**Spring REST Annotations – 2022**

**Difference between ResponseEntity and @ResponseBody**

It is an extension of **[HttpEntity](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpEntity.html" \t "class in org.springframework.http)**that adds an **[HttpStatus](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/HttpStatus.html" \t "enum in org.springframework.http)** status code.

Example

**@GetMapping(path = "/test")**

public **ResponseEntity<User>** handle() {

HttpHeaders responseHeaders = new HttpHeaders();

return **new ResponseEntity<>(user, responseHeaders, HttpStatus.OK)**;

or

return **new ResponseEntity<>(user, HttpStatus.OK)**;

}

Also, you can write  **return ResponseEntity.ok().headers(responseHeaders).body(user)**;

**HttpEntity** represents an HTTP **request** or **response** consists of **headers** and **body**.

In classic Spring MVC applications, endpoints usually return rendered HTML pages. In such cases, we can mark the request handler method with **@ResponseBody**, and **Spring treats the result value of the method as the HTTP response body** itself.

**Difference between @RestController and @Controller**

* **@Controller** is used to **mark classes as Spring MVC Controller**, returns a view in Spring web MVC
* **@RestController** is the combination of **@Controller** and **@ResponseBody**
* **If you use @RestController you cannot return a view** (By using Viewresolver in Spring/springboot) and yes **@ResponseBody is not needed in this case**.
* **@Controller** returns View. **@RestController** returns ResponseBody.

So the following two controller definitions should do the same.

@RestController

public class MyRestController { }

@Controller

@ResponseBody

public class MyController { }

**@Configuration:** **@Configuration** is an analogus for xml file. Such classes are sources of bean definitions by defining methods with the @Bean annotation.

**@Configuration is**:

* **not required**, if you already pass the annotated class in the sources parameter when calling the SpringApplication.run() method;
* **required**, when you don't pass the annotated class explicitly, but it's in the package that's specified in the @ComponentScan annotation of your main configuration class.

**@ComponentScan:** It is used to scan a package for beans.

**@ComponentScan**(basePackages **= {"com.ddlab.rnd.app1", " com.ddlab.rnd.app2"})**

**Difference Between @ComponentScan and @EnableAutoConfiguration**

**@ComponentScan scans for Spring components while @EnableAutoConfiguration is used for auto-configuring beans present in the classpath.**

**@EnableAutoConfiguration:** Can be used with or without @Configuration. **It tells Spring to setup some basic infrastructure judging by what you have in the classpat**h. It's done by invoking a so called import class that's derived from the value of the **@Import annotation that @EnableAutoConfiguration includes**. **Only one class should be annotated with @EnableAutoConfiguration**, duplicating it doesn't do anything.

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean.

**@Required:** The **@Required** annotation in spring is **a method-level annotation used in the setter method of a bean property and therefore making the setter-injection compulsory**. **Deprecated as of Spring 5.1, use @Autowired**.

**public** **class** Student {

**private** **int** rollNo;

**private** String name;

**private** **int** age;

    // Annotation

    @Required

    @Value("${student.rollNo}")

**public** **void** setRollNo(**int** rollNo) {

**this**.rollNo = rollNo;

    }

    @Value("${student.name}")

**public** **void** setName(String name) {

**this**.name = name;

    }

    // Setter

    @Value("${student.age}")

**public** **void** setAge(**int** age) {

**this**.age = age;

    }

**public** **void** display() {

        System.out.println("Roll No: " + rollNo);

        System.out.println("Name: " + name);

        System.out.println("Age: " + age);

    }

}

public class Student {

private Integer age;

private String name;

**@Required**

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

**@Required**

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

**@Value**: This annotation is used at the field, constructor parameter, and method parameter level. The @Value annotation indicates a default value expression for the field or parameter to initialize the property with. As the @Autowired annotation tells Spring to inject object into another when it loads your application context, you can also use @Value annotation to inject values from a property file into a bean’s attribute.

