(/)

Spring Bean Annotations

Last modified: February 17, 2022

by baeldung (https://www.baeldung.com/author/baeldung/)

Spring (https://www.baeldung.com/category/spring/) +

Spring Annotations (https://www.baeldung.com/tag/spring-annotations/)

Spring Core Basics (https://www.baeldung.com/tag/spring-core-basics/)

This article is part of a series:

1. Overview

In this tutorial, we'll discuss the most **common Spring bean annotations** used to define different types of beans.

There are several ways to configure beans in a Spring container. Firstly, we can declare them using XML configuration. We can also declare beans using the *@Bean* annotation in a configuration class.

Finally, we can mark the class with one of the annotations from the *org.springframework.stereotype* package, and leave the rest to component scanning.

2. Component Scanning

Spring can automatically scan a package for beans if component scanning is enabled.

@ComponentScan configures which **packages to scan for classes with annotation configuration**. We can specify the base package names directly with one of the *basePackages* or *value* arguments (*value* is an alias for *basePackages*):

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
class VehicleFactoryConfig {}
```

Also, we can point to classes in the base packages with the *basePackageClasses* argument:

```
@Configuration
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Both arguments are arrays so that we can provide multiple packages for each.

If no argument is specified, the scanning happens from the same package where the *@ComponentScan* annotated class is present.

@ComponentScan leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Alternatively, we can use *@ComponentScans* to specify multiple *@ComponentScan* configurations:

```
@Configuration
@ComponentScans({
  @ComponentScan(basePackages = "com.baeldung.annotations"),
  @ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
})
class VehicleFactoryConfig {}
```

When **using XML configuration**, the configuring component scanning is just as easy:

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

3. *@Component*

@Component is a class level annotation. During the component scan, **Spring Framework automatically detects classes annotated with *@Component:***

```
@Component
class CarUtility {
    // ...
}
```

By default, the bean instances of this class have the same name as the class name with a lowercase initial. In addition, we can specify a different name using the optional *value* argument of this annotation.

Since *@Repository*, *@Service*, *@Configuration*, and *@Controller* are all meta-annotations of *@Component*, they share the same bean naming behavior. Spring also automatically picks them up during the component scanning process.

4. *@Repository*

DAO or Repository classes usually represent the database access layer in an application, and should be annotated with *@Repository:*

```
@Repository
class VehicleRepository {
    // ...
}
```

One advantage of using this annotation is that **it has automatic persistence exception translation enabled**. When using a persistence framework, such as Hibernate, native exceptions thrown within classes annotated with *@Repository* will be automatically translated into subclasses of Spring's *DataAccessExeption*.

To enable exception translation, we need to declare our own *PersistenceExceptionTranslationPostProcessor* bean:

```
@Bean
public PersistenceExceptionTranslationPostProcessor
exceptionTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
}
```

Note that in most cases, Spring does the above step automatically.

Or via XML configuration:

```
<bean class=

"org.springframework.dao.annotation.PersistenceExceptionTranslationPost
Processor"/>
```

5. *@Service*

The **business logic** of an application usually resides within the service layer, so we'll use the *@Service* annotation to indicate that a class belongs to that layer:

```
@Service
public class VehicleService {
    // ...
}
```

6. *@Controller*

@Controller is a class level annotation, which tells the Spring Framework that this class serves as a **controller in Spring MVC**:

```
@Controller
public class VehicleController {
    // ...
}
```

7. *@Configuration*

Configuration classes can **contain bean definition methods** annotated with *@Bean*:

```
@Configuration
class VehicleFactoryConfig {

    @Bean
    Engine engine() {
        return new Engine();
    }

}
```

8. Stereotype Annotations and AOP

When we use Spring stereotype annotations, it's easy to create a pointcut that targets all classes that have a particular stereotype.

For instance, suppose we want to measure the execution time of methods from the DAO layer. We'll create the following aspect (using AspectJ annotations), taking advantage of the *@Repository* stereotype:

```
@Aspect
@Component
public class PerformanceAspect {
    @Pointcut("within(@org.springframework.stereotype.Repository *)")
    public void repositoryClassMethods() {};

    @Around("repositoryClassMethods()")
    public Object measureMethodExecutionTime(ProceedingJoinPoint 
joinPoint)
      throws Throwable {
        long start = System.nanoTime();
        Object returnValue = joinPoint.proceed();
        long end = System.nanoTime();
        String methodName = joinPoint.getSignature().getName();
        System.out.println(
          "Execution of " + methodName + " took " + 
          TimeUnit.NANOSECONDS.toMillis(end - start) + " ms");
        return returnValue;
    }
}
```

In this example, we created a pointcut that matches all the methods in classes annotated with *@Repository*. Then we used the *@Around* advice to target that pointcut, and determine the execution time of the intercepted methods calls.

Furthermore, using this approach, we can add logging, performance management, audit, and other behaviors to each application layer.

9. Conclusion

In this article, we examined the Spring stereotype annotations and discussed what type of semantics they each represent.

We also learned how to use component scanning to tell the container where to find annotated classes.

Finally, we learned how these annotations **lead to a clean, layered design,** and separation between the concerns of an application. They also make configuration smaller, as we no longer need to explicitly define beans manually.

As usual, the examples are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-annotations).

« Previous
Spring Data Annotations (/spring-data-annotations)

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)