# Interacting with Redfish/Swordfish using REST:

The following screenshots and explanations are done using the Postman REST client, and talking to the OFMF-GenZ-POC emulated environment.

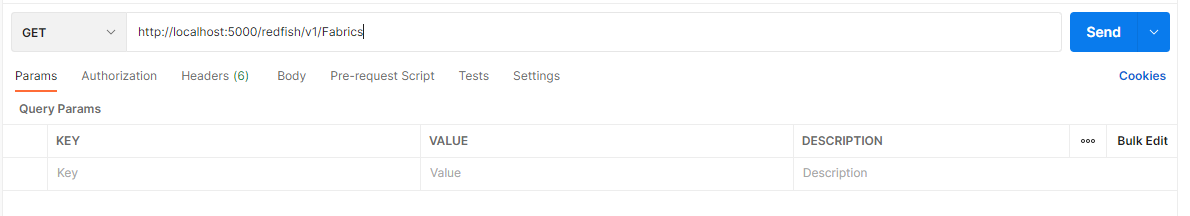
The emulator supports these primary REST commands for objects: GET, POST, PATCH, PUT, and DELETE

It supports the following commands for collections: GET, POST, PUT, and DELETE

## Using GET to walk the system:

To read the existing data, use GET. No additional information is required in the Body of a GET command. Use the URI format: http(s)://systemname:port/redfish/v1/ to get to the ServiceRoot.

Add to this base URI as you navigate through the system. This example shows a request to see what is in the Fabrics collection (Collections are groups of objects).



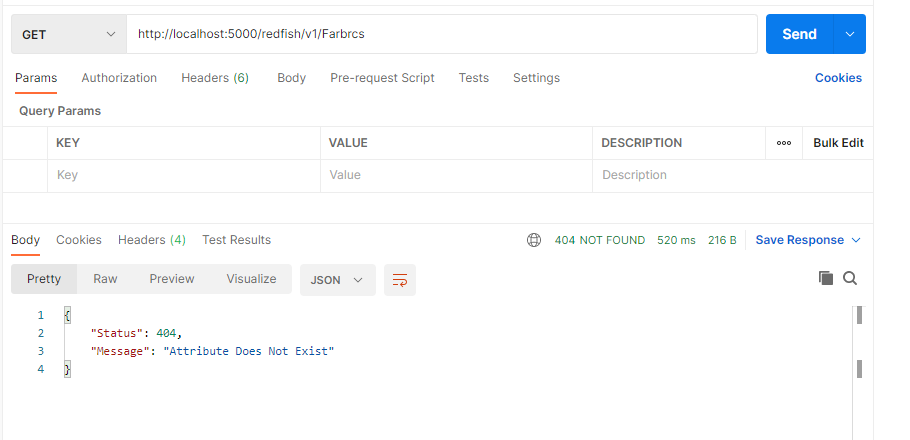
This return from this command shows the Fabrics collection (if it exists).



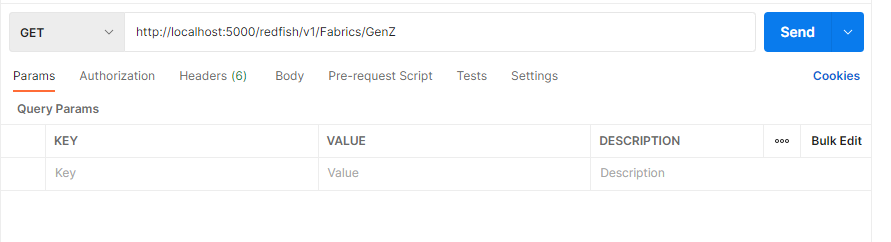
Note the return value in the upper right – this returned a 200 OK status, with the body containing the information about the requested item.

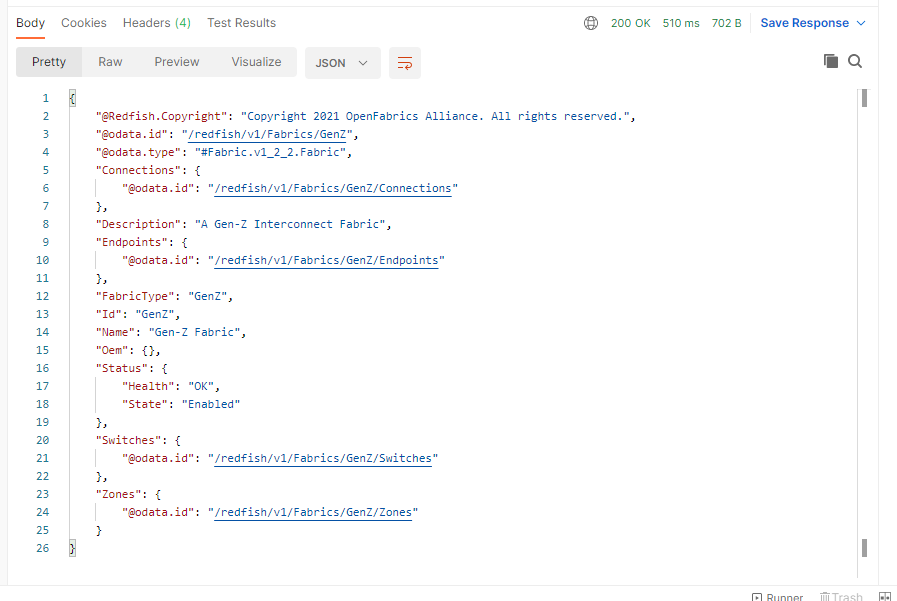
If the URI does not exist, but is generally valid, the system should return a 404 Not Found; you may also see a 500 Internal Server Error.

Below – the response when “fabrics” is misspelled:



You can “navigate” around the system (in postman, click on the highlighted links in the body text; it will send it to the URI box, and you can just click send):

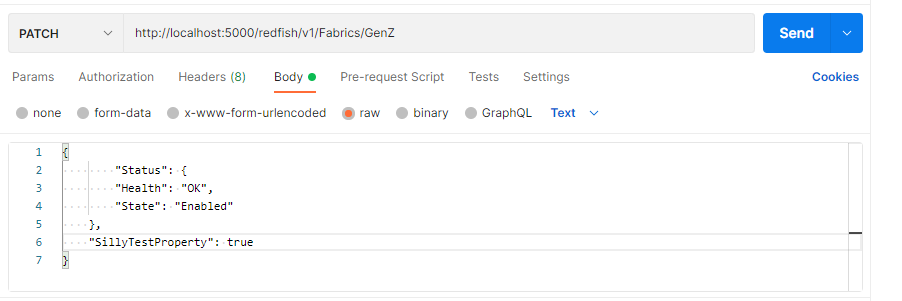




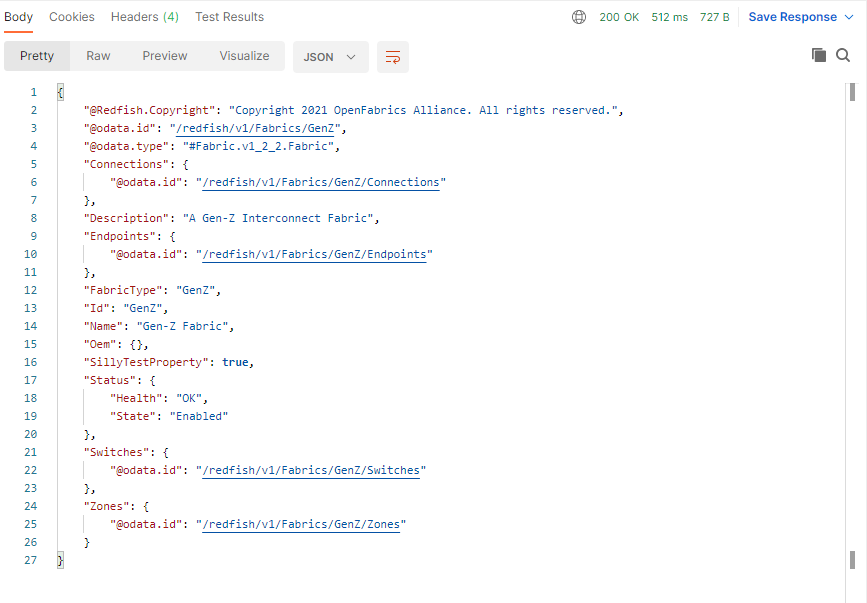
## Using PATCH

Use PATCH to change properties or add new ones. (Note – the emulator will not tell you if it is a valid property or value. It does not check against schema for correctness.)

When using PATCH, you will put the properties you want to modify in the “body” of the request. (In Postman, select “raw” to support json formatted text in the body.)



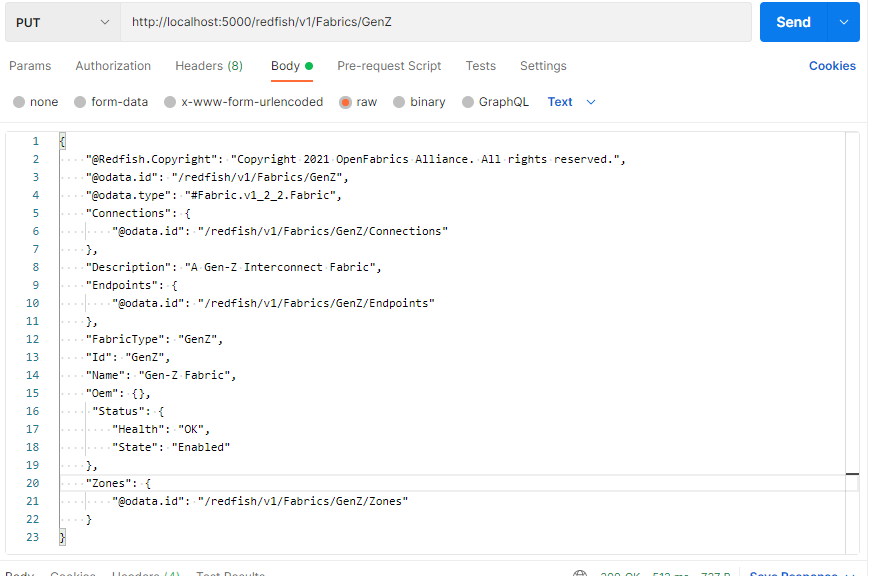
The results of the command should show the modified object body data:



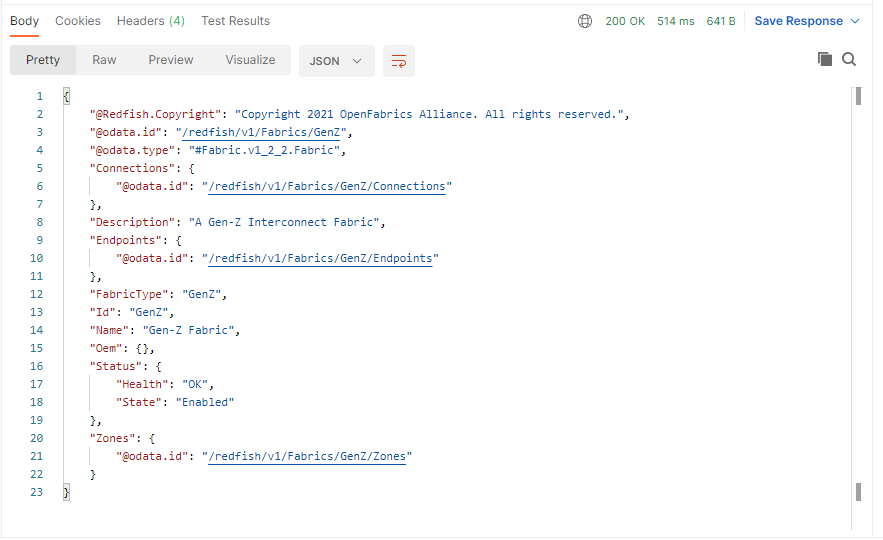
Using PUT:

If you want to delete a property from the object, you will need to use PUT.

For the body of the PUT command, you will need to enter \*everything\* you want to remain in the object (in this example, I’m removing “SillyTestProperty”, as well as the “Switches” collection. I have removed them from the text in the request body.

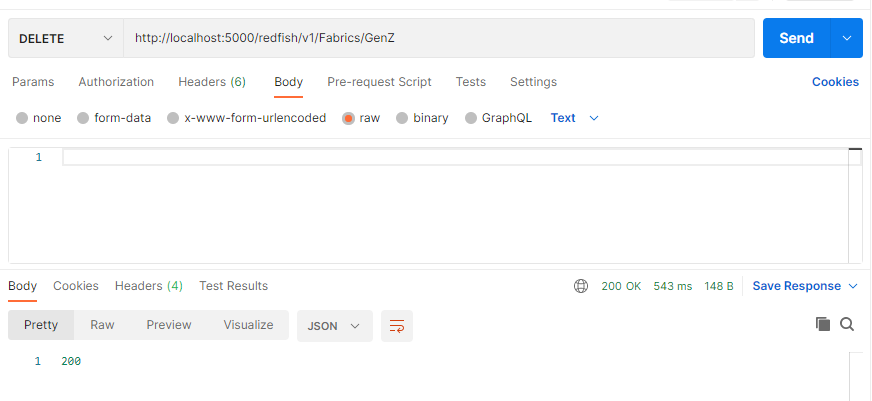


You can see from the results below that the properties have been removed.



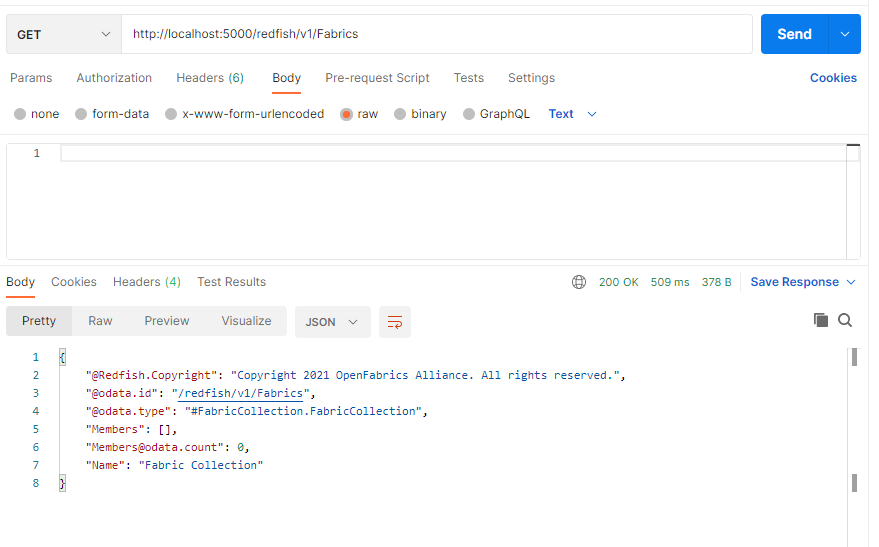
## DELETE the object:

If you want to completely remove the object, use the DELETE command. This command also does not require any body text; the URI indicates what you want to DELETE. (The emulator will delete anything subordinate to the object when it processes the DELETE request.)



As you can see, a successful DELETE returns a 200, with no body text. To verify the object was deleted, you can use GET. If you try the same URI, you will get a 404 Not Found.

The object reference was also deleted in the collection it belongs to; you can see by querying the Fabrics collection that there are no Fabrics in the system now.



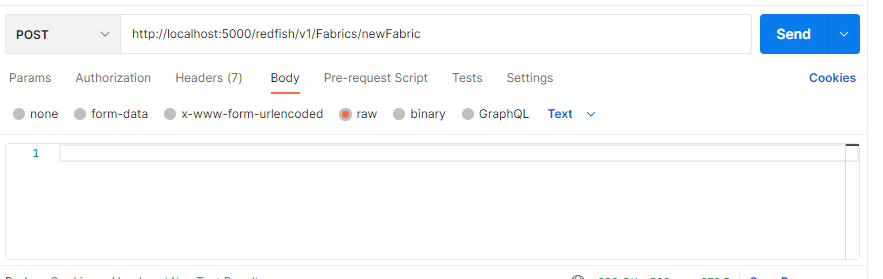
## Creating a new object using POST:

## 

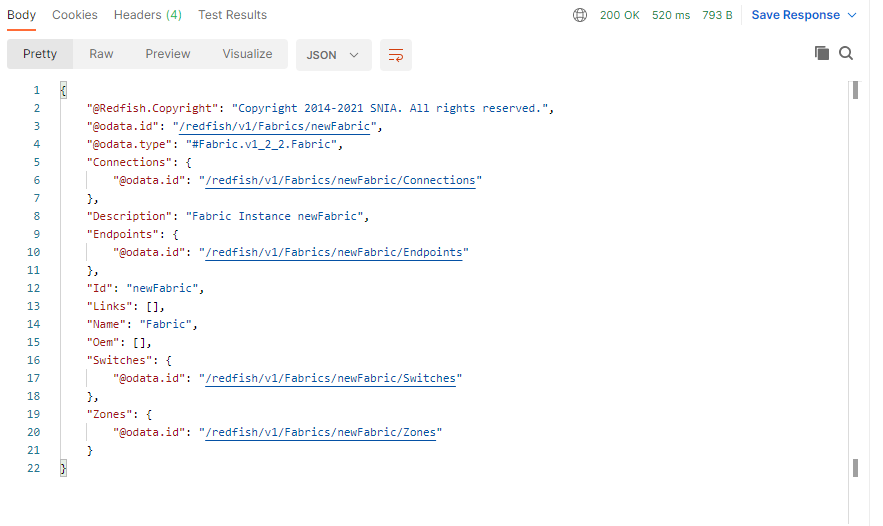
The POST commands can be used two different ways:

1. With no request body, a default object (based on a template in the emulator) is created.
2. With properties in the request body, these will be applied to the object when it is created.

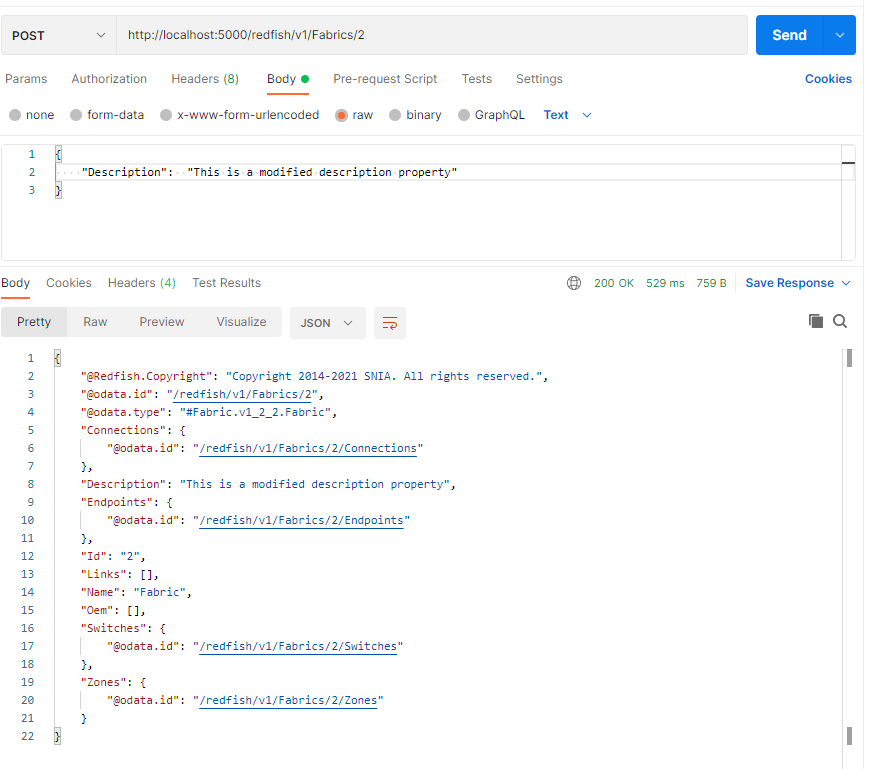
POST with no additional properties specified, other than the object ID:



Here is the new object based on the template. The odata.id, Id, and URI references have all been set to the designated id.



To add properties when creating the object:



## Working with Collections

The GET, POST, PUT and DELETE commands also work very similarly when applied to collections instead of objects. Depending on how the system is constructed, PUT is likely to only be used infrequently, if ever.