**Spring Core Annotations**

**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:

|  |
| --- |
| **class** Car {  Engine engine;    @Autowired  Car(Engine engine) {  **this**.engine = engine;  }  } |

Setter injection:

|  |
| --- |
| **class** Car {  Engine engine;    @Autowired  **void** setEngine(Engine engine) {  **this**.engine = engine;  }  } |

Field injection:

|  |
| --- |
| **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:

|  |
| --- |
| @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*):

|  |
| --- |
| @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:

|  |
| --- |
| **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:

|  |
| --- |
| @Autowired  Biker(@Qualifier(**"bike"**) Vehicle vehicle) {  **this**.vehicle = vehicle;  } |

Using setter injection:

|  |
| --- |
| @Autowired  **void** setVehicle(@Qualifier(**"bike"**) Vehicle vehicle) {  **this**.vehicle = vehicle;  } |

Alternatively:

|  |
| --- |
| @Autowired  @Qualifier(**"bike"**)  **void** setVehicle(Vehicle vehicle) {  **this**.vehicle = vehicle;  } |

Using field injection:

|  |
| --- |
| @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:

|  |
| --- |
| @Required  **void** setColor(String color) {  **this**.color = color;  } |

|  |
| --- |
| <**bean** class=**"com.baeldung.annotations.Bike"**>  <**property** name=**"color"** value=**"green"** />  </**bean**> |

Otherwise, *BeanInitializationException* will be thrown.

2.5. *@Value*

We can use [*@Value*](https://www.baeldung.com/spring-value-annotation) *for injecting property values into beans. It’s compatible with constructor, setter, and field injection.*

Constructor injection:

|  |
| --- |
| Engine(@Value(**"8"**) **int** cylinderCount) {  **this**.cylinderCount = cylinderCount;  } |

Setter injection:

|  |
| --- |
| @Autowired  **void** setCylinderCount(@Value(**"8"**) **int** cylinderCount) {  **this**.cylinderCount = cylinderCount;  } |

Alternatively:

|  |
| --- |
| @Value(**"8"**)  **void** setCylinderCount(**int** cylinderCount) {  **this**.cylinderCount = cylinderCount;  } |

Field injection:

|  |
| --- |
| @Value(**"8"**)  **int** cylinderCount; |

Of course, injecting static values isn’t useful. Therefore, we can use **placeholder strings** in *@Value* to wire values **defined in external sources**, for example, in *.properties* or *.yaml* files.

Let’s assume the following *.properties* file:

|  |
| --- |
| engine.fuelType=petrol |

We can inject the value of *engine.fuelType* with the following:

|  |
| --- |
| @Value(**"${engine.fuelType}"**)  String fuelType; |

We can use *@Value* even with SpEL. More advanced examples can be found in our [article about *@Value*](https://www.baeldung.com/spring-value-annotation).

2.6. *@DependsOn*

We can use this annotation to make Spring **initialize other beans before the annotated one**. Usually, this behavior is automatic, based on the explicit dependencies between beans.

We only need this annotation **when the dependencies are implicit**, for example, JDBC driver loading or static variable initialization.

We can use *@DependsOn* on the dependent class specifying the names of the dependency beans. The annotation’s *value* argument needs an array containing the dependency bean names:

|  |
| --- |
| @DependsOn(**"engine"**)  **class** Car **implements** Vehicle {} |

Alternatively, if we define a bean with the *@Bean* annotation, the factory method should be annotated with *@DependsOn*:

|  |
| --- |
| @Bean  @DependsOn(**"fuel"**)  Engine engine() {  **return** **new** Engine();  } |

2.7. *@Lazy*

We use [*@Lazy*](https://www.baeldung.com/spring-lazy-annotation) *when we want to initialize our bean lazily. By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context.*

However, there are cases when **we need to create a bean when we request it, not at application startup**.

