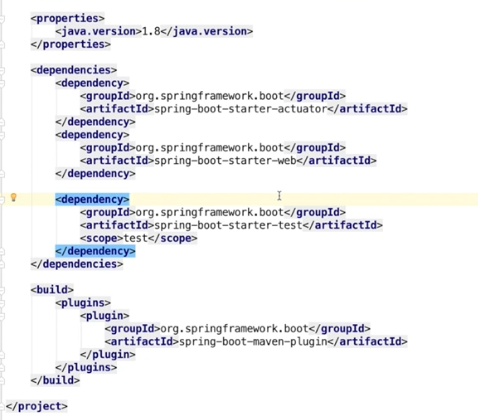
***SpringBoot***

***1st.A simple project***

***This project we use web and actuator dependencies.***

***First we check the pom file***



***Ok we have 2 that we added web and actuator***

***Why starter ?***

***When added like this spring knows sub dependencies and add them***

***You can check by***

***mvn dependency:tree***



*Basic Structure of spring app*

***Controller-Responsible to handle the traffic and also this is the last layer we meet so we have to handle exception here.***

***Service - Handle the business logic ,sometimes states (sessions)***

***Repository-Deal with Databases***

*Start Spring Boot Application*

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

* @EnableAutoConfiguration: enable [Spring Boot’s auto-configuration mechanism](https://docs.spring.io/spring-boot/docs/2.1.14.BUILD-SNAPSHOT/reference/html/using-boot-auto-configuration.html)
* @ComponentScan: enable @Component scan on the package where the application is located (see [the best practices](https://docs.spring.io/spring-boot/docs/2.1.14.BUILD-SNAPSHOT/reference/html/using-boot-structuring-your-code.html))
* @Configuration: allow to register extra beans in the context or import additional configuration classes

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes, as shown in the following example:

**package** com.example.myapplication;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

*@SpringBootApplication* *// same as @Configuration @EnableAutoConfiguration @ComponentScan*

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

}

*Controller*

***@RestController***

***https://www.baeldung.com/spring-request-param***

***Remember, we don't need to annotate the***

***@RestController-annotated controllers with the @ResponseBody annotation since Spring does it by default.***

<https://www.baeldung.com/spring-requestmapping>

## **2. @**RequestMapping**Basics**

Let's start with a simple example – mapping an HTTP request to a method using some basic criteria.

### **2.1.**@RequestMapping**– by Path**

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value = "/ex/foos", method = RequestMethod.GET)  @ResponseBody  public String getFoosBySimplePath() {      return "Get some Foos";  } |

To test out this mapping with a simple curl command, run:

|  |  |
| --- | --- |
| 1 | curl -i http://localhost:8080/spring-rest/ex/foos |

### **2.2.**@RequestMapping**– the HTTP Method**

The HTTP method parameter has **no default** – so if we don't specify a value, it's going to map to any HTTP request.

Here's a simple example, similar to the previous one – but this time mapped to an HTTP POST request:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value = "/ex/foos", method = POST)  @ResponseBody  public String postFoos() {      return "Post some Foos";  } |

To test the POST via a curl command:

|  |  |
| --- | --- |
| 1 | curl -i -X POST http://localhost:8080/spring-rest/ex/foos |

## **3.**RequestMapping**and HTTP Headers**

### **3.1.**@RequestMapping**With the**headers**Attribute**

The mapping can be narrowed even further by specifying a header for the request:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value = "/ex/foos", headers = "key=val", method = GET)  @ResponseBody  public String getFoosWithHeader() {      return "Get some Foos with Header";  } |

To test the operation, we're going to use the curl header support:

|  |  |
| --- | --- |
| 1 | curl -i -H "key:val" http://localhost:8080/spring-rest/ex/foos |

And even multiple headers via the header attribute of @RequestMapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RequestMapping(    value = "/ex/foos",    headers = { "key1=val1", "key2=val2" }, method = GET)  @ResponseBody  public String getFoosWithHeaders() {      return "Get some Foos with Header";  } |

We can test this with the command:

|  |  |
| --- | --- |
| 1 | curl -i -H "key1:val1" -H "key2:val2" http://localhost:8080/spring-rest/ex/foos |

Note that for the curl syntax for separating the header key and the header value is a colon, same as in the HTTP spec, while in Spring the equals sign is used.

