The Spring Framework

What is spring framework?

The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications.

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers.

True, you can use design patterns such as Factory, Abstract Factory, Builder, Decorator, and Service Locator to compose the various classes and object instances that make up an application.

However, **these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth.**

Patterns are formalized best practices that you must implement yourself in your application.

The Spring Framework Inversion of Control (IoC) component addresses this concern by providing a formalized means of composing disparate components into a fully working application ready for use.

The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the various modules of the spring framework?



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The Core Container consists of the Core, Beans, Context, and Expression Language

* **BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.**
* The Context module builds on the solid base provided by the Core and Beans modules: **it is a means to access objects in a framework-style manner that is similar to a JNDI registry.**
* The ApplicationContext interface is the focal point of the Context module.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is Inversion of control or dependency injection?

IoC is also known as *dependency injection* (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object.

ApplicationContext is a subinterface of BeanFactory.

In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is a bean?

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans.

A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the spring ApplicationContext?

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata (applicationcontext.xml).

The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The following example shows the basic structure of XML-based configuration metadata:

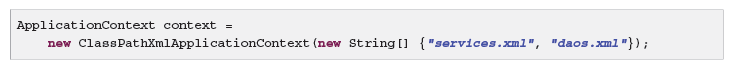


* **id attribute** is a string that you use to identify the individual bean definition.
* **class attribute** defines the **type of the bean** and uses the **fully qualified classname**. The value of the id attribute refers to collaborating objects.

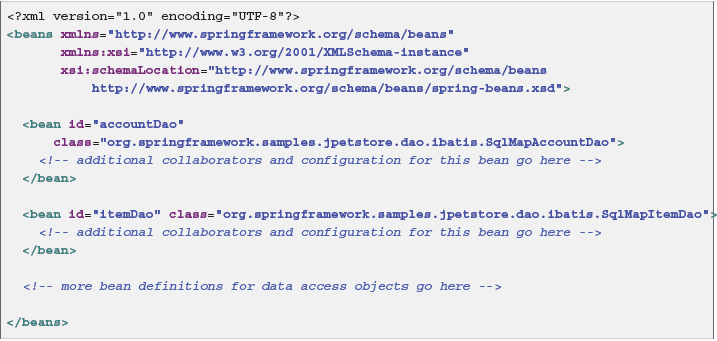
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we instantiate the spring container?

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an ApplicationContext constructor are actually **resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH**, and so on.







The **property name** element refers to the name of the JavaBean property, and the **ref** element refers to the name of another bean definition.

**This linkage between id and ref elements expresses the dependency between collaborating objects.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Can we separate bean definitions in to separate xml files?

Yes we can and this is done as often individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section.

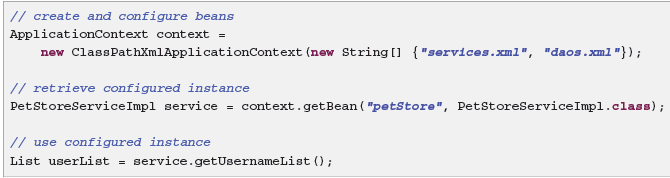
Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file or files.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we get access to the beans present in the application context?

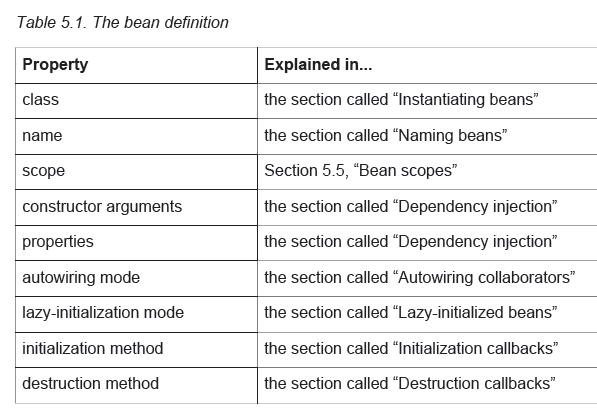
The ApplicationContext enables you to read bean definitions and access them as follows:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the information contained in the bean definition?

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the rules for naming the beans?

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, **you use the id and/or name attributes to specify the bean identifier(s)**. The id attribute allows you to specify exactly one id.

Conventionally these names are alphanumeric ('myBean', 'fooService', etc), but may have special characters as well. If you want to introduce other aliases to the bean, you can also specify them in the name attribute, separated by a comma (,), semicolon (;), or white space.

You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the ref element or Service Locator style lookup, you must provide a name.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Why do we use Aliases?

It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem i.e. each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the <alias/> element to accomplish this.



Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we instantiate using a static factory method?

When defining a bean that you create with a static factory method, you use the **class** attribute to specify the class containing the static factory method and an attribute named **factory-method** to specify the name of the factory method itself.

You should be able to call this method (with optional arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call static factories in legacy code.

The following bean definition specifies that the bean will be created by calling a factory-method. The definition does not specify the type (class) of the returned object, only the class containing the factory method.

