org.springframework.web.bind.annotation

Annotation Type RequestMapping

```
@Target(value={METHOD,TYPE})
 @Retention(value=RUNTIME)
 @Documented
public @interface RequestMapping
```

Annotation for mapping web requests onto specific handler classes and/or handler methods. Provides a consistent style between Servlet and Portlet environments, with the semantics adapting to the concrete environment.

NOTE: The set of features supported for Servlets is a superset of the set of features supported for Portlets. The places where this applies are marked with the label "Servlet-only" in this source file. For Servlet environments there are some further distinctions depending on whether an application is configured with "@MVC 3.0" or "@MVC 3.1" support classes. The places where this applies are marked with "@MVC 3.1-only" in this source file. For more details see the note on the new support classes added in Spring MVC 3.1 further below.

Handler methods which are annotated with this annotation are allowed to have very flexible signatures. They may have parameters of the following types, in arbitrary order (except for validation results, which need to follow right after the corresponding command object, if desired):

- Request and/or response objects (Servlet API or Portlet API). You may choose any specific request/response type, e.g. `ServletRequest` / `HttpServletRequest` or `PortletRequest` / `ActionRequest` / `RenderRequest`. Note that in the Portlet case, an explicitly declared action/render argument is also used for mapping specific request types onto a handler method (in case of no other information given that differentiates between action and render requests).
- Session object (Servlet API or Portlet API): either `HttpSession` or `PortletSession`. An argument of this type will enforce the presence of a corresponding session. As a consequence, such an argument will never be `null`. *Note that session access may not be thread-safe, in particular in a Servlet environment: Consider switching the `"synchronizeOnSession"` flag to "true" if multiple requests are allowed to access a session concurrently.*
- `WebRequest` or `NativeWebRequest`. Allows for generic request parameter access as well as request/session attribute access, without ties to the native Servlet/Portlet API.
- `Locale` for the current request locale (determined by the most specific locale resolver available, i.e. the configured `LocaleResolver` in a Servlet environment and the portal locale in a Portlet environment).
- `InputStream` / `Reader` for access to the request's content. This will be the raw InputStream/Reader as exposed by the Servlet/Portlet API.

- `OutputStream` / `Writer` for generating the response's content. This will be the raw OutputStream/Writer as exposed by the Servlet/Portlet API.
- `HttpMethod` for the HTTP request method
- `@PathVariable` annotated parameters (Servlet-only) for access to URI template values (i.e. /hotels/{hotel}). Variable values will be converted to the declared method argument type. By default, the URI template will match against the regular expression `[^\.]*` (i.e. any character other than period), but this can be changed by specifying another regular expression, like so: /hotels/{hotel:\d+}. Additionally, `@PathVariable` can be used on a `Map<String, String>` to gain access to all URI template variables.
- `@MatrixVariable` annotated parameters (Servlet-only) for access to name-value pairs located in URI path segments. Matrix variables must be represented with a URI template variable. For example /hotels/{hotel} where the incoming URL may be "/hotels/42;q=1". Additionally, `@MatrixVariable` can be used on a `Map<String, String>` to gain access to all matrix variables in the URL or to those in a specific path variable.
- `@RequestParam` annotated parameters for access to specific Servlet/Portlet request parameters. Parameter values will be converted to the declared method argument type. Additionally, `@RequestParam` can be used on a `Map<String, String>` or `MultiValueMap<String, String>` method parameter to gain access to all request parameters.
- `@RequestHeader` annotated parameters for access to specific Servlet/Portlet request HTTP headers. Parameter values will be converted to the declared method argument type. Additionally, `@RequestHeader` can be used on a `Map<String, String>`, `MultiValueMap<String, String>`, or `HttpHeaders` method parameter to gain access to all request headers.
- `@RequestBody` annotated parameters (Servlet-only) for access to the Servlet request HTTP contents. The request stream will be converted to the declared method argument type using message converters. Such parameters may optionally be annotated with `@Valid` and also support access to validation results through an `Errors` argument. Instead a `MethodArgumentNotValidException` exception is raised.
- `@RequestPart` annotated parameters (Servlet-only, @MVC 3.1-only) for access to the content of a part of "multipart/form-data" request. The request part stream will be converted to the declared method argument type using message converters. Such parameters may optionally be annotated with `@Valid` and support access to validation results through a `Errors` argument. Instead a `MethodArgumentNotValidException` exception is raised.
- `@SessionAttribute` annotated parameters for access to existing, permanent session attributes (e.g. user authentication object) as opposed to model attributes temporarily stored in the session as part of a controller workflow via `SessionAttributes`.
- `@RequestAttribute` annotated parameters for access to request attributes.
- `HttpEntity<?>` parameters (Servlet-only) for access to the Servlet request HTTP headers and contents. The request stream will be converted to the entity body using message converters.
- `Map` / `Model` / `ModelMap` for enriching the implicit model that will be exposed to the web view.
- `RedirectAttributes` (Servlet-only, @MVC 3.1-only) to specify the exact set of attributes to use in case of a redirect and also to add flash attributes (attributes stored temporarily on the server-side to make them available to the request after the redirect). `RedirectAttributes` is used instead of the implicit model if the method returns a "redirect:" prefixed view name or `RedirectView`.
- Command/form objects to bind parameters to: as bean properties or fields, with customizable type conversion, depending on `InitBinder` methods and/or the