This annotation behaves differently depending on where we exactly place it. We can put it on:

* a *@Bean* annotated bean factory method, to delay the method call (hence the bean creation)
* a @*Configuration* class and all contained *@Bean* methods will be affected
* a *@Component* class, which is not a *@Configuration* class, this bean will be initialized lazily
* an *@Autowired* constructor, setter, or field, to load the dependency itself lazily (via proxy)

This annotation has an argument named *value* with the default value of *true*. It is useful to override the default behavior.

For example, marking beans to be eagerly loaded when the global setting is lazy, or configure specific *@Bean* methods to eager loading in a *@Configuration* class marked with *@Lazy*:

|  |
| --- |
| @Configuration  @Lazy  **class** VehicleFactoryConfig {    @Bean  @Lazy(**false**)  Engine engine() {  **return** **new** Engine();  }  } |

For further reading, please visit [this article](https://www.baeldung.com/spring-lazy-annotation).

2.8. *@Lookup*

A method annotated with *@Lookup* tells Spring to return an instance of the method’s return type when we invoke it.

Detailed information about the annotation [can be found in this article](https://www.baeldung.com/spring-lookup).

2.9. *@Primary*

Sometimes we need to define multiple beans of the same type. In these cases, the injection will be unsuccessful because Spring has no clue which bean we need.

We already saw an option to deal with this scenario: marking all the wiring points with *@Qualifier* and specify the name of the required bean.

However, most of the time we need a specific bean and rarely the others. We can use *@Primary* to simplify this case: if **we mark the most frequently used bean with *@Primary*** it will be chosen on unqualified injection points:

|  |
| --- |
| @Component  @Primary  **class** Car **implements** Vehicle {}    @Component  **class** Bike **implements** Vehicle {}    @Component  **class** Driver {  @Autowired  Vehicle vehicle;  }    @Component  **class** Biker {  @Autowired  @Qualifier(**"bike"**)  Vehicle vehicle;  } |

In the previous example *Car* is the primary vehicle. Therefore, in the *Driver* class, Spring injects a *Car* bean. Of course, in the *Biker* bean, the value of the field *vehicle* will be a *Bike* object because it’s qualified.

2.10. *@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:

|  |
| --- |
| @Component  @Scope(**"prototype"**)  **class** Engine {} |

**3. Context Configuration Annotations**

We can configure the application context with the annotations described in this section.

3.1. *@Profile*

If we want Spring to **use a *@Component* class or a *@Bean* method only when a specific profile is active**, we can mark it with *@Profile*. We can configure the name of the profile with the *value* argument of the annotation:

|  |
| --- |
| @Component  @Profile(**"sportDay"**)  **class** Bike **implements** Vehicle {} |

You can read more about profiles in [this article](https://www.baeldung.com/spring-profiles).

3.2. *@Import*

We can use **specific *@Configuration* classes without component scanning** with this annotation. We can provide those classes with *@Import*‘s *value* argument:

|  |
| --- |
| @Import(VehiclePartSupplier.**class**)  **class** VehicleFactoryConfig {} |

3.3. *@ImportResource*

We can **import XML configurations** with this annotation. We can specify the XML file locations with the *locations* argument, or with its alias, the *value* argument:

|  |
| --- |
| @Configuration  @ImportResource(**"classpath:/annotations.xml"**)  **class** VehicleFactoryConfig {} |

3.4. *@PropertySource*

With this annotation, we can **define property files for application settings**:

|  |
| --- |
| @Configuration  @PropertySource(**"classpath:/annotations.properties"**)  **class** VehicleFactoryConfig {} |

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

|  |
| --- |
| @Configuration  @PropertySource(**"classpath:/annotations.properties"**)  @PropertySource(**"classpath:/vehicle-factory.properties"**)  **class** VehicleFactoryConfig {} |

3.5. *@PropertySources*

We can use this annotation to specify multiple *@PropertySource* configurations:

|  |
| --- |
| @Configuration  @PropertySources({  @PropertySource(**"classpath:/annotations.properties"**),  @PropertySource(**"classpath:/vehicle-factory.properties"**)  })  **class** VehicleFactoryConfig {} |

Note, that since Java 8 we can achieve the same with the repeating annotations feature as described above.

