6-Spring Web MVC Annotations

Author: Vincent Lau

Learning Objectives

- List and describe the use of various Spring Web annotations.
- Apply various Spring Web annotations to implement RESTful controllers.
- Implement a centralized exception handler to handle exceptions.

Overview

- In this tutorial, we'll explore Spring Web annotations with a concrete example.
- In the example, we'll create a RESTful controller that allows users to retrieve and persist book data.

Domain Model

- This is what the *Book* class looks like. Note that we use the static *idCount* field to simulate the auto-incrementing book ID generator.
- `AtomicLong` is used for **thread-safety** purpose.

```java
public class Book {
    private static final AtomicLong idCount = new AtomicLong(0);
    private final Long id;
    private final String title;
    private final String author;
    private final Integer page;

    public Book(String title, String author, Integer page) {
        this.id = idCount.incrementAndGet();
        this.title = title;
        this.author = author;
        this.page = page;
    }

          // getters and other methods
}
```

The Book Controller

- We define a bean called *BookController* with `@RestController` and `@RequestMapping`.

@RestController

- Recall the `@RestController` annotation combines `@Controller` and `@ResponseBody`

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

@RequestMapping

- Here we are applying `@RequestMapping` on a class level. It defines the **default settings for all handler methods** in a `@Controller` class.

- The only exception is the URL, which Spring **won't override** with method-level settings, but **appends the two path parts**.

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

    // http://localhost:8080/books/{bookId}
    @GetMapping("/{bookId}")
    public Book getBookById(...) {
                            ...
    }

}
```

@GetMapping

- Note that we're using `@GetMapping`, which is a shortcut for `@RequestMapping`. The equivalent is:

```java
@RequestMapping(value = "/{bookdId}", method = RequestMethod.GET)
public Book getBookById(...) {
                ...
}
```

Mock Data

- We've predefined some mock data in the *BookController* for retrieval purpose.
- Note that we're using *`ConcurrentHashMap`* to store the *Book* objects for thread-safety purpose because an application can receive multiple requests simultaneously, which will spawn multiple threads to retrieve or persist data at the same time.
- The *BookController* is also a Spring bean. With the `@PostConstruct` annotation, the *BookController* will initialize and populate the *ConcurrentHashMap* with mock data right after the bean has been created.

```java
@RestController
@RequestMapping("/books")
public class BookController {
    private final Map<Long, Book> bookmap = new ConcurrentHashMap<>();

    @PostConstruct
    private void init() {
        Book b1 = new Book("Spring in Action", "Craig Walls", 521);
        Book b2 = new Book("Spring Boot in Action", "Craig Walls", 266);
        Book b3 = new Book("Thinking in Java", "Bruce Eckel", 1079);
        Book b4 = new Book("On Java 8", "Bruce Eckel", 1778);
        Book b5 = new Book("Effective Java", "Joshua Bloch", 413);

        bookmap.put(b1.getId(), b1);
        bookmap.put(b2.getId(), b2);
        bookmap.put(b3.getId(), b3);
        bookmap.put(b4.getId(), b4);
        bookmap.put(b5.getId(), b5);
    }

