HTTP PUT vs HTTP PATCH in a REST API

Last modified: November 6, 2018

by baeldung (/author/baeldung/)

1. Overview

In this quick article, we're looking at differences between the HTTP PUT and PATCH verbs and at the semantics of the two operations.

We'll use Spring to implement two REST endpoints that support these two types of operations, and to better understand the differences and the right way to use them.

2. When to use PUT and When PATCH?

Let's start with a simple, and slightly simple statement.

When a client needs to replace an existing Resource entirely, they can use PUT. When they're doing a partial update, they can use HTTP PATCH.

For instance, when updating a single field of the Resource, sending the complete Resource representation might be cumbersome and utilizes a lot of unnecessary bandwidth. In such cases, the semantics of PATCH make a lot more sense.

Another important aspect to consider here is **idempotence; PUT is idempotent; PATCH can be, but isn't required to**. And, so – depending on the semantics of the operation we're implementing, we can also choose one or the other based on this characteristic.

3. Implementing PUT and PATCH Logic

Let's say we want to implement the REST API for updating a *HeavyResource* with multiple fields:

```
public class HeavyResource {
    private Integer id;
    private String name;
    private String address;
    // ...
```

First, we need to create the endpoint that handles a full update of the resource using PUT:

```
@PutMapping("/heavyresource/{id}")
public ResponseEntity<?> saveResource(@RequestBody HeavyResource heavyResource,
  @PathVariable("id") String id) {
    heavyResourceRepository.save(heavyResource, id);
    return ResponseEntity.ok("resource saved");
}
```

This is a standard endpoint for updating resources.

Now, let's say that address field will often be updated by the client. In that case, **we don't want to send the whole *HeavyResource* object with all fields**, but we do want the ability to only update the *address* field – via the PATCH method.

We can create a *HeavyResourceAddressOnly* DTO to represent a partial update of the address field:

```
public class HeavyResourceAddressOnly {
    private Integer id;
    private String address;

    // ...
}
```

Next, we can leverage the PATCH method to send a partial update:

```
@PatchMapping("/heavyresource/{id}")
public ResponseEntity<?> partialUpdateName(
  @RequestBody HeavyResourceAddressOnly partialUpdate, @PathVariable("id") String id) {

    heavyResourceRepository.save(partialUpdate, id);
    return ResponseEntity.ok("resource address updated");
}
```

With this more granular DTO, we can send the field we need to update only – without the overhead of sending whole *HeavyResource*.

If we have a large number of these partial update operations, we can also skip the creation of a custom DTO for each out – and only use a map:

```
@RequestMapping(value = "/heavyresource/{id}", method = RequestMethod.PATCH, consumes = Med
public ResponseEntity<?> partialUpdateGeneric(
  @RequestBody Map<String, Object> updates,
  @PathVariable("id") String id) {

    heavyResourceRepository.save(updates, id);
    return ResponseEntity.ok("resource updated");
}
```

This solution will give us more flexibility in implementing API; however, we do lose a few things as well – such as validation.

4. Testing PUT and PATCH

Finally, let's write tests for both HTTP methods. First, we want to test the update of the full resource via PUT method:

```
mockMvc.perform(put("/heavyresource/1")
  .contentType(MediaType.APPLICATION_JSON_VALUE)
  .content(objectMapper.writeValueAsString(
    new HeavyResource(1, "Tom", "Jackson", 12, "heaven street")))
  ).andExpect(status().isOk());
```

Execution of a partial update is achieved by using the PATCH method:

```
mockMvc.perform(patch("/heavyrecource/1")
  .contentType(MediaType.APPLICATION_JSON_VALUE)
  .content(objectMapper.writeValueAsString(
    new HeavyResourceAddressOnly(1, "5th avenue")))
  ).andExpect(status().isOk());
```

We can also write a test for a more generic approach:

```
HashMap<String, Object> updates = new HashMap<>();
updates.put("address", "5th avenue");

mockMvc.perform(patch("/heavyresource/1")
    .contentType(MediaType.APPLICATION_JSON_VALUE)
    .content(objectMapper.writeValueAsString(updates))
  ).andExpect(status().isOk());
```

5. Handling Partial Requests With *Null* Values

When we are writing an implementation for a PATCH method, we need to specify a contract of how to treat cases when we get *null* as a value for the *address* field in the *HeavyResourceAddressOnly.*

Suppose that client sends the following request:

```
{
   "id" : 1,
   "address" : null
}
```

Then we can handle this as setting a value of the *address* field to *null* or just ignoring such request by treating it as no-change.

We should pick one strategy for handling *null* and stick to it in every PATCH method implementation.

6. Conclusion

In this quick tutorial, we focused on understanding the differences between the HTTP PATCH and PUT methods.

We implemented a simple Spring REST controller to update a Resource via PUT method and a partial update using PATCH.

The implementation of all these examples and code snippets can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/spring-rest) – this is a Maven project, so it should be easy to import and run as it is.