In this example, the createInstance() method must be a static method.

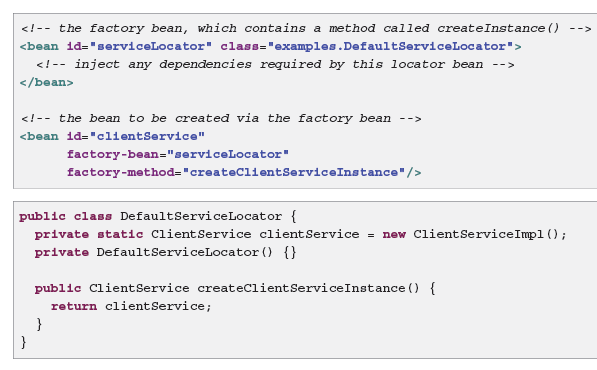


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How to create an instance using an instance factory method?

Similar to instantiation through a static factory method, instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean.

To use this mechanism, leave the class attribute empty, and in the factory-bean attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the factory-method attribute.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are dependencies and what is dependency injection?

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application and these are known as dependencies.

*Dependency injection* (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the *Service Locator* pattern.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the advantages of dependency injection?

Advantages of using dependency Injection:

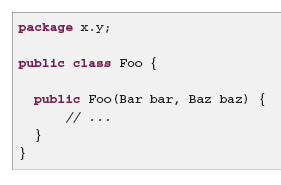
* Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies.
* The object does not look up its dependencies, and does not know the location or class of the dependencies.
* As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

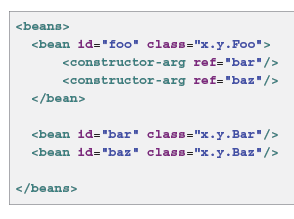
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we use constructor based dependency injection?

* Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
* Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly.

Consider the following class:



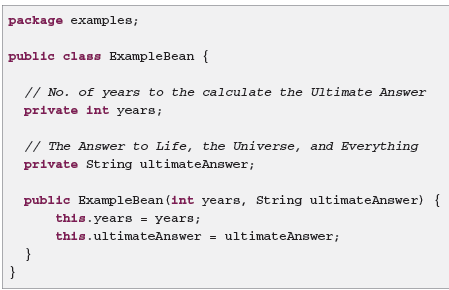


You do not need to specify the constructor argument indexes and/or types explicitly in the <constructor-arg/> element.

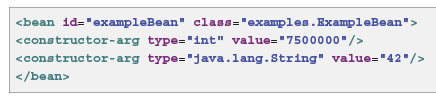
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we prevent ambiguity btw constructor arguments?

Consider the following class:

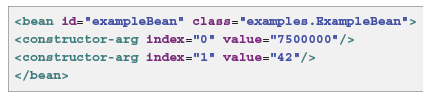


In the above example, the container can use type matching but in case of simple types if you explicitly specify the type of the constructor argument using the **type** attribute. For example:



Constructor argument index

Use the index attribute to specify explicitly the **index** of constructor arguments. For example:

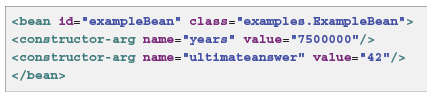


In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type.

Note that the index is 0 based.

Constructor argument name

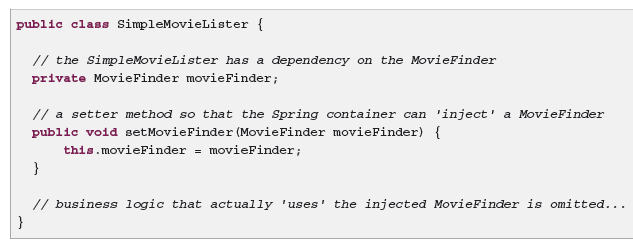
As of Spring 3.0 you can also use the constructor parameter **name** for value disambiguation:

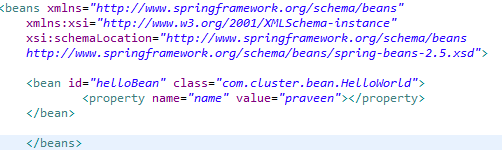


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we provide a setter based dependency injection?

*Setter-based* DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.





\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What to use Constructor-based or setter-based DI?

Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

The Spring team generally advocates setter injection, because large numbers of constructor arguments can get unwieldy, especially when properties are optional.

Setter methods also make objects of that class amenable to reconfiguration or re-injection later.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Can we make setter based dependency injections mandatory?

The use of a @Required annotation on a setter can be used to make setters required dependencies.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Is there any disadvantage in using constructor based dependency injection?

The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is a circular dependency?

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example:

Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors.

Alternatively, avoid constructor injection and use setter injection only.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

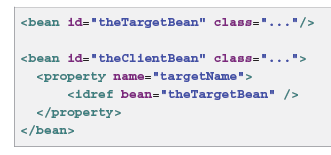
How argument to the constructor based on constructor based dependency injection resolved in the correct order?