@Value("${servicenow.username}")

**private** String userName;

**application.properties**

servicenow.username=john

**@Bean:** It is a method-level annotation. It is an alternative of XML <bean> tag.

**@CrossOrigin:** We use @CrossOrigin annotation to get support for Cross-origin resource sharing(CORS).

**What is CORS** : “**CORS**” stands for **Cross-Origin Resource Sharing**. **It allows you to make requests from one website to another website in the browser**, which is normally **prohibited by another browser policy called the Same-Origin Policy (SOP)**.

**What is Stereotype?**

A stereotype is a generalized belief about a particular category of people. A *stereotype* is a preconceived notion, especially about a group of people.

In real world we know some stereotype, for example: **American like to drink coffee**. **British like to drink tea**. Of course, its not true for all American or British. **Its just an oversimplification of American or British people**.

In Spring, stereotype help us to simplify object creation. You don't need to define the relationship between Type, because you make stereotype of the Type.

**Note:**[**Type in Java**](https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html)**. Class is a Type**

**@Autowired**: This annotation is applied on fields, setter methods, and constructors. The @Autowired annotation injects object dependency implicitly.

**@Qualifier**: This annotation is used along with @Autowired annotation. When you need more control of the dependency injection process, @Qualifier can be used.

@Component

public class BeanB1 implements BeanInterface {

//

}

@Component

public class BeanB2 implements BeanInterface {

//

**}**

**@Lazy**: This annotation is used on component classes. By default all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use @Lazy annotation over the class. This means that the bean will be created and initialized only when it is first requested for. **By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context.**

**@Lazy**

@Lazy  
@Component  
public class Address {  
 public Address() {  
 System.*out*.println("Address constructor invoked");  
 }  
 public String getStreetName() {  
 return "some street name";  
 }  
}

@Data  
@Component  
**public class** Emp {  
  
 @Autowired @Lazy  
 **private** Address **adrs**;  
 **public void** showName() {  
 System.***out***.println(**"Emp Name: John"**);  
 }  
 **public void** showStreetName() {  
 System.***out***.println(**"Street Name: "**+**this**.**adrs**.getStreetName());  
 }  
}

**Test class**

@Autowired  
**private** Emp **emp**;  
  
@EventListener(ApplicationReadyEvent.**class**)  
**public void** onReady() {  
 **emp**.showName();  
 **emp**.showStreetName();  
}

**OUTPUT**

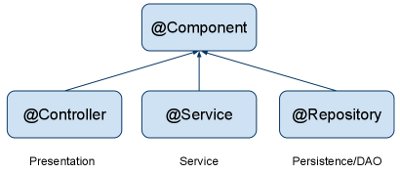
Emp Name: John

Address constructor invoked

**Street Name: some street name**

**Differences between @Component, @Repository, @Controller and @Service**

**@Service, @Controller, @Repository = {@Component + some more special functionality}**



| **Annotation** | **Meaning** |
| --- | --- |
| **@Component** | generic stereotype for any Spring-managed component |
| **@Repository** | **stereotype for persistence layer** |
| **@Service** | **stereotype for service layer** |
| **@Controller** | **stereotype for presentation** layer (spring-mvc) |

**@Component** is a generic stereotype for any Spring-managed component. **@Repository, @Service, and @Controller** are specializations of **@Component** for more specific use cases (in the persistence, service, and presentation layers, respectively).

For example, these stereotype annotations make **ideal targets for pointcuts. @Repository, @Service, and @Controller** can also carry additional semantics in future releases of the Spring Framework. **We can use one in place of another and can still get our way around.**

**@Component:** to indicate that the class is a spring component.

**@Repository:** to indicate that the class defines a data repository.

**@Controller**: to indicates that a particular class serves the role of a controller.

***What’s special about @Controller :*** We cannot switch this annotation with any other like **@Service** or **@Repository**, even though they look same. The dispatcher scans the classes annotated with @Controller and detects method annotated with **@RequestMapping**

**@Service**: holds the business logic and call methods in the repository layer.

***What’s special about @Service?: used*** to indicate, that it holds business logic.

@Service, @Controller and @Repository used based on their layering conventions. Hence, it's always a good idea to respect the convention and use it in line with layers.

