7.5 Bean scopes

When you create a bean definition, you create a *recipe* for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the *scope* of the objects created from a particular bean definition. This approach is powerful and flexible in that you can *choose* the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports seven scopes, five of which are available only if you use a web-aware `ApplicationContext`.

The following scopes are supported out of the box. You can also create a custom scope.

Table 7.3. Bean scopes

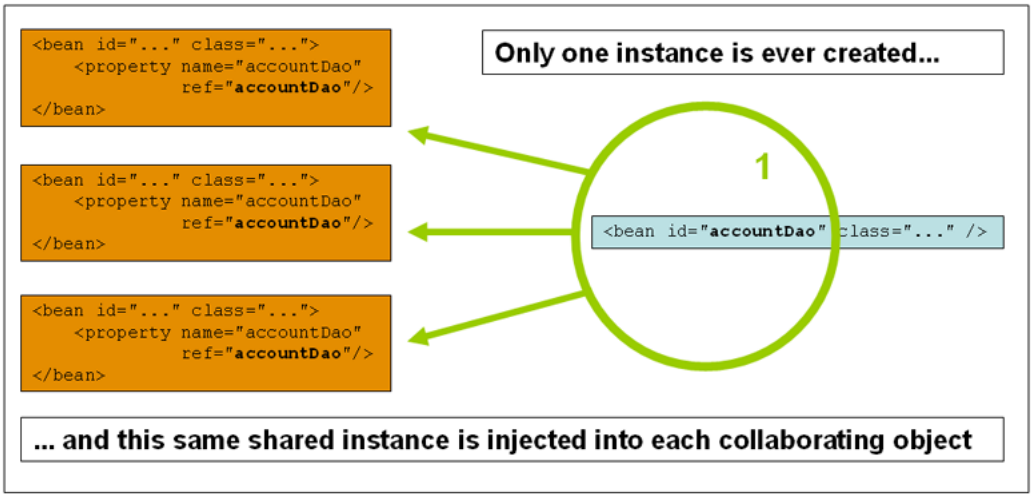
Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
globalSession	Scopes a single bean definition to the lifecycle of a global HTTP `Session`. Typically only valid when used in a Portlet context. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

> **Note**
>
> As of Spring 3.0, a *thread scope* is available, but is not registered by default. For more information, see the documentation for `SimpleThreadScope`. For instructions on how to register this or any other custom scope, see the section called “Using a custom scope”.

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:

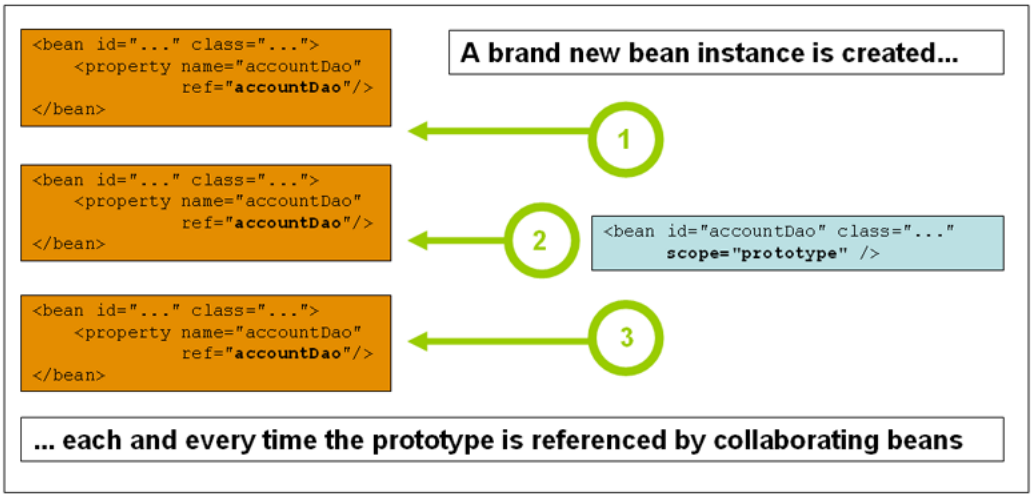
```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- 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"/>
```

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, which holds a reference to beans that need to be cleaned up.