                ...
}
```

The GET Endpoints

- In this example, we'll create two GET endpoints:
    - Finding books by keyword
    - Finding books by book ID
- Although it is possible to combine the two endpoints into one, we've separated them for clarity purpose.

Find Books by Keyword

- This endpoint accepts an optional *keyword* parameter.
- When the *keyword* parameter is not provided, we will return all the available books in the repository.
- When it is provided, we will compare the keyword against the *title* and *author name* of each book, and return the matching results.

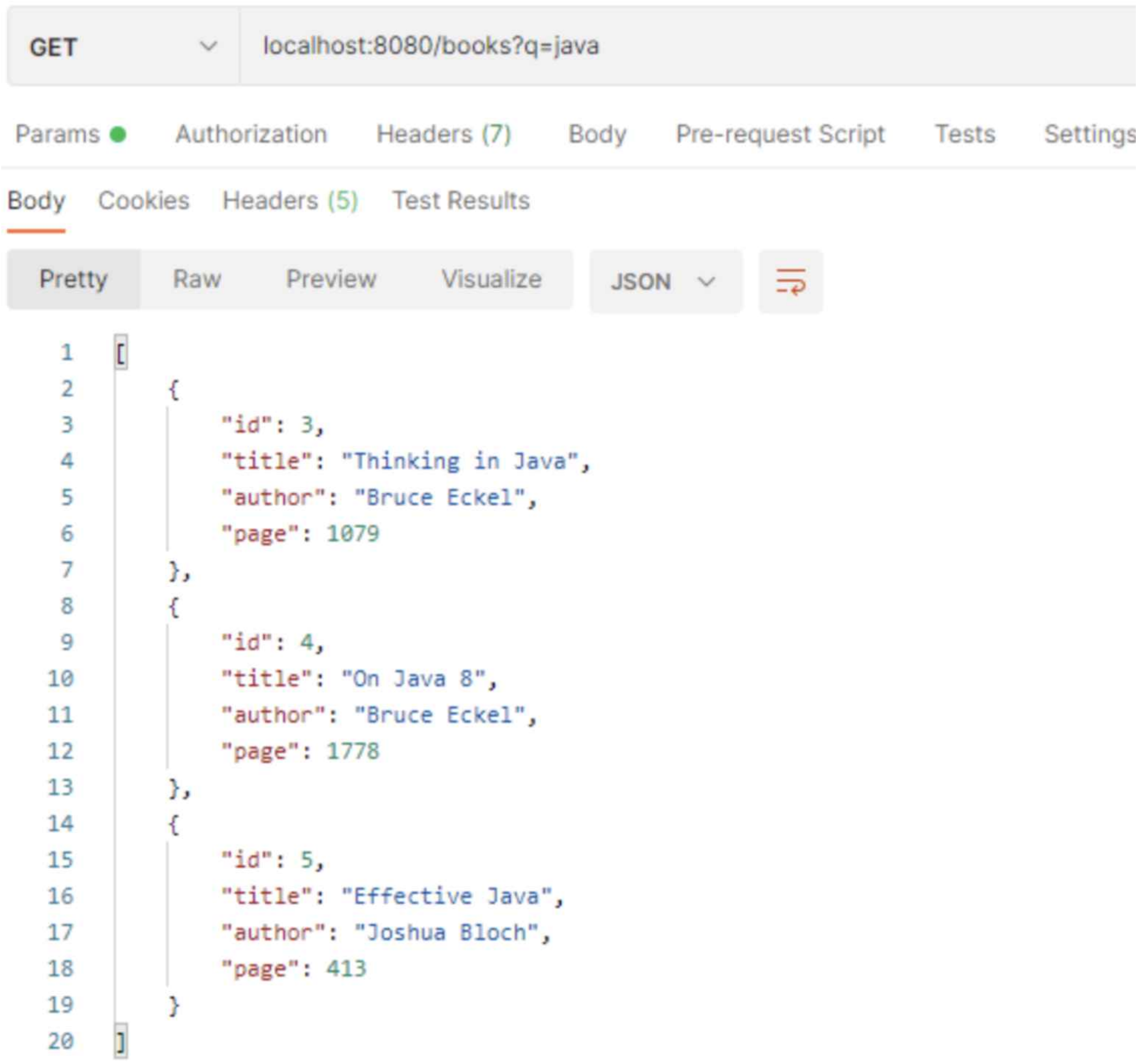
```
// http://localhost:8080/books?q=Java
@GetMapping
public List<Book> findBooks(@RequestParam(name = "q", required = false) String
keyword) {
```

```java
    if (Strings.isBlank(keyword)) {
        return new ArrayList<>();
    }

    final Predicate<Book> keywordMatcher = (book) ->
        book.getTitle().toLowerCase().contains(keyword.toLowerCase())
            || book.getAuthor().toLowerCase().contains(keyword.toLowerCase());

        return bookmap.values().stream()
            .filter(keywordMatcher)
            .collect(Collectors.toList());
}
```

@RequestParam

- We use `@RequestParameter` to access HTTP request parameters.
- The parameter which is marked with this annotation will be injected value coming from the request.
- The equivalent of this is:

```java
@GetMapping
public List<Book> findBooks(HttpServletRequest request) {
    String keyword = request.getParameter("q");
                ...
}
```



```
localhost:8080/books?q=java
```

```json
[
    {
        "id": 3,
        "title": "Thinking in Java",
        "author": "Bruce Eckel",
        "page": 1079
    },
    {
        "id": 4,
        "title": "On Java 8",
        "author": "Bruce Eckel",
        "page": 1778
    },
    {
        "id": 5,
        "title": "Effective Java",
        "author": "Joshua Bloch",
        "page": 413
    }
]
```

Find Books by ID

- Here we use `@PathVariable` to extract the *bookId* from the URL so that we can use it to retrieve the corresponding book.

