**Serverless API Essentials: PUT vs PATCH**

It can be tricky to know when to update with a PUT or a PATCH. Choosing which serverless resources to use is your next hurdle

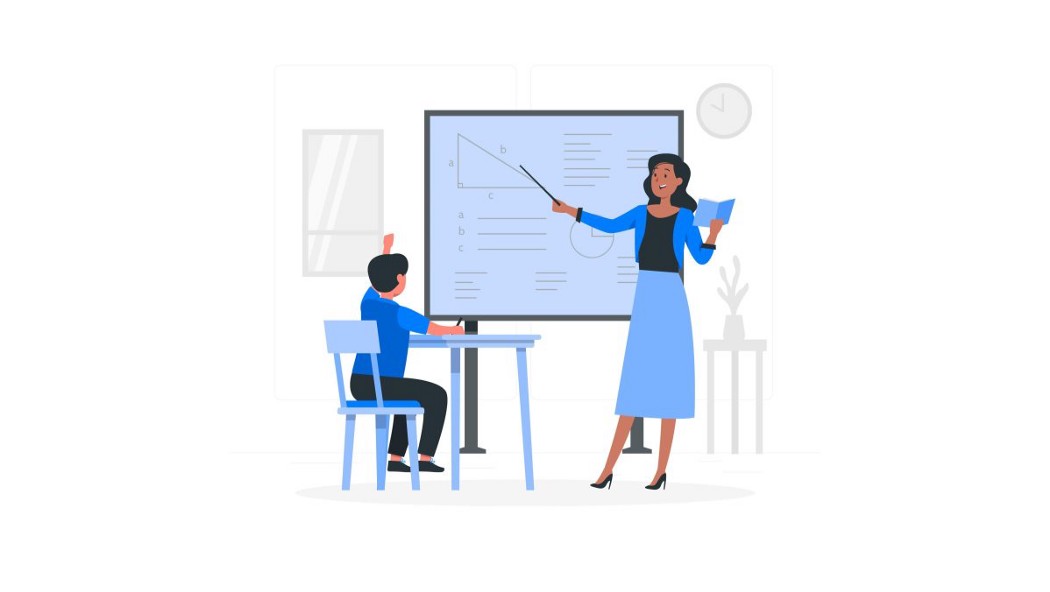


Image by [storyset](https://www.freepik.com/free-vector/teacher-concept-illustration_8449776.htm#query=teacher&position=39&from_view=search) on Freepik

I conduct a decent amount of developer interviews. One of my favorite topics in an interview is REST APIs. I’m passionate when it comes to API design and [API development](https://betterprogramming.pub/are-you-really-api-first-or-do-you-just-think-it-24ffc47ec7ea) in general. So naturally, it’s something I ask about regularly.

We cover the basics, like how endpoints are structured and how to balance fundamental REST construction with developer experience. But something that trips people up more often than you’d expect is when I ask to explain the difference in the [HTTP methods](https://doc.oroinc.com/api/http-methods) and when you would use one over another.

Most people articulate beautifully when to use POST, GET, PUT, and DELETE. But when I ask about PATCH, I often get a “deer in the headlights” look.

Sometimes people tell me they have never heard of it before; other times, they tell me they know it’s for an update operation, but not sure what the difference is between PATCH and PUT.

The simplest difference between the two operations is:

A PUT replaces the entire resource with the request payload, but a PATCH applies partial updates to the resource.

Depending on your decision to use PUT or PATCH, your downstream implementation can change drastically when it comes to serverless. So, let’s talk about when you would use one or the other and discuss the implementation with serverless resources.

**When To Use PUT Operations**

As mentioned earlier, a PUT will replace the entire resource with the request payload. This means that everything is overwritten as is when you make the call, thus making this operation generally [idempotent](https://betterprogramming.pub/serverless-api-essentials-idempotency-e753b7b49680).

Any operation that you need to be idempotent and require the full resource data to be provided should be backed by a PUT operation. For example, if we take the [update gopher hole](https://github.com/allenheltondev/gopher-holes-unlimited/blob/master/openapi.yaml#L670) endpoint from my reference project [Gopher Holes Unlimited](https://github.com/allenheltondev/gopher-holes-unlimited), you can see that we overwrite the entire gopher hole resource.

This resource has limited fields and none of them are restricted via access control (more on this later). As a result, we have a perfect use case for a PUT. It’s not an excessive burden to require the caller to provide all the details of the gopher hole resource, even the values that aren’t changing.

When it comes to implementing a PUT in a serverless system, you can do it a couple of different ways. If you do not need to perform validations (outside of schema validation) and simply need to replace the entity, you can [connect directly to DynamoDB via API Gateway](https://betterprogramming.pub/skip-the-lambda-function-connect-directly-to-your-aws-services-5f10798f2d5a) to replace the resource.

