### Spring Annotations Descriptions (<https://www.journaldev.com/2888/spring-tutorial-spring-core-tutorial>)

*@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.

1. [**Spring @Bean**](https://www.journaldev.com/21577/spring-bean-annotation): Spring @Bean Annotation is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods. It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.
2. [**Spring @Service**](https://www.journaldev.com/21435/spring-service-annotation): Spring @Service annotation is a specialization of @Component annotation. Spring Service annotation can be applied only to classes. It is used to mark the class as a service provider.

@Service annotation is used in your service layer and annotates classes that perform service tasks, often you don't use it but in many case you use this annotation to represent a best practice. It tells the Spring that class contains the **business logic**.

For example, you could directly call a DAO class to persist an object to your database but this is horrible. It is pretty good to call a service class that calls a DAO. This is a good thing to perform the separation of concerns pattern.

1. [**Spring @Repository**](https://www.journaldev.com/21460/spring-repository-annotation): Spring @Repository annotation is used to indicate that the class provides the mechanism for storage, retrieval, search, update and delete operation on objects.
2. [**Spring @Component**](https://www.journaldev.com/21429/spring-component): Spring Component annotation is used to denote a class as Component. It means that the Spring framework will autodetect these classes for dependency injection when annotation-based configuration and classpath scanning is used.
3. [**Spring @RestController**](https://www.journaldev.com/21536/spring-restcontroller): Spring RestController annotation is a convenience annotation that is itself annotated with @Controller and @ResponseBody. This annotation is applied to a class to mark it as a request handler.
4. [**Spring @Configuration**](https://www.journaldev.com/21033/spring-configuration-annotation): Spring @Configuration annotation is part of the spring core framework. Spring Configuration annotation indicates that the class has @Bean definition methods. So Spring container can process the class and generate Spring Beans to be used in the application.
5. [**Spring @Value**](https://www.journaldev.com/21448/spring-value-annotation): Spring @Value annotation is used to assign default values to variables and method arguments. We can read spring environment variables as well as system variables using @Value annotation.
6. [**Spring @PropertySource**](https://www.journaldev.com/21440/spring-propertysource): Spring @PropertySource annotation is used to provide properties file to Spring Environment. This annotation is used with @Configuration classes.

[**Spring @PostConstruct and @PreDestroy**](https://www.journaldev.com/21206/spring-postconstruct-predestroy): When we annotate a method in Spring Bean with @PostConstruct annotation, it gets executed after the spring bean is initialized.

When we annotate a Spring Bean method with PreDestroy annotation, it gets called when bean instance is getting removed from the context.

[**Spring @Async**](https://www.journaldev.com/20457/spring-async-annotation): Spring @Async annotation allows us to create asynchronous methods in spring. Let’s explore @Async in this tutorial on spring framework.

**@ComponentScan(basePackages = "com.test.springmvc")** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.

1. [**Spring @Controller**](https://www.journaldev.com/21515/spring-controller-spring-mvc-controller):

* The @Controller is a class-level annotation. It is a specialization of **@Component**.
* It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.
* @Controller annotation is an annotation used in Spring MVC framework (the component of Spring Framework used to implement Web Application).
* The @Controller annotation indicates that a particular class serves the role of a controller.
* The @Controller annotation acts as a stereotype for the annotated class, indicating its role.
* The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations.

So looking at the Spring MVC architecture you have a DispatcherServlet class (that you declare in your XML configuration) that represent a front controller that dispatch all the HTTP Request towards the appropriate controller classes (annotated by @Controller).

* This class perform the business logic (and can call the services) by its method.
* These classes (or its methods) are typically annotated also with **@RequestMapping** annotation that specify what HTTP Request is handled by the controller and by its method.

### Spring MVC and REST Annotations

**@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.

* **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
* **@PostMapping:** It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
* **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
* **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
* **@PatchMapping:** It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
* **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.

It enables automatic deserialization of the inbound HttpRequest body onto a Java object.

