(/)

Spring Boot Consuming and Producing JSON

Last modified: June 30, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

JSON (https://www.baeldung.com/category/json/)

Spring Boot (https://www.baeldung.com/category/spring/spring-boot/)

1. Overview

In this tutorial, we'll show **how to build a REST service (/rest-with-spring-series) to consume and produce JSON content with Spring Boot**.

We'll also take a look at how we can easily employ RESTful HTTP semantics.

For simplicity, we won't include a persistence layer (/the-persistence-layer-with-spring-and-jpa), but Spring Data (/the-persistence-layer-with-spring-data-jpa) makes this simple to add, too.

2. REST Service

Writing a JSON REST service in Spring Boot is simple, as that's its default opinion when Jackson (/jackson) is on the classpath:

```
@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentService service;

    @GetMapping("/{id}")
    public Student read(@PathVariable String id) {
        return service.find(id);
    }

...
```

By annotating our *StudentController* with *@RestController* (/spring-controller-vs-restcontroller), **we've told Spring Boot to write the return type of the *read* method** to the response body. **Since we also have a *@RequestMapping* at the class level**, it'd be the same for any more public methods that we add.

Though simple, **this approach lacks HTTP semantics.** For example, what should happen if we don't find the requested student? Instead of returning a 200 or a 500 status code, we might want to return a 404.

Let's take a look at how to wrest more control over the HTTP response itself and in turn add some typical RESTful behaviors to our controller.

3. Create

When we need to control aspects of the response other than the body – like the status code – we can instead return a *ResponseEntity*:

```
@PostMapping("/")
public ResponseEntity<Student> create(@RequestBody Student student)
    throws URISyntaxException {
    Student createdStudent = service.create(student);
    if (createdStudent == null) {
        return ResponseEntity.notFound().build();
    } else {
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest()
          .path("/{id}")
          .buildAndExpand(createdStudent.getId())
          .toUri();

        return ResponseEntity.created(uri)
          .body(createdStudent);
    }
}
```

Here, we're doing much more than just returning the created *Student* in the response. **Additionally, we respond with a semantically clear HTTP status and, if creation succeeded, a URI to the new resource.**

4. Read

As mentioned before, if we want to read a single *Student*, it's more semantically clear to return a 404 if we can't find the student:

```
@GetMapping("/{id}")
public ResponseEntity<Student> read(@PathVariable("id") Long id) {
    Student foundStudent = service.read(id);
    if (foundStudent == null) {
        return ResponseEntity.notFound().build();
    } else {
        return ResponseEntity.ok(foundStudent);
    }
}
```

Here, we can clearly see the difference from our initial *read()* implementation.

This way the *Student* object will be properly mapped to the response body and returned with a proper status at the same time.

5. Update

Updating is very similar to creation, except it's mapped to PUT instead of POST, and URI contains an *id* of the resource we're updating:

```
@PutMapping("/{id}")
public ResponseEntity<Student> update(@RequestBody Student student, @PathVariable Long id) {
    Student updatedStudent = service.update(id, student);
    if (updatedStudent == null) {
        return ResponseEntity.notFound().build();
    } else {
        return ResponseEntity.ok(updatedStudent);
    }
}
```

6. Delete

The delete operation is mapped to the DELETE method. URI also contains the *id* of the resource:

```
@DeleteMapping("/{id}")
public ResponseEntity<Object> deleteStudent(@PathVariable Long id) {
    service.delete(id);
    return ResponseEntity.noContent().build();
}
```

We didn't implement specific error handling (/exception-handling-for-rest-with-spring), because the *delete()* method actually fails by throwing an *Exception.*

7. Conclusion

In this article, we saw how to consume and produce JSON content in a typical CRUD REST service developed with a Spring Boot. Additionally, we demonstrated how to implement proper response status control and error handling.

To keep things simple, we didn't go into persistence this time, but Spring Data REST (/spring-data-rest-intro) provides a quick and efficient way to build a RESTful data service.

The complete source code for the example is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-mvc-2).

Comments are closed on this article!

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)