Working Doc. Spring6 MVC constructs focused on the RESTful API

written by Hong Le Nguyen , last update 12.2024

Code to examples on Github :
https://github.com/hong1234/spring-boot3-mvc-jdbc-restApi
https://github.com/hong1234/spring-boot3-mvc-jpa-restApi

RestController ----
The @RestController annotation *applied at the class level* enables request mapping, request input binding, exception handling, and more.

```
@RestController
public class BookController {
   // ...
}
```

1)
Mapping (http-)request to endpoints (controller method)

@RequestMapping ---

You can use the @RequestMapping annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.

You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

```
// Get an book by ID
@RequestMapping(value = "/books/{id}", method = RequestMethod.GET)
public Book getBookById(@PathVariable("id") Integer id) {
   // ...
}
```

There are also HTTP method specific shortcut variants of @RequestMapping:

```
@GetMapping       for GET        (http method)
@PostMapping      for POST
@PutMapping       for PUT
@DeleteMapping    for  DELETE
@PatchMapping     for PATCH
```

```
// Get an book by ID using shortcut for GET
@GetMapping("/books/{id}")
public Book getBookById(@PathVariable("id") Integer id) {
   return bookService.getBookById(id);
}
```

When applied at the class level, it applies to all handler methods as well.

```
@RestController
@RequestMapping("/api")
public class BookController {
```

In this case, it is marked with "/api", meaning that all the methods have this prefix

2)
Send return value to client

@ResponseBody

You can use the @ResponseBody annotation on a method to have *the return serialized to the response body through an HttpMessageConverter*. The following listing shows an example:

```
// Get an book by ID
@GetMapping("/books/{id}", produces = "application/json")
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public Book getBookById(@PathVariable("id") Integer id) {
   return bookService.getBookById(id);
}
```

@ResponseBody is also supported *at the class level*, in which case it is inherited by all controller methods.

```
@Controller
@ResponseBody
@RequestMapping("/api")
public class BookController {
```

@RestController is a meta-annotation marked with @Controller and @ResponseBody. Therefore, the above code is equivalent to

```
@RestController
@RequestMapping("/api")
public class BookController {
```

The @RestController also causes that the return value in the controller methods is serialized and written to the http- response body by an **HttpMessageConverter** => more 3b) page 6.
The *"Accept" header* in request is used to determine the format of data to be written in response body and the appropriate *HttpMessageConverter* that serializes the return value.

It's important to know that by default Spring Boot will produce an output in JSON format.
You can specify it explicitly by using *the "produces" attribute of @RequestMapping annotation*. For example:

```
@RestController
@RequestMapping("/api", produces="application/json")
public class BookController {
   @GetMapping("/books/{id}")
   @ResponseStatus(HttpStatus.OK)
   public Book getBookById(@PathVariable("id") Integer id) {
      return bookService.getBookById(id);
   }
```

The attribute produces="application/json" in the above code ensures that
the client should send a request with an Accept header = "application/json" and
the return value (Book object) will be converted to *JSON format* by the suitable converter.

The client will specify the "Accept" header to "application/json" in the http request :

```
curl --header "Accept: application/json" http://localhost:8000/api/books/1
```

@ResponseStatus
Normally, this annotation is used when a method has a void return type (or null return value). This annotation sends back the HTTP status code specified in the response.

3)
Binding data (from an incoming request) to to method arguments

@PathVariable
is used to retrieve data from the URL path.

```
@GetMapping("/books/{id}")
public Book getBookById( @PathVariable("id") Integer id ) {
```

@RequestParam
using the @RequestParam annotation to bind query parameters to a method argument in a controller.
Query parameters are the key-value pairs that appear after the ? in a request URL.

```
const searchUrl = 'http://localhost:8000/api/books/search?title=spring';

@GetMapping("/books/search")
@ResponseStatus(HttpStatus.OK)
public Iterable<Book> searchBooksByTitle( @RequestParam("title") String title ) {
   return bookService.searchBooksByTitle(title);
}
```

@RequestHeader
You can use the @RequestHeader annotation to bind a request header to a method argument in a controller.

The following example shows a request with headers:

```
Host                localhost:8000
Accept              text/html
Accept-Language     en;q=0.3
Accept-Encoding     gzip,deflate
…
```

The following example gets the value of the Accept-Encoding header:

```
@GetMapping("/demo")
public void handle( @RequestHeader("Accept-Encoding") String encoding ) {
   //...
}
```

@RequestBody

applying the annotation on the argument of a Controller method to indicate that *the http Request body is deserialized to a particular Java object* by a HttpMessageConverter => more 3b) page 6.
The *"Content-Type" header* in request is used to determine the format of data in request body and the corresponding HttpMessageConverter.

