PUT vs PATCH vs JSON-PATCH

 **Phil Sturgeon** on May 3 2016 #api #http #rest #patch #json-patch

A question that is asked with increasing regularity in the APIs You Won't Hate Slack Group is one which has been asked for years, but does not always have a good answer. The question is:

> *What is the different between PUT and PATCH, and when do I use them? And WTF is JSON-PATCH?*

To start off, `PUT` and `PATCH` are two different HTTP methods, which are both commonly used in REST APIs. For people who think of REST APIs as only being CRUD (Create, Read, Update, Delete) there can be confusion over trying to work out which one is "best". People have preferences, people argue, and really the conversation is rarely had in a reasonable way.

You can totally have both `PUT` and `PATCH` in your API, and no, they should not be an alias of each other (looking at you Rails). Quite simply, they do different things. The RFC for `PATCH` (RFC 5789) actually explains the difference rather elegantly in its abstract:

> *The existing HTTP PUT method only allows a complete replacement of a document. This proposal adds a new HTTP method, PATCH, to modify an existing HTTP resource.*

One is for when you know *all* the answers, and the other is for updating little bits at a time. Some consider this a performance benefit (sending less stuff is quicker than sending lots of stuff), but there are some more racy benefits than that.

Conflicts

Think about a resource that has two fields, `field1` and `field2`. Two different requests (Request A and (Request B) try to update one of these field values as a `PUT` after getting the initial value of the resource with a `GET` request. Both `field1` and `field2` are `false` in response of the `GET` request.

Request A

Updating `field1` to be `true`.

```
PUT /foos/123

{
  "field1": true,
  "field2": false
}
```

Request B

Updating `field2` to be `true`.

```
PUT /foos/123

{
  "field1": false,
  "field2": true
}
```

If both fields start `false`, and each request only intends to update one field, little do they know they are clobbering the results and essentially reverting them each time. Instead of ending up with both values being `true`, you'll simply have whatever the last request was, which is going to be `"field1": false` and `"field2": true`.

To some this is a feature, but others consider it a bug because if they only want to update one field, why do they need to send everything?

These people decide to just send the relevant fields they want to change, which is a flagrant misuse of how `PUT` is supposed to work and leads to a lot of problems.

Expectations

When building an API, you and your coworkers are not the only people that need to have fair expectations of how things are going to work. Other systems, such as EmberJS for example, are going to have some expectations of how a `PUT` request is going to work, and if you start going against the grain and making `PUT` send partial updates, you're going to have a bad time.

We had a case at work, where an EmberJS application only had some models representing existing resources in the API, that were populated locally from partial data. They wanted to change the value of one field in this model and save the resource back to the API.

When they saved, EmberData would notice it was a `PUT` and try to send as much data as it could. As it only had some of the field values, it would end up sending a body with every unknown field as `null`, which in turn was emptying values out of the database, and/or triggering validation errors for fields that it didn't want emptied.

PATCHing the Problem

Something that helped a lot at work was implementing `PATCH`. We can

Something that helped a lot at work, was implementing `PATCH`. We can now simply send the fields we intend to update, and anything else is left alone.

Let's just assume we're using JSON-API and building something for a carpooling company, like Ride. We want to be able to "start" a trip, by changing the status from `"pending"` to `"in_progress"`.

In v1.0, we used to use `PUT`. It worked something like this:

```
PUT /trips/123

{
  "data": [{
    "type": "trips",
    "id": "123",
    "attributes": {
      "status": "in_progress",
      "started_at": null,
      "finished_at": null
    }
    "relationships": {
      "driver": {
        "data": { "type": "users", "id": "999" }
      }
    }
  }
}
```

Here we've changed the `status` from whatever it was to `in_progress`.

Sidenote: *Am I meant to update the started_at myself or let the API do it? Who knows!*

The main problem here is back to the conflict example above. If somebody else was changing another value like who the driver is, and accidentally changed the `status`, back to `pending` then they'd get an error message saying "The trip has already started, it cannot go back to pending" and they'd be thinking "I didn't even know it had started, I just wanted Gary to drive today." and everyone just gets confused.

The same request as a `PATCH` could look something like this:

```
PUT /trips/123

{
  "data": [{
    "type": "trips",
    "id": "123",
    "attributes": {
      "status": "in_progress"
    }
  }
}
```

This has solved the problem of not accidentally clobbering other values, as we are no longer sending things along for the sake of it. Only the attributes we send should be validated, and anything missing should be ignored entirely.

WellActually

In RFC 5789 (the RFC for the `PATCH` method), the example shows things working like so:

```
PATCH /file.txt HTTP/1.1
Host: www.example.com
Content-Type: application/example
If-Match: "e0023aa4e"
Content-Length: 100

[description of changes]
```

This `[description of changes]` is considered by some to be a sequence of operations to be done to the resource in question, manifesting itself in a list of JSON objects like this:

```
PATCH /my/data HTTP/1.1
Host: example.org
Content-Length: 326
Content-Type: application/json-patch+json
If-Match: "abc123"

[
  { "op": "test", "path": "/a/b/c", "value": "foo" },
  { "op": "remove", "path": "/a/b/c" },
  { "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
  { "op": "replace", "path": "/a/b/c", "value": 42 },
  { "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
  { "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }
]
```

This example is taken from RFC 6902, which builds on top of the `PATCH` method itself, to provide a standardised way of doing it. It's known as JSON PATCH, and finding information about it is almost as sparse as the website.

