org.springframework.context.annotation

Annotation Type ComponentScan

```
@Retention(value=RUNTIME)
 @Target(value=TYPE)
 @Documented
 @Repeatable(value=ComponentScans.class)
public @interface ComponentScan
```

Configures component scanning directives for use with `@Configuration` classes. Provides support parallel with Spring XML's `<context:component-scan>` element.

Either `basePackageClasses()` or `basePackages()` (or its alias `value()`) may be specified to define specific packages to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

Note that the `<context:component-scan>` element has an `annotation-config` attribute; however, this annotation does not. This is because in almost all cases when using `@ComponentScan`, default annotation config processing (e.g. processing `@Autowired` and friends) is assumed. Furthermore, when using `AnnotationConfigApplicationContext`, annotation config processors are always registered, meaning that any attempt to disable them at the `@ComponentScan` level would be ignored.

See `@Configuration`'s Javadoc for usage examples.

Since:

`3.1`

Author:

`Chris Beams, Juergen Hoeller, Sam Brannen`

See Also:

`Configuration`

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
`Class<?>[]`	`basePackageClasses` Type-safe alternative to `basePackages()` for specifying the packages to scan for annotated components.
`String[]`	`basePackages` Base packages to scan for annotated components.
`ComponentScan.Filter[]`	`excludeFilters`

	Specifies which types are not eligible for component scanning.
`ComponentScan.Filter[]`	`includeFilters` Specifies which types are eligible for component scanning.
`boolean`	`lazyInit` Specify whether scanned beans should be registered for lazy initialization.
`Class<? extends BeanNameGenerator>`	`nameGenerator` The `BeanNameGenerator` class to be used for naming detected components within the Spring container.
`String`	`resourcePattern` Controls the class files eligible for component detection.
`ScopedProxyMode`	`scopedProxy` Indicates whether proxies should be generated for detected components, which may be necessary when using scopes in a proxy-style fashion.
`Class<? extends ScopeMetadataResolver>`	`scopeResolver` The `ScopeMetadataResolver` to be used for resolving the scope of detected components.
`boolean`	`useDefaultFilters` Indicates whether automatic detection of classes annotated with `@Component` `@Repository`, `@Service`, or `@Controller` should be enabled.
`String[]`	`value` Alias for `basePackages()`.

Element Detail

value

```
@AliasFor(value="basePackages")
public abstract String[] value
```

Alias for `basePackages()`.

Allows for more concise annotation declarations if no other attributes are needed — for example, `@ComponentScan("org.my.pkg")` instead of `@ComponentScan(basePackages = "org.my.pkg")`.

Default:

`{}`

basePackages

```
@AliasFor(value="value")
public abstract String[] basePackages
```

Base packages to scan for annotated components.

`value()` is an alias for (and mutually exclusive with) this attribute.

Use `basePackageClasses()` for a type-safe alternative to String-based package names.

Default:

`{}`

basePackageClasses

```
public abstract Class<?>[] basePackageClasses
```

Type-safe alternative to `basePackages()` for specifying the packages to scan for annotated components. The package of each class specified will be scanned.

Consider creating a special no-op marker class or interface in each package that serves no purpose other than being referenced by this attribute.

Default:

`{}`

nameGenerator

```
public abstract Class<? extends BeanNameGenerator> nameGenerator
```

The `BeanNameGenerator` class to be used for naming detected components within the Spring container.

The default value of the `BeanNameGenerator` interface itself indicates that the scanner used to process this `@ComponentScan` annotation should use its inherited bean name generator, e.g. the default `AnnotationBeanNameGenerator` or any custom instance supplied to the application context at bootstrap time.

See Also:

`AnnotationConfigApplicationContext.setBeanNameGenerator(BeanNameGenerator)`

Default:

`org.springframework.beans.factory.support.BeanNameGenerator.class`

scopeResolver

```
public abstract Class<? extends ScopeMetadataResolver> scopeResolver
```

The `ScopeMetadataResolver` to be used for resolving the scope of detected components.

Default:

`org.springframework.context.annotation.AnnotationScopeMetadataResolver.class`

scopedProxy

```
public abstract ScopedProxyMode scopedProxy
```

Indicates whether proxies should be generated for detected components, which may be necessary when using scopes in a proxy-style fashion.

The default is defer to the default behavior of the component scanner used to execute the actual scan.

Note that setting this attribute overrides any value set for `scopeResolver()`.

See Also:

`ClassPathBeanDefinitionScanner.setScopedProxyMode(ScopedProxyMode)`

Default:

`org.springframework.context.annotation.ScopedProxyMode.DEFAULT`

resourcePattern

```
public abstract String resourcePattern
```

Controls the class files eligible for component detection.

Consider use of `includeFilters()` and `excludeFilters()` for a more flexible approach.

Default:

`"**/*.class"`

useDefaultFilters

```
public abstract boolean useDefaultFilters
```

Indicates whether automatic detection of classes annotated with `@Component` `@Repository`, `@Service`, or `@Controller` should be enabled.

Default:

`true`

includeFilters

```
public abstract ComponentScan.Filter[] includeFilters
```

Specifies which types are eligible for component scanning.

Further narrows the set of candidate components from everything in `basePackages()` to everything in the base packages that matches the given filter or filters.

Note that these filters will be applied in addition to the default filters, if specified. Any type under the specified base packages which matches a given filter will be included, even if it does not match the default filters (i.e. is not annotated with `@Component`).

See Also:

`resourcePattern()`, `useDefaultFilters()`

Default:

`{}`

excludeFilters

```
public abstract ComponentScan.Filter[] excludeFilters
```

Specifies which types are not eligible for component scanning.

See Also:

`resourcePattern()`

Default:

`{}`

lazyInit

```
public abstract boolean lazyInit
```

Specify whether scanned beans should be registered for lazy initialization.

Default is `false`; switch this to `true` when desired.

Since:

4.1

Default:

`false`

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }

}
```

> **Note**
>
> For details about the effects of combining various lifecycle mechanisms, see the section called “Combining lifecycle mechanisms”.

7.10 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. The previous section (Section 7.9, “Annotation-based container configuration”) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file, while the annotations only drive 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 will have bean definitions registered with the container.

> **Note**
>
> Starting with Spring 3.0, many features provided by the Spring JavaConfig project are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files. Take a look at the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations for examples of how to use these new 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 the section called “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, for example, 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. It is also possible that `@Repository`, `@Service`, and `@Controller` may 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 above, `@Repository` is already supported as a marker for automatic exception translation in your persistence layer.

Meta-annotations

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

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component // Spring will see this and treat @Service in the same way as @Component
public @interface Service {

    // ....
}
```

Meta-annotations can also be combined to create *composed annotations*. For example, the `@RestController` annotation from Spring MVC is *composed* of `@Controller` and `@ResponseBody`.

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 {
    // ...
}
```

For further details, consult the Spring Annotation Programming Model.