Assuming that the “Content-Type“ header in request is "application/json":

```
curl -i -X POST -H "Content-Type: application/json"  -d '{"title":"test", "desc":"test"}'
http://localhost:8000/api/books
```

In this case the @RequestBody annotation ensures that JSON in the request body is bound to the Book object

```
@RestController
@RequestMapping(path="/api", produces="application/json")
class BookController {

  @PostMapping(path="/books", consumes="application/json")
  @ResponseStatus(HttpStatus.CREATED)
  public void addBook( @RequestBody Book book ) {
    // ...
  }
}
```

Consumable Media Types
The *attribute consumes*="application/json" ensures that the client sends a request with the "Content-Type" header "application/json" and JSON data in a request body.

4)
Validating the target argument object

@Valid

Applying the @Valid annotation to the argument of method in the Controller class tells Spring MVC to perform validation on the target argument object *after* it's bound to the data from request and *before* the method is called.

```
import jakarta.validation.Valid;

@RestController
@RequestMapping(path="/api", produces="application/json")
public class BookController {
  ...

  @PostMapping("/books", consumes="application/json")
  public Book addBook( @Valid @RequestBody Book book ){
    return bookService.addBook(book);
  }
}
```

When the target argument fails to pass the validation, Spring throws a *MethodArgumentNotValidException* exception.

To trigger a validator, it is necessary to annotate the data you want to validate with *the validation annotations* , for example @NotBlank, @Size for **JSR-380 bean validation** => more 4b) page 7.

```
import jakarta.validation.constraints.*;

public class Book {
  ...

  @NotBlank(message = "Title is mandatory")
  @Size(min = 3, max = 50, message = "must be min 3, and max 50 characters long")
  private String title;
}
```

5)
Handling exceptions thrown from a @RestController

@ExceptionHandler

The @ExceptionHandler annotated method is *invoked when the specified exceptions are thrown from a @RestController* or @Controller.
We can define these methods either in a @RestController marked class or in a @RestControllerAdvice marked class.

```
@RestController
@RequestMapping(path="/api")
public class BookController {
   ...

   @ExceptionHandler(ResourceNotFoundException.class)
   public ErrorDetails resourceNotFoundException(ResourceNotFoundException e) { ... }
}
```

The @RestControllerAdvice annotation is used to define a class that will handle *exceptions globally across all controllers*. Its methods are annotated with @ExceptionHandler annotation.

```
@RestControllerAdvice
public class GlobalExceptionHandler {
   ...

   @ExceptionHandler(ResourceNotFoundException.class)
   public ErrorDetails resourceNotFoundException(ResourceNotFoundException e) { ... }
}
```

By default when the DispatcherServlet can't find a handler for a request it sends *a 404 response*. However if its property "throwExceptionIfNoHandlerFound" is set to true the NoHandlerFoundException is raised. Using the following two properties will make spring boot throw NoHandlerFoundException:

```
// src/main/resources/application.properties
spring.mvc.throw-exception-if-no-handler-found=true
spring.web.resources.add-mappings=false
```

A list of exception handlers related REST controller

handler/URL not found exception (thrown by DispatcherServlet) handler

```
@ExceptionHandler(NoHandlerFoundException.class)
public ErrorDetails handlerNotFoundException(NoHandlerFoundException e) {
… ;  return errorDetails; }
```

binding exceptions (thrown by HttpMessageConverter) handler

```
@ExceptionHandler(HttpMessageNotReadableException.class)  // deserialize ex handler
public ErrorDetails validationException(HttpMessageNotReadableException e) {
...;  return errorDetails; }

@ExceptionHandler(HttpMessageNotWritableException.class)   // serialize ex handler
public ErrorDetails validationException(HttpMessageNotWritableException e) {  ... }
```

validation exceptions (thrown by validators) handler

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ErrorDetails handleValidationExceptions(MethodArgumentNotValidException e) {
...;  return errorDetails;}
```

handler for other Exceptions thrown from methods of @RestController

```
@ExceptionHandler
public ErrorDetails otherExceptions(Exception e) {  ...;  return errorDetails;}

@ExceptionHandler(ResourceNotFoundException.class)
 public ErrorDetails resourceNotFoundException(ResourceNotFoundException e) { … }
```

Note : we can collecte information about data-binding and validation errors in object Errors errors as follows

```
import org.springframework.validation.Errors;

@RestController
@RequestMapping(...)
public class BookController {

   @PostMapping(path="/books", consumes="application/json")
   public Book addBook( @Valid @RequestBody Book book, Errors errors ) {
      if (errors.hasErrors())
            throw new ValidationException(createErrorString(errors));
```

then we throw *our own exception*.

3b)

MappingJackson2HttpMessageConverter

implements HttpMessageConverter that can read and write JSON using Jackson 2.x's ObjectMapper. *By default, this converter supports application/json with UTF-8 character set*. This can be overridden by setting the supportedMediaTypes property.

Spring Mvc uses the methods of this converter

```
T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
```

to deserialize (JSON) data from the http-request body and bind it to typed java object.
A *HttpMessageNotReadableException* thrown by HttpMessageConverter implementation when the HttpMessageConverter.read(...) method fails.

