Hewlett Packard Enterprise

TSS Paris 2020

HPE Composable Infrastructure Hands-On Lab

Programming HPE OneView and HPE Global Dashboard

Contents

Programming HPE OneView and HPE Global Dashboard Hands-On Lab

Summary

In December 2015, in London, HPE announced Synergy, the first platform architected from the ground up for composability. Earlier that year, HPE had announced the Composable API, and the Composable ecosystem.
This ecosystem is now growing because more and more software partners have expressed interest in integrating with the Composable API. Some have already demonstrated products, some have products in development, but all of them are using the same Composable API, also referred to as the HPE OneView API.
This API allows HPE OneView and the Synergy Composer (powered by HPE OneView) to be controlled by another software entity, thus it is an essential step in building a Software-Defined infrastructure. The controlling software entity might be as simple as a small script (PowerShell, Python, etc.) it might also be a configuration management system, such as Chef, Ansible or any kind of software that needs to interact with the underlying infrastructure (VMware vCenter, Microsoft System Center, Docker Machine, etc.)
In all cases an API is a prerequisite for a Software-Defined infrastructure and the growing ecosystem around it. HPE OneView and the HPE Synergy Composer is no exception and it comes with an API since v1.0 in September 2013.

REST API

REST (Representational State Transfer) is a style of API that uses basic Create, Read, Update and Delete (CRUD) operations that are performed on resources using HTTP POST, GET, PUT and DELETE requests. To learn more about general REST concepts, see:

http://en.wikipedia.org/wiki/Representational_state_transfer

HPE OneView has a resource-oriented architecture that provides a uniform REST interface. Every resource has one Uniform Resource Identifier (URI) and represents a physical device or logical construct, and may be manipulated using REST APIs.

Resource operations

Basic Create, Read, Update and Delete (CRUD) operations are performed on the appliance resources via the standard HTTP POST, GET, PUT and DELETE methods. RESTful interfaces are based on the world wide web standards, thus most modern web servers can support these operations without modification. Restful APIs are stateless. The resource state is maintained by the resource manager and is reported as the resource representation. Any application state must be maintained by the client and it may manipulate the resource locally, but until a PUT or POST is made, the resource as known by the resource manager is not changed.

Table 1. REST HTTP Operations

Operation	HTTP Verb	Description
Create	POST URI <Payload = Resource data>	New resources are created using the POST operation and including relevant data in the payload. On Success the Resource URI is returned.
Read	GET URI	Returns the requested resource representation(s)
Update	PUT URI <Payload = Update data>	Update an existing resource using the updated data.
Delete	DELETE URI	Delete the addressed resource

URI format

All the appliance URIs point to resources and the client does not need to modify or create URIs. The URI for a specific resource is static and follows this format: `https://{appl}/rest/{resource name}`. The three parts are described below.

Table 2. URI Format

https://{appl} .	The appliance address
/rest	Type of URI.
/{resource name]	Name of the appliance resource such as server-profiles.

REST API calls can be executed using many different scripting languages such as PowerShell, Python, Java, Ruby, Golang and many other scripting and/or programming languages. Other tools like Postman or HTTPRequester for Firefox can be also very useful as they can place REST calls without writing any code, useful when you need to verify a REST call or quickly explore the JSON information you receive in return.

Data transfer format

The appliance resources support JSON (JavaScript Object Notation) as the standard for exchanging data using a REST API. If JSON is not specified in the REST API call, then the default is JSON.

To learn more about JSON, go to www.json.org

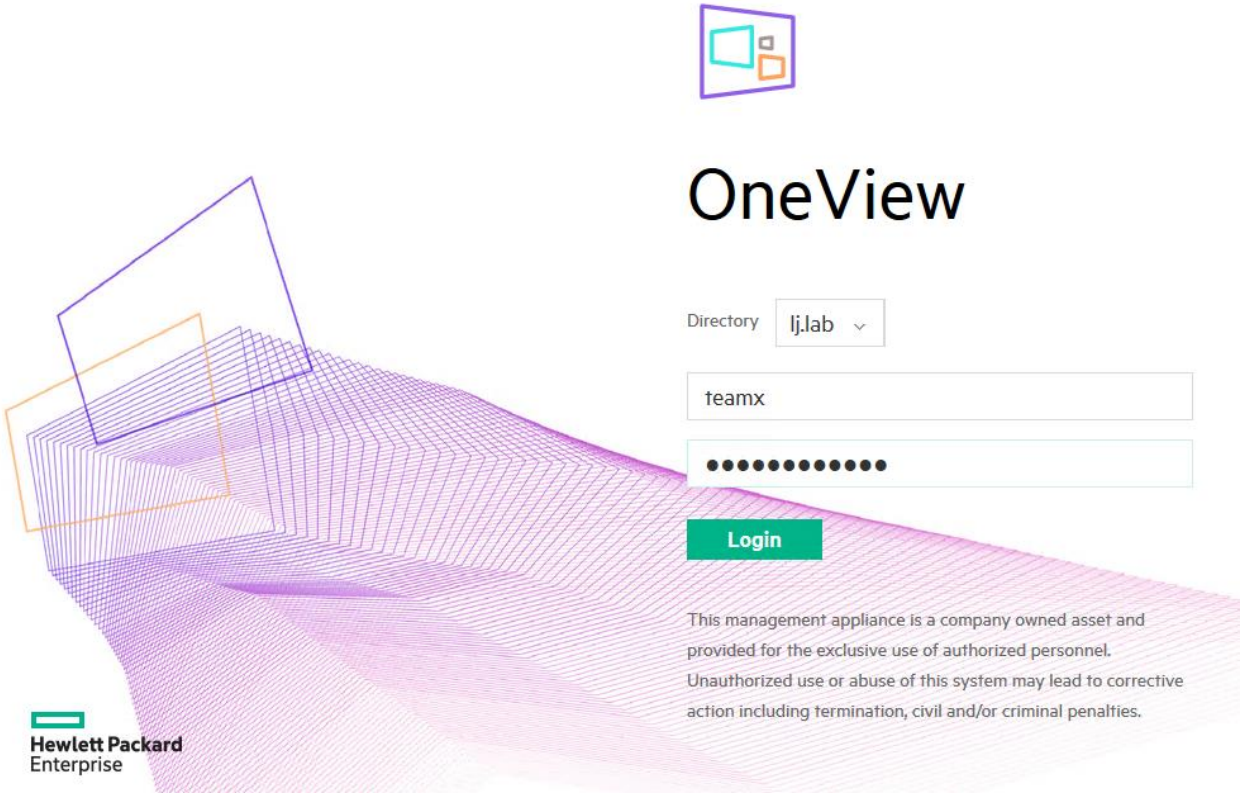
Lab 1: Read the Fantastic Manual

To make our first call to the HPE Composable API, we can refer to the fantastic manual provided with HPE OneView. This is where everyone should start!

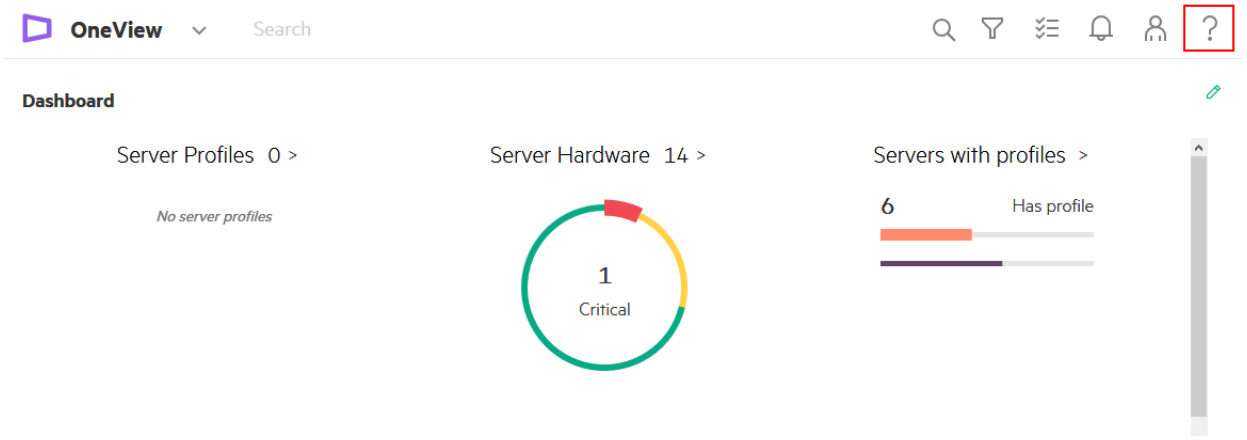
Login to the HPE Composer running OneView (https://composer) using the following credentials:

- Directory: **lj.lab**
- Username: **teamx**
- Password: **Passwordteamx**

with **x** being your team number



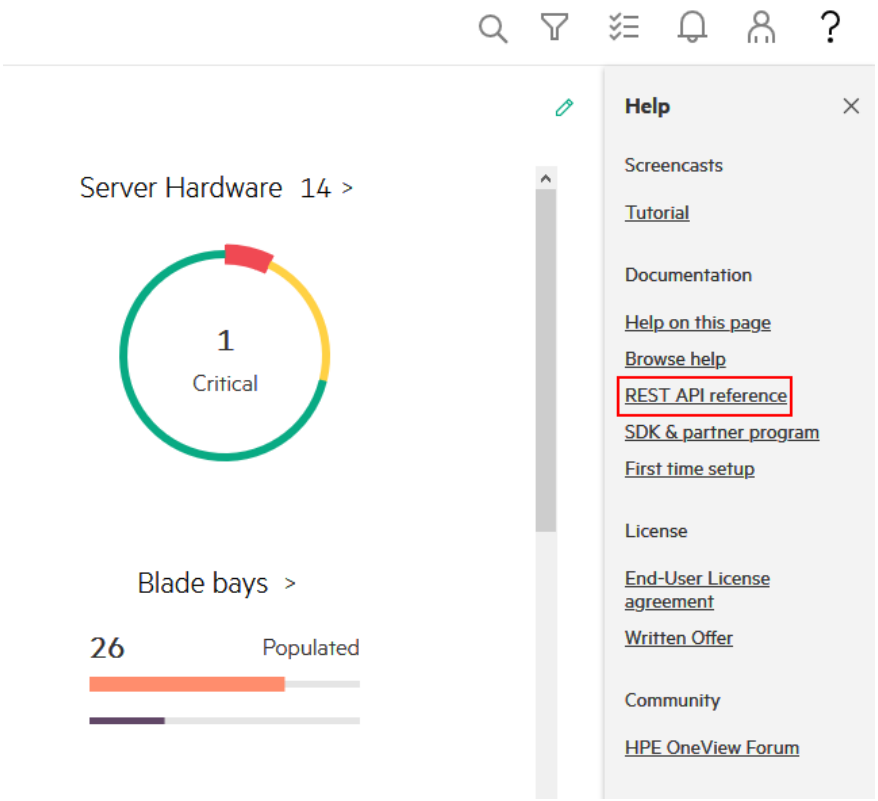
From the OneView interface, click on ? on the top right corner.



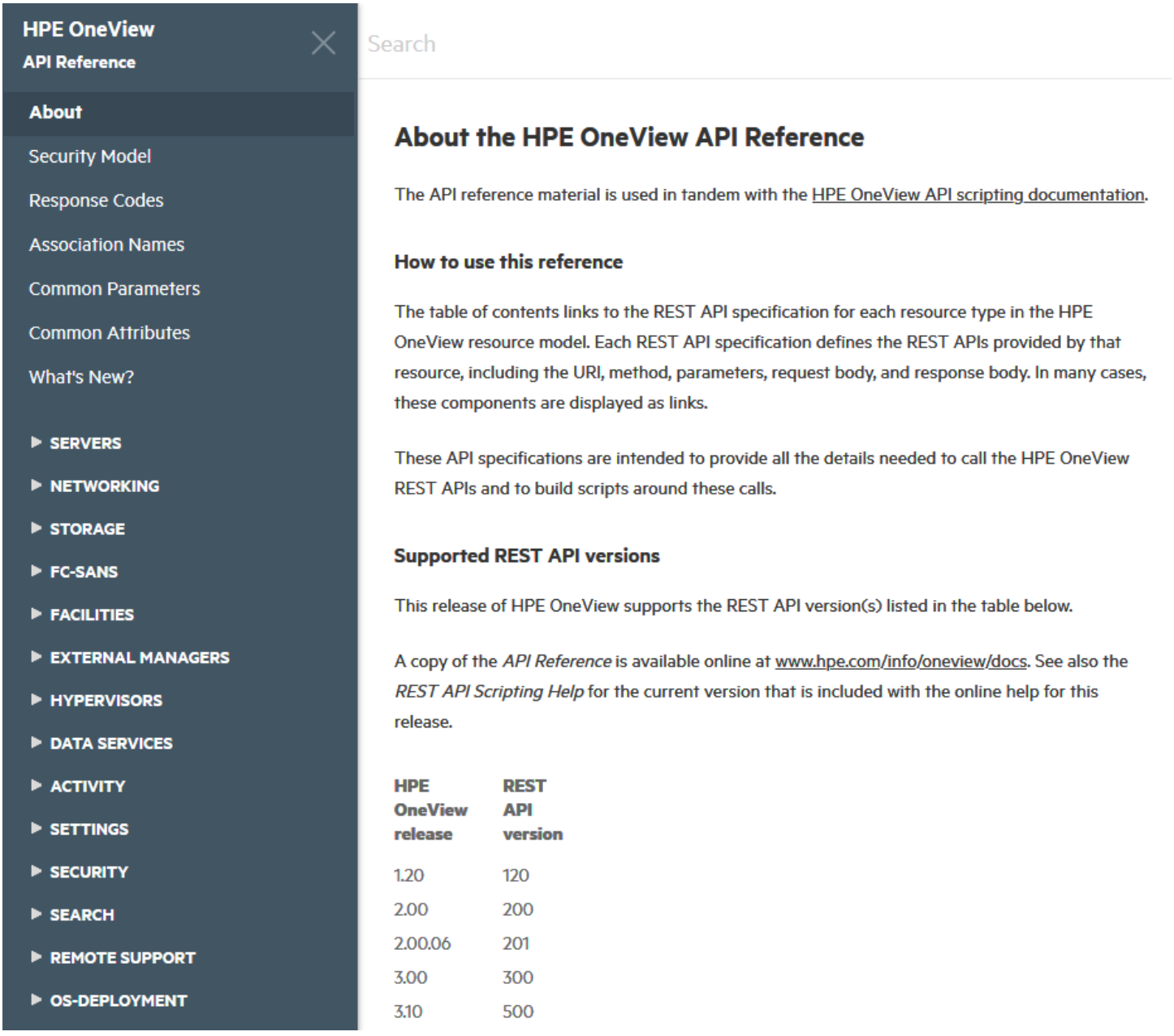
REST APIs are an architectural style following common characteristics and principles; they don't all follow the same standard or specification. Users must really read the documentation to understand how to use the API.

This detailed documentation is found in the *HPE OneView REST API Reference*

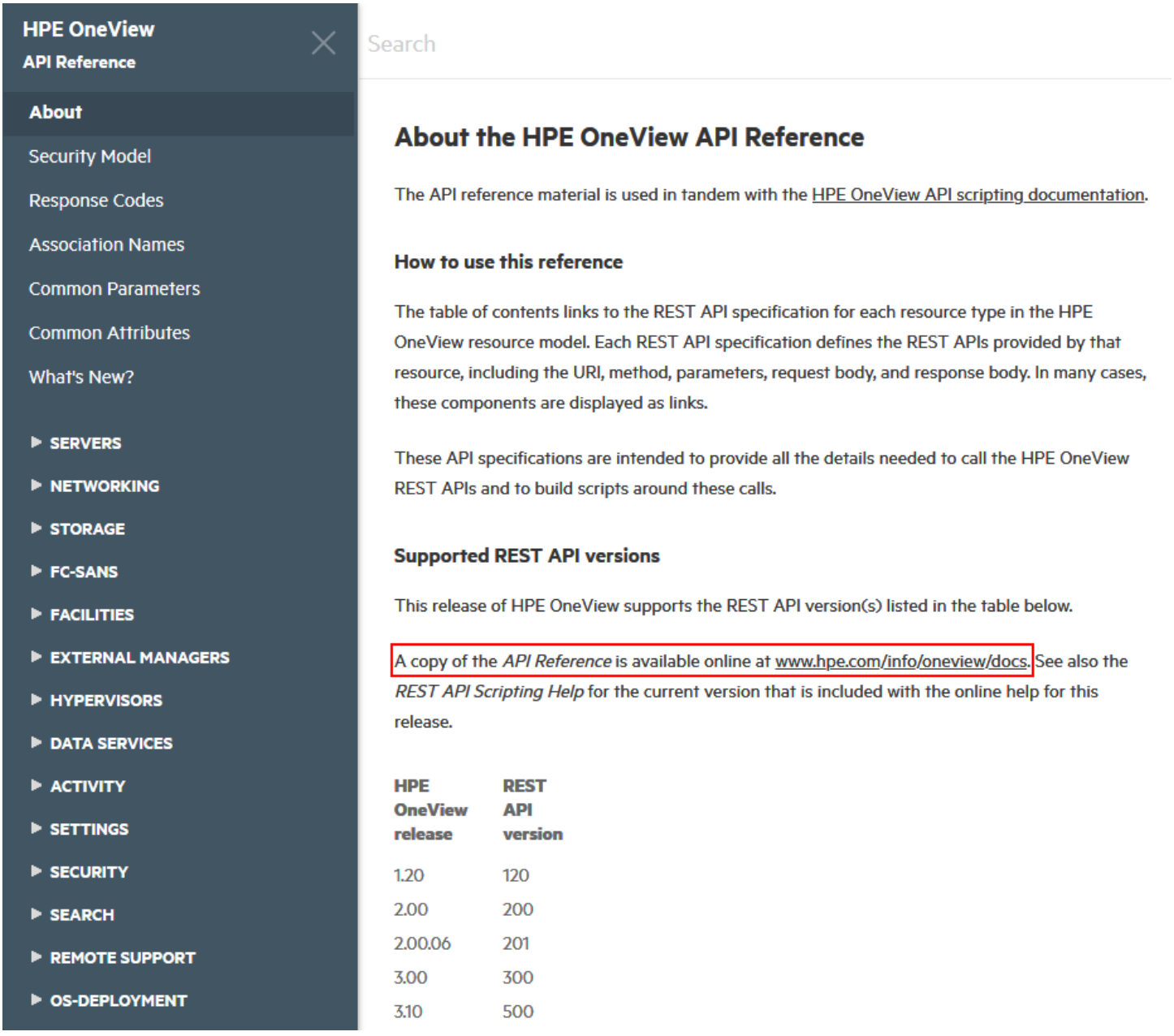
Click on the link **REST API Reference**



The Reference API document is where you can find detailed reference information about the REST API resource model schema and JSON examples.



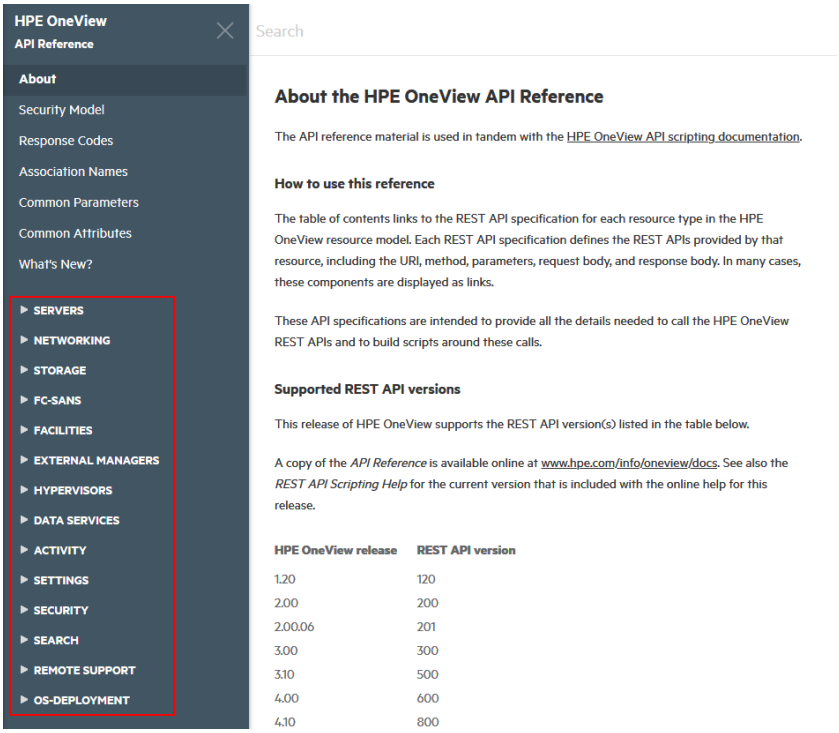
This HPE OneView REST API Reference is also available online, a nice to have when you don't have access to a OneView console



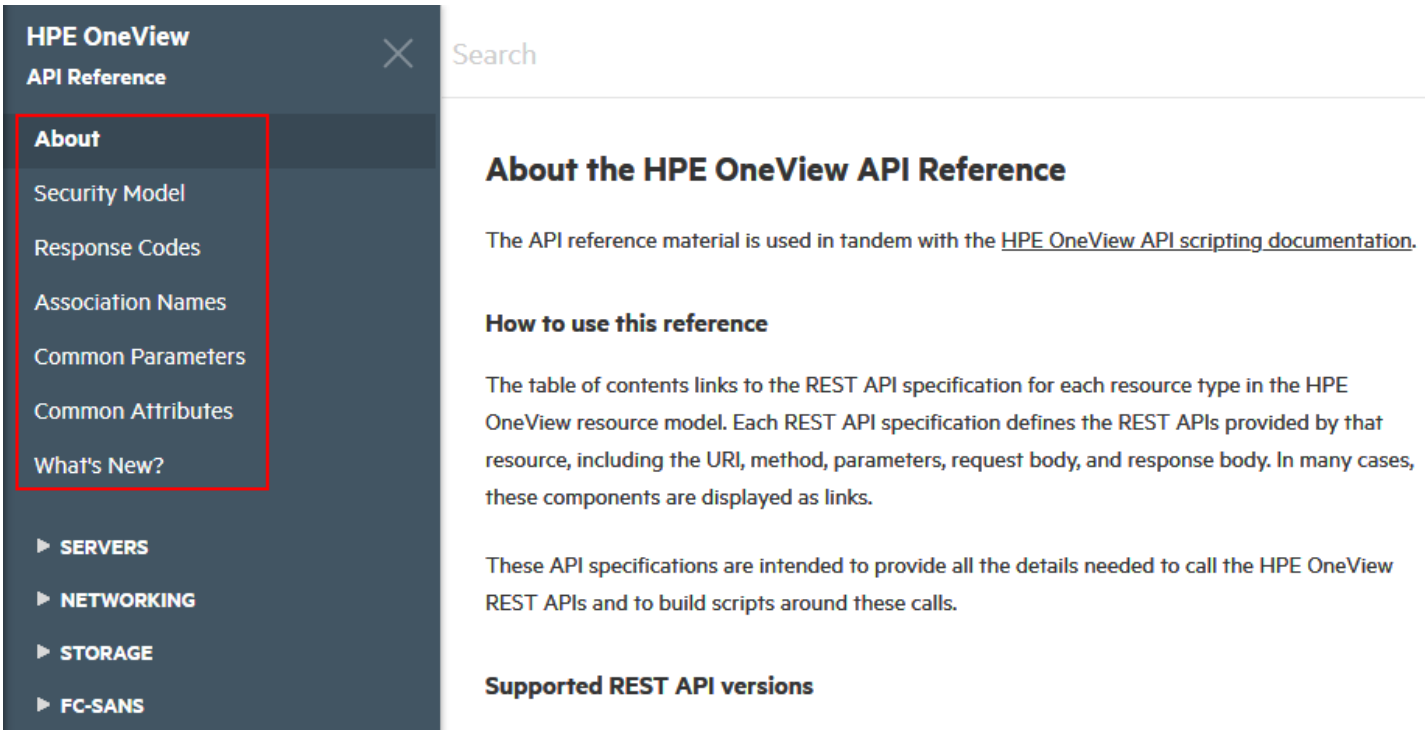
Two online copies of the *API Reference* (one is specific to HPE Synergy) are found in the User Guides section:

User Guides (20)	Type	Size	Date
Previous Versions of HPE OneView Documents on HPE Support Center	HTML		
HPE OneView 5.0 User Guide for HPE Synergy	PDF	2.0 MB	Aug 2019
HPE OneView 5.0 User Guide	PDF	1.3 MB	Aug 2019
HPE OneView 5.0 Help for HPE Synergy	HTML	N/A	Aug 2019
HPE OneView 5.0 Help	HTML	N/A	Aug 2019
HPE OneView 5.0 API Reference for HPE Synergy (API version 1200)	HTML	N/A	Aug 2019
HPE OneView 5.0 API Reference (API version 1200)	HTML	N/A	Aug 2019
HPE OneView 4.2 User Guide for HPE Synergy	PDF	2.0 MB	Feb 2019

Back to the HPE OneView REST API Reference page, we can see that each single API call is regrouped in Resource types (i.e. Servers, Networking, Storage, etc.) Activity, Search and Settings to ease the search. But you can also simply enter a resource name in the search pane to find what you need.

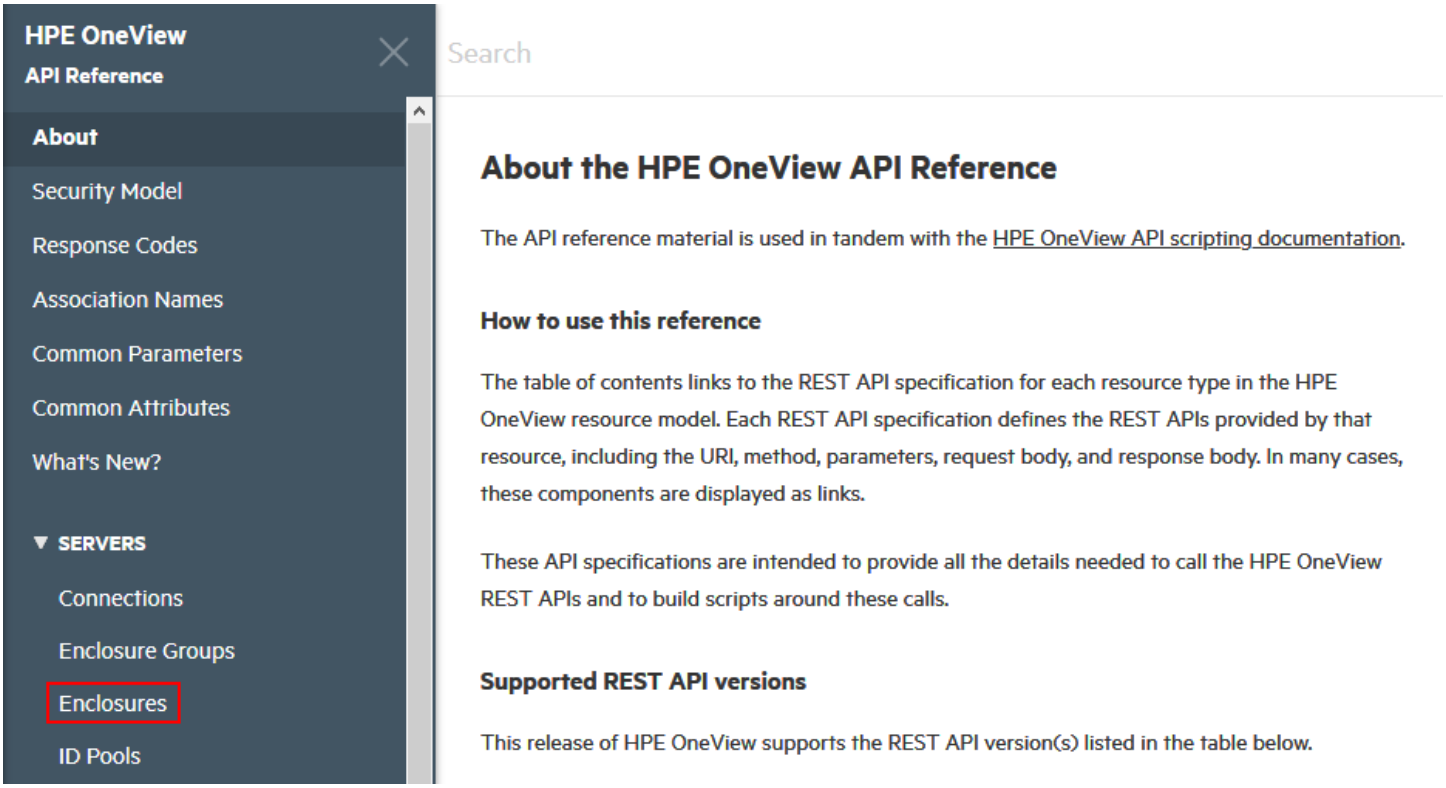


The group above provides important information about the API (supported versions), security model (authentication and authorization), association names (association and relationships between resources), response codes (description of the different status codes), common parameters (how to create filters, count, sort, query), common attributes (description of the type attributes commonly used) and what's new.

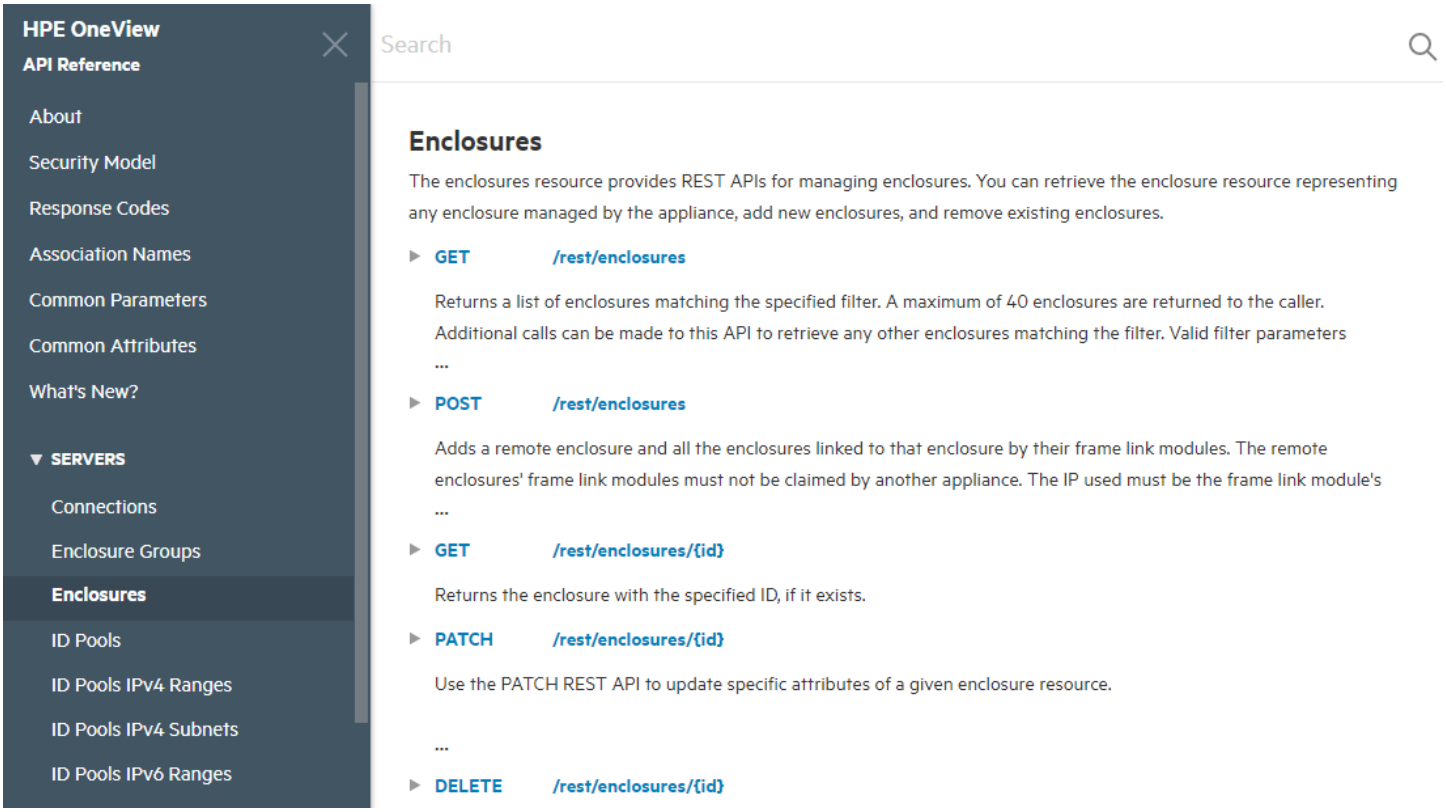


Navigate rapidly through the different menus to explore their content.

Click now on **Server** / **Enclosure** as an example:



The following is displayed:



There is a page like this for each API resource method available in HPE OneView. An API resource method is an API invocation that mostly of the time contains a request and/or response property.

From the information provided, we can see all sorts of API invocation related to Enclosures, some `GET /rest/enclosures`, some `POST`, `PATCH`, `DELETE`, etc.

Each resource method performs a specific task. To get a full description of an invocation, you need to click on the expand icon.

Expand `GET /rest/enclosures`

Enclosures

The enclosures resource provides REST APIs for managing enclosures. You can retrieve the enclosure resource representing any enclosure managed by the appliance, add new enclosures, and remove existing enclosures.

GET **/rest/enclosures**

Returns a list of enclosures matching the specified filter. A maximum of 40 enclosures are returned to the caller. Additional calls can be made to this API to retrieve any other enclosures matching the filter. Valid filter parameters include attributes of an EnclosureV7 resource.

Get a list of the enclosures with a status of OK.

Request

```
GET https://{appl}/rest/enclosures?filter="status='OK'"

Auth: abcdefghijklmnopqrstuvwxyz012345
X-Api-Version: 1200
```

Authorization

The Auth request header must contain a valid session token, and a role of that session must grant the below Action to the Category. Access is not restricted by scope.

Action `Read`

Category `enclosures`

What you get is always following the same pattern:

- **A description**: this invocation will return a list of enclosures.
- **One or more request examples**: showing example(s) of how to build invocations and what needs to be provided in the URL and in the headers:

```
GET https://{appl}/rest/enclosures?filter="status='OK'"

Auth: abcdefghijklmnopqrstuvwxyz012345
X-Api-Version: 1200
```

Hearders

With POST/PATCH/PUT invocations, headers and body are usually required:

```
POST https://{appl}/rest/enclosures

Content-Type: application/json
Auth: abcdefghijklmnopqrstuvwxyz012345
X-Api-Version: 1200

{
   "hostname" : "fe80::1:2:3:4",
}
```

- A **request headers section**: listing all the requested headers that have to be provided:

Request Headers

Accept-Language	The language code requested in the response. If a suitable match to the requested language is not available, en-US or the appliance locale is used.	string
Auth	Session authorization token obtained from logging in. If this header is not included or if the session-token is invalid, the response code will be 401 Unauthorized.	string required
Content-Type	The data format sent in the request body. If the Content-Type header is not provided, application/octet-stream is assumed. Any value other than the required value will result in a response code of 415 Unsupported Media Type. Required Value application/json	string required
X-Api-Version	Specifies the version of the API to invoke. The behavior of a given API version remains the same. It is upward compatible from release to release. New versions of an API are created when new features or changes are introduced. To ensure expected behavior, always provide the X-Api-Version value.	integer required

- Then depending of the type of invocation we have:
  - With GET:
    - **Query parameters**: describing the different query options: start, sort, count, fields, filter, query, etc.:

Query Parameters

`count`	The number of resources to return. A count of -1 requests all the items. The actual number of items in the response can differ from the requested count if the sum of start and count exceed the total number of items, or if returning the requested number of items would take too long. See the Common Parameters page for more information on the usage of count. Default -1	integer
`fields`	Specifies which fields should be returned in the result set.	string
`filter`	A general filter/query string to narrow the list of items returned. The default is no filter - all resources are returned. See the Common Parameters page for more information on the usage of filter.	array of string
`query`	A general query string to narrow the list of resources returned. The default is no query - all resources are returned. See the Common Parameters page for more information on the usage of query.	string

    - **Response body:** describing all the resource members that can be found in the response body:

Response Body	EnclosureListV7	
`category`	Identifies the resource type	string read only
`count`	The actual number of resources returned in the specified page	integer
`created`	Date and time when the resource was created Format yyyy-MM-dd'T'HH:mm:ss.SSS'Z' Pattern [1-2][0-9][0-9][0-9]-([0-1][0-9])-[0-3][0-9]T[0-2][0-9]:[0-5](.[0-9][0-9][0-9])?Z	timestamp read only
`eTag`	Entity tag/version ID of the resource, the same value that is returned in the ETag header on a GET of the resource	string read only
`members` array of EnclosureV7	List of enclosures resources.	
`applianceBayCount`	The number of appliance bays in the enclosure.	integer read only
`applianceBays`	A list of the appliance bays in the enclosure.	Appliance read only
`bayNumber`	The bay number of the appliance.	integer read only
`bayPowerState`	The power state of the appliance bay.	

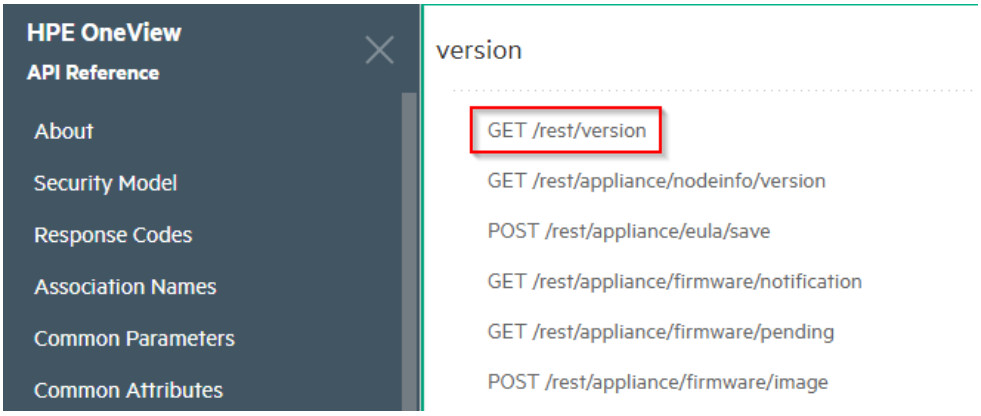
- With POST/PUT/PATCH/DELETE:
  - **Response header:** providing the newly created resource or task created to perform the requested action.

- With POST/PUT/PATCH/DELETE:
  - **Request Body:** describing the content of the body that needs to be provided)

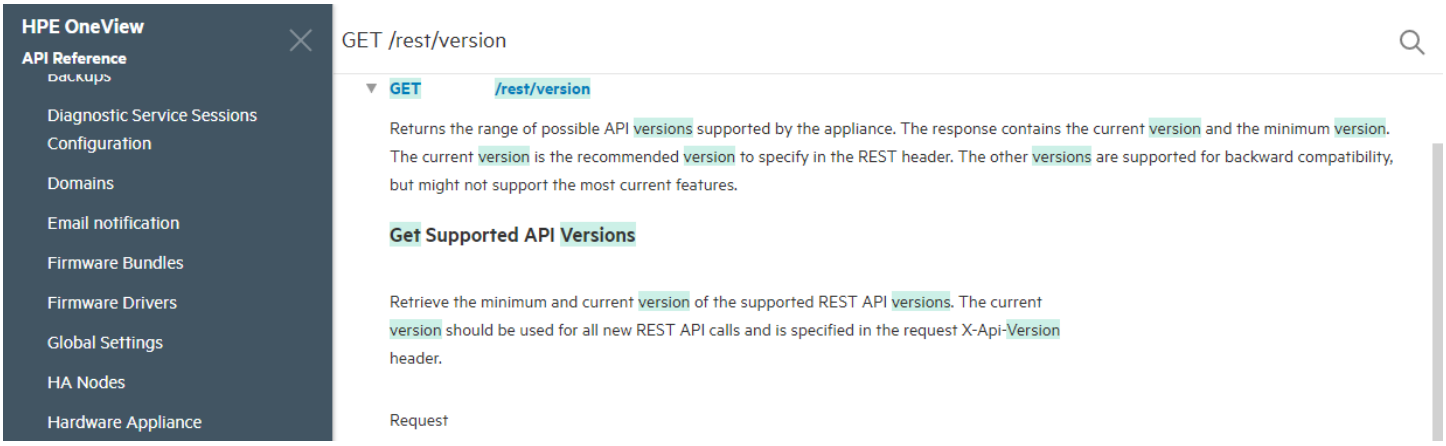
Request Body	array of PATCH Operation	
`from`	The JSON path to use as the source of a "move" or "copy" operation. Not used by "add", "remove", "replace", or "test".	string read only
`op`	The type of operation: one of "add", "copy", "move", "remove", "replace", or "test".	string required read only
	Default Operation	
`path`	The JSON path the operation is to use. The exact meaning depends on the type of operation.	string required
`value`	The value to add or replace for "add" and "replace" operations, or the value to compare against for a "test" operation. Not used by "copy", "move", or "remove".	any type read only

All this might sound very theoretical but don't worry, we are soon going to execute our first API call and you will get a much better understanding and fun.

As a first step toward the fun, enter **version** in the search pane then select `GET /rest/version`



You will find something like this:



From the information provided, we can see that a call to `GET /rest/version` will return the versions supported by the API. We will use this later for our first interaction with the HPE OneView API.