### **3.2.**@RequestMapping**Consumes and Produces**

Mapping **media types produced by a controller** method is worth special attention – we can map a request based on its Accept header via the @RequestMapping headers attribute introduced above:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @RequestMapping(    value = "/ex/foos",    method = GET,    headers = "Accept=application/json")  @ResponseBody  public String getFoosAsJsonFromBrowser() {      return "Get some Foos with Header Old";  } |

The matching for this way of defining the Accept header is flexible – it uses contains instead of equals, so a request such as the following would still map correctly:

|  |  |
| --- | --- |
| 1  2 | curl -H "Accept:application/json,text/html"    http://localhost:8080/spring-rest/ex/foos |

Starting with Spring 3.1, the **@RequestMapping annotation now has the produces and the consumes attributes**, specifically for this purpose:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @RequestMapping(    value = "/ex/foos",    method = RequestMethod.GET,    produces = "application/json"  )  @ResponseBody  public String getFoosAsJsonFromREST() {      return "Get some Foos with Header New";  } |

Also, the old type of mapping with the headers attribute will automatically be converted to the new produces mechanism starting with Spring 3.1, so the results will be identical.

This is consumed via curl in the same way:

|  |  |
| --- | --- |
| 1  2 | curl -H "Accept:application/json"    http://localhost:8080/spring-rest/ex/foos |

Additionally, produces support multiple values as well:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(    value = "/ex/foos",    method = GET,    produces = { "application/json", "application/xml" }  ) |

Keep in mind that these – the old way and the new way of specifying the accept header – are basically the same mapping, so Spring won't allow them together – having both these methods active would result in:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Caused by: java.lang.IllegalStateException: Ambiguous mapping found.  Cannot map 'fooController' bean method  java.lang.String  org.baeldung.spring.web.controller    .FooController.getFoosAsJsonFromREST()  to  { [/ex/foos],    methods=[GET],params=[],headers=[],    consumes=[],produces=[application/json],custom=[]  }:  There is already 'fooController' bean method  java.lang.String  org.baeldung.spring.web.controller    .FooController.getFoosAsJsonFromBrowser()  mapped. |

A final note on the new produces and consumes mechanism – these behave differently from most other annotations: when specified at the type level, **the method level annotations do not complement but override** the type level information.

And of course, if you want to dig deeper into building a REST API with Spring – [check out](https://www.baeldung.com/rest-with-spring-series/)[**the new REST with Spring course**](https://www.baeldung.com/rest-with-spring-course?utm_source=blog&utm_medium=web&utm_content=art1&utm_campaign=rws).

## **4.**RequestMapping**With Path Variables**

Parts of the mapping URI can be bound to variables via the @PathVariable annotation.

### **4.1. Single**@PathVariable

A simple example with a single path variable:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @RequestMapping(value = "/ex/foos/{id}", method = GET)  @ResponseBody  public String getFoosBySimplePathWithPathVariable(    @PathVariable("id") long id) {      return "Get a specific Foo with id=" + id;  } |

This can be tested with curl:

|  |  |
| --- | --- |
| 1 | curl http://localhost:8080/spring-rest/ex/foos/1 |

If the name of the method parameter matches the name of the path variable exactly, then this can be simplified by **using @PathVariable with no value**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @RequestMapping(value = "/ex/foos/{id}", method = GET)  @ResponseBody  public String getFoosBySimplePathWithPathVariable(    @PathVariable String id) {      return "Get a specific Foo with id=" + id;  } |

