(/)

RestTemplate Post Request with JSON

Last modified: December 30, 2020

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

HTTP Client-Side (https://www.baeldung.com/category/http/)

REST (https://www.baeldung.com/category/rest/)

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

RestTemplate (https://www.baeldung.com/tag/resttemplate/)

1. Introduction

In this quick tutorial, we illustrate how to use Spring's *RestTemplate* (/rest-template) to make POST requests sending JSON content.

Further reading:

Exploring the Spring Boot TestRestTemplate (/spring-boot-testresttemplate)

Learn how to use the new TestRestTemplate in Spring Boot to test a simple API.

Read more (/spring-boot-testresttemplate) →

Spring RestTemplate Error Handling (/spring-rest-template-error-handling)

Learn how to handle errors with Spring's RestTemplate

Read more (/spring-rest-template-error-handling) →

2. Setting Up the Example

Let's start by adding a simple *Person* model class to represent the data to be posted:

```
public class Person {
    private Integer id;
    private String name;

    // standard constructor, getters, setters
}
```

To work with *Person* objects, we'll add a *PersonService* interface and implementation with two methods:

```
public interface PersonService {

    public Person saveUpdatePerson(Person person);
    public Person findPersonById(Integer id);
}
```

The implementation of these methods will simply return an object. We're using a dummy implementation of this layer here so we can focus on the web layer.

3. REST API Setup

Let's define a simple REST API for our *Person* class:

```
@PostMapping(
  value = "/createPerson", consumes = "application/json", produces = "application/json")
public Person createPerson(@RequestBody Person person) {
    return personService.saveUpdatePerson(person);
}

@PostMapping(
  value = "/updatePerson", consumes = "application/json", produces = "application/json")
public Person updatePerson(@RequestBody Person person, HttpServletResponse response) {
    response.setHeader("Location", ServletUriComponentsBuilder.fromCurrentContextPath()
      .path("/findPerson/" + person.getId()).toUriString());

    return personService.saveUpdatePerson(person);
}
```

Remember, we want to post the data in JSON format. In order to that, **we added the *consumes* attribute in the *@PostMapping* annotation with the value of "application/json"** for both methods.

Similarly, we set the *produces* attribute to "application/json" to tell Spring that we want the response body in JSON format.

We annotated the *person* parameter with the *@RequestBody* (/spring-request-response-body) annotation for both methods. This will tell Spring that the *person* object will be bound to the body of the *HTTP* request.

Lastly, both methods return a *Person* object that will be bound to the response body. Let's note that we'll annotate our API class with *@RestController* (/spring-controller-vs-restcontroller) to annotate all API methods with a hidden *@ResponseBody* (/spring-request-response-body) annotation.

4. Using *RestTemplate*

Now we can write a few unit tests to test our *Person* REST API. Here, **we'll try to send POST requests to the *Person* API by using the POST methods provided by the *RestTemplate*: *postForObject*, *postForEntity*, and *postForLocation*.**

Before we start to implement our unit tests, let's define a setup method to initialize the objects that we'll use in all our unit test methods:

```
@BeforeClass
public static void runBeforeAllTestMethods() {
    createPersonUrl = "http://localhost:8082/spring-rest/createPerson";
    updatePersonUrl = "http://localhost:8082/spring-rest/updatePerson";

    restTemplate = new RestTemplate();
    headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    personJsonObject = new JSONObject();
    personJsonObject.put("id", 1);
    personJsonObject.put("name", "John");
}
```

Besides this setup method, note that we'll refer to the following mapper to convert the JSON String to a *JSONNode* object in our unit tests:

```
private final ObjectMapper objectMapper = new ObjectMapper();
```

As previously mentioned, **we want to post the data in JSON format. To achieve this, we'll add a *Content-Type* header to our request with the *APPLICATION_JSON* media type.**

Spring's *HttpHeaders* class provides different methods to access the headers. Here, we set the *Content-Type* header to *application/json* by calling the *setContentType* method. We'll attach the *headers* object to our requests.

4.1. Posting JSON With *postForObject*

RestTemplate's *postForObject* method creates a new resource by posting an object to the given URI template. It returns the result as automatically converted to the type specified in the *responseType* parameter.