Constructor argument resolution matching occurs using the argument's type.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is an idref element?

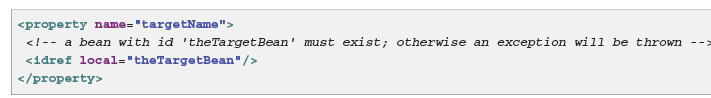
The idref element is simply an error-proof way to pass the id (string value - not a reference) of another bean in the container to a <constructor-arg/> or <property/> element.



What is the advantage of using idref element?

Using the idref tag allows the container to validate at deployment time that the referenced, named bean actually exists.

if the referenced bean is in the same XML unit, and the bean name is the bean id, you can use the local attribute, which allows the XML parser itself to validate the bean id earlier, at XML document parse time.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How are references to other beans made in a bean definition?

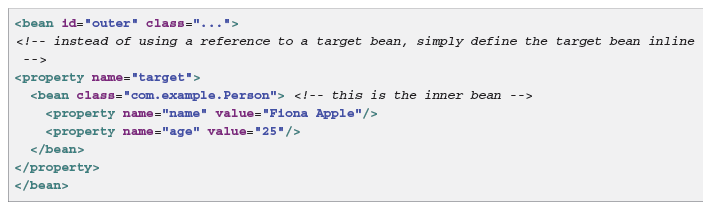
The ref element is the final element inside a <constructor-arg/> or <property/> definition element.

Here you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property will be set, and it is initialized on demand as needed before the property is set.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are inner beans?

A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean.



1. An inner bean definition does not require a defined id or name; the container ignores these values.
2. It also ignores the scope flag.
3. Inner beans are always anonymous and they are always scoped as prototypes.
4. It is not possible to inject inner beans into collaborating beans other than into the enclosing bean.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Write bean definition to dependency inject collections?

In the <list/>, <set/>, <map/>, and <props/> elements, you set the properties and arguments of the Java Collection types List, Set, Map, and Properties, respectively.









\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is collection merging?

As of Spring 2.0, the container supports the merging of collections. An application developer can 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.

That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.



Notice the use of the merge=true attribute on the <props/> element of the adminEmails property of the child bean definition.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the limitations of collection merging?

You cannot merge different collection types (such as a Map and a List), and if you do attempt to do so an appropriate Exception is thrown.

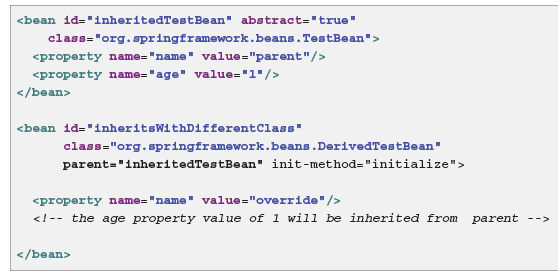
The merge attribute must be specified on the lower, inherited, child definition; specifying the merge attribute on a parent collection definition is redundant and will not result in the desired merging.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is bean definition inheritance?

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

you indicate a child bean the **parent attribute**, specifying the parent bean as the definition by using value of this attribute.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

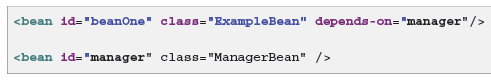
What is the use of depends-on attribute?

The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Give an example for the use of depends-on attribute?

The following example uses the depends-on attribute to express a dependency on a single bean:



To express a dependency on multiple beans, supply a list of bean names as the value of the dependson attribute, with commas, whitespace and semicolons, used as valid delimiters



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we configure lazy-initialization of beans?

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process.

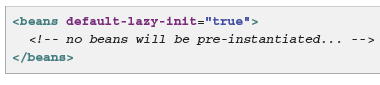
This behavior can be changed by use of the attribute **lazy-init=”true”**

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element; for example:



­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is Autowiring of beans?

It is a option available in spring with which we can avoid explicitly providing the property and constructor args in the bean definition.

By using autowiring we allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext.

Autowiring has the following advantages:

* Autowiring can significantly reduce the need to specify properties or constructor arguments.
* Autowiring can update a configuration as your objects evolve.

You specify autowire mode for a bean definition with the autowire attribute of the <bean/> element. The autowiring functionality has five modes.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

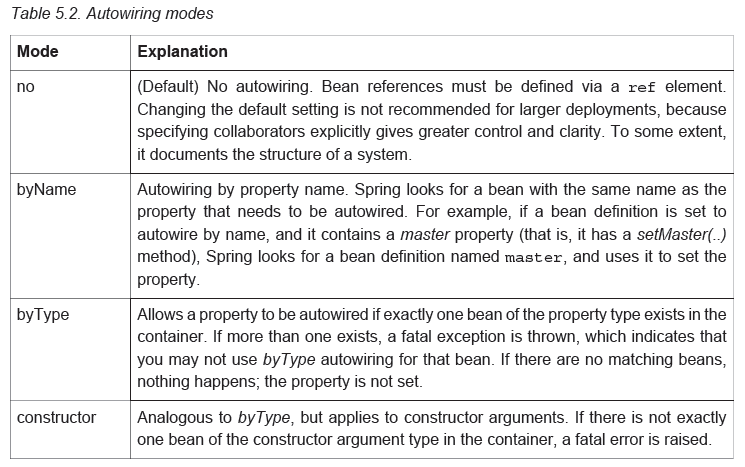
What are the disadvantages of autowiring?

