ApplicationContext pre-instantiates all singletons by default. Therefore, it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the **abstract attribute to true**, otherwise the application context will actually (attempt to) pre-instantiate the abstract bean.

*Programmatically registering BeanPostProcessors*

While the recommended approach for BeanPostProcessor registration is through ApplicationContext auto-detection (as described above), it is also possible to register them *programmatically* against a **ConfigurableBeanFactory** using the **addBeanPostProcessor** method. This can be useful when needing to evaluate conditional logic before registration, or even for copying bean post processors across contexts in a hierarchy.

Note however that BeanPostProcessors added programmatically ***do not respect the Orderedinterface*.** Here it is the *order of registration* that dictates the order of execution. Note also that BeanPostProcessors registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering.

*BeanPostProcessors and AOP auto-proxying*

Classes that implement the BeanPostProcessor interface are *special* and are treated differently by the container. All BeanPostProcessors *and beans that they reference directly* are instantiated on startup, as part of the special startup phase of the ApplicationContext. Next, all BeanPostProcessors are registered in a sorted fashion and applied to all further beans in the container.

Because AOP auto-proxying is implemented as a BeanPostProcessor itself, neither BeanPostProcessors nor the beans they reference directly are eligible for auto-proxying, and thus do not have aspects woven into them.

For any such bean, you should see an informational log message: "*Bean foo is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)*".

Note that if you have beans wired into your BeanPostProcessor using autowiring or @Resource (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates, and therefore make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with @Resource where the field/setter name does not directly correspond to the declared name of a bean and no name attribute is used, then Spring will access other beans for matching them by type.

**public** **class** **InstantiationTracingBeanPostProcessor** **implements** BeanPostProcessor {

*// simply return the instantiated bean as-is*

**public** Object postProcessBeforeInitialization(Object bean, String beanName) {

**return** bean; *// we could potentially return any object reference here...*

}

**public** Object postProcessAfterInitialization(Object bean, String beanName) {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

**return** bean;

}

}

BeanFactoryPostProcessor

BeanPostProcessor, with one major difference: BeanFactoryPostProcessor operates on the bean configuration metadata; that is, the Spring IoC container allows a BeanFactoryPostProcessor to read the configuration metadata and potentially change it before the container instantiates any beans other than BeanFactoryPostProcessors.

 you want to change the actual bean *instances* (i.e., the objects that are created from the configuration metadata), then you instead need to use a BeanPostProcessor (described above in [Customizing beans using a BeanPostProcessor](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html#beans-factory-extension-bpp)). While it is technically possible to work with bean instances within a BeanFactoryPostProcessor (e.g., using BeanFactory.getBean()), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects such as bypassing bean post processing.

Also, BeanFactoryPostProcessors are scoped *per-container*. This is only relevant if you are using container hierarchies. If you define a BeanFactoryPostProcessor in one container, it will *only* be applied to the bean definitions in that container. Bean definitions in one container will not be post-processed by BeanFactoryPostProcessors in another container, even if both containers are part of the same hierarchy.

As with BeanPostProcessors , you typically do not want to configure BeanFactoryPostProcessors for lazy initialization. If no other bean references a Bean(Factory)PostProcessor, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the Bean(Factory)PostProcessorwill be instantiated eagerly even if you set the default-lazy-init attribute to true on the declaration of your <beans /> element.

PropertyPlaceholderConfigurer

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations" value="classpath:com/foo/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

Using context namespacing

<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>

You can use the PropertyPlaceholderConfigurer to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<value>classpath:com/foo/strategy.properties</value>

</property>

<property name="properties">

<value>custom.strategy.class=com.foo.DefaultStrategy</value>

</property>

</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>

If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the preInstantiateSingletons() phase of an ApplicationContext for a non-lazy-init bean.

org.springframework.beans.factory.FactoryBean

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.

When you need to ask a container for an actual FactoryBean instance itself instead of the bean it produces, preface the bean’s id with the ampersand symbol ( &) when calling the getBean() method of the ApplicationContext. So for a given FactoryBeanwith 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.

Annotation Based configurations

its pros and cons.

Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration.

However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It’s worth pointing out that through its [JavaConfig](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html#beans-java) option, Spring allows annotations to be used in a non-invasive way, without touching the target components source code and that in terms of tooling, all configuration styles are supported by the [Spring Tool Suite](https://spring.io/tools/sts).

ANNOTATIONS

As always, you can register them as individual bean definitions, but they can also be implicitly registered by including the following tag in an XML-based Spring configuration (notice the inclusion of the context namespace):

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

https://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

https://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

</beans>

(The implicitly registered post-processors include [AutowiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html),[CommonAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/CommonAnnotationBeanPostProcessor.html), [PersistenceAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html), as well as the aforementioned[RequiredAnnotationBeanPostProcessor](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html).)

<context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined. This means that, if you put <context:annotation-config/> in a WebApplicationContext for a DispatcherServlet, it only checks for @Autowired beans in your controllers, and not your services. See [The DispatcherServlet](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/web.html#mvc-servlet) for more information

**@Required**

The @Required annotation applies to bean property setter methods,

@Required

public void setMovieFinder(MovieFinder movieFinder) {

property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. 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. It is still recommended that you put assertions into the bean class itself, for example, into an init method. Doing so enforces those required references and values even when you use the class outside of a container.

**@Autowired**

As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at **least one must be annotated** to teach the container which one to use.

Make sure that your target components (e.g. MovieCatalog, CustomerPreferenceDao) are consistently declared by the type that you are using for your @Autowired-annotated injection points. Otherwise injection may fail due to no type match found at runtime.

For XML-defined beans or component classes found through a classpath scan, the container usually knows the concrete type upfront. However, for @Bean factory methods, you need to make sure that the **declared return type is sufficiently expressive**. For components implementing several interfaces or for components potentially referred to by their implementation type, consider declaring the most specific return type on your factory method (at least as specific as required by the injection points referring to your bean).

It is also possible to provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method that expects an array of that type:

@Autowired

**private** MovieCatalog**[]** movieCatalogs;

**private** Set<MovieCatalog> movieCatalogs;

@Autowired

**public** **void** setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

Your target beans can implement the org.springframework.core.Ordered interface or use the @Order or standard @Priority annotation if you want items in the array or list to be sorted in a specific order. Otherwise their order will follow the **registration order of the corresponding** target bean definitions in the container.

The @**Order annotation may be declared at target class level but also on @Bean methods,** potentially being very individual per bean definition (in case of multiple definitions with the same bean class**). @Order values may influence priorities at injection points, but please be aware that they do not influence singleton startup order which is an orthogonal concern determined by** dependency relationships and @DependsOn declarations.

Note that the standard **javax.annotation.Priority annotation is not available at the @Bean level** since it cannot be declared on methods. Its semantics can be modeled through @Order values in combination with @Primaryon a single bean per type.

**private** Map<String, MovieCatalog> movieCatalogs;

@Autowired

**public** **void** setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {

???????

**By default, the autowiring fails whenever zero candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating required dependencies**. This behavior can be changed as demonstrated below

@Autowired(required = false)

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**Only *one annotated constructor per-class* can be marked as *required***, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the *greediest*constructor whose dependencies can be satisfied, that is the constructor that has the largest number of arguments.

The *required* attribute of @Autowired is recommended over the @Required annotation. The *required* attribute indicates that the property is not required for autowiring purposes, the property is ignored if it cannot be autowired. @Required, on the other hand, is stronger in that it enforces the property that was set by any means supported by the container. If no value is injected, a corresponding exception is raised.