An example implementation in an [Open API Spec](https://www.openapis.org/) can be seen below:

This simple operation does not require any Lambda function. API Gateway will directly call DynamoDB and replace the data of the requested resource, assuming the resource already exists.

A PUT operation is also allowed to [create a resource if it does not exist](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT#responses). It sets the id of the resource to the id provided in the path. But since we’re talking about update operations, this endpoint will return an error if the resource does not exist.

**When To Use PATCH Operations**

If a resource has elements to it that are under a level of access control or it is too large to feasibly ask the consumer to pass in all the fields, you’re looking at a prime opportunity for a PATCH.

When I refer to access control, I mean elements of a resource that require a certain level of authorization to view or edit. These are fields that only the creator of a resource can manipulate, for example. In Gopher Holes Unlimited, there is a [PATCH /gophers/{gopherId}](https://github.com/allenheltondev/gopher-holes-unlimited/blob/master/openapi.yaml#L231) endpoint that allows the caller to update only a subset of the gopher information via the request schema.

In the downstream implementation, validation occurs to make sure only the person who added the gopher can change the name. Other consumers of the API can change other fields like the number of times they have seen the gopher or the color, but not the critical identifying information.

Because of the additional validation of the content, the PATCH is the preferred route in this scenario. To implement a generic PATCH mechanism in a Node.js Lambda function, refer to the code [in this gist](https://gist.github.com/allenheltondev/ccf585a1470e6be200f3ef970187e636).

Contrary to a PUT operation, when talking about serverless implementation a PATCH will perform an UpdateItem call in Dynamo. The [UpdateItem](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_UpdateItem.html) call will add, update, or remove specific attributes from an item, rather than doing a complete replace like the [PutItem](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_PutItem.html) call we used in the PUT endpoint.

Point of note — when using a generic PATCH mechanism like in the gist above, you must rely on schema validation at the API gateway level. You should define the incoming schema to [scope potential changes down](https://gist.github.com/allenheltondev/ccf585a1470e6be200f3ef970187e636#file-openapi-yaml-L49) to only the mutable fields. Be sure to not allow any additional properties when using generics because you open yourself up to storing potentially unlimited data passed in by malicious users.

There are two schools of thought regarding implementing a PATCH endpoint:

* Provide all the updated properties in a request payload like you would a POST or a PUT

{   
 "location": {   
 "longitude": 24.554   
 },   
 "comment": "This is a nasty gopher"   
}

* Provide the operation, path to property, and value for each changing value as an array

[  
 {  
 "op": "add",  
 "path": "/comment",  
 "value": "This is a nasty gopher"  
 },  
 {  
 "op": "replace",  
 "path": "/location/longitude",  
 "value": 24.554  
 },  
 {  
 "op": "remove",  
 "path": "color"  
 }  
]

As with most things software, these implementations come with tradeoffs. The first implementation is more familiar and generally has a lower cost when it comes to consuming your API. However, it does not provide you with the ability to remove properties.

The first implementation is only an add or update operation. A workaround would be to remove a property if it was passed into the request as a blank value, but that might not be correct 100% of the time.

The other implementation requires passing in the op, path, and value of each field to be changed. This results in a significantly more flexible design but adds complexity to the consumer to track the operations that occurred when data entry is being performed.

Either implementation is perfectly valid in their own scenarios. But be sure to make an intentional decision when [performing your initial API design](https://betterprogramming.pub/going-api-first-your-first-30-days-4c1ef6da320d).

**Summary**

Resource updates are an adventure. You have plenty of options to consider when deciding on which implementation you want to use.

The great thing about API design and serverless is that you don’t have to stick with one decision for your entire application. Maybe a subset of your endpoints makes sense to run as PUT operations while others are better suited for a PATCH.

Of your PATCH operations, maybe some endpoints should only allow updates and never remove a field, while others might require the ability to remove a field. It all depends on your use case.

Your serverless infrastructure decisions vary as well. If your update is a simple PUT with no side effects, you can go straight to DynamoDB from API Gateway. If you have validations to run prior to making a change, a Lambda function is better suited for your needs. If you have multiple validations and side effects that trigger as a result of a change, a [Step Function workflow](https://betterprogramming.pub/how-to-build-lightning-fast-apis-with-aws-step-functions-d1725624aaaa) might be your best bet.

Keep in mind that a PUT or a PATCH is not a silver bullet. It might be smarter to break down large endpoints into smaller ones.

Instead of updating a gopher status via a PUT /gophers/{gopherId} endpoint, it might be a better option to implement a POST /gophers/{gopherId}/statuses to separate side effects or business logic that need to run as a result of updating a particular field.

That’s the beauty of API design. It’s a balance between fundamentalism and the perfect developer experience. Use what makes the most sense to make your consumers happy.