Consider the limitations and disadvantages of autowiring:

* Explicit dependencies in property and constructor-arg settings always override autowiring.
* You cannot autowire so-called simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
* Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
* Wiring information may not be available to tools that may generate documentation from a Spring container.
* Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expects a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the various modes of autowiring?



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

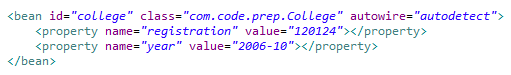
How can we prevent autowiring of beans?

On a per-bean basis, you can exclude a bean from autowiring. In Spring's XML format, set the autowire-candidate attribute of the <bean/> element to false;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the autodetect option of the autowire attribute for?

"autodetect" Chooses "constructor" or "byType" through introspection of the bean class. If a default constructor is found, "byType" gets applied. Note that explicit dependencies, i.e. "property" and "constructor-arg" elements, always override autowiring. Autowire behavior can be combined with dependency checking, which will be performed after all autowiring has been completed.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is method injection and when is it used?

In most application scenarios, most beans in the container are 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, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different.

Problem:

Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A.

The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

Lookup method injection

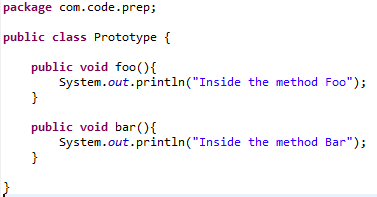
Lookup method injection is the ability of the container to override methods on container managed beans, to return the lookup result for another named bean in the container.

The lookup typically involves a prototype bean as in the scenario described in the preceding section.

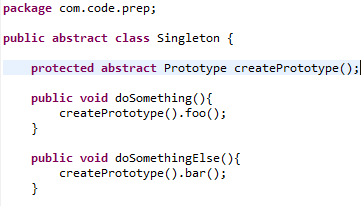
The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method.

Consider the following example:

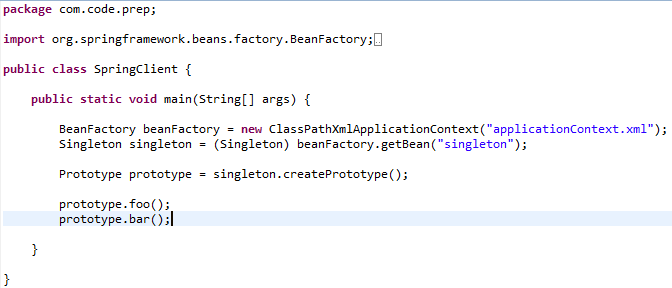
Prototype class



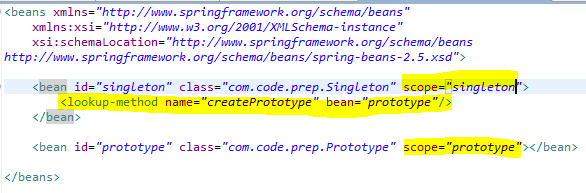
Singleton class



Spring client

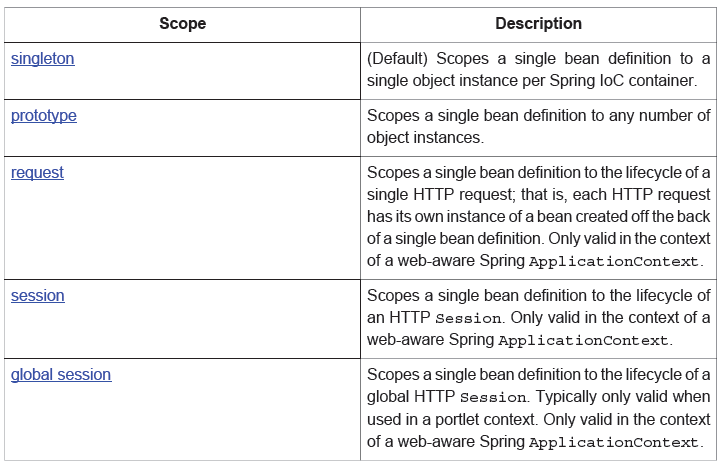


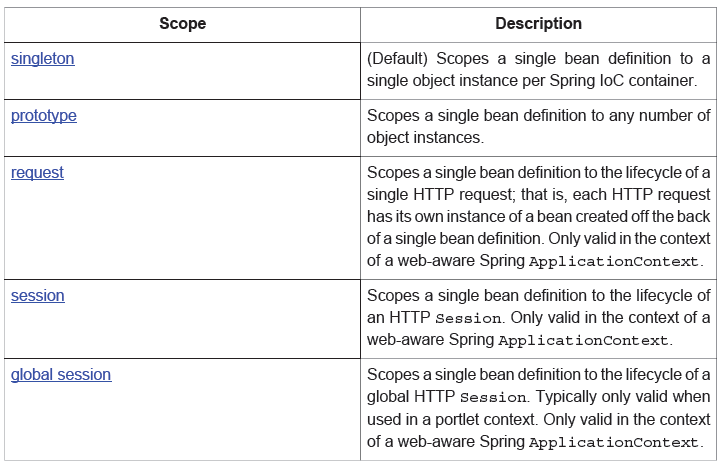
The ApplicationContext.xml



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the various scopes of a bean?





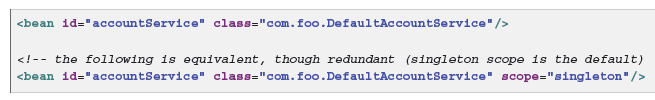
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What happens when a bean is singleton scoped?

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.

The singleton scope is the default scope in Spring.

We can define the scope of an object as follows:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is 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.



Note:

As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the scenario where we go for method injection?

We want the singleton-scoped bean to acquire a new instance of the prototypescoped 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, the we use the mechanism called “Method injection”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the lifecycle methods in Spring?

To interact with the container's management of the bean lifecycle, you can implement the Spring InitializingBean and DisposableBean interfaces.

Two interfaces for lifecycle management

InitializingBean DisposableBean

afterPropertiesSet() destroy()

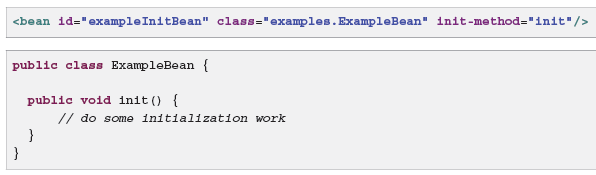
The container calls afterPropertiesSet() for the former and destroy() for the latter to allow the bean to perform certain actions upon initialization and destruction of your beans.

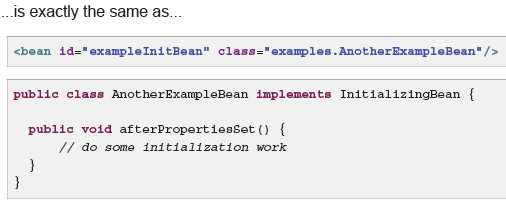
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container.



It is recommended that you do not use the InitializingBean interface because it unnecessarily couples the code to Spring. Alternatively, specify a POJO initialization method. In the case of XMLbased configuration metadata, you use the init-method attribute to specify the name of the method that has a void no-argument signature. For example, the following definition:





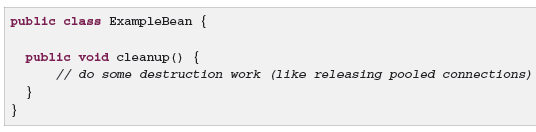
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

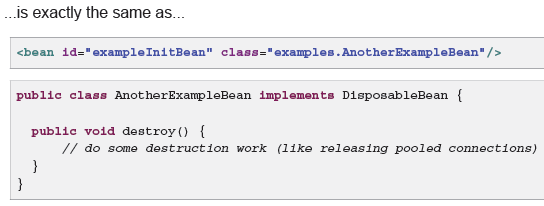
Implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a call back when the container containing it is destroyed. The DisposableBean interface specifies a single method:



Alternatively, specify a generic method that is supported by bean definitions. With XML-based configuration metadata, you use the destroy-method attribute on the <bean/>. For example, the following definition:







\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Can we specify a init() or destroy method() for all the beans in the application context at a single place?

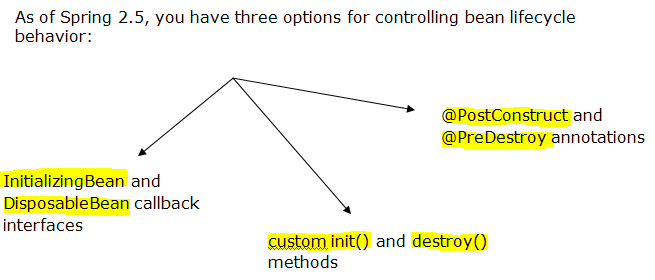
Yes we can as below:



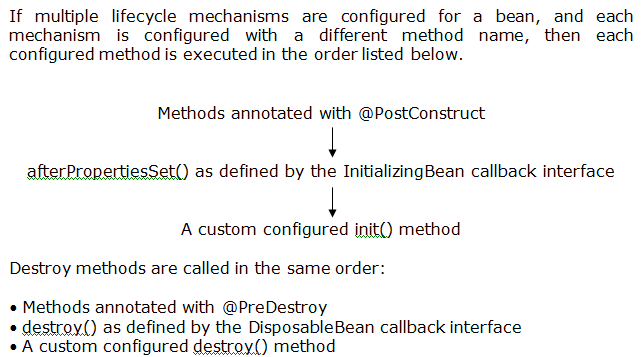
You configure destroy method callbacks similarly (in XML, that is) by using the default-destroymethod attribute on the top-level <beans/> element.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the different ways in which we can configure the bean lifecycle?