Alternatively, you may express the non-required nature of a particular dependency through Java 8’s java.util.Optional:

**public** **class** **SimpleMovieLister** {

@Autowired

**public** **void** setMovieFinder(Optional<MovieFinder> movieFinder) {

As of Spring Framework 5.0, you may also use an @Nullable annotation (of any kind in any package, e.g. javax.annotation.Nullable from JSR-305):

**public** **class** **SimpleMovieLister** {

@Autowired

**public** **void** setMovieFinder(@Nullable MovieFinder movieFinder) {

You can also use @Autowired for interfaces that are well-known resolvable dependencies: BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource. These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

@Autowired, @Inject, @Resource, and @Value annotations are handled by Spring BeanPostProcessorimplementations which in turn means that you cannot apply these annotations within your own BeanPostProcessor or BeanFactoryPostProcessor types (if any). These types must be 'wired up' explicitly via XML or using a Spring @Bean method.

@Primary

@Autowired

**@Qualifier("main")**

**private** MovieCatalog movieCatalog;

@Autowired

**public** **void** prepare(**@Qualifier("main")**MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

Letting qualifier values select against target bean names, within the type-matching candidates, doesn’t even require a @Qualifier annotation at the injection point. If there is no other resolution indicator (e.g. a qualifier or a primary marker), **for a non-unique dependency situation, Spring will match the injection point name (**i.e. field name or parameter name) against the target bean names and choose the same-named candidate, if any

That said, if you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is capable of selecting by bean name among type-matching candidates. **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. @Autowired has rather different semantics: After selecting candidate beans by type, the specified String qualifier value will be considered within those type-selected candidates only, e.g. matching an "account" qualifier against beans marked with the same qualifier label.

For beans that are themselves defined as a collection/map or array type, @Resource is a fine solution, referring to the specific collection or array bean by unique name. That said, as of 4.3, collection/map and array types can be matched through Spring’s @Autowired type matching algorithm as well, as long as the element type information is preserved in @Bean return type signatures or collection inheritance hierarchies. In this case, qualifier values can be used to select among same-typed collections, as outlined in the previous paragraph.

As of 4.3, @Autowired also considers self references for injection, i.e. references back to the bean that is currently injected. Note that self injection is a fallback; regular dependencies on other components always have precedence. In that sense, self references do not participate in regular candidate selection and are therefore in particular never primary; on the contrary, they always end up as lowest precedence. In practice, use self references as a last resort only, e.g. for calling other methods on the same instance through the bean’s transactional proxy: Consider factoring out the affected methods to a separate delegate bean in such a scenario. Alternatively, use @Resource which may obtain a proxy back to the current bean by its 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.

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

**@Qualifier**

**public** @interface Genre {

String value();

}

Next, provide the information for the candidate bean definitions. You can add <qualifier/> tags as sub-elements of the <bean/> tag and then specify the type and value to match your custom qualifier annotations. The type is matched against the fully-qualified class name of the annotation. Or, as a convenience if no risk of conflicting names exists, you can use the short class name. Both approaches are demonstrated in the following example.

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Genre" value="Action"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="example.Genre" value="Comedy"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

Annotation without value

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface Offline {

}

@Autowired

**@Offline**

**private** MovieCatalog offlineCatalog;

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Offline"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface MovieQualifier {

String genre();

Format format();

}

In this case Format is an enum:

**public** **enum** Format {

VHS, DVD, BLURAY

}

**public** **class** **MovieRecommender** {

@Autowired

@MovieQualifier(format=Format.VHS, genre="Action")

**private** MovieCatalog actionVhsCatalog;

@Autowired

@MovieQualifier(format=Format.VHS, genre="Comedy")

**private** MovieCatalog comedyVhsCatalog;

============

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Comedy"/>

</qualifier>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="DVD"/>

<meta key="genre" value="Action"/>

*<!-- inject any dependencies required by this bean -->*

</bean>

CustomAutowireConfigurer

The [CustomAutowireConfigurer](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/beans/factory/annotation/CustomAutowireConfigurer.html) is a BeanFactoryPostProcessor that enables you to register your own custom qualifier annotation types even if they are not annotated with Spring’s @Qualifier annotation.

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

The AutowireCandidateResolver determines autowire candidates by:

* the autowire-candidate value of each bean definition
* any default-autowire-candidates pattern(s) available on the <beans/> element
* the presence of @Qualifier annotations and any custom annotations registered with the CustomAutowireConfigurer

When multiple beans qualify as autowire candidates, the determination of a "primary" is the following: if exactly one bean definition among the candidates has a primary attribute set to true, it will be selected.

#### @Resource

**private** MovieFinder movieFinder;

**@Resource(name="myMovieFinder")**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

The name provided with the annotation is resolved as a bean name by the ApplicationContext of which the CommonAnnotationBeanPostProcessor is aware. The names can be resolved through JNDI if you configure Spring’s [SimpleJndiBeanFactory](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/jndi/support/SimpleJndiBeanFactory.html) explicitly. However, it is recommended that you rely on the default behavior and simply use Spring’s JNDI lookup capabilities to preserve the level of indirection.

In the exclusive case of @Resource usage with no explicit name specified, and similar to @Autowired**, @Resource finds a primary** type match instead of a specific named bean and resolves well-known resolvable dependencies: the BeanFactory,ApplicationContext, ResourceLoader, ApplicationEventPublisher, and MessageSource interfaces.

In the exclusive case of @Resource usage with no explicit name specified, and similar to @Autowired, @Resource finds a primary type match instead of a specific named bean and resolves well-known resolvable dependencies: the BeanFactory, ApplicationContext, ResourceLoader, ApplicationEventPublisher, and MessageSource interfaces.

Thus in the following example, the customerPreferenceDao field first looks for a bean named customerPreferenceDao, then falls back to a primary type match for the type CustomerPreferenceDao. The "context" field is injected based on the known resolvable dependency type ApplicationContext.

# Spring Annotation Programming Model

## Meta-Annotations

A ***meta-annotation*** is an annotation that is declared on another annotation. An annotation is therefore meta-annotated if it is annotated with another annotation. For example, any annotation that is declared to be documented is meta-annotated with @**Documented** from the **java.lang.annotation** package.

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**@Component** *// Spring will see this and treat @Service in the same way as @Component*

**public** @interface Service {

*// ....*

}

## Stereotype Annotations

A ***stereotype annotation*** is an annotation that is used to declare the role that a component plays within the application. For example, the @Repository annotation in the Spring Framework is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO).

@Component is a generic stereotype for any Spring-managed component. Any component annotated with @Component is a candidate for component scanning. Similarly, any component annotated with an annotation that is itself meta-annotated with @Component is also a candidate for component scanning. For example, @Service is meta-annotated with @Component.

Core Spring provides several stereotype annotations out of the box, including but not limited to: @Component, @Service, @Repository, @Controller, @RestController, and @Configuration. @Repository, @Service, etc. are specializations of @Component.

## Composed Annotations

A ***composed annotation*** is an annotation that is meta-annotated with one or more annotations with the intent of combining the behavior associated with those meta-annotations into a single custom annotation. For example, an annotation named @TransactionalService that is meta-annotated with Spring's @Transactional and @Service annotations is a composed annotation that combines the semantics of @Transactional and @Service. @TransactionalService is technically also a custom stereotype annotation.

In addition, composed annotations may optionally redeclare attributes from meta-annotations to allow user customization. This can be particularly useful when you want to only expose a subset of the meta-annotation’s attributes. For example, Spring’s@SessionScope annotation hardcodes the scope name to session but still allows customization of the proxyMode.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Scope(WebApplicationContext.SCOPE\_SESSION)

**public** @interface SessionScope {

*/\*\**

*\* Alias for {@link Scope#proxyMode}.*

*\* <p>Defaults to {@link ScopedProxyMode#TARGET\_CLASS}.*

*\*/*

@AliasFor(annotation = Scope.class)

ScopedProxyMode proxyMode() **default** ScopedProxyMode.TARGET\_CLASS;

}

@SessionScope can then be used without declaring the proxyMode as follows:

@Service

**@SessionScope**

**public** **class** **SessionScopedService** {

*// ...*

}

Or with an overridden value for the proxyMode as follows:

@Service

**@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)**

**public** **class** **SessionScopedUserService** **implements** UserService {

*// ...*

}

## Annotation Presence

The terms ***directly present***, ***indirectly present***, and ***present*** have the same meanings as defined in the class-level Javadoc for **java.lang.reflect.AnnotatedElement** in Java 8.