In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java new operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see the section called "Lifecycle callbacks".)

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 the section called "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 `ApplicationContext` implementation (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>
```

Alternatively, if there are issues with your listener setup, consider using Spring's `RequestContextFilter`. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate.

```
<web-app>
    ...
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

`DispatcherServlet`, `RequestContextListener`, and `RequestContextFilter` all do exactly the same thing, namely bind the HTTP request object to the `Thread` that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

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, the `loginAction` 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 `userPreferences` bean 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), 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 {
    // ...
}
```

Scoped beans as dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request scoped bean into another bean of a longer-lived scope, you may choose to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real target object from the relevant scope (such as an HTTP request) and delegate method calls onto the real object.

> **Note**
>
> You may also use `<aop:scoped-proxy/>` between beans that are scoped as `singleton`, with the reference then going through an intermediate proxy that is serializable and therefore able to re-obtain the target singleton bean on deserialization.
>
> When declaring `<aop:scoped-proxy/>` against a bean of scope `prototype`, every method call on the shared proxy will lead to the creation of a new target instance which the call is then being forwarded to.
>
> Also, scoped proxies are not the only way to access beans from shorter scopes in a lifecycle-safe fashion. You may also simply declare your injection point (i.e. the constructor/setter argument or autowired field) as `ObjectFactory<MyTargetBean>`, allowing for a `getObject()` call to retrieve the current instance on demand every time it is needed - without holding on to the instance or storing it separately.
>
> The JSR-330 variant of this is called `Provider`, used with a `Provider<MyTargetBean>` declaration and a corresponding `get()` call for every retrieval attempt. See here for more details on JSR-330 overall.

The configuration in the following example is only one line, but it is important to understand the "why" as well as the "how" behind it.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/>
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.foo.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

To create such a proxy, you insert a child `<aop:scoped-proxy/>` element into a scoped bean definition. See the section called “Choosing the type of proxy to create” and Chapter 41, *XML Schema-based configuration*.) Why do definitions of beans scoped at the `request`, `session`, `globalSession` and custom-scope levels require the `<aop:scoped-proxy/>` element ? Let’s examine the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes. (The following `userPreferences` bean definition as it stands is *incomplete.)*

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

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

In the preceding example, the singleton bean `userManager` is injected with a reference to the HTTP `Session`-scoped bean `userPreferences`. The salient point here is that the `userManager` bean is a singleton: it will be instantiated *exactly once* per container, and its dependencies (in this case only one, the `userPreferences` bean) are also injected only once. This means that the `userManager` bean will only operate on the exact same `userPreferences` object, that is, the one that it was originally injected with.

This is *not* the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean, for example injecting an HTTP `Session`-scoped collaborating bean as a dependency into singleton bean. Rather, you need a single `userManager` object, and for the lifetime of an HTTP `Session`, you need a `userPreferences` object that is specific to said HTTP `Session`. Thus the container creates an object that exposes the exact same public interface as the `UserPreferences` class (ideally an object that *is a* `UserPreferences` instance) which can fetch the real `UserPreferences` object from the scoping mechanism (HTTP request, `Session`, etc.). The container injects this proxy object into the `userManager` bean, which is unaware that this `UserPreferences` reference is a proxy. In this example, when a `UserManager` instance invokes a method on the dependency-injected `UserPreferences` object, it actually is invoking a method on the proxy. The proxy then fetches the real `UserPreferences` object from (in this case) the HTTP `Session`, and delegates the method invocation onto the retrieved real `UserPreferences` object.

Thus you need the following, correct and complete, configuration when injecting `request-`, `session-`, and `globalSession-scoped` beans into collaborating objects:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

Choosing the type of proxy to create

By default, when the Spring container creates a proxy for a bean that is marked up with the `<aop:scoped-proxy/>` element, *a CGLIB-based class proxy is created*.

> **Note**
>
> CGLIB proxies only intercept public method calls! Do not call non-public methods on such a proxy; they will not be delegated to the actual scoped target object.

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying `false` for the value of the `proxy-target-class` attribute of the `<aop:scoped-proxy/>` element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to effect such proxying. However, it also means that the class of the scoped bean must implement at least one interface, and *that all* collaborators into which the scoped bean is injected must reference the bean through one of its interfaces.

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

For more detailed information about choosing class-based or interface-based proxying, see Section 11.6, “Proxying mechanisms”.

Custom scopes

The bean scoping mechanism is extensible; You can define your own scopes, or even redefine existing scopes, although the latter is considered bad practice and you *cannot* override the built-in `singleton` and `prototype` scopes.

Creating a custom scope

To integrate your custom scope(s) into the Spring container, you need to implement the `org.springframework.beans.factory.config.Scope` interface, which is described in this section. For an idea of how to implement your own scopes, see the `Scope` implementations that are supplied with the Spring Framework itself and the Scope javadocs, which explains the methods you need to implement in more detail.

The `Scope` interface has four methods to get objects from the scope, remove them from the scope, and allow them to be destroyed.

The following method returns the object from the underlying scope. The session scope implementation, for example, returns the session-scoped bean (and if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference).

