Spring Boot Annotations

Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide supplemental information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

@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

```
@Required
public void setCost(Integer cost)
{  this.cost = cost;  }
```

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

@Configuration: It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

@ComponentScan: 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.

```
@ComponentScan(basePackages = "com.javatpoint")
@Configuration
public class ScanComponent
{
```

@Bean: 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.

```
@Bean
public BeanExample beanExample()
{  return new BeanExample ();  }
```

@Component: It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with @Component is found during the classpath. The Spring Framework pick it up and configure it in the application context as a Spring Bean.

@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
@RequestMapping("books")
public class BooksController
{
```

@Service: It is also used at class level. It tells the Spring that class contains the business logic.

```
@Service
public class TestService
{
```

@Repository: It is a class-level annotation. The repository is a DAOs (Data Access Object) that access the database directly. The repository does all the operations related to the database.

```
@Repository
public class TestRepository
{
```

@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.

@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.*

```
@Controller
public class BooksController
{
@RequestMapping("/computer-science/books")
public String getAllBooks(Model model)
{
return "bookList";
}
```

@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.

```
@PostMapping("/request")
public ResponseEntity postController(@RequestBody LoginForm 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.

```
@PostMapping(value = "/response", produces = application/JSON)
@ResponseBody
 public ResponseTransfer postController(@RequestBody LoginForm loginForm) {
        return new ResponseTransfer("Thanks For Posting!!!");
    }
```

We don't need to annotate the @RestController-annotated controllers with the @ResponseBody annotation since Spring does it by default.

@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.

```
@GetMapping("/api/employees/{id}")
@ResponseBody
public String getEmployeesById(@PathVariable String id) {
    return "ID: " + id;
}
```

Below is an example for Specifying the Path Variable Name

```
@GetMapping("/api/employeeswithvariable/{id}")
@ResponseBody
public String getEmployeesByIdWithVariableName(@PathVariable("id") String employeeId) {
    return "ID: " + employeeId;
}
```

We can also define the path variable name as @PathVariable(value="id") instead of PathVariable("id")

Multiple Path Variables in a Single Request

```
@GetMapping("/api/employees/{id}/{name}")
@ResponseBody
public String getEmployeesByIdAndName(@PathVariable String id, @PathVariable String name) {
    return "ID: " + id + ", name: " + name;
}
```

We can handle more than one @PathVariable parameter using a method parameter of type java.util.Map<String, String>

```
@GetMapping("/api/employeeswithmapvariable/{id}/{name}")
@ResponseBody
public String getEmployeesByIdAndNameWithMapVariable(@PathVariable Map<String, String> pathVarsMap) {
    String id = pathVarsMap.get("id");
    String name = pathVarsMap.get("name");
        return "ID: " + id + ", name: " + name;
    }
}
```

Optional Path Variables

```
@GetMapping(value = { "/api/employees", "/api/employees/{id}" })
@ResponseBody
public String getEmployeesById (@PathVariable (required = false) String id) {
    return "ID: " + id;
}
```

There isn't a provision to define a default value for method parameters annotated with @PathVariable.

@RequestParam: We can use @RequestParam to extract query parameters, form parameters, and even files from the request.

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam String id) {
    return "ID: " + id;
}
```

Specifying the Request Parameter Name

```
@PostMapping("/api/foos")
@ResponseBody
public String addFoo(@RequestParam(name = "id") String fooId, @RequestParam
String name) {
    return "ID: " + fooId + " Name: " + name;
}
```

Optional Request Parameters

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam(required = false) String id) {
```

```
    return "ID: " + id;

}
```

@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.