## 3.3.2.1.1. The idref element

|  |
| --- |
| <bean id="theTargetBean" class="..."/>  <bean id="theClientBean" class="...">  <property name="targetName">  <idref bean="theTargetBean" />  </property>  </bean> |

The main reason the first form is preferable to the second is that using the idref tag allows the container to validate *at deployment time* that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the 'targetName' property of the 'client' bean. Any typo will only be discovered (with most likely fatal results) when the 'client' bean is actually instantiated. If the 'client' bean is a [prototype](file:///E:\DevTools\Spring\spring-framework-2.5.6.SEC03\docs\reference\html_single\index.html#beans-factory-scopes) bean, this typo (and the resulting exception) may only be discovered long after the container is actually deployed.

Additionally, if the bean being referred to is in the same XML unit, and the bean name is the bean *id*, the 'local' attribute may be used, which allows the XML parser itself to validate the bean id even earlier, at XML document parse time.

<property name="targetName">

*<!-- a bean with an id of 'theTargetBean' must exist; otherwise an XML exception will be thrown -->*

<idref local="theTargetBean"/>

</property>

By way of an example, one common place (at least in pre-Spring 2.0 configuration) where the <idref/> element brings value is in the configuration of [AOP interceptors](file:///E:\DevTools\Spring\spring-framework-2.5.6.SEC03\docs\reference\html_single\index.html#aop-pfb-1) in a ProxyFactoryBean bean definition. If you use <idref/> elements when specifying the interceptor names, there is no chance of inadvertently misspelling an interceptor id.

## 3.3.2.3. Inner beans

A <bean/> element inside the <property/> or <constructor-arg/> elements is used to define a so-called *inner bean*. An inner bean definition does not need to have any id or name defined, and it is best not to even specify any id or name value because the id or name value simply will be ignored by the container.

## 3.3.2.4. Collections

*Note that the value of a map key or value, or a set value, can also again be any of the following elements:*

bean | ref | idref | list | set | map | props | value | null

## 3.3.2.4.1. Collection merging

<beans>

<bean id="parent" abstract="true" class="example.ComplexObject">

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.com</prop>

<prop key="support">support@example.com</prop>

</props>

</property>

</bean>

<bean id="child" parent="parent">

<property name="adminEmails">

*<!-- the merge is specified on the \*child\* collection definition -->*

<props merge="true">

<prop key="sales">sales@example.com</prop>

<prop key="support">support@example.co.uk</prop>

</props>

</property>

</bean>

<beans>

Notice the use of the merge=true attribute on the <props/> element of the adminEmails property of the child bean definition. When the child bean is actually resolved and instantiated by the container, the resulting instance will have an adminEmails Properties collection that contains the result of the merging of the child's adminEmails collection with the parent's adminEmails collection

## 3.3.2.5. Nulls

The <null/> element is used to handle null values. Spring treats empty arguments for properties and the like as empty Strings. The following XML-based configuration metadata snippet results in the email property being set to the empty String value ("")

<bean class="ExampleBean">

<property name="email"><value/></property>

</bean>

This is equivalent to the following Java code: exampleBean.setEmail(""). The special <null> element may be used to indicate a null value. For example:

<bean class="ExampleBean">

<property name="email"><null/></property>

</bean>

The above configuration is equivalent to the following Java code: exampleBean.setEmail(null).

## 3.3.2.7. Compound property names

Compound or nested property names are perfectly legal when setting bean properties, as long as all components of the path except the final property name are not null. Consider the following bean definition...

<bean id="foo" class="foo.Bar">

<property name="fred.bob.sammy" value="123" />

</bean>

The foo bean has a fred property which has a bob property, which has a sammy property, and that final sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the bob property of fred must not be null be non-null after the bean is constructed, or a NullPointerException will be thrown.

## 3.3.3. Using depends-on

 The 'depends-on' attribute may be used to explicitly force one or more beans to be initialized before the bean using this element is initialized. Find below an example of using the 'depends-on' attribute to express a dependency on a single bean.

<bean id="beanOne" class="ExampleBean" depends-on="**manager**"/>

<bean id="**manager**" class="ManagerBean" />

If you need to express a dependency on multiple beans, you can supply a list of bean names as the value of the 'depends-on' attribute, with commas, whitespace and semicolons all valid delimiters, like so:

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

<bean id="manager" class="ManagerBean" />

<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />

The 'depends-on' attribute at the bean definition level is used not only to specify an initialization time dependency, but also to specify the corresponding destroy time dependency (in the case of [singleton](file:///E:\DevTools\Spring\spring-framework-2.5.6.SEC03\docs\reference\html_single\index.html#beans-factory-scopes-singleton) beans only). Dependent beans that define a 'depends-on' relationship with a given bean will be destroyed first - prior to the given bean itself being destroyed. As a consequence, 'depends-on' may be used to control shutdown order too

## 3.3.4. Lazily-instantiated beans

The default behavior for ApplicationContext implementations is to eagerly pre-instantiate all singleton beans at startup. Pre-instantiation means that an ApplicationContext will eagerly create and configure all of its [singleton](file:///E:\DevTools\Spring\spring-framework-2.5.6.SEC03\docs\reference\html_single\index.html#beans-factory-scopes-singleton) beans as part of its initialization process. Generally this is *a good thing*, because it means that any errors in the configuration or in the surrounding environment will be discovered immediately (as opposed to possibly hours or even days down the line).

One thing to understand about lazy-initialization is that even though a bean definition may be marked up as being lazy-initialized, if the lazy-initialized bean is the dependency of a singleton bean that is not lazy-initialized, when the ApplicationContext is eagerly pre-instantiating the singleton, it will have to satisfy all of the singletons dependencies, one of which will be the lazy-initialized bean! So don't be confused if the IoC container creates one of the beans that you have explicitly configured as lazy-initialized at startup; all that means is that the lazy-initialized bean is being injected into a non-lazy-initialized singleton bean elsewhere.

## 3.3.7. Method Injection

 There is a problem when the bean lifecycles are different. Consider a singleton bean A which needs to use a non-singleton (prototype) bean B, perhaps on each method invocation on A. The container will only create the singleton bean A once, and thus only get the opportunity to set the properties once. There is no opportunity for the container to provide bean A with a new instance of bean B every time one is needed.

One solution to this issue is to forego some inversion of control. Bean A can be [made aware of the container](file:///E:\DevTools\Spring\spring-framework-2.5.6.SEC03\docs\reference\html_single\index.html#beans-factory-aware-beanfactoryaware) by implementing the BeanFactoryAware interface, and [use programmatic means](file:///E:\DevTools\Spring\spring-framework-2.5.6.SEC03\docs\reference\html_single\index.html#beans-factory-client) to ask the container via a getBean("B") call for (a typically new) bean B instance every time it needs it. Find below an admittedly somewhat contrived example of this approach:

*// a class that uses a stateful Command-style class to perform some processing*

package fiona.apple;

*// lots of Spring-API imports*

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.BeanFactoryAware;

public class CommandManager implements BeanFactoryAware {

private BeanFactory beanFactory;

public Object process(Map commandState) {

*// grab a new instance of the appropriate Command*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

return command.execute();

}

*// the Command returned here could be an implementation that executes asynchronously, or whatever*

protected Command createCommand() {

return (Command) this.beanFactory.getBean("command"); *// notice the Spring API dependency*

}

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

this.beanFactory = beanFactory;

}

}