```java
// http://localhost:8080/books/{bookId} -> e.g. http://localhost:8080/books/1
@GetMapping("/{bookId}")
public Book getBookById(@PathVariable(name = "bookId") Long bookId) {
    return bookRepository.get(bookId);
}
```

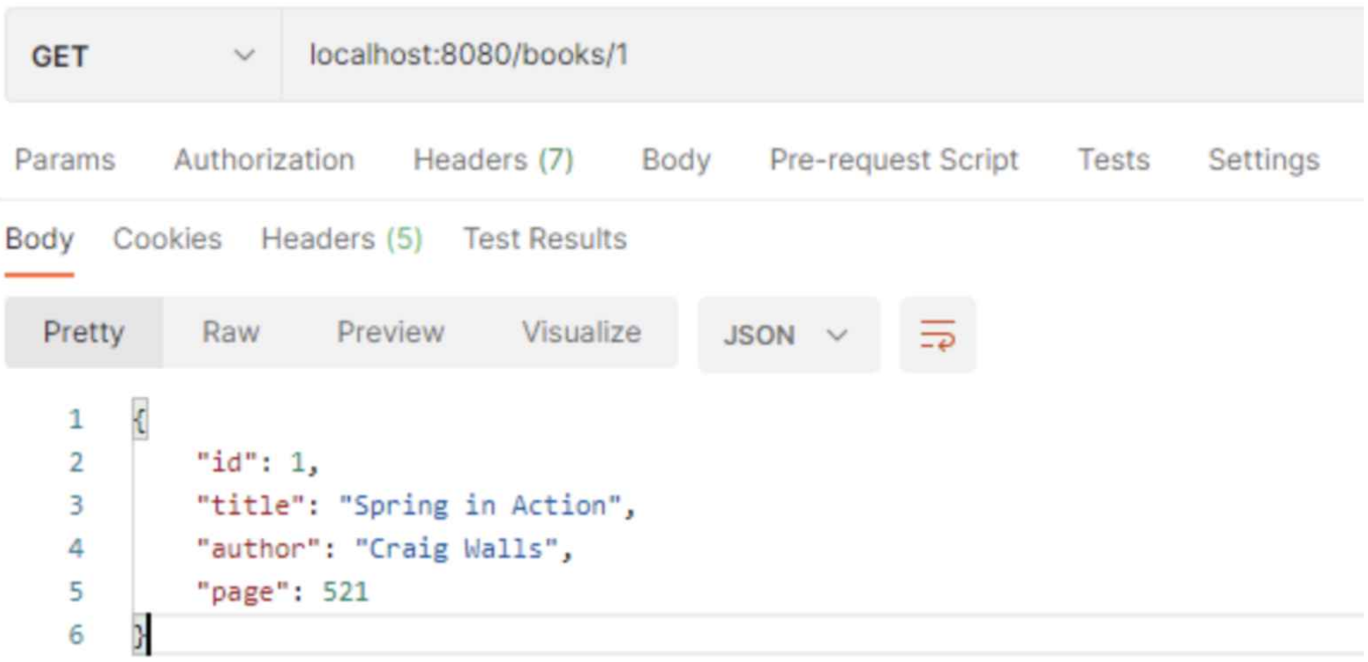
@PathVariable

- The `@PathVariable` is used to map a URL path value to a parameter of the handling method.
- Moreover, we can mark a path variable optional by setting the argument *required* to false:

```java
@GetMapping("/{bookId}")
public Book getBookById(@PathVariable(name = "bookId", required = false) Long bookId) {
                ...
}
```

@RequestParam vs @PathVariable

- Even though both are used to extract data from a URL,
  - `@RequestParam` is used to retrieve *query parameters* (i.e. anything after `?` in the URL)
  - `@PathVariable` is used to retrieve *values from the URL* itself



