Thursday, February 9, 2023

Home About Privacy Policy Terms & Conditions Disclaimer Contact Us Guest Post

Making Java easy to learn

Java Technology and Beyond

You are here

Home > Spring Boot >

Spring Boot MVC REST Annotations With Examples

Spring Boot Java Spring *by devs5003 February 28, 2021* 1

In this article, we will discuss on 'Spring Boot MVC REST Annotations With Examples'. Needless to say, these annotations are very important for creating a web application in Spring Boot. If you want to learn all annotations which are generally used in a Spring Boot Project, kindly visit our article 'Spring Boot Annotations with Examples'. Let's discuss about 'Spring Boot MVC REST Annotations With Examples' here only.

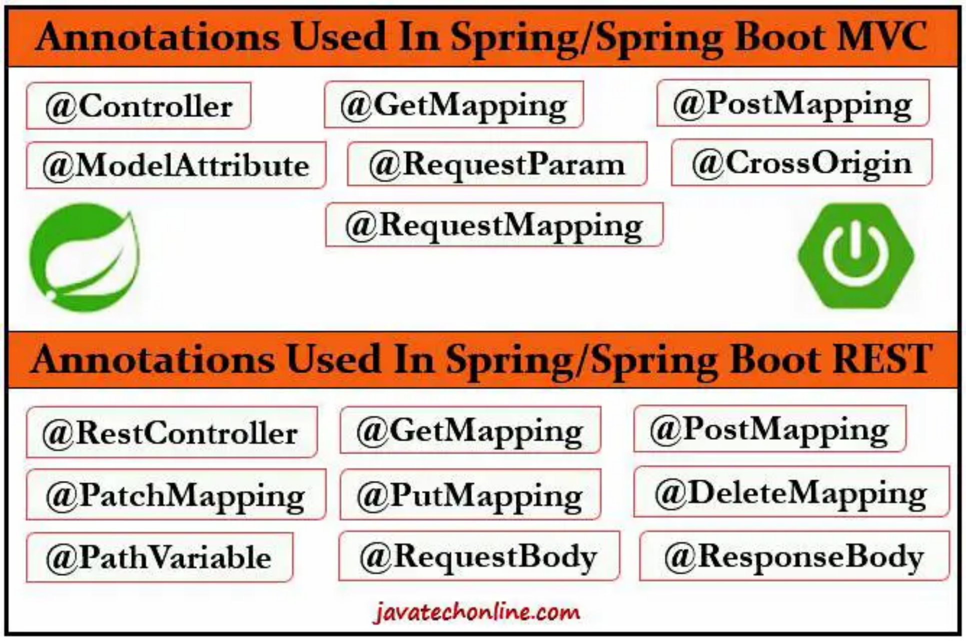

Table of Contents

TOP

Spring Boot MVC Annotations

We will start our first part of article 'Spring Boot MVC Spring Boot MVC REST Annotations With ExamplesREST Annotations With Examples' with Spring Boot MVC annotations. In order to use Spring Boot MVC annotations, make sure that you have the below dependency in your pom.xml.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>  spring-boot-starter-web   </artifactId>
</dependency>
```

@Controller

@Controller annotation comes under the Stereotype category of annotations that works as specialization of *@Component* annotation. This annotation tells the Spring IOC container to treat this class just as a Spring MVC controller. For example:

```
@Controller
public class MyMvcController { }
```

@RequestMapping

TOP

The second important annotation in Spring MVC is the *@RequestMapping*. We use this annotation to map client requests with the appropriate method to serve the request. Apart from Spring MVC, Spring WebFlux also supports this annotation. *@RequestMapping* annotation provides various options as its attributes to offer its customized behavior. Below is the list of attributes it provides.

value : represents the primary mapping which becomes a part of our URL. Most of the times, for a simple request, we use only this attribute.
path : Alias for value, supports Ant-style path patterns, relative paths at method level. Path mapping URIs may contain placeholders e.g. *"/${profile_path}"*
method : represents the HTTP request method. However, now-a-days we use *@GetMapping*, *@PostMapping* annotations as a replacement of this attribute.
params : represents the parameters of the mapped request, yet another way of narrowing the request mapping.
consumes : represents the consumable media types of the mapped request.
headers : represents the headers of the mapped request.
name : assigns a name to this mapping.
produces : represents the producible media types of the mapped request.

