Spring Boot MVC REST 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.

@RequestMapping

We use this annotation to map client requests with the appropriate method to serve the request. *@RequestMapping* annotation provides various options as its attributes to offer its customized behavior.

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";
    }
}
```

@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

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

        is equivalent to ⇓

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

@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.

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) {
        ...
    }
}
```

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) {....}

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

Here, we applied @CrossOrigin at class level. Hence, both methods getUser() and deleteUser() 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 deleteUser() will get a maxAgeof 2400 seconds. The method deleteUser() 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*:

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);
}
```

@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”};

@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 annotations : @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}".

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;
}
```

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

@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 handler 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.

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.

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 REST

What is REST?

REST stands for Representational State Transfer. It transfers state(data) in global format(representational) between two different applications running on different servers. In the process of data transfer, who requests data is called Consumer/Client application and who provides data is called a producer application. REST is an architectural style that follows a set of rules to create webservices. Webservices provide reusable data to multiple applications and interoperability between them on the internet. Web services that conform to the REST architectural style, called RESTful Web services.

What is Rest Template ?

In a nutshell, RestTemplate is a predefined class in Spring Boot REST project. Moreover It helps in making HTTP calls to Producer application with all method types eg. GET, POST, PUT, DELETE etc. However Spring Boot framework doesn't auto configure this class. It also supports JSON/XML to Object and Object to JSON/XML auto-conversion. Moreover, it requires Endpoint details from a producer application like IP, PORT, Paths, Method Type, Input data format and Output data format etc. Additionally, RestTemplate provides the exchange() method to consume the web services for all HTTP methods. In fact, RestTemplate helps in making HTTP Rest Calls.

What is difference between getForObject() and getForEntity() ?

getForObject(url, T.class) : It retrieves an entity using HTTP GET method on the given URL and returns T. It doesn't return Status, Header params but only Response Body.

getForEntity(url, T.class) : It retrieves an entity by using HTTP GET method for the given URL and returns ResponseEntity<T>.

What is exchange() method in RestTemplate used for ?

exchange() method supports making call to any Http method (GET/POST/PUT/DELETE/…..). Generally, it has below syntax.

exchange(String url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType, Object... uriVariables): ResponseEntity<T>

url : Producer application URL (RestController's mrthod path)
HttpMethod : is an enum to provide method type.
HttpEntity : Request Body + HttpHeaders (it can also be null)
responseType : Class type of response
Object-var/args : used for sending multiple pathVariables

What all parameters are expected to write consumer methods in RestTemplate ?

In fact consumer method could be predictable on the already existing producer method. The producer method's url, return type, Http method type, path variables etc. will decide the structure of your consumer method. Generally we expect following things to pass as parameters to RestTemplate's methods.

1. URL of Producer webservice
2. Body of the request (in case of POST/PUT...)
3. Media Type like APPLICATION_JSON, APPLICATION_XML, APPLICATION_PDF etc. (in case of POST/PUT...)
4. Http Method type
5. Return type of producer method
6. Path Variables (If any)

```
String url = "http://localhost:8080/api/invoices";

String body = "{\"name\":\"INV11\",
\"amount\":234.11,\"number\":\"INVOICE11\",\"receivedDate\":\"28-10-2020\",\
"type\":\"Normal\",\"vendor\":\"ADHR001\",\"comments\" :\"On Hold\"}";


HttpHeaders       headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);

HttpEntity<String> request = new HttpEntity<String> (body,headers);

ResponseEntity<String> response = restTemplate.exchange(url,
HttpMethod.POST,request, String.class);
```

Response Body :

Status code value :

Status code :