Note that @PathVariable benefits from automatic type conversion, so we could have also declared the id as:

|  |  |
| --- | --- |
| 1 | @PathVariable long id |

### **4.2. Multiple**@PathVariable

More complex URI may need to map multiple parts of the URI to **multiple values**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RequestMapping(value = "/ex/foos/{fooid}/bar/{barid}", method = GET)  @ResponseBody  public String getFoosBySimplePathWithPathVariables    (@PathVariable long fooid, @PathVariable long barid) {      return "Get a specific Bar with id=" + barid +        " from a Foo with id=" + fooid;  } |

This is easily tested with a curl in the same way:

|  |  |
| --- | --- |
| 1 | curl http://localhost:8080/spring-rest/ex/foos/1/bar/2 |

### **4.3.**@PathVariable**With RegEx**

Regular expressions can also be used when mapping the @PathVariable; for example, we will restrict the mapping to only accept numerical values for the id:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @RequestMapping(value = "/ex/bars/{numericId:[\\d]+}", method = GET)  @ResponseBody  public String getBarsBySimplePathWithPathVariable(    @PathVariable long numericId) {      return "Get a specific Bar with id=" + numericId;  } |

This will mean that the following URIs will match:

|  |  |
| --- | --- |
| 1 | http://localhost:8080/spring-rest/ex/bars/1 |

But this will not:

|  |  |
| --- | --- |
| 1 | http://localhost:8080/spring-rest/ex/bars/abc |

## **5.**RequestMapping**With Request Parameters**

@RequestMapping allows easy **mapping of URL parameters with the @RequestParam annotation**.

We are now mapping a request to a URI such as:

|  |  |
| --- | --- |
| 1 | http://localhost:8080/spring-rest/ex/bars?id=100 |
| 1  2  3  4  5  6 | @RequestMapping(value = "/ex/bars", method = GET)  @ResponseBody  public String getBarBySimplePathWithRequestParam(    @RequestParam("id") long id) {      return "Get a specific Bar with id=" + id;  } |

We are then extracting the value of the id parameter using the @RequestParam(“id”) annotation in the controller method signature.

To send a request with the id parameter, we'll use the parameter support in curl:

|  |  |
| --- | --- |
| 1 | curl -i -d id=100 http://localhost:8080/spring-rest/ex/bars |

In this example, the parameter was bound directly without having been declared first.

For more advanced scenarios, **@RequestMapping can optionally define the parameters** – as yet another way of narrowing the request mapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @RequestMapping(value = "/ex/bars", params = "id", method = GET)  @ResponseBody  public String getBarBySimplePathWithExplicitRequestParam(    @RequestParam("id") long id) {      return "Get a specific Bar with id=" + id;  } |

Even more flexible mappings are allowed – multiple params values can be set, and not all of them have to be used:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @RequestMapping(    value = "/ex/bars",    params = { "id", "second" },    method = GET)  @ResponseBody  public String getBarBySimplePathWithExplicitRequestParams(    @RequestParam("id") long id) {      return "Narrow Get a specific Bar with id=" + id;  } |

And of course, a request to a URI such as:

|  |  |
| --- | --- |
| 1 | http://localhost:8080/spring-rest/ex/bars?id=100&second=something |

Will always be mapped to the best match – which is the narrower match, which defines both the id and the second parameter.

## **6.**RequestMapping**Corner Cases**

### **6.1.**@RequestMapping**– Multiple Paths Mapped to the Same Controller Method**

Although a single @RequestMapping path value is usually used for a single controller method, this is just good practice, not a hard and fast rule – there are some cases where mapping multiple requests to the same method may be necessary. For that case, **the value attribute of @RequestMapping does accept multiple mappings**, not just a single one:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RequestMapping(    value = { "/ex/advanced/bars", "/ex/advanced/foos" },    method = GET)  @ResponseBody  public String getFoosOrBarsByPath() {      return "Advanced - Get some Foos or Bars";  } |

Now, both of these curl commands should hit the same method:

|  |  |
| --- | --- |
| 1  2 | curl -i http://localhost:8080/spring-rest/ex/advanced/foos  curl -i http://localhost:8080/spring-rest/ex/advanced/bars |

