Dependencies

Summary

Description

Injecting dependencies

The basic principle of Dependency Injection (DI) is that the instance defines the dependency (the instance that requires) through the constructor or set method. Then, it is the job of the container to actually *inject* those dependencies when it creates the bean. This is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself being in control of instantiating or locating its dependencies on its own using direct construction of classes, or something like the *Service Locator* pattern.

DI exists in two major variants, namely Constructor Injection and Setter Injection.

Constructor Injection

The constructor based DI calls the constructor that has many arguments and injects dependency. The <constructor-arg> element is used.

```
package x.y;

public class Foo {

    public Foo(Bar bar, Baz baz) {
        // ...
    }
}
<beans>
    <bean name="foo" class="x.y.Foo">
        <constructor-arg>
            <bean class="x.y.Bar"/>
        </constructor-arg>
        <constructor-arg>
            <bean class="x.y.Baz"/>
        </constructor-arg>
    </bean>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true<value>, Spring cannot determine the type of the value, and so cannot match by type without help.

```
package examples;

public class ExampleBean {

    // No. of years to the calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Constructor Argument Type Matching

In the case above, the type of each argument can be designated through the 'type' attribute.

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

Constructor Argument Index

In the case above, the location of each argument can be designated through the 'index' attribute.

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

(*Note that the *index is 0 based*.)

Setter Injection

The setter based DI creates the bean instance through the constructor with no argument and calls the setter method to inject dependency. The <property> element is used.

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

Detailed Dependency Configuration

This chapter explains the <constructor-arg> and the sub-element type of the <property> element used in DI.

Straight Values (primitives, Strings, etc.)

The form of recognizable string is expressed using the <value> tag. The string is converted according to the type of argument and property.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">

  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
  <property name="password">
    <value>masterkaoli</value>
  </property>
</bean>
```

The 'value' attribute can be used instead of <value> element.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">

  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
  <property name="username" value="root"/>
  <property name="password" value="masterkaoli"/>
</bean>
```

References to other beans (collaborators)

The ref element refers to other beans in the container. There are three variations to determine referenced object.

1. bean attribute
   Specifying the target bean by using the bean attribute of the <ref/> tag is the most general form, and will allow creating a reference to any bean in the same container (whether or not in the same XML file), or parent container. The value of the 'bean' attribute may be the same as either the 'id' attribute of the target bean, or one of the values in the 'name' attribute of the target bean.

   ```
   <ref bean="someBean"/>
   ```

2. local attribute
   Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the local attribute must be the same as the id attribute of the target bean.

   ```
   <ref local="someBean"/>
   ```

3. parent attribute
   Specifying the target bean by using the 'parent' attribute allows a reference to be created to a bean which is in a parent container of the current container. The value of the 'parent' attribute may be the same as either the 'id' attribute of the target bean, or one of the values in the

'name' attribute of the target bean, and the target bean must be in a parent container to the current one.

```
4.    <!-- in the parent context -->
5.    <bean id="accountService" class="com.foo.SimpleAccountService">
6.          <!-- insert dependencies as required as here -->
      </bean>
      <!-- in the child (descendant) context -->
      <bean id="accountService"   <-- notice that the name of this bean is the same as the name of
      the 'parent' bean
            class="org.springframework.aop.framework.ProxyFactoryBean">
            <property name="target">
                <ref parent="accountService"/>   <-- notice how we refer to the parent bean
            </property>
          <!-- insert other configuration and dependencies as required as here -->
      </bean>
```

Inner Beans

The <bean/> element in the <property/> or <constructor-arg/> element is 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.

```
<bean id="outer" class="...">
  <!-- instead of using a reference to a target bean, simply define the target bean inline -->
  <property name="target">
    <bean class="com.example.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

Note that in the specific case of inner beans, the 'scope' flag and any 'id' or 'name' attribute are effectively ignored. Inner beans are *always* anonymous and they are *always* scoped as prototypes. Please also note that it is *not* possible to inject inner beans into collaborating beans other than the enclosing bean.

Collections

To express the java collection type, List, Set, Map, Properties, the <list/>, <set/>, <map/>, <props/> elements are used.

```
<bean id="moreComplexInstance" class="example.ComplexInstance">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@example.org</prop>
        <prop key="support">support@example.org</prop>
        <prop key="development">development@example.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
    </list>
  </property>
  <!-- results in a setSomeMap(java.util.Map) call -->
  <property name="someMap">
    <map>
```

```
            <entry>
                <key>
                    <value>an entry</value>
                </key>
                <value>just some string</value>
            </entry>
            <entry>
                <key>
                    <value>a ref</value>
                </key>
                <ref bean="myDataSource" />
            </entry>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

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

Collection Merging

Container provides the collection merge function. This allows an application developer to define a parent-style <list/>, <map/>, <set/> or <props/> element, and have child-style <list/>, <map/>, <set/> or <props/> elements inherit and override values from the parent collection.

```
<beans>
<bean id="parent" abstract="true" class="example.ComplexInstance">
    <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>
```

The child’s adminEmails created by above configuration will have values below.