```json
{
    "id": 1,
    "title": "Spring in Action",
    "author": "Craig Walls",
    "page": 521
}
```

The POST Endpoint

- Let's add a POST endpoint for adding new books.

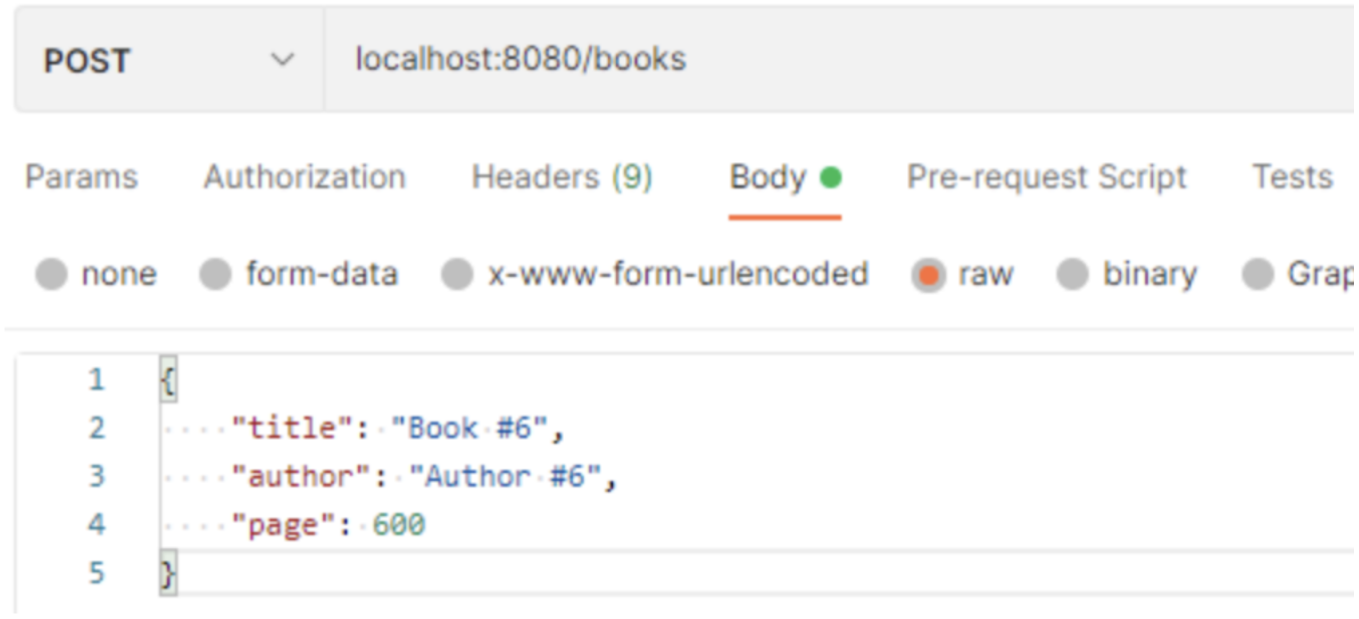
```
/*
http://localhost:8080/books