HandlerAdapter configuration - see the "webBindingInitializer" property on RequestMappingHandlerMethodAdapter. Such command objects along with their validation results will be exposed as model attributes, by default using the non-qualified command class name in property notation (e.g. "orderAddress" for type "mypackage.OrderAddress"). Specify a parameter-level `@ModelAttribute` annotation for declaring a specific model attribute name.

- `Errors` / `BindingResult` validation results for a preceding command/form object (the immediate preceding argument).
- `SessionStatus` status handle for marking form processing as complete (triggering the cleanup of session attributes that have been indicated by the `@SessionAttributes` annotation at the handler type level).
- `UriComponentsBuilder` (Servlet-only, @MVC 3.1-only) for preparing a URL relative to the current request's host, port, scheme, context path, and the literal part of the servlet mapping.

Note: Java 8's `java.util.Optional` is supported as a method parameter type with annotations that provide a `required` attribute (e.g. `@RequestParam`, `@RequestHeader`, etc.). The use of `java.util.Optional` in those cases is equivalent to having `required=false`.

The following return types are supported for handler methods:

- A `ModelAndView` object (Servlet MVC or Portlet MVC), with the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `Model` object, with the view name implicitly determined through a `RequestToViewNameTranslator` and the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `Map` object for exposing a model, with the view name implicitly determined through a `RequestToViewNameTranslator` and the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `View` object, with the model implicitly determined through command objects and `@ModelAttribute` annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
- A `String` value which is interpreted as view name, with the model implicitly determined through command objects and `@ModelAttribute` annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a `ModelMap` argument (see above).
- `@ResponseBody` annotated methods (Servlet-only) for access to the Servlet response HTTP contents. The return value will be converted to the response stream using message converters.
- An `HttpEntity<?>` or `ResponseEntity<?>` object (Servlet-only) to access to the Servlet response HTTP headers and contents. The entity body will be converted to the response stream using message converters.
- An `HttpHeaders` object to return a response with no body.
- A `Callable` which is used by Spring MVC to obtain the return value asynchronously in a separate thread transparently managed by Spring MVC on behalf of the application.
- A `DeferredResult` which the application uses to produce a return value in a separate thread of its own choosing, as an alternative to returning a Callable.
- A `ListenableFuture` which the application uses to produce a return value in a separate thread of its own choosing, as an alternative to returning a Callable.
- A `CompletionStage` (implemented by `CompletableFuture` for example) which the application uses to produce a return value in a separate thread of its own choosing, as an

