



Spring Annotations



Spring Annotations

- **@GetMapping**: It is used to create a web service endpoint with HTTP GET mapping that fetches It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
- **@PostMapping**: It is used to create a web service endpoint with HTTP POST mapping that creates It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
- **@PutMapping**: It is used to create a web service endpoint with HTTP PUT mapping that creates or updates It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
- **@DeleteMapping**: It is used to create a web service endpoint HTTP DELETE mapping that deletes a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**

Spring Annotations

- **@PatchMapping**: It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)** on the specific handler method
- **@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.
- **@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.
- **@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.

Spring Annotations

- **@RequestParam**: It is used to extract the query parameters from 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.
- **@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 times 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.

@RequestParam

```
@GetMapping("/echo")
```

```
//http://localhost:8080/echo?text=Good Morning
```

// We can pass the name of the url param we want as an argument to the RequestParam annotation. The value will be stored in the annotated variable

```
public String echo(@RequestParam(name = "text") String echoText) {
```

// The response will be "Echo: " followed by the param that was passed in

```
return "Echo: " + echoText;
```

```
}
```

@PathVariable

```
@GetMapping("/echo/{text}")
```

```
// http://localhost:8080/echo/Good Evening
```

```
// The PathVariable annotation assigns the text from the actual  
request to the `echoText` argument
```

```
public String echoPath(@PathVariable(name = "text") String  
echoText) {  
    return "Echo in path: " + echoText;  
}
```

Reading Headers

```
@GetMapping("/echo/headers")
public String echoHeaders(@RequestHeader Map<String, String> headers) {
    // Create a new StringBuilder to print the list of headers
    StringBuilder sb = new StringBuilder("Headers: \n");
    // loop through each header key-value pair
    for (Entry<String, String> header : headers.entrySet()) {
        // Add the header to the string in "key:value" format
        sb.append(header.getKey());
        sb.append(":");
        sb.append(header.getValue());
        sb.append("\n");
    }
    // return the completed string
    return sb.toString();
}
```

HTTP Servlet Response

```
@GetMapping("/custom-header")
```

```
// the HttpServletResponse argument is injected by the Spring  
framework
```

```
public String setCustomHeader(HttpServletResponse response) {  
    response.setHeader("X-Custom-Header", "Some-Custom-Value");  
    return "ok";  
}
```


HTTP Status Code

```
@GetMapping("/sample-error")
public String sampleError(HttpServletResponse response) {
    // We can only set the status once. Here we set it to a 500, or internal server
    // error status
    response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
    // Along with the status code, we can return a plaintext body as the response
    return "error";
}
```

@ExceptionHandler

- ***HandlerExceptionResolver*** implementation manages unexpected exceptions.
- ***HandlerExceptionResolver*** looks very similar to the exception mappings of web.xml.
- For example, when an exception was thrown, which handler was executing ?-kind of information is provided by ***HandlerExceptionResolver*** .
- Implementation of both interfaces ***HandlerExceptionResolver*** and ***SimpleMappingExceptionResolver*** will allow you , declaratively, to map Exceptions to specific views with some nonmandatory Java logic.
- The **@ExceptionHandler** annotation should be used on methods which are called to handle exception. It can be defined locally inside an **@Controller** or within an **@ControllerAdvice** class to apply it globally to all **@RequestMapping** methods

@ExceptionHandler

```
@ExceptionHandler(IOException.class)
```

```
public ResponseEntity<String> handleIOException(IOException ex) {
```

```
// prepare ResponseEntity
```

```
return ResponseEntity;
```

```
}
```

Exceptions With @ResponseStatus

- The **@ResponseStatus** annotation is annotated on business exceptions.
- As the exception occurs the **ResponseStatusExceptionHandler** handles it by setting the status of the response according to the exception.
- The **ResponseStatusExceptionHandler** is by default registered by the **DispatcherServlet**.

@ResponseStatus

- **@ResponseStatus** marks a method or exception class with the status code and reason message that should be returned. The status code is applied to the HTTP response when the handler method is invoked, or whenever the specified exception is thrown. It overrides status information set by other means, like ResponseEntity or redirect.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
```

```
public class ResourceNotFoundException extends RuntimeException {
```

```
    private static final long serialVersionUID = 1L;
```

```
    private String message;
```

```
    public ResourceNotFoundException(String message) {
```

```
        this.message = message;
```

```
    }
```

```
    public String getMessage() {
```

```
        return message;
```

```
    }
```

```
    public void setMessage(String message) {
```

```
        this.message = message;
```

```
    }}
```

@RestController

@ResponseStatus

```
public class UserController {
```

```
    @GetMapping("/users/{id}")
```

```
    public ResponseEntity < User > getUser(
```

```
        @PathVariable(value = "id") Integer userId) throws ResourceNotFoundException {
```