{
    "title": "Sample Book",
    "author": "Sample Author",
    "page": 500
}

*/
@PostMapping
public ResponseEntity<Void> addBook(@RequestBody Book book) {
    bookRepository.put(book.getId(), book);
    return ResponseEntity.ok().build();
}
```

@PostMapping

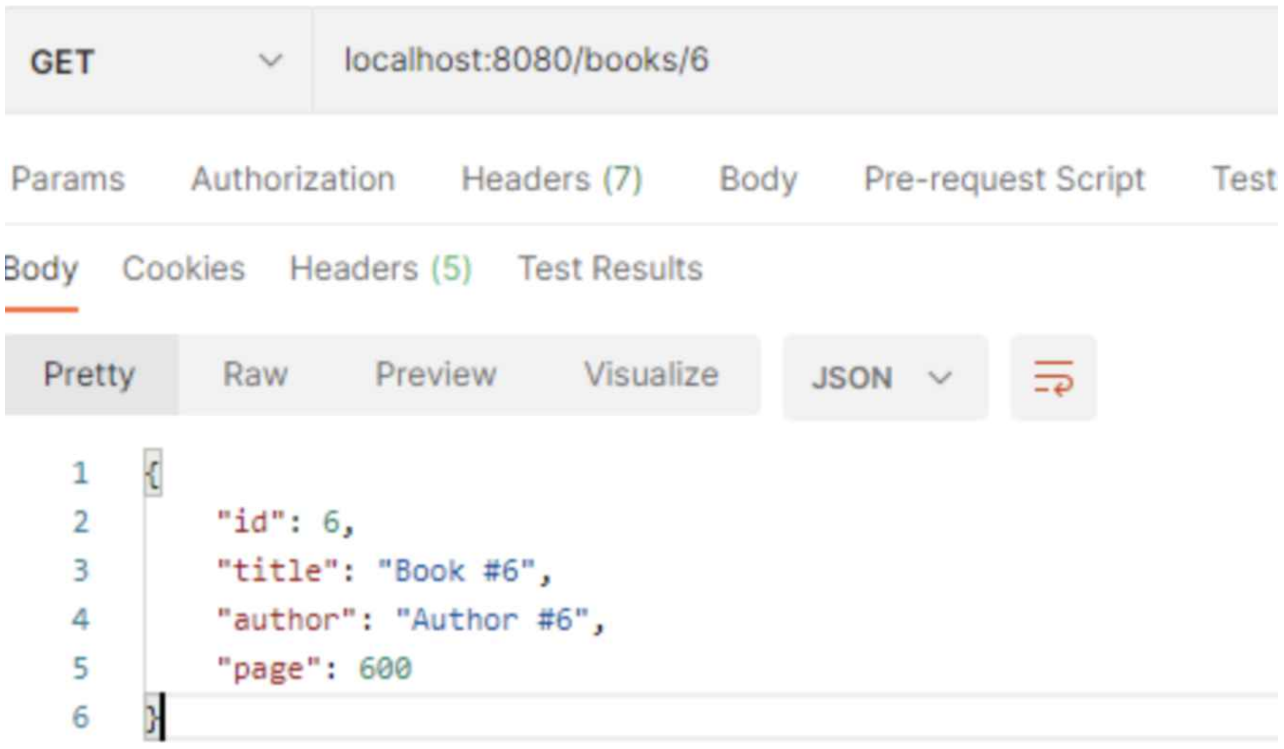
- Note that we apply the `@PostMapping` annotation to map HTTP POST requests to the corresponding handling method.
- Specifically, `@PostMapping` is a shortcut for `@RequestMapping(method = RequestMethod.POST)` .

@RequestBody

- The `@RequestBody` annotation maps the body of the HTTP request to an object.
- The deserilziation is automatic and depends on the content type of the request



```
{
    "title": "Book #6",
    "author": "Author #6",
    "page": 600
}
```



localhost:8080/books/6

```
{
    "id": 6,
    "title": "Book #6",
    "author": "Author #6",
    "page": 600
}
```

Exception Handling

- Requests sent to an application will not always succeed. An application should be able to handle exceptions when they occur.
- One of the scenarios to throw an exception is when the requesting *book ID* does not exist in the repository.
- For this, we define a custom *ResourceNotFoundException* class:

```
public class ResourceNotFoundException extends Exception { //check exception
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

- The GET endpoint for finding books by ID should be updated to throw an exception in case of non-existent book ID:

```
@GetMapping("/{bookId}")
public Book getBookById(@PathVariable(name = "bookId") Long bookId)
throws ResourceNotFoundException {
    if (!bookRepository.containsKey(bookId)) {
        throw new ResourceNotFoundException(
            String.format("Book [%s] not found", bookId));
```

```java
    }
    return bookRepository.get(bookId);
}
```

- We should define a global exception handler that handles various types of exceptions that will be potentially thrown in the application.

@ControllerAdvice

- The `@ControllerAdvice` annotation is used to indicate a class which is responsible for handling exceptions thrown when processing client requests.
- `@ControllerAdvice` mostly applies to the MVC architecture pattern. For RESTful services, `@RestControllerAdvice` is a better idea.