In Spring, an annotation is considered to be ***meta-present*** on an element if the annotation is declared as a meta-annotation on some other annotation which is present on the element. For example, given the aforementioned @TransactionalService, we would say that @Transactional is meta-present on any class that is directly annotated with @TransactionalService.

## Attribute Aliases and Overrides

An ***attribute alias*** is an alias from one annotation attribute to another annotation attribute. Attributes within a set of aliases can be used interchangeably and are treated as equivalent. Attribute aliases can be categorized as follows.

1. **Explicit Aliases**: if two attributes in one annotation are declared as aliases for each other via @AliasFor, they are explicit aliases.
2. **Implicit Aliases**: if two or more attributes in one annotation are declared as explicit overrides for the same attribute in a meta-annotation via @AliasFor, they are implicit aliases.
3. **Transitive Implicit Aliases**: given two or more attributes in one annotation that are declared as explicit overrides for attributes in meta-annotations via @AliasFor, if the attributes effectivelyoverride the same attribute in a meta-annotation following the [law of transitivity](https://en.wikipedia.org/wiki/Transitive_relation), they are transitive implicit aliases.

An ***attribute override*** is an annotation attribute that overrides (or shadows) an annotation attribute in a meta-annotation. Attribute overrides can be categorized as follows.

1. **Implicit Overrides**: given attribute A in annotation @One and attribute A in annotation @Two, if @One is meta-annotated with @Two, then attribute A in annotation @One is an implicit override for attribute A in annotation @Two based solely on a naming convention (i.e., both attributes are named A).
2. **Explicit Overrides**: if attribute A is declared as an alias for attribute B in a meta-annotation via @AliasFor, then A is an explicit override for B.
3. **Transitive Explicit Overrides**: if attribute A in annotation @One is an explicit override for attribute B in annotation @Two and B is an explicit override for attribute C in annotation @Three, then A is a transitive explicit override for C following the [law of transitivity](https://en.wikipedia.org/wiki/Transitive_relation).

To autodetect these classes and register the corresponding beans, you need to add @ComponentScan to your @Configurationclass, where the basePackages attribute is a common parent package for the two classes. (Alternatively, you can specify a comma/semicolon/space-separated list that includes the parent package of each class.)

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

https://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

https://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.example"/>

</beans>

The use of <context:component-scan> implicitly enables the functionality of <context:annotation-config>. There is usually no need to include the <context:annotation-config> element when using <context:component-scan>.

Furthermore, the AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor are both included implicitly when you use the component-scan element. That means that the two components are autodetected and wired together - all without any bean configuration metadata provided in XML.

You can disable the registration of AutowiredAnnotationBeanPostProcessor andCommonAnnotationBeanPostProcessor by including the *annotation-config* attribute with a value of false.

#### Using filters to customize scanning

By default, classes annotated with **@Component, @Repository, @Service, @Controller**, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior simply by applying custom filters. Add them as ***includeFilters* or *excludeFilters*** parameters of the @ComponentScan annotation (or as *include-filter* or *exclude-filter* sub-elements of the component-scan element). Each filter element requires the type and expression attributes. The following table describes the filtering options

| **Filter Type** | **Example Expression** | **Description** |
| --- | --- | --- |
| annotation (default) | org.example.SomeAnnotation | An annotation to be present at the type level in target components. |
| assignable | org.example.SomeClass | A class (or interface) that the target components are assignable to (extend/implement). |
| aspectj | org.example..\*Service+ | An AspectJ type expression to be matched by the target components. |
| regex | org\.example\.Default.\* | A regex expression to be matched by the target components class names. |
| custom | org.example.MyTypeFilter | A custom implementation of the org.springframework.core.type .TypeFilter interface. |

The following example shows the configuration ignoring all @Repository annotations and using "stub" repositories instead.

@Configuration

@ComponentScan(basePackages = "org.example",

includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example">

<context:include-filter type="regex"

expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

You can also disable the default filters by setting **useDefaultFilters=false** on the annotation or providing use-default-filters="false" as an attribute of the <component-scan/> element. This will in effect disable automatic detection of classes annotated with **@Component, @Repository, @Service, @Controller, or @Configuration.**

*// use of a custom qualifier and autowiring of method parameters*

@Bean

**protected** TestBean protectedInstance(

@Qualifier("public") TestBean spouse,

@Value("#{privateInstance.age}") String country) {

TestBean tb = **new** TestBean("protectedInstance", 1);

tb.setSpouse(spouse);

tb.setCountry(country);

**return** tb;

}

@Bean

**private** TestBean privateInstance() {

**return** **new** TestBean("privateInstance", i++);

}

@Bean

@RequestScope

**public** TestBean requestScopedInstance() {

**return** **new** TestBean("requestScopedInstance", 3);

}

The example autowires the String method parameter country to the value of the age property on another bean named privateInstance. A Spring Expression Language element defines the value of the property through the notation #{ <expression> }. For @Value annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

**You may declare @Bean methods as static, allowing for them to be called without creating their containing configuration class as an instance. This makes particular sense when defining post-processor beans, e.g. of type BeanFactoryPostProcessor or BeanPostProcessor, since such beans will get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.**

Note that calls to static **@Bean methods will never get intercepted by the container, not even within @Configuration classes** (see above). This is due to technical limitations: CGLIB **subclassing can only override non-static methods**. As a consequence, a direct call to another @Bean method will have standard Java semantics, resulting in an independent instance being returned straight from the factory method itself.

The Java language visibility of @Bean methods does not have an immediate impact on the resulting bean definition in **Spring’s container. You may freely declare your factory methods as you see fit in non-@Configuration classes and also for static methods anywhere**. However, regular @Bean methods in @Configuration classes need to **be overridable, i.e. they must not be declared as private or final.**

@Bean methods will also be discovered on base classes of a given component or configuration class, as well as on Java 8 default methods declared in interfaces implemented by the component or configuration class. This allows for a lot of flexibility in composing complex configuration arrangements, with even multiple inheritance being possible through Java 8 default methods as of Spring 4.2.

Finally, note that a single class may hold **multiple @Bean methods for the same bean**, as an arrangement of multiple factory methods to use depending on available dependencies at runtime. This is the same algorithm as for choosing the "greediest" constructor or factory method in other configuration scenarios: **The variant with the largest number of satisfiable dependencies will be picked at construction time, analogous to how the container selects between multiple @Autowired constructors.**

Custom Bean naming statergy

If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the [**BeanNameGenerator**](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/beans/factory/support/BeanNameGenerator.html) interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner:

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

**public** **class** **AppConfig** {

...

}

Same as

<beans>

<context:component-scan base-package="org.example"

name-generator="org.example.MyNameGenerator" />

</beans>

**@Scope annotations are only introspected on the concrete bean class (for annotated components) or the factory method (for @Bean methods).** In contrast to XML bean definitions, there is **no notion of bean definition inheritance**, and inheritance hierarchies at the class level are irrelevant for metadata purposes.

Custom Scope Resolver

To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, implement the [**ScopeMetadataResolver**](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/ScopeMetadataResolver.html) interface, and be **sure to include a default no-arg constructor**. Then, provide the fully-qualified class name when configuring the scanner

@Configuration

@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example" scope-resolver="org.example.MyScopeResolver"/>

</beans>

????

@Configuration

@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example" scoped-proxy="interfaces"/>

</beans>

#### Generating an index of candidate components

While classpath scanning is very fast, it is possible to improve the startup performance of large applications by creating a static list of candidates at compilation time. In this mode, all modules of the application must use this mechanism as, when the ApplicationContext detects such index, it will automatically use it rather than scanning the classpath.

To generate the index, simply add an additional dependency to each module that contains components that are target for component scan directives:

dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-indexer</artifactId>

<version>5.0.14.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

For gradle

dependencies {

compileOnly("org.springframework:spring-context-indexer:5.0.14.RELEASE")

}

That process will generate a **META-INF/spring.components** file that is going to be included in the jar.