@PostMapping("/request")

public ResponseEntity postController(

  @RequestBody LoginForm loginForm) {

    exampleService.fakeAuthenticate(loginForm);

    return ResponseEntity.ok(HttpStatus.OK);

}

* **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

@Controller

@RequestMapping("/post")

public class ExamplePostController {

    @Autowired

    ExampleService exampleService;

    @PostMapping("/response")

    @ResponseBody

    public ResponseTransfer postResponseController(

      @RequestBody LoginForm loginForm) {

        return new ResponseTransfer("Thanks For Posting!!!");

     }

}

{"text":"Thanks For Posting!!!"}

* **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
* **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

**http://:8080/api/users?page=1&limit=50**

* **HTTP** – is the protocol being used,
* **localhost**  – is the domain name,
* **8080 –**is the port number,
* **api** – is the root path of your web services application,
* **users –**is most likely the @Path value of your Root Resource class and,
* **page**– is the URL Query parameter which we can read with @RequestParam annotation,
* **limit** – is the URL Query parameter which we can also read with @RequestParam annotation.

**Example: public List<UserRest> getUsers(@RequestParam(value = "page", defaultValue = "0") int page,**

**@RequestParam(value = "limit", defaultValue = "30") int limit) {**

**List<UserRest> returnValue = new ArrayList<>();**

**List<UserDto> users = userService.getUsers(page, limit);**

}

* **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.Accessing pre-existing request attributes using @RequestAttribute

===================

## Difference between @RequestParam and @RequestAttribute

[**@RequestParam**](https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/spring-mvc-request-param.html) is used to bind parameter values from 'query string' e.g. in http://www.example.com?myParam=3, myParam=3 can populate @RequestParam parameter.

**@RequestAttribute** is to access objects which have been populated on the server-side but during the same HTTP request, for example they can be populated in an interceptor or a filter. The difference is same as of the difference between ServletRequest#getParameter(name) and ServletRequest#getAttribute(name)

======================================

**The @ModelAttribute**

1. @ModelAttribute can be used either as a method parameter or at the method level.
2. When the annotation is used at the method level it indicates the purpose of that method is to add one or more model attributes. Such methods support the same argument types as [@RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) methods but that cannot be mapped directly to requests.

Let's have a look at a quick example here to start understanding how this works:

|  |  |
| --- | --- |
|  | @ModelAttribute  public void addAttributes(Model model) {      model.addAttribute("msg", "Welcome to the Netherlands!"); |

}

1. **@ModelAttribute methods are invoked before the controller methods annotated with @RequestMapping are invoked.**
2. **As a Method Argument:** When used as a method argument, it indicates the argument should be retrieved from the model. When not present, it should be first instantiated and then added to the model and once present in the model, the arguments fields should be populated from all request parameters that have matching names.

|  |  |
| --- | --- |
|  | @RequestMapping(value = "/addEmployee", method = RequestMethod.POST)  public String submit(@ModelAttribute("employee") Employee employee) {      // Code that uses the employee object        return "employeeView";  } |

==============================

**@Qualifier annotation(**<https://www.baeldung.com/spring-qualifier-annotation>**)**

## Autowire Need for Disambiguation

The [@Autowired](https://www.baeldung.com/spring-autowire) annotation is a great way of making the need to inject a dependency in Spring explicit. And although it's useful, there are use cases for which this annotation alone isn't enough for Spring to understand which bean to inject.

By default, Spring resolves autowired entries by type.

**If more than one bean of the same type is available in the container, the framework will throw NoUniqueBeanDefinitionException**, indicating that more than one bean is available for autowiring.

Let's imagine a situation in which two possible candidates exist for Spring to inject as bean collaborators in a given instance:

|  |  |
| --- | --- |
|  | @Component("fooFormatter")  public class FooFormatter implements Formatter {        public String format() {          return "foo";      }  }  @Component("barFormatter")  public class BarFormatter implements Formatter {        public String format() {          return "bar";      }  }  @Component  public class FooService {      @Autowired      private Formatter formatter;  } |

If we try to load FooService into our context, the Spring framework will throw a NoUniqueBeanDefinitionException. This is because **Spring doesn't know which bean to inject**. To avoid this problem, there are several solutions. The @Qualifier annotation is one of them.