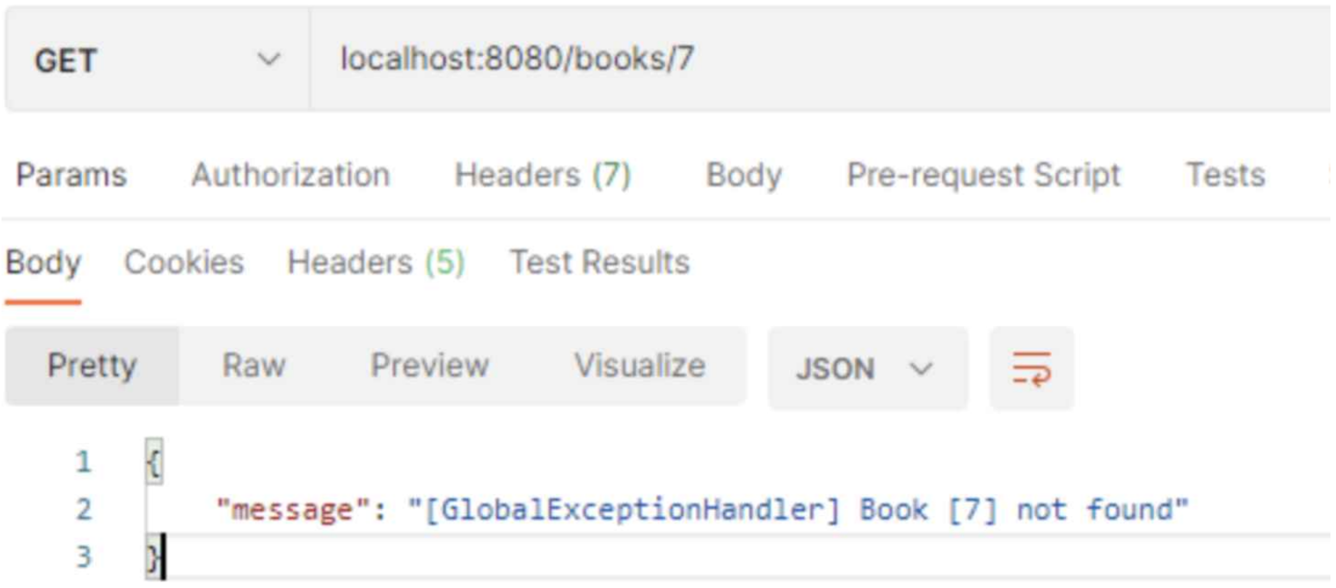
@RestControllerAdvice

- Although `@RestControllerAdvice` has similar responsibility as `@ControllerAdvice`, the returning result will be serialized to JSON or XML.
- In fact, `@RestControllerAdvice` is a combination of `@ControllerAdvice` and `@ResponseBody`.

@ExceptionHandler

- In the global exception handler, we can specify a method to handle a particular set of exceptions.
- An exception handler can contain several of these methods. We apply `@ExceptionHandler` to the handling methods.
- Spring calls these exception handling methods when any of the specified exceptions is thrown. The caught exception can be passed to the method as an argument:

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(ResourceNotFoundException.class)
    public ErrorResult handleException(ResourceNotFoundException ex) {
        return new ErrorResult("[GlobalExceptionHandler] " + ex.getMessage());
    }

}

public class ErrorResult {
    private final String message;

    public ErrorResult(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```



```json
{
    "message": "[GlobalExceptionHandler] Book [7] not found"
}
```

@ResponseStatus

- We can specify the desired HTTP status of the response if we annotate a request handler method with this annotation.
- We can declare the status code with the *code* argument, or its alias, the *value* argument.
- Also, we can provide a reason using the *reason* argument.

```java
@ResponseStatus(code = HttpStatus.NOT_FOUND, reason = "Book Not Found")
...
```

Other @RequestMapping Shortcuts

- There are a few more shortcuts for `@RequestMapping` which have not been covered here. To understand more, read this.

  - `@PutMapping` (Make sure you know the difference between PutMapping and PostMapping)
  - `@DeleteMapping`
  - `@PatchMapping`

Questions

- What are the different ways to extract data from a request URL?
- What is the difference between `@Controller` and `@RestController`?
- What is the difference between `@ControllerAdvice` and `@RestControllerAdvice`?
- How do we implement an exception handler in Spring Web MVC?