```
Object get(String name, ObjectFactory objectFactory)
```

The following method removes the object from the underlying scope. The session scope implementation for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return null if the object with the specified name is not found.

```
Object remove(String name)
```

The following method registers the callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed. Refer to the javadocs or a Spring scope implementation for more information on destruction callbacks.

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

The following method obtains the conversation identifier for the underlying scope. This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

```
String getConversationId()
```

Using a custom scope

After you write and test one or more custom `Scope` implementations, you need to make the Spring container aware of your new scope(s). The following method is the central method to register a new `Scope` with the Spring container:

```
void registerScope(String scopeName, Scope scope);
```

This method is declared on the `ConfigurableBeanFactory` interface, which is available on most of the concrete `ApplicationContext` implementations that ship with Spring via the BeanFactory property.

The first argument to the `registerScope(..)` method is the unique name associated with a scope; examples of such names in the Spring container itself are `singleton` and `prototype`. The second argument to the `registerScope(..)` method is an actual instance of the custom `Scope` implementation that you wish to register and use.

Suppose that you write your custom `Scope` implementation, and then register it as below.

> **Note**
>
> The example below uses `SimpleThreadScope` which is included with Spring, but not registered by default. The instructions would be the same for your own custom `Scope` implementations.

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

You then create bean definitions that adhere to the scoping rules of your custom `Scope`:

```
<bean id="..." class="..." scope="thread">
```

With a custom `Scope` implementation, you are not limited to programmatic registration of the scope. You can also do the `Scope` registration declaratively, using the `CustomScopeConfigurer` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="org.springframework.context.support.SimpleThreadScope"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>

</beans>
```

> **Note**
>
> When you place `<aop:scoped-proxy/>` in a `FactoryBean` implementation, it is the factory bean itself that is scoped, not the object returned from `getObject()`.

7.6 Customizing the nature of a bean

Lifecycle callbacks

To interact with the container's management of the bean lifecycle, you can implement the Spring `InitializingBean` and `DisposableBean` interfaces. The container calls `afterPropertiesSet()` for the former and `destroy()` for the latter to allow the bean to perform certain actions upon initialization and destruction of your beans.

> **Tip**
>
> The JSR-250 `@PostConstruct` and `@PreDestroy` annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring specific interfaces. For details see the section called "@PostConstruct and @PreDestroy".
>
> If you don't want to use the JSR-250 annotations but you are still looking to remove coupling consider the use of init-method and destroy-method object definition metadata.

Internally, the Spring Framework uses `BeanPostProcessor` implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle