PHASE 6

PREPARED BY BRIJESH KAMANI

Brijesh Kamani
FULL STACK DEVELOPER TRAINEE | RKIT

Contents

27. Building Web API

27.1. Understanding HTTP Verbs

- *The HTTP verbs comprise a major portion of our "uniform interface" constraint and provide us with the action counterpart to the noun-based resource. The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations, respectively. There are several other verbs, too, but are utilized less frequently. Of those less-frequent methods, OPTIONS and HEAD are used more often than others.*
- *Below is a table summarizing recommended return values of the primary HTTP methods in combination with the resource URIs:*

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), the single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

27.2. Implement GET, POST, PUT, DELETE

- *Below is a more detailed discussion of the main HTTP methods. Click on a tab for more information about the desired HTTP method.*

POST

- *The POST verb is most often utilized to **create** new resources. In particular, it's used to create subordinate resources. That is subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.*
- *On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.*
- *POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most likely result in two resources containing the same information.*

Examples:

- *POST http://www.example.com/customers*
- *POST http://www.example.com/customers/12345/orders*

GET

- *The HTTP GET method is used to **read** (or retrieve) a representation of a resource. In the "happy" (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).*
- *According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.*
- *Do not expose unsafe operations via getting—it should never modify any resources on the server.*

Examples:

- *GET http://www.example.com/customers/12345*
- *GET http://www.example.com/customers/12345/orders*
- *GET http://www.example.com/buckets/sample*

PUT

- *PUT is most often utilized for **update** capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.*
- *However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.*
- *Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).*
- *On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for creating, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.*
- *PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.*
- *If, for instance, calling PUT on resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.*

Examples:

- *PUT http://www.example.com/customers/12345*
- *PUT http://www.example.com/customers/12345/orders/98765*
- *PUT http://www.example.com/buckets/secret_stuff*

PATCH

- *PATCH is used for **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.*
- *This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch languages like JSON Patch or XML Patch.*
- *PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag is an If-Match header on the PATCH request.*

  Examples:

  - *PATCH http://www.example.com/customers/12345*
  - *PATCH http://www.example.com/customers/12345/orders/98765*
  - *PATCH http://www.example.com/buckets/secret_stuff*

DELETE

- *It is pretty easy to understand. It is used to **delete** a resource identified by a URI.*
- *On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with nobody, or the JSEND-style response and HTTP status 200 are the recommended responses.*
- *HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be*

updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

- *There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.*

Examples:

- *DELETE http://www.example.com/customers/12345*
- *DELETE http://www.example.com/customers/12345/orders*
- *DELETE http://www.example.com/bucket/sample*

27.3. Understanding JSON Structure

- *JSON is a text-based data format following JavaScript object syntax, which was popularized by Douglas Crockford. Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to read (parse) and generate JSON.*

JSON structure

- *As described above, JSON is a string whose format very much resembles JavaScript object literal format. You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals. This allows you to construct a data hierarchy, like so:*

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
```

```
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

- *If we loaded this string into a JavaScript program, parsed it into a variable called `superHeroes`, for example, we could then access the data inside it using the same dot/bracket notation we looked at in the JavaScript object basics article. For example:*

```
superHeroes.homeTown
superHeroes['active']
```

- *To access data further down the hierarchy, you have to chain the required property names and array indexes together. For example, to access the third superpower of the second hero listed in the member's list, you'd do this:*

```
superHeroes['members'][1]['powers'][2]
```

1. First, we have the variable name — `superHeroes`.
2. Inside that, we want to access the `member's` property, so we use `["members"]`.
3. `members` contain an array populated by objects. We want to access the second object inside the array, so we use `[1]`.
4. Inside this object, we want to access the `powers` property, so we use `["powers"]`.
5. Inside the `powers,` the property is an array containing the selected hero's superpowers. We want the third one, so we use `[2]`.

Arrays as JSON

- *Above we mentioned that JSON text looks like a JavaScript object inside a string. We can also convert arrays to/from JSON. Below is also valid JSON, for example:*

```
[
  {
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": [
      "Radiation resistance",
      "Turning tiny",
      "Radiation blast"
    ]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
```

```
  }
]
```

- *The above is perfectly valid JSON. You'd just have to access array items (in its parsed version) by starting with an array index, for example `[0]["powers"][0]`.*