When working with this mode in your IDE, the spring-context-indexer must be registered as an annotation processor to make sure the index is up to date when candidate components are updated.

The index is enabled automatically when a META-INF/spring.components is found on the classpath. If an index is partially available for some libraries (or use cases) but couldn’t be built for the whole application, you can fallback to a regular classpath arrangement (i.e. as no index was present at all) by setting spring.index.ignoreto true, either as a system property or in a spring.properties file at the root of the classpath.

#### Limitations of JSR-330 standard annotations

| **Spring** | **javax.inject.\*** | **javax.inject restrictions / comments** |
| --- | --- | --- |
| @Autowired | @Inject | @Inject has no 'required' attribute; can be used with Java 8’s Optional instead. |
| @Component | @Named / @ManagedBean | JSR-330 does not provide a composable model, just a way to identify named components. |
| @Scope("singleton") | @Singleton | The JSR-330 default scope is like Spring’s prototype. However, in order to keep it consistent with Spring’s general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring’s @Scope annotation. javax.inject also provides a [@Scope](https://download.oracle.com/javaee/6/api/javax/inject/Scope.html)annotation. Nevertheless, this one is only intended to be used for creating your own annotations. |
| @Qualifier | @Qualifier / @Named | javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring’s @Qualifier with a value) can be associated through javax.inject.Named. |
| @Value | - | no equivalent |
| @Required | - | no equivalent |
| @Lazy | - | no equivalent |
| ObjectFactory | Provider | javax.inject.Provider is a direct alternative to Spring’s ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring’s @Autowiredor with non-annotated constructors and setter methods. |

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

**import** javax.inject.Inject;

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

As with @Autowired, it is possible to use @Inject at the field level, method level and constructor-argument level. Furthermore, you may declare your injection point as a Provider, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a Provider.get() call. As a variant of the example above

**import** javax.inject.Inject;

**import** javax.inject.Provider;

**public** **class** **SimpleMovieLister** {

**private** Provider<MovieFinder> movieFinder;

@Inject

**public** **void** setMovieFinder(Provider<MovieFinder> movieFinder) {

this.movieFinder = movieFinder;

}

**public** **void** listMovies() {

this.movieFinder.get().findMovies(...);

...

}

}

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

**import** javax.inject.Inject;

**import** javax.inject.Named;

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(@Named("main") MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

Like @Autowired, @Inject can also be used with java.util.Optional or @Nullable. This is even more applicable here since @Inject does not have a required attribute.

**public** **class** **SimpleMovieLister** {

@Inject

**public** **void** setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

**public** **class** **SimpleMovieLister** {

@Inject

**public** **void** setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

Full @Configuration vs 'lite' @Bean mode?

When @Bean methods are declared within classes that are *not* annotated with @Configuration they are referred to as being processed in a **'lite'** mode. Bean methods declared in a @Component or even in a *plain old class* will be considered 'lite', with a different primary purpose of the containing class and an @Bean method just being a sort of bonus there. For example, service components may expose management views to the container through an additional @Bean method on each applicable component class. In such scenarios, @Bean methods are a simple general-purpose factory method mechanism.

Unlike full @Configuration, lite @Bean methods **cannot declare inter-bean dependencies**. Instead, they operate on their containing component’s internal state and optionally on arguments that they may declare. Such an @Bean method should therefore not invoke other @Bean methods; **each such method is literally just a factory method for a particular bean reference, without any special runtime semantics.** The positive side-effect here is that **no CGLIB subclassing** has to be applied at runtime, so there are no limitations in terms of class design (i.e. the containing class may nevertheless be final etc).

In common scenarios, @Bean methods are to be declared within @Configuration classes, ensuring that 'full' mode is always used and that cross-method references will therefore get redirected to the container’s lifecycle management. This will prevent the same @Bean method from accidentally being invoked through a regular Java call which helps to reduce subtle bugs that can be hard to track down when operating in 'lite' mode.

#### Instantiating the Spring container using

#### AnnotationConfigApplicationContext

The sections below document Spring’s AnnotationConfigApplicationContext, new in Spring 3.0. 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.

When **@Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition, and all declared @Bean methods within the class are also registered as bean definitions.**

When @Component and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as @Autowired or @Inject are used within those classes where necessary.

##### **Simple construction**

In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, @Configuration classes may be used as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container:

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

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:

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

The above assumes that MyServiceImpl, Dependency1 and Dependency2 use Spring dependency injection annotations such as @Autowired.

##### **Building the container programmatically using register(Class<?>…​)**

An AnnotationConfigApplicationContext may be instantiated using a no-arg constructor and then configured using the register() method. This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext.

**public** **static** **void** main(String**[]** args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

**ctx.refresh();**

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

##### **Enabling component scanning with scan(String…​)**

To enable component scanning, just annotate your @Configuration class as follows:

@Configuration

@ComponentScan(basePackages = "com.acme")

**public** **class** **AppConfig** {

...

}

|  |
| --- |
| Experienced Spring users will be familiar with the XML declaration equivalent from Spring’s   context:namespace  <beans>  <context:component-scan base-package="com.acme"/>  </beans> |

In the example above, the com.acme package will be scanned, looking for any @Component-annotated classes, and those classes will be registered as Spring bean definitions within the container. AnnotationConfigApplicationContext exposes thescan(String…​) method to allow for the same component-scanning functionality:

**public** **static** **void** main(String**[]** args) {

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

}

Remember that @Configuration classes are [meta-annotated](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html#beans-meta-annotations) with @Component, so they are candidates for component-scanning! In the example above, assuming that **AppConfig** is declared within the **com.acme** package (or any package underneath), it will be picked up during the call to scan(), and upon refresh() all its @Beanmethods will be processed and registered as bean definitions within the container.

##### **Support for web applications with AnnotationConfigWebApplicationContext**

A **WebApplicationContext** variant of **AnnotationConfigApplicationContext** is available with **AnnotationConfigWebApplicationContext**. This implementation may be used when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet, etc. What follows is a web.xml snippet that configures a typical Spring MVC web application. **Note the use of the contextClass context-param and init-param:**

<web-app>

*<!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

*<!-- Configuration locations must consist of one or more comma- or space-delimited*

*fully-qualified @Configuration classes. Fully-qualified packages may also be*

*specified for component-scanning -->*

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.AppConfig</param-value>

</context-param>

*<!-- Bootstrap the root application context as usual using ContextLoaderListener -->*

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

*<!-- Declare a Spring MVC DispatcherServlet as usual -->*

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

*<!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

*<!-- Again, config locations must consist of one or more comma- or space-delimited*

*and fully-qualified @Configuration classes -->*

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.web.MvcConfig</param-value>

</init-param>

</servlet>

*<!-- map all requests for /app/\* to the dispatcher servlet -->*

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

You may also declare your @Bean method with an interface (or base class) return type:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl();

}

}

However, this limits the visibility for advance type prediction to the specified interface type (TransferService) then, with the full type (TransferServiceImpl) only known to the container once the affected singleton bean has been instantiated. Non-lazy singleton beans get instantiated according to their declaration order, so you may see different type matching results depending on when another component tries to match by a non-declared type (such as @Autowired TransferServiceImpl which will only resolve once the "transferService" bean has been instantiated).

If you consistently refer to your types by a declared service interface, your @Bean return types may safely join that design decision. However, for components implementing several interfaces or for components potentially referred to by their implementation type, it is safer to declare the most specific return type possible (at least as specific as required by the injection points referring to your bean).

**public** **class** **Foo** {

**public** **void** init() {

*// initialization logic*

}

}

**public** **class** **Bar** {

**public** **void** cleanup() {

*// destruction logic*

}

}

@Configuration

**public** **class** **AppConfig** {

@Bean(initMethod = "init")

**public** Foo foo() {

**return** **new** Foo();

}

@Bean(destroyMethod = "cleanup")

**public** Bar bar() {

**return** **new** Bar();

}

}

|  |  |
| --- | --- |
|  |  |

By default, beans defined using Java config that **have a public close or shutdown method are automatically enlisted with a destruction callback. If you have a public close** or shutdown method and you do not wish for it to be called when the container shuts down, simply add @Bean(destroyMethod="") to your bean definition to disable the default (inferred) mode.

