Spring Bean Annotations

**1. Overview**

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

There’re several ways to configure beans in a Spring container. We can declare them using XML configuration. We can declare beans using the *@Bean* annotation in a configuration class.

Or 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*):

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @ComponentScan(basePackages = "com.baeldung.annotations")  class VehicleFactoryConfig {} |

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

|  |  |
| --- | --- |
| 1  2  3 | @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:

|  |  |
| --- | --- |
| 1  2  3  4 | @Configuration  @ComponentScan(basePackages = "com.baeldung.annotations")  @ComponentScan(basePackageClasses = VehicleFactoryConfig.class)  class VehicleFactoryConfig {} |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @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:

|  |  |
| --- | --- |
| 1 | <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***.

For example:

|  |  |
| --- | --- |
| 1  2  3  4 | @Component  class CarUtility {      // ...  } |

By default, the bean instances of this class have the same name as the class name with a lowercase initial. On top of that, 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. Also, Spring 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:*

|  |  |
| --- | --- |
| 1  2  3  4 | @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:

|  |  |
| --- | --- |
| 1  2  3  4 | @Bean  public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {      return new PersistenceExceptionTranslationPostProcessor();  } |

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

Or, via XML configuration:

|  |  |
| --- | --- |
| 1  2 | <bean class=    "org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/> |

**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:

|  |  |
| --- | --- |
| 1  2  3  4 | @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**:

|  |  |
| --- | --- |
| 1  2  3  4 | @Controller  public class VehicleController {      // ...  } |

**7. *@Configuration***

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @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 example, 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 *@Repository* stereotype:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | @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 methods in classes annotated with *@Repository*. We used the *@Around* advice to then target that pointcut and determine the execution time of the intercepted methods calls.

Using this approach, we may add logging, performance management, audit, or other behaviors to each application layer.

**9. Conclusion**

In this article, we have examined the Spring stereotype annotations and learned what type of semantics these each represent.

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

Finally – we saw 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.