### **6.2.**@RequestMapping**– Multiple HTTP Request Methods to the Same Controller Method**

Multiple requests using different HTTP verbs can be mapped to the same controller method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @RequestMapping(    value = "/ex/foos/multiple",    method = { RequestMethod.PUT, RequestMethod.POST }  )  @ResponseBody  public String putAndPostFoos() {      return "Advanced - PUT and POST within single method";  } |

With curl, both of these will now hit the same method:

|  |  |
| --- | --- |
| 1  2 | curl -i -X POST http://localhost:8080/spring-rest/ex/foos/multiple  curl -i -X PUT http://localhost:8080/spring-rest/ex/foos/multiple |

### **6.3.**@RequestMapping**– a Fallback for All Requests**

To implement a simple fallback for all requests using a particular HTTP method – for example, for a GET:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value = "\*", method = RequestMethod.GET)  @ResponseBody  public String getFallback() {      return "Fallback for GET Requests";  } |

Or even for all requests:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @RequestMapping(    value = "\*",    method = { RequestMethod.GET, RequestMethod.POST ... })  @ResponseBody  public String allFallback() {      return "Fallback for All Requests";  } |

### **6.4. Ambiguous Mapping Error**

The ambiguous mapping error occurs when Spring evaluates two or more request mappings to be the same for different controller methods.  A request mapping is the same when it has the same HTTP method, URL, parameters, headers, and media type. This, for example, is an ambiguous mapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @GetMapping(value = "foos/duplicate" )  public String duplicate() {      return "Duplicate";  }    @GetMapping(value = "foos/duplicate" )  public String duplicateEx() {      return "Duplicate";  } |

The exception thrown usually does have error messages along these lines:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Caused by: java.lang.IllegalStateException: Ambiguous mapping.    Cannot map 'fooMappingExamplesController' method    public java.lang.String org.baeldung.web.controller.FooMappingExamplesController.duplicateEx()    to {[/ex/foos/duplicate],methods=[GET]}:    There is already 'fooMappingExamplesController' bean method    public java.lang.String org.baeldung.web.controller.FooMappingExamplesController.duplicate() mapped. |

A careful reading of the error message points to the fact that Spring is unable to map the method org.baeldung.web.controller.FooMappingExamplesController.duplicateEx() as it has a conflicting mapping with an already mapped org.baeldung.web.controller.FooMappingExamplesController.duplicate().

**The code snippet below will not result in ambiguous mapping error because both methods return different content types:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @GetMapping(value = "foos/duplicate", produces = MediaType.APPLICATION\_XML\_VALUE)  public String duplicateXml() {      return "<message>Duplicate</message>";  }    @GetMapping(value = "foos/duplicate", produces = MediaType.APPLICATION\_JSON\_VALUE)  public String duplicateJson() {      return "{\"message\":\"Duplicate\"}";  } |

This differentiation allows our controller to return the correct data representation based on the Accepts header supplied in the request.

Another way to resolve this is to update the URL assigned to either of the two methods involved.

## **7. New Request Mapping Shortcuts**

Spring Framework 4.3 introduced [a few new](https://www.baeldung.com/spring-new-requestmapping-shortcuts) HTTP mapping annotations, all based on @RequestMapping:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

These new annotations can improve the readability and reduce the verbosity of the code. Let us look at these new annotations in action by creating a RESTful API that supports CRUD operations:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | @GetMapping("/{id}")  public ResponseEntity<?> getBazz(@PathVariable String id){      return new ResponseEntity<>(new Bazz(id, "Bazz"+id), HttpStatus.OK);  }    @PostMapping  public ResponseEntity<?> newBazz(@RequestParam("name") String name){      return new ResponseEntity<>(new Bazz("5", name), HttpStatus.OK);  }    @PutMapping("/{id}")  public ResponseEntity<?> updateBazz(    @PathVariable String id,    @RequestParam("name") String name) {      return new ResponseEntity<>(new Bazz(id, name), HttpStatus.OK);  }    @DeleteMapping("/{id}")  public ResponseEntity<?> deleteBazz(@PathVariable String id){      return new ResponseEntity<>(new Bazz(id), HttpStatus.OK);  } |

A deep dive into these can be found [here](https://www.baeldung.com/spring-new-requestmapping-shortcuts).