You may want to do that by default for a resource that you acquire via JNDI as its lifecycle is managed outside the application. In particular, make sure to always do it for a DataSource as it is known to be problematic on Java EE application servers.

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** NamingException {

**return** (DataSource) jndiTemplate.lookup("MyDS");

}

Also, with @Bean methods, you will typically choose to use programmatic JNDI lookups: either using Spring’s JndiTemplate/JndiLocatorDelegate helpers or straight JNDI InitialContext usage, but not the JndiObjectFactoryBean variant which would force you to declare the return type as the FactoryBean type instead of the actual target type, making it harder to use for cross-reference calls in other @Bean methods that intend to refer to the provided resource here.

##### **Injecting inter-bean dependencies**

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

**return** **new** Foo(bar());

}

@Bean

**public** Bar bar() {

**return** **new** Bar();

}

}

**This method of declaring inter-bean dependencies only works when the @Bean method is declared within a @Configuration class**. You cannot declare inter-bean dependencies using plain @Component classes.

##### **Lookup method injection**

As noted earlier, [lookup method injection](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html#beans-factory-method-injection) is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern.

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object 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();

}

Using Java-configuration support , you can create a subclass of CommandManager where the abstract createCommand() method is overridden in such a way that it looks up a new (prototype) command object:

@Bean

@Scope("prototype")

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// inject dependencies here as required*

**return** command;

}

@Bean

**public** CommandManager commandManager() {

*// return new anonymous implementation of CommandManager with command() overridden*

*// to return a new prototype Command object*

**return** **new** CommandManager() {

**protected** Command createCommand() {

**return** asyncCommand();

}

}

}

##### **Further information about how Java-based configuration works internally**

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** ClientService clientService1() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientService clientService2() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientDao clientDao() {

**return** **new** ClientDaoImpl();

}

}

clientDao() has been called once in clientService1() and once in clientService2(). Since this method creates a new instance of ClientDaoImpl and returns it, you would normally expect having 2 instances (one for each service). That definitely would be problematic: in Spring, instantiated beans have a singleton scope by default. This is where the magic comes in: All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

Note that as of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under org.springframework.cglib and included directly within the spring-core JAR.

here are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time, in particular that configuration classes must not be final. However, as of 4.3, any constructors are allowed on configuration classes, including the use of @Autowired or a single non-default constructor declaration for default injection.

If you prefer to avoid any CGLIB-imposed limitations, consider declaring your @Bean methods on non-@Configuration classes, e.g. on plain @Component classes instead. Cross-method calls between @Bean methods won’t get intercepted then, so you’ll have to exclusively rely on dependency injection at the constructor or method level there.

@Configuration

**public** **class** **ServiceConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

@Bean

**public** AccountRepository accountRepository(DataSource dataSource) {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

There is another way to achieve the same result. Remember that @Configuration classes are ultimately just another bean in the container: This means that they can take advantage of @Autowired and @Value injection etc just like any other bean!

Make sure that the dependencies you inject that way are of the simplest kind only. @Configuration classes are processed quite early during the initialization of the context and forcing a dependency to be injected this way may lead to unexpected early initialization. Whenever possible, resort to parameter-based injection as in the example above.

**Also, be particularly careful with BeanPostProcessor and BeanFactoryPostProcessor definitions via @Bean. Those should usually be declared as static @Bean methods, not triggering the instantiation of their containing configuration class. Otherwise, @Autowired and @Value won’t work on the configuration class itself since it is being created as a bean instance too early.**

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** AccountRepository accountRepository;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

**private** **final** DataSource dataSource;

@Autowired

**public** RepositoryConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

**Constructor injection in @Configuration classes is only supported as of Spring Framework 4.3. Note also that there is no need to specify @Autowired if the target bean defines only one constructor; in the example above, @Autowired is not necessary on the RepositoryConfig constructor**.

***Fully-qualifying imported beans for ease of navigation***

In the scenario above, using @Autowired works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat ambiguous. For example, as a developer looking at ServiceConfig, how do you know exactly where the @Autowired AccountRepository bean is declared? It’s not explicit in the code, and this may be just fine. Remember that the [Spring Tool Suite](https://spring.io/tools/sts) provides tooling that can render graphs showing how everything is wired up - that may be all you need. Also, your Java IDE can easily find all declarations and uses of the AccountRepository type, and will quickly show you the location of @Bean methods that return that type.

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one @Configuration class to another, consider autowiring the configuration classes themselves:

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

*// navigate 'through' the config class to the @Bean method!*

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

In the situation above, it is completely explicit where AccountRepository is defined. However, ServiceConfig is now tightly coupled to RepositoryConfig; that’s the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based @Configuration classes. Consider the following:

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

@Configuration

**public** **interface** **RepositoryConfig** {

@Bean

AccountRepository accountRepository();

}

@Configuration

**public** **class** **DefaultRepositoryConfig** **implements** RepositoryConfig {

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(...);

}

}

@Configuration

@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) *// import the concrete config!*

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return DataSource*

}

}

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

Now ServiceConfig is loosely coupled with respect to the concrete DefaultRepositoryConfig, and built-in IDE tooling is still useful: it will be easy for the developer to get a type hierarchy of RepositoryConfig implementations. In this way, navigating @Configuration classes and their dependencies becomes no different than the usual process of navigating interface-based code.

Combining XML and JAVA

XML centric approach

*Declaring @Configuration classes as plain Spring <bean/> elements*

Remember that @Configuration classes are ultimately just bean definitions in the container. In this example, we create a @Configuration class named AppConfig and include it within system-test-config.xml as a <bean/> definition. Because<context:annotation-config/> is switched on, the container will recognize the @Configuration annotation and process the @Bean methods declared in AppConfig properly.

@Configuration

**public** **class** **AppConfig** {

@Autowired

**private** DataSource dataSource;

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

@Bean

**public** TransferService transferService() {

**return** **new** TransferService(accountRepository());

}

}

**system-test-config.xml**:

<beans>

*<!-- enable processing of annotations such as @Autowired and @Configuration -->*

<context:annotation-config/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

**jdbc.properties**:

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

|  |
| --- |
|  |

*Using <context:component-scan/> to pick up @Configuration classes*

Because @Configuration is meta-annotated with @Component, @Configuration-annotated classes are automatically candidates for component scanning. Using the same scenario as above, we can redefine system-test-config.xml to take advantage of component-scanning. Note that in this case, we don’t need to explicitly declare <context:annotation-config/>, because <context:component-scan/> enables the same functionality.

**system-test-config.xml**:

<beans>

*<!-- picks up and registers AppConfig as a bean definition -->*

<context:component-scan base-package="com.acme"/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

JAVA CENTRIC APPROACH

In applications where @Configuration classes are the primary mechanism for configuring the container, it will still likely be necessary to use at least some XML. In these scenarios, simply use @ImportResource and define only as much XML as is needed. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum.

