(/)

Get and Post Lists of Objects with RestTemplate

Last modified: July 13, 2018

by Michael Pratt (/author/michael-pratt/)

REST (/category/rest/) Spring (/category/spring/) + RestTemplate (/tag/resttemplate/)

1. Introduction

The *RestTemplate* class is the central tool for performing client-side HTTP operations in Spring. It provides several utility methods for building HTTP requests and handling responses.

And, since *RestTemplate* integrates well with Jackson, (https://github.com/FasterXML/jackson) it can serialize/deserialize most objects to and from JSON without much effort. However, **working with collections of objects is not so straightforward**.

In this tutorial, we'll see how to use *RestTemplate* to both *GET* and *POST* a list of objects.

2. Example Service

We will be using an employee API that has two HTTP endpoints – get all and create:

- *GET /employees*
- *POST /employees*

For communication between client and server, we'll use a simple DTO to encapsulate basic employee data:

```
public class Employee {
    public long id;
    public String title;

    // standard constructor and setters/getters
}
```

We're now ready to write code that uses *RestTemplate* to get and create lists of *Employee* objects.

3. Get a List of Objects with *RestTemplate*

Normally when calling GET, you can use one of the simplified methods in *RestTemplate*, such as*:*

getForObject(URI url, Class<T> responseType)

This sends a request to the specified URI using the GET verb and converts the response body into the requested Java type. This works great for most classes, but it has a limitation: **we cannot send lists of objects.**

The problem is due to type erasure with Java generics. When the application is running, it has no knowledge of what type of object is in the list. **This means the data in the list cannot be deserialized into the appropriate type.**

Luckily we have two options to get around this.

3.1. Using *ParameterizedTypeReference*

Using a combination of *ResponseEntity* and *ParameterizedTypeReference*, we can easily get a list of objects using *RestTemplate*:

```
RestTemplate restTemplate = new RestTemplate();
ResponseEntity<List<Employee>> response = restTemplate.exchange(
  "http://localhost:8080/employees/ (http://localhost:8080/employees/)",
  HttpMethod.GET,
  null,
  new ParameterizedTypeReference<List<Employee>>(){});
List<Employee> employees = response.getBody();
```

There are a couple of things happening in the code above. First, we use *ResponseEntity* as our return type, using it to wrap the list of objects we really want. Second, we are calling *RestTemplate.exchange()* instead of *getForObject()*.

This is the most generic way to use *RestTemplate*. It requires us to specify the HTTP method, optional request body, and a response type. In this case, we use an anonymous subclass of *ParameterizedTypeReference* for the response type.

This last part is what allows us to convert the JSON response into a list of objects that are the appropriate type. When we create an anonymous subclass of *ParameterizedTypeReference*, it uses reflection to capture information about the class type we want to convert our response to.

It holds on to this information using Java's *Type* object, and we no longer have to worry about type erasure.

The upside of this approach is that we can use all of our existing code. Our Employee object works as-is, as does our application endpoint. The downside is that the code is a little more verbose.

3.2. Using a Wrapper Class

Some APIs will return a top-level object that contains the list of employees instead of returning the list directly. To handle this situation, we can use a wrapper class that contains the list of employees.

```
public class EmployeeList {
    private List<Employee> employees;

    public EmployeeList() {
        employees = new ArrayList<>();
    }

    // standard constructor and getter/setter
}
```

Now we can use the simpler *getForObject()* method to get the list of employees:

```
EmployeeList response = restTemplate.getForObject(
  "http://localhost:8080/employees (http://localhost:8080/employees)",
  EmployeeList.class);
List<Employee> employees = response.getEmployees();
```

This code is much simpler but requires an additional wrapper object.

4. Post a List of Objects with *RestTemplate*

Now let's look at how to send a list of objects from our client to the server. Just like above, *RestTemplate* provides a simplified method for calling POST:

postForObject(URI url, Object request, Class<T> responseType)

This sends an HTTP POST to the given URI, with the optional request body, and converts the response into the specified type. Unlike the GET scenario above, **we don't have to worry about type erasure**.

This is because now we're going from Java objects to JSON. The list of objects and their type are known by the JVM, and therefore properly be serialized:

```
List<Employee> newEmployees = new ArrayList<>();
newEmployees.add(new Employee(3, "Intern"));
newEmployees.add(new Employee(4, "CEO"));

restTemplate.postForObject(
  "http://localhost:8080/employees/ (http://localhost:8080/employees/)",
  newEmployees,
  ResponseEntity.class);
```

4.1. Using a Wrapper Class

If we need to use a wrapper class to be consistent with GET scenario above, that's simple too. We can send a new list using *RestTemplate*:

```
List<Employee> newEmployees = new ArrayList<>();
newEmployees.add(new Employee(3, "Intern"));
newEmployees.add(new Employee(4, "CEO"));

restTemplate.postForObject(
  "http://localhost:8080/employees (http://localhost:8080/employees)",
  new EmployeeList(newEmployees),
  ResponseEntity.class);
```

5. Conclusion

Using RestTemplate is a simple way of building HTTP clients to communicate with your services.

It provides a number of methods for working with every HTTP method and simple objects. With a little bit of extra code, we can easily use it to work with lists of objects.

As usual, the complete code is available in the Github project (https://github.com/eugenp/tutorials/tree/master/spring-rest).

Start the discussion...

Subscribe

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JACKSON/)
HTTPCLIENT (/CATEGORY/HTTP/)
KOTLIN (/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 (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
CONTACT (/CONTACT)
EDITORS (/EDITORS)
MEDIA KIT (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)