## **8. Spring Configuration**

The Spring MVC Configuration is simple enough – considering that our FooController is defined in the following package:

|  |  |
| --- | --- |
| 1  2  3  4 | package org.baeldung.spring.web.controller;    @Controller  public class FooController { ... } |

We simply need a @Configuration class to enable the full MVC support and configure classpath scanning for the controller:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Configuration  @EnableWebMvc  @ComponentScan({ "org.baeldung.spring.web.controller" })  public class MvcConfig {      //  } |

## **1. Overview**

Spring 4.3. [introduced](https://jira.spring.io/browse/SPR-13442) some very cool method-level composed annotations to smooth out the handling @RequestMapping in typical Spring MVC projects.

In this article, we will learn how to use them in an efficient way.

## **2. New Annotations**

Typically, if we want to implement the URL handler using traditional @RequestMapping annotation, it would have been something like this:

|  |  |
| --- | --- |
| 1 | @RequestMapping(value = "/get/{id}", method = RequestMethod.GET) |

The new approach makes it possible to shorten this simply to:

|  |  |
| --- | --- |
| 1 | @GetMapping("/get/{id}") |

Spring currently supports five types of inbuilt annotations for handling different types of incoming HTTP request methods which are GET, POST, PUT, DELETE and PATCH. These annotations are:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

From the naming convention we can see that each annotation is meant to handle respective incoming request method type, i.e. @GetMapping is used to handle GET type of request method, @PostMapping is used to handle POST type of request method, etc.

## **3. How It Works**

All of the above annotations are already internally annotated with @RequestMapping and the respective value in the method element.

For example, if we'll look at the source code of @GetMapping annotation, we can see that it's already annotated with RequestMethod.GET in the following way:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Target({ java.lang.annotation.ElementType.METHOD })  @Retention(RetentionPolicy.RUNTIME)  @Documented  @RequestMapping(method = { RequestMethod.GET })  public @interface GetMapping {      // abstract codes  } |

All the other annotations are created in the same way, i.e. @PostMapping is annotated with RequestMethod.POST, @PutMapping is annotated with RequestMethod.PUT, etc.

The full source code of the annotations is available [here](https://github.com/spring-projects/spring-framework/tree/master/spring-web/src/main/java/org/springframework/web/bind/annotation).

## **4. Implementation**

Let's try to use these annotations to build a quick REST application.

Please note that since we would use Maven to build the project and Spring MVC to create our application, we need to add necessary dependencies in the pom.xml:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-webmvc</artifactId>      <version>5.2.2.RELEASE</version>  </dependency> |

The latest version of spring-webmvc is available in the [Central Maven Repository](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework%22%20AND%20a%3A%22spring-webmvc%22).

Now, we need to create the controller to map incoming request URL. Inside this controller, we would use all of these annotations one by one.

### **4.1.**@GetMapping

|  |  |
| --- | --- |
| 1  2  3  4 | @GetMapping("/get")  public @ResponseBody ResponseEntity<String> get() {      return new ResponseEntity<String>("GET Response", HttpStatus.OK);  } |
| 1  2  3  4  5  6 | @GetMapping("/get/{id}")  public @ResponseBody ResponseEntity<String>    getById(@PathVariable String id) {      return new ResponseEntity<String>("GET Response : "        + id, HttpStatus.OK);  } |

### **4.2.**@PostMapping

|  |  |
| --- | --- |
| 1  2  3  4 | @PostMapping("/post")  public @ResponseBody ResponseEntity<String> post() {      return new ResponseEntity<String>("POST Response", HttpStatus.OK);  } |

