(http://baeldung.com)

Solicitud de primaveraMapeo

Última modificación: 6 de abril de 2018

por Eugen Paraschiv (http://www.baeldung.com/author/eugen/)
(http://www.baeldung.com/author/eugen/)

Primavera (http://www.baeldung.com/category/spring/) +
Spring MVC (http://www.baeldung.com/category/spring-mvc/)

1. Información general

En este artículo, nos centraremos en una de las principales anotaciones en **Spring MVC - *@RequestMapping.***

En pocas palabras, la anotación se utiliza para asignar las solicitudes web a los métodos de Spring Controller.

Otras lecturas:

Sirve recursos estáticos con Spring (http://www.baeldung.com/spring-)

Getting Started with Forms in Spring MVC (http://www.baeldung.com/spring-)

Http Message Converters with the Spring Framework

mvc-static-resources)

Cómo mapear y manejar recursos estáticos con Spring MVC - use la configuración simple, luego la 3.1 más flexible y finalmente la nueva resolución de recursos 4.1.

Leer más (http://www.baeldung.com/spring-mvc-static-resources) →

mvc-form-tutorial)

Learn how to work with forms using Spring MVC - mapping a basic entity, submit, displaying errors.

Read more (http://www.baeldung.com/spring-mvc-form-tutorial) →

(http://www.baeldung.com/httpmessageconverter-rest)

How to configure HttpMessageConverters for a REST API with Spring, and how to use these converters with the RestTemplate.

Read more (http://www.baeldung.com/spring-httpmessageconverter-rest) →

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

```
@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:

```
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:

```
@RequestMapping(value = "/ex/foos", method = POST)
@ResponseBody
public String postFoos() {
    return "Post some Foos";
}
```

To test the POST via a *curl* command:

```
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:

```
@RequestMapping(value = "/ex/foos", headers = "key=val", method = GET)
@ResponseBody
public String getFoosWithHeader() {
    return "Get some Foos with Header";
}
```

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

```
@RequestMapping(
  value = "/ex/foos",
  headers = { "key1=val1", "key2=val2" }, method = GET)
@ResponseBody
public String getFoosWithHeaders() {
    return "Get some Foos with Header";
}
```

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

```
curl -i -H "key:val" 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:

```
@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:

```
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:

```
@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:

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

Additionally, *produces* support multiple values as well:

```
@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:

```
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 (http://www.baeldung.com/rest-with-spring-series/) **the new *REST with Spring course* (http://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:

```
@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*:

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

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

```
@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:

```
@PathVariable long id
```

4.2. Multiple *@PathVariable*

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

```
@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:

```
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*:

```
@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:

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

But this will not:

```
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:

```
http://localhost:8080/spring-rest/ex/bars?id=100
```

```
@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.

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

```
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:

```
@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:

```
@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:

```
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:

```
@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:

```
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:

```
@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:

```
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:

```
@RequestMapping(value = "*", method = RequestMethod.GET)
@ResponseBody
public String getFallback() {
    return "Fallback for GET Requests";
}
```

Or even for all requests:

```
@RequestMapping(
  value = "*",
  method = { RequestMethod.GET, RequestMethod.POST ... })
@ResponseBody
public String allFallback() {
    return "Fallback for All Requests";
}
```


7. New Request Mapping Shortcuts

Spring Framework 4.3 introduced a few new (http://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:

```
@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 (http://www.baeldung.com/spring-new-requestmapping-shortcuts).

8. Spring Configuration

La configuración Spring MVC es lo suficientemente simple, teniendo en cuenta que nuestro *FooController* se define en el siguiente paquete:

```
package org.baeldung.spring.web.controller;

@Controller
public class FooController { ... }
```

Simplemente necesitamos una clase *@Configuration* para habilitar el soporte completo de MVC y configurar el escaneo classpath para el controlador:

```
@Configuration
@EnableWebMvc
@ComponentScan({ "org.baeldung.spring.web.controller" })
public class MvcConfig {
    //
}
```

9. Conclusión

Este artículo se centra en la **anotación *@RequestMapping* en Spring** , donde se analiza un caso de uso simple, la asignación de encabezados HTTP, la unión de partes del URI con *@PathVariable* y el trabajo con parámetros de URI y la anotación *@RequestParam* .

Si desea aprender a usar otra anotación central en Spring MVC, puede explorar la anotación *@ModelAttribute* aquí (http://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation) .

El código completo del artículo está disponible en Github (https://github.com/eugenp/tutorials/tree/master/spring-rest-simple) . Este es un proyecto de Maven, por lo que debería ser fácil importarlo y ejecutarlo tal como está.

Huésped

Me alegra que hayas encontrado el artículo útil.

+ 3 −

Hace 4 años

Lajos Incze

Huésped

+1

+ 0 −

Hace 2 años

Jonathan Gibran Hernandez Antu

Huésped

pienso lo mismo

+ 1 −

hace 1 año

Huésped

grooha

OK, pero ¿qué pasa con los valores múltiples en un parámetro 1, por ejemplo? http: // ... / age = 23 & fav = 12 & fav = 15 Como puede ver, tenemos dos parámetros: edad y favoritos. El param favorito tiene dos valores: 12 y 15. Supongamos que hay una clase: public class MyClass {private int age; privado int fav; ...} No hay problema con un solo valor para fav - se vinculará a int fav en la clase MyClass. Pero en ese caso tenemos dos valores ... supongo, Spring MVC vinculará solo el primer valor (12) a fav, ¿no? ¿Qué debo hacer para hacer esto ... Leer más »

+ 0 −

Hace 4 años

Huésped

Eugen Paraschiv (http://www.baeldung.com/)

Pregunta interesante: si tiene una opción, le sugiero que mantenga los nombres de los parámetros únicos (puede tener los valores separados por coma y luego analizarlos). Si eso no es una opción, trataría de mapearlo en una matriz, y si Spring no puede hacerlo, siempre puedes abrir una JIRA.
Siempre puede ir al nivel inferior y simplemente inyectar en su método de controlador la solicitud http, y analizar los parámetros usted mismo.
Saludos,
Eugen.

+ 0 −

Hace 4 años

Huésped

grooha

Gracias por tu respuesta. He considerado analizar todos los parámetros necesarios en mi controlador, tal vez sea la mejor opción. Le informaré y publicaré una solución de trabajo; tal vez ayude a alguien en el futuro. ¡Gracias de nuevo!

+ 0 −

Hace 4 años

Huésped

Norbert Csaba Herczeg (http://norbertherczeg.me)

En las operaciones de tipo receptor: headers = "Accept = application / json" en lugar de definir manualmente los tipos MIME, puede usar consumes = MediaType.APPLICATION_JSON_VALUE.

¡De esta manera es muy fácil refactorizar, y más errores / errores tipográficos! 😶

+ 0 −

hace 3 años

Huésped

Eugen Paraschiv (http://www.baeldung.com/)

Hola Norbert, sí, lo hago en código, pero en el artículo quería que las muestras fueran lo más legibles posible, así que ir por el texto en bruto es un poco más fácil de leer y entender. Gracias por la sugerencia. ¡Salud,
Eugen!

+ 0 −

hace 3 años

Huésped

Manish Sahni

Hola Eugen,
Bonito artículo informativo sobre Rest. Cómo podemos pasar un Object del cliente y manejarlo en el Rest utilizando Spring. Soy nuevo en Rest y he visto algunos artículos que describen que podemos enviarlos como una cadena JSON o XML.
¿Hay alguna otra manera de lograr esto aparte de estas?

+ 0 −

hace 3 años

Huésped

Eugen Paraschiv (http://www.baeldung.com/)

Hey Manish: claro, puedes mapear un objeto pasado en el cuerpo de la solicitud en la capa de tu controlador. Tendrás que usar la anotación @RequestBody para eso. Puede consultar este artículo introductorio (http://www.baeldung.com/2011/10/25/building-a-restful-web-service-with-spring-3-1-and-java-based-configuration-part-2/) para ver cómo funciona. Espero eso ayude. Saludos,
Eugen.

+ 2 −

hace 3 años

Cargar más comentarios

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

¿Cuál de estos es el más cercano a su trabajo / función actual?

Desarrollador

Desarrollador Senior

Desarrollador principal

Arquitecto

Gerente

KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORES (HTTP://WWW.BAELDUNG.COM/EDITORS)
KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)