Classpath Scanning and Managed Components

Most examples in this chapter use XML to specify the configuration metadata that produces each `BeanDefinition` within the Spring container.

only the dependency injection. This section describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, `@Component`), AspectJ type expressions, or your own custom filter criteria to select which classes have bean definitions registered with the container.

NOTE

You can define beans using Java rather than using XML files. Take a look at the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations for examples of how to use these features.

`@Component` and Further Stereotype Annotations

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

Spring provides further stereotype annotations: `@Component`, `@Service`, and `@Controller`. `@Component` is a generic stereotype for any Spring-managed component. `@Repository`, `@Service`, and `@Controller` are specializations of `@Component` for more specific use cases (in the persistence, service, and presentation layers, respectively). Therefore, you can annotate your component classes with `@Component`, but, by annotating them with `@Repository`, `@Service`, or `@Controller` instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. `@Repository`, `@Service`, and `@Controller` may also carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using `@Component` or `@Service` for your service layer, `@Service` is clearly the better choice. Similarly, as stated earlier, `@Repository` is already supported as a marker for automatic exception translation in your persistence layer.

Using Meta-annotations and Composed Annotations

Many of the annotations provided by Spring can be used as meta-annotations in your own code. A meta-annotation is an annotation that can be applied to another annotation. For example, the `@Service` annotation mentioned earlier is meta-annotated with `@Component`, as the following example shows:

Java Kotlin

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component (1)
public @interface Service {

    // ...
}
```

(1) The `@Component` causes `@Service` to be treated in the same way as `@Component`.

You can also combine meta-annotations to create “composed annotations”. For example, the `@RestController` annotation from Spring MVC is composed of `@Controller` and `@ResponseBody`.

In addition, composed annotations can optionally redeclare attributes from meta-annotations to allow 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 hard codes the scope name to `session` but still allows customization of the `proxyMode`. The following listing shows the definition of the `SessionScope` annotation:

Java Kotlin

```java
@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;

}
```

You can then use `@SessionScope` without declaring the `proxyMode` as follows:

Java Kotlin

```java
@Service
@SessionScope
public class SessionScopedService {
    // ...
}
```

You can also override the value for the `proxyMode`, as the following example shows:

Java Kotlin

```java
@Service
@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)
public class SessionScopedUserService implements UserService {
    // ...
}
```

For further details, see the Spring Annotation Programming Model wiki page.

Automatically Detecting Classes and Registering Bean Definitions

Spring can automatically detect stereotyped classes and register corresponding `BeanDefinition` instances with the `ApplicationContext`. For example, the following two classes are eligible for such autodetection:

Java Kotlin

```java
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

Java Kotlin

```java
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

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

Java Kotlin

```java
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    // ...
}
```

NOTE

For brevity, the preceding example could have used the `value` attribute of the annotation (that is, `@ComponentScan("org.example")`).

The following alternative uses XML:

```xml
<?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:component-scan base-package="org.example"/>

</beans>
```

TIP

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>`.

NOTE

The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do not activate the files-only switch of the JAR task. Also, classpath directories may not be exposed based on security policies in some environments — for example, standalone apps on JDK 1.7.0_45 and higher (which requires 'Trusted-Library' setup in your manifests — see stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources).

On the module path (Java Module System), Spring's classpath scanning generally works as expected. However, make sure that your component classes are exported in your `module-info` descriptors. If you expect Spring to invoke non-public members of your classes, make sure that they are 'opened' (that is, that they use an `opens` declaration instead of an `exports` declaration in your `module-info` descriptor).

Furthermore, the `AutowiredAnnotationBeanPostProcessor` and `CommonAnnotationBeanPostProcessor` are both implicitly included 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.

NOTE

You can disable the registration of `AutowiredAnnotationBeanPostProcessor` and `CommonAnnotationBeanPostProcessor` 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`, `@Configuration`, or a custom annotation that itself is annotated with `@Component` are the only detected candidate components. However, you can modify and extend this behavior by applying custom filters. Add them as `includeFilters` or `excludeFilters` attributes of the `@ComponentScan` annotation (or as `<context:include-filter />` or `<context:exclude-filter />` child elements of the `<context:component-scan>` element in XML configuration). Each filter element requires the `type` and `expression` attributes. The following table describes the filtering options:

Table 1. Filter Types

Filter Type	Example Expression	Description
annotation (default)	`org.example.SomeAnnotation`	An annotation to be *present* or *meta-present* at the type level in target components.
assignable	`org.example.SomeClass`	A class (or interface) that the target components are assignable to (extend or 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:

Java Kotlin