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



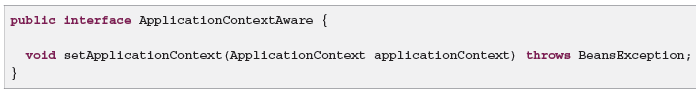
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

However, if the same method name is configured - for example, init() for an initialization method - for more than one of these lifecycle mechanisms, that method is executed only once

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can a bean access an ApplicationContext object?

When an ApplicationContext creates a class that implements the org.springframework.context.ApplicationContextAware interface, the class is provided with a reference to that ApplicationContext.



Thus beans can manipulate programmatically the ApplicationContext that created them, through the ApplicationContext interface, or by casting the reference to a known subclass of this interface, such as ConfigurableApplicationContext, which exposes additional functionality.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the BeanPostProcessor interface?

The BeanPostProcessor interface defines *callback methods* that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

When to use the BeanPostProcessor?

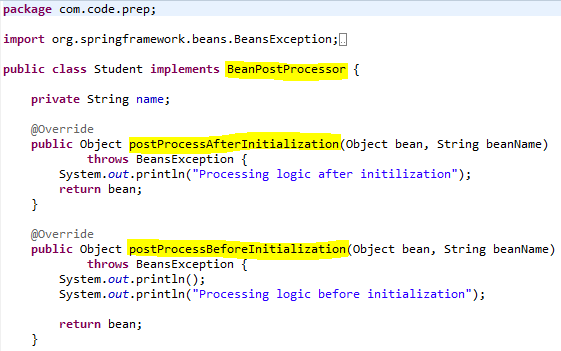
If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more BeanPostProcessor implementations.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Why do we have to use a BeanPostProcessor?

So basically the BeanPostProcessor can be used to do custom instantiation logic for several beans where the others are defined on a per bean basis.

The logic in the methods of the BeanPostProcessor i.e. **logic in the methods Before and AfterInitialization actually gets applied to all the beans other than itself** (the bean implementing the BeanPostProcessor interface).



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The purpose of BeanPostProcessor is to intercept initialization of other beans. It cannot intercept its own initialization, because Spring guarantees that post-processor is fully initialized when its post-process methods are called.

If you need to perform an action after initialization of this very bean, you can do one of the following:

* Implement InitializingBean and override afterPropertiesSet
* Use init-method attribute of <bean> to specify name of the method to be called
* Annotate the method to be called with @PostConstruct

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is a factory bean?

Implement the org.springframework.beans.factory.FactoryBean interface for objects that are themselves factories

The FactoryBean interface is a point of pluggability into the Spring IoC container's instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own FactoryBean, write the complex initialization inside that class, and then plug your custom FactoryBean into the container.

The FactoryBean interface provides three methods:

* Object getObject(): returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.
* boolean isSingleton(): returns true if this FactoryBean returns singletons, false otherwise.
* Class getObjectType(): returns the object type returned by the getObject() method or null if the type is not known in advance.

Note

So for a given FactoryBean with an id of myBean, invoking

* getBean("myBean") on the container returns the product of the FactoryBean;

Whereas, invoking

* getBean("&myBean") returns the FactoryBean instance itself.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

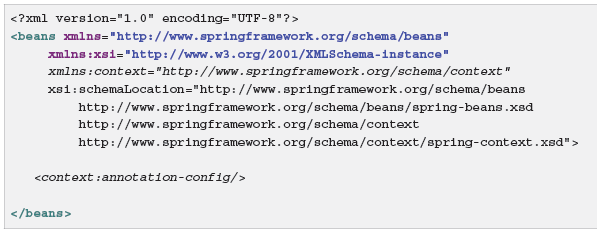
Note:

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How are annotations activated?

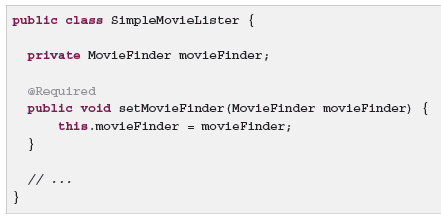
Annotations are implicitly registered by including the following tag in an XML-based Spring configuration.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is @Required?

The @Required annotation applies to bean property setter methods.



This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the importance of @Required annotation?

The container throws an exception if the affected bean property has not been populated; this allows for eager and explicit failure, avoiding NullPointerExceptions or the like later on.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is @Autowired?