@Configuration

@ImportResource("classpath:/com/acme/properties-config.xml")

**public** **class** **AppConfig** {

@Value("${jdbc.url}")

**private** String url;

@Value("${jdbc.username}")

**private** String username;

@Value("${jdbc.password}")

**private** String password;

@Bean

**public** DataSource dataSource() {

**return** **new** DriverManagerDataSource(url, username, password);

}

}

properties-config.xml

<beans>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

</beans>

jdbc.properties

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

**public** **static** **void** main(String**[]** args) {

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

Conditional Bean definition.

package com.javapapers.spring4.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Conditional;

import org.springframework.context.annotation.Configuration;

import com.javapapers.spring4.condition.DevDataSourceCondition;

import com.javapapers.spring4.condition.ProdDataSourceCondition;

import com.javapapers.spring4.util.DataSource;

import com.javapapers.spring4.util.DevDatabaseUtil;

import com.javapapers.spring4.util.ProductionDatabaseUtil;

@Configuration

public class EmployeeDataSourceConfig {

@Bean(name="dataSource")

@Conditional(**value=DevDataSourceCondition.class**)

public DataSource getDevDataSource() {

return new DevDatabaseUtil();

}

@Bean(name="dataSource")

@Conditional(ProdDataSourceCondition.class)

public DataSource getProdDataSource() {

return new ProductionDatabaseUtil();

}

}

Creating new condition by implementing Condition interface

package com.javapapers.spring4.condition;

import org.springframework.context.annotation.Condition;

import org.springframework.context.annotation.ConditionContext;

import org.springframework.core.type.AnnotatedTypeMetadata;

public class **DevDataSourceCondition** implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

String dbname = context.getEnvironment().getProperty("database.name");

return dbname.equalsIgnoreCase("dev");

}

}

Similarly, create for production environment condition class.

Meta aanotation @Production using @Profile

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

**@Profile("production")**

**public** @interface Production {

}

With @Profile on @Bean methods, a special scenario may apply: In the case of overloaded @Bean methods of the same Java method name (analogous to constructor overloading), an @Profile condition needs to be consistently declared on all overloaded methods. If the conditions are inconsistent, only the condition on the first declaration among the overloaded methods will matter. @Profile can therefore not be used to select an overloaded method with a particular argument signature over another; resolution between all factory methods for the same bean follows Spring’s constructor resolution algorithm at creation time.

If you would like to define alternative beans with different profile conditions, use distinct Java method names pointing to the same bean name via the @Bean name attribute, as indicated in the example above. If the argument signatures are all the same (e.g. all of the variants have no-arg factory methods), this is the only way to represent such an arrangement in a valid Java class in the first place (since there can only be one method of a particular name and argument signature).

PropertySource

A PropertySource is a simple abstraction over any source of key-value pairs, and Spring’s [StandardEnvironment](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/core/env/StandardEnvironment.html) is configured with two PropertySource objects — one representing the set of JVM system properties (a la System.getProperties()) and one representing the set of system environment variables (a la System.getenv()).

These default property sources are present for StandardEnvironment, for use in standalone applications. [StandardServletEnvironment](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/web/context/support/StandardServletEnvironment.html) is populated with additional default property sources including servlet config and servlet context parameters. It can optionally enable a [JndiPropertySource](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/jndi/JndiPropertySource.html). See the javadocs for details.

The search performed is hierarchical. By default, system properties have precedence over environment variables, so if the foo property happens to be set in both places during a call to env.getProperty("foo"), the system property value will 'win' and be returned preferentially over the environment variable. Note that property values will not get merged but rather completely overridden by a preceding entry.

For a common StandardServletEnvironment, the full hierarchy looks as follows, with the highest-precedence entries at the top:

* ServletConfig parameters (if applicable, e.g. in case of a DispatcherServlet context)
* ServletContext parameters (web.xml context-param entries)
* JNDI environment variables ("java:comp/env/" entries)
* JVM system properties ("-D" command-line arguments)
* JVM system environment (operating system environment variables)

Most importantly, the entire mechanism is configurable. Perhaps you have a custom source of properties that you’d like to integrate into this search. No problem — simply implement and instantiate your own PropertySource and add it to the set of PropertySources for the current Environment:

ConfigurableApplicationContext ctx = **new** GenericApplicationContext();

MutablePropertySources sources = ctx.getEnvironment().getPropertySources();

sources.addFirst(**new** MyPropertySource());

In the code above, MyPropertySource has been added with highest precedence in the search. If it contains a foo property, it will be detected and returned ahead of any foo property in any other PropertySource. The [MutablePropertySources](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/core/env/MutablePropertySources.html) API exposes a number of methods that allow for precise manipulation of the set of property sources.

The [@PropertySource](https://docs.spring.io/spring-framework/docs/5.0.14.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html) annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment.

Given a file "app.properties" containing the key/value pair testbean.name=myTestBean, the following @Configuration class uses @PropertySource in such a way that a call to testBean.getName() will return "myTestBean".

@Configuration

**@PropertySource("classpath:/com/myco/app.properties")**

**public** **class** **AppConfig** {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty("testbean.name"));

**return** testBean;

}

}

Any ${…​} placeholders present in a @PropertySource resource location will be resolved against the set of property sources already registered against the environment. For example:

@Configuration

@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")

**public** **class** **AppConfig** {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty("testbean.name"));

**return** testBean;

}

}

Assuming that "my.placeholder" is present in one of the property sources already registered, e.g. system properties or environment variables, the placeholder will be resolved to the corresponding value. If not, then "default/path" will be used as a default. If no default is specified and a property cannot be resolved, an IllegalArgumentException will be thrown.

The @PropertySource annotation is repeatable according to Java 8 conventions. However, all such @PropertySource annotations need to be declared at the same level: either directly on the configuration class or as meta-annotations within the same custom annotation. Mixing of direct annotations and meta-annotations is not recommended since direct annotations will effectively override meta-annotations.

#### BeanFactory or ApplicationContext

For many extended container features such as annotation processing and AOP proxying, the [BeanPostProcessor extension point](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html#beans-factory-extension-bpp) is essential. If you use only a plain DefaultListableBeanFactory, such post-processors will not get detected and activated by default. This situation could be confusing because nothing is actually wrong with your bean configuration; it is rather the container which needs to be fully bootstrapped through additional setup in such a scenario.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

| *Table 9. Feature Matrix* | | |
| --- | --- | --- |
| **Feature** | BeanFactory | ApplicationContext |
| Bean instantiation/wiring | Yes | Yes |
| Integrated lifecycle management | No | Yes |
| Automatic BeanPostProcessorregistration | No | Yes |
| Automatic BeanFactoryPostProcessorregistration | No | Yes |
| Convenient MessageSource access (for internalization) | No | Yes |
| Built-in ApplicationEvent publication mechanism | No | Yes |

To explicitly register a bean post-processor with a DefaultListableBeanFactory, you need to programmatically call addBeanPostProcessor:

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

*// populate the factory with bean definitions*

*// now register any needed BeanPostProcessor instances*

factory.addBeanPostProcessor(**new** AutowiredAnnotationBeanPostProcessor());

factory.addBeanPostProcessor(**new** MyBeanPostProcessor());

*// now start using the factory*

To apply a BeanFactoryPostProcessor to a plain DefaultListableBeanFactory, you need to call its postProcessBeanFactorymethod:

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(**new** FileSystemResource("beans.xml"));

*// bring in some property values from a Properties file*

PropertyPlaceholderConfigurer cfg = **new** PropertyPlaceholderConfigurer();

cfg.setLocation(**new** FileSystemResource("jdbc.properties"));

*// now actually do the replacement*

cfg.postProcessBeanFactory(factory);

In both cases, the explicit registration steps are inconvenient, which is why the various ApplicationContext variants are preferred over a plain DefaultListableBeanFactory in Spring-backed applications, especially when relying on BeanFactoryPostProcessors and BeanPostProcessors for extended container functionality in a typical enterprise setup.

## **Resources**

Java’s standard java.net.URL class and standard handlers for various URL prefixes, unfortunately, are not quite adequate enough for all access to low-level resources. For example, there is no standardized URL implementation that may be used to access a resource that needs to be obtained from the classpath or relative to a ServletContext. While it is possible to register new handlers for specialized URL prefixes (similar to existing handlers for prefixes such as http:), this is generally quite complicated, and the URL interface still lacks some desirable functionality, such as a method to check for the existence of the resource being pointed to.