## 3. @Qualifier Annotation

By using the @Qualifier annotation, we can **eliminate the issue of which bean needs to be injected**.

Let's revisit our previous example and see how we solve the problem by including the @Qualifier annotation to indicate which bean we want to use:

|  |  |
| --- | --- |
|  | public class FooService {        @Autowired      @Qualifier("fooFormatter")      private Formatter formatter;  } |

By including the @Qualifier annotation together with the name of the specific implementation we want to use – in this example, Foo – we can avoid ambiguity when Spring finds multiple beans of the same type.

We need to take into consideration that the qualifier name to be used is the one declared in the @Component annotation.

Note that we could've also used the @Qualifier annotation on the Formatter implementing classes, instead of specifying the names in their @Component annotations, to obtain the same effect:

|  |  |
| --- | --- |
|  | @Component  @Qualifier("fooFormatter")  public class FooFormatter implements Formatter {      //...  }    @Component  @Qualifier("barFormatter")  public class BarFormatter implements Formatter {      //...  } |

## 4. @Qualifier vs @Primary

There's another annotation called [@Primary](https://www.baeldung.com/spring-primary) that we can use to decide which bean to inject when ambiguity is present regarding dependency injection.

This annotation **defines a preference when multiple beans of the same type are present**. The bean associated with the @Primary annotation will be used unless otherwise indicated.

Let's see an example:

|  |  |
| --- | --- |
|  | @Configuration  public class Config {        @Bean      public Employee johnEmployee() {          return new Employee("John");      }        @Bean      @Primary      public Employee tonyEmployee() {          return new Employee("Tony");      }  } |

In this example, both methods return the same Employee type. The bean that Spring will inject is the one returned by the method tonyEmployee. This is because it contains the @Primary annotation. This annotation is useful when we want to **specify which bean of a certain type should be injected by default**.

And in case we require the other bean at some injection point, we would need to specifically indicate it. We can do that via the @Qualifier annotation. For instance, we could specify that we want to use the bean returned by the johnEmployee method by using the @Qualifier annotation.

It's worth noting that **if both the @Qualifier and @Primary annotations are present, then the @Qualifier annotation will have precedence.** Basically, @Primary defines a default, while @Qualifier is very specific.

Let's see another way of using the @Primary annotation, this time using the initial example:

|  |  |
| --- | --- |
|  | @Component  @Primary  public class FooFormatter implements Formatter {      //...  }    @Component  public class BarFormatter implements Formatter {      //...  } |

**In this case, the @Primary annotation is placed in one of the implementing classes** and will disambiguate the scenario.

## 5. @Qualifier vs Autowiring by Name

Another way to decide between multiple beans when autowiring is by using the name of the field to inject. **This is the default in case there are no other hints for Spring**. Let's see some code based on our initial example:

|  |  |
| --- | --- |
|  | public class FooService {        @Autowired      private Formatter fooFormatter;  } |

In this case, Spring will determine that the bean to inject is the FooFormatter one since the field name is matched to the value that we used in the @Component annotation for that bean.

## Core Spring Framework Annotations

**@Required:** It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

public class Machine

{

private Integer cost;

@Required

public void setCost(Integer cost)

{

this.cost = cost;

}

public Integer getCost()

{

return cost;

}

}

**@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

@Component

public class Customer

{

private Person person;

@Autowired

public Customer(Person person)

{

this.person=person;

}

}

## Spring Boot Annotations

* **@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.
* **@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

**Extra Annotations to read:**

@profile

**Difference between @RestController and @Controller in Spring**

1. The @Controller is a common annotation which is used to mark a class as Spring MVC Controller while the @RestController is a special controller used in [RESTFul web services](https://javarevisited.blogspot.sg/2015/08/difference-between-soap-and-restfull-webservice-java.html" \t "_blank) and the equivalent of @Controller + @ResponseBody.
2. The @RestController is relatively new, added only on Spring 4.0 but @Controller is an old annotation, exists since Spring started supporting annotation, and officially it was added on Spring 2.5 version.
3. The @Controller is a specialization of @Component annotation while @RestController is a specialization of @Controller annotation.