**Spring Bean Annotations**

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**](mailto:@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.

**Spring Data Annotations**

Introduction

Spring Data provides an abstraction over data storage technologies. Therefore, our business logic code can be much more independent of the underlying persistence implementation. Also, Spring simplifies the handling of implementation-dependent details of data storage.

In this tutorial, we’ll see the most common annotations of the Spring Data, Spring Data JPA, and Spring Data MongoDB projects.

2. Common Spring Data Annotations

2.1. *@Transactional*

When we want to **configure the transactional behavior of a method**, we can do it with:

|  |  |
| --- | --- |
| 1  2 | @Transactional  **void** pay() {} |

If we apply this annotation on class level, then it works on all methods inside the class. However, we can override its effects by applying it to a specific method.

It has many configuration options, which can be found in [this article](https://www.baeldung.com/transaction-configuration-with-jpa-and-spring).

2.2. *@NoRepositoryBean*

**Sometimes we want to create repository interfaces with the only goal of providing common methods for the child repositories**.

Of course, we don’t want Spring to create a bean of these repositories since we won’t inject them anywhere. *@NoRepositoryBean* does exactly this: when we mark a child interface of *org.springframework.data.repository.Repository*, Spring won’t create a bean out of it.

For example, if we want an *Optional<T> findById(ID id)* method in all of our repositories, we can create a base repository:

|  |  |
| --- | --- |
| 1  2  3  4 | @NoRepositoryBean  **interface** MyUtilityRepository<T, ID **extends** Serializable> **extends** CrudRepository<T, ID> {  Optional<T> findById(ID id);  } |

This annotation doesn’t affect the child interfaces; hence Spring will create a bean for the following repository interface:

|  |  |
| --- | --- |
| 1  2 | @Repository  **interface** PersonRepository **extends** MyUtilityRepository<Person, Long> {} |

Note, that the example above isn’t necessary since Spring Data version 2 which includes this method replacing the older *T findOne(ID id)*.

2.3. *@Param*

We can pass named parameters to our queries using *@Param*:

|  |  |
| --- | --- |
| 1  2 | @Query(**"FROM Person p WHERE p.name = :name"**)  Person findByName(@Param(**"name"**) String name); |

Note, that we refer to the parameter with the *:name* syntax.

For further examples, please visit [this article](https://www.baeldung.com/spring-data-jpa-query).

2.4. *@Id*

*@Id* marks a field in a model class as the primary key:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **class** Person {    @Id  Long id;    // ...    } |

Since it’s implementation-independent, it makes a model class easy to use with multiple data store engines.

2.5. *@Transient*

We can use this annotation to mark a field in a model class as transient. Hence the data store engine won’t read or write this field’s value:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **class** Person {    // ...    @Transient  **int** age;    // ...    } |

Like *@Id*, *@Transient* is also implementation-independent, which makes it convenient to use with multiple data store implementations.

2.6. *@CreatedBy*, *@LastModifiedBy*, *@CreatedDate*, *@LastModifiedDate*

With these annotations, we can audit our model classes: Spring automatically populates the annotated fields with the principal who created the object, last modified it, and the date of creation, and last modification:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **public** **class** Person {    // ...    @CreatedBy  User creator;    @LastModifiedBy  User modifier;    @CreatedDate  Date createdAt;    @LastModifiedDate  Date modifiedAt;    // ...    } |

Note, that if we want Spring to populate the principals, we need to use Spring Security as well.

For a more thorough description, please visit [this article](https://www.baeldung.com/database-auditing-jpa).