**public** **interface** **Resource** **extends** InputStreamSource {

**boolean** exists();

**boolean** isOpen();

URL getURL() **throws** IOException;

File getFile() **throws** IOException;

Resource createRelative(String relativePath) **throws** IOException;

String getFilename();

String getDescription();

}

**public** **interface** **InputStreamSource** {

InputStream getInputStream() **throws** IOException;

}

Some of the most important methods from the Resource interface are:

* getInputStream(): Locates and opens the resource, returning an InputStream for reading from the resource. It is expected that each invocation returns a fresh InputStream. It is the responsibility of the caller to close the stream.
* exists(): Returns a boolean indicating whether this resource actually exists in physical form.
* isOpen(): Returns a boolean indicating whether this resource represents a handle with an open stream. If true, the InputStream cannot be read multiple times and must be read once only and then closed to avoid resource leaks. Returns false for all usual resource implementations, with the exception of InputStreamResource.
* getDescription(): Returns a description for this resource, to be used for error output when working with the resource. This is often the fully qualified file name or the actual URL of the resource.

Other methods let you obtain an actual URL or File object representing the resource (if the underlying implementation is compatible and supports that functionality).

#### UrlResource

UrlResource wraps a java.net.URL and can be used to access any object that is normally accessible with a URL, such as files, an HTTP target, an FTP target, and others. All URLs have a standardized String representation, such that appropriate standardized prefixes are used to indicate one URL type from another. This includes file: for accessing filesystem paths, http: for accessing resources through the HTTP protocol, ftp: for accessing resources through FTP, and others.

A UrlResource is created by Java code by explicitly using the UrlResource constructor but is often created implicitly when you call an API method that takes a String argument meant to represent a path. For the latter case, a JavaBeans PropertyEditorultimately decides which type of Resource to create. If the path string contains well-known (to it, that is) prefix (such as classpath:), it creates an appropriate specialized Resource for that prefix. However, if it does not recognize the prefix, it assume the string is a standard URL string and creates a UrlResource.

#### 2.3.2. ClassPathResource

This class represents a resource that should be obtained from the classpath. It uses either the thread context class loader, a given class loader, or a given class for loading resources.

This Resource implementation supports resolution as java.io.File if the class path resource resides in the file system but not for classpath resources that reside in a jar and have not been expanded (by the servlet engine or whatever the environment is) to the filesystem. To address this, the various Resource implementations always support resolution as a java.net.URL.

A ClassPathResource is created by Java code by explicitly using the ClassPathResource constructor but is often created implicitly when you call an API method that takes a String argument meant to represent a path. For the latter case, a JavaBeans PropertyEditor recognizes the special prefix, classpath:, on the string path and creates a ClassPathResource in that case.

#### 2.3.3. FileSystemResource

This is a Resource implementation for java.io.File and java.nio.file.Path handles. It supports resolution as a File and as a URL.

#### 2.3.4. ServletContextResource

This is a Resource implementation for ServletContext resources that interprets relative paths within the relevant web application’s root directory.

It always supports stream access and URL access but allows java.io.File access only when the web application archive is expanded and the resource is physically on the filesystem. Whether or not it is expanded and on the filesystem or accessed directly from the JAR or somewhere else like a database (which is conceivable) is actually dependent on the Servlet container.

#### 2.3.5. InputStreamResource

An InputStreamResource is a Resource implementation for a given InputStream. It should be used only if no specific Resourceimplementation is applicable. In particular, prefer ByteArrayResource or any of the file-based Resource implementations where possible.

In contrast to other Resource implementations, this is a descriptor for an already-opened resource. Therefore, it returns truefrom isOpen(). Do not use it if you need to keep the resource descriptor somewhere or if you need to read a stream multiple times.

#### 2.3.6. ByteArrayResource

This is a Resource implementation for a given byte array. It creates a ByteArrayInputStream for the given byte array.

It is useful for loading content from any given byte array without having to resort to a single-use InputStreamResource.

### **2.4. The ResourceLoader**

The ResourceLoader interface is meant to be implemented by objects that can return (that is, load) Resource instances. The following listing shows the ResourceLoader interface definition:

**public** **interface** **ResourceLoader** {

Resource getResource(String location);

}

All application contexts implement the ResourceLoader interface. Therefore, all application contexts may be used to obtain Resource instances.

When you call getResource() on a specific application context, and the location path specified doesn’t have a specific prefix, you get back a Resource type that is appropriate to that particular application context. For example, assume the following snippet of code was executed against a ClassPathXmlApplicationContext instance:

Resource template = ctx.getResource("some/resource/path/myTemplate.txt");

| **Prefix** | **Example** | **Explanation** |
| --- | --- | --- |
| classpath: | classpath:com/myapp/config.xml | Loaded from the classpath. |
| file: | [file:///data/config.xml](file:///\\data\config.xml) | Loaded as a URL from the filesystem. See also [FileSystemResource Caveats](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources-filesystemresource-caveats). |
| http: | <https://myserver/logo.png> | Loaded as a URL. |
| (none) | /data/config.xml | Depends on the underlying ApplicationContext. |

### **The ResourceLoaderAware interface**

The ResourceLoaderAware interface is a special callback interface which identifies components that expect to be provided with a ResourceLoader reference. The following listing shows the definition of the ResourceLoaderAware interface:

**public** **interface** **ResourceLoaderAware** {

**void** setResourceLoader(ResourceLoader resourceLoader);

}

Since an ApplicationContext is a ResourceLoader, the bean could also implement the ApplicationContextAware interface and use the supplied application context directly to load resources. However, in general, it is better to use the specialized ResourceLoader interface if that is all you need. The code would be coupled only to the resource loading interface (which can be considered a utility interface) and not to the whole Spring ApplicationContext interface.

In application components, you may also rely upon autowiring of the ResourceLoader as an alternative to implementing the ResourceLoaderAware interface.

### **Resources as Dependencies**

What makes it trivial to then inject these properties is that all application contexts register and use a special JavaBeans PropertyEditor, which can convert String paths to Resource objects. So, if myBean has a template property of type Resource, it can be configured with a simple string for that resource, as the following example shows:

<bean id="myBean" class="...">

<property name="template" value="some/resource/path/myTemplate.txt"/>

</bean>

#### Constructing Application Contexts

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location paths of the resources, such as XML files that make up the definition of the context.

When such a location path does not have a prefix, the specific Resource type built from that path and used to load the bean definitions depends on and is appropriate to the specific application context. For example, consider the following example, which creates a ClassPathXmlApplicationContext:

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("conf/appContext.xml");

The bean definitions are loaded from the classpath, because a ClassPathResource is used. However, consider the following example, which creates a FileSystemXmlApplicationContext:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/appContext.xml");

Now the bean definition is loaded from a filesystem location (in this case, relative to the current working directory).

Note that the use of the special classpath prefix or a standard URL prefix on the location path overrides the default type of Resource created to load the definition. Consider the following example:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("classpath:conf/appContext.xml");

Using FileSystemXmlApplicationContext loads the bean definitions from the classpath. However, it is still aFileSystemXmlApplicationContext. If it is subsequently used as a ResourceLoader, any unprefixed paths are still treated as filesystem paths.

##### **Constructing ClassPathXmlApplicationContext Instances — Shortcuts**

The ClassPathXmlApplicationContext exposes a number of constructors to enable convenient instantiation. The basic idea is that you can supply merely a string array that contains only the filenames of the XML files themselves (without the leading path information) and also supplies a Class. The ClassPathXmlApplicationContext then derives the path information from the supplied class.

Consider the following directory layout:

com/

foo/

services.xml

daos.xml

MessengerService.class

The following example shows how a ClassPathXmlApplicationContext instance composed of the beans defined in files named services.xml and daos.xml (which are on the classpath) can be instantiated:

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(

**new** String**[]** {"services.xml", "daos.xml"}, MessengerService.class);

#### 2.7.2. Wildcards in Application Context Constructor Resource Paths

The resource paths in application context constructor values may be simple paths (as shown earlier), each of which has a one-to-one mapping to a target Resource or, alternately, may contain the special "classpath\*:" prefix or internal Ant-style regular expressions (matched by using Spring’s PathMatcher utility). Both of the latter are effectively wildcards.

One use for this mechanism is when you need to do component-style application assembly. All components can 'publish' context definition fragments to a well-known location path, and, when the final application context is created using the same path prefixed with classpath\*:, all component fragments are automatically picked up.