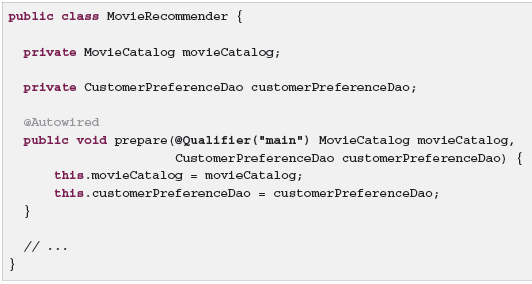
@Autowired annotation is commonly used on setter methods to get rid of the <property> and <constructor-args> element in XML configuration file.

When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is @Qualifier?

Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring's @Qualifier annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument. In the simplest case, this can be a plain descriptive value:



The corresponding bean definitions appear as follows. The bean with qualifier value "main" is wired with the constructor argument that is qualified with the same value.



For a fallback match, the bean name is considered a default qualifier value. Thus you can define the bean with an id "main" instead of the nested qualifier element, leading to the same matching result. However, although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is @Resource and why should we use it?

If you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is technically capable of referring to a bean name through @Qualifier values. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process.

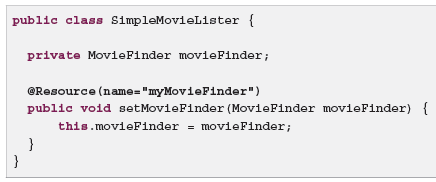
As a specific consequence of this semantic difference, beans that are themselves defined as a collection or map type cannot be injected through @Autowired, because type matching is not properly applicable to them.

* Use @Resource for such beans, referring to the specific collection or map bean by unique name.
* @Autowired applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level.
* By contrast, @Resource is supported only for fields and bean property setter methods with a single argument. As a consequence, stick with qualifiers if your injection target is a constructor or a multi-argument method.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

@Resource takes a **name attribute**, and by default Spring **interprets that value as the bean name to be injected.**

In other words, it follows **by-name** semantics, as demonstrated in this example:



If no name is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Both @Autowired (or @Inject) and @Resource work equally well. But there is a conceptual difference or a difference in the meaning

* @Resource means get me a known resource by name. The name is extracted from the name of the annotated setter or field, or it is taken from the name-Parameter.
* @Inject or @Autowired try to wire in a suitable other component by type.

So, basically these are two quite distinct concepts. Unfortunately the Spring-Implementation of @Resource has a built-in fallback, which kicks in when resolution by-name fails. In this case, it falls back to the @Autowired-kind resolution by-type. While this fallback is convenient, IMHO it causes a lot of confusion, because people are unaware of the conceptual difference and tend to use @Resource for type-based autowiring.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the use of @PostConstruct and @PreDestroy?

**@PostConstruct** and **@PreDestroy** which work similar to init-method and destroy-method in bean configuration file or implement the InitializingBean and DisposableBean in your bean class.

To use @PostConstruct and @PreDestroy you have to register the CommonAnnotationBeanPostProcessor at bean configuration or specifying the **<context: annotation-config />** in the bean configuration file.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is classpath scanning?

Classpath scanning is an option for implicitly detecting the candidate components by scanning the classpath.

Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration instead you can use annotations (for example @Component).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are stereo annotations?

@Repository annotation is a marker for any class that fulfills the role or *stereotype* (also known as Data Access Object or DAO) of a repository. Among the uses of this marker is the automatic translation of exceptions

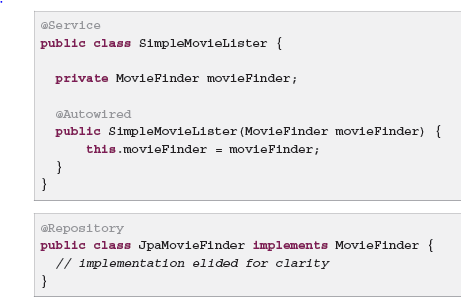
Spring 2.5 introduces further stereotype annotations:

* @Component
* @Service, and
* @Controller.

@Component is a generic stereotype for any Spring-managed component.

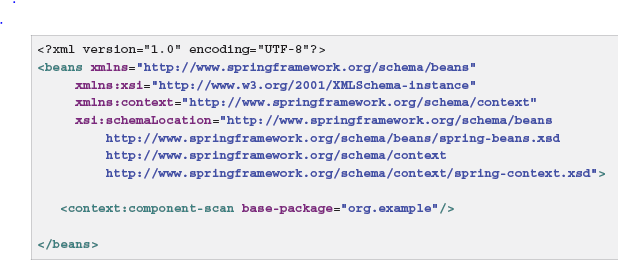
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we automatically detect classes and register their bean definitions?



To autodetect these classes and register the corresponding beans, you need to include the following element in XML, where the base-package element is a common parent package for the two classes.

(Alternatively, you can specify a comma-separated list that includes the parent package of each class.)