3. Spring Data JPA Annotations

3.1. *@Query*

With *@Query*, we can provide a JPQL implementation for a repository method:

|  |  |
| --- | --- |
| 1  2 | @Query(**"SELECT COUNT(\*) FROM Person p"**)  **long** getPersonCount(); |

Also, we can use named parameters:

|  |  |
| --- | --- |
| 1  2 | @Query(**"FROM Person p WHERE p.name = :name"**)  Person findByName(@Param(**"name"**) String name); |

Besides, we can use native SQL queries, if we set the *nativeQuery* argument to *true*:

|  |  |
| --- | --- |
| 1  2 | @Query(value = **"SELECT AVG(p.age) FROM person p"**, nativeQuery = **true**)  **int** getAverageAge(); |

For more information, please visit [this article](https://www.baeldung.com/spring-data-jpa-query).

3.2. *@Procedure*

**With Spring Data JPA we can easily call stored procedures from repositories.**

First, we need to declare the repository on the entity class using standard JPA annotations:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @NamedStoredProcedureQueries({  @NamedStoredProcedureQuery(  name = **"count\_by\_name"**,  procedureName = **"person.count\_by\_name"**,  parameters = {  @StoredProcedureParameter(  mode = ParameterMode.IN,  name = **"name"**,  type = String.**class**),  @StoredProcedureParameter(  mode = ParameterMode.OUT,  name = **"count"**,  type = Long.**class**)  }  )  })    **class** Person {} |

After this, we can refer to it in the repository with the name we declared in the *name* argument:

|  |  |
| --- | --- |
| 1  2 | @Procedure(name = **"count\_by\_name"**)  **long** getCountByName(@Param(**"name"**) String name); |

3.3. *@Lock*

We can configure the lock mode when we execute a repository query method:

|  |  |
| --- | --- |
| 1  2  3 | @Lock(LockModeType.NONE)  @Query(**"SELECT COUNT(\*) FROM Person p"**)  **long** getPersonCount(); |

The available lock modes:

* *READ*
* *WRITE*
* *OPTIMISTIC*
* *OPTIMISTIC\_FORCE\_INCREMENT*
* *PESSIMISTIC\_READ*
* *PESSIMISTIC\_WRITE*
* *PESSIMISTIC\_FORCE\_INCREMENT*
* *NONE*

3.4. *@Modifying*

We can modify data with a repository method if we annotate it with *@Modifying*:

|  |  |
| --- | --- |
| 1  2  3 | @Modifying  @Query(**"UPDATE Person p SET p.name = :name WHERE p.id = :id"**)  **void** changeName(@Param(**"id"**) **long** id, @Param(**"name"**) String name); |

For more information, please visit [this article](https://www.baeldung.com/spring-data-jpa-query).

3.5. *@EnableJpaRepositories*

To use JPA repositories, we have to indicate it to Spring. We can do this with *@EnableJpaRepositories.*

Note, that we have to use this annotation with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableJpaRepositories  **class** PersistenceJPAConfig {} |

Spring will look for repositories in the sub packages of this *@Configuration* class.

We can alter this behavior with the *basePackages* argument:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableJpaRepositories(basePackages = **"org.baeldung.persistence.dao"**)  **class** PersistenceJPAConfig {} |

Also note, that Spring Boot does this automatically if it finds Spring Data JPA on the classpath.

4. Spring Data Mongo Annotations

Spring Data makes working with MongoDB much easier. In the next sections, we’ll explore the most basic features of Spring Data MongoDB.

For more information, please visit our [article about Spring Data MongoDB](https://www.baeldung.com/spring-data-mongodb-tutorial).

4.1. *@Document*

This annotation marks a class as being a domain object that we want to persist to the database:

|  |  |
| --- | --- |
| 1  2 | @Document  **class** User {} |

It also allows us to choose the name of the collection we want to use:

|  |  |
| --- | --- |
| 1  2 | @Document(collection = **"user"**)  **class** User {} |

Note, that this annotation is the Mongo equivalent of *@Entity* in JPA.

4.2. *@Field*

With *@Field*, we can configure the name of a field we want to use when MongoDB persists the document:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Document  **class** User {    // ...    @Field(**"email"**)  String emailAddress;    // ...    } |

Note, that this annotation is the Mongo equivalent of *@Column* in JPA.