alternative to returning a Callable.
- A `ResponseBodyEmitter` can be used to write multiple objects to the response asynchronously; also supported as the body within `ResponseEntity`.
- An `SseEmitter` can be used to write Server-Sent Events to the response asynchronously; also supported as the body within `ResponseEntity`.
- A `StreamingResponseBody` can be used to write to the response asynchronously; also supported as the body within `ResponseEntity`.
- `void` if the method handles the response itself (by writing the response content directly, declaring an argument of type `ServletResponse` / `HttpServletResponse` / `RenderResponse` for that purpose) or if the view name is supposed to be implicitly determined through a `RequestToViewNameTranslator` (not declaring a response argument in the handler method signature; only applicable in a Servlet environment).
- Any other return type will be considered as single model attribute to be exposed to the view, using the attribute name specified through `@ModelAttribute` at the method level (or the default attribute name based on the return type's class name otherwise). The model will be implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.

NOTE: `@RequestMapping` will only be processed if an an appropriate `HandlerMapping-HandlerAdapter` pair is configured. This is the case by default in both the `DispatcherServlet` and the `DispatcherPortlet`. However, if you are defining custom `HandlerMappings` or `HandlerAdapters`, then you need to add `DefaultAnnotationHandlerMapping` and `AnnotationMethodHandlerAdapter` to your configuration..

NOTE: Spring 3.1 introduced a new set of support classes for `@RequestMapping` methods in Servlet environments called `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter`. They are recommended for use and even required to take advantage of new features in Spring MVC 3.1 (search "@MVC 3.1-only" in this source file) and going forward. The new support classes are enabled by default from the MVC namespace and with use of the MVC Java config (`@EnableWebMvc`) but must be configured explicitly if using neither.

NOTE: When using controller interfaces (e.g. for AOP proxying), make sure to consistently put *all* your mapping annotations - such as `@RequestMapping` and `@SessionAttributes` - on the controller *interface* rather than on the implementation class.

Since:

2.5

Author:

Juergen Hoeller, Arjen Poutsma, Sam Brannen

See Also:

`GetMapping`, `PostMapping`, `PutMapping`, `DeleteMapping`, `PatchMapping`, `RequestParam`, `RequestAttribute`, `PathVariable`, `ModelAttribute`, `SessionAttribute`, `SessionAttributes`, `InitBinder`, `WebRequest`, `RequestMappingHandlerAdapter`, `DefaultAnnotationHandlerMapping`, `AnnotationMethodHandlerAdapter`

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
`String[]`	**`consumes`** The consumable media types of the mapped request, narrowing the primary mapping.
`String[]`	**`headers`** The headers of the mapped request, narrowing the primary mapping.
`RequestMethod[]`	**`method`** The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.
`String`	**`name`** Assign a name to this mapping.
`String[]`	**`params`** The parameters of the mapped request, narrowing the primary mapping.
`String[]`	**`path`** In a Servlet environment only: the path mapping URIs (e.g.
`String[]`	**`produces`** The producible media types of the mapped request, narrowing the primary mapping.
`String[]`	**`value`** The primary mapping expressed by this annotation.

Element Detail

name

```
public abstract String name
```

Assign a name to this mapping.

Supported at the type level as well as at the method level! When used on both levels, a combined name is derived by concatenation with "#" as separator.

See Also:

`MvcUriComponentsBuilder`, `HandlerMethodMappingNamingStrategy`

Default:

""

value

```
@AliasFor(value="path")
public abstract String[] value
```

The primary mapping expressed by this annotation.

In a Servlet environment this is an alias for `path()`. For example `@RequestMapping("/foo")` is equivalent to `@RequestMapping(path="/foo")`.

In a Portlet environment this is the mapped portlet modes (i.e. "EDIT", "VIEW", "HELP" or any custom modes).

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings inherit this primary mapping, narrowing it for a specific handler method.

Default:

{}

path

```
@AliasFor(value="value")
public abstract String[] path
```

In a Servlet environment only: the path mapping URIs (e.g. "/myPath.do"). Ant-style path patterns are also supported (e.g. "/myPath/*.do"). At the method level, relative paths (e.g. "edit.do") are supported within the primary mapping expressed at the type level. Path mapping URIs may contain placeholders (e.g. "/${connect}")

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings inherit this primary mapping, narrowing it for a specific handler method.