```
@Controller
public class HelloController {

    @RequestMapping(
        value = {"/hello"},
        params = {"id","name"},
        method = {RequestMethod.GET},
        consumes = {"text/plain"},
        produces = {"application/json","application/xml"},
        headers = {"name=Robert", "id=1"}
    )
    public String helloWorld() {
        return "Hello";
    }
}
```

We can use this annotation at both class level and method level. However, we generally use it only at class level because we have more handy modern annotations(available in Spring 4.3+) to use at method level such as *@GetMapping*, *@PostMapping* etc. However, it is recommended to use it at the class level if the application has multiple controllers to have a clear difference between URLs.

@GetMapping, @PostMapping, @PutMapping,

TOP

@PatchMapping, @DeleteMapping

@GetMapping annotation is the HTTP method 'GET' specific variant of the annotation *@RequestMapping*. It is a shortcut version of the '*@RequestMapping*' and applies on top of the method that expects HTTP 'GET' request. Let's look once at the API source code of annotation *@GetMapping*.

```
@Target({ java.lang.annotation.ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = { RequestMethod.GET })
public @interface GetMapping {
        String name( ) default "";
        String[ ] value() default {};
        String[ ] path() default {};
        String[ ] params() default {};
        String[ ] headers() default {};
        String[ ] consumes() default {};
        String[ ] produces() default {};
}
```

@GetMapping vs @RequestMapping

As source code of *@GetMapping* shows that it is already annotated with *@RequestMapping(method = { RequestMethod*.GET }). It indicates that we don't need to specify '*method = { RequestMethod.GET }*' when using *@GetMapping*. For example, below code demonstrates the concept:

```
@RequestMapping(value = "/emp/{empid}", method = RequestMethod.GET)

        is equivalent to ⇓

@GetMapping("/emp/{empid}")
```

Similarly, to handle different types of incoming HTTP requests, Spring supports five types of HTTP method specific annotations. They are *@GetMapping, @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping* for HTTP methods GET, POST, PUT, PATCH and DELETE respectively.

♥ **Note :** In Spring MVC applications, we generally use *@GetMapping* and *@PostMapping*. All other annotations, including both of them are useful in Spring REST applications.

TOP

@ModelAttribute

In Spring MVC , you will definitely have a form to fill the values and submit it on click of a provided button. Instead of parsing each field value of your form individually, you can populate all fields value at one go with the help of *@ModelAttribute* annotation. This is commonly known as data binding in Spring MVC. For example, the code below that follows the '*Invoice*' model attribute is populated with data from a form submitted to the saveInvoice endpoint. Spring MVC does this behind the scenes before invoking the submit method:

@ModelAttribute at method argument level

```
@PostMapping("/saveInvoice")
public String saveInvoice(@ModelAttribute("invoice") Invoice invoice) {
            // Code that uses the invoice object to save it
           // service.saveInvoice(invoice);
      return "invoiceListView";
}
```

@ModelAttribute at method level

Moreover, besides using *@ModelAttribute* at method argument, we can also use it at method level. When used at the method level, it adds one or more model attribute values in the Model that can be identified globally. It means, methods annotated with *@ModelAttribute* apply to all methods that are annotated with *@RequestMapping* or even *@GetMapping* and *@PostMapping*. However, our method annotated with *@ModelAttribute* will be the very first to run, before the rest of the *@RequestMapping* methods.

```
@ModelAttribute
public void includeAttributes(Model model) {
       model.addAttribute("message", "additional attribute to all methods");
}
```

In the example above, includeAttributes() method will add an attribute named 'message' to all models defined in the controller class. Generally, Spring MVC will always make first call to this method, prior to calling any other request handler methods.
♥ **Note:** It is also important to note that you need to annotate the respective controller class with @ControllerAdvice. Then only, you can add values in the Model that will be identified globally(in all request handler methods).

TOP

@CrossOrigin

We use *@CrossOrigin* annotation to get support for Cross-origin resource sharing(CORS). CORS is a W3C specification implemented by most of the browsers that allow you to specify what kind of cross domain requests are authorized in a flexible way. This is required because for security reasons, browsers don't permit AJAX calls to resources residing outside the current origin. Spring Framework 4.2 GA provides first class support for CORS out-of-the-box, offering us an easier and more robust way to configure it in a Spring or Spring Boot web application.

In short, The CORS mechanism supports secure cross-origin requests and data transfers between browsers and servers. @CrossOrigin annotation permits cross-origin requests on specific handler classes and/or handler methods. Both Spring Web MVC and Spring WebFlux support this annotation through their respective modules.

@CrossOrigin with method (annotated with @RequestMapping)

```
@RestController
@RequestMapping("/user")
public class UserController {

    @CrossOrigin
    @RequestMapping(method = RequestMethod.GET, path = "/{uid}")
    public User getUser(@PathVariable Integer uid) {
        ...
    }
}
```

In the example above, we applied @CrossOrigin at method level. Hence, this method will allow all origins, HTTP method 'GET' requests with the default value of the maximum age(30 minutes) of the cache duration for preflight responses.

@CrossOrigin with Controller class

```
@CrossOrigin(origins = "https://javatechonline.com")
@RestController
@RequestMapping("/user")
public class UserController {