If you scroll down in the help page, you will also find what kind of response is expected:

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "currentVersion" : 1200,
  "minimumVersion" : 1
}
```

We can see that this API returns a Content-Type of JSON (JavaScript Object Notation). This is a very popular format used to exchange data with an API. It is like XML, just a little more human readable.

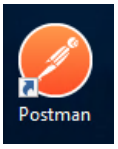
Many APIs are moving away from XML as the exchange format in preference of JSON. HPE OneView uses JSON for input parameters, and responses. There are many web sites for you to learn about JSON, but I have found this online parser very helpful to check the syntax of a JSON payload before using it: http://json.parser.online.fr/

Lab 2: Getting started with Postman

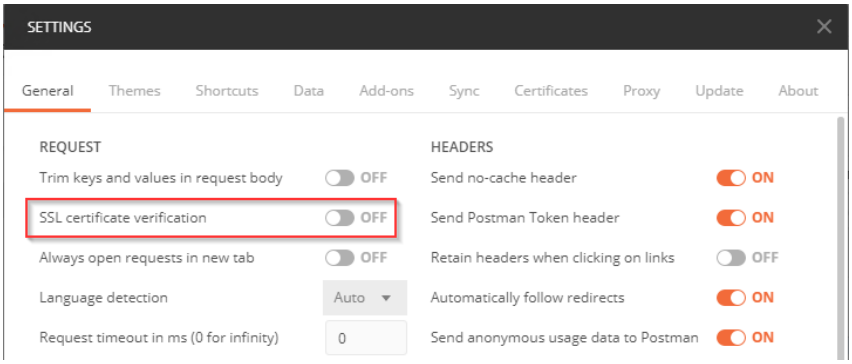
So now you have all the information you need to place your first HPE OneView API call to retrieve the version numbers. However, if you are new to REST programming, you might ask yourself: how do I place a REST call to an API without writing any code?
There are several simple solutions for this. My favorite is Postman but there are also several browser plug-ins available. You can install Postman as a Native App or a Chrome Extension.
Postman is one of the most complete REST tools and offers useful features like:

- You can save your REST calls and share the collections with others
- You can use variables to store for instance the OneView authentication session keys

Open **Postman** using the shortcut on your desktop:

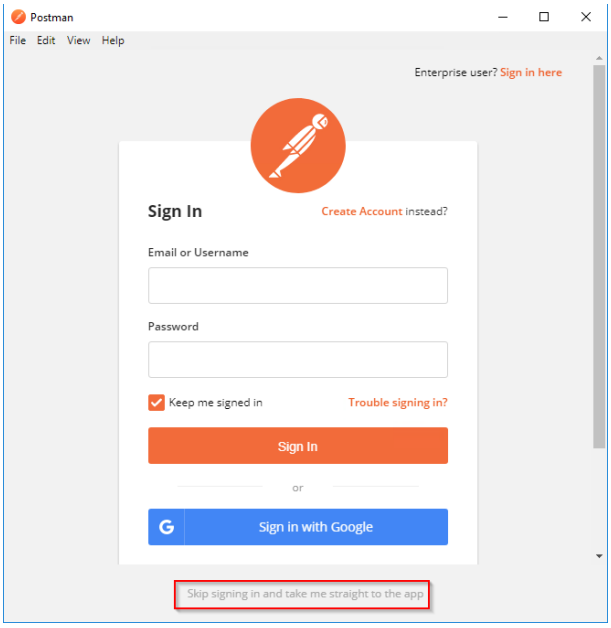


Note: Postman can be downloaded from https://dl.pstmn.io/download/latest/win?arch=64
Note: In Settings, you must set **SSL certificate verification** to **OFF** if OneView uses a self-signed certificate

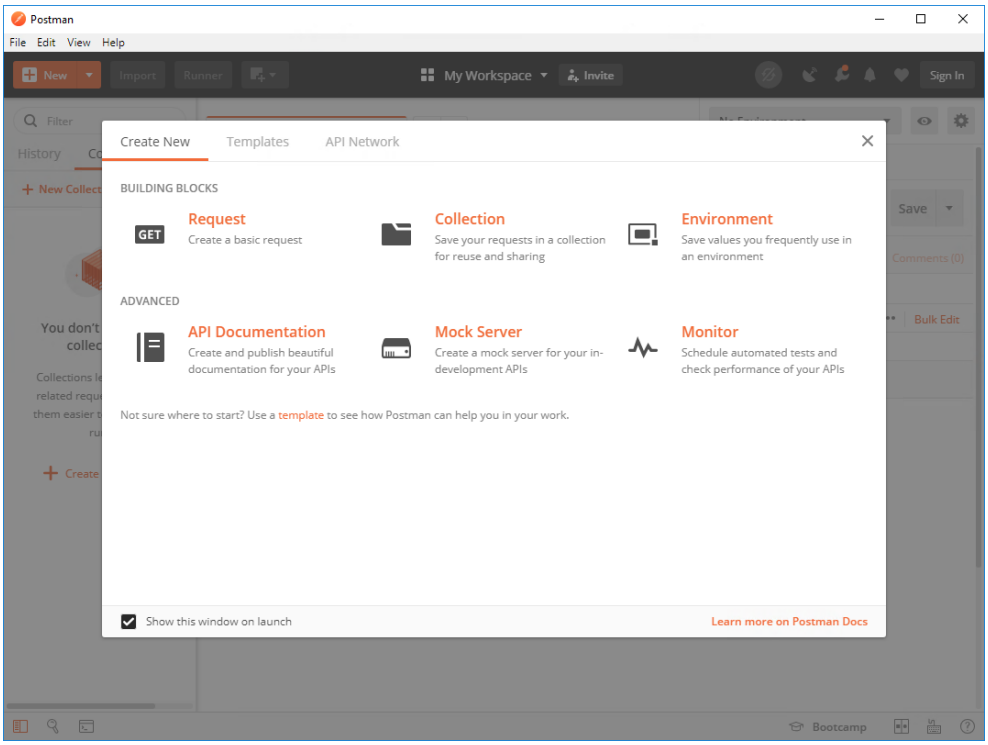


Note: If you are not very comfortable with Postman, two Postman collections are available in e:\Postman\ with all the REST calls we are creating in these labs. If these collections are not already in the Postman Collections pane, click the *Import* button in the header bar and go to e:\Postman then select the two files to import (*OneView Lab collection.postman_collection.json* and *Global DashBoard collection.postman_collection.json*)

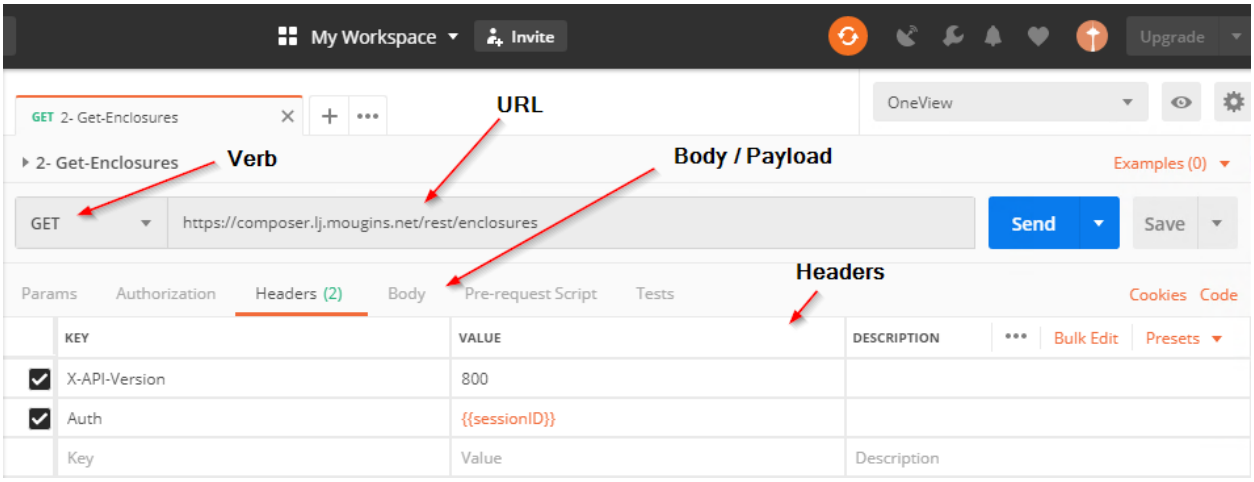
If the following window opens, you must click on **Skip signing in and take me straight to the app** at the bottom to start Postman without signing:



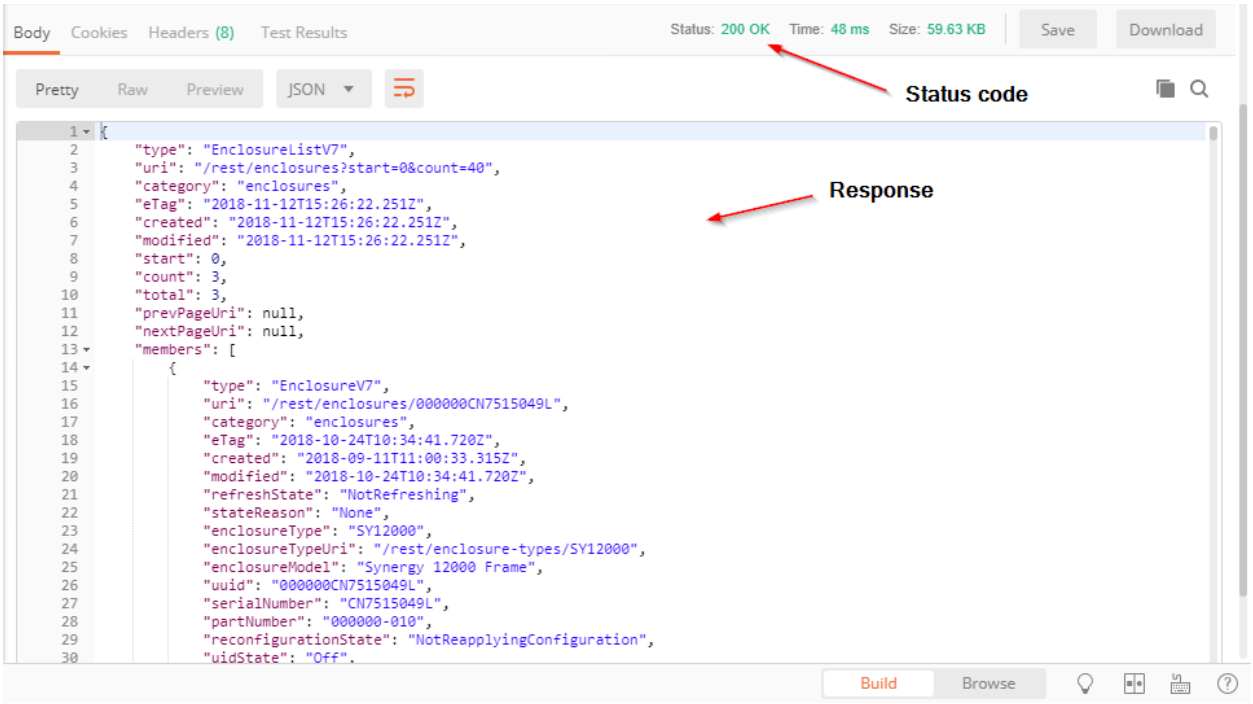
Then you can close the following window:



Before using Postman, let's take a quick look at what the main pane provides. There are two main sections, on the upper one, we find where we can enter the query information (verb, URL, header/body, etc.):



And on the lower section is where the result of the REST query resides with the body response, the response status, etc.:



Let's make a try with `GET /rest/version` we saw previously.

If you go back to the *HPE OneView API Reference* documentation, you will find in the help what type of request is required to pass the `GET /rest/version` (returns the range of possible API versions supported by the appliance):

Request

```
GET https://{appl}/rest/version

Accept-Language: en_US
```

So, in more details, the documentation provides the following:

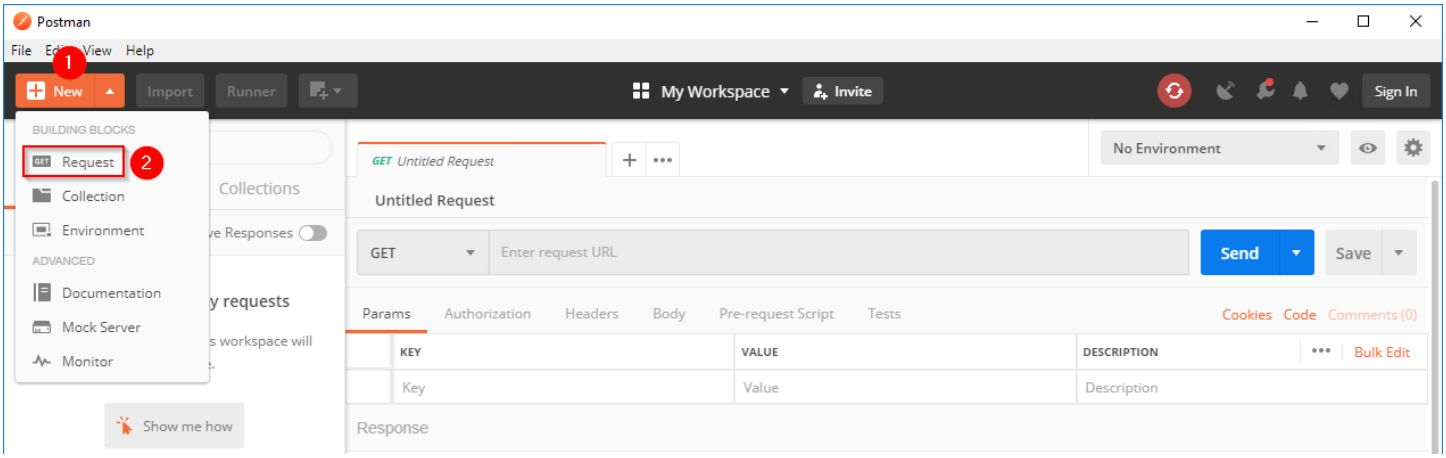
Field	Value
URL	`https://{appl}/rest/version`
Verb	`GET`
Header	`Accept-Language: en_US`

`Accept-Language: en_US` in the HTTP Header is optional. It just advertises which languages the client is able to understand so it will work without it.

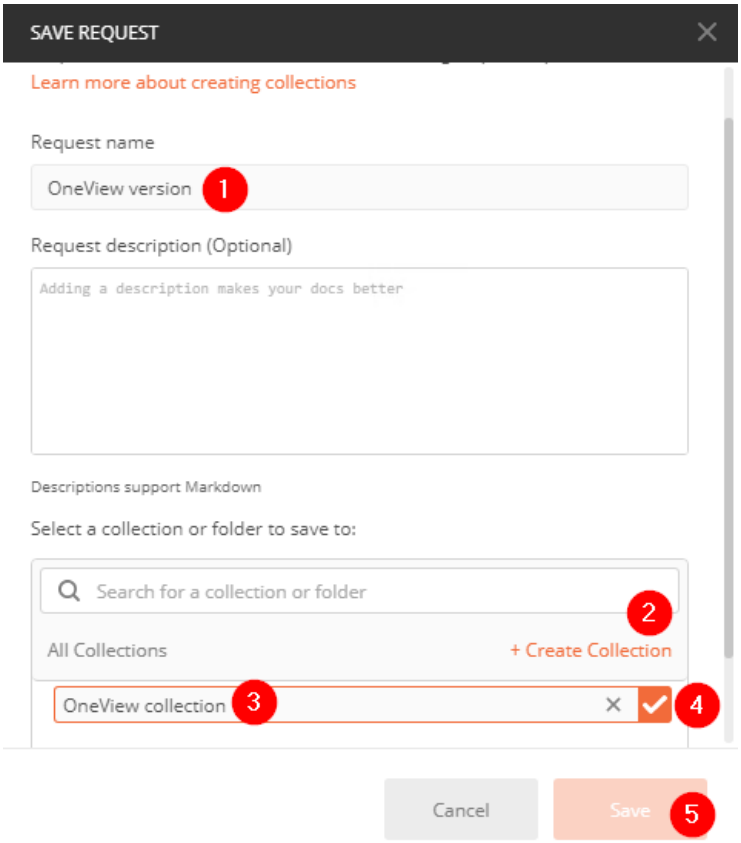
Optionally, you can specify in the header `Accept=application/json` to explicitly tell the API that we expect a response in JSON. You might need to do that sometimes to avoid an XML response.

You can note that this request does not require `Auth` (any authentication) or `X-Api-Version` headers. So, there is no request payload to worry about for the `GET /rest/version` call

Back to Postman, click the **New** button in the header bar and select **Request** to create our first API query:

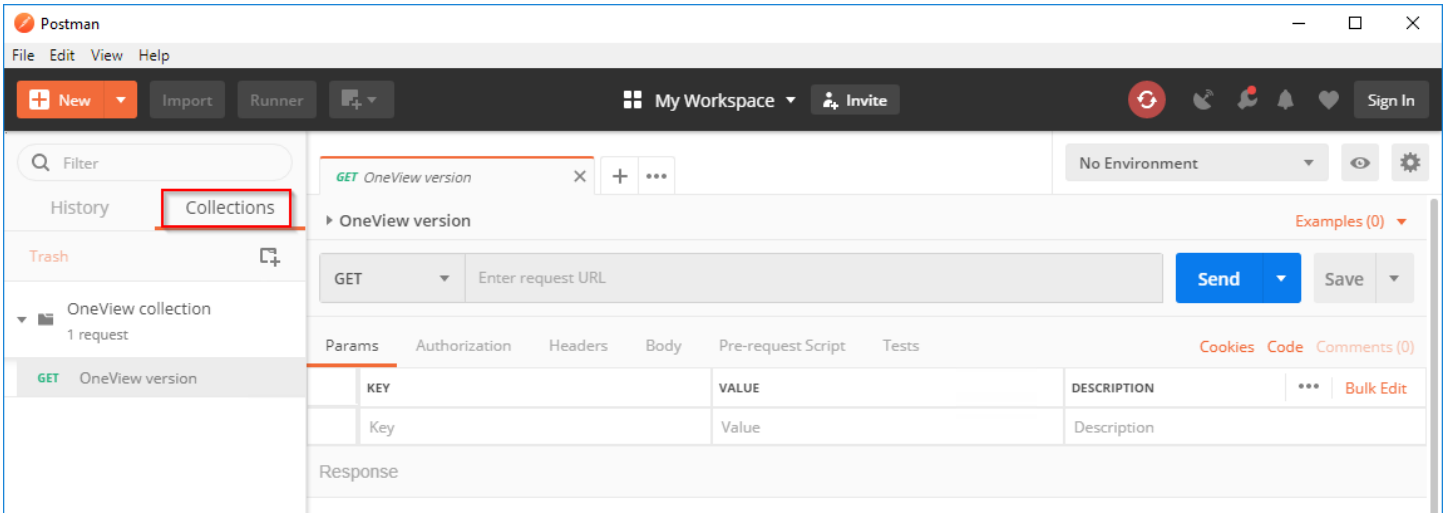


Enter a request name like **OneView Version** then click **Create Collection** and enter a name like **OneView collection**:



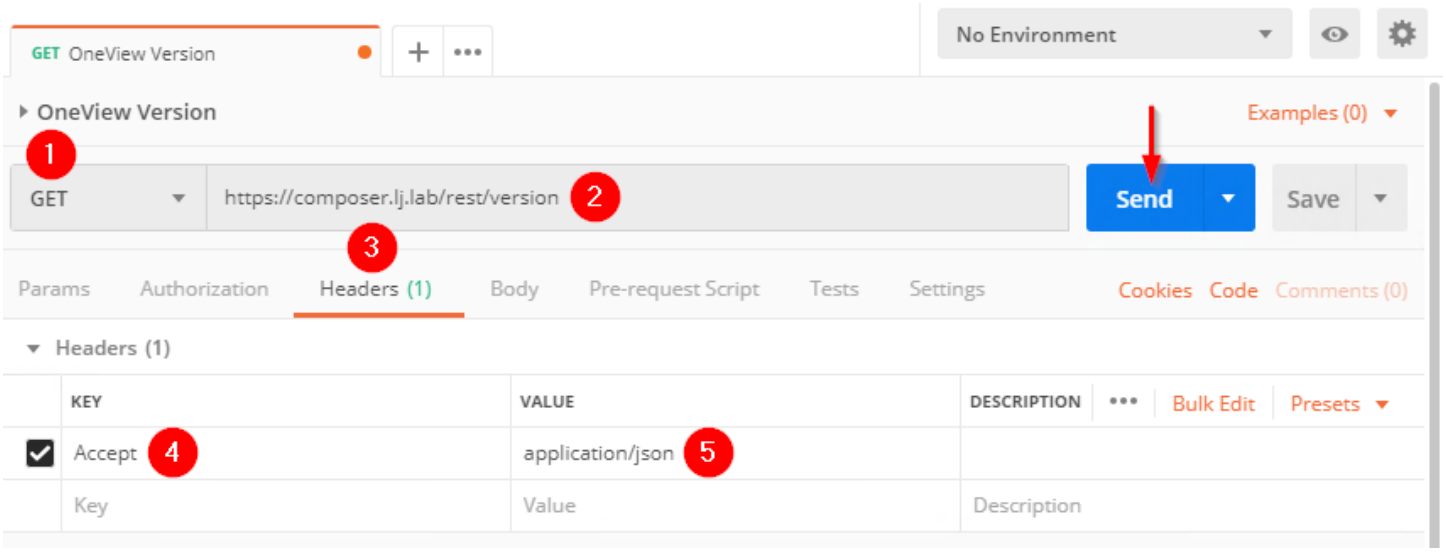
Click on the validate icon to save the collection name then select your new collection and click **Save to OneView collection**.

To display the new collection, you can select **Collections** on the left-hand pane:



Then you can enter the following information to create your first query:

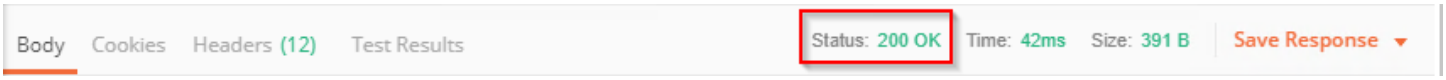
- In ① Select in Verb: **Get**
- In ② Enter the URL: **https://composer.lj.lab/rest/version**
- In ③ Select **Headers**
- (optional) Enter in *key* ④: **Accept**
- (optional) Enter in *value* ⑤: **application/json**



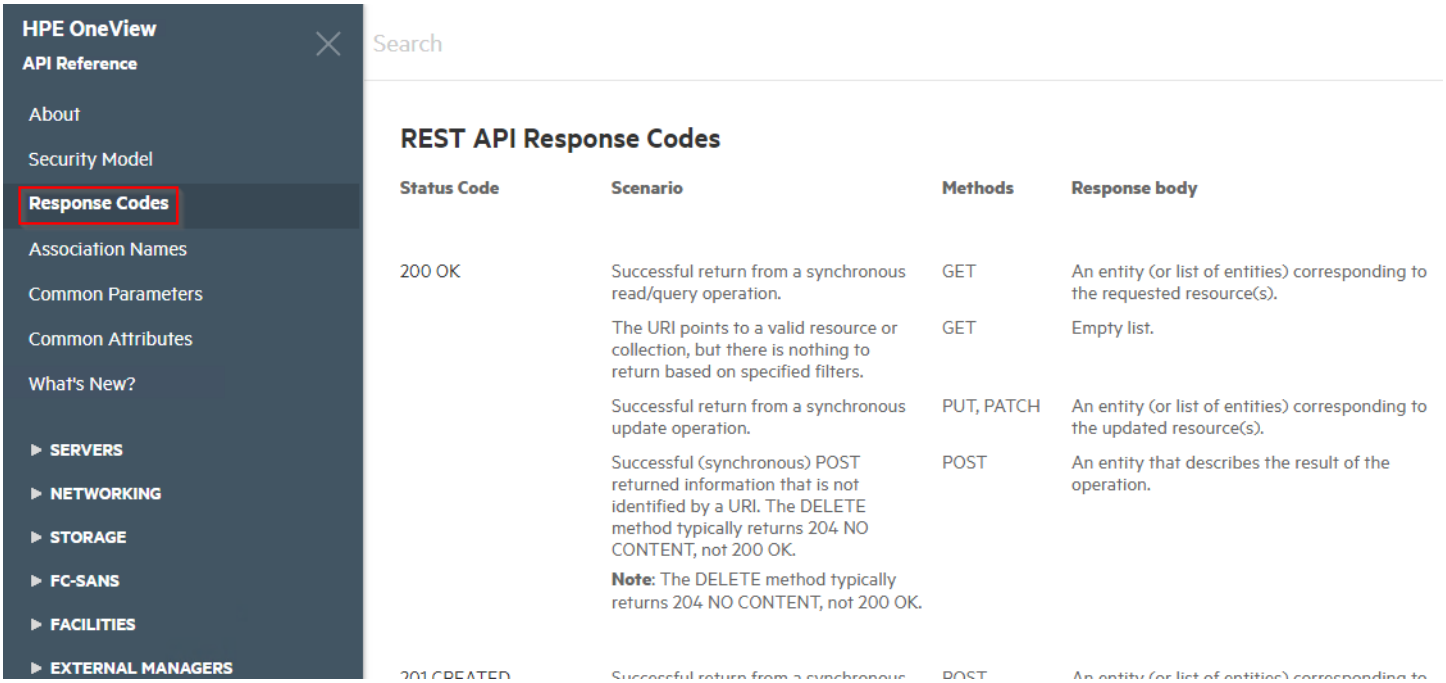
When ready, press the **Send** button.

Postman will contact the URL specified and invoke `GET version` from the API.

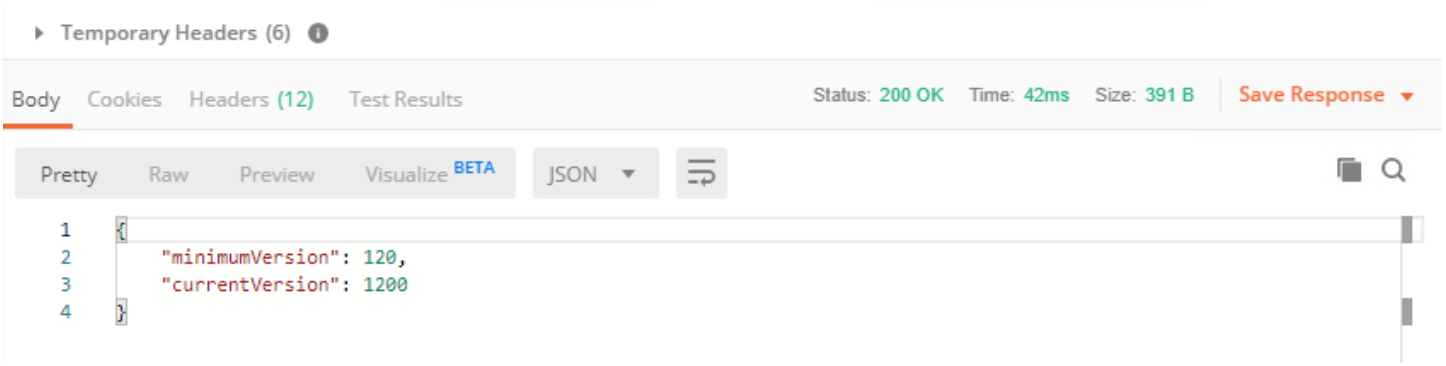
In the Response section, as illustrated below, you want to check the HTTP status code. A value of `200` means it was successful.



Unsuccessful status code can be *400 Bad request*, *401 Unauthorized*, *415 Unsupported Media Type*, etc. The list of all status codes with some troubleshooting indications is available in the OneView *API Reference*. Those indications can be useful for troubleshooting when your request is failing:



Next, you can check the Response body formatted as JSON, to find out about the version. This response will have to be parsed to retrieve the `currentVersion` value. But we can read easily that `currentVersion=1200` (meaning HPE OneView v5.0)



Congratulations! You have successfully placed your first call to the HPE OneView API. As you have seen, `GET version` does not require any authentication. In fact, it is one of the few calls you can place without having to provide proper authentication.

In subsequent sections, we will dive deeper into the API and learn about how to authenticate and retrieve information about the HPE OneView resources.

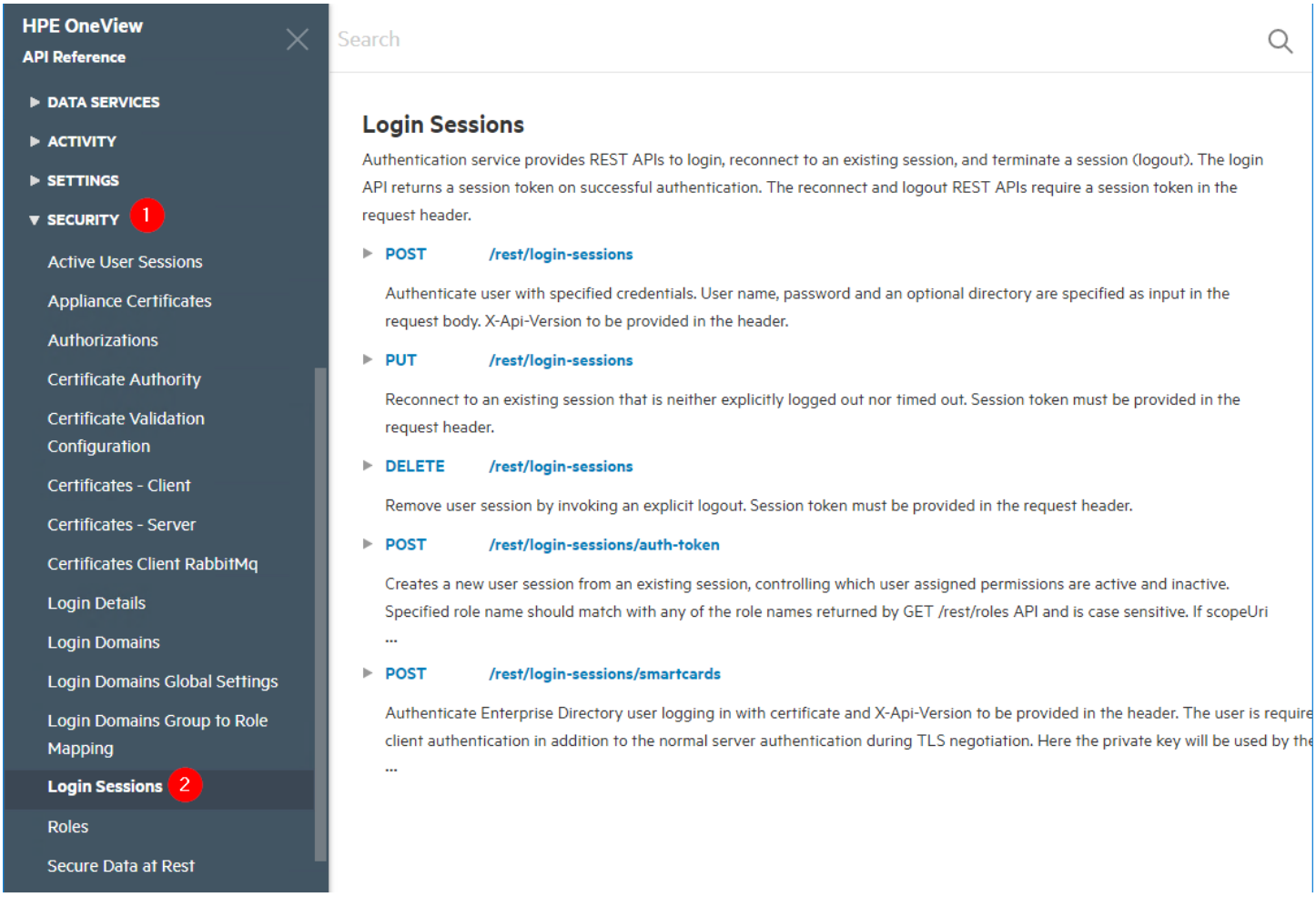
Although this example might seem very basic, any integration with HPE OneView should always start by querying the API versions supported, therefore, this is always going to be your very first step.

Lab 3: Login Sessions

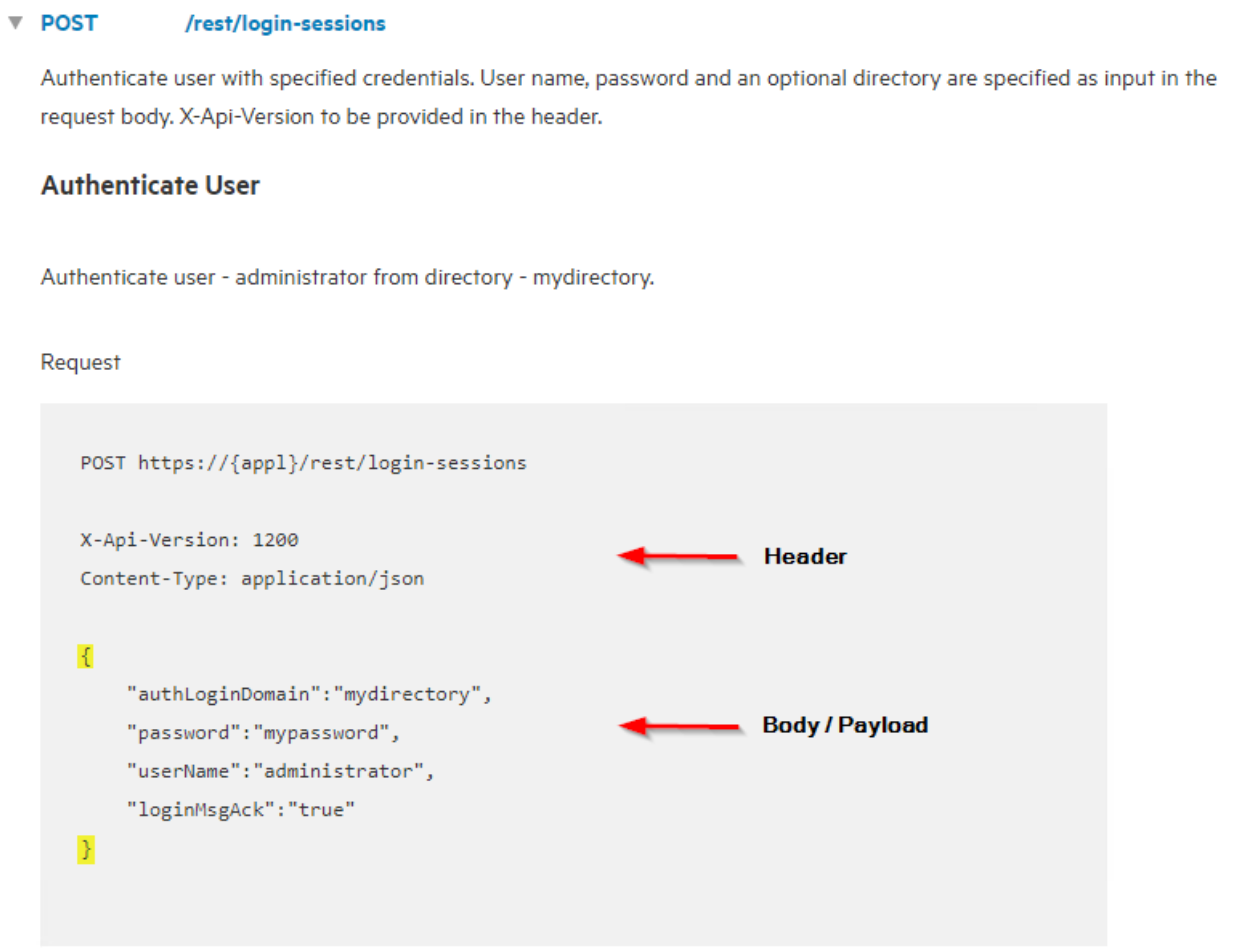
In the previous section, we learned how to query the version of the HPE OneView API, using the `GET /rest/version` call. Let us now move forward and open a login session to HPE OneView.

First, let's check the syntax of the login session by selecting **Security** / **Login Sessions** in the *API Reference*.

This is what we find:



If you expand the first POST, we have:



From the documentation, we can see that in order to open a login session, we need to use the `POST` method against the URI `/rest/login-sessions`. This will create (remember `POST` is the verb used in HTTP to create something) a session token, which we will use in the subsequent calls to the API.

We will ignore the login domain information for now.

We can also see in the documentation, that we will need to set two HTTP Headers:

- `X-API-Version`: set to the version of the API we want to use (let's assume `1200`)
- `Content-Type`: set to `application/json`, to tell the API, that the payload we will be providing is in JSON notation

The payload expected by the API and notified by the `{ … }` must contains (at minimum) the following parameters:

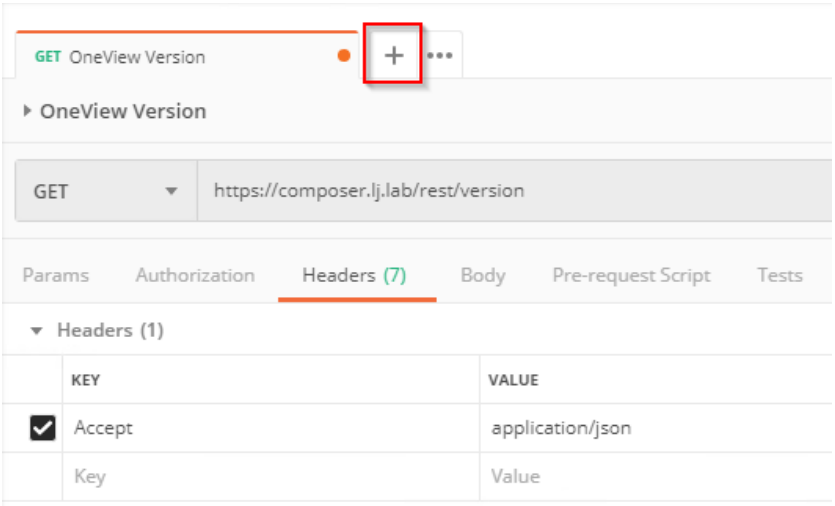
- `userName`: set to the username to use for authentication
- `password`: set to the password for that username

Note: Be careful, JSON is case sensitive. userName is expected here, not username, nor UserName. As a rule, the API expects lower case words, but capitalized, after the first word, if there are more than one word.

You can test this now using Postman using the provided parameters:

Field	Value
URL	`https://{appl}/rest/login-sessions`
Verb	`POST`
Header	`Content-type:application/json` `X-API-Version:1200`
Payload	`{ "authLoginDomain":"mydirectory", "password":"mypassword", "userName":"administrator", "loginMsgAck":"true" }`

So, click on **+** to create a new request

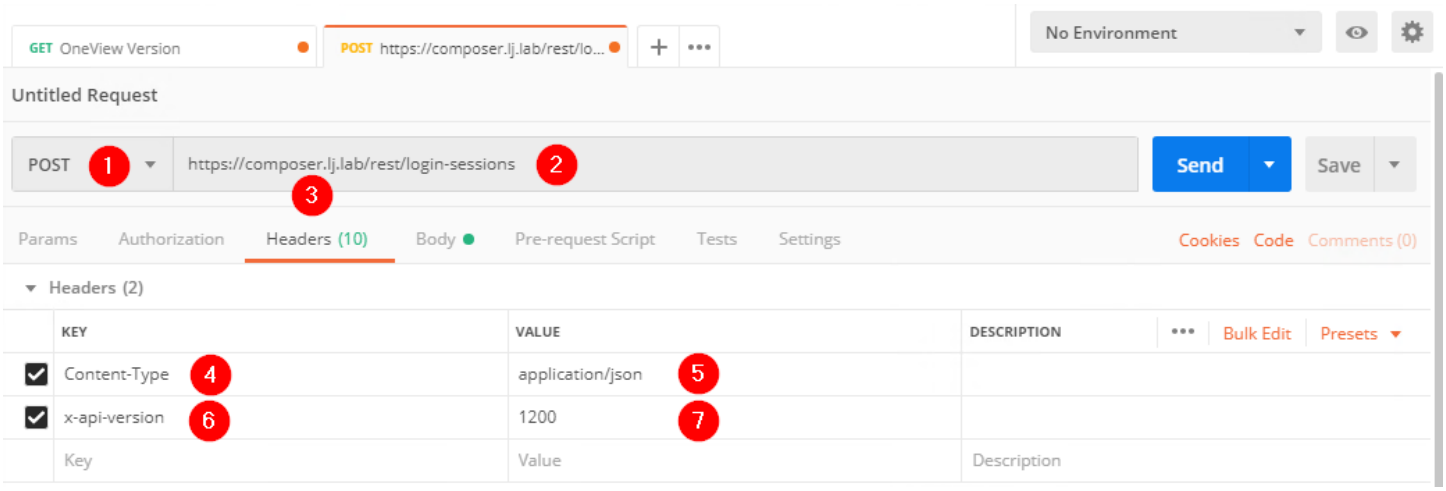


And enter the following information:

- In ① Select in Verb: **Post**
- In ② Enter the URL: **https://composer.lj.lab/rest/login-sessions**

Content of the headers:

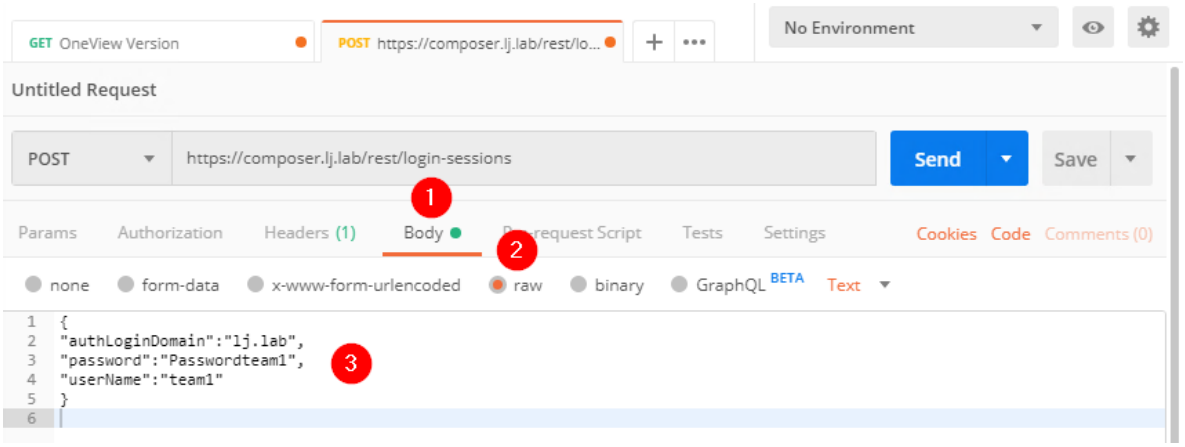
- In ③ Select **Headers**
- Enter in *key* ④: **Content-type**
- Enter in *value* ⑤: **application/json**
- Enter in *key* ⑥: **x-api-version**
- Enter in *value* ⑦: **1200**
-



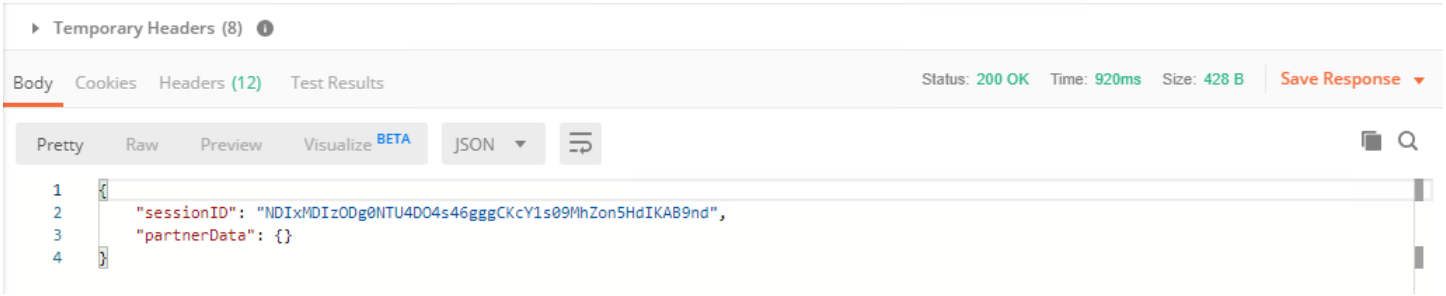
Content of the Body:

- Select **Body** in ①
- Select **raw** in ②
- Enter the payload in ③:

```
{
"authLoginDomain":"lj.lab",
"password":"<mypassword>",
"userName":"<myusername>"
}
```



Then press **Send**.
The response we get is the following:



The `sessionID`, also known as the session token, is what we are looking for. This is the token that will be used for authentication in the subsequent calls to the API.
So, in order to be authorized to call any other API methods, we will need to provide the session token as another HTTP Header called `Auth`.

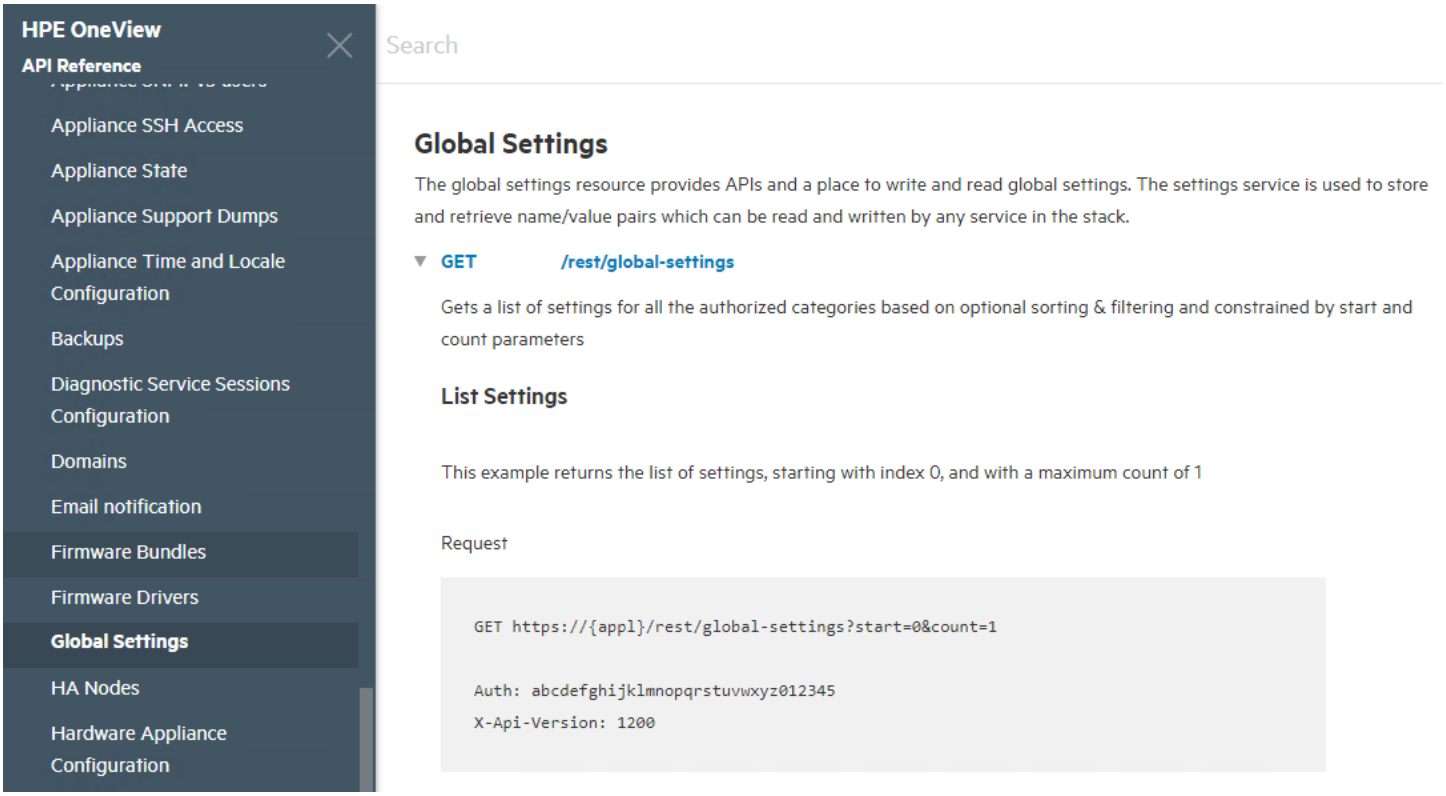
And remember, we now have two HTTP Headers, which must be provided at each call to the API: `Auth` and `X-API-Version`.

Note: You should also be aware that session tokens expire 24 hours after the last utilization.

We can now test this token and call another API method. Let us pick `GET /rest/global-settings` which returns the current settings of the HPE OneView appliance.

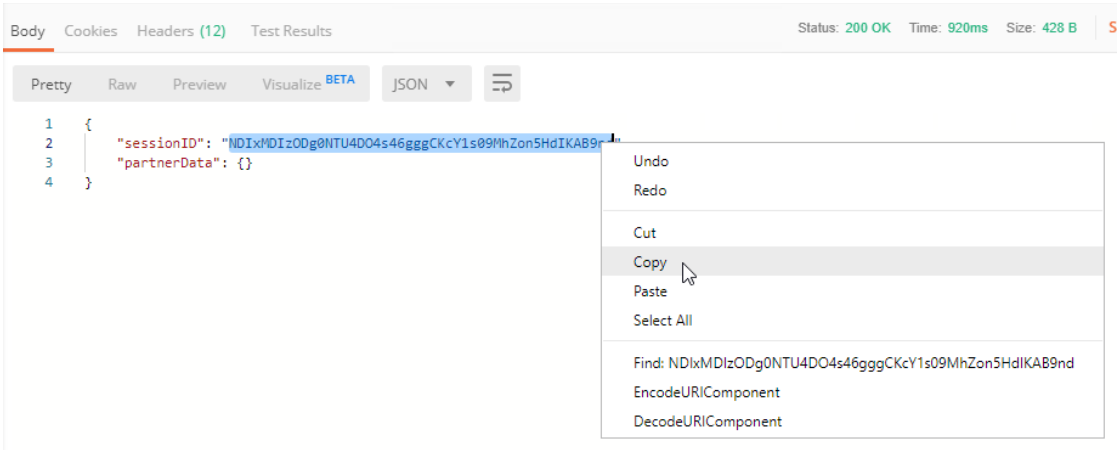
Back to the API documentation, enter **global** in the search pane and select `GET /rest/global-settings`

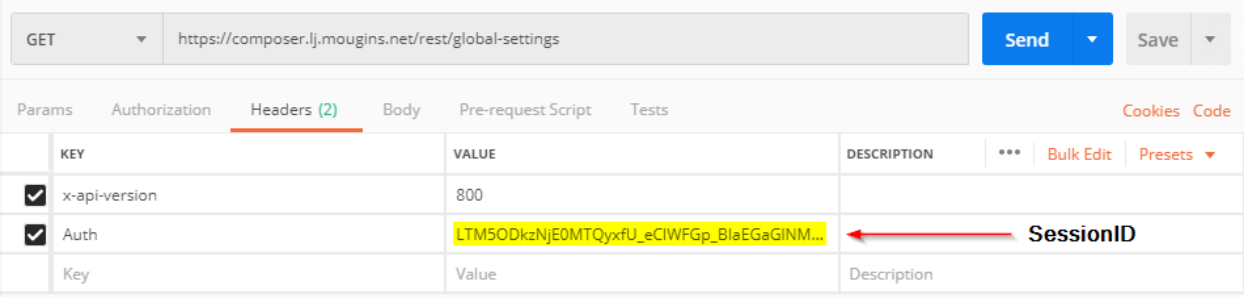
This is what we can read:



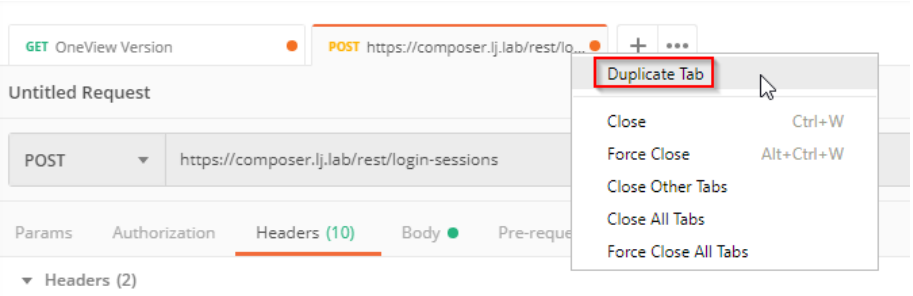
Notice now the presence of `Auth` with the `sessionID` in the header for this call.

So, before we use Postman with the parameters found here, it is necessary to record your *SessionID* from your *POST /rest/login-sessions*:



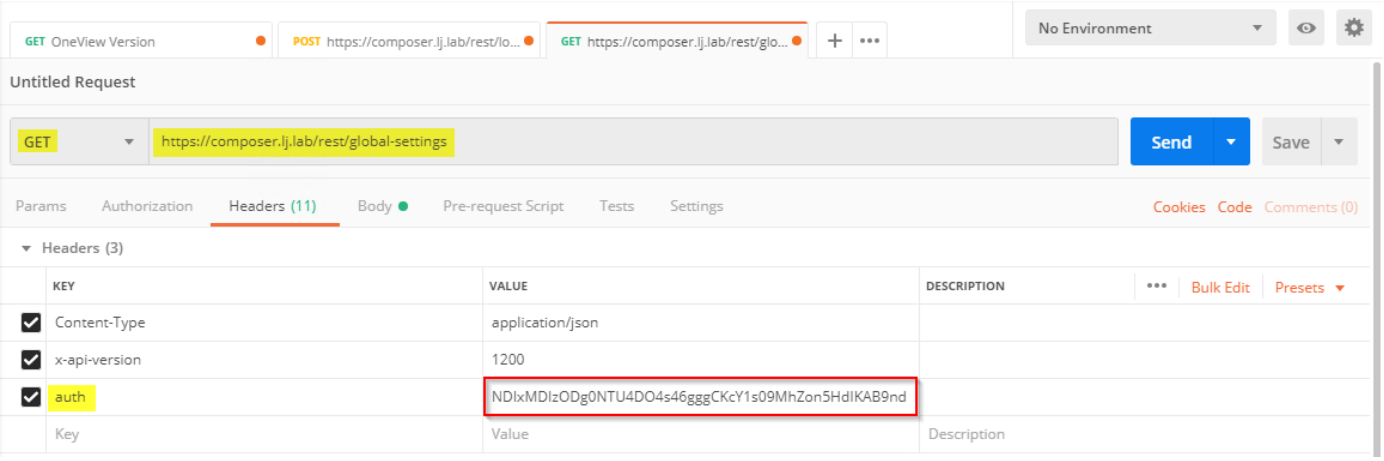


Then to create the new request, you can use the **Duplicate Tab** option to duplicate your *POST /rest/login-sessions* so that you don't have to re-enter the default headers parameters again:



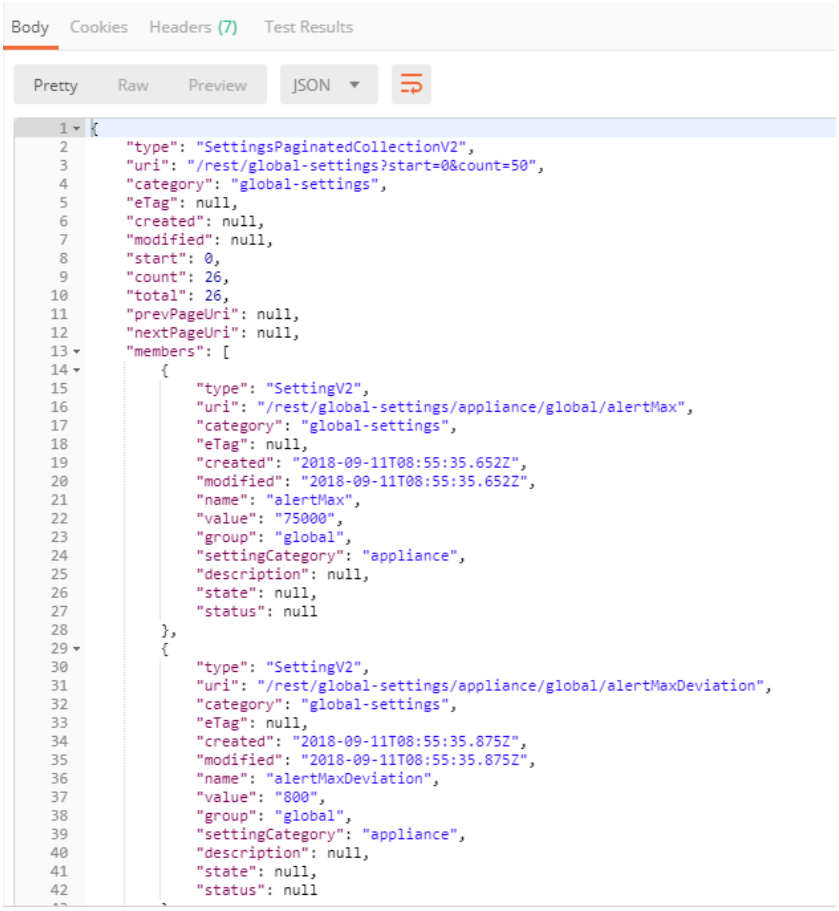
and change the parameters using the following configuration:

Field	**Value**
URL	`https://composer.lj.lab/rest/global-settings`
Verb	`GET`
Header	`Auth : <your sesionID>`

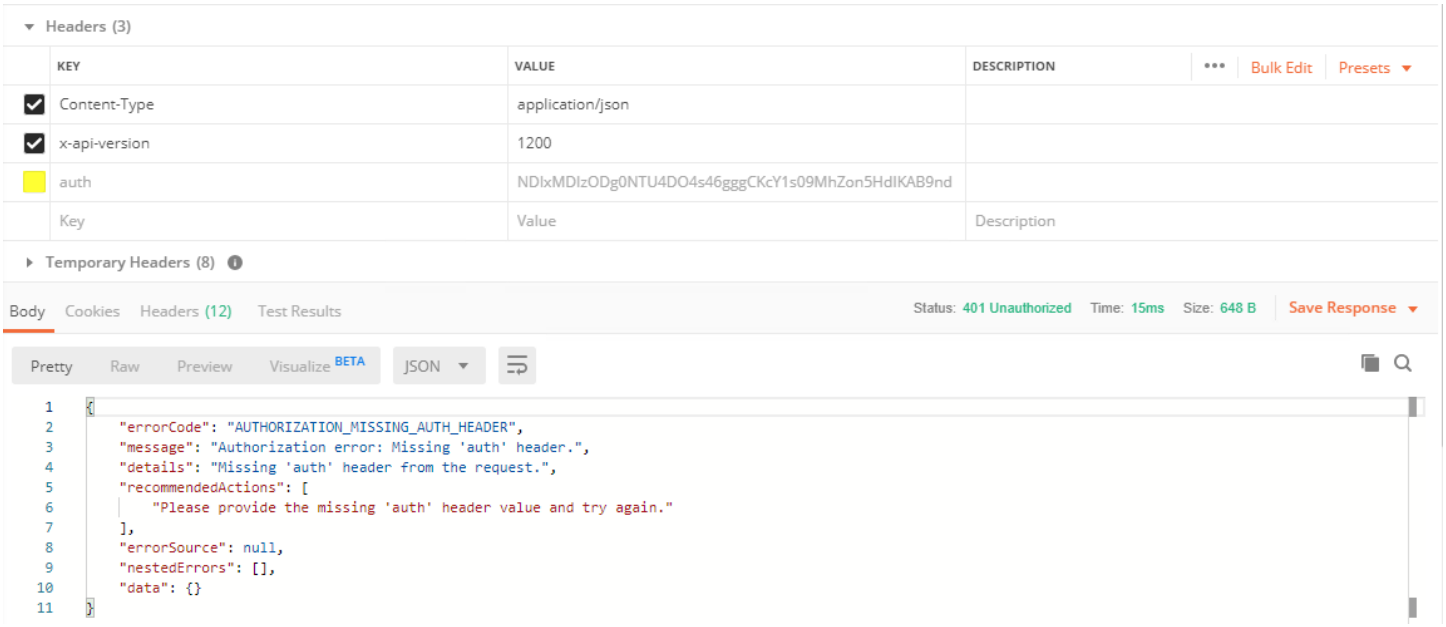


Then when all set, press **Send**

You should get the global settings information in the response body, again in a JSON format:



As a quick check, remove the `Auth` HTTP Header (uncheck the option), and try again.



You should get a `401 Unauthorized Status` code. The details of the error clearly state that you forgot to provide an `Auth` header for that request.

Note: HTTP Headers are case insensitive: `Auth` or `auth` would work fine.

Lab 4: API Version and backward compatibility

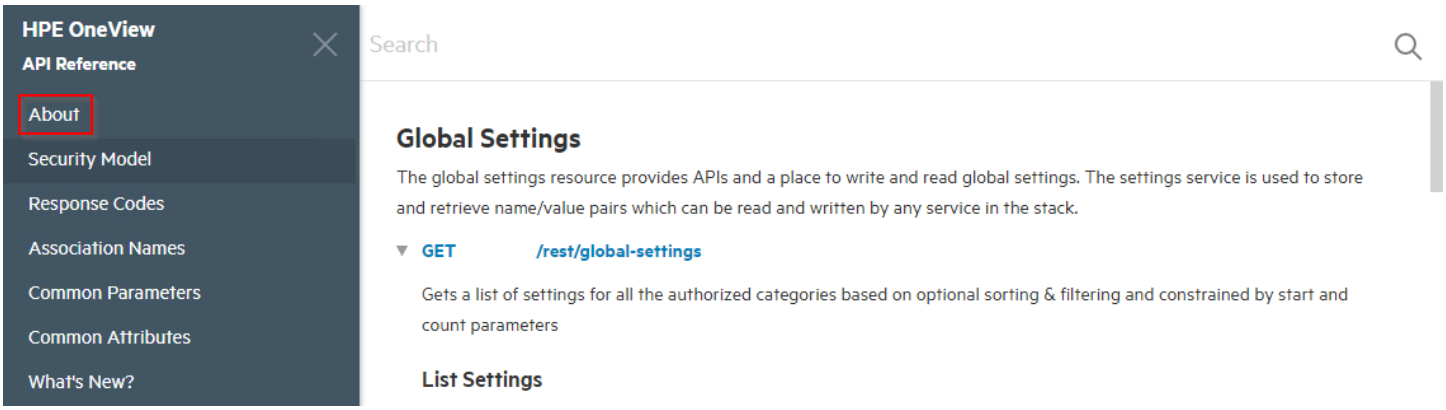
We discussed previously about how important it is to provide an API with software so that your ecosystem can grow and other software entities can integrate with it. But what happens when another release of the software comes up? New capability may be added, older behavior modified, which is normal, but it makes it more difficult to allow the software to evolve without disrupting the growing ecosystem. This is where backward compatibility becomes critical. We, as an API provider, need to guarantee that existing software integrations are not going to break when a new release is introduced.

In order to make this happen, API providers usually require integrators to specify the version of the API that they expect. In most REST APIs, this is done with a HTTP Header. And the API provider guarantees that older versions of the API remain unchanged. Of course, if an integrator wants to benefit from the newest functionality, it will have to update their software to use the newest API, but the older version, running against the older API, should continue to work without modification.

It is the API provider's decision to decide how many older versions of the API are supported at any given release and deprecate older ones if necessary.

This information is found in the *HPE OneView API Reference* documentation.

From the main menu, select **About**



Scroll down to **Supported REST API versions**

The HPE OneView API has been around for a few years already and the table below shows the versions of the API supported for each release:

Supported REST API versions

This release of HPE OneView supports the REST API version(s) listed in the table below.

A copy of the *API Reference* is available online at www.hpe.com/info/oneview/docs. See also the *REST API Scripting Help* for the current version that is included with the online help for this release.

HPE OneView release	REST API version
1.20	120
2.00	200
2.00.06	201
3.00	300
3.10	500
4.00	600
4.10	800
4.20	1000
5.00	1200

You can see that `1200` is the API version for OneView 5.0.

The second table lists the API versions that are no longer supported in our release version (5.0)

Note: The following API versions are *no longer supported* in this release

HPE OneView release	REST API version
1.0, 1.01	3
1.05	4
1.10	101

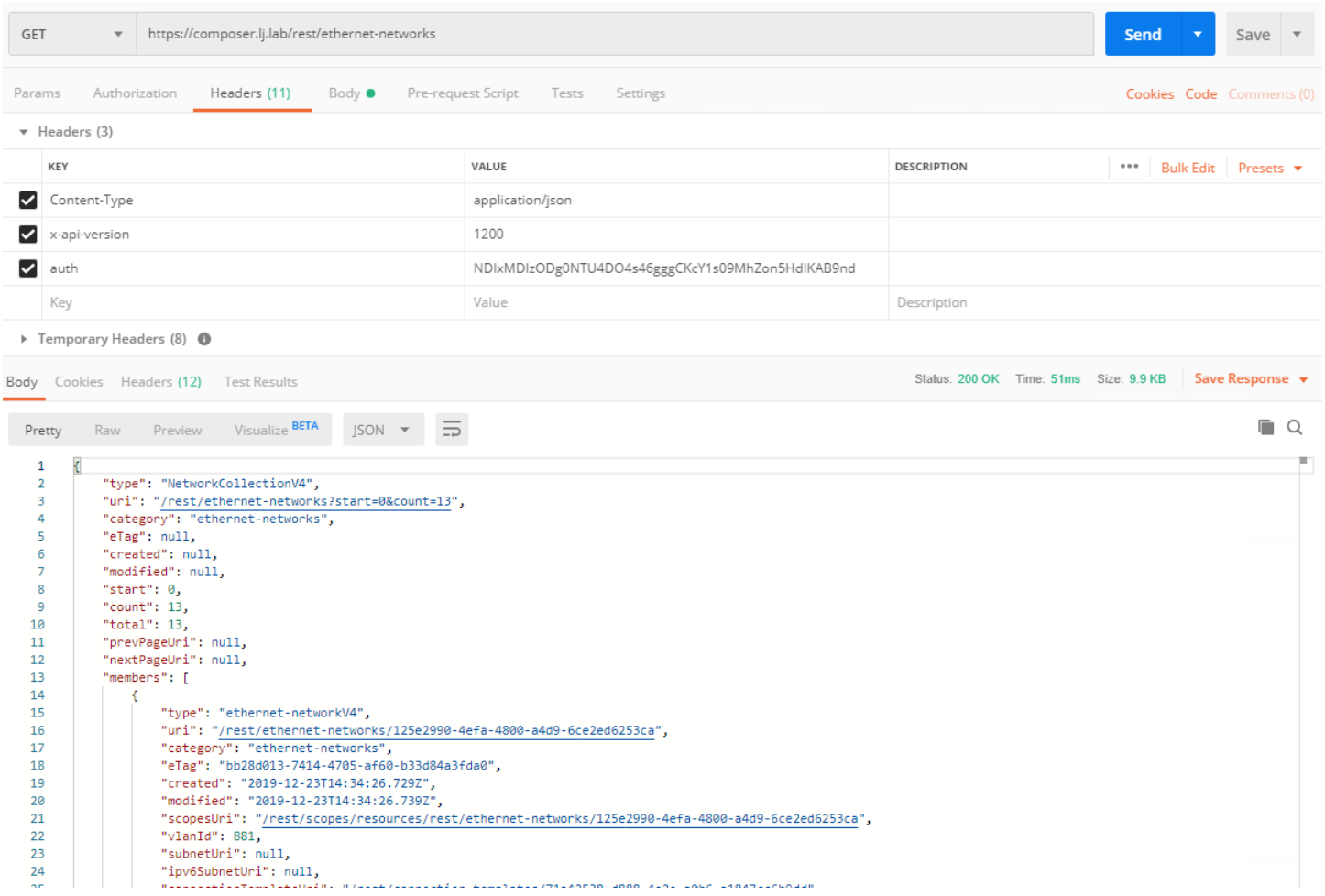
Any REST call using these API versions usually works but will not be supported.

In order to illustrate the impact of using a different API version, let us pick `GET /rest/ ethernet-networks` which returns the current Ethernet network present in the HPE OneView appliance.

So, use Postman with the following parameters:

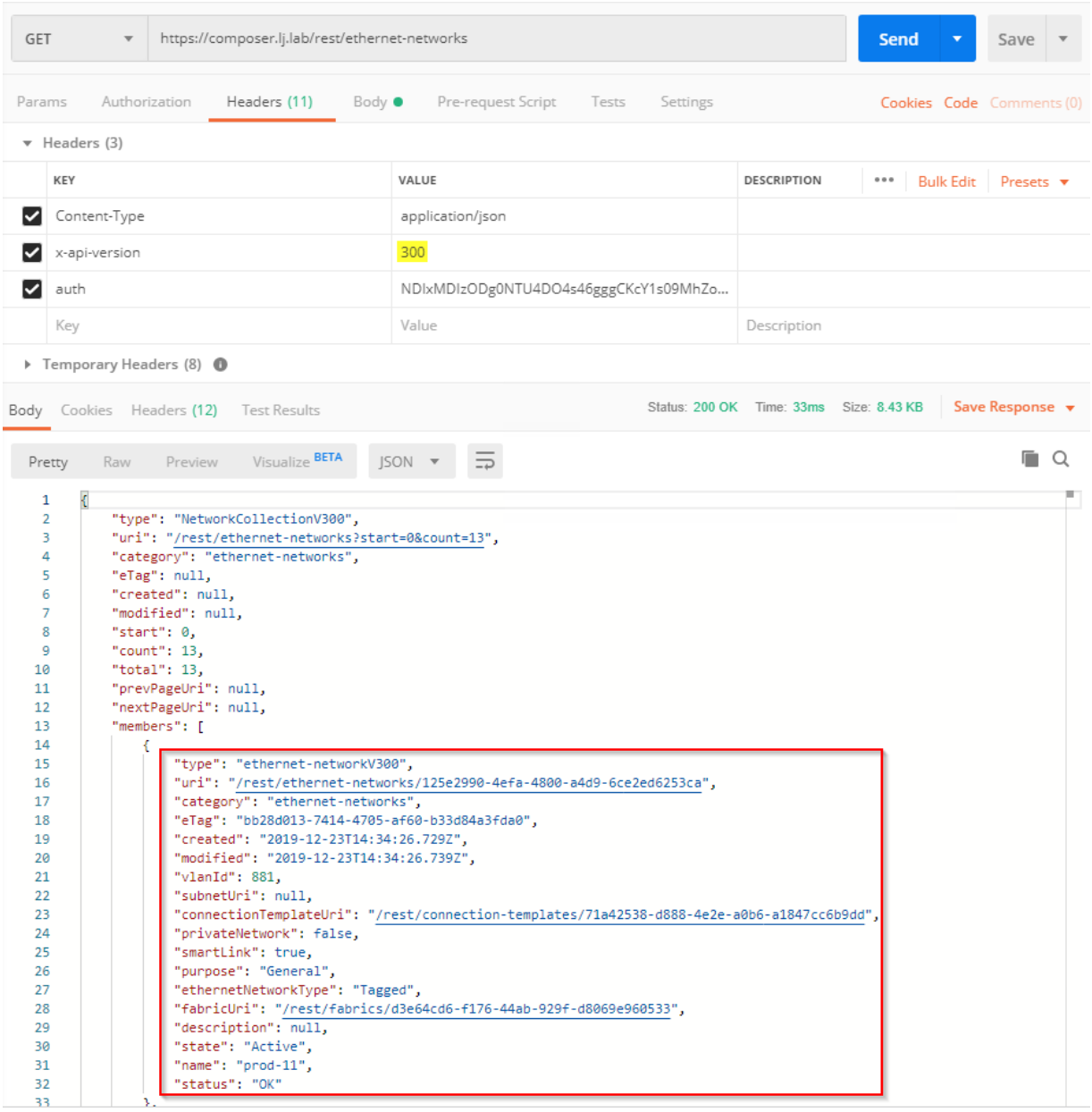
Field	Value
URL	`https://composer.lj.lab/rest/ethernet-networks`
Verb	`GET`
Header	`Auth : your sesionID` `X-API-Version:1200`

The body response is the following:



Then place the same method (you can do a duplicate tab using a right-click) but this time using `X-API-Version:300`

We can see that the response in the body is slightly different than previously when using the version 1200:



With version 1200, Scope Based Access Control and Associate an IPv6 subnet with an Ethernet are present They were both introduced in OneView 4.00 (REST API Version 600).

API Version 1200	API Version 300
{ "type": "ethernet-networkV4", "uri": "/rest/ethernet-networks/89ad8bd5-718c-4f5c-be57-33acb49ff8d5", "category": "ethernet-networks", "eTag": "753c9ac6-75aa-4550-8c00-3ad179251c9e", "created": "2019-05-28T09:53:47.219Z", "modified": "2019-05-28T09:54:13.625Z", "scopesUri": "/rest/scopes/resources/rest/ethernet-networks/89ad8bd5-718c-4f5c-be57-33acb49ff8d5", "vlanId": 5, "subnetUri": "/rest/id-pools/ipv4/subnets/1d2213d7-c487-4166-93fd-e2d2c523f6ec", "ipv6SubnetUri": null, "connectionTemplateUri": "/rest/connection-templates/35cbd8ca-b73e-45b5-8858-77fd975aabac", "privateNetwork": false, "smartLink": false, "purpose": "Management", "ethernetNetworkType": "Tagged", "fabricUri": "/rest/fabrics/d3e64cd6-f176-44ab-929f-d8069e960533", "description": null, "state": "Active", "name": "Management", "status": "OK" },	{ "type": "ethernet-networkV300", "uri": "/rest/ethernet-networks/89ad8bd5-718c-4f5c-be57-33acb49ff8d5", "category": "ethernet-networks", "eTag": "753c9ac6-75aa-4550-8c00-3ad179251c9e", "created": "2019-05-28T09:53:47.219Z", "modified": "2019-05-28T09:54:13.625Z", "vlanId": 5, "subnetUri": "/rest/id-pools/ipv4/subnets/1d2213d7-c487-4166-93fd-e2d2c523f6ec", "connectionTemplateUri": "/rest/connection-templates/35cbd8ca-b73e-45b5-8858-77fd975aabac", "privateNetwork": false, "smartLink": false, "purpose": "Management", "ethernetNetworkType": "Tagged", "fabricUri": "/rest/fabrics/d3e64cd6-f176-44ab-929f-d8069e960533", "description": null, "state": "Active", "name": "Management", "status": "OK" },

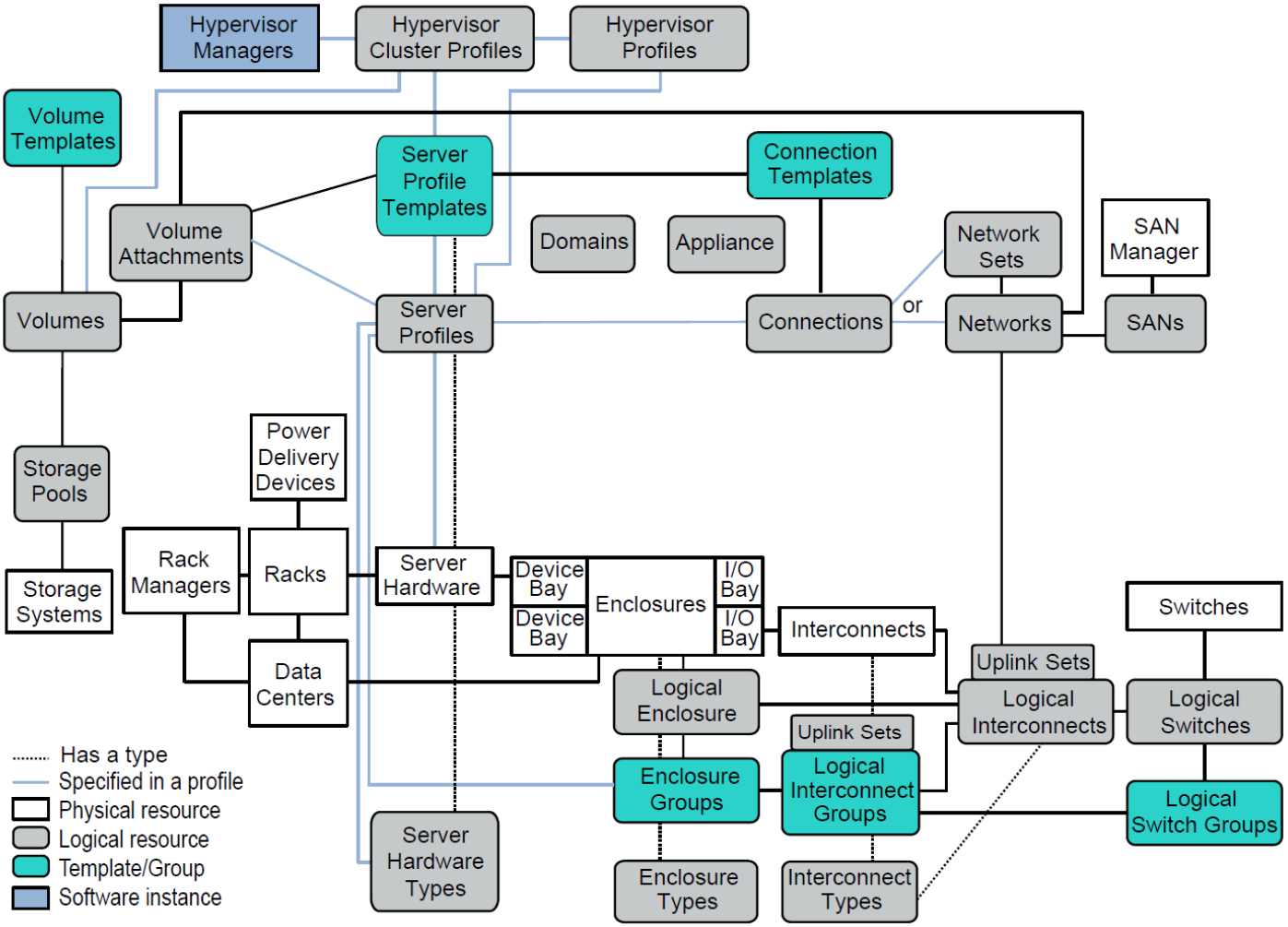
So basically, any version below 600 will never provide/support any scopes and IPv6 associated subnet information.

Lab 5: API Resource model

In previous labs, we saw some of the concept on the versioning, the authentication and the discovery of the infrastructure, via a REST API approach.

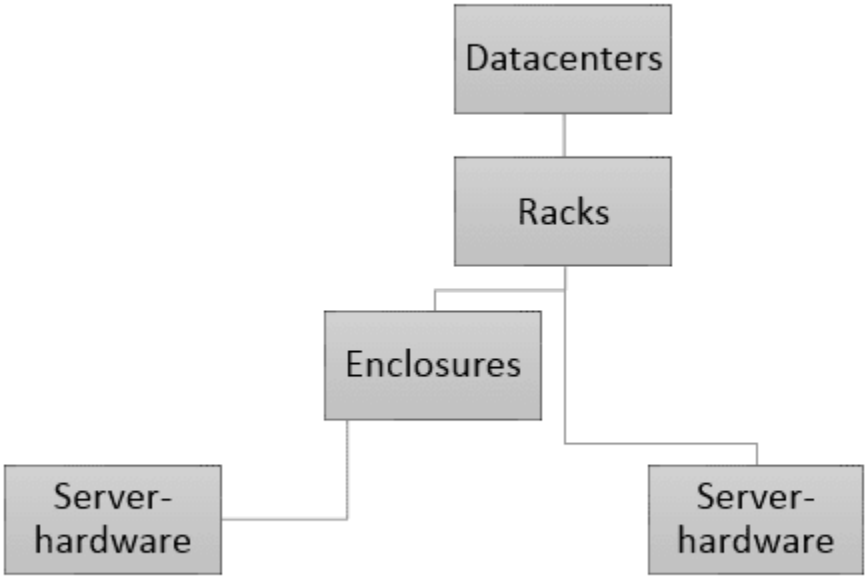
The next important subject that we need to understand is the API resource model.

HPE OneView provides a fully connected resource model with associations or more precisely relationships between the HPE OneView resources. These associations are fundamental during script developments. The following figure summarizes some of the most frequently used resources and shows the relationships between them.



So, let us discover the API resource model, in a top down approach through the objects available.

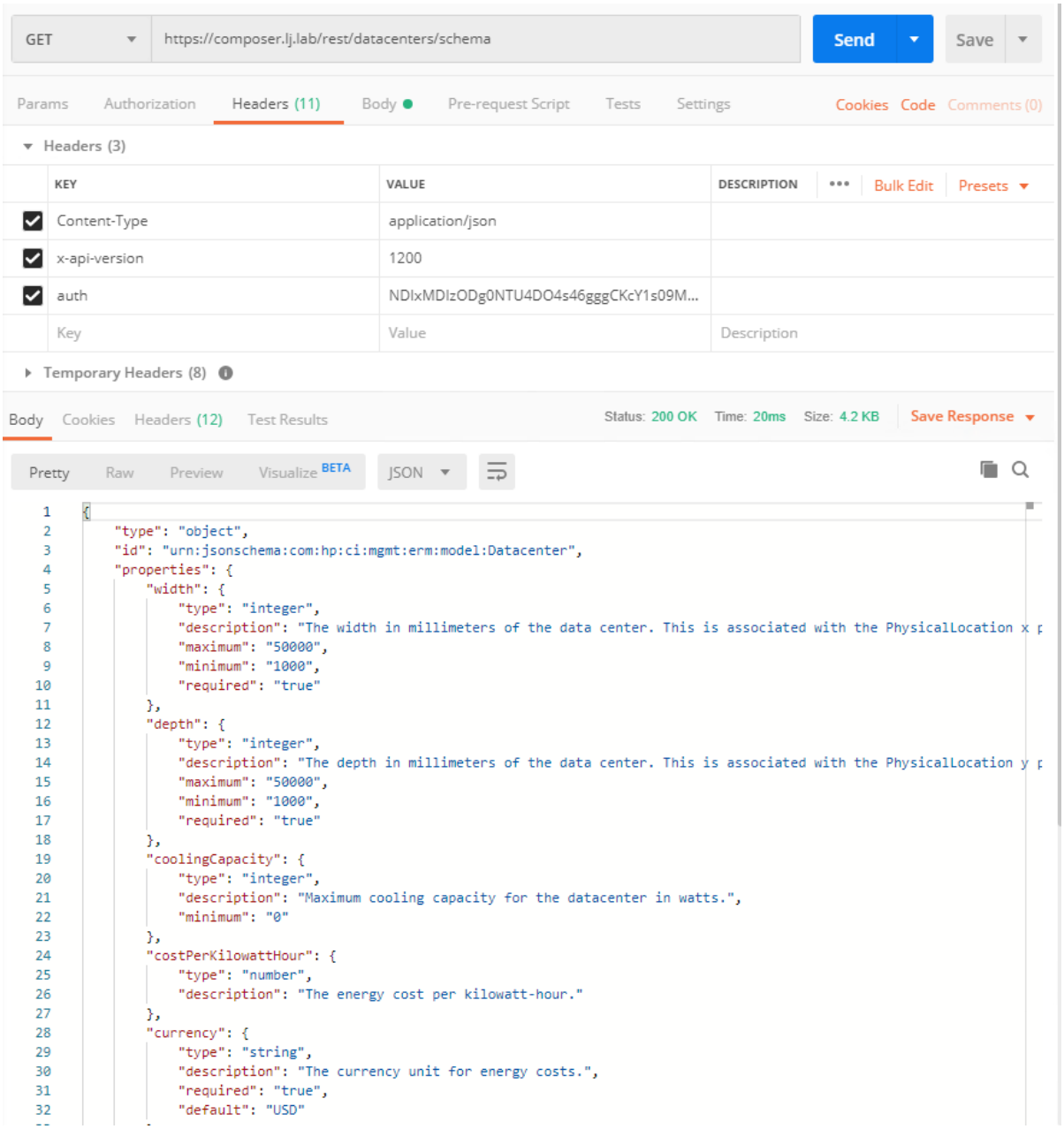
The following figure illustrates the objects we will examine during this process:



From this representation, we know that the topmost object is a Datacenter, and we will use our REST client to browse for available Datacenters in our configuration using `/rest/datacenters`.

Before we do so, how do we know what are the properties that make up a Datacenter object?
There is a simple method to find out about this. This technique is to add a `/schema` at the end of the URL, for example `/rest/datacenters/schema`, to retrieve the schema for datacenter.

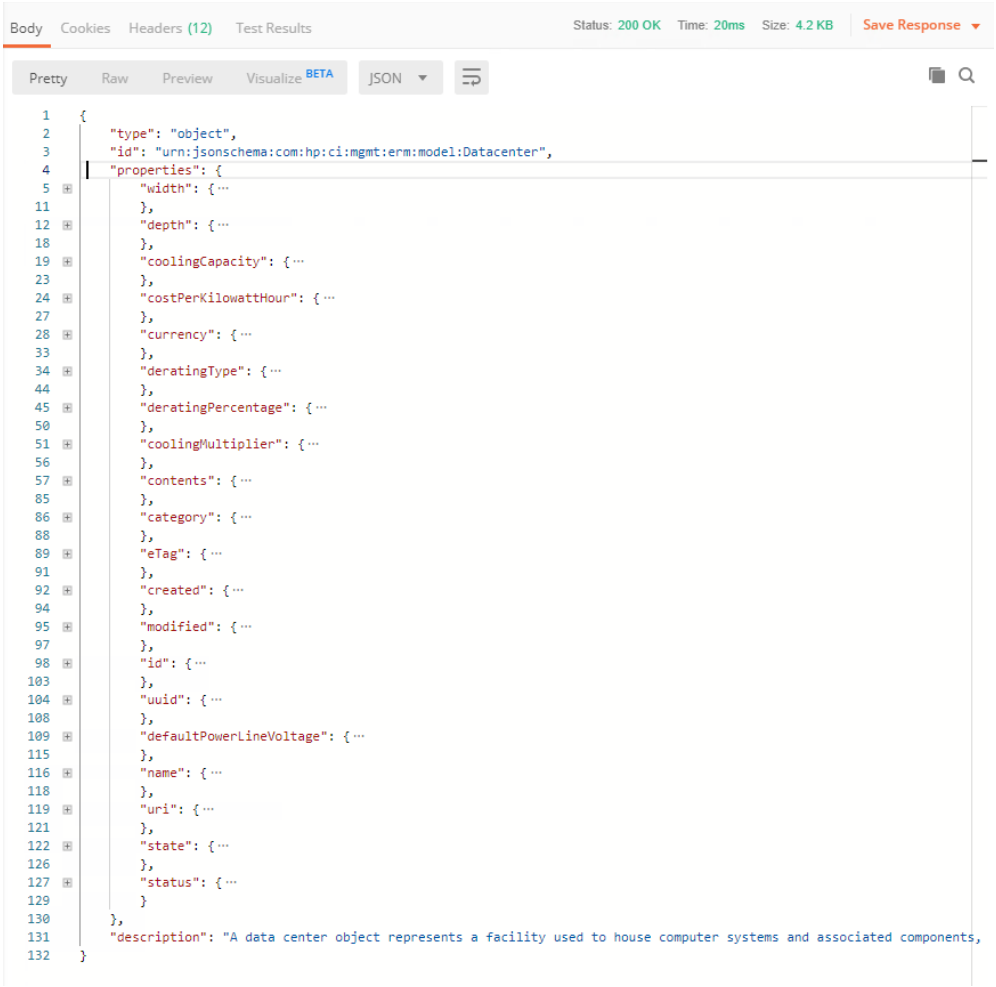
Let us try `/rest/datacenters/schema` with our Postman utility.



If you find the result difficult to read, you can collapse and expand the properties of an object and understand what is available.

Note: To fold all regions, you can use **Alt** + **0**

For example, from the capture below, we can see that a Datacenter has a dozen properties such as *width*, *depth*, or *status*.




```
{
    "type": "object",
    "id": "urn:jsonschema:com:hp:ci:mgmt:erm:model:Datacenter",
    "properties": {
        "width": {…
        },
        "depth": {…
        },
        "coolingCapacity": {…
        },
        "costPerKilowattHour": {…
        },
        "currency": {…
        },
        "deratingType": {…
        },
        "deratingPercentage": {…
        },
        "coolingMultiplier": {…
        },
        "contents": {…
        },
        "category": {…
        },
        "eTag": {…
        },
        "created": {…
        },
        "modified": {…
        },
        "id": {…
        },
        "uuid": {…
        },
        "defaultPowerLineVoltage": {…
        },
        "name": {…
        },
        "uri": {…
        },
        "state": {…
        },
        "status": {…
        }
    },
    "description": "A data center object represents a facility used to house computer systems and associated components,
}
```

We can continue this exploration and drill down on **width**, which has a *description*, a *type* (integer), and a *minimum* and a *maximum* value.

```
{
    "type": "object",
    "id": "urn:jsonschema:com:hp:ci:mgmt:erm:model:Datacenter",
    "properties": {
        "width": {
            "type": "integer",
            "description": "The width in millimeters of the data center. This is associated with the PhysicalLocation x
                position.",
            "maximum": "50000",
            "minimum": "1000",
            "required": "true"
        },
```

We also can find a **required** flag, which is important, as required properties are mandatory when creating new objects with the `POST` method. Here *width* is mandatory when you create a datacenter object.

We can find other types of object. For example, expand the **currency** property and you will see that the type is *string*.

```
27      "currency": {
28          "type": "string",
29          "description": "The currency unit for energy costs.",
30          "required": "true",
31          "default": "USD"
32      },
33      "deratingType": {...},
```

Integer and *string* are simple types, but if you continue to scroll down the list of properties, you will see the **contents** property, which is an *array* of items, of type *object*. An item is constructed from several properties such as *rotation*, *x*, *y* and a *resourceUri*. We also call this a *collection*.

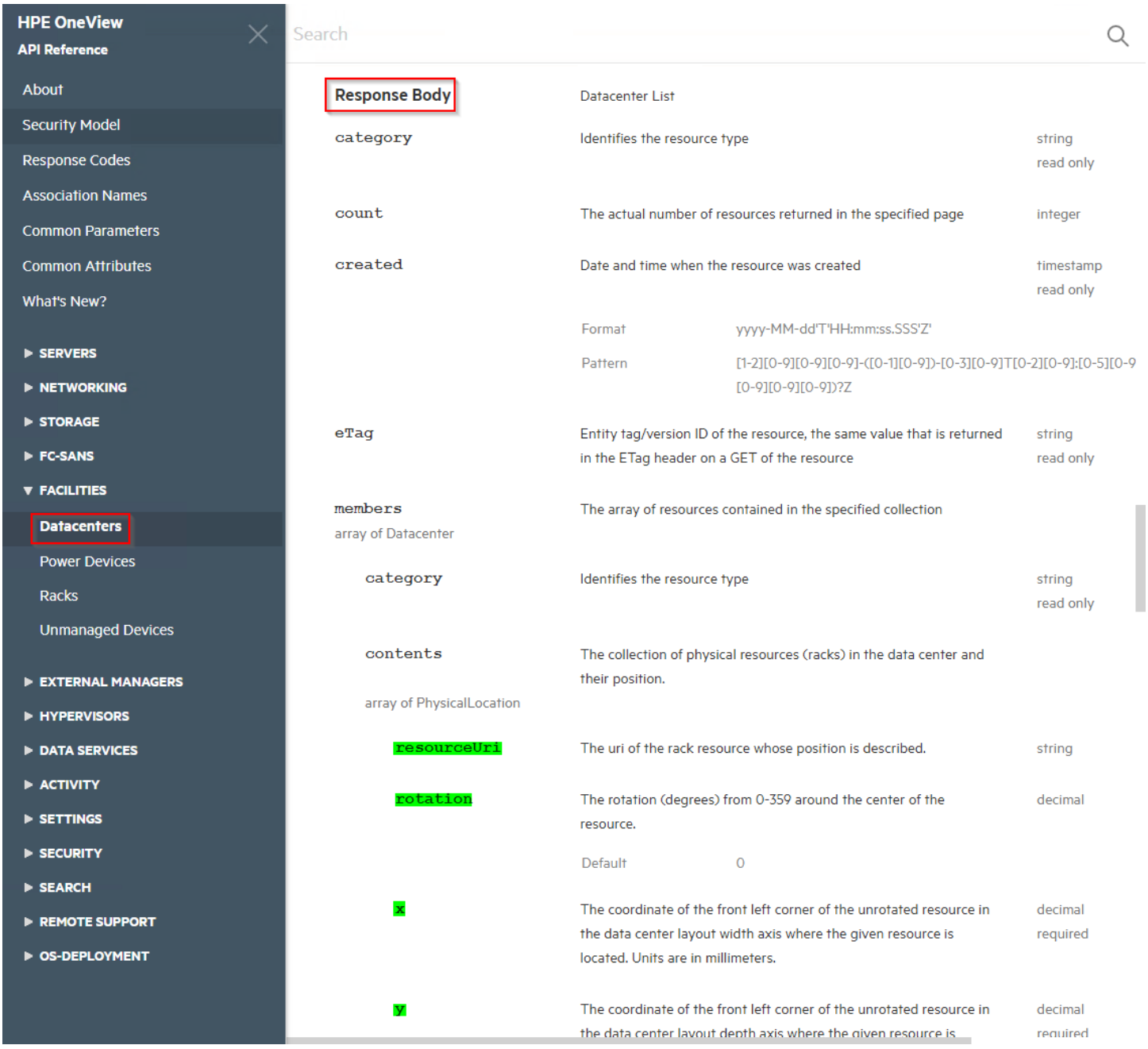
```
50      "coolingMultiplier": {...},
56      "contents": {
57          "type": "array",
58          "items": {
59              "type": "object",
60              "properties": {
61                  "x": {
62                      "type": "number",
63                      "description": "The coordinate of the front left corner of the unrotated
                            resource in the data center layout width axis where the given resource is
                            located. Units are in millimeters.",
64                      "required": "true"
65                  },
66                  "y": {
67                      "type": "number",
68                      "description": "The coordinate of the front left corner of the unrotated
                            resource in the data center layout depth axis where the given resource is
                            located. Units are in millimeters.",
69                      "required": "true"
70                  },
71                  "rotation": {
72                      "type": "number",
73                      "description": "The rotation (degrees) from 0-359 around the center of the
                            resource.",
74                      "default": "0"
75                  },
76                  "resourceUri": {
77                      "type": "string",
78                      "description": "The uri of the rack resource whose position is described. "
79                  }
80              }
81          },
82          "description": "The collection of physical resources (racks) in the data center and their
                position."
83      },
84      "category": {...},
```

Note: In JSON, *integer* are integral numeric values (3, -24 ...) while *number* can be an integer or a floating-point value (3, -24, 1.5, -3.3333 ...)

We understand from this exploration of the datacenter schema that a datacenter is composed of collection of zero or more racks with their *resourceUri*.

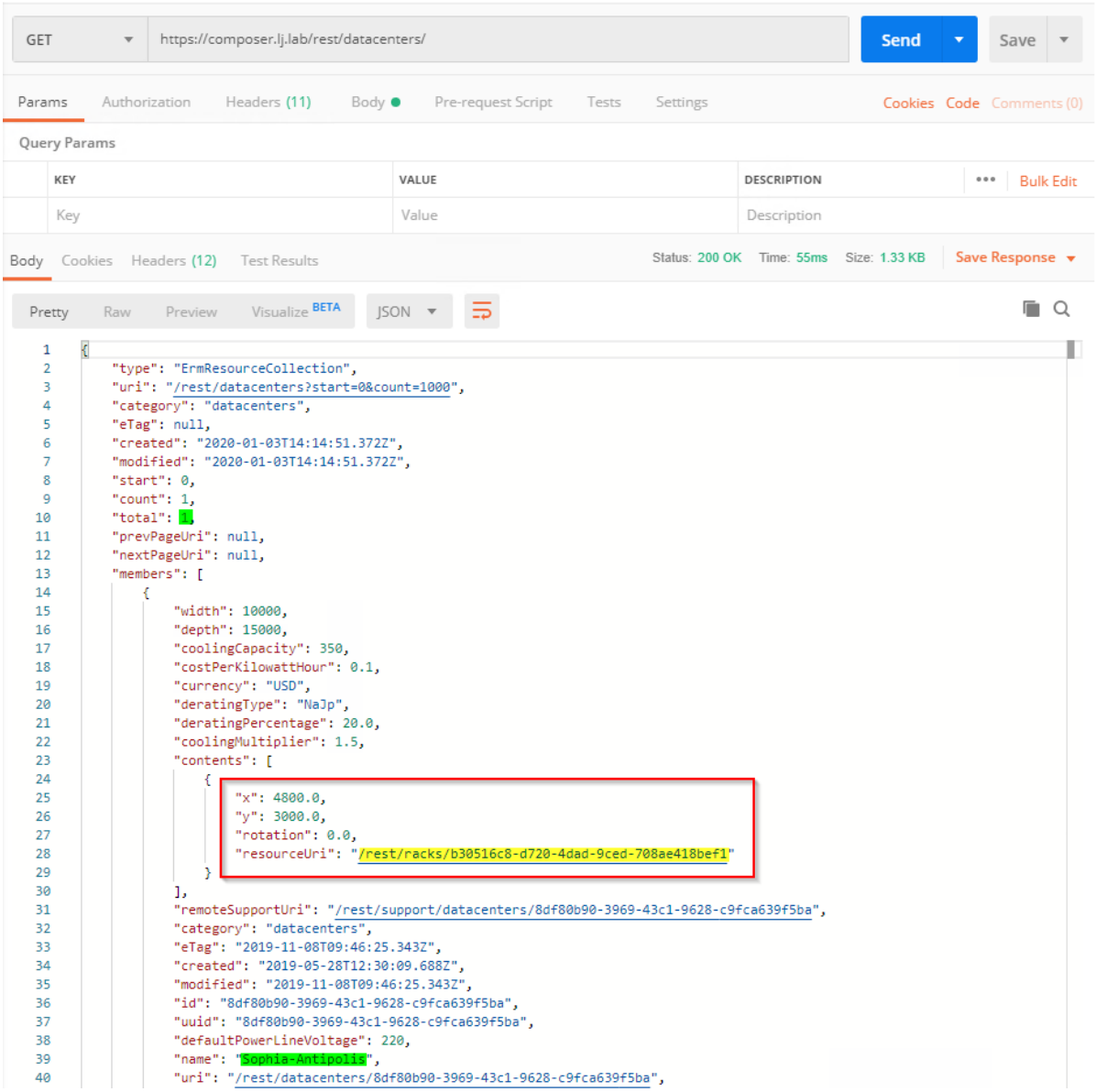
Read The Fantastic Manual, remember? In the OneView *API Reference* document the same object property information that we discover here is all listed in the *Response body* section.

If you do a search for **GET /rest/datacenters** and scroll down to **Response Body**, you will find the same properties and descriptions of each object available in the response body, useful if you need to find a specific property for your script developments:



So now, we know enough to start our first inventory and explore the topmost object instances found in our environment, the datacenters.

Let us remove the `/schema` from the Postman URL, and send the call using this time: https://composer.lj.lab/rest/datacenters

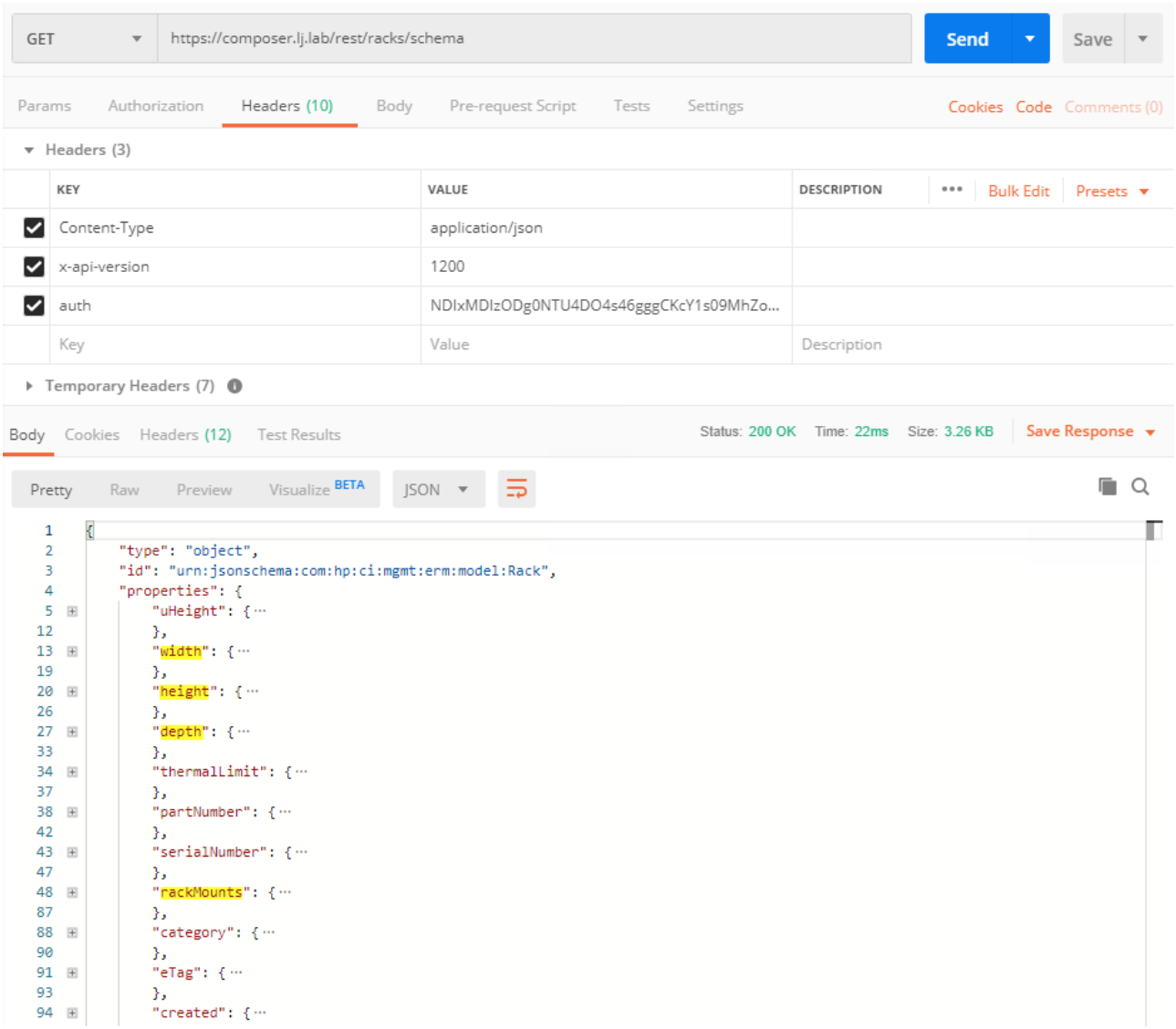


From the response JSON we can see that we get the same object information as with the `/schema` but this time with the actual datacenters' values.

We can discover that there is only one datacenter called "*Sophia-Antipolis*" and in this datacenter, there is a single rack, we have its position (x, y) and URI: `/rest/racks/b30516c8-d720-4dad-9ced-708ae418bef1"`

Note: REST APIs uses URI (Universal Resource Identifier) to identify/address resources

What is a Rack?
We can now drill down to the content of rack `b30516c8-d720-4dad-9ced-708ae418bef1`
But before we start exploring the content of our rack, let us look at the schema for a rack by using the `/rest/racks/schema` URL using our REST client tool.

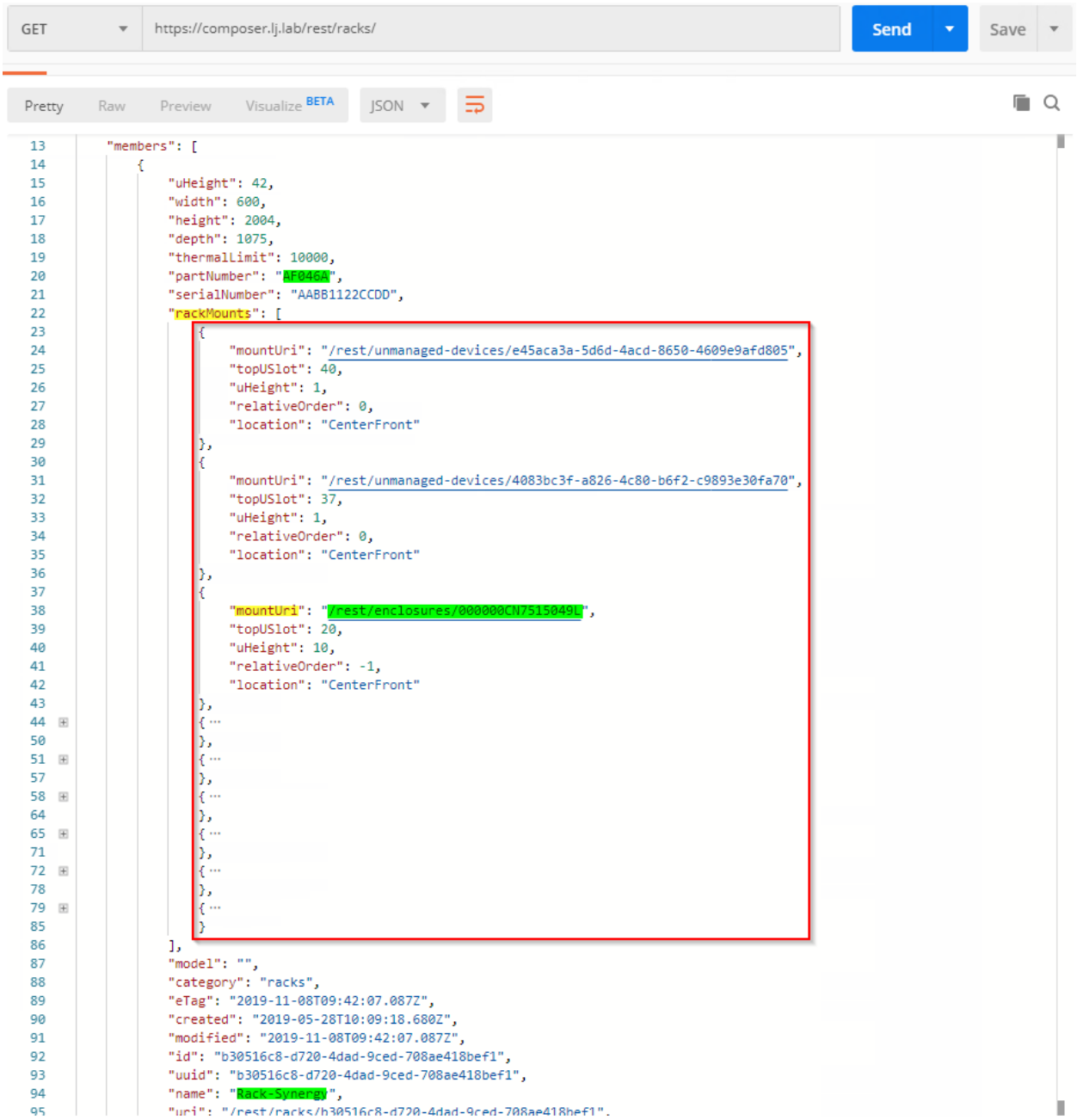


We can discover the properties of a rack, such as *width*, *height*, *depth*, and a collection of *rackMounts*, which represent zero or more items contained in the rack (*enclosure*, *server*-hardware, *power delivery device*)

We can expand the *rackMounts* property to understand what composes a rack. We discover that it is made of a collection of items (*array*), and that each item has a *mountUri* to access it.

```
42        },
43        "serialNumber": {···
47        },
48        "rackMounts": {
49            "type": "array",
50            "items": {
51                "type": "object",
52                "id": "urn:jsonschema:com:hp:ci:mgmt:erm:model:RackMount",
53                "properties": {
54                    "mountUri": {···
59                    },
60                    "topUSlot": {···
63                    },
64                    "uHeight": {···
67                    },
68                    "relativeOrder": {···
71                    },
72                    "location": {···
83                    }
84                }
85            },
86            "description": "References to the resources contained in the rack."
87        },
```

Let us now do a `GET` request on `/rest/racks`.



From that query, we discover that our rack is named "Rack-*Synergy*", its part number is *AF046A* and in this rack we have several enclosures, servers and unmanaged components (switches). We have, with the *mountUri* property, a direct access to the URI of each component in the rack.

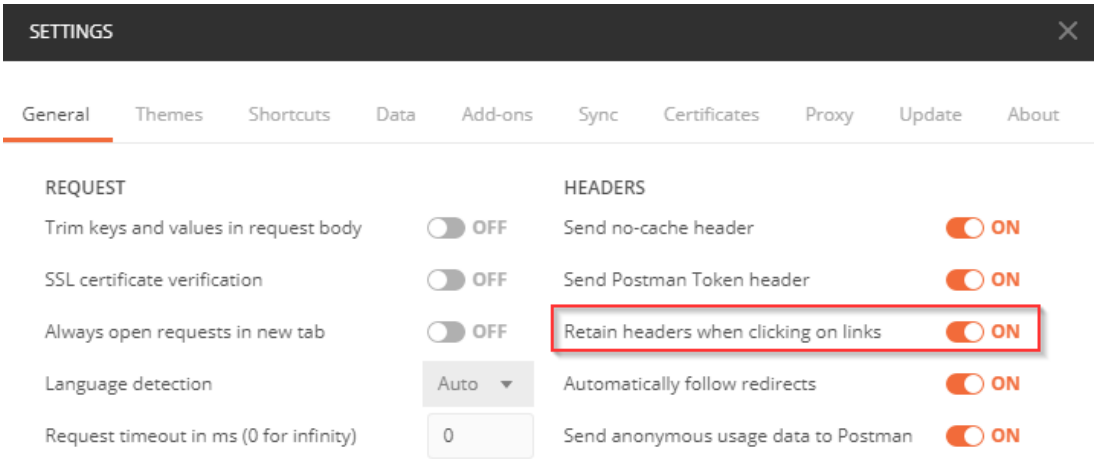
What is an Enclosure?
We can go further down by taking a few minutes to explore an enclosure.

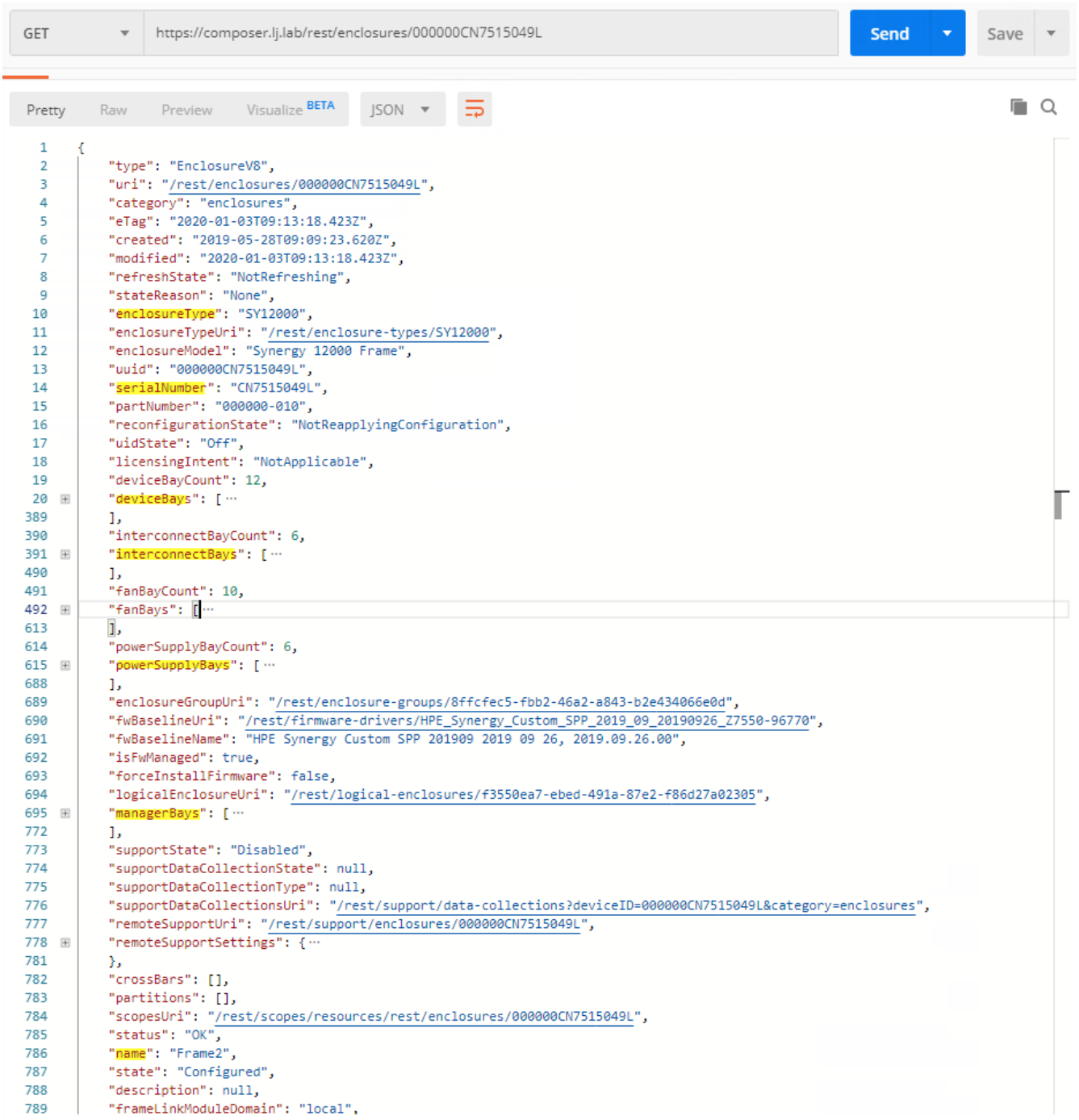
One of the cool Postman features is that you can click on links inside the response body, so click on one of the enclosures *mountUri* link:


```
"members": [
    {
        "uHeight": 36,
        "width": 600,
        "height": 2032,
        "depth": 1075,
        "thermalLimit": 10000,
        "partNumber": "AF046A",
        "serialNumber": "AABB1122CCDD",
        "rackMounts": [
            {
                "mountUri": "/rest/enclosures/000000CN7515049L",
                "topUSlot": 20,
                "uHeight": 10,
                "relativeOrder": -1,
                "location": "CenterFront"
            },
```


This will automatically load a `GET` Request in Postman with the link URL and retains our very important headers content (auth and x-api-version) then click **Send**

Note: If you don't get the header content retained, make sure the option is enabled in Postman settings:

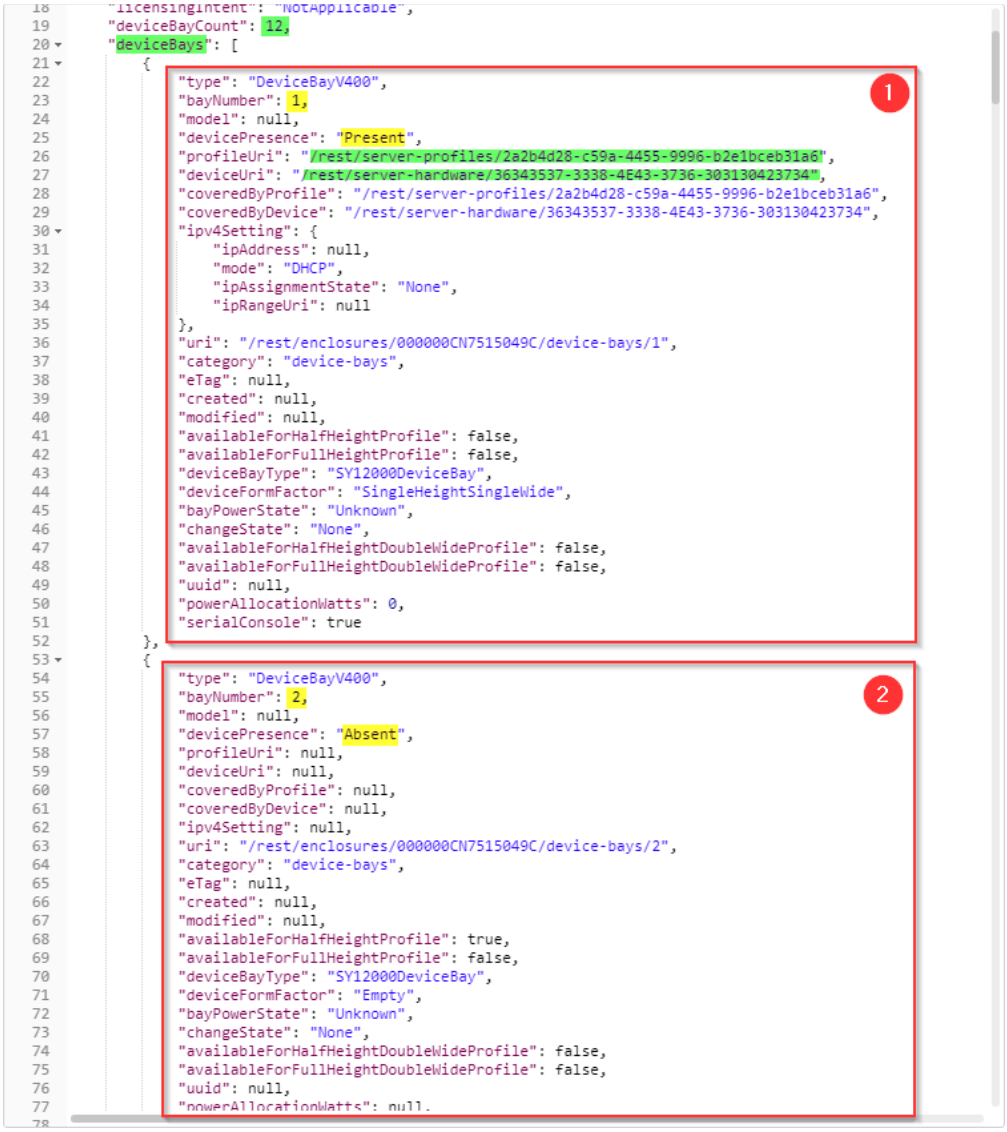




We discover many elements and useful parameter that make up an enclosure, such as name, Serial Number (*serialNumber*), Enclosure Type (*enclosureType*), Composable Infrastructure Appliances (*managerBays*), Interconnect Modules (*interconnectBays*), Power Supplies (*powerSupplyBays*), Compute Modules (*deviceBays*), etc.

We can also see that an enclosure is composed of a *deviceBays* array of items.

We can also discover that it is composed of 12 device bays. Each item in *devicebays* contains the description of a device bay. We see the bay number, the presence of a server hardware, if a profile is assigned, etc.




```
"licensingIntent": "NotApplicable",
"deviceBayCount": 12,
"deviceBays": [
    {
        "type": "DeviceBayV400",
        "bayNumber": 1,
        "model": null,
        "devicePresence": "Present",
        "profileUri": "/rest/server-profiles/2a2b4d28-c59a-4455-9996-b2e1bceb31a6",
        "deviceUri": "/rest/server-hardware/36343537-3338-4E43-3736-303130423734",
        "coveredByProfile": "/rest/server-profiles/2a2b4d28-c59a-4455-9996-b2e1bceb31a6",
        "coveredByDevice": "/rest/server-hardware/36343537-3338-4E43-3736-303130423734",
        "ipv4Setting": {
            "ipAddress": null,
            "mode": "DHCP",
            "ipAssignmentState": "None",
            "ipRangeUri": null
        },
        "uri": "/rest/enclosures/000000CN7515049C/device-bays/1",
        "category": "device-bays",
        "eTag": null,
        "created": null,
        "modified": null,
        "availableForHalfHeightProfile": false,
        "availableForFullHeightProfile": false,
        "deviceBayType": "SY12000DeviceBay",
        "deviceFormFactor": "SingleHeightSingleWide",
        "bayPowerState": "Unknown",
        "changeState": "None",
        "availableForHalfHeightDoubleWideProfile": false,
        "availableForFullHeightDoubleWideProfile": false,
        "uuid": null,
        "powerAllocationWatts": 0,
        "serialConsole": true
    },
    {
        "type": "DeviceBayV400",
        "bayNumber": 2,
        "model": null,
        "devicePresence": "Absent",
        "profileUri": null,
        "deviceUri": null,
        "coveredByProfile": null,
        "coveredByDevice": null,
        "ipv4Setting": null,
        "uri": "/rest/enclosures/000000CN7515049C/device-bays/2",
        "category": "device-bays",
        "eTag": null,
        "created": null,
        "modified": null,
        "availableForHalfHeightProfile": true,
        "availableForFullHeightProfile": false,
        "deviceBayType": "SY12000DeviceBay",
        "deviceFormFactor": "Empty",
        "bayPowerState": "Unknown",
        "changeState": "None",
        "availableForHalfHeightDoubleWideProfile": false,
        "availableForFullHeightDoubleWideProfile": false,
        "uuid": null,
        "powerAllocationWatts": null,
```

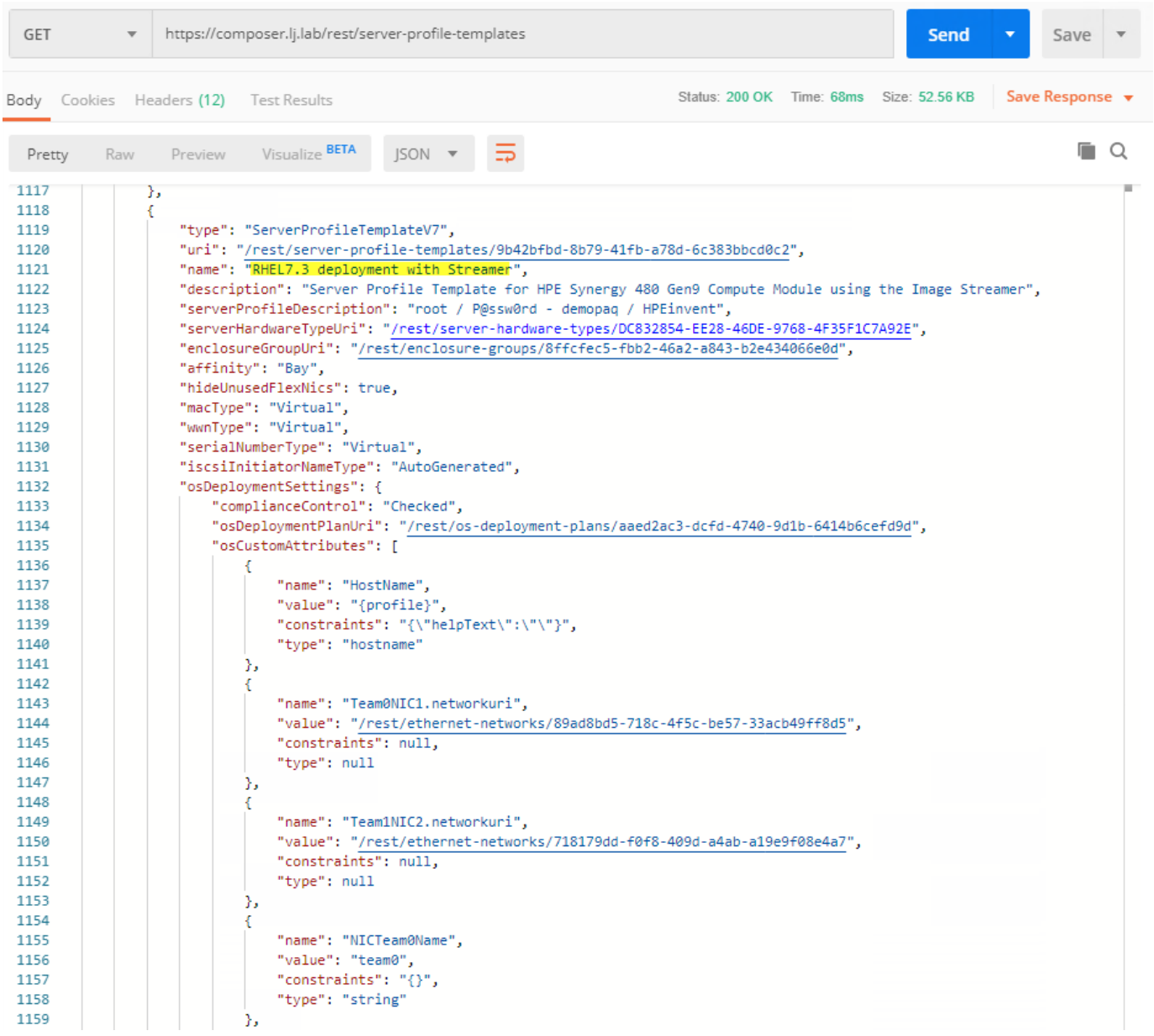

In our case, bay 1 is populated with a Synergy Compute module for which we are given the server-hardware resource, *deviceUri* (`/rest/server-hardware/36343537-3338-4E43-3736-303130423734`) and the profile resource *profileUri* currently assigned to that bay (`/rest/server-profiles/2a2b4d28-c59a-4455-9996-b2e1bceb31a6`). On the other hand, Bay 2 is empty and without any profile assigned.

We now understand that an HPE OneView instance contains a collection of one or more datacenters, which is composed of a collection of zero or more racks, which contains a collection of one or more items such as a server, enclosure, profiles, etc.

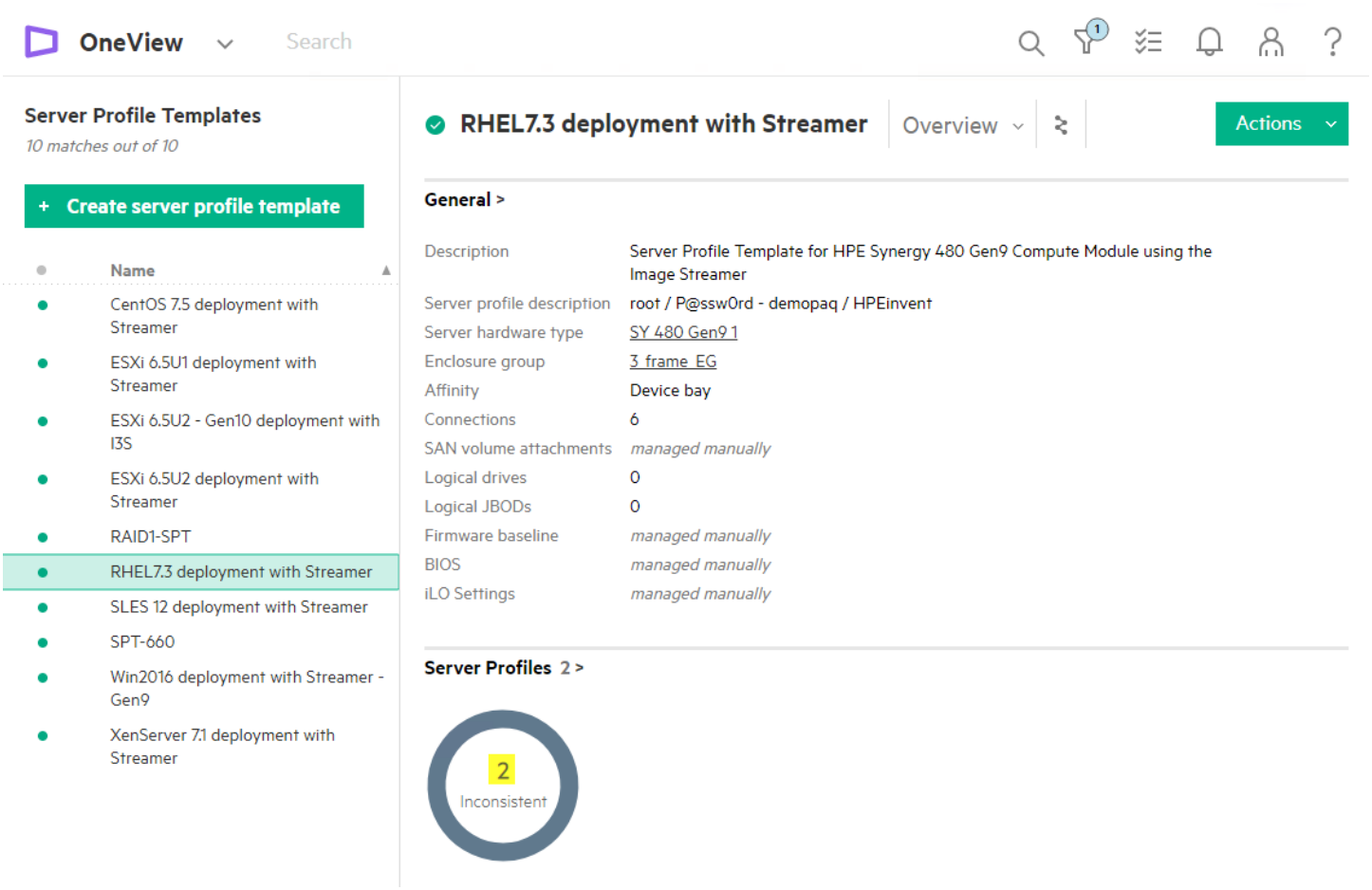
Associations

We just discovered the relationships between the HPE OneView resources, this is an important subject. Very often when retrieving information from the API, it can be difficult to figure out which child resources are in association with the current resource.

If you send a `GET /rest/server-profile-templates`, you will see that the response body does not provide any information about the server profiles that are associated with that Server Profile Template.

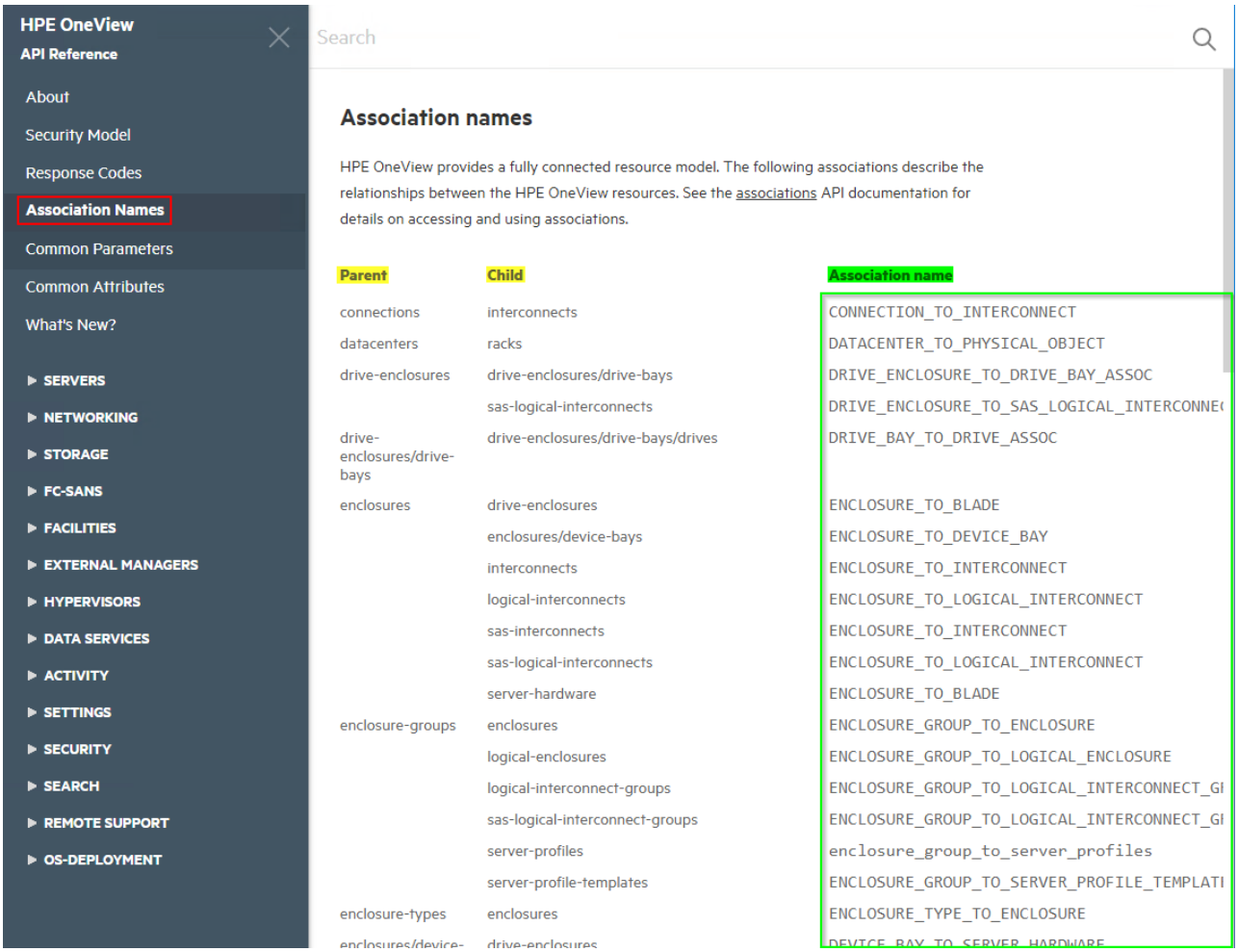


This may seem particularly unexpected when you know that this information is directly available in the Overview pane:

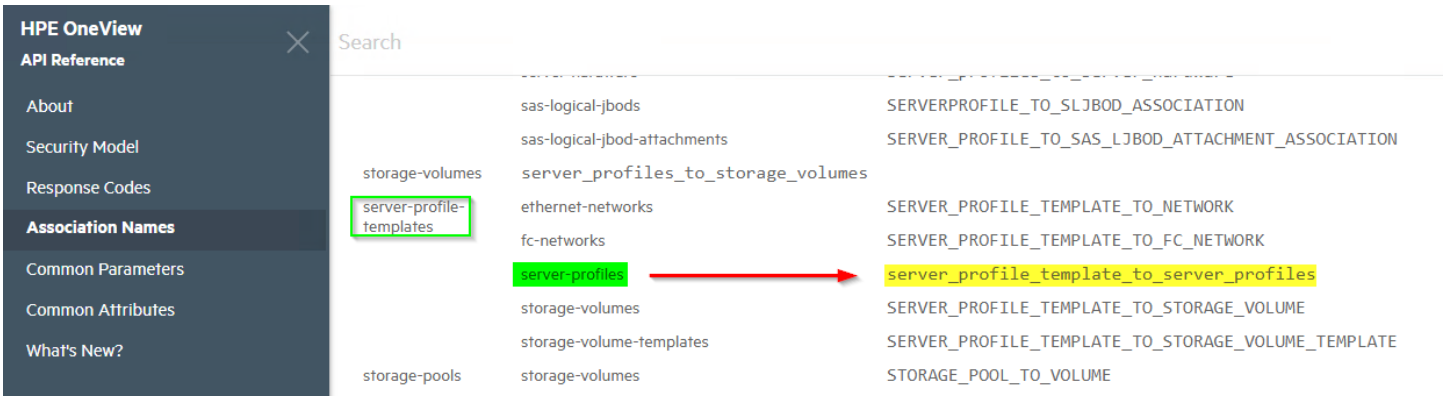


Additionally, enquiring a list of server profiles managed by a server profile template is a quite a basic request and you might struggle a long time to find out this information using a simple GET call on the resource itself...

The good news is that there is an easy way to find that, using an API feature called **Associations names**. OneView provides a fully connected resource model with **child/parent associations** and these associations are described in the API reference documentation:

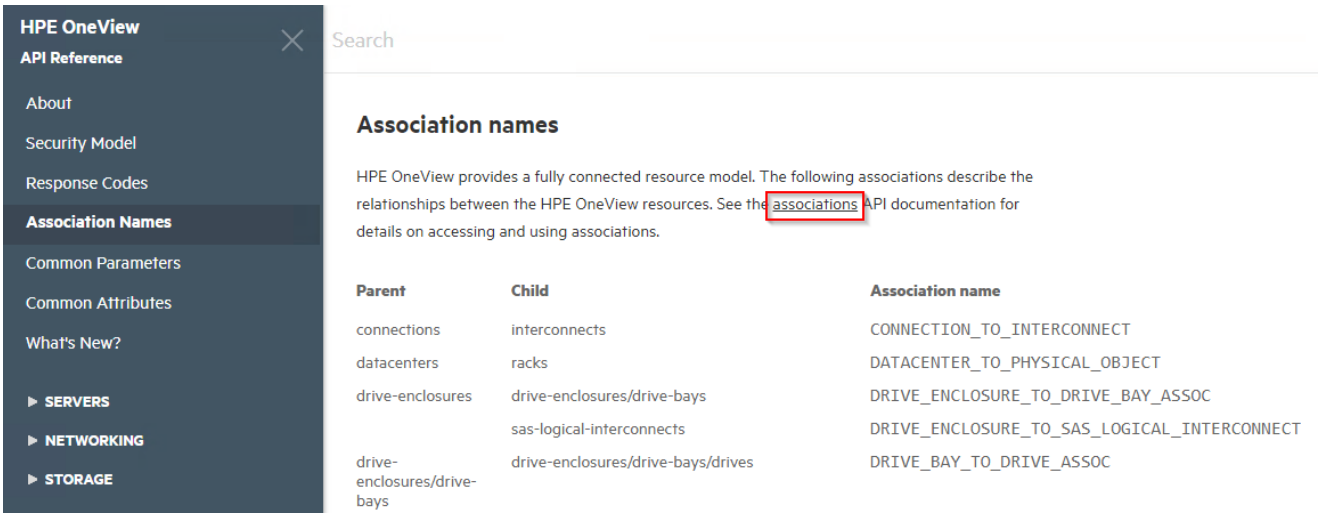


Back to our Server Profile Template issue, if you scroll down to the **Server-Profile-Templates** parent, there is a **Server-Profile** child item with its associated name: *server_profile_template_to_server_profiles*

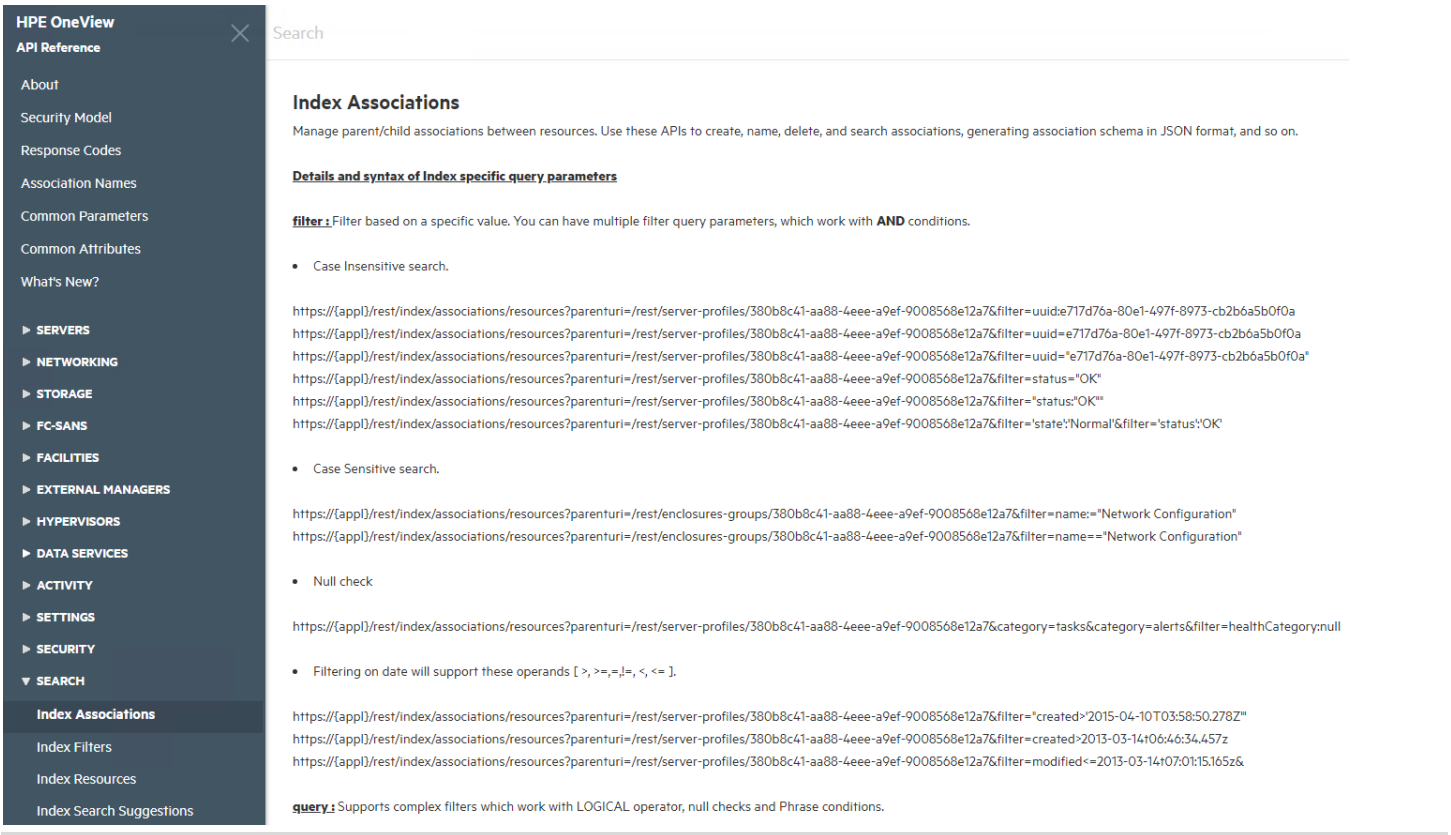


This association name *server_profile_template_to_server_profiles* can be used as a query name.
`name=server_profile_template_to_server_profiles`

Scroll back to the begin of the **Association Names** web page and click on **associations**.



In this page, we can find several examples of API request to create, delete, name and search associations.



The query parameters to get the list of associations that match the requested parameters use the following syntax: `https://{appl}/rest/ index/associations?parenturi={resource URI}&{parameters}`

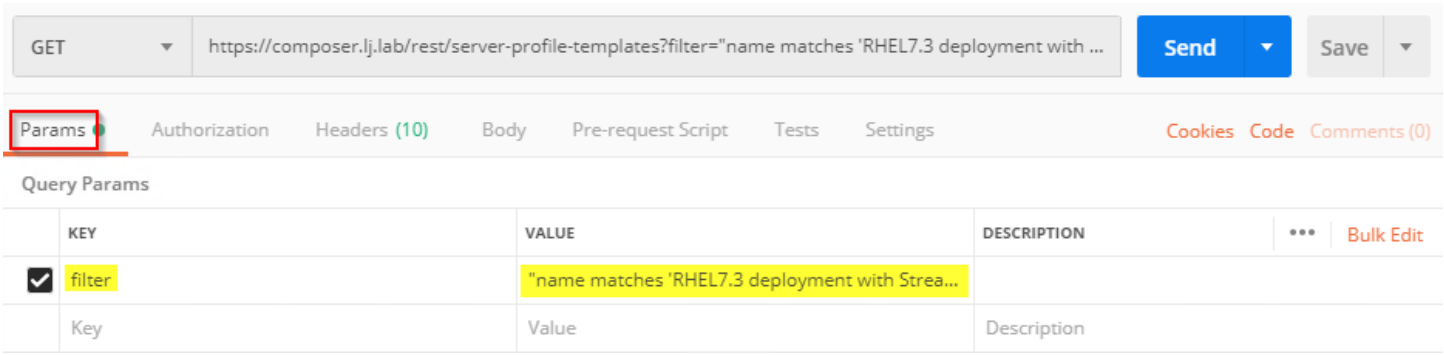
The query parameters can be used with `parenturi`/`childuri`/`name` filtering.

As an example, use Postman with the following parameters to get first the URI of an existing server profile template:

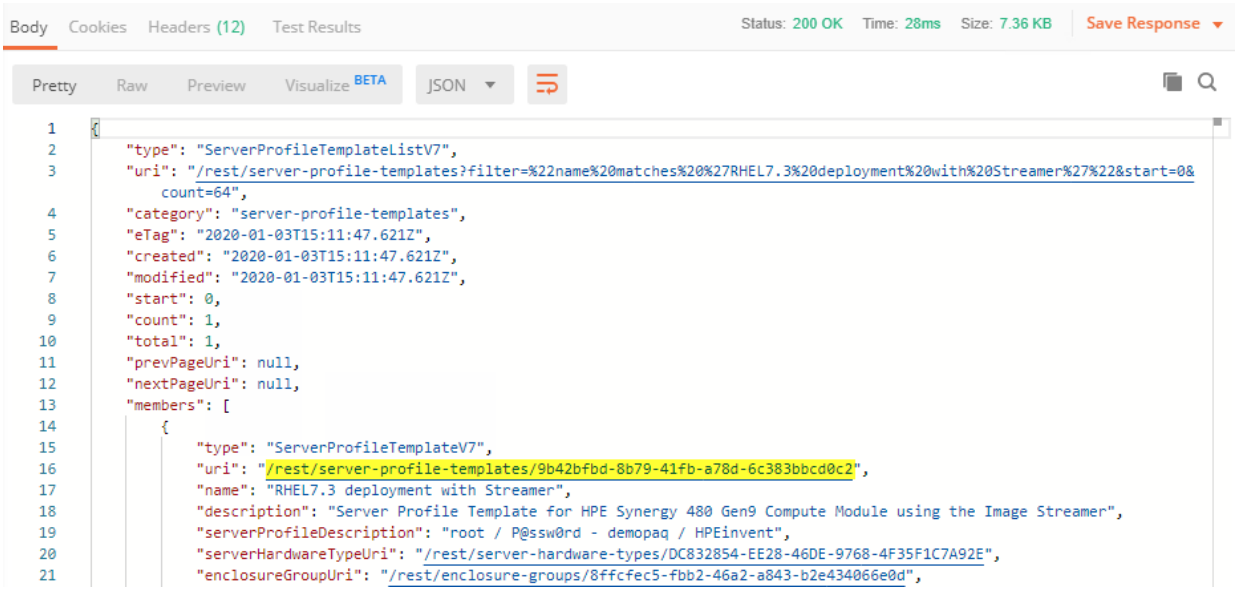
Field	**Value**
URL	`https://composer.lj.lab/rest/server-profile-templates`
Params Key	`filter`
Params Value	`"name matches 'RHEL7.3 deployment with Streamer'"`
Verb	`GET`
Header	`Auth : your sesionID` `X-API-Version:1200`

This request is a little bit different than what we saw previously as we are using here a query parameter to filter the response to only get the Server Profile Template that matches with the name “Template for Prod-ESX servers”.

Notice that using those parameters modify the URL to `/rest/server-profile-templates?filter="name matches 'RHEL7.3 deployment with Streamer'"`



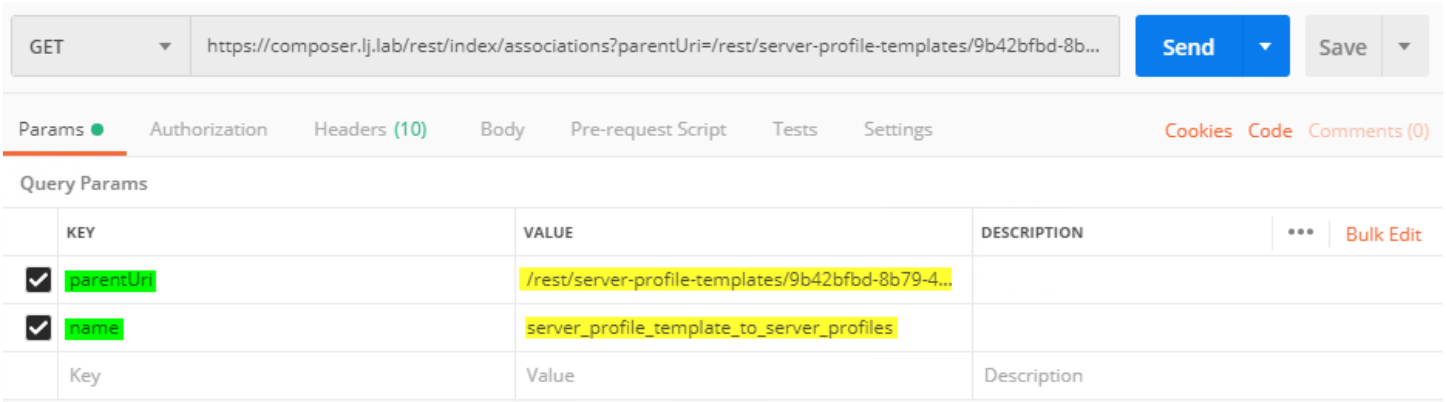
The response is then filtered with only the Server Profile Template we are looking for.



Now record the URI of the Server Profile Template highlighted above as we are going to use it in the next request.

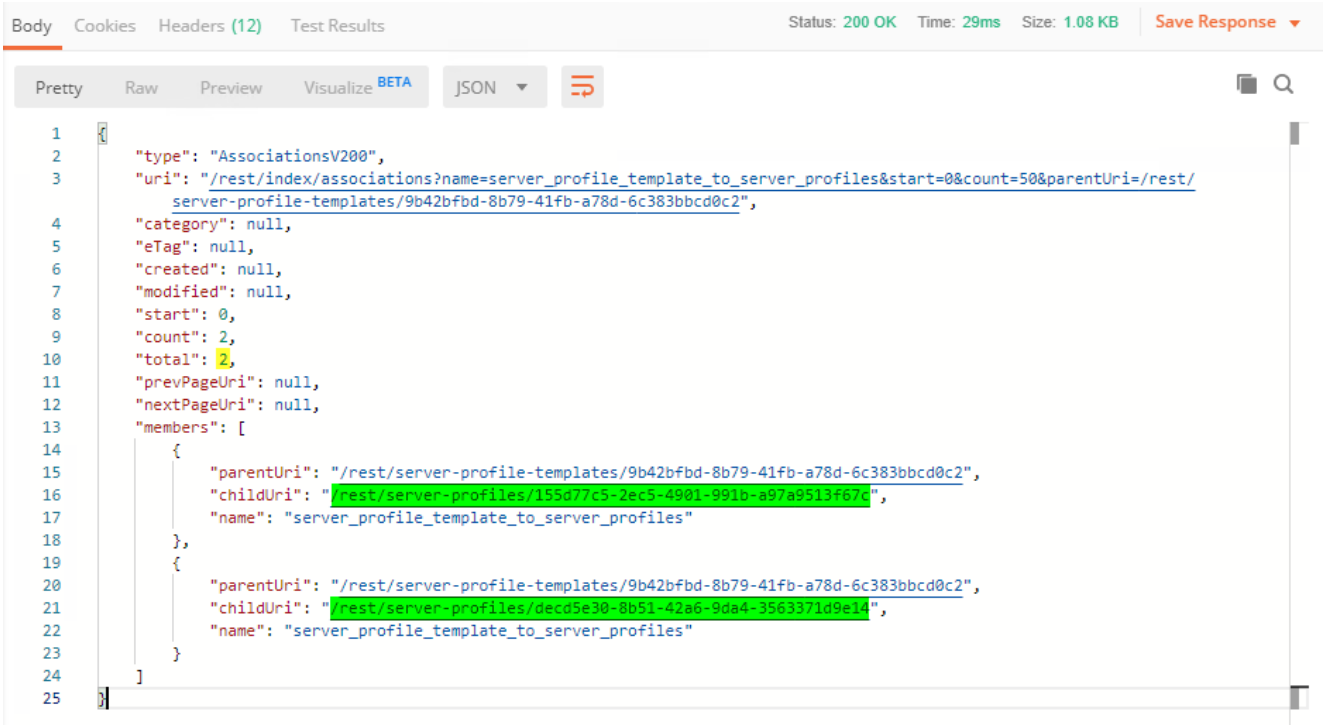
Now let's create a new Postman request to get the association between a Server Profile Template and Server Profiles using the association name `server_profile_template_to_server_profiles` and the parentURI of our Server Profile Template recorded previously:

Field	Value
URL	`https://composer.lj.lab/rest/index/associations`
Parameter key	`parentUri`
Parameter value	`/rest/server-profile-templates/9b42bfbd-8b79-41fb-a78d-6c383bbcd0c2`
Parameter key	`name`
Parameter value	`server_profile_template_to_server_profiles`
Verb	`GET`
Header	`Auth : your sesionID` `X-API-Version:1200`



Notice that using those parameters modify the URL to `https://composer.lj.lab/rest/index/associations?parentUri=/rest/server-profile-templates/9b42bfbd-8b79-41fb-a78d-6c383bbcd0c2&name=server_profile_template_to_server_profiles`

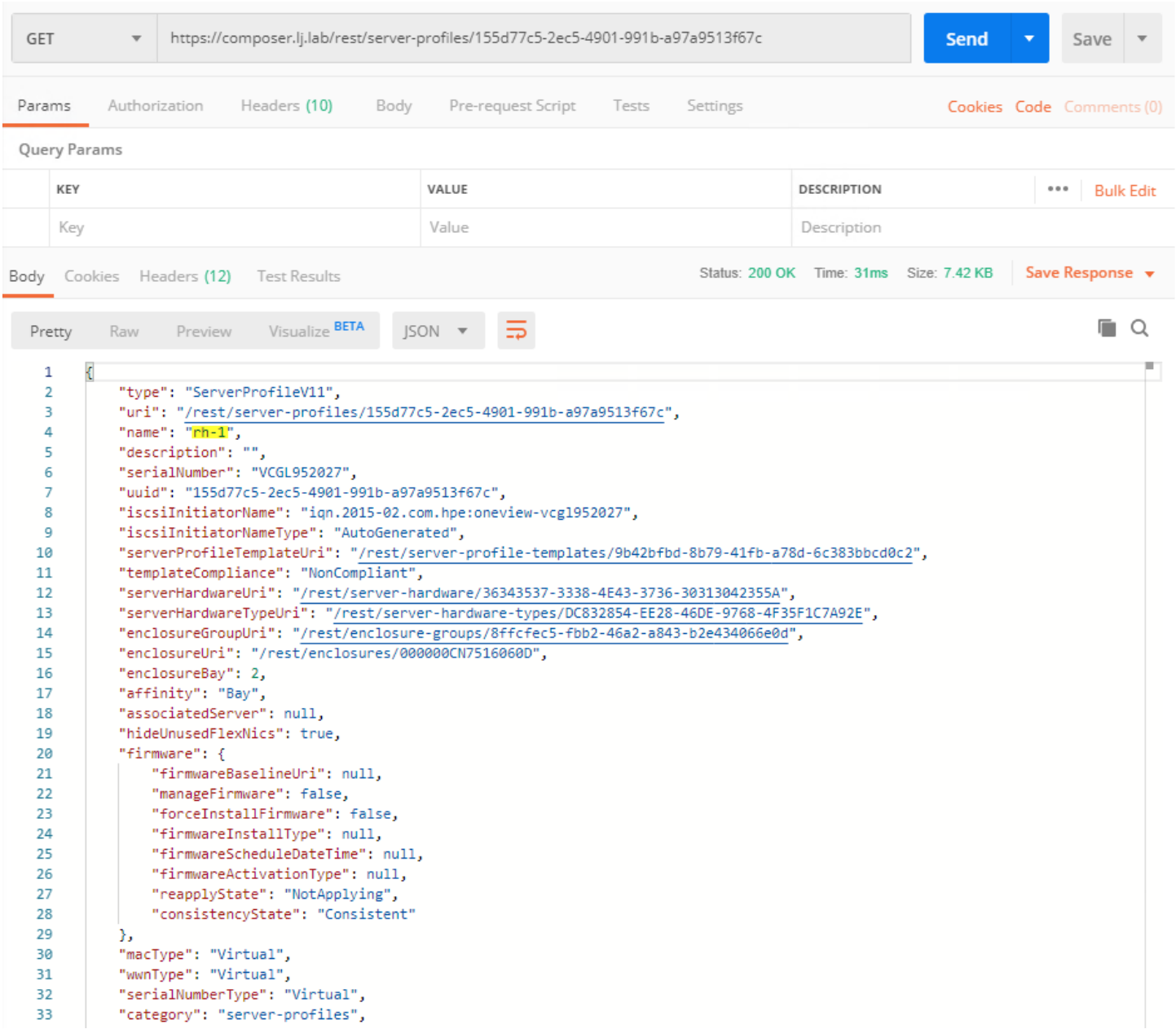
This is what the response body looks like:



The response shows that the Server Profile Template `=/rest/server-profile-templates/9b42bfbd-8b79-41fb-a78d-6c383bbcd0c2` has two Server profiles `/rest/server-profiles/155d77c5-2ec5-4901-991b-a97a9513f67c` and `/rest/server-profiles/decd5e30-8b51-42a6-9da4-3563371d9e14` associated with it.

To access to the server profile information, you can simply click on one of the *childUri* link:

Then click **Send**



This was an example of how to use associations to find which child resources are in association with a parent resource. Other association requests using filtering (created, modified, etc.), conditions (and, or, not) and regular expression can also be used.

This concludes Lab-1 to 5 and the OneView API programing.

You know enough now to run your own invocation. We showed you how API invocation works, how to get authenticated, how to find the information and how to create your own API call.

For fun, you can try to solve the following exercises to validate your skills:

Exercise 1

How can you get using a REST call the number of Ethernet network managed by OneView that have smartlink disabled and with an iSCSI purpose?

Hint: RTFM, everything can be found in the documentation!

Answer: on next page.

Exercise 2

How can you configure a remote SYSLOG server in HPE OneView GUI?

Note: Remote Syslog is different than Audit Logs Forwarding which enables HPE OneView to forward audit logs to remote Security Information and Event Management (SIEM) systems. Remote Syslog provides the ability to receive the logs from supported devices managed by OneView like iLOs, interconnect modules, etc.

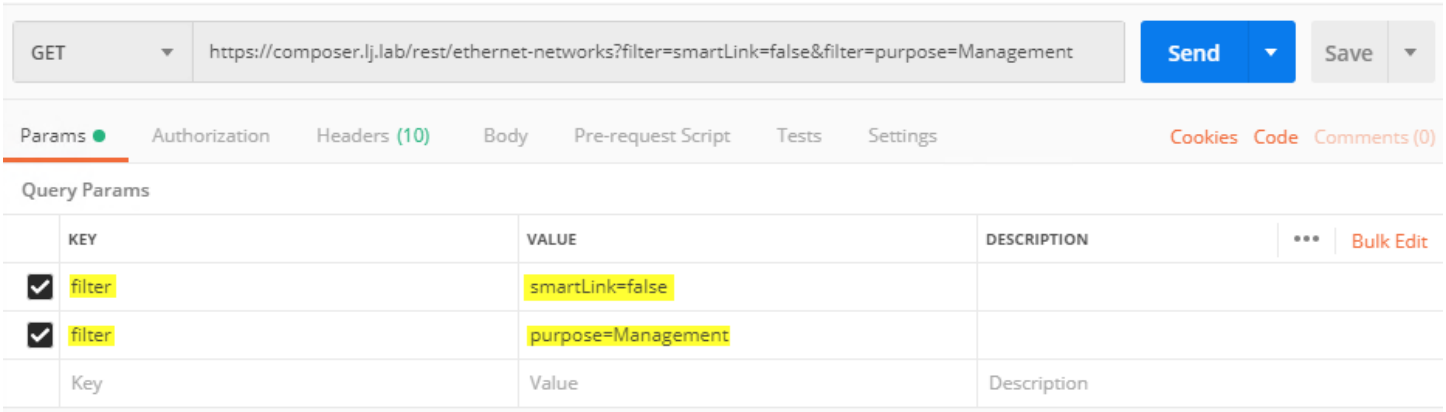
Hint: Look at the REST API documentation

https://composer.lj.lab/api-docs/current/#rest/remote-syslog

This example illustrates the need to have a good understanding of the REST API because SYSLOG since OneView 4.1 is one of the features that can only be enabled through the RESTful API.

In the next Lab, we will discover the HPE OneView Global Dashboard API and how we can interact with it.

Answer to exercise 1:



GET https://composer.lj.lab/rest/ethernet-networks?filter=smartLink=true&filter=purpose=ISCSI

Lab 6: HPE Global Dashboard API

HPE OneView Global Dashboard is a standalone appliance which allows you to manage up to 75 HPE OneView or HPE Synergy instances, and 150 HPE Hyper Converged 380 instances across data centers.

It helps your IT staff troubleshoot alerts and view core inventory data from one place and make more informed, faster decisions with better infrastructure visibility. With single sign-on, you are one click away from powerful, device-level lifecycle management.

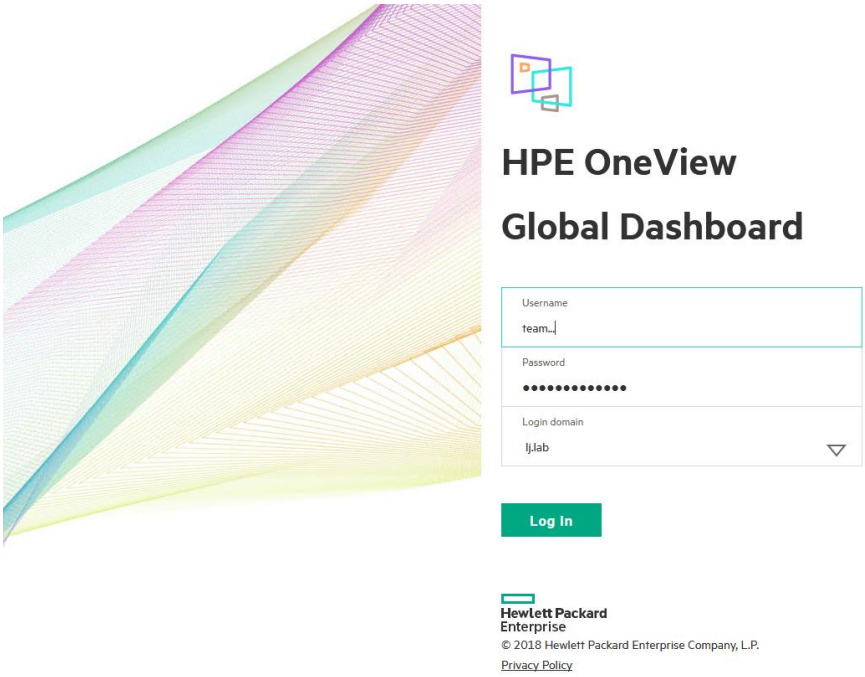
At no additional cost, you can simply download and run HPE OneView Global Dashboard alongside HPE OneView, allowing you to see your HPE Synergy, HPE BladeSystem c-Class, HPE ProLiant DL, HPE Hyper Converged 380, HPE SimpliVity 380, HPE Apollo, HPE ML350 Gen9 servers and HPE ML Gen10 servers, HPE Composable Cloud for DL systems, and HPE Superdome FLEX systems as one infrastructure.

Since version 1.6, released in October 2018, a RESTful API is available with HPE OneView Global Dashboard.
A RESTful API means that you can query all resources that have been added to Global Dashboard to access inventory, health status, and alerting information via command lines, and even scripting if you are ambitious enough.

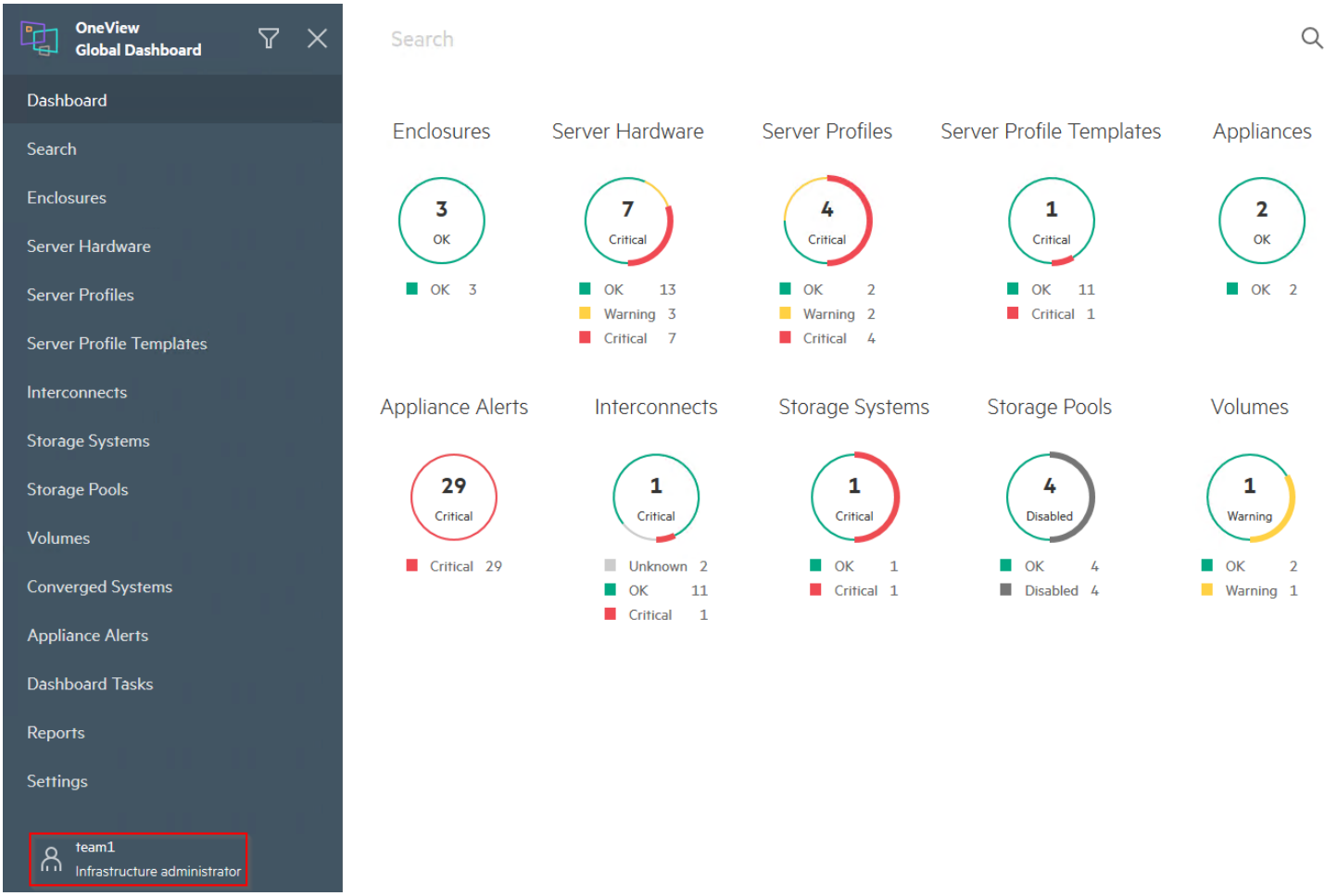
Login to the HPE OneView Global Dashboard (https://oneview-global-dashboard.lj.lab) using the following credentials:

- Username: **teamx**
- Password: **Passwordteamx**
- Domain: **lj.lab**

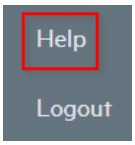
with **x** being your team number



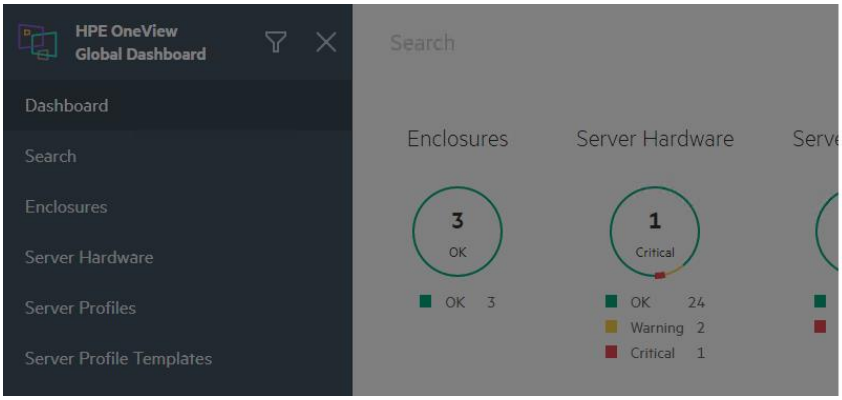
Once connected, click the user icon to open the session menu, which provides options to access online help, REST API reference, and logout.

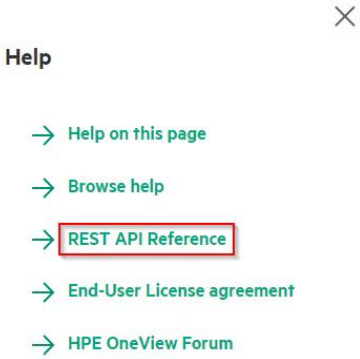


Select **Help**



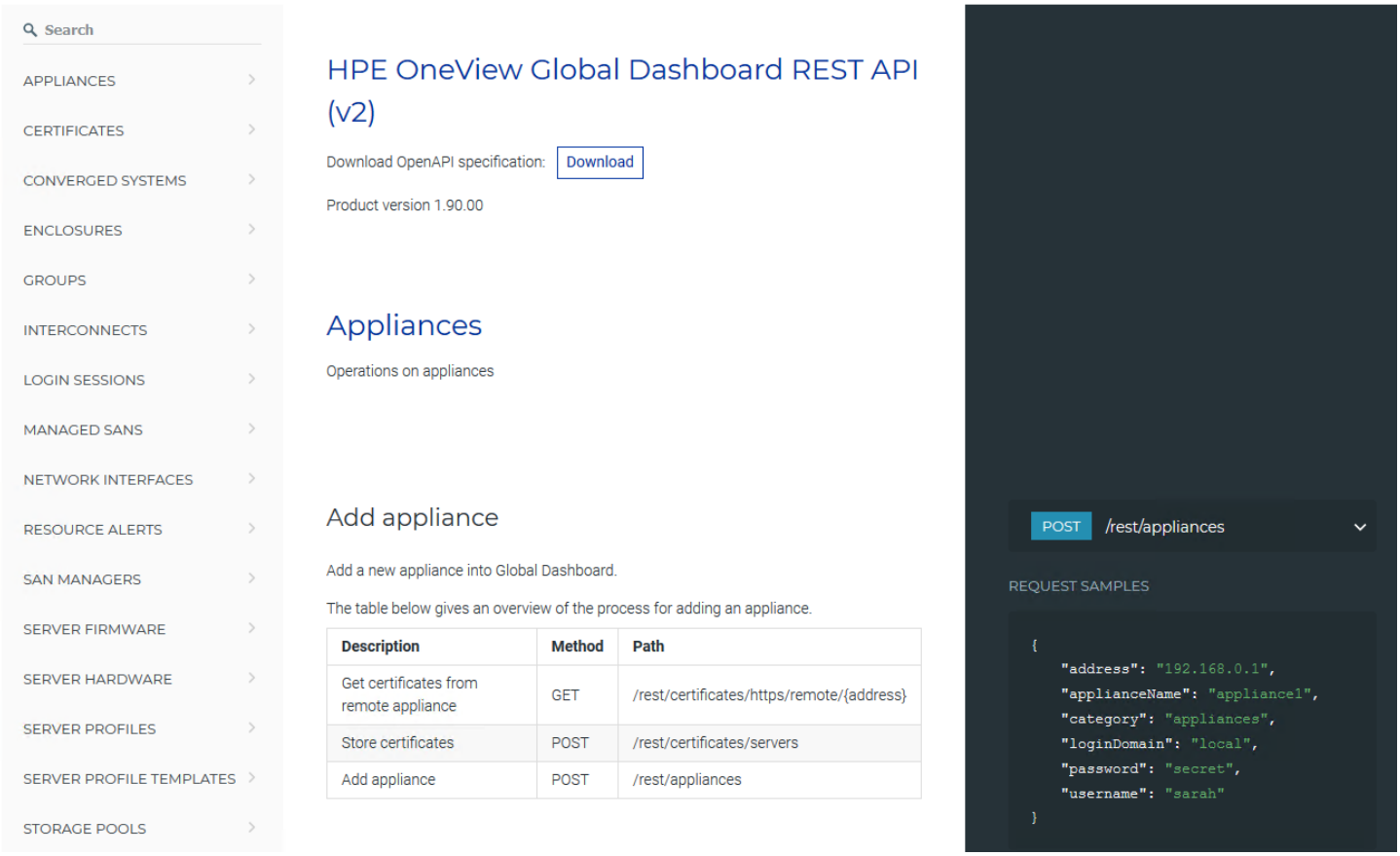
Then click on **REST API Reference**





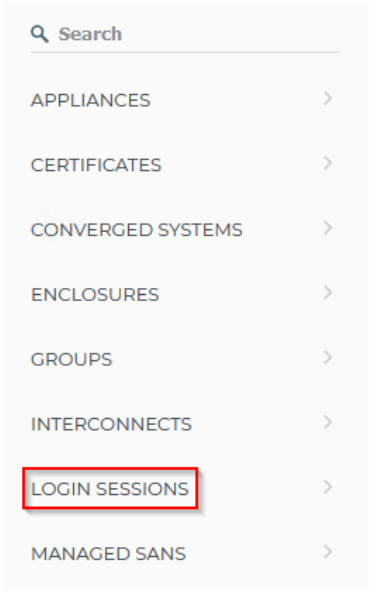
This opens the REST API reference in a new tab.

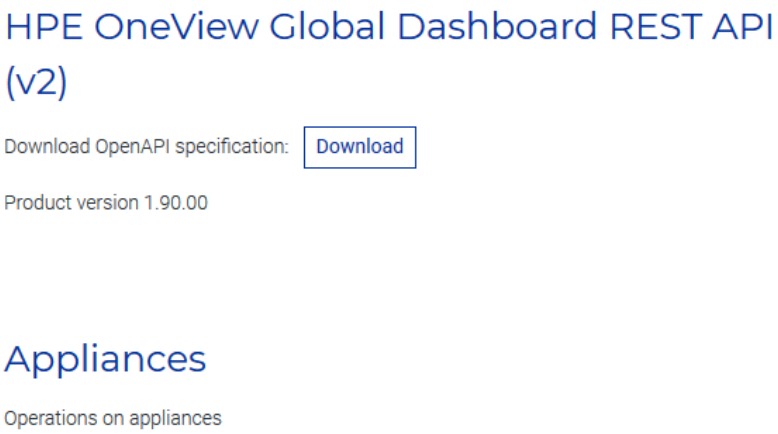
Like under HPE OneView, the API reference document provides all necessary information about the RESTful API.



On the left pane, you can find the list of resources supported by the API.

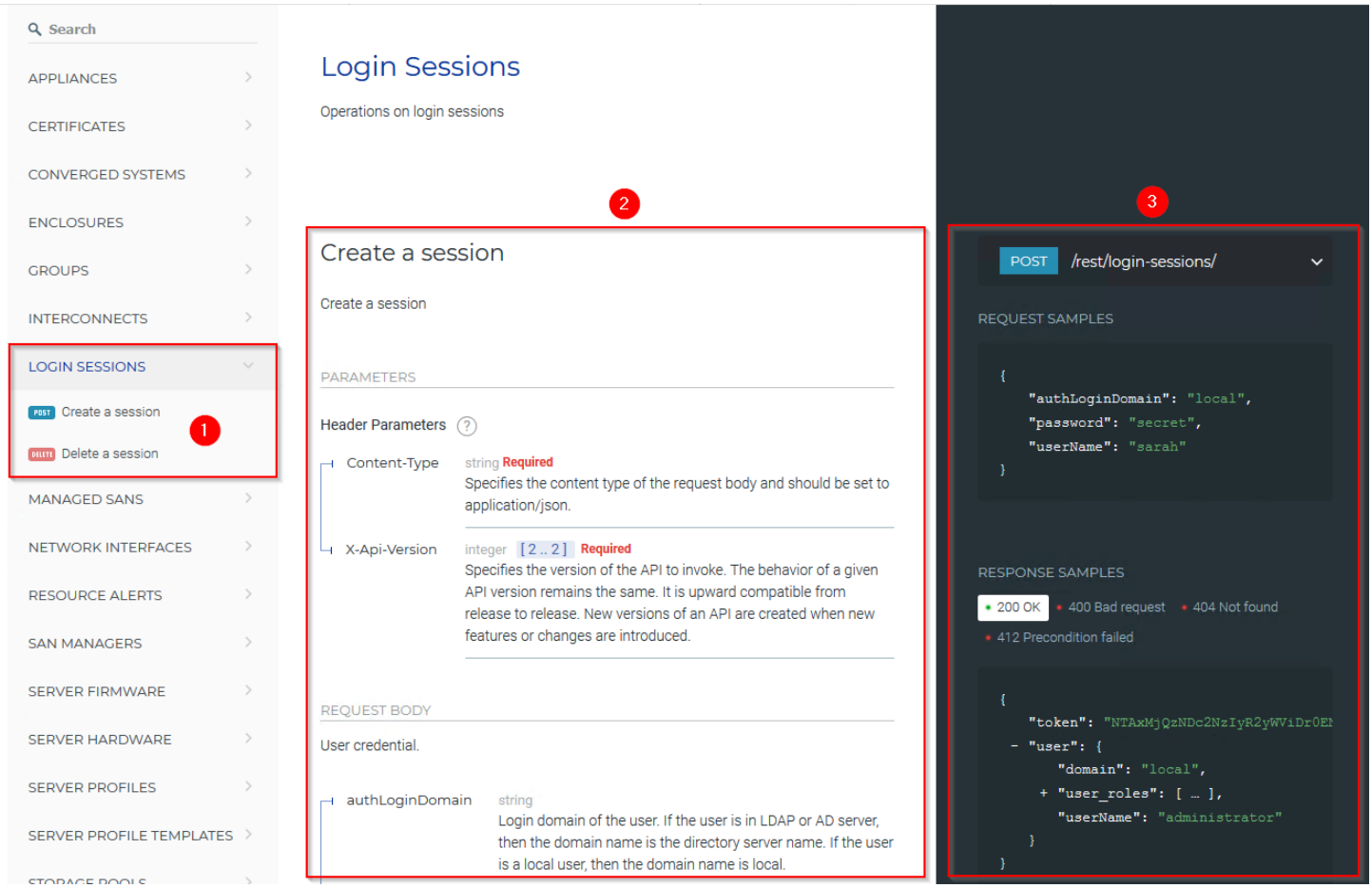
Click on **Login Sessions**



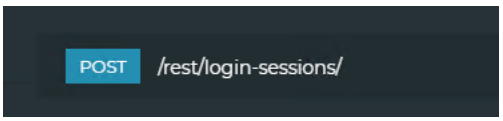


For each resource, the guide provides the following information:

1- A list of supported API request (`GET`, `POST`, `DELETE`, etc.)
2- A description of the specific task performed by each API request with a list of all required parameters for the header and body.
3- A pane with samples for API request and response.

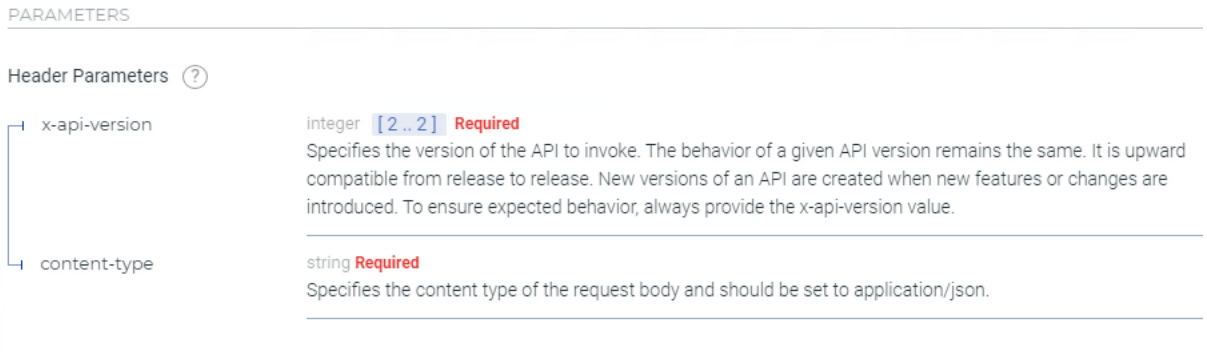


For our first interaction with the Global Dashboard API, let's open a login session to HPE OneView Global Dashboard. From the documentation, we can see that in order to open (i.e. create) a login session, we need to use the `POST` method against the URI `/rest/login-sessions`.



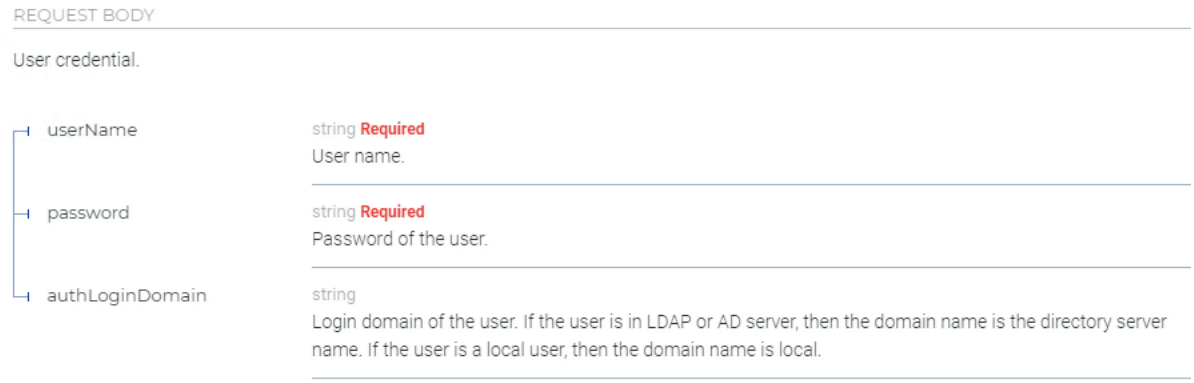
We can also see that we need to set two HTTP Headers:

- **X-API-Version**: set to the version of the API we want to use (set to *2*)
- **Content-Type**: set to *application/json*, to tell the API, that the payload we will be providing is in JSON notation



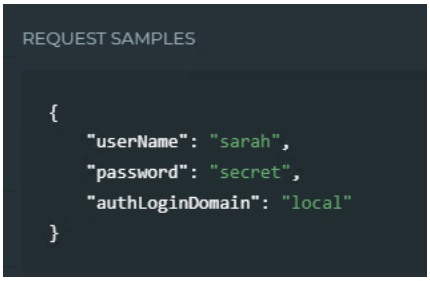
The payload expected by the API must contains (at minimum) the following parameters:

- **userName**: set to the username to use for authentication
- **password**: set to the password for that username
- **authLoginDomain**: set the login domain of the user



Note: Be careful, JSON is case sensitive. userName is expected here, not username, nor UserName. As a rule, the API expects lower case words, but capitalized, after the first word, if there are more than one word.

The payload is notified by the `{ … }` in the sample pane:



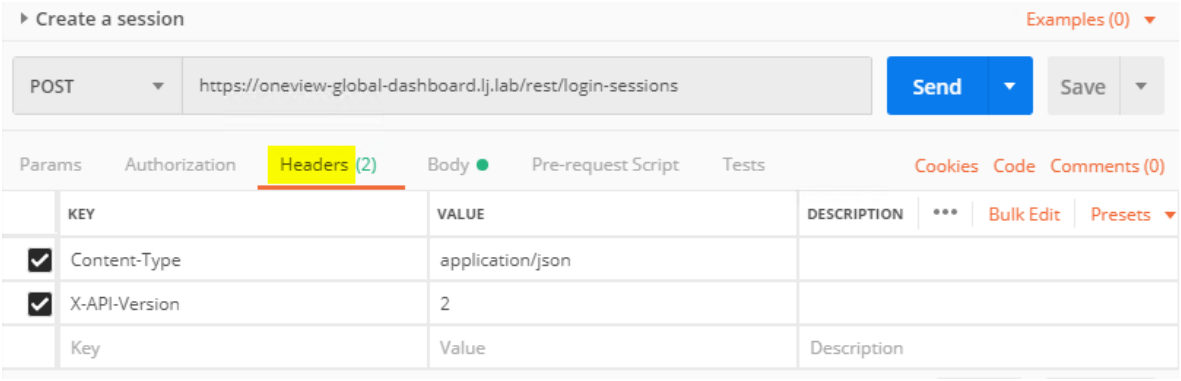
You can test this now using Postman, using the following parameters:

Field	Value
URL	`https://oneview-global-dashboard.lj.lab/rest/login-sessions`
Verb	`POST`
Header	`Content-type:application/json` `X-API-Version:2`

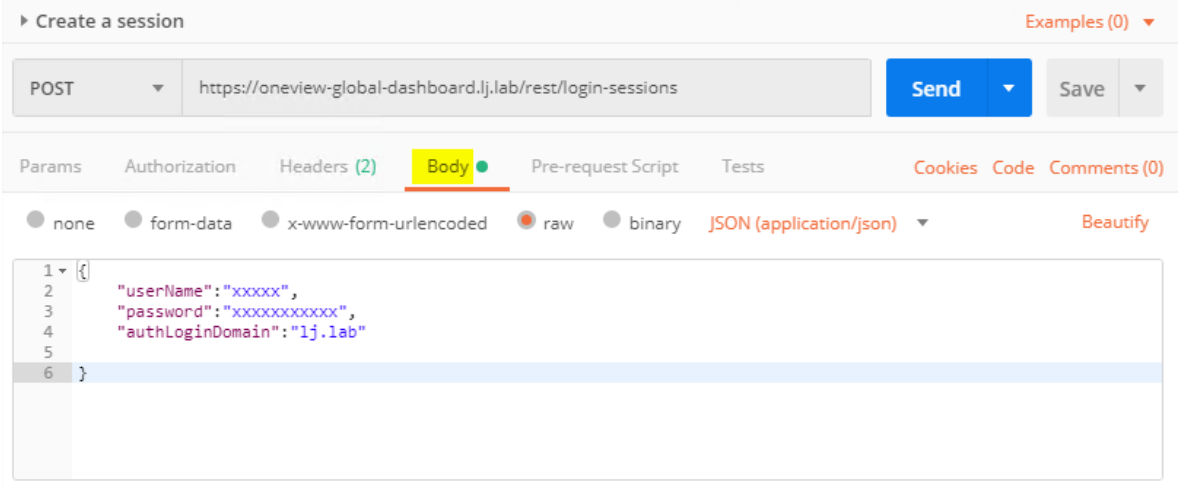
Payload	`{`
	`"userName": "xxxxx",`
	`"password": "xxxx",`
	`"authLoginDomain": "lj.lab"`
	`}`

Make sure you set your appropriate credentials for `userName` and `password`.

Content of the header:

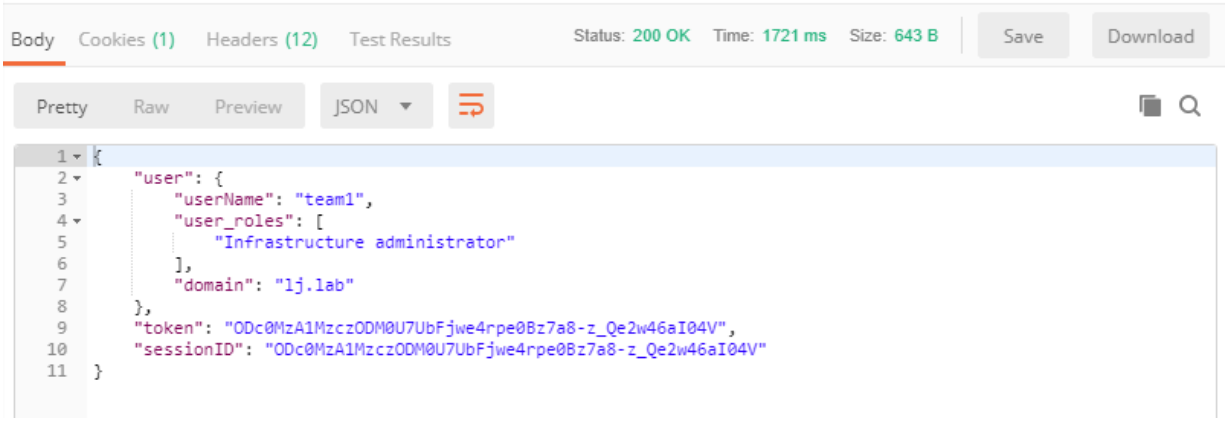


Content of the body payload:



Press **Send**.

The response we get is the following:

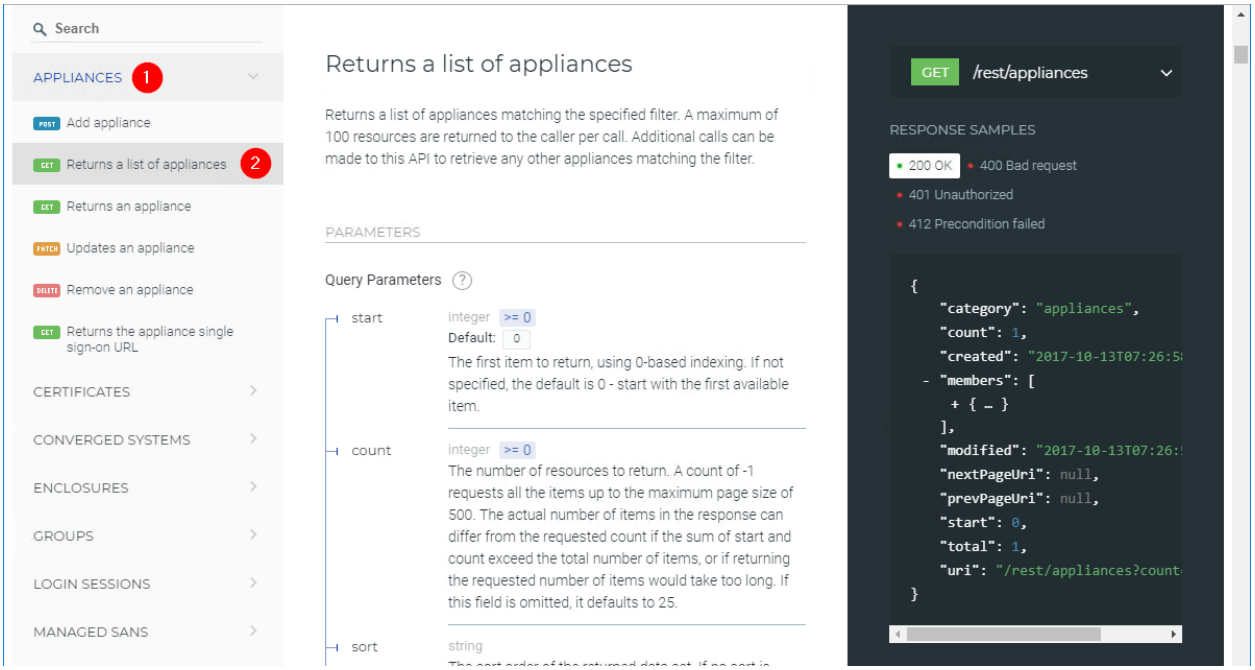


The *sessionID*, also known as the session token, is what we are looking for. This is the token that will be used for authentication in the subsequent calls to the API.

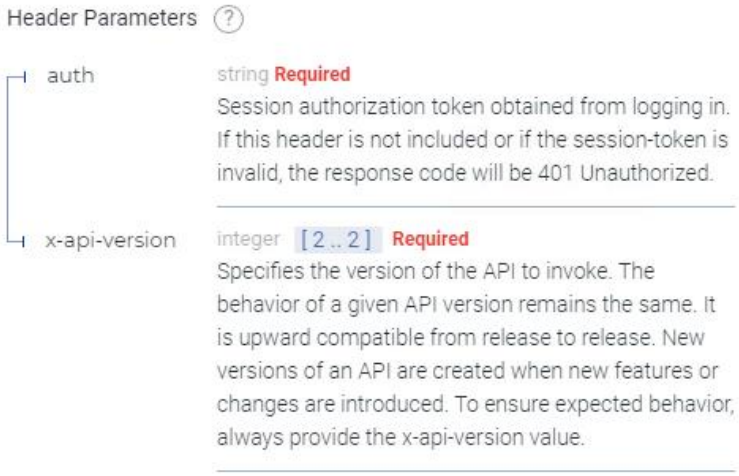
So, in order to be authorized to call any other API methods, we will need to provide the session token as another HTTP Header called `Auth`.

We can now test this token and call another API method. Let us pick `GET /rest/appliances` which return a list of appliances managed by Global Dashboard.

This is what we can read:



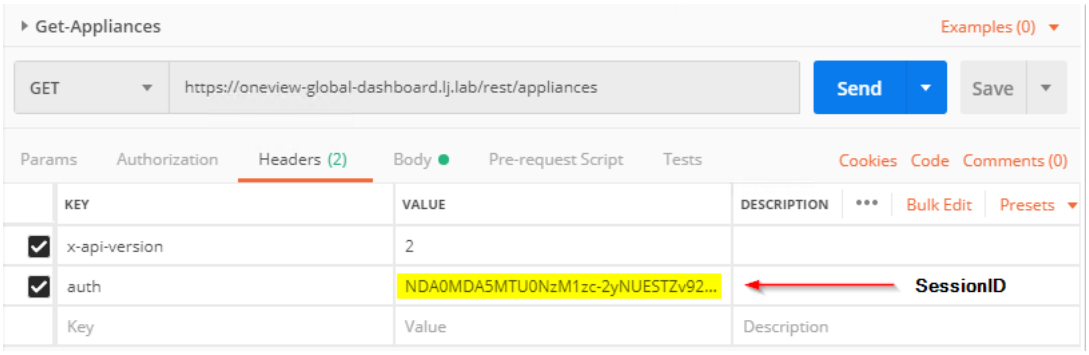
Notice now the presence of Auth with the sessionID in the header for this call.



So, use Postman with the parameters found here:

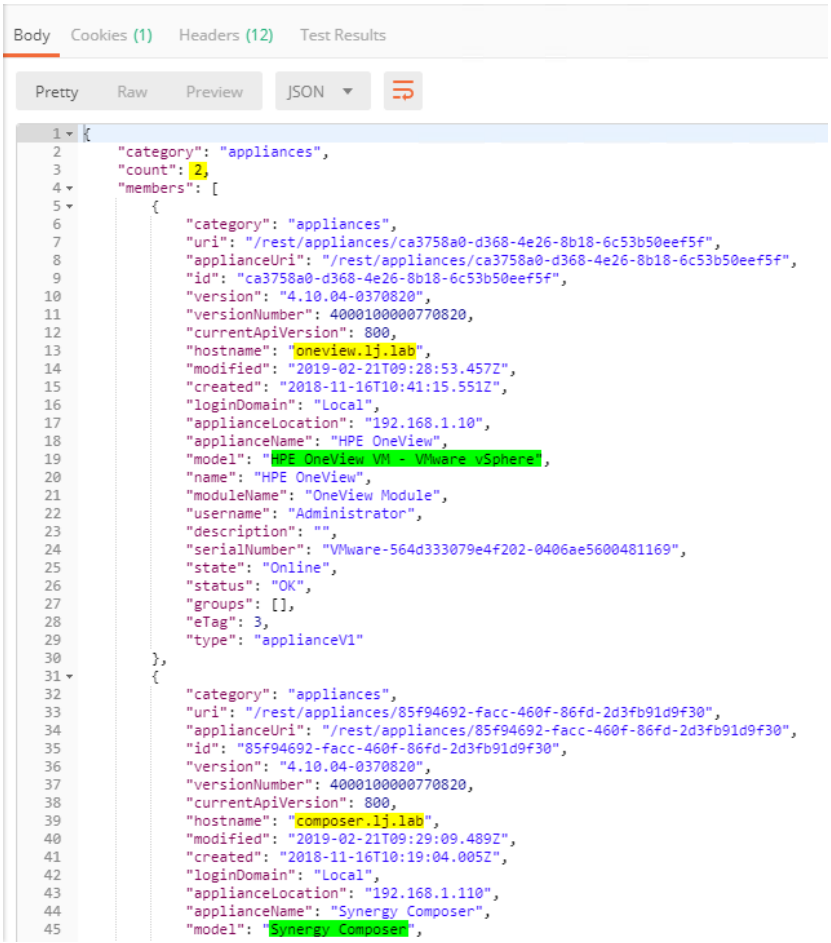
Field	**Value**
URL	`https://oneview-global-dashboard.lj.lab/rest/appliances`
Verb	`GET`
Header	`Auth : your sesionID` `X-API-Version:2`

So, it is necessary to copy/paste your *SessionID* from your `POST /rest/login-sessions` in the `auth` value field:



Then when all set, press **Send**

You should get the list of appliances managed by Global Dashboard in the response body, again in a JSON format. We can easily see that we have 2 appliances, one is a OneView VM VMware appliance, another is a OneView Synergy Composer:



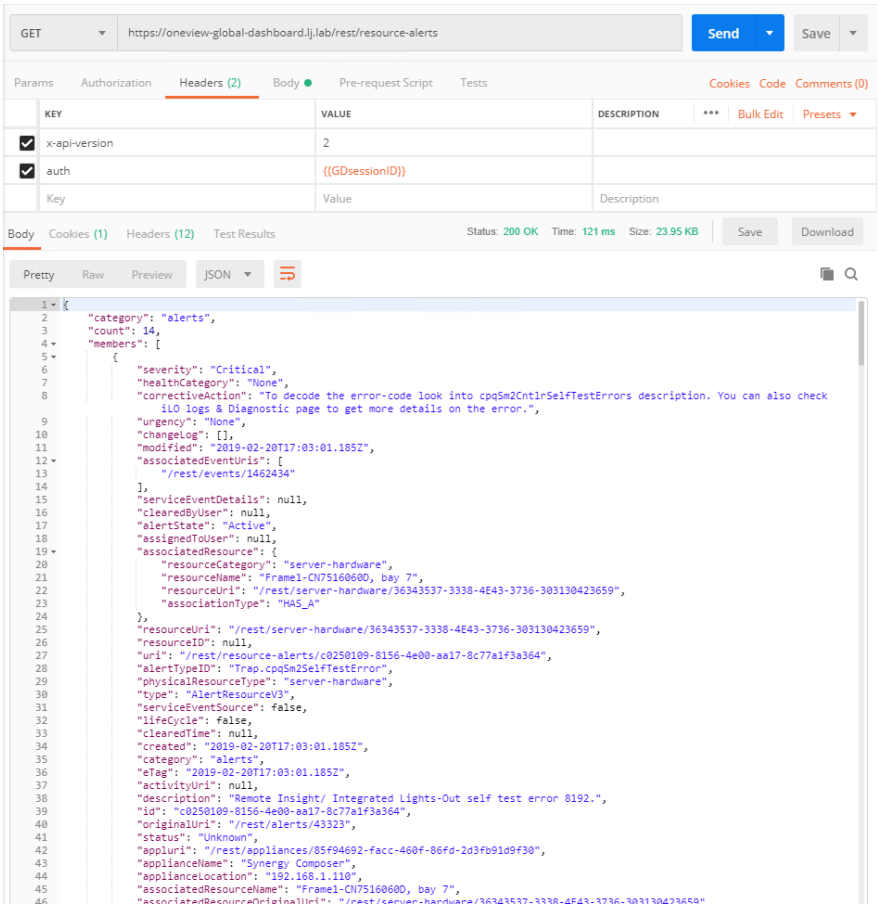
Note: You can remove the *Auth* sessionID and try again as a quick check, you should get a 401 Unauthorized status code.

The most interesting data that can be found in Global Dashboard is the resource alerts. Global Dashboard collects all the critical alerts from the appliances it monitors. Using the REST API and some simple scripting, we can develop a script that would print out all alerts.

So, let's see now what we can get from the resource alerts, in Postman, enter the following parameters:

Field	**Value**
URL	`https://oneview-global-dashboard.lj.lab/rest/resource-alerts`
Verb	`GET`
Header	`Auth : your sesionID` `X-API-Version:2`

You should get a list of alerts received by Global Dashboard from the two appliances it monitors:



This concludes our Lab 6.

Lab 6 provided a quick overview of the HPE OneView Global Dashboard API, we discovered that the approach is identical to the HPE OneView API, we use the same techniques, the same API patterns and like in OneView, the help in the documentation is very helpful.

In the next lab, we will see how we can use scripting language like PowerShell and Python to interact with these APIs.

Lab 7: Scripting languages

To conclude these labs, we can explore scripting languages and see how we can create scripts to execute REST calls and eventually automate their execution.

Like with Postman, scripting languages can be used to interact with the OneView and Global Dashboard REST APIs but they provide much more benefits, you can automate the execution of tasks, the scripts can be combined into more complex programs and provide lot of flexibility and agility.

Using the REST API and some simple scripting, we are going to build in the lab, a script that would print out the number of alerts available in HPE Global Dashboard with a description of alerts, the severity and the associated corrective actions. This exercise can be helpful to see how someone could easily create extremely useful tools for anyone troubleshooting at a data center.

In this lab, we will only glance at two languages, PowerShell and Python. Feel free to run both or to jump to the scripting language of your choice.

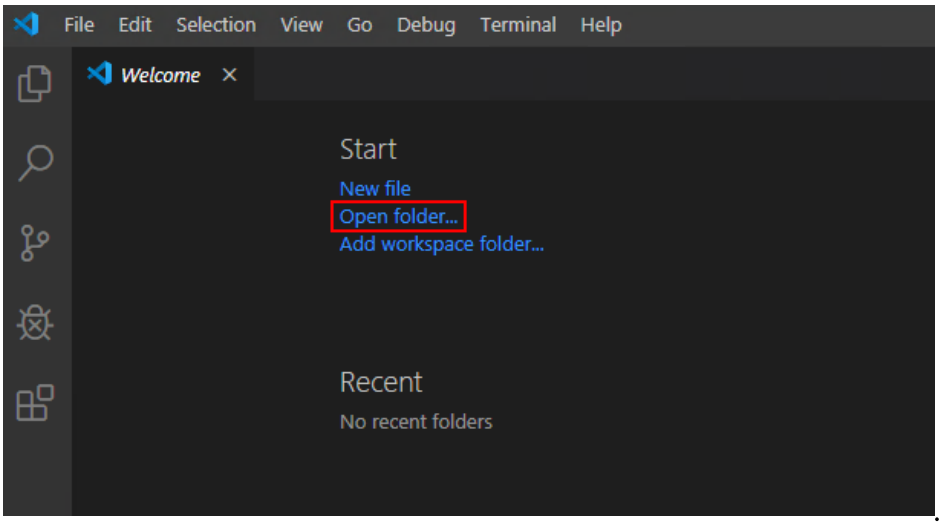
PowerShell

A PowerShell library for HPE OneView is available from HPE, it provides many functions to manage the appliance. We are not going to use this library in this lab as we want you to understand the way a REST call can be realized in PowerShell. Besides that, a library cannot provide all REST calls provided by an API, so it is essential to know how to pass a REST request natively in PowerShell.

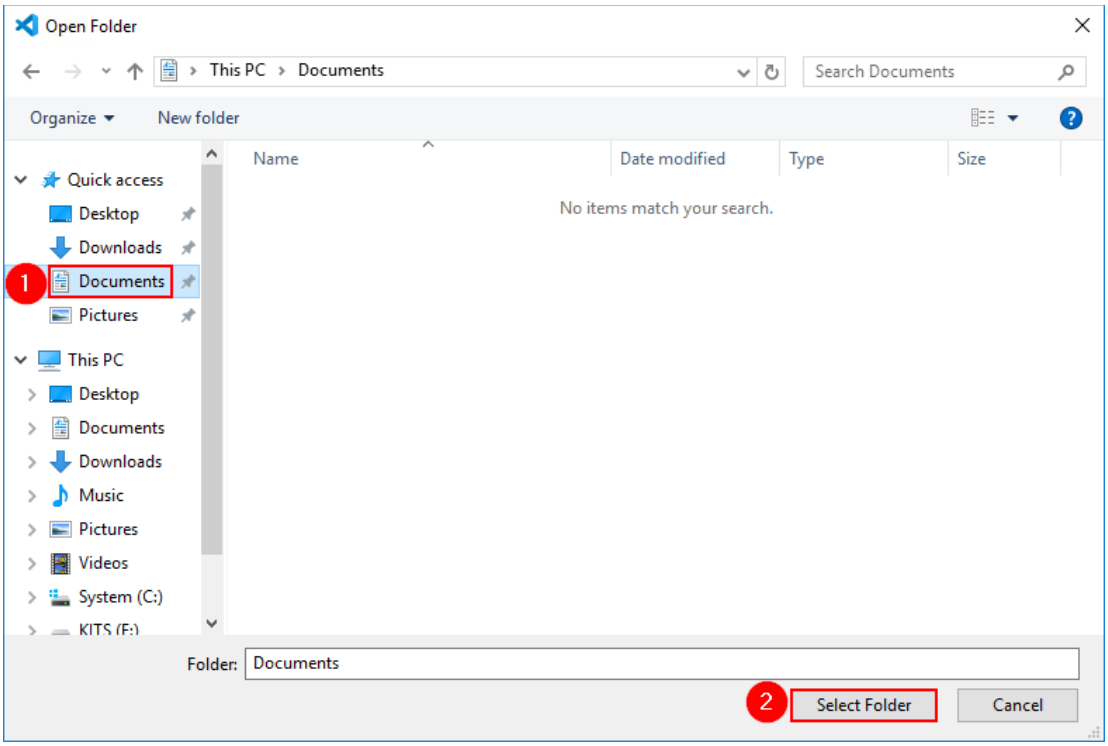
For more information about the HPE OneView PowerShell library, see https://hewlettpackard.github.io/POSH-HPOneView/

Start Visual Studio Code using the shortcut on the desktop:

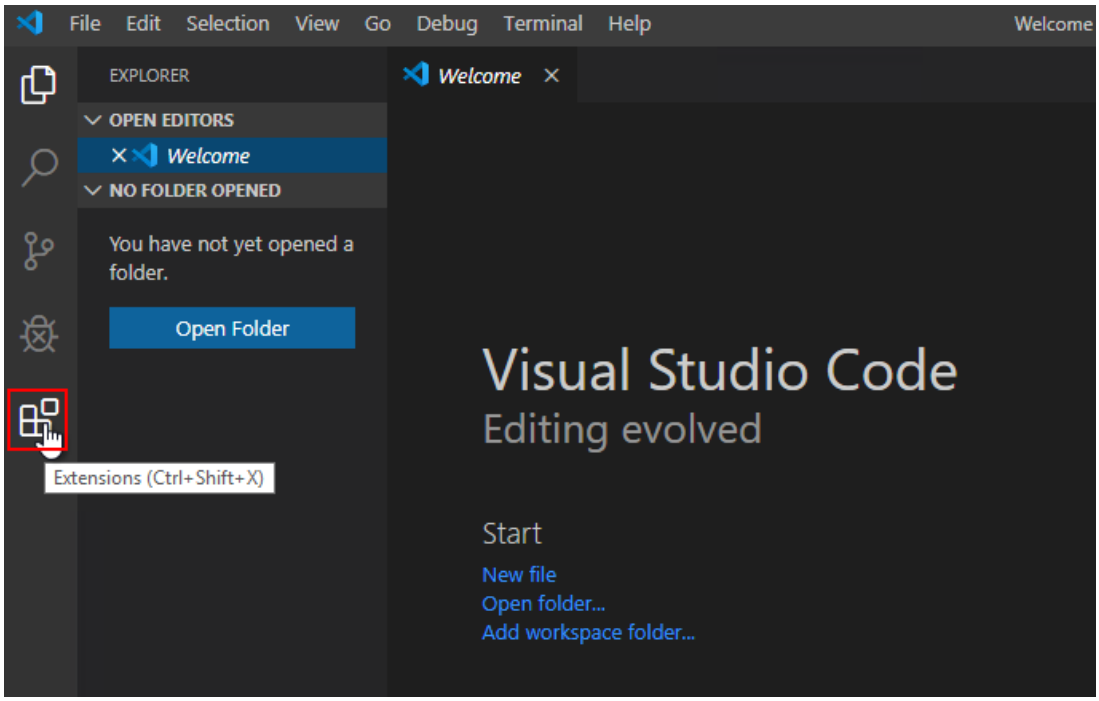
Click on **Open Folder...**



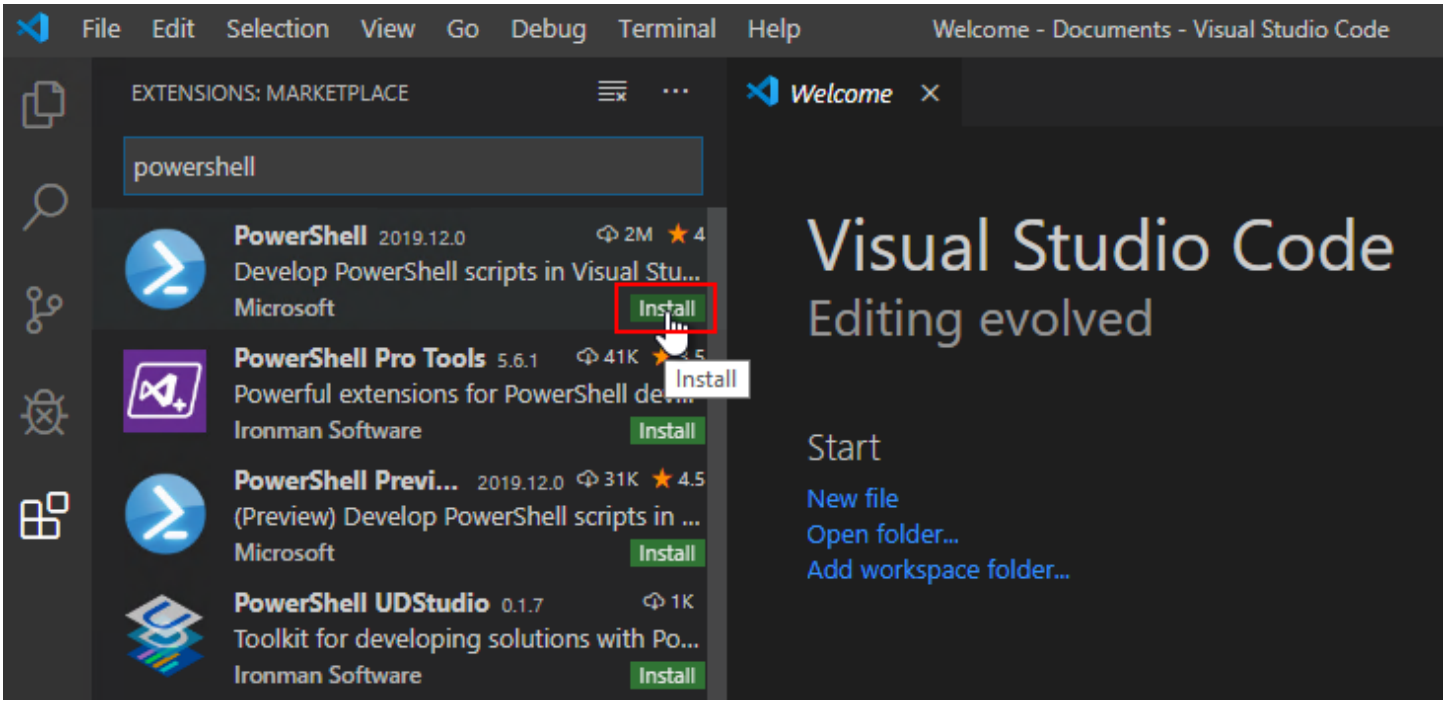
And select **Documents** and click **Select Folder**



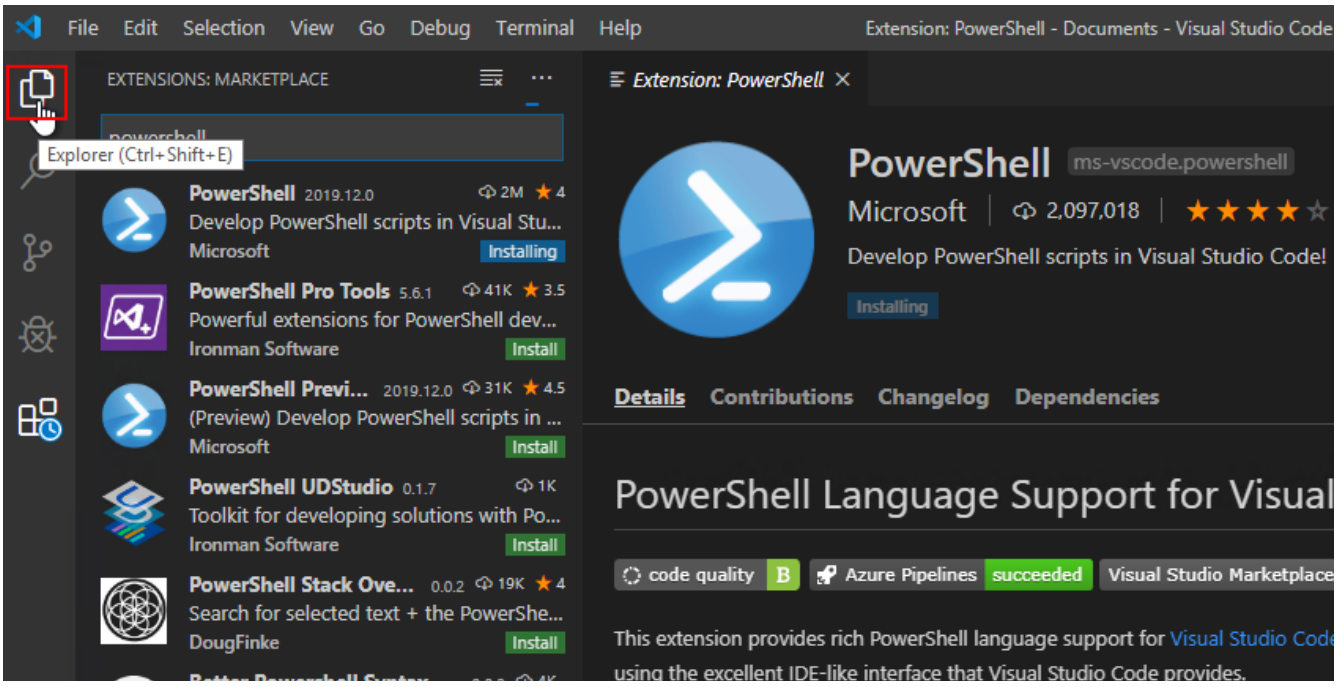
Then click on **Extensions** on the Activity bar



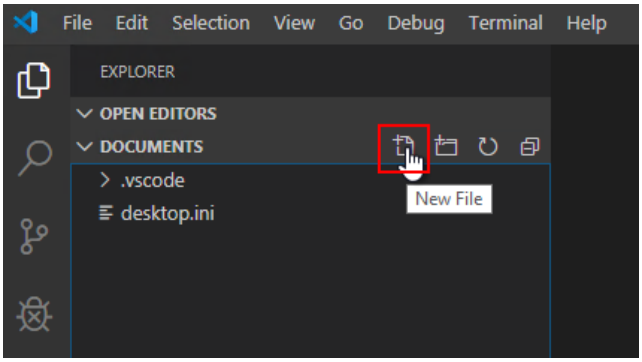
If not already installed, install the *PowerShell* extension usually listed at the top of the list. You can enter **PowerShell** in the search tool to find the extension. Click on the **Install** button:



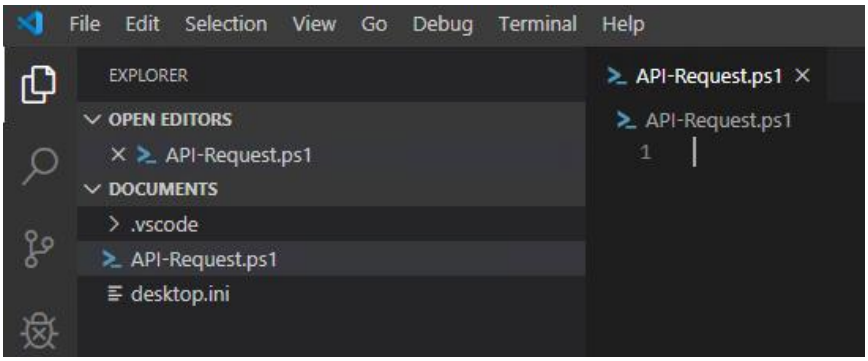
While the extension is getting installed, click on **Explorer** on the Activity Bar



Then click on **New File**



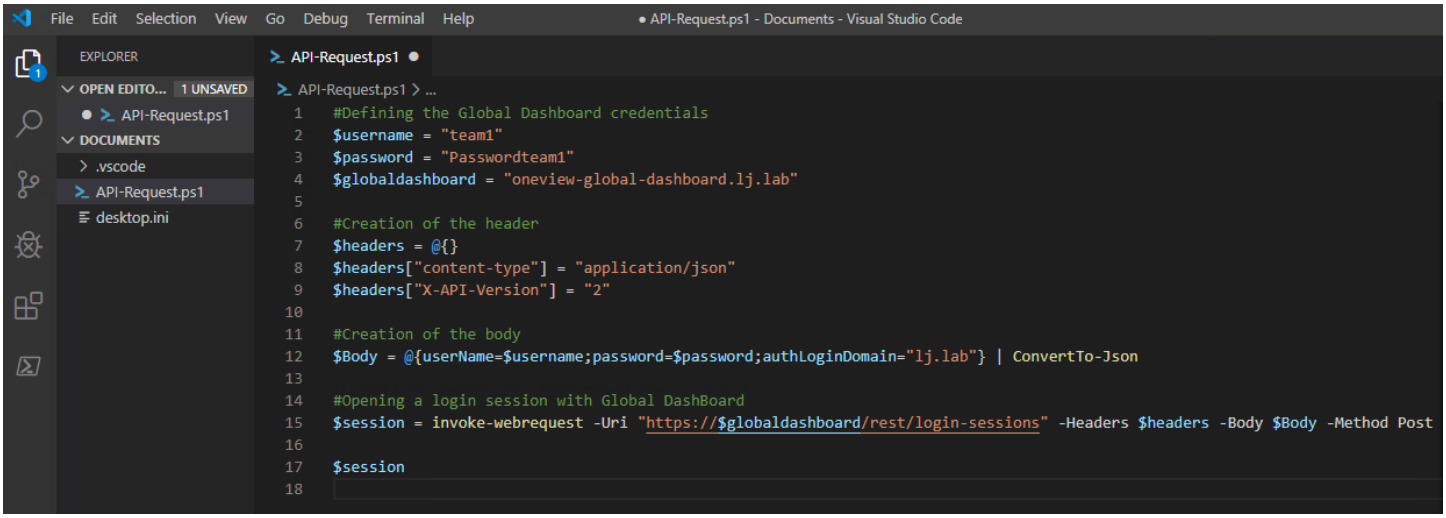
Enter a name like **API-Request.ps1** then press **ENTER**



You can then enter the follow-on lines to build your PowerShell script.
The script we are going to use starts as follows:

```
#Defining the Global Dashboard credentials
$username = "team1"
$password = "Passwordteam1"
$globaldashboard = "oneview-global-dashboard.lj.lab"

#Creation of the header
$headers = @{}
$headers["content-type"] = "application/json"
$headers["X-API-Version"] = "2"

#Creation of the body
$Body = @{userName=$username;password=$password;authLoginDomain="lj.lab"} | ConvertTo-Json

#Opening a login session with Global DashBoard
$session = invoke-webrequest -Uri "https://$globaldashboard/rest/login-sessions" -Headers $headers
-Body $Body -Method Post

$session
```

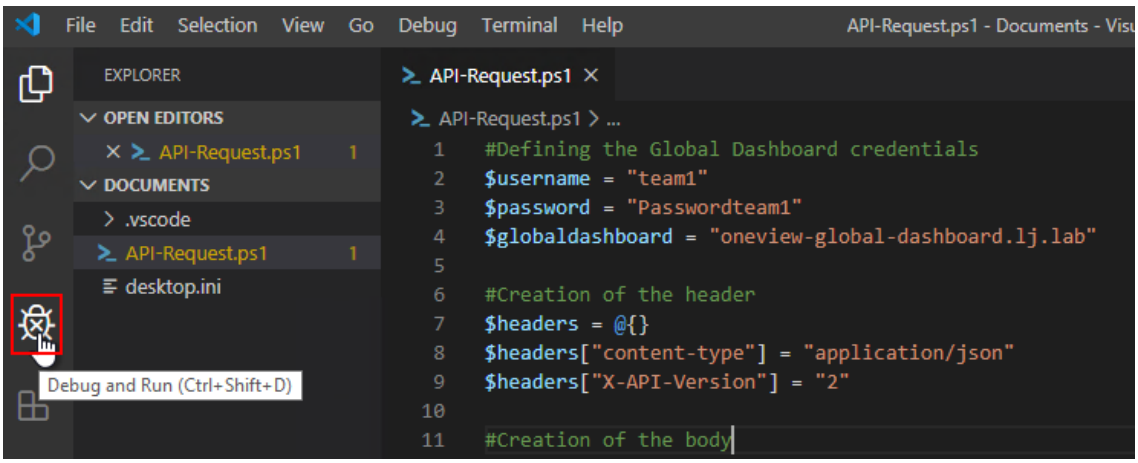
Just copy/paste these lines in VS Code:



Notice that the script follows the same steps we saw previously with Postman: the creation of the header (with `content-type` and `x-api-version`), the creation of the body for the login session (with `userName`/`password`/`authLoginDomain`) then we use the PowerShell cmdlet `invoke-webrequest`. This cmdlet is the PowerShell command to send HTTP/HTTPS/FTP and FILE request to a web page or web service. In our case, we will use it for a HTTPS request to our Global Dashboard REST API using a `POST` method on `\rest\login-sessions`. The *URI, headers, body* and *method* are provided as well as parameters.

Once properly set, you can press **CTRL** + **S** to save the file.

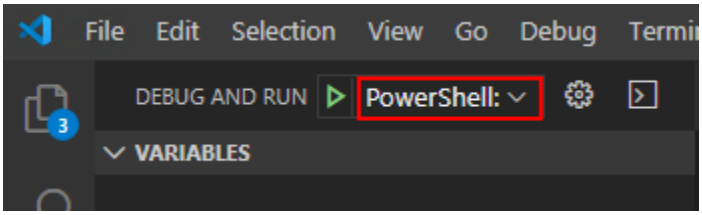
Now let's run the script, click on **Debug and Run** on the Activity Bar



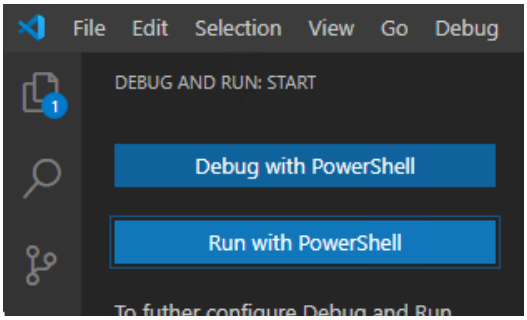
One of the key features of Visual Studio Code is its great debugging support. VS Code's built-in debugger helps accelerate edit, compile and debug script.

Visual Studio Code's built-in debugger needs some quick setup to run and debug PowerShell scripts. Please follow the procedure that applies to your environment:

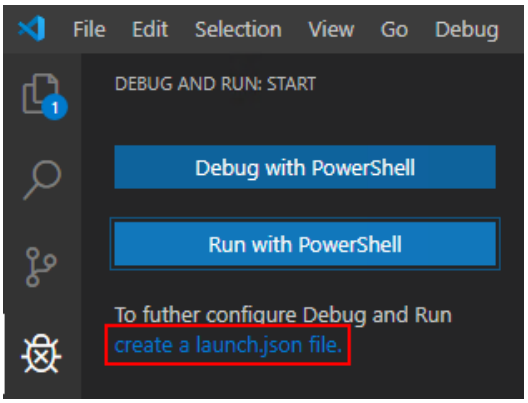
1. If *PowerShell* is displayed in the Debug and Run dialog, your environment is all set for PowerShell scripting, you can skip the setup section and step over points 2. and 3.:



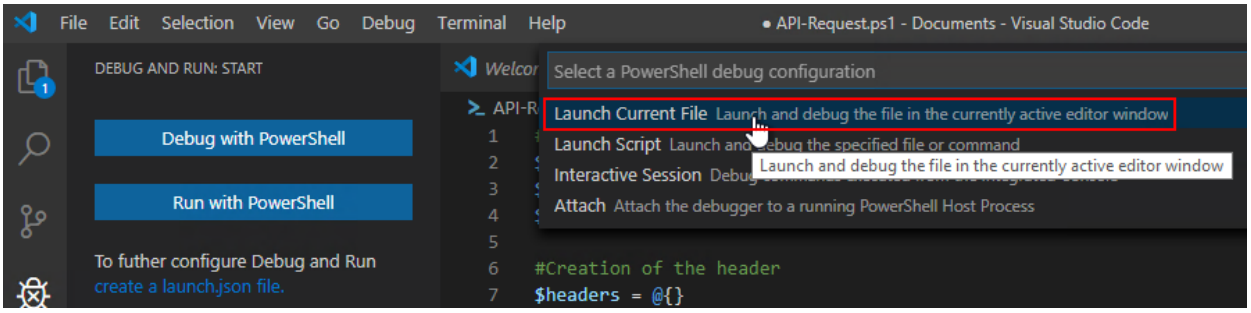
2. If the following dialog is displayed in the Debug and Run pane:



   a. Click on **Create a launch.json file** to configure Debug and Run



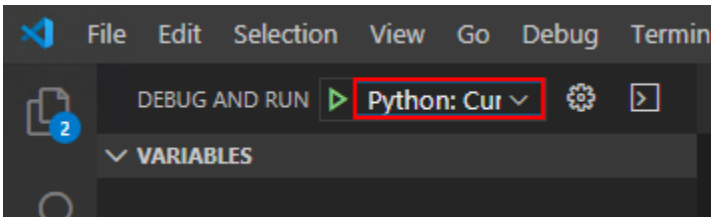
b. Select the first option **Launch Current File**:



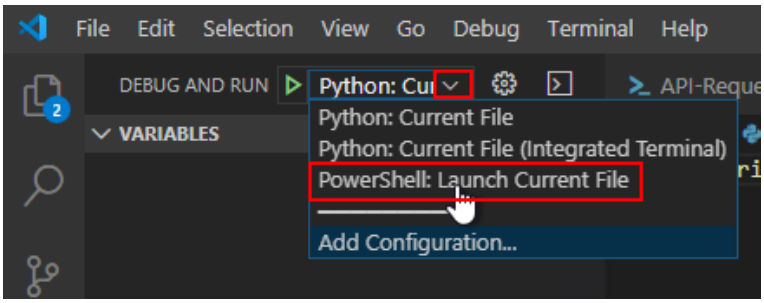
c. Press **CTRL + S** to save the configuration file

d. Press **CTRL + F4** to close the file

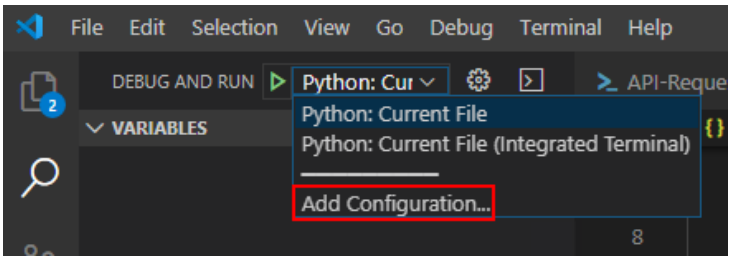
3. If *Python* is displayed in the Debug and Run dialog:



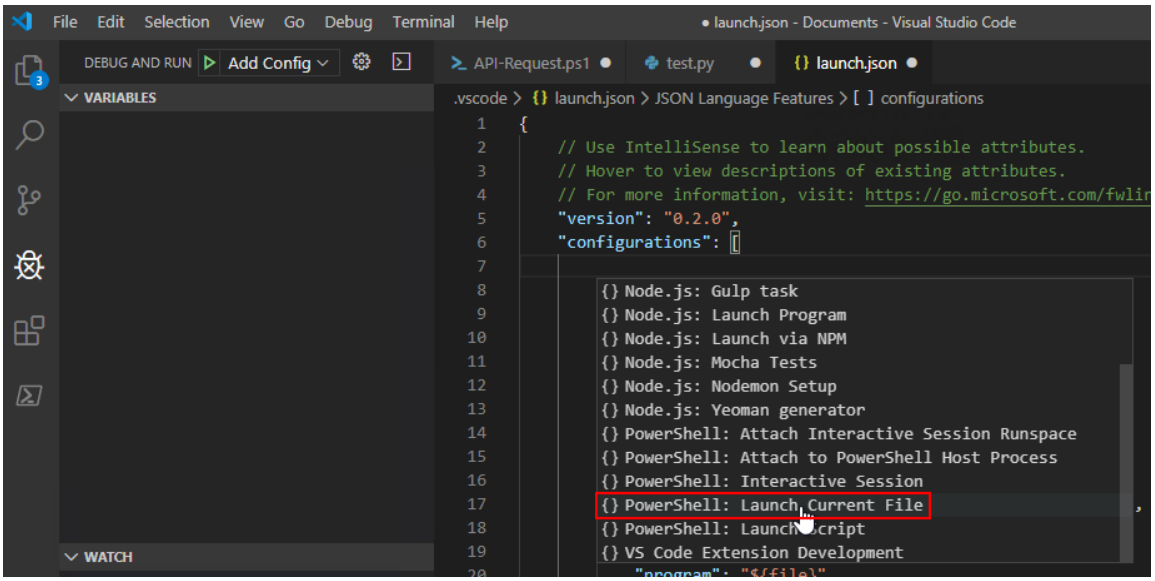
a. Select **PowerShell: Launch Current File** from the Debug and Run drop-down menu and you are done.



i. If there is no PowerShell configuration, you need to select **Add configuration** from the drop-down menu:



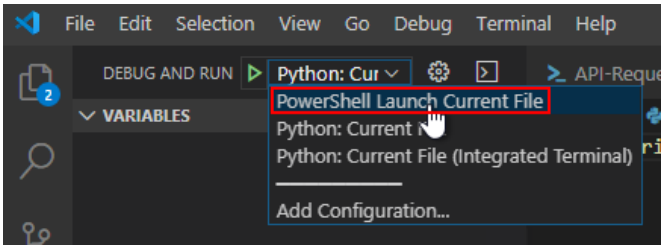
ii. Then select **PowerShell: Launch Current File**



iii. Press **CTRL + S** to save the configuration file

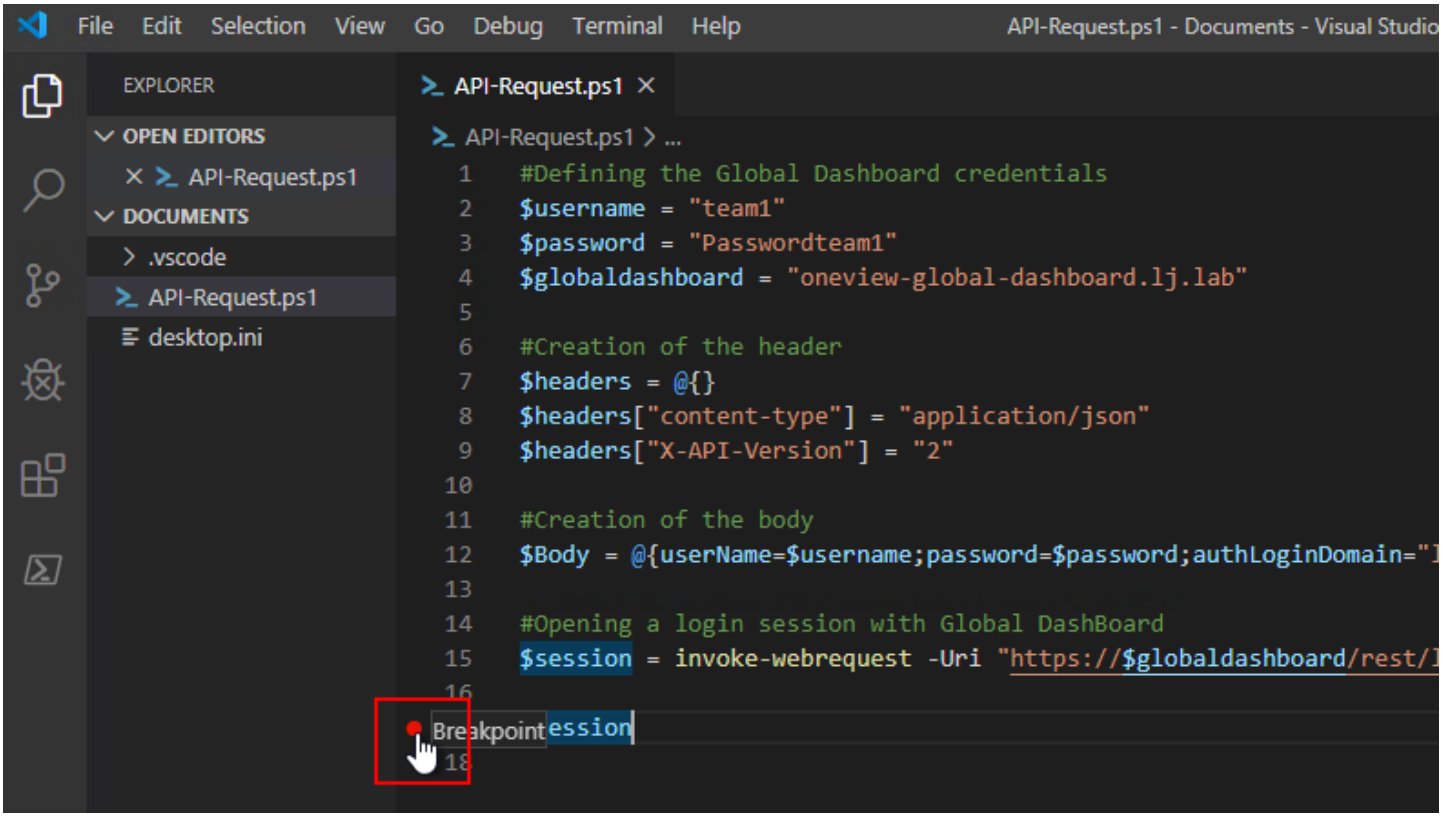
iv. Press **CTRL + F4** to close the file

v. Then select **PowerShell: Launch Current File** from the Debug and Run drop-down menu

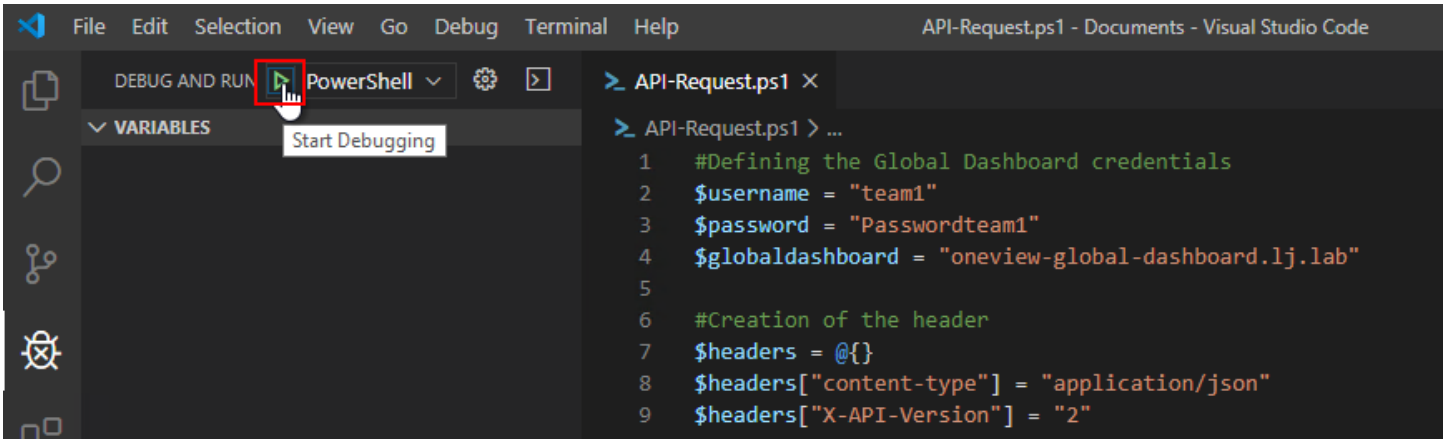


The Visual Studio Code debug and run setup is now complete, we can start debugging our script.

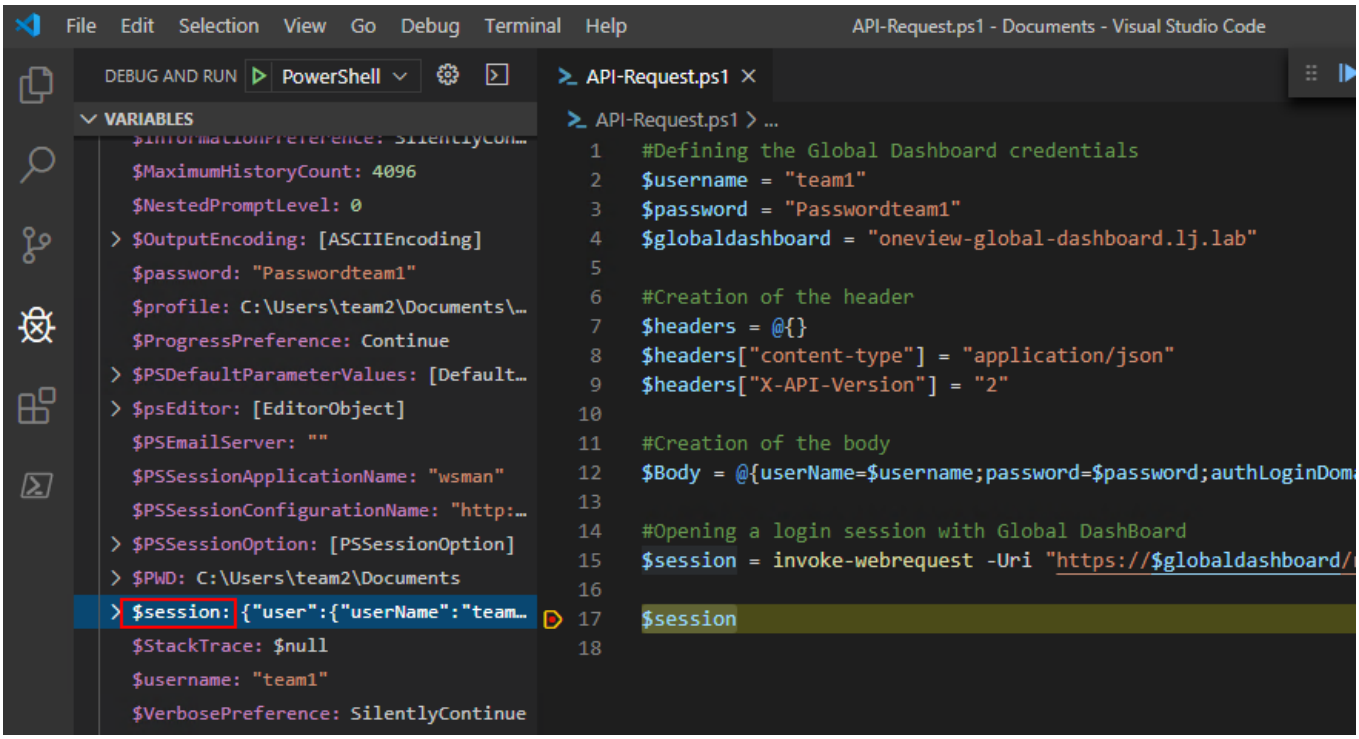
Using your mouse, drop a Breakpoint on the last line. Breakpoints are normally shown as red filled circles, can be toggled by clicking on the editor margin:



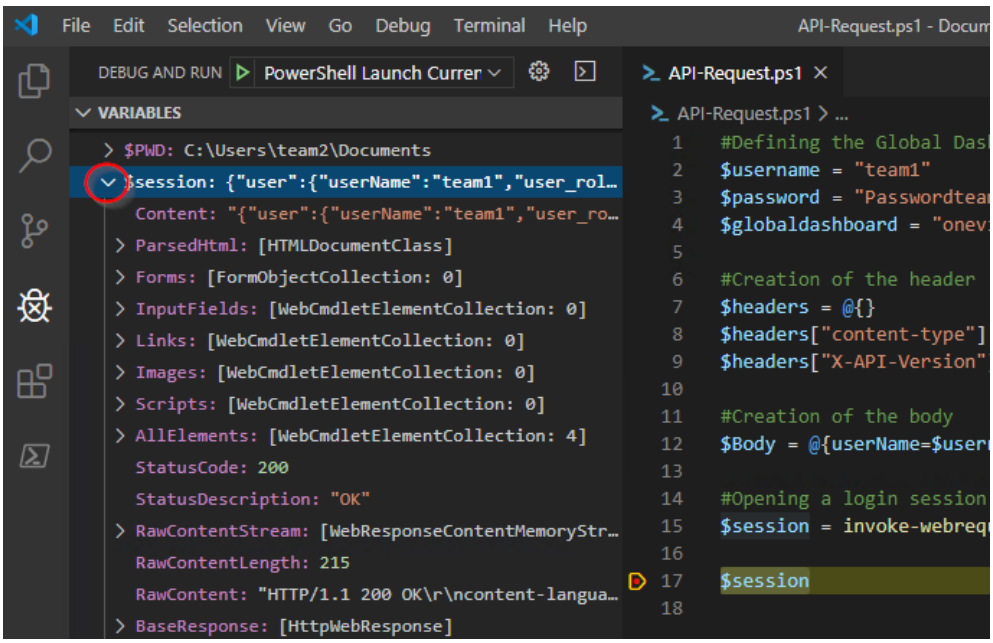
Then click on **Start Debugging** to run the script



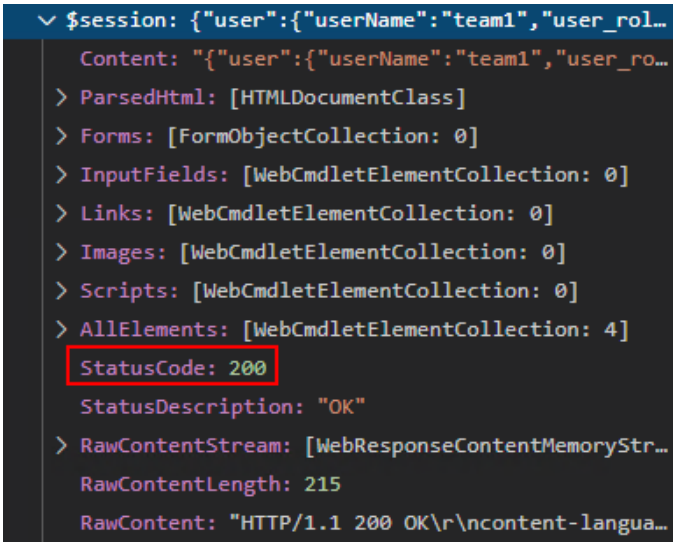
On the left-hand pane, the variables content is displayed, `$session` is the variable we are using to store the result of `invoke-webrequest`



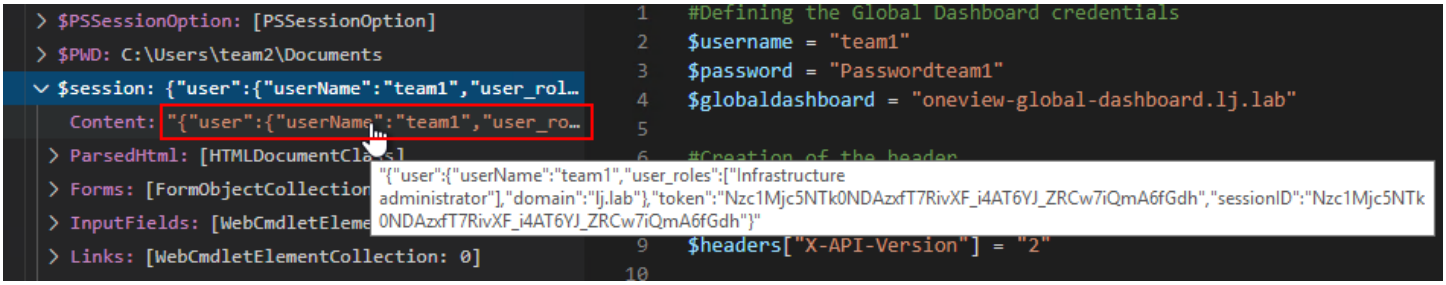
You can discover the content of `$session` by using the expand icon:



A successful API request response code **200** should be displayed:



If you hover your mouse over the `content` description you will get the following popup box:

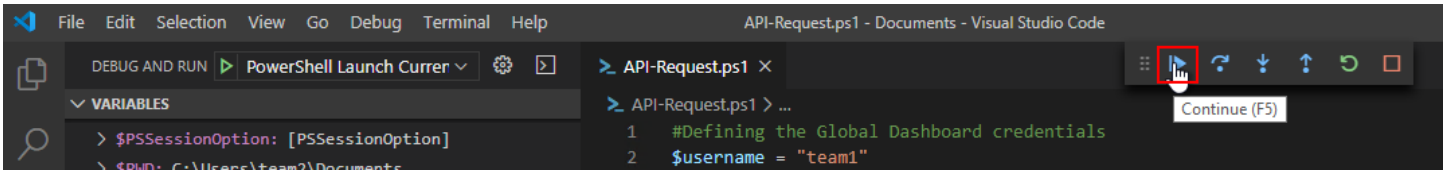


This is the response received from the API

"{"user":{"userName":"team1","user_roles":["Infrastructure administrator"],"domain":"lj.lab"},"token":"Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh","sessionID":"Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh"}"

Notice that the content of *content* is in JSON format. So, to access to the *sessionID*, we need to convert the response into a PowerShell object, easier to manipulate.

Click on **Continue** or **F5** to continue the script execution.



The last step of the script executes `$session`. This is another way to get the result of our API request displayed in the console:

```
11  #Creation of the body
12  $Body = @{userName=$username;password=$password;authLoginDomain="lj.lab"} | ConvertTo-Json
13
14  #Opening a login session with Global DashBoard
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    2: PowerShell Integrate

[DBG]: PS C:\Users\team2\Documents>


StatusCode        : 200
StatusDescription : OK
Content           : {"user":{"userName":"team1","user_roles":["Infrastructure administrator"],"domain":"lj.lab"},"to
                    ken":"Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh","sessionID":"Nzc1Mjc5NTk0NDAzxfT7RivXF_i
                    4AT6YJ_Z...
RawContent        : HTTP/1.1 200 OK
                    content-language: en-us
                    Vary: Accept-Encoding
                    Keep-Alive: timeout=15, max=100
                    Connection: Keep-Alive
                    Content-Length: 215
                    Cache-Control: no-cache, private
                    Content-Type: applicati...
Forms             : {}
Headers           : {[content-language, en-us], [Vary, Accept-Encoding], [Keep-Alive, timeout=15, max=100],
                    [Connection, Keep-Alive]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 215
```

As we can see, the result holds several property fields, like a *statuscode*, a *statusdescription* and more importantly a *content* property.

To access the *content* property, it's easy, just enter in the console:
`$session.content`

```
PS C:\Users\team2\Documents> $session.content
{"user":{"userName":"team1","user_roles":["Infrastructure administrator"],"domain":"lj.lab"},"token":"Nzc1Mjc5NTk0NDA
zxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh","sessionID":"Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh"}
PS C:\Users\team2\Documents>
```

So, the content of *content* is in JSON format. With PowerShell, we can easily convert JSON data into something easier to manipulate by using `convertfrom-json` cmdlet, enter:

```
$session.Content | ConvertFrom-Json
```

```
PS C:\Users\team2\Documents> $session.content | ConvertFrom-Json

user                                                    token                                                   sessionID
----                                                    -----                                                   ---------
@{userName=team1; user_roles=System.Object[]; domain=lj.lab} Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6...
```

The result is now structured as a PowerShell custom object, an object that can be manipulated using properties.

Notice the presence of the *token* property, the session token provided by the Global Dashboard API, this is what we are looking for. An easy way to retrieve the *sessionID* is to enter:

```
($session.Content | ConvertFrom-Json).token
```

```
PS C:\Users\team2\Documents> ($session.Content | ConvertFrom-Json).token
Nzc1Mjc5NTk0NDAzxfT7RivXF_i4AT6YJ_ZRCw7iQmA6fGdh
```

Then we can easily capture this *sessionID* token in a variable `$key` for all API call methods we will do later on.

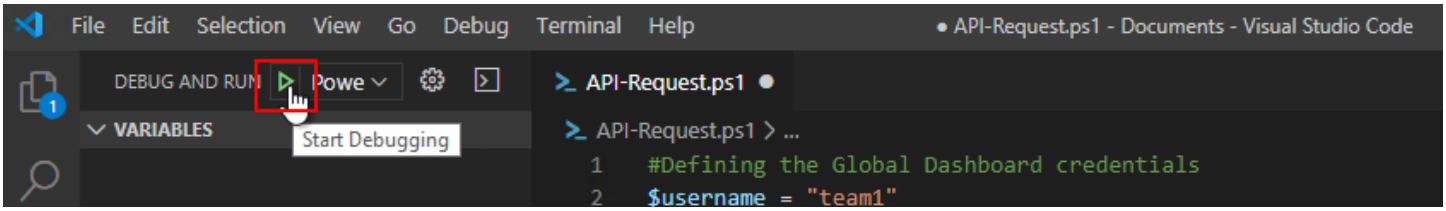
So, replace `$session` at the bottom of the script with the following lines:

```
#Capturing the OneView Global DashBoard Session ID and adding it to the header
$key = ($session.content | ConvertFrom-Json).token
```

```
14  #Opening a login session with Global DashBoard
15  $session = invoke-webrequest -Uri "https://$globaldashboard/rest/login-sessions" -Headers $headers -Body $Body -Method Post
16
17  #Capturing the OneView Global DashBoard Session ID and adding it to the header
18  $key = ($session.content | ConvertFrom-Json).token
19
20
```

Then you can remove the breakpoint by directly clicking on the breakpoint or by using the menu **Debug** / **Remove all Breakpoints**.

Then press **Start Debugging** to run the script again with the two new lines



Now type `$key` in the console and press Enter, it should display the Global DashBoard authentication ID.

```
PS C:\Users\team2\Documents> $key
Mjg5MjUwNjcyNTM0nERXhv60aem9pnHhvW42HrWb4jV2MXr9
```

Then we can use this `$key` to modify our header. If you enter `$headers`, you get two items:

```
PS C:\Users\team2\Documents> $headers

Name                           Value
----                           -----
X-API-Version                  2
content-type                   application/json
```

Let's add the *auth* item to the header with the `$key` value using:

```
$headers["auth"] = $key
```

If you re-enter `$headers`, we see now that *auth* has been added with our *session ID*.

```
PS C:\Users\team2\Documents> $headers["auth"] = $key
PS C:\Users\team2\Documents> $headers

Name                           Value
----                           -----
auth                           Mjg5MjUwNjcyNTM0nERXhv60aem9pnHhvW42HrWb4jV2MXr9
content-type                   application/json
X-API-Version                  2
```

Add `$headers["auth"] = $key` at the bottom of the script:

```
17  #Capturing the OneView Global DashBoard Session ID and adding it to the header
18  $key = ($session.content | ConvertFrom-Json).token
19
20  $headers["auth"] = $key
```

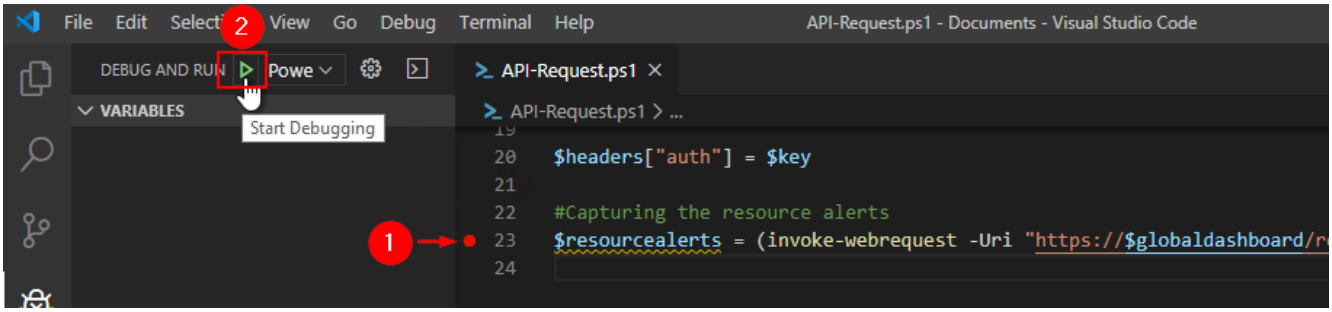
Now every Global Dashboard API call method we do will use this header with the *auth* token for the authentication.

As an example, we can use the `GET /rest/resource-alerts` method to retrieve the critical alerts available in Global Dashboard, we will use once again *invoke-webrequest*, enter now:

```
#Capturing the resource alerts

$resourcealerts = (invoke-webrequest -Uri "https://$globaldashboard/rest/resource-alerts" -Headers
$headers -Method GET) | ConvertFrom-Json
```

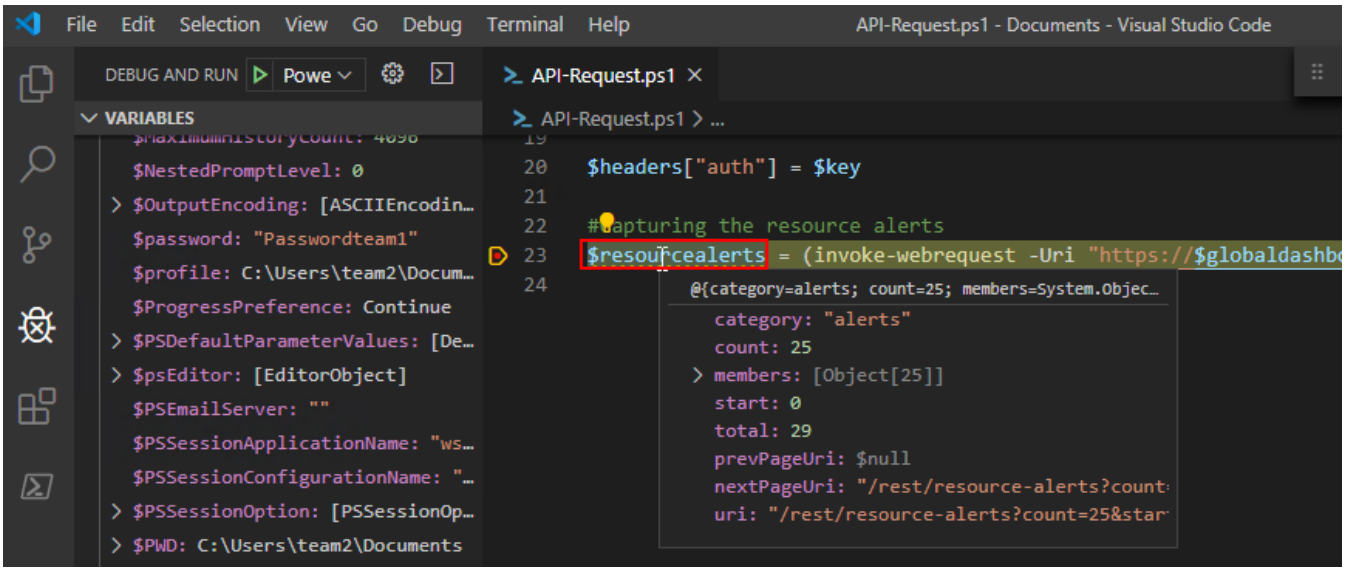
Note: make sure you type or copy/paste this line as a single line

Add a breakpoint on the last line and click on **Start Debugging**



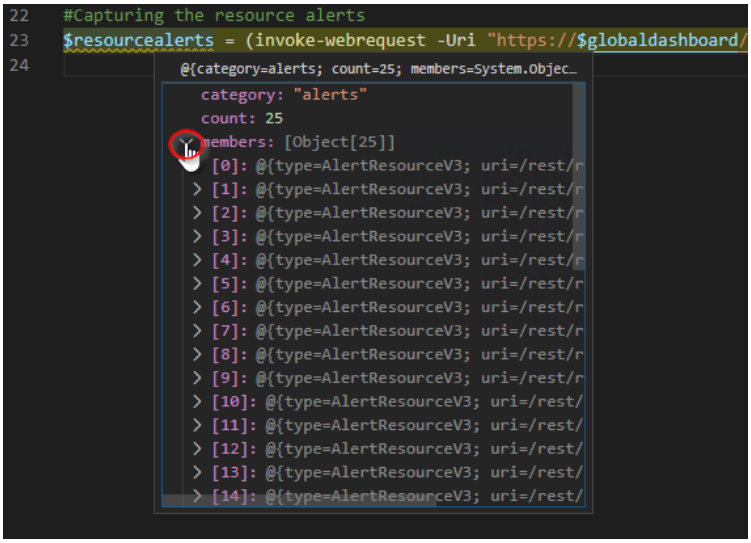
The result of the webrequest is saved in `$resourcealerts`

You can see the content of `$resourcealerts` by hovering your mouse over `$resourcealerts`



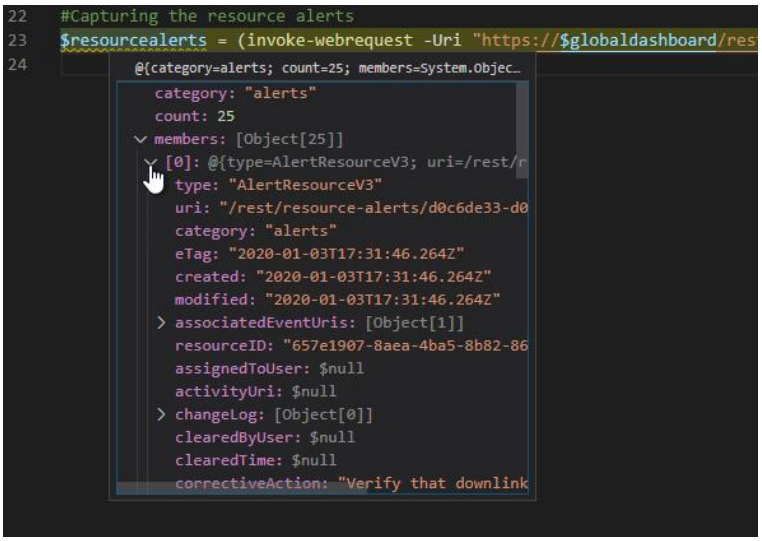
We see that the alerts are grouped into a list, sometimes referred as a collection. This list is called *members*.

To better understand its content, expand *members*

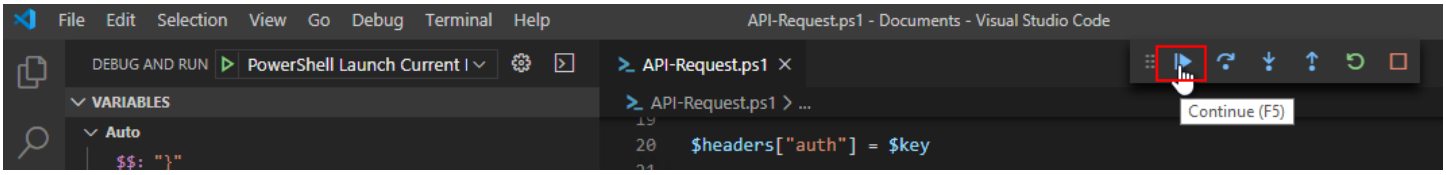


In this list, we have several objects (25), each representing an alert.

Then each object is represented by a paired list of `"items": "value"`



Click now on **Continue**



Another way to see the content information of `$resourcealerts` is to use `get-member` in the console, enter:

```
$resourcealerts | Get-Member
```

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL                    2: PowerShell Integrate

Hit Line breakpoint on 'C:\Users\team2\Documents\API-Request.ps1:23'

[DBG]: PS C:\Users\team2\Documents> $resourcealerts | Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name         MemberType   Definition
----         ----------   ----------
Equals       Method       bool Equals(System.Object obj)
GetHashCode  Method       int GetHashCode()
GetType      Method       type GetType()
ToString     Method       string ToString()
category     NoteProperty string category=alerts
count        NoteProperty int count=25
members      NoteProperty Object[] members=System.Object[]
nextPageUri  NoteProperty string nextPageUri=/rest/resource-alerts?count=25&start=25&query='severity%20EQ%20%22C...
prevPageUri  NoteProperty object prevPageUri=null
start        NoteProperty int start=0
total        NoteProperty int total=29
uri          NoteProperty string uri=/rest/resource-alerts?count=25&start=0&query='severity%20EQ%20%22Critical%22'
```

`Get-member` is a PowerShell function which returns the members, the properties and methods of objects. And as we can see, the result holds several property fields, like *uri, category*, etc. and more importantly the *members* property.

To access the *members* property, it's easy, just enter:

```
$resourcealerts.members
```

```
[DBG]: PS C:\Users\team2\Documents> $resourcealerts.members


type                    : AlertResourceV3
uri                     : /rest/resource-alerts/d0c6de33-d008-40ae-95b7-1537cb2a9465
category                : alerts
eTag                    : 2020-01-03T17:31:46.264Z
created                 : 2020-01-03T17:31:46.264Z
modified                : 2020-01-03T17:31:46.264Z
associatedEventUris     : {/rest/events/1915015}
resourceID              : 657e1907-8aea-4ba5-8b82-86582a3a3254
assignedToUser          :
activityUri             :
changeLog               : {}
clearedByUser           :
clearedTime             :
correctiveAction        : Verify that downlink port is enabled and link status is linked. Verify that the
                          interconnect link topology is valid, the interconnect is in a configured state
                          and there are no network connectivity issues between OneView and the
                          interconnect. If the problem persists, contact your authorized support
                          representative and provide them with a support dump.
description             : An error has occurred on connection 8.  Interconnect {"name":"Frame2,
```

The list can be very long and not easy to read so let's work on methods to get a better presentation.

We can retrieve the members of this object using this time:

```
$resourcealerts.members | Get-Member
```

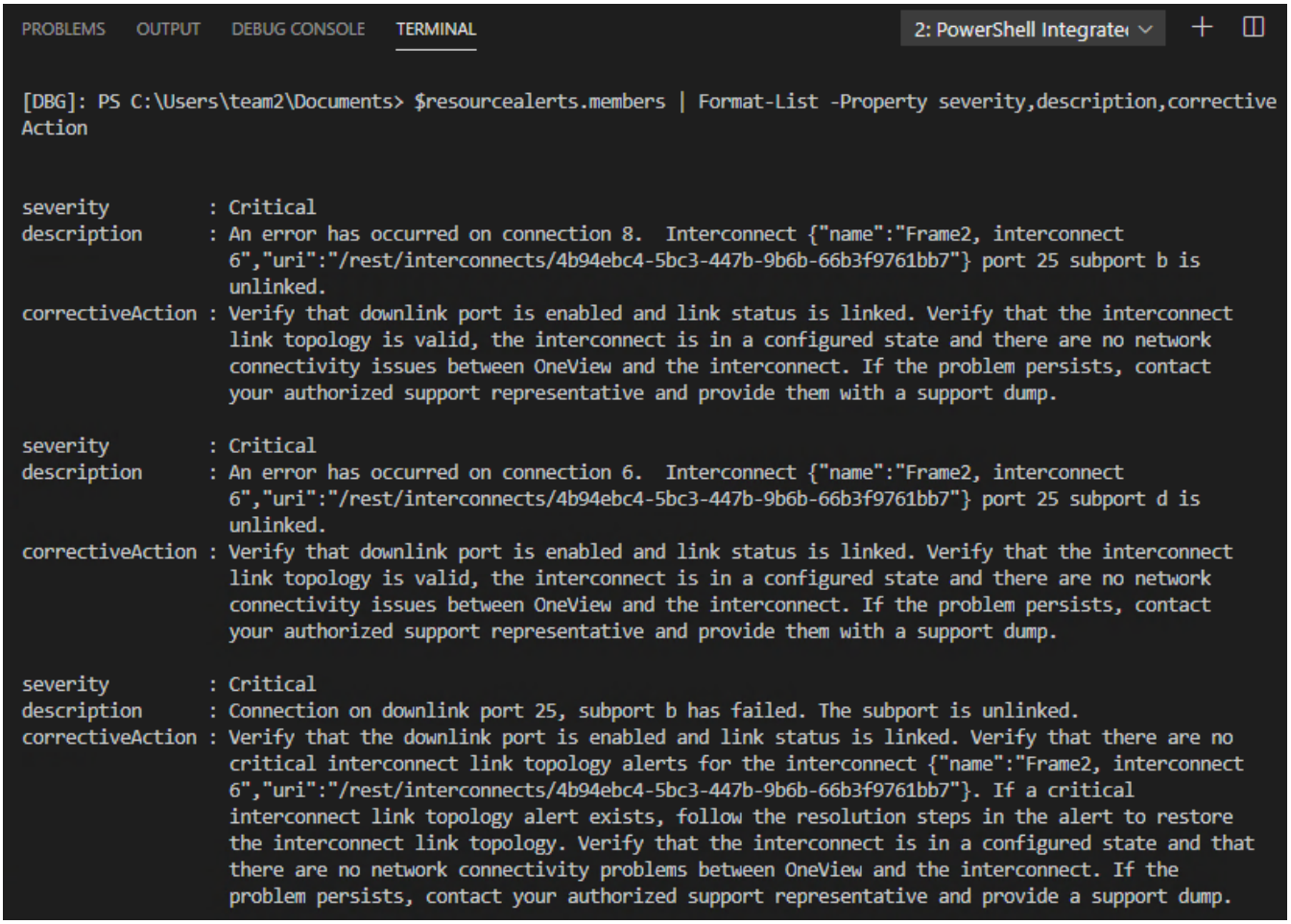
```
[DBG]: PS C:\Users\team2\Documents> $resourcealerts.members | Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name                        MemberType   Definition
----                        ----------   ----------
Equals                      Method       bool Equals(System.Object obj)
GetHashCode                 Method       int GetHashCode()
GetType                     Method       type GetType()
ToString                    Method       string ToString()
activityUri                 NoteProperty object activityUri=null
alertState                  NoteProperty string alertState=Locked
alertTypeID                 NoteProperty string alertTypeID=profilemgr.Connections.CONNECTION_SCMB_ERROR
applianceLocation           NoteProperty string applianceLocation=192.168.1.110
applianceModel              NoteProperty string applianceModel=Synergy Composer
applianceName               NoteProperty string applianceName=composer
applianceVersion            NoteProperty string applianceVersion=5.00.01-0410269
appluri                     NoteProperty string appluri=/rest/appliances/ec4638ac-5cdb-4baa-b0ed-2fb90dd1664c
assignedToUser              NoteProperty object assignedToUser=null
associatedEventUris         NoteProperty Object[] associatedEventUris=System.Object[]
associatedResource          NoteProperty System.Management.Automation.PSCustomObject associatedResource=@{res...
associatedResourceName      NoteProperty string associatedResourceName=win-1
associatedResourceOriginalUri NoteProperty string associatedResourceOriginalUri=/rest/server-profiles/a42f4b31-...
category                    NoteProperty string category=alerts
changeLog                   NoteProperty Object[] changeLog=System.Object[]
clearedByUser               NoteProperty object clearedByUser=null
clearedTime                 NoteProperty object clearedTime=null
correctiveAction            NoteProperty string correctiveAction=Verify that downlink port is enabled and lin...
created                     NoteProperty string created=2020-01-03T17:31:46.264Z
description                 NoteProperty string description=An error has occurred on connection 8.  Interconn...
eTag                        NoteProperty string eTag=2020-01-03T17:31:46.264Z
healthCategory              NoteProperty string healthCategory=Connection
id                          NoteProperty string id=d0c6de33-d008-40ae-95b7-1537cb2a9465
lifeCycle                   NoteProperty bool lifeCycle=False
modified                    NoteProperty string modified=2020-01-03T17:31:46.264Z
originalUri                 NoteProperty string originalUri=/rest/alerts/27183
physicalResourceType        NoteProperty string physicalResourceType=server-profiles
resourceID                  NoteProperty string resourceID=657e1907-8aea-4ba5-8b82-86582a3a3254
resourceUri                 NoteProperty string resourceUri=/rest/server-profiles/a42f4b31-ca97-44b8-8b72-4c7...
serviceEventDetails         NoteProperty object serviceEventDetails=null
serviceEventSource          NoteProperty bool serviceEventSource=False
severity                    NoteProperty string severity=Critical
status                      NoteProperty string status=Unknown
type                        NoteProperty string type=AlertResourceV3
urgency                     NoteProperty string urgency=High
uri                         NoteProperty string uri=/rest/resource-alerts/d0c6de33-d008-40ae-95b7-1537cb2a9465
```

These properties highlighted above are all the properties available for this object, some are very useful for what we are looking for, some others are not.

So, let's work on the formatting of this object to display only the important information we need. We are going to use the cmdlet *Format-List* to display only the `severity`, `description` and `correctiveAction` properties, enter in the console:

```
$resourcealerts.members | Format-List -Property severity,description,correctiveAction
```

The display result is now much better and easier to read:



So finally, add to your script the following lines to complete the script.:

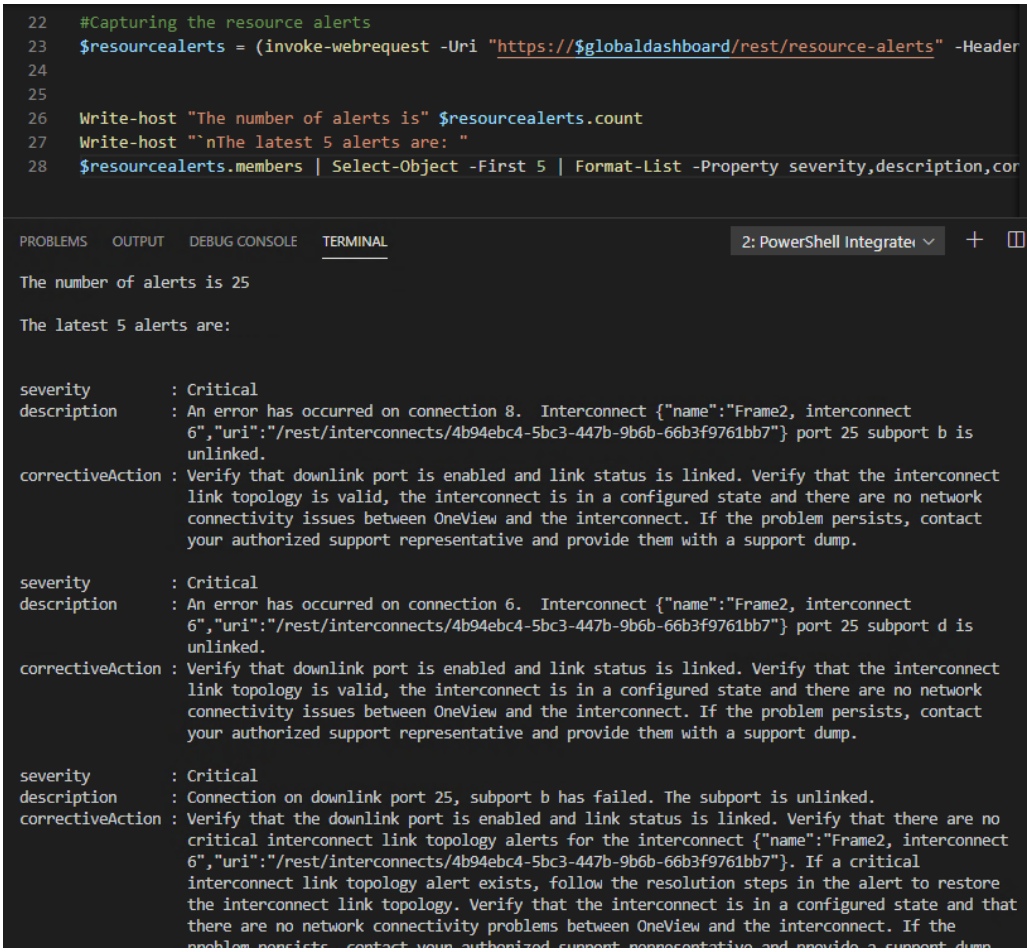
```
Write-host "The number of alerts is" $resourcealerts.count

Write-host "`nThe latest 5 alerts are: "

$resourcealerts.members | Select-Object -First 5 | Format-List -Property
severity,description,correctiveAction
```

The cmdlet *Select-Object* is used to only display the first 5 alerts.

> **Note**: make sure you type or copy/paste this last line as a single line in VS Code

> **Note**: `n is used to insert a space (a newline) between each alert for better clarity

Once the three lines are added, you can remove all breakpoints you may have and re-run the script to see the result:



This concludes the PowerShell Lab.

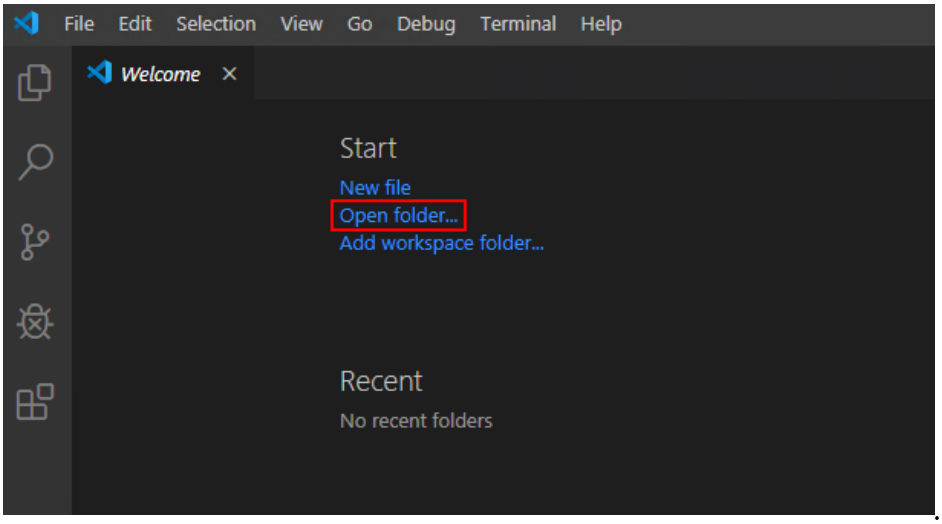
Python

A Python library for HPE OneView is available from HPE, it provides basic functions to manage the appliance. We are not going to use this library in this lab as we want you to understand the way a REST call can be realized in Python. Besides that, a library cannot provide all REST calls provided by an API, so it is essential to know how to pass a REST request natively in Python.

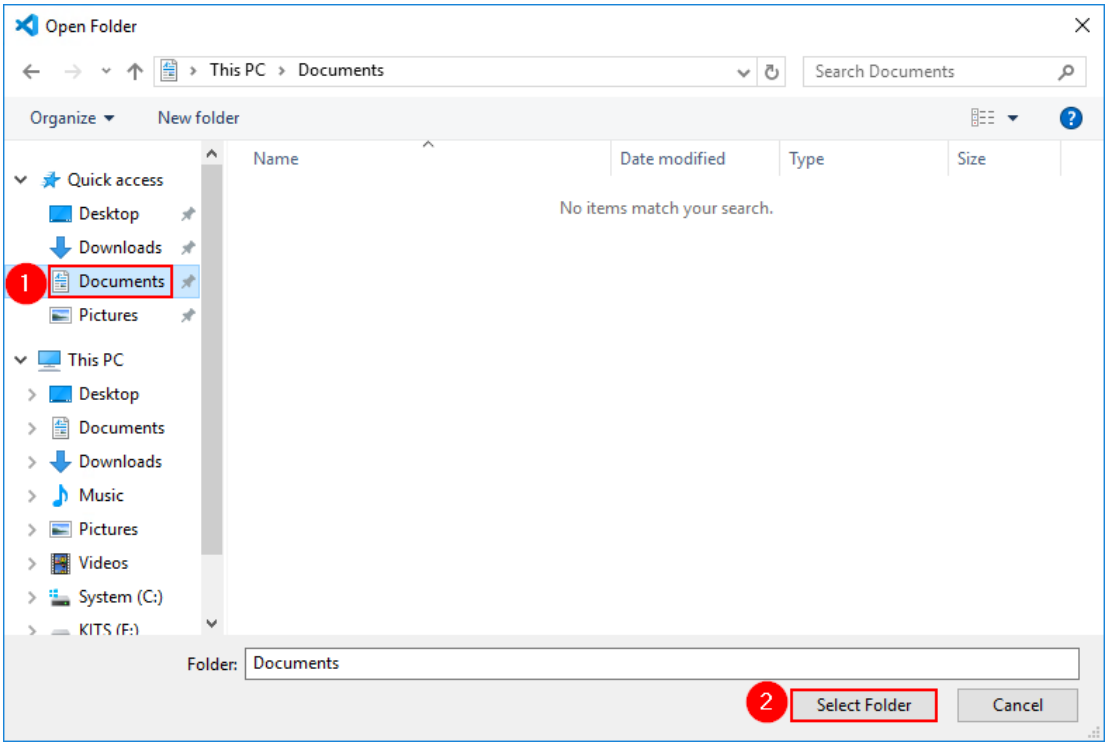
For more information about the HPE OneView Python library, see https://github.com/HewlettPackard/python-hpOneView

Start Visual Studio Code using the shortcut on the desktop:

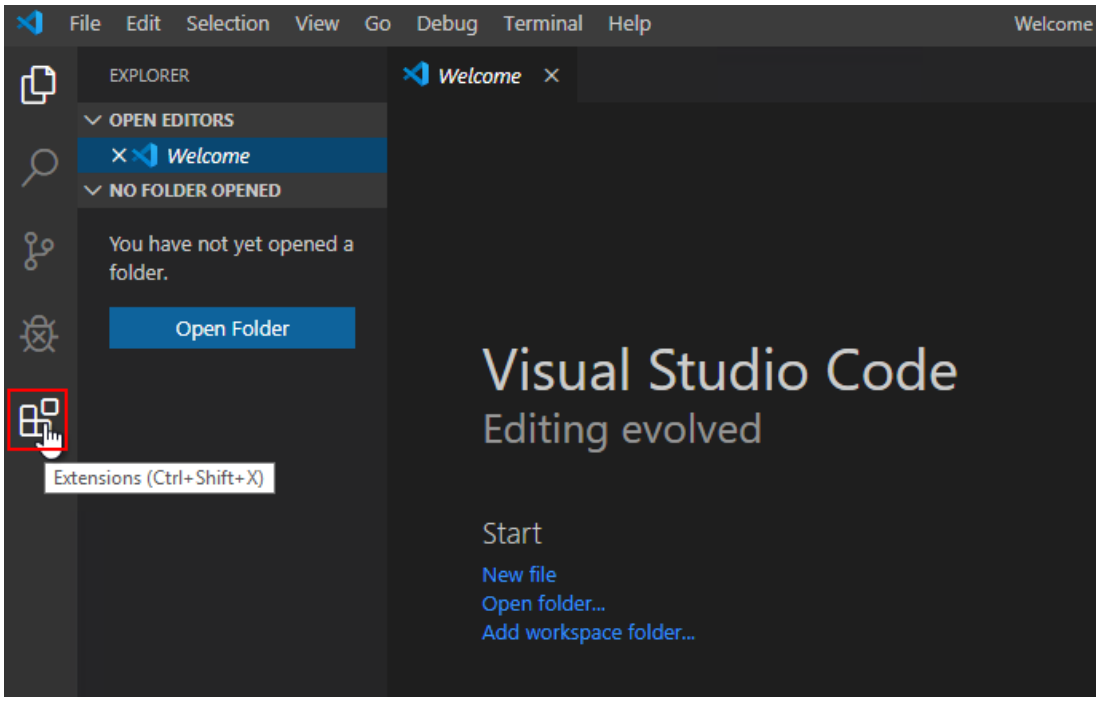
Click on **Open Folder...**



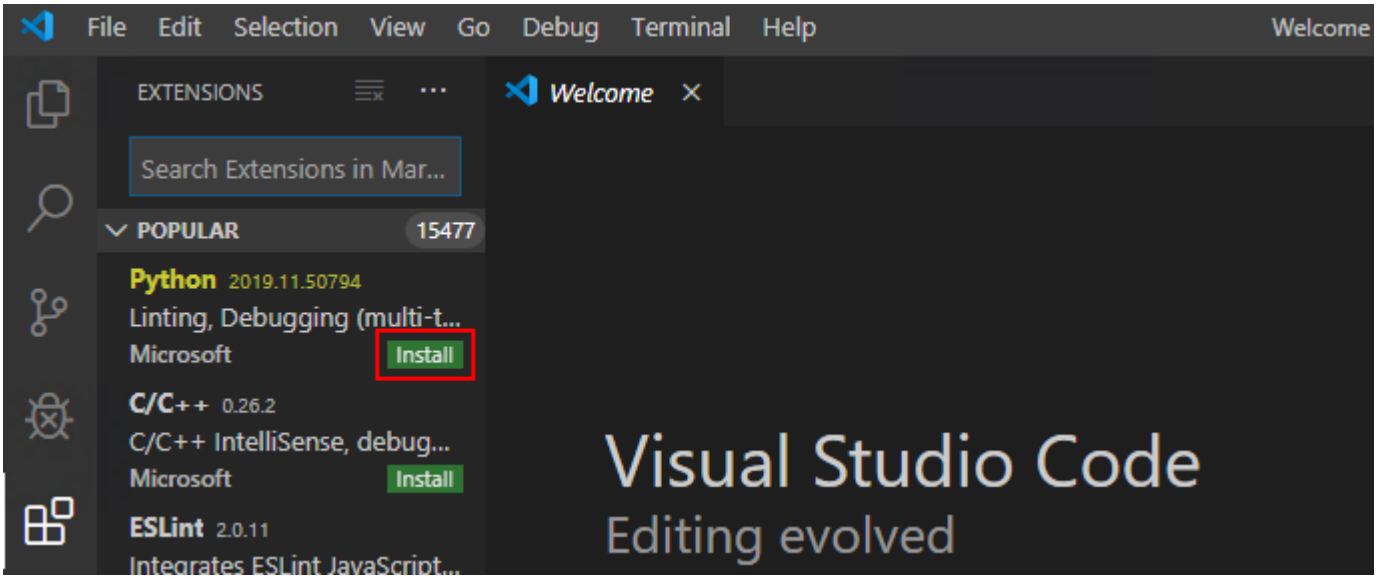
And select **Documents** and click **Select Folder**