4.3. *@Query*

With *@Query*, we can provide a finder query on a MongoDB repository method:

|  |  |
| --- | --- |
| 1  2 | @Query(**"{ 'name' : ?0 }"**)  List<User> findUsersByName(String name); |

4.4. *@EnableMongoRepositories*

To use MongoDB repositories, we have to indicate it to Spring. We can do this with *@EnableMongoRepositories.*

Note, that we have to use this annotation with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableMongoRepositories  **class** MongoConfig {} |

Spring will look for repositories in the sub packages of this *@Configuration* class. We can alter this behavior with the *basePackages* argument:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableMongoRepositories(basePackages = **"org.baeldung.repository"**)  **class** MongoConfig {} |

Also note, that Spring Boot does this automatically if it finds Spring Data MongoDB on the classpath.

**Spring Scheduling Annotations**

Overview

When single-threaded execution isn’t enough, we can use annotations from the *org.springframework.scheduling.annotation* package.

In this quick tutorial, we’re going to explore the Spring Scheduling Annotations.

2. *@EnableAsync*

With this annotation, we can enable asynchronous functionality in Spring.

We must use it with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableAsync  **class** VehicleFactoryConfig {} |

Now, that we enabled asynchronous calls, we can use *@Async* to define the methods supporting it.

3. *@EnableScheduling*

With this annotation, we can enable scheduling in the application.

We also have to use it in conjunction with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableScheduling  **class** VehicleFactoryConfig {} |

As a result, we can now run methods periodically with *@Scheduled*.

4. *@Async*

We can define methods we want to **execute on a different thread**, hence run them asynchronously.

To achieve this, we can annotate the method with *@Async*:

|  |  |
| --- | --- |
| 1  2  3  4 | @Async  **void** repairCar() {  // ...  } |

If we apply this annotation to a class, then all methods will be called asynchronously.

Note, that we need to enable the asynchronous calls for this annotation to work, with *@EnableAsync* or XML configuration.

More information about *@Async* can be found in [this article](https://www.baeldung.com/spring-async).

5. *@Scheduled*

If we need a method to **execute periodically**, we can use this annotation:

|  |  |
| --- | --- |
| 1  2  3  4 | @Scheduled(fixedRate = 10000)  **void** checkVehicle() {  // ...  } |

We can use it to execute a method at **fixed intervals**, or we can fine-tune it with **cron-like expressions**.

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Scheduled(fixedRate = 10000)  @Scheduled(cron = **"0 \* \* \* \* MON-FRI"**)  **void** checkVehicle() {  // ...  } |

Note, that the method annotated with *@Scheduled* should have a *void* return type.

Moreover, we have to enable scheduling for this annotation to work for example with *@EnableScheduling* or XML configuration.

For more information about scheduling read [this article](https://www.baeldung.com/spring-scheduled-tasks).

6. *@Schedules*

We can use this annotation to specify multiple *@Scheduled* rules:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Schedules({  @Scheduled(fixedRate = 10000),  @Scheduled(cron = **"0 \* \* \* \* MON-FRI"**)  })  **void** checkVehicle() {  // ...  } |

Note, that since Java 8 we can achieve the same with the repeating annotations feature as described above.

**Spring Boot Annotations**

Overview

Spring Boot made configuring Spring easier with its auto-configuration feature.

In this quick tutorial, we’ll explore the annotations from the *org.springframework.boot.autoconfigure* and *org.springframework.boot.autoconfigure.condition*packages.

2. *@SpringBootApplication*

We use this annotation to **mark the main class of a Spring Boot application**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @SpringBootApplication  **class** VehicleFactoryApplication {    **public** **static** **void** main(String[] args) {  SpringApplication.run(VehicleFactoryApplication.**class**, args);  }  } |

*@SpringBootApplication* encapsulates ***@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan*** annotations with their default attributes.