**Spring Boot Annotations**

**@SpringBootApplication**: This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package.

The @SpringBootApplication is a convenient annotation that adds all the following:

**@Configuration**

**@EnableAutoConfiguration**

**@ComponentScan**

**@SpringBootApplication**: a combination of **@EnableAutoConfiguration** + **@ComponentScan** **+ @Configuration**.

**@ModelAttribute** refers to a property of the Model object (the M in MVC ;) so let's say we have a form with a form backing object that is called "Person" then you can have Spring MVC supply this object to a Controller method by using the **@ModelAttribute** annotation:

public String processForm(**@ModelAttribute**("person") Person person){

person.getStuff();

}

On the other hand the annotation is used to define objects which should be part of a Model. So if you want to have a Person object referenced in the Model you can use the following method:

**@ModelAttribute("person")**

public Person getPerson(){

return new Person();

}

This annotated method will allow access to the Person object in your View, since it gets automatically added to the Models by Spring.

**Spring MVC and REST Annotations**

**@RequestMapping**: This annotation is used both at class and method level. The @RequestMapping annotation is used to map web requests onto specific handler classes and handler methods.

@RequestMapping(value = "/ex/foos", method = RequestMethod.GET)

@ResponseBody

public String getFoosBySimplePath() {

return "Get some Foos";

}

@RequestMapping(value = "/ex/foos", headers = "key=val", RequestMethod.GET)

@ResponseBody

public String getFoosWithHeader() {

return "Get some Foos with Header";

}

**@GetMapping**: This annotation is used for mapping HTTP GET requests onto specific handler methods.

@GetMapping("users/{id}")

public ResponseEntity<User> getById(@PathVariable long id) {

return new ResponseEntity<>(user.get(), HttpStatus.OK);

}

**@PostMapping**: This annotation is used for mapping HTTP POST requests onto specific handler methods.

a shortcut for @RequestMapping(method = RequestMethod.POST)

@PostMapping(path = "users", consumes = MediaType.APPLICATION\_JSON\_VALUE,

produces = MediaType.APPLICATION\_JSON\_VALUE)

public ResponseEntity<User> create(@RequestBody User newUser) {

User user = userService.save(newUser);

return new ResponseEntity<>(user, HttpStatus.CREATED);

}

**@PutMapping**: This annotation is used for mapping HTTP PUT requests onto specific handler methods.

@PutMapping("/employees/{id}")

public ResponseEntity<Employee> updateEmployee(@PathVariable(value = "id") Long employeeId) {

Employee employee = .....

return ResponseEntity.ok(updatedEmployee);

}

**@PatchMapping**: This annotation is used for mapping HTTP PATCH requests onto specific handler methods.

@PatchMapping("/employees/{id}/{firstName}")

public ResponseEntity<Employee> updateEmployeePartially(@PathVariable Long id, @PathVariable String firstName) {

Employee employee = ....

return new ResponseEntity<Employee>(employeeRepository.save(employee), HttpStatus.OK);

}

**@DeleteMapping**: This annotation is used for mapping HTTP DELETE requests onto specific handler methods.

@DeleteMapping(value = "/posts/{id}")

public ResponseEntity<Long> deletePost(@PathVariable Long id) {

return new ResponseEntity<>(id, HttpStatus.OK);

}

**@CookieValue**: This annotation is used at method parameter level. @CookieValue is used as argument of request mapping method.

@RequestMapping("test2")

public String handleRequest (@CookieValue(value = "testCookie", defaultValue = "defaultCookieValue")

String cookieValue, Model model) {

model.addAttribute("cookieValue", cookieValue);

return "my-page2";

}

**@CrossOrigin**: This annotation is used both at class and method level to enable cross origin requests.

@CrossOrigin(origins = "http://localhost:8080")

@GetMapping("/greeting")

public Greeting greeting(@RequestParam(required = false, defaultValue = "World") String name) {

return new Greeting(counter.incrementAndGet(), String.format(template, name));

}

or

@CrossOrigin(origins = "http://example.com", maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@RequestMapping(method = RequestMethod.GET, path = "/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@RequestMapping(method = RequestMethod.DELETE, path = "/{id}")

public void remove(@PathVariable Long id) {

// ...