```java
@Configuration
@ComponentScan(basePackages = "org.example",
        includeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Stub.*Repository"),
        excludeFilters = @Filter(Repository.class))
public class AppConfig {
    // ...
}
```

The following listing shows the equivalent XML:

```xml
<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>
```

NOTE

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

Defining Bean Metadata within Components

Spring components can also contribute bean definition metadata to the container. You can do this with the same `@Bean` annotation used to define bean metadata within `@Configuration` annotated classes. The following example shows how to do so:

Java Kotlin

```java
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }
}
```

The preceding class is a Spring component that has application-specific code in its `doWork()` method. However, it also contributes a bean definition that has a factory method referring to the method `publicInstance()`. The `@Bean` annotation identifies the factory method and other bean definition properties, such as a qualifier value through the `@Qualifier` annotation. Other method-level annotations that can be specified are `@Scope`, `@Lazy`, and custom qualifier annotations.

TIP

In addition to its role for component initialization, you can also place the `@Lazy` annotation on injection points marked with `@Autowired` or `@Inject`. In this context, it leads to the injection of a lazy-resolution proxy. However, such a proxy approach is rather limited. For sophisticated lazy interactions, in particular in combination with optional dependencies, we recommend `ObjectProvider<MyTargetBean>` instead.

Autowired fields and methods are supported, as previously discussed, with additional support for autowiring of `@Bean` methods. The following example shows how to do so:

Java Kotlin

```java
@Component
public class FactoryMethodComponent {

    private static int i;

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    // 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.

As of Spring Framework 4.3, you may also declare a factory method parameter of type `InjectionPoint` (or its more specific subclass: `DependencyDescriptor`) to access the requesting injection point that triggers the creation of the current bean. Note that this applies only to the actual creation of bean instances, not to the injection of existing instances. As a consequence, this feature makes most sense for beans of prototype scope. For other scopes, the factory method only ever sees the injection point that triggered the creation of a new bean instance in the given scope (for example, the dependency that triggered the creation of a lazy singleton bean). You can use the provided injection point metadata with semantic care in such scenarios. The following example shows how to use `InjectionPoint`:

Java Kotlin

JAVA

```
@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember());
    }
}
```

The `@Bean` methods in a regular Spring component are processed differently than their counterparts inside a Spring `@Configuration` class. The difference is that `@Component` classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within `@Bean` methods in `@Configuration` classes creates bean metadata references to collaborating objects. Such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans, even when referring to other beans through programmatic calls to `@Bean` methods. In contrast, invoking a method or field in a `@Bean` method within a plain `@Component` class has standard Java semantics, with no special CGLIB processing or other constraints applying.

NOTE

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 (for example, of type `BeanFactoryPostProcessor` or `BeanPostProcessor`), since such beans get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.

Calls to static `@Bean` methods never get intercepted by the container, not even within `@Configuration` classes (as described earlier in this section), due to technical limitations: CGLIB subclassing can override only non-static methods. As a consequence, a direct call to another `@Bean` method has 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 can 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 — that is, they must not be declared as `private` or `final`.

`@Bean` methods are also 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, 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 is picked at construction time, analogous to how the container selects between multiple `@Autowired` constructors.

Naming Autodetected Components

When a component is autodetected as part of the scanning process, its bean name is generated by the `BeanNameGenerator` strategy known to that scanner.

By default, the `AnnotationBeanNameGenerator` is used. For Spring stereotype annotations, if you supply a name via the annotation's `value` attribute that name will be used as the name in the corresponding bean definition. This convention also applies when the following JSR-250 and JSR-330 annotations are used instead of Spring stereotype annotations: `@jakarta.annotation.ManagedBean`, `@javax.annotation.ManagedBean`, `@jakarta.inject.Named`, and `@javax.inject.Named`.