3. *@EnableAutoConfiguration*

*@EnableAutoConfiguration*, as its name says, enables auto-configuration. It means that **Spring Boot looks for auto-configuration beans** on its classpath and automatically applies them.

Note, that we have to use this annotation with *@Configuration*:

|  |  |
| --- | --- |
| 1  2  3 | @Configuration  @EnableAutoConfiguration  **class** VehicleFactoryConfig {} |

4. Auto-Configuration Conditions

Usually, when we write our **custom auto-configurations**, we want Spring to **use them conditionally**. We can achieve this with the annotations in this section.

We can place the annotations in this section on *@Configuration* classes or *@Bean* methods.

In the next sections, we’ll only introduce the basic concept behind each condition. For further information, please visit [this article](https://www.baeldung.com/spring-boot-custom-auto-configuration).

4.1. *@ConditionalOnClass* and *@ConditionalOnMissingClass*

Using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation’s **argument is present/absent**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Configuration  @ConditionalOnClass(DataSource.**class**)  **class** MySQLAutoconfiguration {  //...  } |

4.2. *@ConditionalOnBean* and *@ConditionalOnMissingBean*

We can use these annotations when we want to define conditions based on the **presence or absence of a specific bean**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Bean  @ConditionalOnBean(name = **"dataSource"**)  LocalContainerEntityManagerFactoryBean entityManagerFactory() {  // ...  } |

4.3. *@ConditionalOnProperty*

With this annotation, we can make conditions on the **values of properties**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Bean  @ConditionalOnProperty(  name = **"usemysql"**,  havingValue = **"local"**  )  DataSource dataSource() {  // ...  } |

4.4. *@ConditionalOnResource*

We can make Spring to use a definition only when a specific **resource is present**:

|  |  |
| --- | --- |
| 1  2  3  4 | @ConditionalOnResource(resources = **"classpath:mysql.properties"**)  Properties additionalProperties() {  // ...  } |

4.5. *@ConditionalOnWebApplication* and *@ConditionalOnNotWebApplication*

With these annotations, we can create conditions based on if the current **application is or isn’t a web application**:

|  |  |
| --- | --- |
| 1  2  3  4 | @ConditionalOnWebApplication  HealthCheckController healthCheckController() {  // ...  } |

4.6. *@ConditionalExpression*

We can use this annotation in more complex situations. Spring will use the marked definition when the **SpEL expression is evaluated to true**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Bean  @ConditionalOnExpression(**"${usemysql} && ${mysqlserver == 'local'}"**)  DataSource dataSource() {  // ...  } |

4.7. *@Conditional*

For even more complex conditions, we can create a class evaluating the **custom condition**. We tell Spring to use this custom condition with *@Conditional*:

|  |  |
| --- | --- |
| 1  2  3  4 | @Conditional(HibernateCondition.**class**)  Properties additionalProperties() {  //...  } |

**Spring Web Annotations**

Overview

In this tutorial, we’ll explore Spring Web annotations from the *org.springframework.web.bind.annotation* package.

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) {  // ...  } |

6.3. *@ResponseStatus*

We can specify the **desired HTTP status of the response** if we annotate a request handler method with this annotation. We can declare the status code with the *code* argument, or its alias, the *value* argument.

Also, we can provide a reason using the *reason* argument.

We also can use it along with *@ExceptionHandler*:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @ExceptionHandler(IllegalArgumentException.**class**)  @ResponseStatus(HttpStatus.BAD\_REQUEST)  **void** onIllegalArgumentException(IllegalArgumentException exception) {  // ...  } |

For more information about HTTP response status, please visit [this article](https://www.baeldung.com/spring-mvc-controller-custom-http-status-code).

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).

7.4. *@CrossOrigin*

*@CrossOrigin* **enables cross-domain communication** for the annotated request handler methods:

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

If we mark a class with it, it applies to all request handler methods in it.

We can fine-tune CORS behavior with this annotation’s arguments.

For more details, please visit [this article](https://www.baeldung.com/spring-cors).