# Spring6 MVC constructs focused on the RESTful API

Autor : Hong Le Nguyen     Update : **05.2024**

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

## @RestController annotation

regarding the REST, the annotation tells Spring that all handler methods in the controller should have their *return value serialized and written directly to the body of the HTTP Response*. The *"Accept" header* in request will be used to determine the data format in response body and the appropriate *HttpMessageConverter* that serializes the return value.

For example

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

        @GetMapping("/books/{bookId}")
        public Book getBookById(@PathVariable("bookId") Integer bookId){
                return bookService.getBookById(bookId);
        }
```

The attribute produces="application/json" ensures that the client should send a request with an "Accept" header "application/json" and the return value should be converted to *JSON format* by the converter.

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

```
curl --header "Accept: application/json" http://localhost:8080/api/books/1
```

## @RequestBody annotation

applying the annotation on the argument of a Controller method to indicate that *the body of the HTTP Request is deserialized to a particular Java object* by a HttpMessageConverter. The *"Content-Type" header* in the request will be used to determine the data format 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:8080/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")
        public void addBook(@RequestBody Book book) {
                // ...
        }
```

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.

## 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() { ...
```

## @Valid annotation

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(path="/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-349 bean validation).

```
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;
```

## JSR-349 validator enable

The validation annotations defined in the jakarta.validation.constraints.* package trigger the JSR-349 validator. If a (JSR-349) 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.

## @ExceptionHandler Annotation

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

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

        public Book addBook(@Valid @RequestBody Book book, Errors errors){
                if (errors.hasErrors())
                        throw new ValidationException(createErrorString(errors));
```

then we throw *our own exception*.