Since:

4.2

See Also:

`ValueConstants.DEFAULT_NONE`

Default:

{}

method

```
public abstract RequestMethod[] method
```

The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings inherit this HTTP method restriction (i.e. the type-level restriction gets checked before the handler method is even resolved).

Supported for Servlet environments as well as Portlet 2.0 environments.

Default:

{}

params

```
public abstract String[] params
```

The parameters of the mapped request, narrowing the primary mapping.

Same format for any environment: a sequence of "myParam=myValue" style expressions, with a request only mapped if each such parameter is found to have the given value. Expressions can be negated by using the "!=" operator, as in "myParam!=myValue". "myParam" style expressions are also supported, with such parameters having to be present in the request (allowed to have any value). Finally, "!myParam" style expressions indicate that the specified parameter is *not* supposed to be present in the request.

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings inherit this parameter restriction (i.e. the type-level restriction gets checked before the handler method is even resolved).

In a Servlet environment, parameter mappings are considered as restrictions that are enforced at the type level. The primary path mapping (i.e. the specified URI value) still has to uniquely identify the target handler, with parameter mappings simply expressing preconditions for invoking the handler.

In a Portlet environment, parameters are taken into account as mapping differentiators, i.e. the primary portlet mode mapping plus the parameter conditions uniquely identify the target handler. Different handlers may be mapped onto the same portlet mode, as long as their parameter mappings differ.

Default:

{}

headers

```
public abstract String[] headers
```

The headers of the mapped request, narrowing the primary mapping.

Same format for any environment: a sequence of "My-Header=myValue" style expressions, with a request only mapped if each such header is found to have the given value. Expressions can be negated by using the "!=" operator, as in "My-Header!=myValue". "My-Header" style expressions are also supported, with such headers having to be present in the request (allowed to have any value). Finally, "!My-Header" style expressions indicate that the specified header is *not* supposed to be present in the request.

Also supports media type wildcards (*), for headers such as Accept and Content-Type. For instance,

```
@RequestMapping(value = "/something", headers = "content-type=text/*")
```

will match requests with a Content-Type of "text/html", "text/plain", etc.

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings inherit this header restriction (i.e. the type-level restriction gets checked before the handler method is even resolved).

Maps against HttpServletRequest headers in a Servlet environment, and against PortletRequest properties in a Portlet 2.0 environment.

See Also:

MediaType

Default:

{}

consumes

```
public abstract String[] consumes
```

The consumable media types of the mapped request, narrowing the primary mapping.

The format is a single media type or a sequence of media types, with a request only mapped if the `Content-Type` matches one of these media types. Examples:

```
consumes = "text/plain"
consumes = {"text/plain", "application/*"}
```

Expressions can be negated by using the "!" operator, as in "!text/plain", which matches all requests with a `Content-Type` other than "text/plain".

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings override this consumes restriction.

See Also:

MediaType, ServletRequest.getContentType()

Default:

{}

produces

```
public abstract String[] produces
```

The producible media types of the mapped request, narrowing the primary mapping.

The format is a single media type or a sequence of media types, with a request only mapped if the `Accept` matches one of these media types. Examples:

```
produces = "text/plain"
produces = {"text/plain", "application/*"}
produces = "application/json; charset=UTF-8"
```

It affects the actual content type written, for example to produce a JSON response with UTF-8 encoding, `"application/json; charset=UTF-8"` should be used.

Expressions can be negated by using the "!" operator, as in "!text/plain", which matches all requests with a `Accept` other than "text/plain".

Supported at the type level as well as at the method level! When used at the type level, all method-level mappings override this produces restriction.

See Also:

`MediaType`

Default:

{}

(http://baeldung.com)

Spring RequestMapping

Last modified: July 19, 2017

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

Spring (http://www.baeldung.com/category/spring/) +

1. Overview

In this article, we'll focus on one of the main annotations in **Spring MVC – *@RequestMapping.***

Simply put, the annotation is used to map web requests to Spring Controller methods.