**Q. @Controller vs RestController**

**Ans:** some important differences between these two annotations.  
The job of @Controller is to create a Map of model object and find a view but @RestController simply return the object and object data is directly written into HTTP response as JSON or XML.  
This can also be done with traditional @Controller and use @ResponseBody annotation but since this is the default behavior of RESTful Web services, Spring introduced @RestController which combined the behavior of @Controller and @ResponseBody together.

1. The @Controller is a common annotation which is used to mark a class as Spring MVC Controller while @RestController is a special controller used in [RESTFul web services](http://javarevisited.blogspot.sg/2015/08/difference-between-soap-and-restfull-webservice-java.html" \t "_blank) and the equivalent of @Controller + @ResponseBody.  
  
2. The **@RestController** is relatively new, added only on Spring 4.0 but @Controller is an old annotation, exists since Spring started supporting annotation, officially it was added on Spring 2.5 version.  
  
3. The @Controller annotation indicates that the class is a "Controller" like a web controller while @RestController annotation indicates that the class is a controller where @RequestMapping methods assume **@ResponseBody** semantics by default i.e. servicing REST API.

@Controller

@ResponseBody

public class MVCController {

.. your logic

}

@RestController

public class RestFulController {

.... your logic

}

**Q. @EnableAutoConfiguration vs @SpringBootApplication**

Even though both **@SpringBootApplication** and **@EnableAutoConfiguration** can be used to enable the **auto-configuration feature of Spring Boot** there is a subtle difference between them. The @SpringBootApplication does much more than what @EnableAutoConfiguration do. It's actually a combination of three annotations: **@Configuration** which is used in Java-based configuration on Spring framework, **@ComponentScan** to enable component scanning of components you write e.g. [@Controller](http://javarevisited.blogspot.sg/2017/11/difference-between-component-service.html) classes, and **@EnableAutoConfgiuration** itself, which is used to enable auto-configuration in [Spring Boot](https://javarevisited.blogspot.sg/2018/02/top-5-spring-microservices-courses-with-spring-boot-and-spring-cloud.html)application. Spring Boot designers realize that these three annotations are frequently used together so they bundled them into @SpringBootApplicaiton. Now, instead of three annotations you just need to specify one annotation on your Main class.

**Q: What is the difference between @EnableAutoConfiguration and @SpringBootApplication is also a common Spring** Boot interview question and it was recently asked one of my friends on his Java interview. Given the popularity of [Spring Boot framework](http://www.java67.com/2017/11/top-5-free-core-spring-mvc-courses-learn-online.html), it's good to know such questions before you go for your next Java interview.

The @EnableAutoConfiguration annotation allows you to **selectively exclude certain classes** from auto-configuration using exclude attribute as shown below:

@Configuration

@EnableAutoConfiguration(exclude**=**{DataSourceAutoConfiguration.class})

**public** class MyConfiguration {

//.. Java code

}

The @EnableAutoConfiguration annotations enable auto-configuration features of Spring Boot which configures modules based on the presence of certain classes on the classpath. For example, if Thymeleaf JAR is present in classpath and [Spring MVC](https://javarevisited.blogspot.sg/2017/06/how-spring-mvc-framework-works-web-flow.html) is enabled e.g. using spring-boot-web-starter package then it can automatically configure template resolver and view resolver for you.  
  
The @EnableAutoConfiguration annotation is based on @Conditional annotation of Spring 4.0 which enables conditional configuration.  
  
In case of auto-configuration, manually declared beans can override beans automatically created by auto-configuration feature. This is achieved by using @ConditionalOnMissingBean of Spring 4.0