Spring Annotations

# **Spring Core Annotations**

## 2. DI-Related Annotations

### 2.1. @Autowired

We can use the @Autowired to **mark a dependency which Spring is going to resolve and inject**. We can use this annotation with a constructor, setter, or field injection.

Constructor injection:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Car {      Engine engine;        @Autowired      Car(Engine engine) {          this.engine = engine;      }  } |

Setter injection:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Car {      Engine engine;        @Autowired      void setEngine(Engine engine) {          this.engine = engine;      }  } |

Field injection:

|  |  |
| --- | --- |
| 1  2  3  4 | class Car {      @Autowired      Engine engine;  } |

@Autowired has a boolean argument called required with a default value of true. It tunes Spring’s behavior when it doesn’t find a suitable bean to wire. When true, an exception is thrown, otherwise, nothing is wired.

Note, that if we use constructor injection, all constructor arguments are mandatory.

Starting with version 4.3, we don’t need to annotate constructors with @Autowired explicitly unless we declare at least two constructors.

For more details visit our articles about [@Autowired](https://www.baeldung.com/spring-autowire) and [constructor injection](https://www.baeldung.com/constructor-injection-in-spring).

### 2.2. @Bean

@Bean marks a factory method which instantiates a Spring bean:

|  |  |
| --- | --- |
| 1  2  3  4 | @Bean  Engine engine() {      return new Engine();  } |

**Spring calls these methods** when a new instance of the return type is required.

The resulting bean has the same name as the factory method. If we want to name it differently, we can do so with the name or the value arguments of this annotation (the argument value is an alias for the argument name):

|  |  |
| --- | --- |
| 1  2  3  4 | @Bean("engine")  Engine getEngine() {      return new Engine();  } |

Note, that all methods annotated with @Bean must be in @Configuration classes.

### 2.3. @Qualifier

We use @Qualifier along with @Autowired to **provide the bean id or bean name** we want to use in ambiguous situations.

For example, the following two beans implement the same interface:

|  |  |
| --- | --- |
| 1  2  3 | class Bike implements Vehicle {}    class Car implements Vehicle {} |

If Spring needs to inject a Vehicle bean, it ends up with multiple matching definitions. In such cases, we can provide a bean’s name explicitly using the @Qualifier annotation.

Using constructor injection:

|  |  |
| --- | --- |
| 1  2  3  4 | @Autowired  Biker(@Qualifier("bike") Vehicle vehicle) {      this.vehicle = vehicle;  } |

Using setter injection:

|  |  |
| --- | --- |
| 1  2  3  4 | @Autowired  void setVehicle(@Qualifier("bike") Vehicle vehicle) {      this.vehicle = vehicle;  } |

Alternatively:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Autowired  @Qualifier("bike")  void setVehicle(Vehicle vehicle) {      this.vehicle = vehicle;  } |

Using field injection:

|  |  |
| --- | --- |
| 1  2  3 | @Autowired  @Qualifier("bike")  Vehicle vehicle; |

For a more detailed description, please read [this article](https://www.baeldung.com/spring-autowire).

### 2.4. @Required

@Required on setter methods to mark dependencies that we want to populate through XML:

|  |  |
| --- | --- |
| 1  2  3  4 | @Required  void setColor(String color) {      this.color = color;  } |
| 1  2  3 | <bean class="com.baeldung.annotations.Bike">      <property name="color" value="green" />  </bean> |

Otherwise, BeanInitializationException will be thrown.

### 2.5. @Scope

We use @Scope to define the [scope](https://www.baeldung.com/spring-bean-scopes) of a @Component class or a @Bean definition. It can be either singleton, prototype, request, session, globalSession or some custom scope.

For example:

|  |  |
| --- | --- |
| 1  2  3 | @Component  @Scope("prototype")  class Engine {} |

# **Spring Web Annotations**

## 2. @RequestMapping

Simply put, [@RequestMapping](https://www.baeldung.com/spring-requestmapping) **marks request handler methods** inside @Controller classes; it can be configured using:

* path, or its aliases, name, and value: which URL the method is mapped to
* method: compatible HTTP methods
* params: filters requests based on presence, absence, or value of HTTP parameters
* headers: filters requests based on presence, absence, or value of HTTP headers
* consumes: which media types the method can consume in the HTTP request body
* produces: which media types the method can produce in the HTTP response body

Here’s a quick example of what that looks like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Controller  class VehicleController {        @RequestMapping(value = "/vehicles/home", method = RequestMethod.GET)      String home() {          return "home";      }  } |

We can provide **default settings for all handler methods in a @Controller class** if we apply this annotation on the class level. The only **exception is the URL which Spring won’t override** with method level settings but appends the two path parts.

For example, the following configuration has the same effect as the one above:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Controller  @RequestMapping(value = "/vehicles", method = RequestMethod.GET)  class VehicleController {        @RequestMapping("/home")      String home() {          return "home";      }  } |

Moreover, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and @PatchMapping are different variants of @RequestMapping with the HTTP method already set to GET, POST, PUT, DELETE, and PATCH respectively.

These are available since Spring 4.3 release.