}

}

**@ExceptionHandler**: This annotation is used at method levels to handle exception at the controller level.

**How do you handle exception with http status code in Spring**

Quite good way is to use exceptions and class for handling called @ControllerAdvice:

**@ControllerAdvice**

**class GlobalControllerExceptionHandler {**

**@ResponseStatus(HttpStatus.CONFLICT) // 409**

**@ExceptionHandler(DataIntegrityViolationException.class)**

**public void handleConflict() {**

**// Nothing to do**

**}**

**//Specific Examples given below.**

**@ResponseStatus**(HttpStatus.***BAD\_REQUEST***)  
 **@ExceptionHandler**(NullPointerException.**class**)  
 **public void** handleNullValue() {  
 System.***out***.println(**"It is a null value ..."**);  
 }

**@ExceptionHandler**(NullPointerException.**class**)  
 **public** ResponseEntity<String> handleNull(Exception ex) {  
 System.***out***.println(**"Exception details: "**+ex.getMessage());  
 **return new** ResponseEntity<>(ex.getMessage(),HttpStatus.***BAD\_REQUEST***);  
 }

**}**

**@RestControllerAdvice**: This annotation is applied on Java classes. **@RestControllerAdvice** is a convenience annotation which combines **@ControllerAdvice** and **@ResponseBody**. This annotation is used along with the @ExceptionHandler annotation to handle exceptions that occur within the controller

**@RestControllerAdvice is just a syntactic sugar for @ControllerAdvice + @ResponseBody**

**@RestControler = @Controller + @ResponseBody**

**@RestControllerAdvice = @ControllerAdvice + @ResponseBody**.

**@ResponseStatus**: This annotation is used on methods and exception classes. @ResponseStatus marks a method or exception class with a status code and a reason that must be returned.

@RestControllerAdvice

public class WebRestControllerAdvice {

@ExceptionHandler(CustomNotFoundException.class)

public ResponseMsg handleNotFoundException(CustomNotFoundException ex) {

ResponseMsg responseMsg = new ResponseMsg(ex.getMessage());

return responseMsg;

}

}

**@InitBinder**: WebDataBinder works as a preprocessor for each request coming to the controller.

To customize request parameter data binding, we can use @InitBinder annotated methods within our controller.

It can be used to register custom formatter, validators and PropertyEditors.

WebDataBinder.addCustomFormatter(..);

WebDataBinder.addValidators(..);

WebDataBinder.registerCustomEditor(..);

The @InitBinder annotated methods will get called on each HTTP request if we don't specify the 'value' element of this annotation. Each time this method is called a new instance of WebDataBinder is passed to it.

To be more specific about which objects our InitBinder method applies to, we can supply 'value' element of the annotation @InitBinder. The 'value' element is a single or multiple names of command/form attributes and/or request parameters that this init-binder method is supposed to apply to.

@InitBinder("user")

public void customizeBinding (WebDataBinder binder) {...}

@Controller

@RequestMapping("/register")

public class UserRegistrationController {

@Autowired

private UserService userService;

@InitBinder("user")

public void customizeBinding (WebDataBinder binder) {

SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-dd");

dateFormatter.setLenient(false);

binder.registerCustomEditor(Date.class, "dateOfBirth",

new CustomDateEditor(dateFormatter, true));

}

@RequestMapping(method = RequestMethod.GET)

public String handleGetRequest (Model model) {

model.addAttribute("user", new User());

return "user-registration";

}

}

## **Using multiple InitBinder methods**

@InitBinder(value = "student")

void initStudentValidator(WebDataBinder binder) {

binder.setValidator(new StudentValidator());

}

@InitBinder(value = "address")

void initAddressValidator(WebDataBinder binder) {

binder.setValidator(new AddressValidator());

}

**@RequestParam:** it binds the value of web request parameter with the value of Controller’s method parameter.

We send data to application using URL in the form of “**URL?key=val**“.