```
void write(T t, MediaType contentType, HttpOutputMessage outputMessage)
```

to serialize the object to (JSON) data and write it to the http-response body.
A *HttpMessageNotWritableException* thrown by HttpMessageConverter implementation when the HttpMessageConverter.write(...) method fails.

MappingJackson2HttpMessageConverter configuration *in the Spring 6 MVC*

The following example adds Jackson JSON converter with *a customized ObjectMapper* instead of the default one. You use Jackson2ObjectMapperBuilder to create ObjectMapper object easily, and customize the ObjectMapper so that it can serialize the DateTime object to the string “DD-MM-YYYY HH:mm”.

```
@Configuration
@EnableWebMvc
public class WebMvcConfig implements WebMvcConfigurer {

   public static final String DATETIME_FORMAT = "dd-MM-yyyy HH:mm";

   @Bean
   public ObjectMapper objectMapper() {
      Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder();
      DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern(DATETIME_FORMAT);
      // serializers
      builder.serializers(new LocalDateTimeSerializer(dateTimeFormatter));
      builder.serializationInclusion(JsonInclude.Include.NON_NULL);
      // deserializers
      builder.deserializers(new LocalDateTimeDeserializer(dateTimeFormatter));
      return builder.build();
   }

   @Bean
   public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {
      return new MappingJackson2HttpMessageConverter(objectMapper());
   }

   @Override
   public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
      onverters.add(mappingJackson2HttpMessageConverter());
   }

}
```

Note: *Spring Boot* autoconfigures and registers the MappingJackson2HttpMessageConverter bean. To customize the ObjectMapper bean, you just need to configure it explicitly:

```
@Configuration
public class HttpConverterConfig {
   ...
   @Bean
   @Primary
   public ObjectMapper objectMapper() { …
```

4b)

JSR-380 validator enable

The validation annotations defined in the jakarta.validation.constraints.* package trigger the JSR-380 validator. If a (JSR-380) Bean Validation is present on the classpath (for example, *Hibernate Validator*), the Validator bean enables it as a global validator for use (for example with @Valid on controller method arguments).

```
// pom.xml
<dependency>
   <groupId>org.hibernate.validator</groupId>
   <artifactId>hibernate-validator</artifactId>
   <version>${hibernate.validator.version}</version>
</dependency>
```

```
@Configuration
@EnableWebMvc
public class WebMvcConfig  implements WebMvcConfigurer {
   @Bean
   public Validator validator() {
      Validator validator = new LocalValidatorFactoryBean();
      return validator;
   }

   @Override
   public Validator getValidator() {
      return validator();
   }
}
```

Note: Spring Boot autoconfigures a default (hibernate-) validator if dependency *spring-boot-starter-validation* added to pom.xml file

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Custom validator

Generally, when we need to validate user input, Spring MVC offers standard predefined validators. However, when we need to validate a more particular type of input, we have the ability to create our *own custom validator*. Creating a custom validator entails rolling out our *own annotation* and using it in our model to enforce the validation rules.

So let's create our custom validator that checks status options. The status *must* be a string with one of the values *Low, Medium, or High*.

The New Annotation

```
@Target({FIELD, PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(validatedBy = StatusValidator.class)
public @interface StatusValidation {
   //error message
   public String message() default "Invalid status: must be Low, Medium or High";
   //represents group of constraints
   public Class<?>[] groups() default {};
   //represents additional information about annotation
   public Class<? extends Payload>[] payload() default {};
}
```

Creating a Validator

```
public class StatusValidator implements ConstraintValidator<StatusValidation, String> {
   public boolean isValid(String colorName, ConstraintValidatorContext cxt) {
      List list = Arrays.asList(new String[]{"Low","Medium","High"});
      return list.contains(colorName);
   }
}
```

Apply the validation annotation to the domain field

```
@Data
public class Review {

   @NotBlank(message = "Email is mandatory")
   @Email(message="must be valid")
   private String email;
   …

   @StatusValidation()  // throws a MethodArgumentNotValidException exception.
   private String likeStatus;

}
```

Validating the model Review in Spring MVC after binding

```
@RestController
@RequestMapping(path="/api", produces="application/json")
public class BookController {

   @PostMapping(path="/reviews /{bookId} ", consumes="application/json")
   public Review addBookReview(@PathVariable("bookId") Integer bookId,
                                                    @Valid @RequestBody Review review){
         return bookService.addReviewToBook(bookId, review);
   }
```

Note : When the target argument fails to pass the validation, in our case @StatusValidation, Spring throws a *MethodArgumentNotValidException* exception.

Note : Customizing SpringMVC configuration in Spring Boot project

Suppose you want to take advantage of Spring Boot's autoconfiguration and add some MVC configuration (interceptors, formatters, view controllers, etc.). If so, you can create a configuration class *without* the @EnableWebMvc annotation, which implements WebMvcConfigurer and supplies additional configuration.

```
@Configuration
public class WebConfig implements WebMvcConfigurer
{
```