Note that this wildcarding is specific to the use of resource paths in application context constructors (or when you use the PathMatcher utility class hierarchy directly) and is resolved at construction time. It has nothing to do with the Resource type itself. You cannot use the classpath\*: prefix to construct an actual Resource, as a resource points to just one resource at a time.

##### **Ant-style Patterns**

Path locations can contain Ant-style patterns, as the following example shows:

/WEB-INF/\*-context.xml

com/mycompany/\*\*/applicationContext.xml

file:C:/some/path/\*-context.xml

classpath:com/mycompany/\*\*/applicationContext.xml

When the path location contains an Ant-style pattern, the resolver follows a more complex procedure to try to resolve the wildcard. It produces a Resource for the path up to the last non-wildcard segment and obtains a URL from it. If this URL is not a jar: URL or container-specific variant (such as zip: in WebLogic, wsjar in WebSphere, and so on), a java.io.File is obtained from it and used to resolve the wildcard by traversing the filesystem. In the case of a jar URL, the resolver either gets a java.net.JarURLConnection from it or manually parses the jar URL and then traverses the contents of the jar file to resolve the wildcards.

###### **Implications on Portability**

If the specified path is already a file URL (either implicitly because the base ResourceLoader is a filesystem one or explicitly), wildcarding is guaranteed to work in a completely portable fashion.

If the specified path is a classpath location, the resolver must obtain the last non-wildcard path segment URL by making a Classloader.getResource() call. Since this is just a node of the path (not the file at the end), it is actually undefined (in theClassLoader javadoc) exactly what sort of a URL is returned in this case. In practice, it is always a java.io.File representing the directory (where the classpath resource resolves to a filesystem location) or a jar URL of some sort (where the classpath resource resolves to a jar location). Still, there is a portability concern on this operation.

If a jar URL is obtained for the last non-wildcard segment, the resolver must be able to get a java.net.JarURLConnection from it or manually parse the jar URL, to be able to walk the contents of the jar and resolve the wildcard. This does work in most environments but fails in others, and we strongly recommend that the wildcard resolution of resources coming from jars be thoroughly tested in your specific environment before you rely on it.

##### **The classpath\*: Prefix**

When constructing an XML-based application context, a location string may use the special classpath\*: prefix, as the following example shows:

ApplicationContext ctx =

**new** ClassPathXmlApplicationContext("classpath\*:conf/appContext.xml");

This special prefix specifies that all classpath resources that match the given name must be obtained (internally, this essentially happens through a call to ClassLoader.getResources(…​)) and then merged to form the final application context definition.

|  |  |
| --- | --- |
|  | The wildcard classpath relies on the getResources() method of the underlying classloader. As most application servers nowadays supply their own classloader implementation, the behavior might differ, especially when dealing with jar files. A simple test to check if classpath\* works is to use the classloader to load a file from within a jar on the classpath: getClass().getClassLoader().getResources("<someFileInsideTheJar>"). Try this test with files that have the same name but are placed inside two different locations. In case an inappropriate result is returned, check the application server documentation for settings that might affect the classloader behavior. |

You can also combine the classpath\*: prefix with a PathMatcher pattern in the rest of the location path (for example, classpath\*:META-INF/\*-beans.xml). In this case, the resolution strategy is fairly simple: A ClassLoader.getResources() call is used on the last non-wildcard path segment to get all the matching resources in the class loader hierarchy and then, off each resource, the same PathMatcher resolution strategy described earlier is used for the wildcard subpath.

##### **Other Notes Relating to Wildcards**

Note that classpath\*:, when combined with Ant-style patterns, only works reliably with at least one root directory before the pattern starts, unless the actual target files reside in the file system. This means that a pattern such as classpath\*:\*.xml might not retrieve files from the root of jar files but rather only from the root of expanded directories.

Spring’s ability to retrieve classpath entries originates from the JDK’s ClassLoader.getResources() method, which only returns file system locations for an empty string (indicating potential roots to search). Spring evaluates URLClassLoader runtime configuration and the java.class.path manifest in jar files as well, but this is not guaranteed to lead to portable behavior.

|  |  |
| --- | --- |
|  | The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, do not activate the files-only switch of the JAR task. Also, classpath directories may not get exposed based on security policies in some environments — for example, stand-alone applications on JDK 1.7.0\_45 and higher (which requires 'Trusted-Library' to be set up in your manifests. See<https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>).  On JDK 9’s module path (Jigsaw), Spring’s classpath scanning generally works as expected. Putting resources into a dedicated directory is highly recommendable here as well, avoiding the aforementioned portability problems with searching the jar file root level. |

Ant-style patterns with classpath: resources are not guaranteed to find matching resources if the root package to search is available in multiple class path locations. Consider the following example of a resource location:

com/mycompany/package1/service-context.xml

Now consider an Ant-style path that someone might use to try to find that file:

classpath:com/mycompany/\*\*/service-context.xml

Such a resource may be in only one location, but when a path such as the preceding example is used to try to resolve it, the resolver works off the (first) URL returned by getResource("com/mycompany");. If this base package node exists in multiple classloader locations, the actual end resource may not be there. Therefore, in such a case you should prefer using classpath\*:with the same Ant-style pattern, which searches all class path locations that contain the root package.

#### 2.7.3. FileSystemResource Caveats

A FileSystemResource that is not attached to a FileSystemApplicationContext (that is, when a FileSystemApplicationContextis not the actual ResourceLoader) treats absolute and relative paths as you would expect. Relative paths are relative to the current working directory, while absolute paths are relative to the root of the filesystem.

For backwards compatibility (historical) reasons however, this changes when the FileSystemApplicationContext is the ResourceLoader. The FileSystemApplicationContext forces all attached FileSystemResource instances to treat all location paths as relative, whether they start with a leading slash or not. In practice, this means the following examples are equivalent:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/context.xml");

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("/conf/context.xml");

The following examples are also equivalent (even though it would make sense for them to be different, as one case is relative and the other absolute):

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("some/resource/path/myTemplate.txt");

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("/some/resource/path/myTemplate.txt");

In practice, if you need true absolute filesystem paths, you should avoid using absolute paths with FileSystemResource or FileSystemXmlApplicationContext and force the use of a UrlResource by using the file: URL prefix. The following examples show how to do so:

*// actual context type doesn't matter, the Resource will always be UrlResource*

ctx.getResource("file:///some/resource/path/myTemplate.txt");

*// force this FileSystemXmlApplicationContext to load its definition via a UrlResource*

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("file:///conf/context.xml");

## **4. Spring Expression Language (SpEL)**

While there are several other Java expression languages available — OGNL, MVEL, and JBoss EL, to name a few — the Spring Expression Language was created to provide the Spring community with a single well supported expression language that can be used across all the products in the Spring portfolio. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the Eclipse-based Spring Tool Suite. That said, SpEL is based on a technology-agnostic API that lets other expression language implementations be integrated, should the need arise.

SpEL serves as the foundation for expression evaluation within the Spring portfolio, it is not directly tied to Spring and can be used independently. To be self contained, many of the examples in this chapter use SpEL as if it were an independent expression language. This requires creating a few bootstrapping infrastructure classes, such as the parser. Most Spring users need not deal with this infrastructure and can, instead, author only expression strings for evaluation. An example of this typical use is the integration of SpEL into creating XML or annotation-based bean definitions, as shown in [Expression support for defining bean definitions](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions-beandef).

The expression language supports the following functionality:

* Literal expressions
* Boolean and relational operators
* Regular expressions
* Class expressions
* Accessing properties, arrays, lists, and maps
* Method invocation
* Relational operators
* Assignment
* Calling constructors
* Bean references
* Array construction
* Inline lists
* Inline maps
* Ternary operator
* Variables
* User-defined functions
* Collection projection
* Collection selection
* Templated expressions

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("'Hello World'");

String message = (String) exp.getValue();

|  |
| --- |
|  |

The SpEL classes and interfaces you are most likely to use are located in the org.springframework.expression package and its sub-packages, such as spel.support.