**@RequestParam with multiple values of 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”};

@GetMapping("/api/foos")

@ResponseBody

public String getFoos(@RequestParam String id) {

return "ID: " + id;

} 🡺 http://localhost:8080/spring-mvc-basics/api/foos?id=abc

**@PathVariable:** to bind value of variable at URL path with request handler’s method parameter.

“**URL/user/{id}“**

**@PathVariable("key") DataType localVariable**

--OR--

**@PathVariable DataType key**

**@GetMapping("/api/employees/{id}")**

**@ResponseBody**

**public String getEmployeesById(@PathVariable String id) {**

**return "ID: " + id;**

**} 🡺 http://localhost:8080/api/employees/111**

**@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. In @RequestBody is annotated with object model and data can be accepted with different types of Content-Type like application/json or application/xml.

**@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**.

@GetMapping("/greeting")

public ResponseEntity<String> greeting(@RequestHeader(HttpHeaders.ACCEPT\_LANGUAGE) String language) {

// code that uses the language variable

return new ResponseEntity<String>(greeting, HttpStatus.OK);

}

@GetMapping("/double")

public ResponseEntity<String> doubleNumber(@RequestHeader("my-number") int myNumber) {

return new ResponseEntity<String>(String.format("%d \* 2 = %d",

myNumber, (myNumber \* 2)), HttpStatus.OK);

}

@GetMapping("/listHeaders")

public ResponseEntity<String> listAllHeaders(

@RequestHeader Map<String, String> headers) {

headers.forEach((key, value) -> {

LOG.info(String.format("Header '%s' = %s", key, value));

});

return new ResponseEntity<String>(String.format("Listed %d headers", headers.size()), HttpStatus.OK);

}

If we use a Map and one of the headers has more than one value, we'll get only the first value. This is the equivalent of using the getFirst method on a MultiValueMap.

If our headers may have multiple values, we can get them as a MultiValueMap:

@GetMapping("/multiValue")

public ResponseEntity<String> multiValue(

@RequestHeader MultiValueMap<String, String> headers) {

headers.forEach((key, value) -> {

LOG.info(String.format(

"Header '%s' = %s", key, value.stream().collect(Collectors.joining("|"))));

});

return new ResponseEntity<String>(

String.format("Listed %d headers", headers.size()), HttpStatus.OK);

}

**@MatrixVariable**

**Partial Binding:** matrix variables can be used in a variety of different ways. For example, we can get each variable from each path segment. Consider the following request:

http://localhost:8080/spring-mvc-java-2/ companyData/**id=2;name=Xpto/employeeData/id=1;name=John; contactNumber=2200112334**

If we only want to know the matrix variable *name* of the *companyData* segment, then, we should use as an input parameter the following:

**@MatrixVariable(value="name", pathVar="company") String name**

***If the application uses Spring Security, then*StrictHttpFirewall*is used by default***. This blocks requests that appear to be malicious, including Matrix Variables with semicolon separator.

**@MatrixVariable Map<String, String> matrixVars**

**StrictHttpFirewall is the default and stricter implementation of HttpFirewall.**In contrast, unlike DefaultHttpFirewall, [StrictHttpFirewall](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/firewall/StrictHttpFirewall.html) rejects any un-normalized URLs providing more stringent protection.

@Bean **public** HttpFirewall **getHttpFirewall**() {

**StrictHttpFirewall** strictHttpFirewall = **new** **StrictHttpFirewall**(); strictHttpFirewall.setAllowSemicolon(true);

**return** strictHttpFirewall;

}

@GetMapping("employeeData/{employee}")

@ResponseBody

public ResponseEntity<Map<String, String>> getEmployeeData(

@MatrixVariable Map<String, String> matrixVars) {

return new ResponseEntity<>(matrixVars, HttpStatus.OK);

}

http://localhost:8080/spring-mvc-java-2/employeeData/id=1;name=John;contactNumber=2200112334

**@Mappings and @Mapping**

This annotation is used on fields. The @Mapping annotation is a meta annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field and to do that you have to add the @Mappings annotation.

