HTTP Method Overview by Use Case

* Let's start with reading resources. If we want to read the collection resource like authors, we'd send a GET request to authors. That then results in a 200 OK with an authors collection in the response body. A 404 Not Found is also possible if the URI doesn't exist. In ASP.NET Core, we didn't have to code anything for that, the framework handled this for us. If we want to read one specific resource, we add the ID and we should get back a 200 OK with the author in the response body. If the resource doesn't exist, a 404 is warranted.

A picture containing text

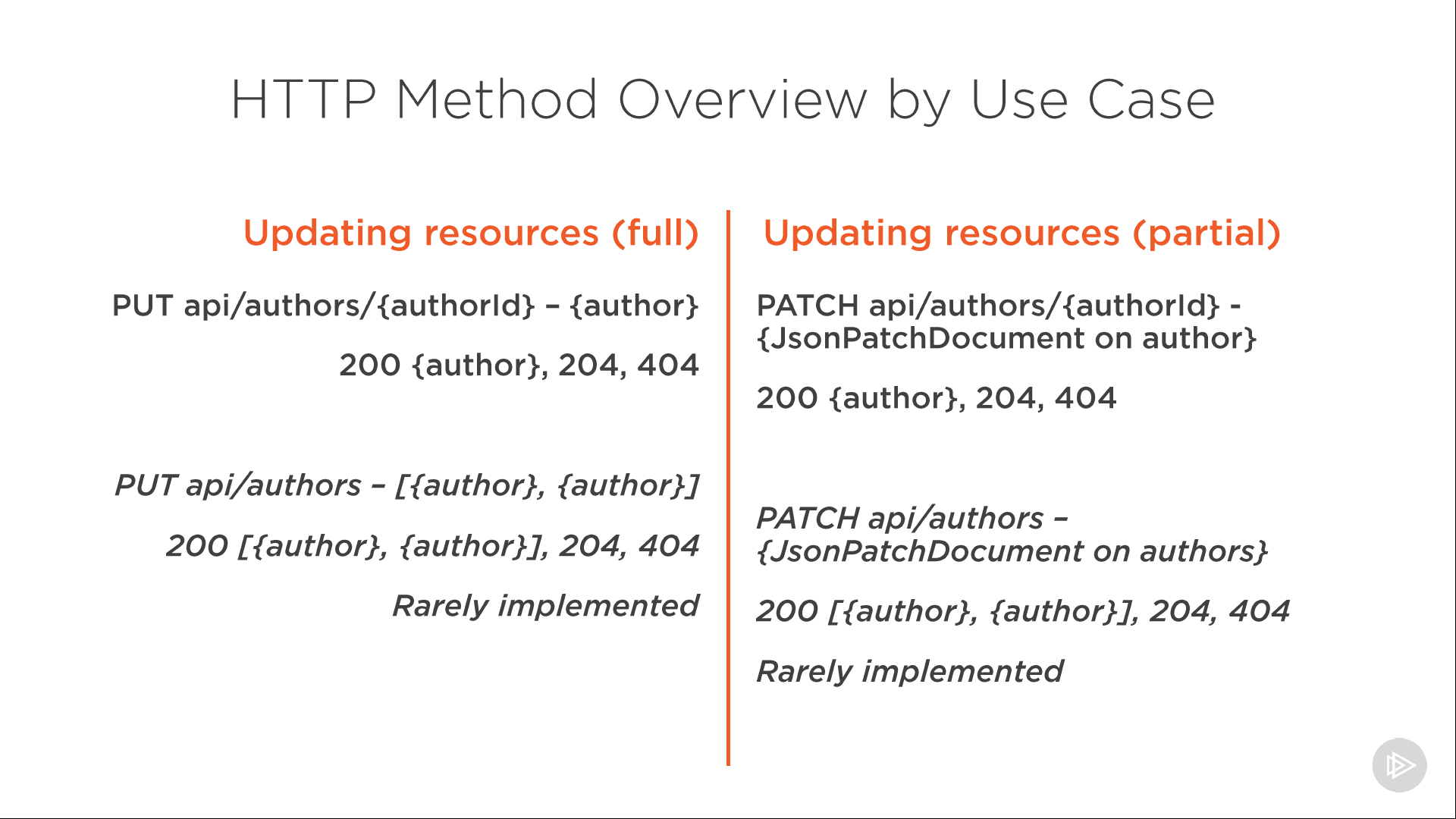
Description automatically generated

* Deleting a single resource is done by sending a Delete request to that specific resource URI. If the Delete is successful, we send back a 204 No Content. If the resource doesn't exist, we should return a 404 Not Found. Deleting a resource collection follows the same principles, but it's rarely implemented because it can be very destructive.
* For creating resources, we have to make a separation between those requests for the responsibility of creating the URI, is at the server versus when it's at the consumer of the API. So let's start with the most obvious case, at the server. To create a new author, we should send the POST request to the author's resource with the author representation as the request body. If creation is successful, we send back a 201 created response with the author representation in the response body and location in the location error. If there is no author's resource, a 404 is warranted. Posting to a single resource URI can never result in a successful request. It either returns a 404 Not Found if the author doesn't exist, or a 409 conflict if the author already exists. And to add a collection in one go, we should create a new resource for that collection, for example, author collections. A post to that resource then contains an author collection, in other words, an array of authors and can result in a 201 Created or 404 Not Found.

A picture containing table

Description automatically generated

* When the consumer of the API can create resource URIs, for example, when we're working with GUIDs, we can upsert to create resources. A PUT request to a previously un-existing URI for an author then results in that author being created. It warrants a 201 Created response with the author in the body. There is no 404 for this URI because if it's not found, it will be created. And the same is possible with PATCH, with that difference that we're passing in a JSON PATCH document for that author.
* Lastly, updating resources. We're starting with full updates with PUT. A PUT request to a specific author resource updates that author. We should pass in the full author representation. Fields that are omitted are set to their default values. That can result in a 200 OK with the author representation in the body, or a 204 No Content. If the author isn't found, a 404 Not Found is warranted. Of course if the consumer is allowed to choose the URI, you could upsert instead of returning a 404. PUT requests to collection resources are allowed, but rarely implemented.



* A partial update is what PATCH is for. To update a specific author, we'd send the PATCH request to that author's URI. A PATCH request body can be defined by the JSON PATCH standard, a list of operations on that specific author. If the PATCH request is successful, a 200 OK with the updated author representation in the body, or a 204 No Content is warranted. A 404 Not Found is returned if the author isn't found, unless, as said with PUT, we're upserting. PATCH requests to collection resources are, just as PUT requests, allowed, but rarely implemented. And that should cover all our use cases. When in doubt, have a look at that method safety and method idempotency table from a previous module. Now there are also other status codes that can be returned, 500 is always possible, and once additional functionality like validation is added, validation-specific codes can be returned. But this overview is the gist of it, and it should go a long way in mapping required functionality to an outer-facing contract. Let's have a look at what's next.