.web.context.request.RequestContextListener ServletRequestListener. For Servlet 3.0+, this can be done programmatically via the WebApplicationInitializer interface. Alternatively, or for older containers, add the following declaration to your web application’s web.xml file:

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

#### **Request scope**

Consider the following XML configuration for a bean definition:

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

The Spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, theloginAction bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same loginAction bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java Config, the @RequestScope annotation can be used to assign a component to the request scope.

**@RequestScope**

*@Component*

**public** **class** LoginAction {

*// ...*

}

#### **Session scope**

Consider the following XML configuration for a bean definition:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

The Spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferencesbean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded

When using annotation-driven components or Java Config, the @SessionScope annotation can be used to assign a component to the session scope.

**@SessionScope**

*@Component*

**public** **class** UserPreferences {

*// ...*

}

#### **Global session scope**

Consider the following bean definition:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>

The globalSession scope is similar to the standard HTTP Session scope ([described above](https://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/htmlsingle/#beans-factory-scopes-session)), and applies only in the context of portlet-based web applications. The portlet specification defines the notion of a global Session that is shared among all portlets that make up a single portlet web application. Beans defined at the globalSession scope are scoped (or bound) to the lifetime of the global portlet Session.

If you write a standard Servlet-based web application and you define one or more beans as having globalSession scope, the standard HTTP Session scope is used, and no error is raised.

#### **Application scope**

Consider the following XML configuration for a bean definition:

<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>

The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application. That is, the appPreferences bean is scoped at the ServletContext level, stored as a regular ServletContext attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a ServletContext attribute.

When using annotation-driven components or Java Config, the @ApplicationScope annotation can be used to assign a component to the application scope

**@ApplicationScope**

*@Component*

**public** **class** AppPreferences {

*// ...*

}

### Spring Bean Configuration

Spring Framework provides three ways to configure beans to be used in the application.

1. **Annotation Based Configuration** – By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
2. **XML Based Configuration** – By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
3. **Java Based Configuration** – Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.