There are a few articles around, but unfortunately the #WellActually starts to get pretty heavy.

One of the most prolific articles out there on using PATCH is from William Durand. It's a great technical article, but the assertions of "do this really complicated thing which you might not need or you're Doing It Wrong" do rub me up the wrong way a bit. Read this article, consider implementing it, and if you don't want it, don't do it. You're fine.

Yes, JSON PATCH is lovely, and you might need it for your API, but it also might be a complication you don't need to worry about at this point.

While not the end all or be all of anything, JSON-API used to recommend using JSON PATCH, but ended up settling on the plain "just send what you need" approach to `PATCH` which a lot of other people settle for. Something very similar to this "just send what you need" approach is being worked on as *yet another* RFC: RFC 7396.

Remember, one of the best things about HTTP-based APIs is the ability to respond to `Content-Type` headers. You can work with plain-old JSON in your `PATCH` requests for now, and in the future add JSON

PATCH support if you find that you need it.

And you can keep `PUT` in there too, if you like the idea of having idempotent saves. We don't use them for resources at Ride anymore, but they're pretty great for file uploads which could fail and need idempotent retries.

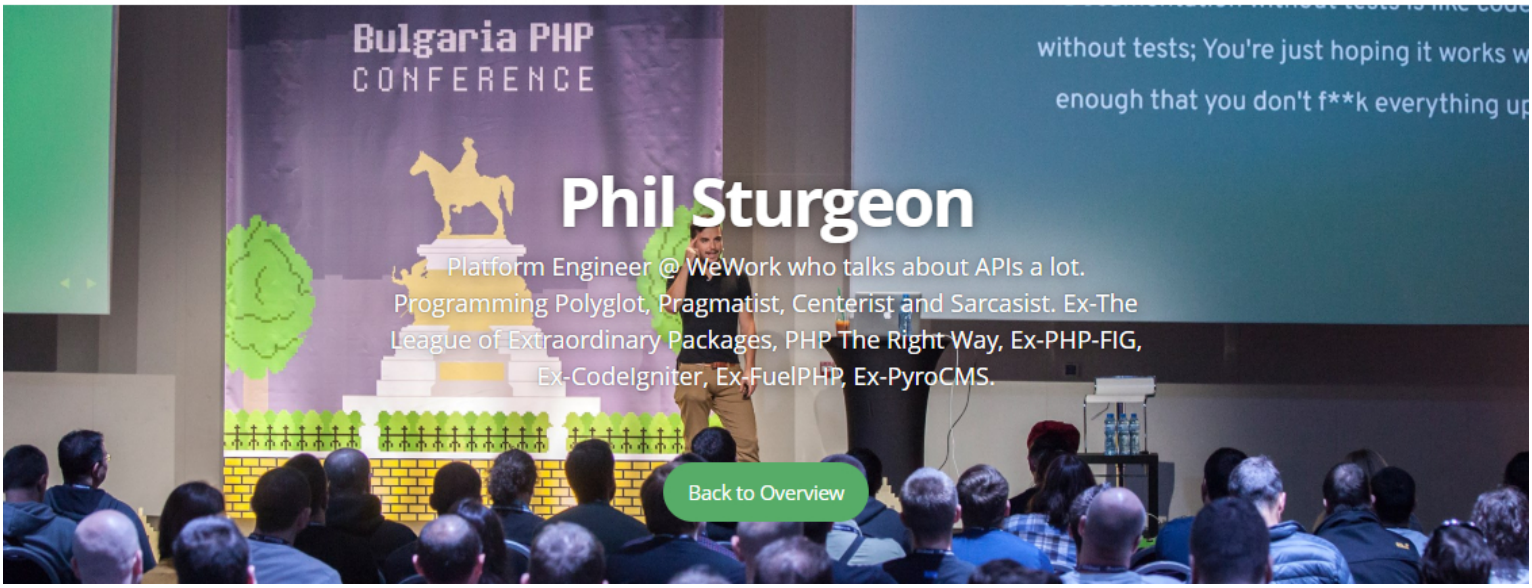
Previous: Loudly Ignoring How CoCs Work

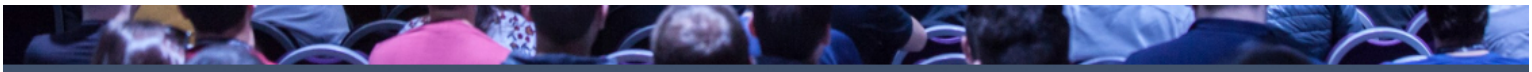
Next: Why Care About PHP Middleware?

WRITTEN BY

Phil Sturgeon

Platform Engineer @ WeWork who talks about APIs a lot. Programming Polyglot, Pragmatist, Centerist and Sarcasist. Ex-The League of Extraordinary Packages, PHP The Right Way, Ex-PHP-FIG, Ex-CodeIgniter, Ex-FuelPHP, Ex-PyroCMS.





Content

About
Books
Speaking

Follow Me

Twitter
GitHub
API Busters
Podcast
Phil.Bike

Recent Posts

Picking the right API Paradigm
API Evolution for REST/HTTP APIs
Solving OpenAPI and JSON Schema Divergence