**@RequestAttribute**: This annotation is used to bind the request attribute to a handler method parameter.

**@RequestBody**: This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body.

**@RequestPart**: This annotation is used to annotate request handler method arguments. The @RequestPart annotation can be used instead of @RequestParam to get the content of a specific multipart and bind to the method argument annotated with @RequestPart. This annotation takes into consideration the “Content-Type” header in the multipart(request part).

@PostMapping(path = "/resume", consumes = { MediaType.***MULTIPART\_FORM\_DATA\_VALUE*** })

**public** ResponseEntity<String> uploadResume(

**@RequestPart("profileAsString") String profileStr,**

**@RequestPart("file") MultipartFile document**) {

**return** **new** ResponseEntity<>("Your resume submitted...", HttpStatus.***CREATED***);

}

**@ResponseBody**: This annotation is used to annotate request handler methods. The @ResponseBody annotation is similar to the @RequestBody annotation. The @ResponseBody annotation indicates that the result type should be written straight in the response body in whatever format you specify like JSON or XML.

**Spring Boot Initialization Steps :** Here are the steps:

1. SpringApplication.run() creates **EmbeddedWebApplicationContext** application context;
2. Calls **its refresh()** method;
3. Refresh process reads annotations of the starting class TestSpring. It looks for **import** annotations. **EnableAutoConfiguration** is one of them. For an import annotation the refresh process gets the corresponding class from the annotation value and invokes its **selectImports()** method;
4. In case of @EnableAutoConfiguration the corresponding class is **EnableAutoConfigurationImportSelector** whose **selectImports() loads tons of other import** selectors from the META-INF/spring.factories;
5. This process continues recursively. Also, all bean definitions, that are inside these import selectors, are read. It includes beans defined by a method with the **@Bean** annotation.
6. The **resfresh()** continues and **reaches onRefresh()**, the **createEmbeddedServletContainer() method** is called inside;
7. Among read bean defitions at the previous step, beans implementing **ServletContextInitializer** are searched for and instantiated. One of them is the bean, defined by the **DispatcherServletAutoConfiguration**.**DispatcherServletRegistrationConfiguration#dispatcherServletRegistration()** method of ServletRegistrationBean type that extends ServletContextInitializer. As you can guess from the name of the class, such initializers add a given servlet (in this case **DispatcherServlet**) to a given ServletContext, when their onStartup() method is invoked;
8. A tomcat embedded server is created (not started completely yet). All found **ServletContextInitializers** at the previous step are passed to this tomcat initialization - **this is where the onStartup() methods of those ServletContextInitializers are called and DispatcherServlet gets created and registered as servlet**;
9. End of onRefresh() of application context;
10. The finishRefresh() is called where tomcat is finally started by TomcatEmbeddedServletContainer.start();
11. End of refresh() of application context and other final initialization steps;
12. The app is running.

**What is Hypermedia:** Hypermedia, an extension of the term hypertext, is a nonlinear medium of information that includes graphics, audio, video, plain text and hyperlinks.

**HATEOAS**

HATEOAS acronyms for **Hypermedia as the Engine of Application State**.

**Generally speaking, the principle implies that the API should guide the client through the application by returning relevant information about the next potential steps, along with each response.**

**HATEOAS, a client interacts with a network application, whose application server provides information dynamically through Hypermedia**.

Think about how your browser interacts with a site like [Wikipedia](https://en.wikipedia.org/wiki/HATEOAS). It doesn’t have any special Wikipedia-specific code – it knows how to render HTML and CSS, and that’s about it (relatively speaking). Everything the browser needs to know about what it can do next is included in the document itself!