Let's say that we want to make a POST request to our *Person* API to create a new *Person* object and return this newly created object in the response.

First, we'll build the *request* object of type *HttpEntity* based on the *personJsonObject* and the headers containing the *Content-Type*. This allows the *postForObject* method to send a JSON request body:

```
@Test
public void givenDataIsJson_whenDataIsPostedByPostForObject_thenResponseBodyIsNotNull()
  throws IOException {
    HttpEntity<String> request =
      new HttpEntity<String>(personJsonObject.toString(), headers);

    String personResultAsJsonStr =
      restTemplate.postForObject(createPersonUrl, request, String.class);
    JsonNode root = objectMapper.readTree(personResultAsJsonStr);

    assertNotNull(personResultAsJsonStr);
    assertNotNull(root);
    assertNotNull(root.path("name").asText());
}
```

The *postForObject()* method returns the response body as a *String* type.

We can also return the response as a *Person* object by setting the *responseType* parameter:

```
Person person = restTemplate.postForObject(createPersonUrl, request, Person.class);

assertNotNull(person);
assertNotNull(person.getName());
```

Actually, our request handler method matching with the *createPersonUrl* URI produces the response body in JSON format.

But this is not a limitation for us — *postForObject* is able to automatically convert the response body into the requested Java type (e.g. *String*, *Person*) specified in the *responseType* parameter.

4.2. Posting JSON With *postForEntity*

Compared to *postForObject()*, ***postForEntity()* returns the response as a *ResponseEntity* (/spring-response-entity) object.** Other than that, both methods do the same job.

Let's say that we want to make a POST request to our *Person* API to create a new *Person* object and return the response as a *ResponseEntity*.

We can make use of the *postForEntity* method to implement this:

```
@Test
public void givenDataIsJson_whenDataIsPostedByPostForEntity_thenResponseBodyIsNotNull()
  throws IOException {
    HttpEntity<String> request =
      new HttpEntity<String>(personJsonObject.toString(), headers);

    ResponseEntity<String> responseEntityStr = restTemplate.
      postForEntity(createPersonUrl, request, String.class);
    JsonNode root = objectMapper.readTree(responseEntityStr.getBody());

    assertNotNull(responseEntityStr.getBody());
    assertNotNull(root.path("name").asText());
}
```

Similar to the *postForObject*, *postForEntity* has the *responseType* parameter to convert the response body to the requested Java type.

Here, we were able to return the response body as a *ResponseEntity<String>*.

We can also return the response as a *ResponseEntity<Person>* object by setting the *responseType* parameter to *Person.class*:

```
ResponseEntity<Person> responseEntityPerson = restTemplate.
  postForEntity(createPersonUrl, request, Person.class);

assertNotNull(responseEntityPerson.getBody());
assertNotNull(responseEntityPerson.getBody().getName());
```

4.3. Posting JSON With *postForLocation*

Similar to the *postForObject* and *postForEntity* methods, *postForLocation* also creates a new resource by posting the given object to the given URI. The only difference is that it returns the value of the *Location* header.

Remember, we already saw how to set the *Location* header of a response in our *updatePerson* REST API method above:

```
response.setHeader("Location", ServletUriComponentsBuilder.fromCurrentContextPath()
  .path("/findPerson/" + person.getId()).toUriString());
```

Now let's imagine that **we want to return the *Location* header of the response after updating the *person* object we posted.**

We can implement this by using the *postForLocation* method:

```
@Test
public void givenDataIsJson_whenDataIsPostedByPostForLocation_thenResponseBodyIsTheLocationHeader()
  throws JsonProcessingException {
    HttpEntity<String> request = new HttpEntity<String>(personJsonObject.toString(), headers);
    URI locationHeader = restTemplate.postForLocation(updatePersonUrl, request);

    assertNotNull(locationHeader);
}
```

5. Conclusion

In this article, we explored how to use *RestTemplate* to make a POST request with JSON.

As always, all the examples and code snippets can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-web-modules/spring-resttemplate-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/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

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)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
JOBS (/TAG/ACTIVE-JOB/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)