Further reading:

Serve Static Resources with Spring (http://www.baeldung.com/spring-mvc-static-resources)

How to map and handle static resources with Spring MVC - use the simple configuration, then the 3.1 more flexible one and finally the new 4.1 resource resolvers.

Read more (http://www.baeldung.com/spring-mvc-static-resources) →

Getting Started with Forms in Spring MVC (http://www.baeldung.com/spring-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 Message Converters with the Spring Framework (http://www.baeldung.com/spring-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

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

```
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:

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

9. Conclusion

This article focus on the ***@RequestMapping* annotation in Spring** – discussing a simple use case, the mapping of HTTP headers, binding parts of the URI with *@PathVariable* and working with URI parameters and the *@RequestParam* annotation.

If you'd like to learn how to use another core annotation in Spring MVC, you can explore the *@ModelAttribute* annotation here (http://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation).

The full code from the article is available on Github (https://github.com/eugenp/tutorials/tree/master/spring-rest-simple). This is a Maven project, so it should be easy to import and run as it is.

Pham Vu Minh Hoang
(http://www.facebook.com/pvmhoang)

Guest

sorry, how i can get source code from git, i have try but i cann't

0

4 years 8 months ago

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

Guest

The link goes into the relevant submodule of the project – you need to go back one level (click on tutorials) and you'll see the git access URI – wich you can use to clone the project: https://github.com/eugenp/tutorials (https://github.com/eugenp/tutorials)

1

4 years 8 months ago

Cody Burleson (http://codyburleson.com/)

Guest

I really love the way you structured this post. It is extremely informative in a way that's very easy to scan. I learned a few things I didn't know about Spring, which I've been using for quite a long time now, in just minutes. Thanks for taking the time to share!

3

4 years 7 months ago

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

Guest

I'm glad you found the article helpful.

3

4 years 7 months ago

Lajos Incze

Guest

+1

0

2 years 7 months ago

Jonathan Gibran Hernandez Antu

Guest

I think the same

1

1 year 8 days ago

grooha

Guest

OK, but what's about multiple values pass in a 1 param, e.g.? http://…/age=23&fav=12&fav=15 As you can see, we have two parameters: age, and fav. The fav param. has two values: 12, and 15. Suppose, that there is a class: public class MyClass { private int age; private int fav; … } No problem with a single value for fav – it will be binded to int fav in the MyClass class. But in that case we have two values… I suppose, Spring MVC will bind only the first value (12) to fav, right? What should I do to make this… Read more »

0

4 years 4 months ago

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

Guest

Interesting question – if you have a choice, I would suggest keeping the parameter names unique (you can have the values as comma separated and then parse them out). If that's not an option, I would try to map it to an array – and if Spring isn't able to do so, you can always open a JIRA.
You can always go lower level and simply inject into your controller method the http request, and parse out the parameters yourself.
Cheers,
Eugen.

0

4 years 4 months ago

grooha

Guest

Thanks for your answer. I've considered parsing out all needed params in my controller, maybe it will be the best choice. I will let you know and publish a working solution – maybe it will help someone in the future. Thanks again!

0

4 years 4 months ago

Manish Sahni

Guest

Hi Eugen,
Nice informative artice on Rest .How can we pass a Object from the client and handle it in the Rest using Spring.I am new to Rest and seen some articles hat describes we can send it as a JSON string or XML.
Is thier any other way to achieve this other than these ?

0

3 years 4 months ago

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

Guest

Hey Manish – sure, you can map an object passed in the body of the request in your controller layer – you'll need to use the @RequestBody annotation for that. You can check out this introductory article (http://www.baeldung.com/2011/10/25/building-a-restful-web-service-with-spring-3-1-and-java-based-configuration-part-2/) to see how that works. Hope it helps. Cheers,
Eugen.

2

3 years 4 months ago

Tobi Bod

Guest

Great article and summary of RequestMapping! Thanks for your effort!

2

2 years 9 months ago

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

Guest

Hey Tobi – glad you found it useful. Cheers,
Eugen.

0

2 years 9 months ago

Load More Comments (http://www.baeldung.com/spring-requestmapping?wpdParentID=8197)

CATEGORIES

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

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)