## 3. @RequestBody

Let’s move on to [@RequestBody](https://www.baeldung.com/spring-request-response-body) – which maps the **body of the HTTP request to an object**:

|  |  |
| --- | --- |
| 1  2  3  4 | @PostMapping("/save")  void saveVehicle(@RequestBody Vehicle vehicle) {      // ...  } |

The deserialization is automatic and depends on the content type of the request.

## 4. @PathVariable

Next, let’s talk about @PathVariable.

This annotation indicates that a **method argument is bound to a URI template variable**. We can specify the URI template with the @RequestMapping annotation and bind a method argument to one of the template parts with @PathVariable.

We can achieve this with the name or its alias, the value argument:

|  |  |
| --- | --- |
| 1  2  3  4 | @RequestMapping("/{id}")  Vehicle getVehicle(@PathVariable("id") long id) {      // ...  } |

If the name of the part in the template matches the name of the method argument, we don’t have to specify it in the annotation:

|  |  |
| --- | --- |
| 1  2  3  4 | @RequestMapping("/{id}")  Vehicle getVehicle(@PathVariable long id) {      // ...  } |

Moreover, we can mark a path variable optional by setting the argument required to false:

|  |  |
| --- | --- |
| 1  2  3  4 | @RequestMapping("/{id}")  Vehicle getVehicle(@PathVariable(required = false) long id) {      // ...  } |

## 5. @RequestParam

We use @RequestParam for **accessing HTTP request parameters**:

|  |  |
| --- | --- |
| 1  2  3  4 | @RequestMapping  Vehicle getVehicleByParam(@RequestParam("id") long id) {      // ...  } |

It has the same configuration options as the @PathVariable annotation.

In addition to those settings, with @RequestParam we can specify an injected value when Spring finds no or empty value in the request. To achieve this, we have to set the defaultValue argument.

Providing a default value implicitly sets required to false:

|  |  |
| --- | --- |
| 1  2  3  4 | @RequestMapping("/buy")  Car buyCar(@RequestParam(defaultValue = "5") int seatCount) {      // ...  } |

Besides parameters, there are **other HTTP request parts we can access: cookies and headers**. We can access them with the annotations **@CookieValue and @RequestHeader** respectively.

We can configure them the same way as @RequestParam.

## 6. Response Handling Annotations

In the next sections, we will see the most common annotations to manipulate HTTP responses in Spring MVC.

### 6.1. @ResponseBody

If we mark a request handler method with [*@ResponseBody*](https://www.baeldung.com/spring-request-response-body), **Spring treats the result of the method as the response itself**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @ResponseBody  @RequestMapping("/hello")  String hello() {      return "Hello World!";  } |

If we annotate a @Controller class with this annotation, all request handler methods will use it.

### 6.2. @ExceptionHandler

With this annotation, we can declare a **custom error handler method**. Spring calls this method when a request handler method throws any of the specified exceptions.

The caught exception can be passed to the method as an argument:

|  |  |
| --- | --- |
| 1  2  3  4 | @ExceptionHandler(IllegalArgumentException.class)  void onIllegalArgumentException(IllegalArgumentException exception) {      // ...  } |

## 7. Other Web Annotations

Some annotations don’t manage HTTP requests or responses directly. In the next sections, we’ll introduce the most common ones.

### 7.1. @Controller

We can define a Spring MVC controller with @Controller. For more information, please visit [our article about Spring Bean Annotations](https://www.baeldung.com/spring-bean-annotations).

### 7.2. @RestController

The @RestController **combines @Controller and @ResponseBody**.

Therefore, the following declarations are equivalent:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Controller  @ResponseBody  class VehicleRestController {      // ...  } |
| 1  2  3  4 | @RestController  class VehicleRestController {      // ...  } |

### 7.3. @ModelAttribute

With this annotation we can **access elements that are already in the model** of an MVC @Controller,by providing the model key:

|  |  |
| --- | --- |
| 1  2  3  4 | @PostMapping("/assemble")  void assembleVehicle(@ModelAttribute("vehicle") Vehicle vehicleInModel) {      // ...  } |

Like with @PathVariable and @RequestParam, we don’t have to specify the model key if the argument has the same name:

|  |  |
| --- | --- |
| 1  2  3  4 | @PostMapping("/assemble")  void assembleVehicle(@ModelAttribute Vehicle vehicle) {      // ...  } |

Besides, @ModelAttribute has another use: if we annotate a method with it, Spring will **automatically add the method’s return value to the model**:

|  |  |
| --- | --- |
| 1  2  3  4 | @ModelAttribute("vehicle")  Vehicle getVehicle() {      // ...  } |

Like before, we don’t have to specify the model key, Spring uses the method’s name by default:

|  |  |
| --- | --- |
| 1  2  3  4 | @ModelAttribute  Vehicle vehicle() {      // ...  } |

Before Spring calls a request handler method, it invokes all @ModelAttribute annotated methods in the class.

More information about @ModelAttribute can be found in [this article](https://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation).

## 8. Conclusion

In this article, we saw how we can handle HTTP requests and responses with Spring MVC.

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

# **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.