### **4.3.**@PutMapping

|  |  |
| --- | --- |
| 1  2  3  4 | @PutMapping("/put")  public @ResponseBody ResponseEntity<String> put() {      return new ResponseEntity<String>("PUT Response", HttpStatus.OK);  } |

### **4.4.**@DeleteMapping

|  |  |
| --- | --- |
| 1  2  3  4 | @DeleteMapping("/delete")  public @ResponseBody ResponseEntity<String> delete() {      return new ResponseEntity<String>("DELETE Response", HttpStatus.OK);  } |

### **4.5.**@PatchMapping

|  |  |
| --- | --- |
| 1  2  3  4 | @PatchMapping("/patch")  public @ResponseBody ResponseEntity<String> patch() {      return new ResponseEntity<String>("PATCH Response", HttpStatus.OK);  } |

**Points to note:**

* We have used the necessary annotations to handle proper incoming HTTP methods with URI. For example, @GetMapping to handle “/get” URI, @PostMapping to handle “/post” URI and so on
* Since we are making an REST-based application, we are returning a constant string (unique to each request type) with 200 response code to simplify the application. We have used Spring's @ResponseBody annotation in this case.
* If we had to handle any URL path variable, we can simply do it in much less way we used to do in case of using @RequestMapping.

## **5. Testing the Application**

To test the application we need to create a couple of test cases using JUnit. We would use SpringJUnit4ClassRunner to initiate the test class. We would create five different test cases to test each annotation and every handler we declared in the controller.

Let's simple the example test case of @GetMapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Test  public void giventUrl\_whenGetRequest\_thenFindGetResponse()    throws Exception {        MockHttpServletRequestBuilder builder = MockMvcRequestBuilders        .get("/get");        ResultMatcher contentMatcher = MockMvcResultMatchers.content()        .string("GET Response");        this.mockMvc.perform(builder).andExpect(contentMatcher)        .andExpect(MockMvcResultMatchers.status().isOk());    } |

As we can see, we are expecting a constant string “GET Response“, once we hit the GET URL “/get”.

Now, let's create the test case to test @PostMapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Test  public void givenUrl\_whenPostRequest\_thenFindPostResponse()    throws Exception {        MockHttpServletRequestBuilder builder = MockMvcRequestBuilders        .post("/post");        ResultMatcher contentMatcher = MockMvcResultMatchers.content()        .string("POST Response");        this.mockMvc.perform(builder).andExpect(contentMatcher)        .andExpect(MockMvcResultMatchers.status().isOk());    } |

**In the same way, we created the rest of the test cases to test all of the HTTP methods.**

Alternatively, we can always use any common REST client, for example, PostMan, RESTClient etc, to test our application. In that case, we need to be a little careful to choose correct HTTP method type while using the rest client. Otherwise, it would throw 405 error status.

## **6. Conclusion**

In this article, we had a quick introduction to the different types of @RequestMapping shortcuts for quick web development using traditional Spring MVC framework. **We can utilize these quick shortcuts to create a clean code base.**

***Difference***

7

1) @RequestParam is used to extract *query parameters*

http://localhost:3000/api/group/test?id=4

@GetMapping("/group/test")

public ResponseEntity<?> test(@RequestParam Long id) {

System.out.println("This is test");

return ResponseEntity.ok().body(id);

}

while @PathVariable is used to extract data right from the URI:

http://localhost:3000/api/group/test/4

@GetMapping("/group/test/{id}")

public ResponseEntity<?> test(@PathVariable Long id) {

System.out.println("This is test");

return ResponseEntity.ok().body(id);

}

2) @RequestParam is more useful on a traditional web application where data is mostly passed in the query parameters while @PathVariable is more suitable for RESTful web services where URL contains values.

3) @RequestParam annotation can specify *default values* if a query parameter is not present or empty by using a defaultValue attribute, provided the required attribute is false:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/name")

String getName(@RequestParam(value = "person", defaultValue = "John") String personName) {

return "Required element of request param";

}

}