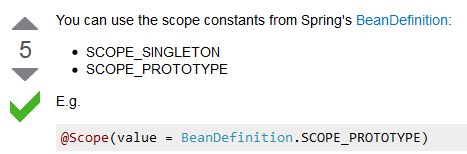


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The use of <context: component-scan > implicitly enables the function of <context: annotation-config>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we specify bean scope using annotations?



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are JSR 330 standard annotations?

These are annotations specified by Java Specification Request for dependency injection and they are @Inject and @Named

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the use of @Inject?

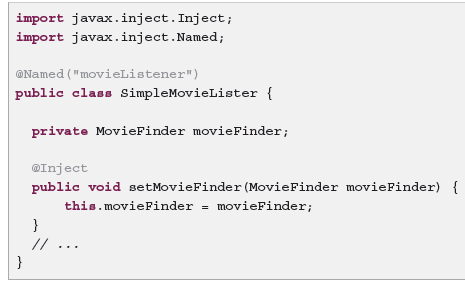
Instead of @Autowired, @javax.inject.Inject may be used as follows:



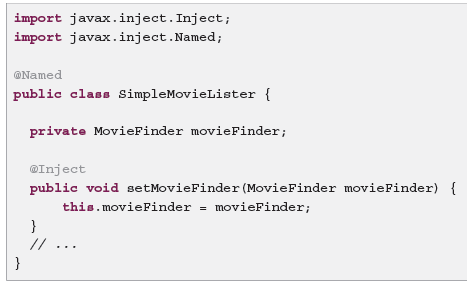
If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation as follows:



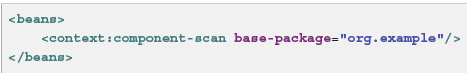
**@Named: a standard equivalent to the @Component annotation**



It is very common to use @Component without specifying a name for the component. @Named can be used in a similar fashion:

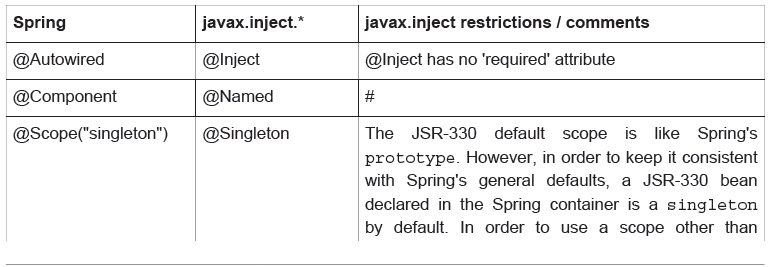


When using @Named, it is possible to use component-scanning in the exact same way as when using Spring annotations:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the limitations of the JSR annotations?



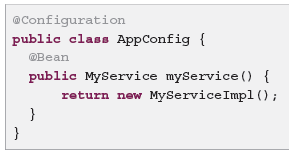


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the support for java based container configuration in Spring 3.0?

Spring 3.0 provides two annotations @Configuration and @Bean for the container to provide support for IOC

@Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions.



@Bean annotation plays the same role as the <bean/> element.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is AnnotationConfigApplicationContext?

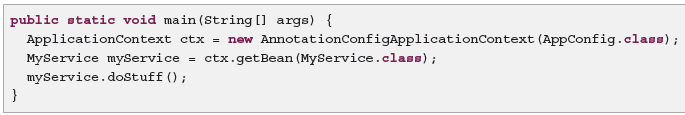
This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input, but also plain @Component classes and classes annotated with JSR-330 metadata.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we configure AnnotationConfigApplicationContext?

@Configuration classes may be used as input when instantiating an AnnotationConfigApplicationContext.

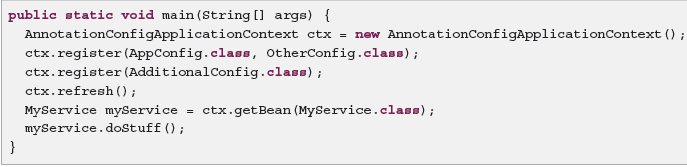
This allows for completely XML-free usage of the Spring container:



As mentioned above, AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor. For example:



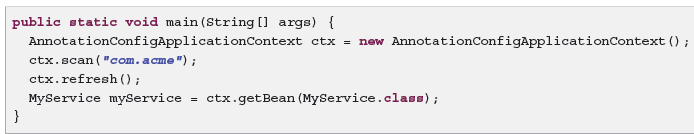
An AnnotationConfigApplicationContext may be instantiated using a no-arg constructor and then configured using the register() method.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How can we enable component scanning in AnnotationConfigApplicationContext?

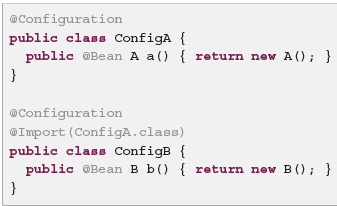
Enabling component scanning with scan(String...)



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is @Import?

Like the <import/> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_