As of Spring Framework 6.1, the name of the annotation attribute that is used to specify the bean name is no longer required to be `value`. Custom stereotype annotations can declare an attribute with a different name (such as `name`) and annotate that attribute with `@AliasFor(annotation = Component.class, attribute = "value")`. See the source code declaration of `ControllerAdvice#name()` for a concrete example.

WARNING

As of Spring Framework 6.1, support for convention-based stereotype names is deprecated and will be removed in a future version of the framework. Consequently, custom stereotype annotations must use `@AliasFor` to declare an explicit alias for the `value` attribute in `@Component`. See the source code declaration of `Repository#value()` and `ControllerAdvice#name()` for concrete examples.

If an explicit bean name cannot be derived from such an annotation or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following component classes were detected, the names would be `myMovieLister` and `movieFinderImpl`.

Java Kotlin

JAVA

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

Java Kotlin

JAVA

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

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` interface, and be sure to include a default no-arg constructor. Then, provide the fully qualified class name when configuring the scanner, as the following example annotation and bean definition show.

TIP

If you run into naming conflicts due to multiple autodetected components having the same non-qualified class name (i.e., classes with identical names but residing in different packages), you may need to configure a `BeanNameGenerator` that defaults to the fully qualified class name for the generated

bean name. The `FullyQualifiedAnnotationBeanNameGenerator` located in package `org.springframework.context.annotation` can be used for such purposes.

Java Kotlin

```java
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)
public class AppConfig {
    // ...
}
```

```xml
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

Providing a Scope for Autodetected Components

As with Spring-managed components in general, the default and most common scope for autodetected components is `singleton`. However, sometimes you need a different scope that can be specified by the `@Scope` annotation. You can provide the name of the scope within the annotation, as the following example shows:

Java Kotlin

```java
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

NOTE

`@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.

For details on web-specific scopes such as “request” or “session” in a Spring context, see Request, Session, Application, and WebSocket Scopes. As with the pre-built annotations for those scopes, you may also compose your own scoping annotations by using Spring’s meta-annotation approach: for example, a custom annotation meta-annotated with `@Scope("prototype")`, possibly also declaring a custom scoped-proxy mode.

NOTE

To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, you can implement the `ScopeMetadataResolver` interface. Be sure to include a default no-arg constructor. Then you can provide the fully qualified class name when configuring the scanner, as the following example of both an annotation and a bean definition shows:

Java Kotlin

```java
@Configuration
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)
public class AppConfig {
    // ...
}
```

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

When using certain non-singleton scopes, it may be necessary to generate proxies for the scoped objects. The reasoning is described in Scoped Beans as Dependencies. For this purpose, a scoped-proxy attribute is available on the component-scan element. The three possible values are: `no`, `interfaces`, and `targetClass`. For example, the following configuration results in standard JDK dynamic proxies:

Java Kotlin

```java
@Configuration
@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)
public class AppConfig {
    // ...
}
```

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

Providing Qualifier Metadata with Annotations

The `@Qualifier` annotation is discussed in Fine-tuning Annotation-based Autowiring with Qualifiers. The examples in that section demonstrate the use of the `@Qualifier` annotation and custom qualifier annotations to provide fine-grained control when you resolve autowire candidates. Because those examples were based on XML bean definitions, the qualifier metadata was provided on the candidate bean definitions by using the `qualifier` or `meta` child elements of the `bean` element in the XML. When relying upon classpath scanning for auto-detection of components, you can provide the qualifier metadata with type-level annotations on the candidate class. The following three examples demonstrate this technique:

Java Kotlin

```java
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

Java Kotlin

```java
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
```

```
    // ...
}
```

Java Kotlin

```java
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

NOTE

As with most annotation-based alternatives, keep in mind that the annotation metadata is bound to the class definition itself, while the use of XML allows for multiple beans of the same type to provide variations in their qualifier metadata, because that metadata is provided per-instance rather than per-class.