    @RequestMapping(method = RequestMethod.GET, path = "/{uid}")
    public User getUser(@PathVariable Integer uid) {....}
```

TOP

```
    @RequestMapping(method = RequestMethod.DELETE, path = "/{uid}")
    public void deleteUser(@PathVariable Integer id){....}
}
```

Here, we applied @CrossOrigin at class level. Hence, both methods getUser() and delelteUser() will get qualification of cross-origin.

@CrossOrigin with both(Controller class and Handler Method)

```
@CrossOrigin(maxAge = 2400)
@RestController
@RequestMapping("/user")
public class UserController {

    @CrossOrigin(origins = "https://javatechonline.com")
    @RequestMapping(method = RequestMethod.GET, path = "/{uid}")
    public User getUser(@PathVariable Integer uid) {....}

    @RequestMapping(method = RequestMethod.DELETE, path = "/{uid}")
    public void deleteUser(@PathVariable Integer id) {....}
}
```

In the example above, both methods getUser() and delelteUser() will get a maxAgeof 2400 seconds. The method delelteUser() will permit all origins, but the method getUser() will permit single origin that is from 'https://javatechonline.com'.

@RequestParam

@RequestParam annotation binds the value of web request parameter with the value of Controller's method parameter. It acts just like getRequestParameter() method of HttpServletRequest. We send data to application using URL in the form of "URL?key=val". Below is the Syntax to use *@RequestParam*:

```
@RequestParam("key") DataType localVariable
             --OR--
@RequestParam DataType key
```

For example, below code demonstrates the concept:

TOP

Expected URL to access this method : ..../user?uid=5&uname=Robin

```
@GetMapping("/user")
public String getUserDetails(
                @RequestParam("uid") int id,
                @RequestParam(value = "uname", required = false, defaultValue = "Mary") String name
                )
{
        System.out.println("id is : "+id);
}
```

From the example above, it should be clear that how we can use multiple attributes also.

@RequestParam with multiple values of a field

Sometimes, it is recommended to declare variable as Array or List types. In such a case we need to supply multiple values to @RequestParam. Of course, we can compare it with getParametervalues() method of HttpServletRequest. Here is the example to assign multiple values to a field:

Expected URL to access this method : ..../user?subject=IT&subject=CS&subject=EC

```
@RequestParam("subject")String[] sub
                --OR--
@RequestParam("subject")List<String> sub
```

However, internally it will act as String[] sub = {"IT", "CS", "EC"};

Spring Boot REST Annotations

This section of our article 'Spring Boot MVC REST Annotations With Examples' is dedicated to Spring Boot REST annotations. All annotations which offers support for Spring MVC web applications are also applicable for Spring REST application. In this section, we will discuss about those annotations which are additional to REST applications. Furthermore, in order to learn how to create REST API in Spring Boot, kindly visit our article 'How to create CRUD REST API using Spring Boot ?'.

@RestController

We use @RestContoller annotation to tell Spring that this class is a controller for Spring REST applications. Moreover, @RestController(introduced in Spring 4.0) is a combination of two ann

TOP

: @Controller and @ResponseBody. It means we can use one annotation instead of using two for getting the same feature. Hence, if you use @Controller, you need to add @ResponseBody additionally to get features of REST API.

```
@RestController
@RequestMapping("/user")
public class UserRestController {

   @GetMapping("/getUser/{uid}")
   public User getUser(@PathVariable String uid) {....}
}
```

The code above indicates that the class UserRestController will act as a RestController for the application.

@PathVariable

We use *@PathVariable* to bind value of variable at URL path with request handler's method parameter. In *@RequestParam*, we were sending data via URL with query string (?) and then 'key=value'. In @PathVariable also, we can send multiple data separated by '/' but without using '?' and key-value. In order to read the dynamic value, variable in the URL path should be enclosed with curly braces such as "URL/user/{id}". Below is the syntax of using *@PathVariable*:

```
@PathVariable("key") DataType localVariable
                --OR--
@PathVariable DataType key
```

For example, below code demonstrates the use of *@PathVariable*:

```
@GetMapping("/user/{id}/{name}")
public String getUserDetails(
          @PathVariable Integer id,
          @PathVariable String name
          )
{
      return "Path variable data is: " + id + "-" + name;
}
```

TOP

In order to access getUserDetails(), your URL should be like http://localhost:8080/user/10/Robin

In Spring, method parameters annotated with *@PathVariable* are required by default. However, to make it optional, we can set the *required* property of *@PathVariable* to *false as below:*

@PathVariable(required = false) String name

Moreover, Since Spring 4.1, we can also use java.util.Optional<T> (introduced in Java 8) to handle a non-mandatory path variable as below:

```
@GetMapping("/user/{id}/{name}")
public String getUserByIdAndName(
        @PathVariable Optional<String> id,
        @PathVariable String name
        )
{
    if (id.isPresent()) {
          return "ID: " + id.get();
    } else {
         return "ID not present";
    }
}
```

@RequestParam vs @PathVariable

We generally use *@RequestParam* in form-based Spring MVC projects whereas *@PathVariables* in Spring REST applications. Almost all Web technologies in Java supports @RequestParam as it is a basic concept of Servlet API. But it is not true with *@PathVariable*. One major difference is that *@PathVarible* follows the order of variables whereas in *@RequestParam* order of variables doesn't matter.

URL pattern	Validity
/user?id=10&name=Robin	Valid
/user?name=Robin&id=10	Valid
/user/10/Robin	Valid
/user/Robin/10	Invalid : MethodArgumentTypeMismatchException(NumberFormatException) with "status": 400, "error": "Bad Request"

From the table above, it is clear that we have to take extra care while accessing a request handl

TOP

method in terms of variable order in *@PathVariable*.

As opposed to *@RequestParam*, URL created for *@PathVariable* is called as Clean URL and it also takes less characters to send data.

@ResponseBody

In Spring Boot, @ResponseBody, by default, converts return type data into JSON format in case of non-string return type data(e.g. Employee, Employee<String> etc.). Hence, if return type of request handler method is String, @Responsebody will not do any conversion of returning data. We don't need to apply this annotation explicitly as Spring will internally apply it by default when we annotate a class with @RestController.

@RequestBody

If we are sending input data in form of JSON/XML(Global data format) to a Spring Boot REST producer application via request handler method, then it will convert Global data into Object format(based on the provided type) and pass it to method argument. This annotation converts input data from JSON/XML format into Object. We provide this object as method parameter. In Spring Boot application, @RequestBody annotation does all this behind the scene. Below is the syntax to use @RequestBody:

```
@RequestBody ClassName objectName
```

For example, below code demonstrates the use of @RequestBody

```
@RestController
public class EmployeeRestController {

    @PostMapping("/save")
    public String saveEmp(@RequestBody Employee employee) {
            return employee.toString();
    }
}
```

Moreover, if we send invaild JSON/XML, like format is wrong, key-val are wrong, then spring boot throws : 400 BAD REQUEST.

@ResponseBody & @RequestBody for XML format da

TOP

As aforementioned Spring Boot by default works for JSON format data. To work with XML format data, add below dependency and provide header param as Accept : application/xml

```
<dependency>
<groupId>com.fasterxml.jackson.dataformat</groupId>
<artifactId>jackson-datafornmat-xml</artifactId>
</dependency>
```

If Producer REST application has no dependency for XML, still Request Header has Accept : application/xml, then you will get Http Status: 406 Not Acceptable.

If above XML dependency is not in place and trying to accept XML data using *@RequestBody*, then you will get Http Status : 415 Unsupported MediaTypes.

Spring Boot Annotations With Examples
February 20, 2021
In "Spring Boot"

Spring Boot Errors and AOP Annotations With Examples
February 20, 2021
In "Spring Boot"

Spring Security Annotations With Examples
August 4, 2022
In "java"

Tagged @Controller @controller annotation in spring @CrossOrigin @DeleteMapping @deletemapping example in spring boot @GetMapping @ModelAttribute @modelattribute example @modelattribute in spring @modelattribute in spring mvc @PatchMapping @PathVariable @pathvariable in spring @PostMapping @PutMapping @RequestBody @requestbody in spring mvc @requestbody in spring mvc example @requestbody vs @modelattribute @RequestMapping @requestmapping in spring @RequestParam @requestparam example @requestparam in spring @RequestParam vs @PathVariable @ResponseBody @ResponseBody & @RequestBody for XML format data @responsebody in spring @restcontroller in spring @restcontroller in spring boot annotations in spring mvc controller annotation controller in spring boot controller vs restcontroller difference between controller and rest controller difference between pathvariable and requestparam getmapping in spring boot getmapping spring boot getmapping vs requestmapping modelattribute in spring boot modelattribute vs requestparam mvc annotations in spring boot pathvariable in spring boot postmapping spring boot example putmapping spring boot request mapping annotations request mapping java spring requestmapping annotation in spring boot requestmapping in spring boot requestmapping vs getmapping requestparam in spring boot rest annotations in spring rest annotations in spring boot rest api annotations rest api annotations in spring boot rest controller in spring boot restcontroller annotation restcontroller annotation in spring boot restcontroller spring boot restcontroller vs controller restful annotations Spring Boot allowed HTTP methods spring boot annotations interview questions spring boot rest spring boot rest annotations spring b rest api annotations spring boot rest controller spring getmapping spring mvc annotations spring rest spring r

TOP