```
        Map < Integer, User > map = new HashMap < > ();
```

```
        map.put(1, new User(1, "Ramesh"));
```

```
        map.put(2, new User(2, "Tony"));
```

```
        map.put(3, new User(3, "Tom"));
```

```
        if (!map.containsKey(userId)) {
```

```
            throw new ResourceNotFoundException("Resource not found for " + userId);
```

```
        }
```

```
        return ResponseEntity.ok(map.get(userId));
```

```
    }
```

Spring Boot REST Web Services Content Negotiation

- REST resources can have multiple presentations (e.g. JSON or XML) as different clients can request different representation.
- The mechanism for selecting a correct representation is known as content negotiation.
- Content negotiation allows clients to request specific content type(s) to be returned by the server.
- With content negotiation, we enable a single endpoint to support different types of resource representations

Content Negotiation Strategies

- Content Negotiation can be done in following ways
- ***Using Path Extension*** This has the highest preference. In the request we specify the required response type using the extension like .json,.xml or .txt.
- ***Using url parameter*** This has the second highest preference. In the request we specify the required response type using the url parameter like format=xml or format=json.
- ***Using Accept Headers*** When making a request using HTTP we specify required response by setting the Accept header property.

Using Path Extension

```
Logger logger = LoggerFactory.getLogger(BooksController.class);
```

```
@GetMapping("/all")
```

```
public ResponseEntity<List<Book>> getAll() {
```

```
    logger.info(">>>");
```

```
    List<Book> books = new ArrayList<>();
```

```
    books.add(new Book(1, "Spring Microservices in Action", "John Carnell", 2525.00));
```

```
    books.add(new Book(2, "Learning Spring Boot 2.0", "Greg L. Turnquist", 2879.00));
```

```
    return ResponseEntity.ok().body(books);
```

```
}
```

```
@GetMapping(value = "/book/{id}")
```

```
public ResponseEntity<Book> getBookById(@PathVariable long id) {
```

```
    return ResponseEntity.ok().body(new Book(1, "Spring Microservices in Action", "John Carnell",  
    2525.00));
```

```
}
```

Url Parameter Strategy

- Add below Configuration class and overwrite configure Content Negotiation method.
- We can enable this strategy by setting the value of the favorParameter property to true
- `http://localhost:8080/books/all?mediaType=xml`
- `http://localhost:8080/books/all?mediaType=json`
- `http://localhost:8080/books/book/1?mediaType=json`
-

Url Parameter Strategy

@Configuration

```
public class WebConfig implements WebMvcConfigurer {
```

@Override

```
public void configureContentNegotiation(final ContentNegotiationConfigurer  
configurer) {
```

```
    configurer.favorPathExtension(true).favorParameter(true).parameterName("mediaT  
ype").ignoreAcceptHeader(false)
```

```
    .useRegisteredExtensionsOnly(false).defaultContentType(MediaType.APPLICATION  
JSON)
```

```
    .mediaType("xml", MediaType.APPLICATION_XML).mediaType("json",  
    MediaType.APPLICATION_JSON);
```

```
}
```

```
}
```

Accept Headers strategy:

If the Accept header is enabled, Spring MVC will look for its value in the incoming request to determine the representation type.

set `ignoreAcceptHeader` value to false to enable this approach.

with Accept — `application/json`


Accept Headers strategy:

GET localhost:8080/books/book/1

Authorization Headers (1) Body Pre-request Script Tests

	Key	Value	Description
<input checked="" type="checkbox"/>	Accept	application/xml	
	New key	Value	Description

Body Cookies Headers (5) Test Results

Pretty Raw Preview XML 

```
1 <Book>
2   <id>1</id>
3   <name>Spring Microservices in Action</name>
4   <author>John Carnell</author>
5   <price>2525.0</price>
6 </Book>
```

Accept Headers strategy:

The screenshot shows a REST client interface with the following components:

- Method and URL:** GET `localhost:8080/books/book/1`
- Headers Tab:** Contains one header: `Accept: application/json`. A "New key" section is also visible.
- Body Tab:** Shows the response body in JSON format:

```
{
  "id": 1,
  "name": "Spring Microservices in Action",
  "author": "John Carnell",
  "price": 2525
}
```

Decorative yellow hexagons are present in the bottom-left and bottom-right corners of the slide.