Sunday, 17 January, 2021

REST: Partial updates with PATCH

In previous posts we learned how to update/replace resources using the HTTP PUT operation (https://www.mscharhag.com/api-design/updating-resources-put). We also learned about the differences between POST, PUT and PATCH (https://www.mscharhag.com/api-design/http-post-put-patch). In this post we will now see how to perform partial updates with the HTTP PATCH method.

Before we start, let's quickly check why partial updates can be useful:

- Simplicity - If a client only wants to update a single field, a partial update request can be simpler to implement.
- Bandwidth - If your resource representations are quite large, partial updates can reduce the amount of bandwidth required.
- Lost updates - Resource replacements with PUT can be susceptible for the lost update problem. While partial updates do not solve this problem, they can help reducing the number of possible conflicts.

The PATCH HTTP method

Other like PUT or POST the PATCH method is not part of the original HTTP RFC. It has later been added via RFC 5789 (https://tools.ietf.org/html/rfc5789). The PATCH method is neither safe nor idempotent (https://www.mscharhag.com/api-design/http-idempotent-safe). However, PATCH it is often used in an idempotent way.

A PATCH request can contain one or more requested changes to a resource. If more than one change is requested the server must ensure that all changes are applied atomically. The RFC says:

> The server MUST apply the entire set of changes atomically and never provide ([..]) a partially modified representation. If the entire patch document cannot be successfully applied, then the server MUST NOT apply any of the changes.

The request body for PATCH is quite flexible. The RFC only says the request body has to contain instructions on how the resource should be modified:

> With PATCH, [..], the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.

This means we do not have to use the same resource representation for PATCH requests as we might use for PUT or GET requests. We can use a completely different Media-Type to describe the resource changes.

PATCH can be used in two common ways which both have their own pros and cons. We will look into both of them in the next sections.

Using the standard resource representation to send changes (JSON Merge Patch)

The most intuitive way to use PATCH is to keep the standard resource representation that is used in GET or PUT requests. However, with PATCH we only include the fields that should be changed.

Assume we have a simple *product* resource. The response of a simple GET request might look like this:

```
GET /products/123
```

```
{
    "name": "Cool Gadget",
    "description": "It looks very cool",
    "price": 4.50,
    "dimension": {
        "width": 1.3,
        "height": 2.52,
        "depth": 0.9
    }
    "tags": ["cool", "cheap", "gadget"]
}
```

Now we want to increase the *price*, remove the *cheap* tag and update the product *width*. To accomplish this, we can use the following PATCH request:

```
PATCH /products/123
{
    "price": 6.20,
    "dimension": {
        "width": 1.35
    }
    "tags": ["cool", "gadget"]
}
```

Fields not included in the request should stay unmodified. In order to remove an element from the *tags* array we have to include all remaining array elements.

This usage of PATCH is called *JSON Merge Patch* and is defined in RFC 7396 (https://tools.ietf.org/html/rfc7396). You can think of a PUT request that only uses a subset of fields. Patching this way makes PATCH requests usually idempotent.

JSON Merge Patch and null values

There is one caveat with JSON Merge Patch you should be aware of: The processing of *null* values.

Assume we want to remove the *description* of the previously used *product* resource. The PATCH request looks like this:

```
PATCH /products/123
{
    "description": null
}
```

To fulfill the client's intent the server has to differentiate between the following situations:

- The *description* field is not part of the JSON document. In this case, the description should stay unmodified.
- The *description* field is part of the JSON document and has the value *null*. Here, the server should remove the current description.

Be aware of this differentiation when using JSON libraries that map JSON documents to objects. In strongly typed programming languages like Java it is likely that both cases produce the same result when mapped to a strongly typed object (the *description* field might result in being *null* in both cases).

So, when supporting *null* values, you should make sure you can handle both situations.

Using a separate Patch format

As mentioned earlier it is fine to use a different media type for PATCH requests.

Again we want to increase the *price*, remove the *cheap* tag and update the product *width*. A different way to accomplish this, might look like this:

```
PATCH /products/123
{
    "$.price": {
        "action": "replace",
        "newValue": 6.20
    },
    "$.dimension.width": {
        "action": "replace",
        "newValue": 1.35
    },
    "$.tags[?(@ == 'cheap')]": {
        "action": "remove"
    }
}
```

Here we use JSONPath (https://goessner.net/articles/JsonPath/) expressions to select the values we want to change. For each selected value we then use a small JSON object to describe the desired action.

To replace simple values this format is quite verbose. However, it also has some advantages, especially when working with arrays. As shown in the example we can remove an array element without sending all remaining array elements. This can be useful when working with large arrays.

JSON Patch

A standardized media type to describe changes using JSON is JSON Patch (described in RFC 6902 (https://tools.ietf.org/html/rfc6902)). With JSON Patch our request looks this:

```
PATCH /products/123
Content-Type: application/json-patch+json

[
    {
        "op": "replace",
        "path": "/price",
        "value": 6.20
    },
    {
        "op": "replace",
        "path": "/dimension/width",
        "value": 1.35
    },
    {
        "op": "remove",
        "path": "/tags/1"
    }
]
```

This looks a bit similar to our previous solution. JSON Patch uses the *op* element to describe the desired action. The *path* element contains a JSON Pointer (https://tools.ietf.org/html/rfc6901) (yet another RFC) to select the element to which the change should be applied.

Note that the current version of JSON Patch does not support removing an array element by value. Instead, we have to remove the element using the array index. With */tags/1* we can select the second array element.

Before using JSON Patch, you should evaluate if it fulfills your needs and if you are fine with its limitations. In the issues of the GitHub repository json-patch2 (https://github.com/json-patch/json-patch2) you can find a discussion about a possible revision of JSON Patch.

If you are using XML instead of JSON you should have a look at XML Patch (RFC 5261 (https://tools.ietf.org/html/rfc5261)) which works similar, but uses XML.

The Accept-Patch header

The RFC for HTTP PATCH also defines a new response header for HTTP OPTIONS requests: *Accept-Patch*. With *Accept-Patch* the server can communicate which media types are supported by the PATCH operation for a given resource. The RFC says:

> Accept-Patch SHOULD appear in the OPTIONS response for any resource that supports the use of the PATCH method.

An example HTTP OPTIONS request/response for a resource that supports the PATCH method and uses JSON Patch might look like this:

Request:

```
OPTIONS /products/123
```

Response:

```
HTTP/1.1 200 OK
Allow: GET, PUT, POST, OPTIONS, HEAD, DELETE, PATCH
Accept-Patch: application/json-patch+json
```

Responses to HTTP PATCH operations

The PATCH RFC does not mandate how the response body of a PATCH operation should look. It is fine to return the updated resource. It is also fine to leave the response body empty.

The server responds to HTTP PATCH requests usually with one of the following HTTP status codes (https://www.mscharhag.com/api-design/http-status-codes):

- 204 (No Content) - Indicates that the operation has been completed successfully and no data is returned
- 200 (Ok) - The operation has been completed successfully and the response body contains more information (for example the updated resource).
- 400 (Bad request) - The request body is malformed and cannot be processed.
- 409 (Conflict) - The request is syntactically valid but cannot be applied to the resource. For example it can be used with JSON Patch if the element selected by a JSON pointer (the *path* field) does not exist.

Summary

The PATCH operation is quite flexible and can be used in different ways. *JSON Merge Patch* uses standard resource representations to perform partial updates. *JSON Patch* however uses a separate PATCH format to describe the desired changes. it also fine to come up with a custom PATCH format. Resources that support the PATCH operation should return the *Accept-Patch* header for OPTIONS requests.

(https://plus.google.com/share?url=https://www.mscharhag.com/api-design/rest-partial-updates-patch) (//www.reddit.com/submit?title=REST%3A+Partial+updates+with+PATCH&url=https%3A%2F%2Fwww.mscharhag.com%2Fapi-design%2Frest-partial-updates-patch)

Tags: REST (/tags/rest), HTTP (/tags/http)

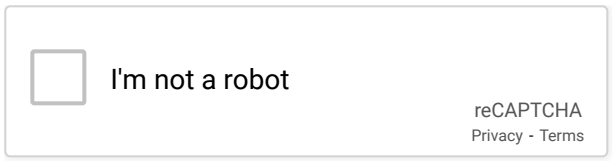
Related posts:

- REST: Updating resources (/api-design/updating-resources-put)
- REST: Retrieving resources (/api-design/rest-retrieving-resources)
- REST: Creating resources (/api-design/resource-creation-post)
- REST / HTTP methods: POST vs. PUT vs. PATCH (/api-design/http-post-put-patch)
- HTTP methods: Idempotency and Safety (/api-design/http-idempotent-safe)

Leave a reply

Your comment

Name / Pseudonym



Submit

Michael Scharhag

I am a Java Web Developer, Stackoverflower and 3D graphic hobbyist living in Mainz, Germany.

More about me (/about)
Follow me on Twitter (https://twitter.com/mscharhag)
Subscribe this blog (/feeds)

Most popular

- REST / HTTP methods: POST vs. PUT vs. PATCH (/api-design/http-post-put-patch)
- JSON Schema validation in Java (/java/json-schema-validation)
- A deeper look into the Java 8 Date and Time API (/java/java-8-date-time-api)
- Understanding JUnit's Runner architecture (/java/understanding-junits-runner-architecture)
- Composing custom annotations with Spring (/spring/annotation-composition)
- Extending JUnit 5 (Jupiter) (/java/junit5-custom-extensions)
- REST: Managing Many-To-Many relations (/api-design/rest-many-to-many-relations)
- OCR in Java with Tess4J (/java/ocr-tess4j)

Tags

- Android (/tags/android)
- Architecture (/tags/architecture)
- Code Quality (/tags/code-quality)
- Grails (/tags/grails)
- Groovy (/tags/groovy)
- Hibernate (/tags/hibernate)
- HTTP (/tags/http)
- IntelliJ IDEA (/tags/intellij-idea)
- Java (/tags/java)
- Java EE (/tags/java-ee)
- JPA (/tags/jpa)
- JUnit (/tags/junit)
- Kotlin (/tags/kotlin)
- Localization (/tags/localization)
- Microservices (/tags/microservices)
- MongoDB (/tags/mongodb)
- MySQL (/tags/mysql)

- REST (/tags/rest)
- Solr (/tags/solr)
- Spring (/tags/spring)
- Spring Data (/tags/spring-data)
- Spring Security (/tags/spring-security)
- Tools (/tags/tools)