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

Nulls

To use the null value, <null/> element is used. When there is no argument, the Spring is recognized as an empty string (“”).

```
<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>
```

Shortcuts and other convenience options for XML-based configuration metadata

XML-based configuration metadata shortcuts

All <property/>, <constructor-arg/>, <entry/> elements can use ‘value’ attribute instead of <value/> element.

```
<property name="myProperty">
  <value>hello</value>
</property>
<constructor-arg>
  <value>hello</value>
</constructor-arg>
<entry key="myKey">
  <value>hello</value>
</entry>
```

are equivalent to:

```
<property name="myProperty" value="hello"/>
<constructor-arg value="hello"/>
<entry key="myKey" value="hello"/>
```

<property/>, <constructor-arg/> element can use the 'ref' attribute instead of the <ref/> element.

```
<property name="myProperty">
  <ref bean="myBean">
</property>
<constructor-arg>
  <ref bean="myBean">
</constructor-arg>
```

are equivalent to:

```
<property name="myProperty" ref="myBean"/>
<constructor-arg ref="myBean"/>
```

Note however that the shortcut form is equivalent to a <ref bean="xxx"> element; there is no shortcut for <ref local="xxx">.

The entry element allows a shortcut form to specify the key and/or value of the map, in the form of the 'key' / 'key-ref' and 'value' / 'value-ref' attributes.

```
<entry>
  <key>
    <ref bean="myKeyBean" />
  </key>
  <ref bean="myValueBean" />
</entry>
```

is equivalent to:

```
<entry key-ref="myKeyBean" value-ref="myValueBean"/>
```

The p-namespace and how to use it to configure properties

Instead of using nested <property/> elements, using the p-namespace you can use attributes as part of the bean element that describe your property values. The following classic bean and p-namespace bean use same bean configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="foo@bar.com/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
          p:email="foo@bar.com"/>
</beans>
```

This next example includes two more bean definitions that both have a reference to another bean. the '-ref' part indicates that this is not a straight value but rather a reference to another bean.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

Compound property names

Compound or nested property names are perfectly legal when setting bean properties.

```
<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.

Using depends-on

For most situations, the fact that a bean is a dependency of another is expressed by the fact that one bean is set as a property of another. For the relatively infrequent situations, 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 (for example, when a static initializer in a class needs to be triggered, such as database driver registration). 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" />
```

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 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

However, there are times when this behavior is *not* what is wanted. If you do not want a singleton bean to be pre-instantiated when using an ApplicationContext, you can selectively control this by marking a bean definition as lazy-initialized. A lazily-initialized bean indicates to the IoC container whether or not a bean instance should be created at startup or when it is first requested.

When configuring beans via XML, this lazy loading is controlled by the 'lazy-init' attribute on the <bean/> element.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

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.

It is also possible to control lazy-initialization at the container level by using the 'default-lazy-init' attribute on the <beans/> element.

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

Autowiring collaborators

The Spring container is able to *autowire* relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory. The autowiring functionality has five modes. When using XML-based configuration metadata, the autowire mode for a bean definition is specified by using the autowire attribute of the <bean/> element.

Mode	Explanation
no	No autowiring at all. Bean references must be defined via a ref element. This is the default.
byName	Autowiring by property name. This option will inspect the container and look for a bean named exactly the same as the property which needs to be autowired.
byType	Allows a property to be autowired if there is exactly one bean of the property type in the container. If there is more than one, a fatal exception is thrown, and this indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	This is analogous to *byType*, but applies to constructor arguments. If there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised
autodetect	Chooses *constructor* or *byType* through introspection of the bean class. If a default constructor is found, the *byType* mode will be applied.

Note that explicit dependencies in property and constructor-arg settings always override autowiring.

Excluding a bean from being available from autowiring

When configuring beans using Spring's XML format, the 'autowire-candidate' attribute of the <bean/> element can be set to 'false'; this has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

Checking for dependencies

The Spring IoC container also has the ability to check for the existence of unresolved dependencies of a bean deployed into the container. This feature is sometimes useful when you want to ensure that all properties (or all properties of a certain type) are set on a bean. Dependency checking can also be enabled and disabled per bean, just as with the autowiring functionality. Dependency checking can be handled in four different modes. When using XML-based configuration metadata, this is specified via the 'dependency-check' attribute in a bean definition, which may have the following values.

Mode	Explanation
none	No dependency checking. A default mode.
simple	Dependency checking is performed for primitive types and collections.
instance	Dependency checking is performed for collaborators only.
all	Dependency checking is done for collaborators, primitive types and collections.

Method Injection

For most application scenarios, the majority of the beans in the container will be singletons. When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, the typical and common approach of handling this dependency by defining one bean to be a property of the other is quite adequate. 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 by implementing the BeanFactoryAware interface, and use programmatic means to ask the container via a getBean("B") call for (a typically new) bean B instance every time it needs it.

```
// 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 Instance 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;
    }
}
```

The above example is generally not a desirable solution since the business code is then aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

Lookup method injection

Lookup method injection refers to the ability of the container to override methods on *container managed beans*, to return the result of looking up another named bean in the container. Spring Framework implements this method injection by dynamically generating a subclass overriding the method, using bytecode generation via the CGLIB library.

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Instance process(Instance commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

     // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

The method that is to be 'injected' must have a signature of the following form:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

If the method is abstract, the dynamically-generated subclass will implement the method. Otherwise, the dynamically-generated subclass will override the concrete method defined in the original class.

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
  <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
  <lookup-method name="createCommand" bean="command"/>
</bean>
```

The bean identified as *commandManager* will call its own method createCommand() whenever it needs a new instance of the *command* bean. If it is deployed as a singleton, the same instance of the command bean will be returned each time.

Please be aware that in order for this dynamic subclassing to work, you will need to have the CGLIB jar(s) on your classpath. Additionally, the class that the Spring container is going to subclass cannot be final, and the method that is being overridden cannot be final either.