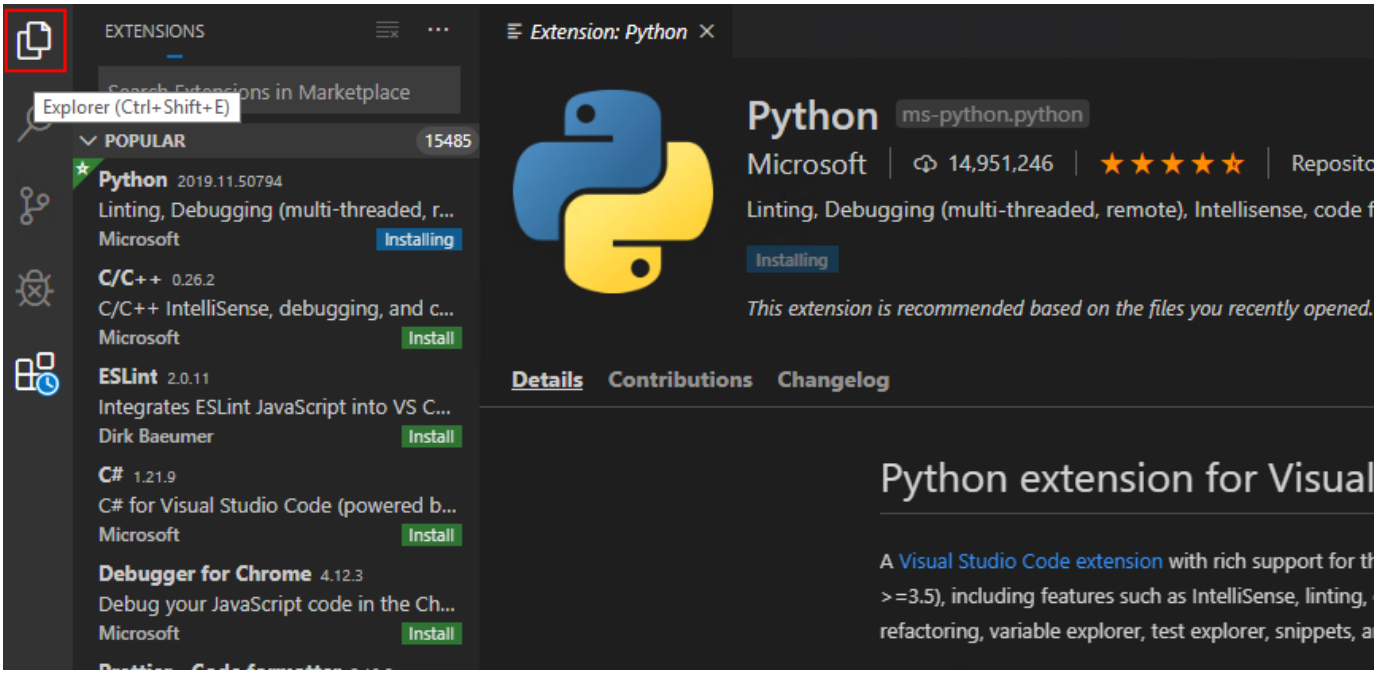
Then click on **Extensions** on the Activity bar



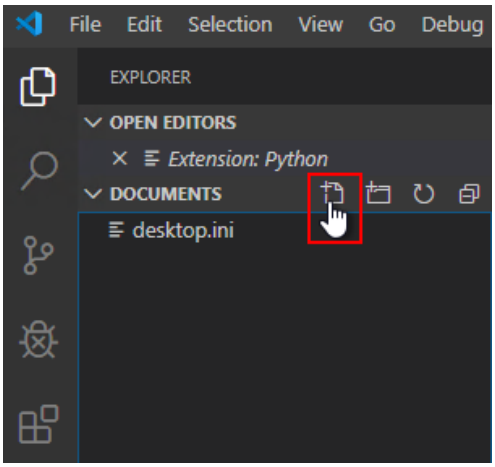
If not already installed, install the *Python* extension usually listed at the top of the list. You can enter **python** in the search tool to find the extension. Click on the **Install** button:



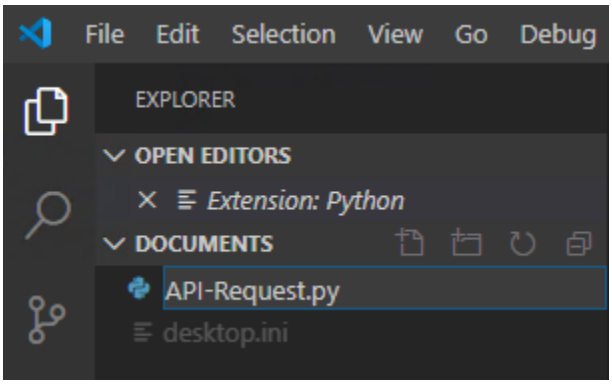
While the extension is getting installed, click on **Explorer** on the Activity Bar



Then click on **New File**

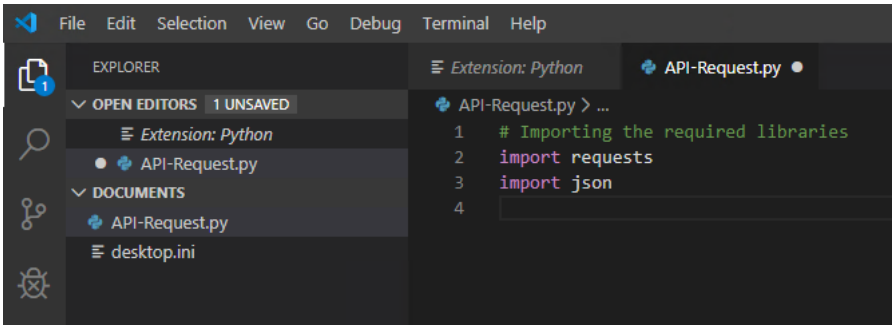


Enter a name like **API-Request.py** then press **ENTER**



You can then enter the follow-on lines to build your Python script.
The script we are going to use starts as follows, just copy/paste these lines in VS Code:

```
# Importing the required libraries
import requests
import json
```



The `requests` function is imported at the beginning of the script from the Python library *Requests* used to send HTTP/HTTPS/FTP and FILE request to a web page or web service. We will use it for a HTTPS request to our Global Dashboard REST API using a `POST` method on `\rest\login-sessions`. The *URL*, *headers*, *body* are provided as well as parameters.

A second function `json` is also imported, we will use it to convert the Global DashBoard json response into a python dictionary, much easier to manipulate.

Next, we are defining our Global DashBoard credentials in variables

```
# Defining the Global Dashboard credentials
username = "<username>"
password = "<password>"
globaldashboard = "oneview-global-dashboard.lj.lab"
```

You need to update <username> and <password> with your provided credentials.

Next, enter the following lines to create the body and header content:

```
# Creating the body content
body = {
              'userName': username,
              'password': password,
              'authLoginDomain': 'lj.lab'
       }

# Creating the header content
header= {
              'Content-Type': 'application/json',
              'X-API-VERSION': '2'
        }
```

The script just follows the same steps we saw previously with Postman: the creation of the body for the login session (with *userName/password/authLoginDomain*) with the creation of the header (with *content-type* and *x-api-version*)

Next, we can finally create the API query using the Python *requests* function using the *POST* action with our body, header and URL then we display on the console the *response* content:

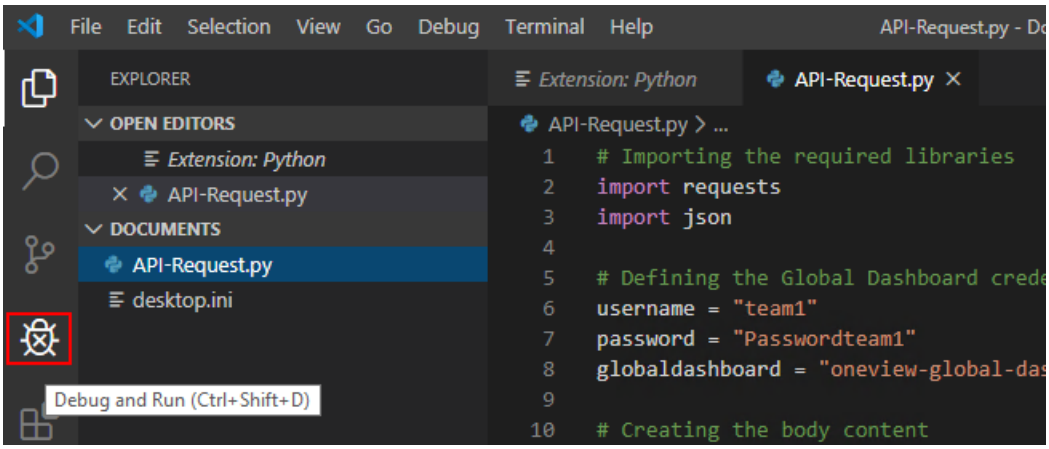
```
# Creating the web request with body and header content
response = requests.post(
        url="https://" + globaldashboard + "/rest/login-sessions",
        verify=False,
        json=body,
        headers=header
        )

print(response)
```

Your VS Code script should look like this:

```python
# Importing the required libraries
import requests
import json

# Defining the Global Dashboard credentials
username = "team1"
password = "Passwordteam1"
globaldashboard = "oneview-global-dashboard.lj.lab"

# Creating the body content
body = {
        'userName': username,
        'password': password,
        'authLoginDomain': 'lj.lab'
       }

# Creating the header content
header= {
        'Content-Type': 'application/json',
        'X-API-VERSION': '2'
        }

# Creating the web request with body and header content
response = requests.post(
        url="https://" + globaldashboard + "/rest/login-sessions",
        verify=False,
        json=body,
        headers=header
        )

print(response)
```

You can press **CTRL** + **S** to save the file.

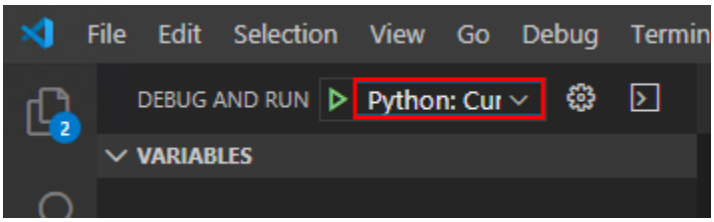
Now let's run the script, click on **Debug and Run** on the Activity Bar



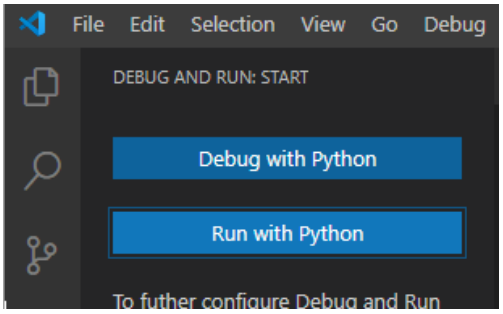
One of the key features of Visual Studio Code is its great debugging support. VS Code's built-in debugger helps accelerate edit, compile and debug script.

Visual Studio Code's built-in debugger needs some quick setup to run and debug Python scripts. Please follow the procedure that applies to your environment:

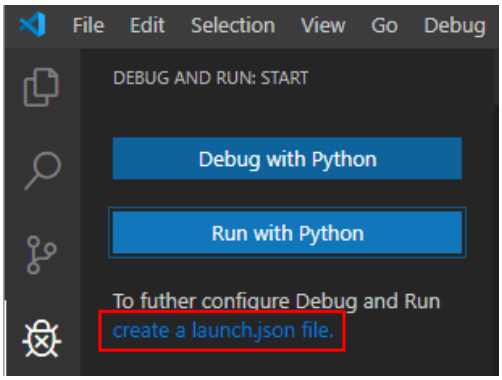
1. If *Python* is displayed in the Debug and Run dialog, your environment is all set for Python scripting, you can skip the setup section and step over points 2. and 3.:



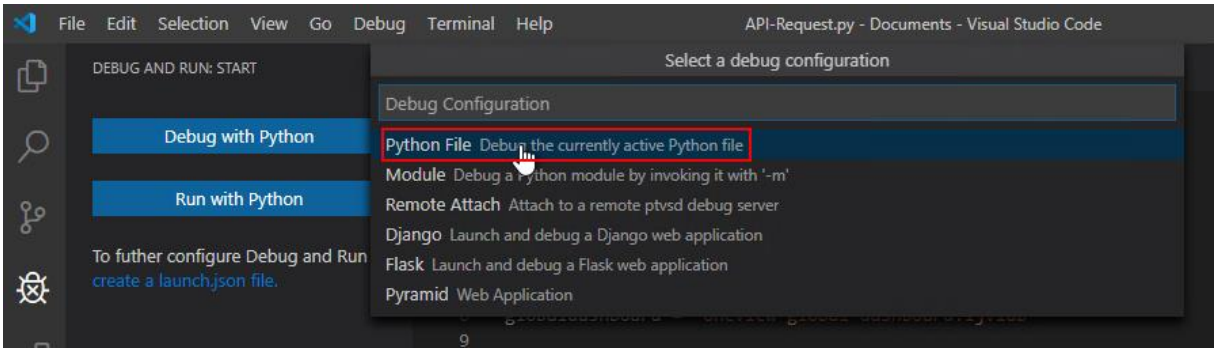
2. If the following dialog is displayed in the Debug and Run pane:



   a. Click on **Create a launch.json file** to configure Debug and Run



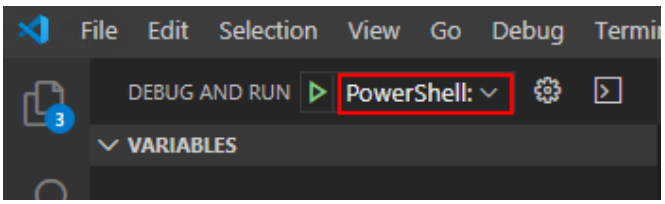
b. Select the first option **Python File**:



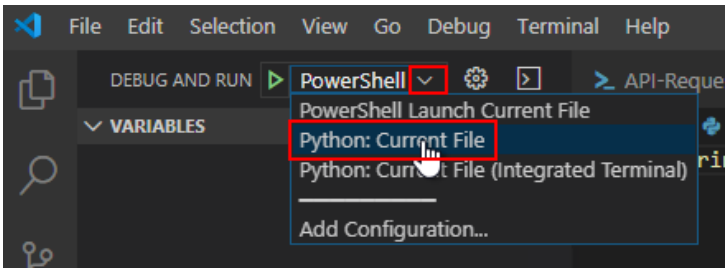
c. Press **CTRL + S** to save the configuration file

d. Press **CTRL + F4** to close the file

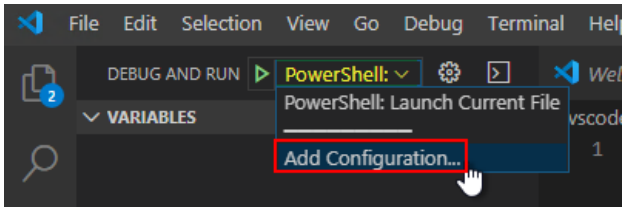
3. If *PowerShell* is displayed in the Debug and Run dialog:



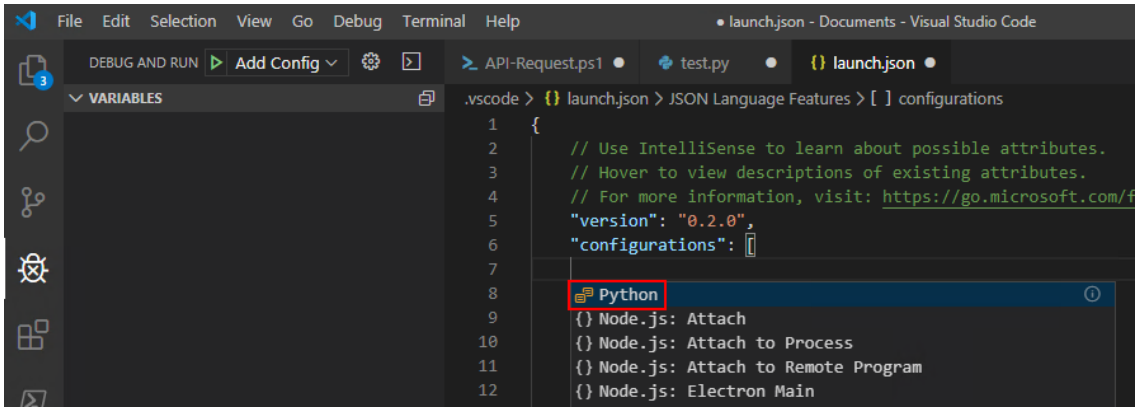
a. Select **Python: Current File** from the Debug and Run drop-down menu and you are done.



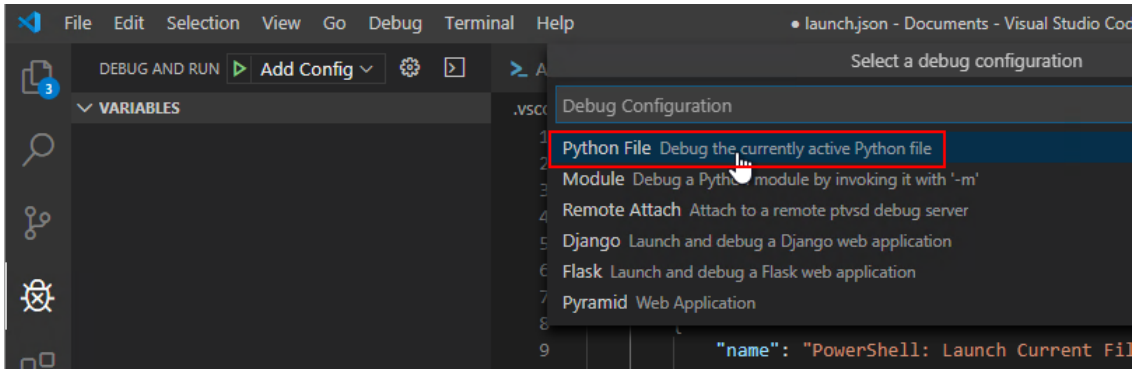
i. If there is no Python configuration, you need to select **Add configuration** from the drop-down menu:



ii. Select **Python**



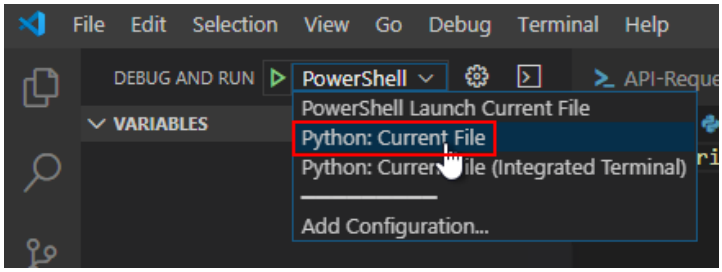
iii. Then select **Python File**



iv. Press **CTRL + S** to save the configuration file

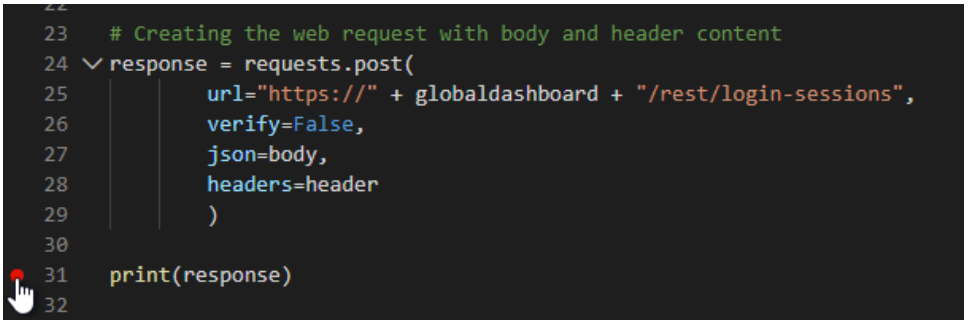
v. Press **CTRL + F4** to close the file

vi. Then select **Python: Current File** from the Debug and Run drop-down menu

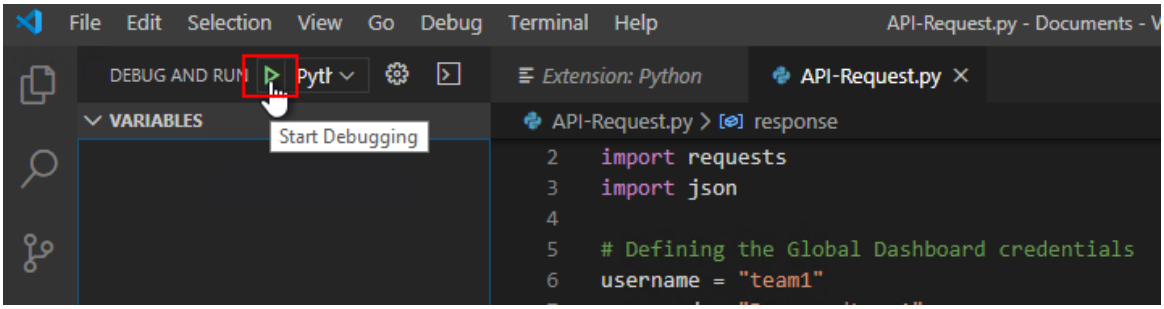


The Visual Studio Code debug and run setup is now complete, we can start debugging our script.

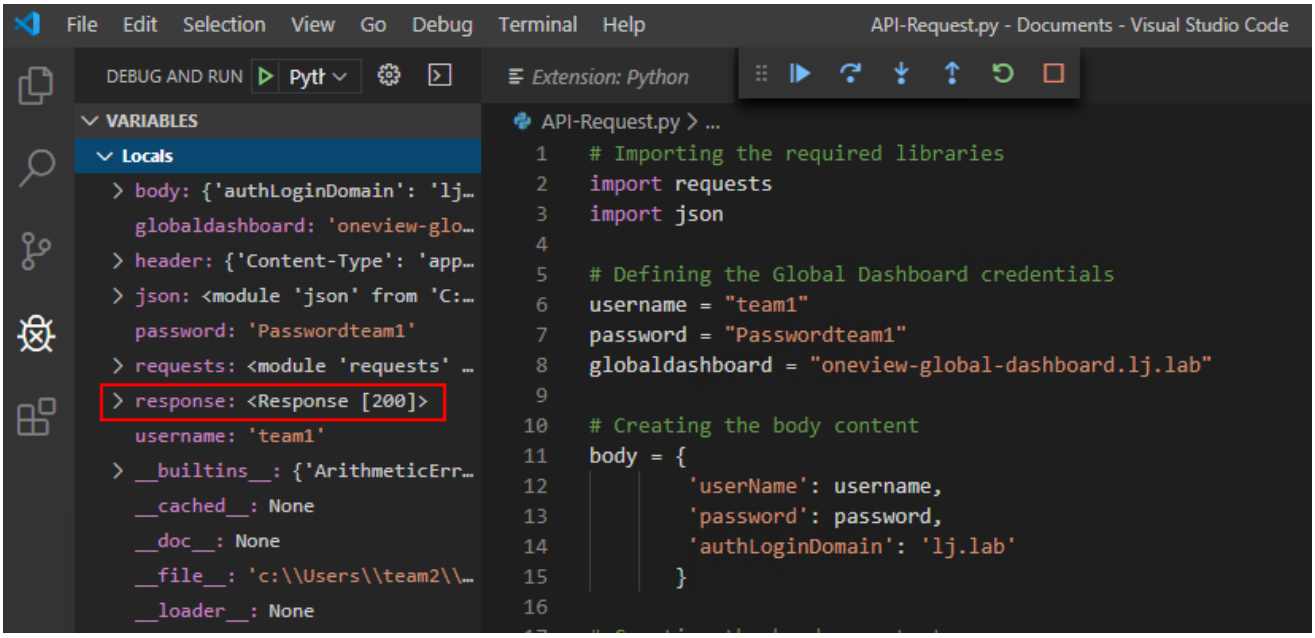
Using your mouse, drop a Breakpoint on the `print(response)` line. Breakpoints are normally shown as red filled circles, can be toggled by clicking on the editor margin:



Then click on **Start Debugging** to run the script

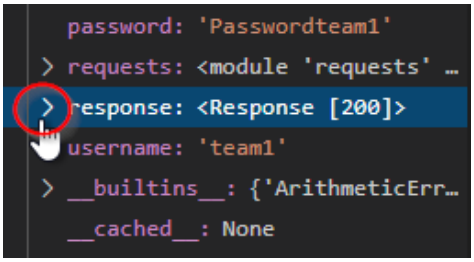


On the left-hand pane, the variables content is displayed, a successful API request response code **200** should be displayed for the *response* variable:

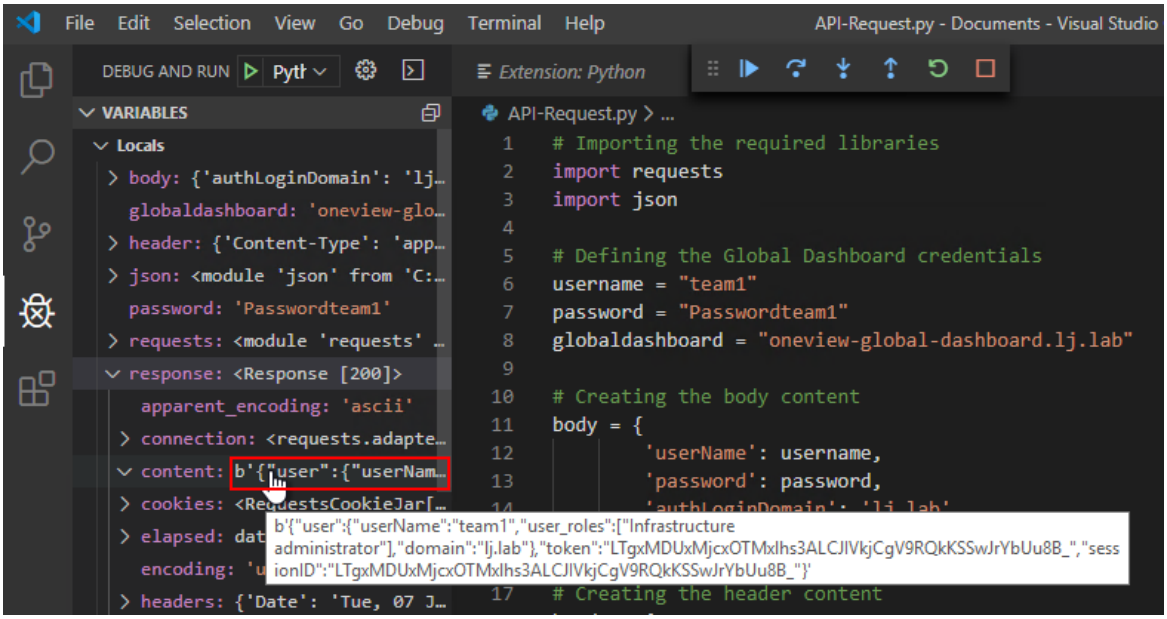


`response` is the variable we are using to store the result of `requests.post`

You can discover the content of `response` by using the expand icon:



If you hover your mouse over the `content` description you will get the following popup box:

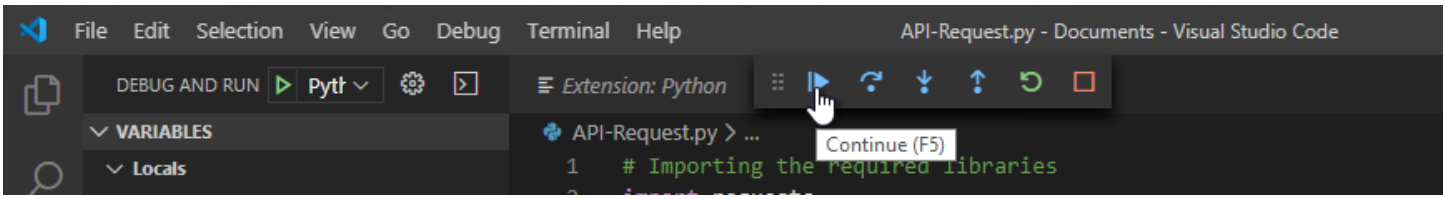


This is the response received from the API

b'{"user":{"userName":"team1","user_roles":["Infrastructure administrator"],"domain":"lj.lab"},"token":"LTgxMDUxMjcxOTMxIhs3ALCJIVkjCgV9RQkKSSwJrYbUu8B_","sessionID":"LTgxMDUxMjcxOTMxIhs3ALCJIVkjCgV9RQkKSSwJrYbUu8B_"}'

Notice that the content of *content* is in JSON format. So, to access to the *sessionID*, we need to convert the response into a Python dictionary, easier to manipulate.

Click on **Continue** or **F5** to continue the script execution.



Then you can remove the breakpoint by directly clicking on the breakpoint or by using the menu **Debug** / **Remove all Breakpoints**.

So, to convert the JSON response into a Python dictionary, we are going to use the *json* library, imported at the beginning of our script. A Python dictionary is a mapping of unique keys to values that can be easily manipulated and accessed.

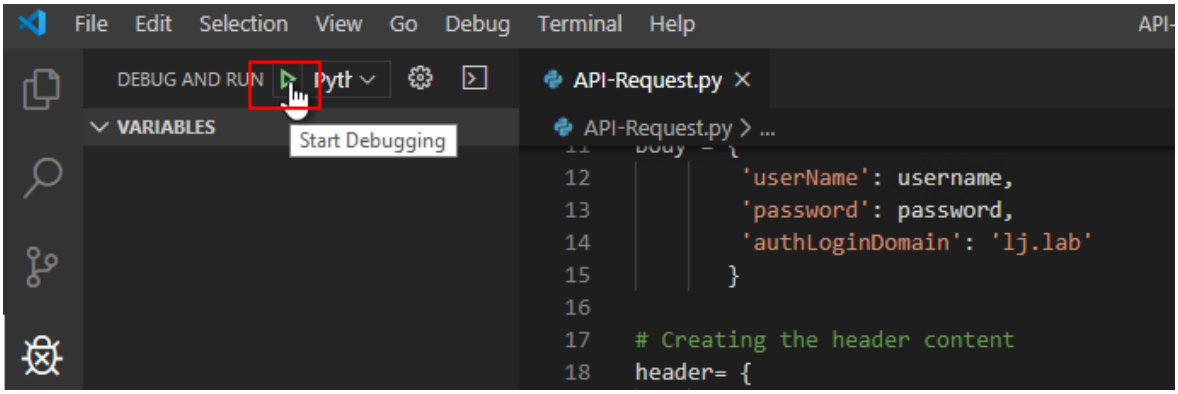
Add the following lines at the end of the script:

```
loginsessionresponse = json.loads(response.content)
print(json.dumps(loginsessionresponse, indent=4))
```

The first line converts the content of `response.content` into a dictionary. The second line is used to display `loginsessionresponse` in a nice format on the console:

```
23  # Creating the web request with body and header content
24  response = requests.post(
25          url="https://" + globaldashboard + "/rest/login-sessions",
26          verify=False,
27          json=body,
28          headers=header
29          )
30
31  print(response)
32
33  loginsessionresponse = json.loads(response.content)
34
35  print(json.dumps(loginsessionresponse, indent=4))
36
```

Then press **Start Debugging** to run the script again with the two new lines



On the terminal window, you should see the content of `loginsessionresponse` structured as a Python dictionary:

Notice the presence of the *sessionID* property, the session token provided by the Global Dashboard API, this is what we are looking for.

An easy way to retrieve the *sessionID* from the dictionary is to use `loginsessionresponse['sessionID']`

So, add the next line in your script to save the Global Dashboard session ID into the variable `sessionID`:

```
sessionid = loginsessionresponse['sessionID']
```

Now every Global Dashboard API call method we do will use this `sessionID` token for the authentication.

Let's for instance get all resource alerts in Global DashBoard, we can add the following lines:

```
response_alerts = requests.get(
    url="https://" + globaldashboard + "/rest/resource-alerts",
    verify=False,
    headers={
           'Content-Type': 'application/json',
           'X-API-VERSION': '2',
           'auth': sessionid
          }
    )
```

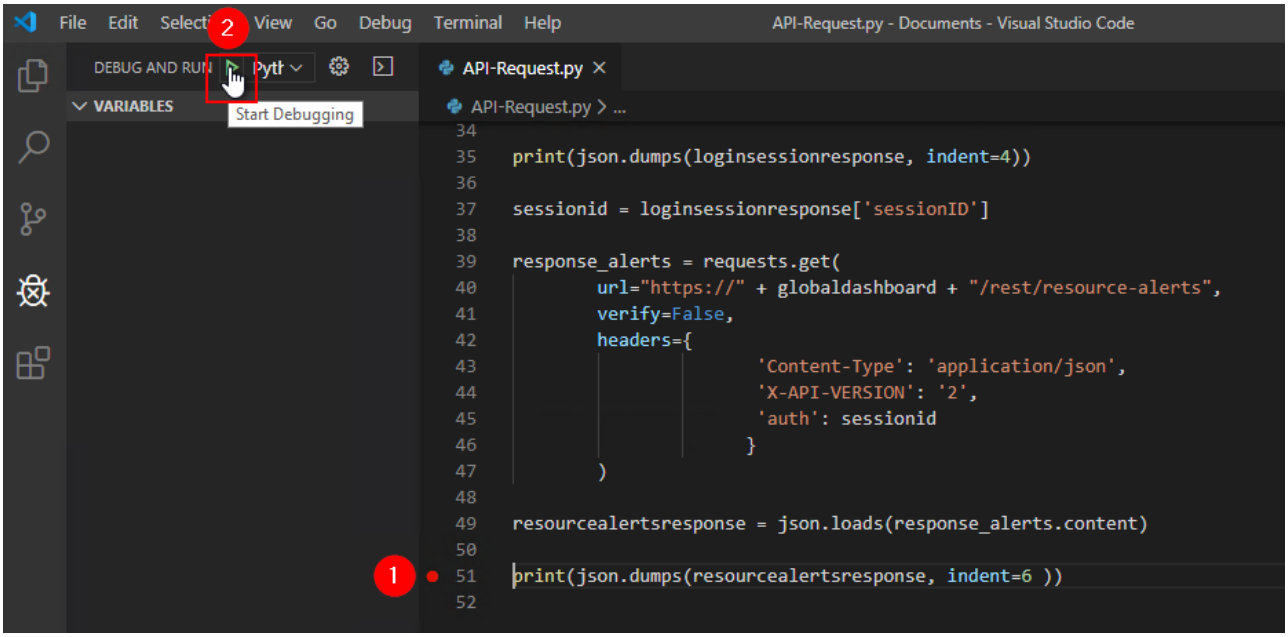
We use the variable `sessionid` to store the value of the authentication token. Then we use it in the header with `auth`. We also use the `verify=False` option to avoid a certificate verification failure as we are using a self-signed certificate in Global Dashboard.

In this example, we use the `GET` method on `/rest/resource-alerts` to retrieve the critical alerts available in Global Dashboard.

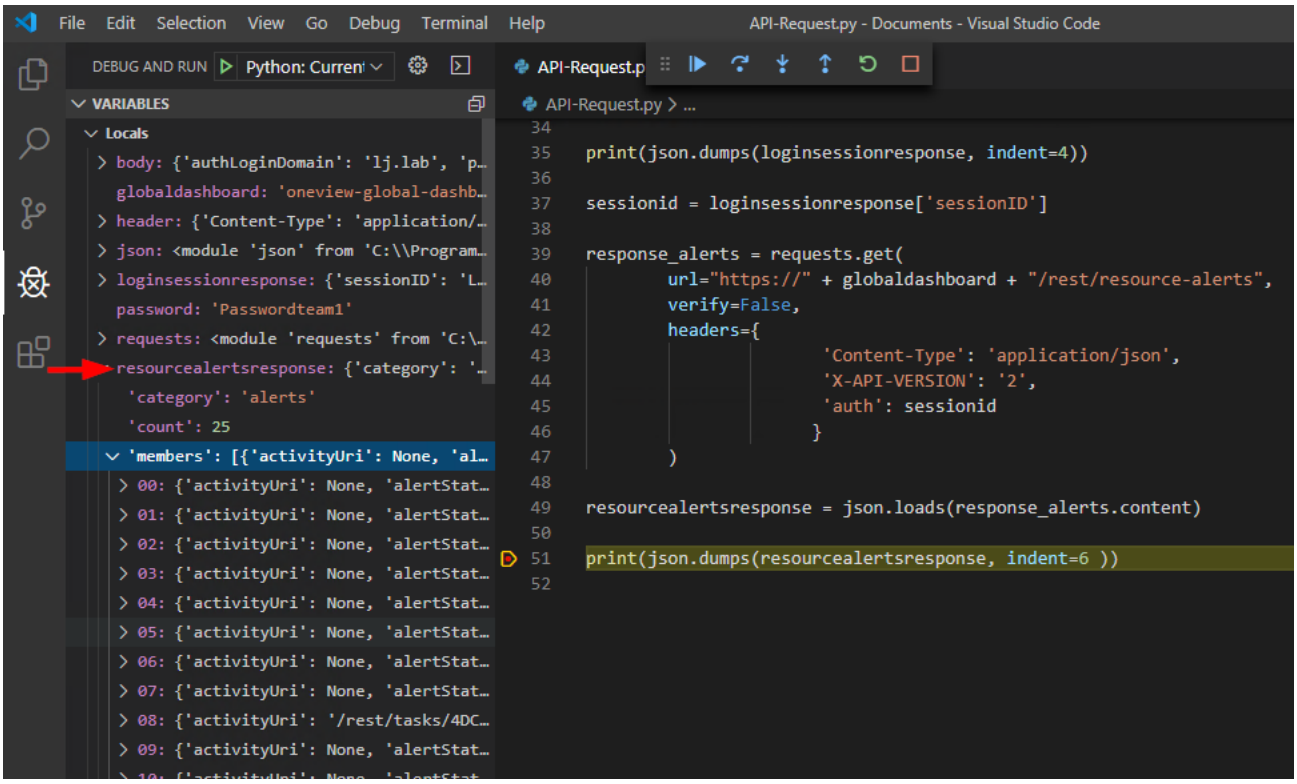
Then to display the list of alerts, we can use the same previous *json()* and *print()* methods:

```
resourcealertsresponse = json.loads(response_alerts.content)
print(json.dumps(resourcealertsresponse, indent=6))
```

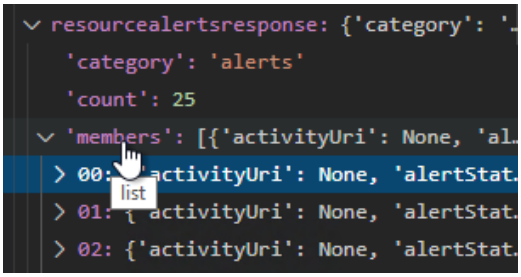
Now put a breakpoint on the last line and click **Run**



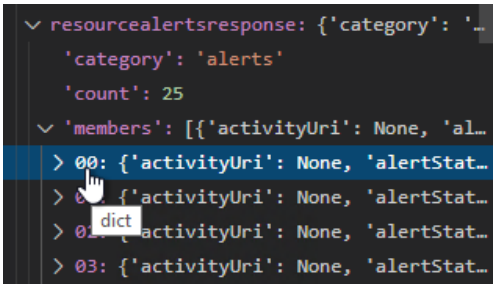
Then look for the `resourcealertsresponse` variable in the left-hand debug pane to understand its content:



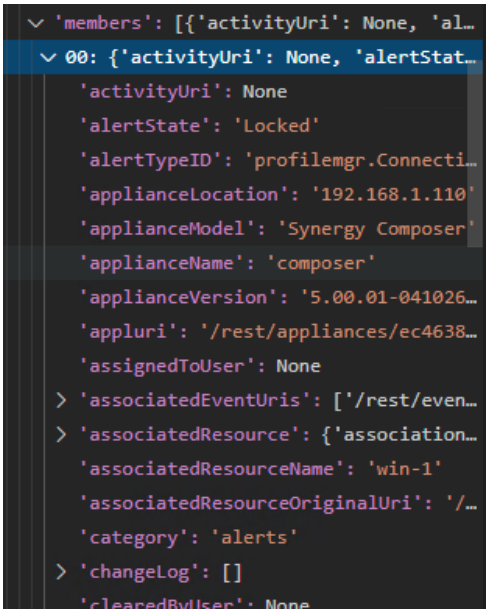
We see that the alerts are grouped into a list, sometimes referred as a collection. This list is called *members*.



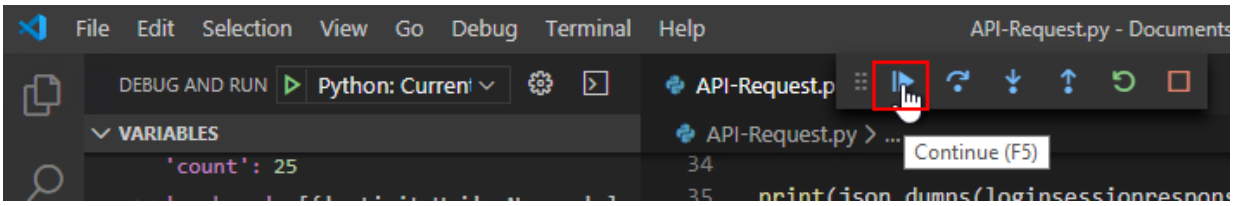
In this list, we have several dictionaries (25), each representing an alert.



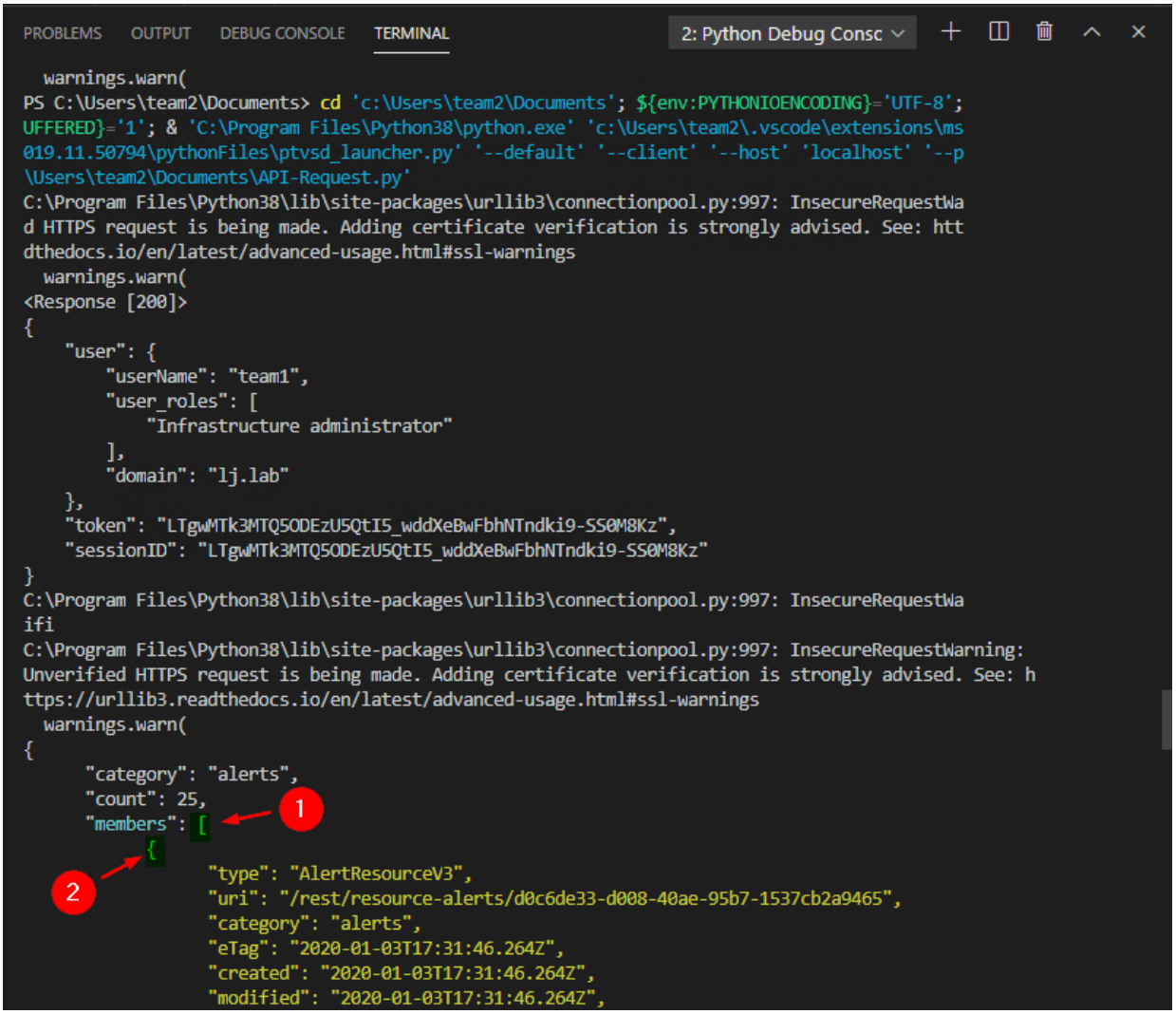
Then each dictionary is represented by a paired list of `“items”: “value”`



Click now on **Continue**



The alerts are displayed on the terminal window as requested in the script. Now scroll up to the beginning of the alert



The list here is identified by the [...] (1) and the dictionary by the {...} (2)

So, we can reduce the result to the alerts only by printing only the *member* list:

```
alerts = resourcealertsresponse['members']
```

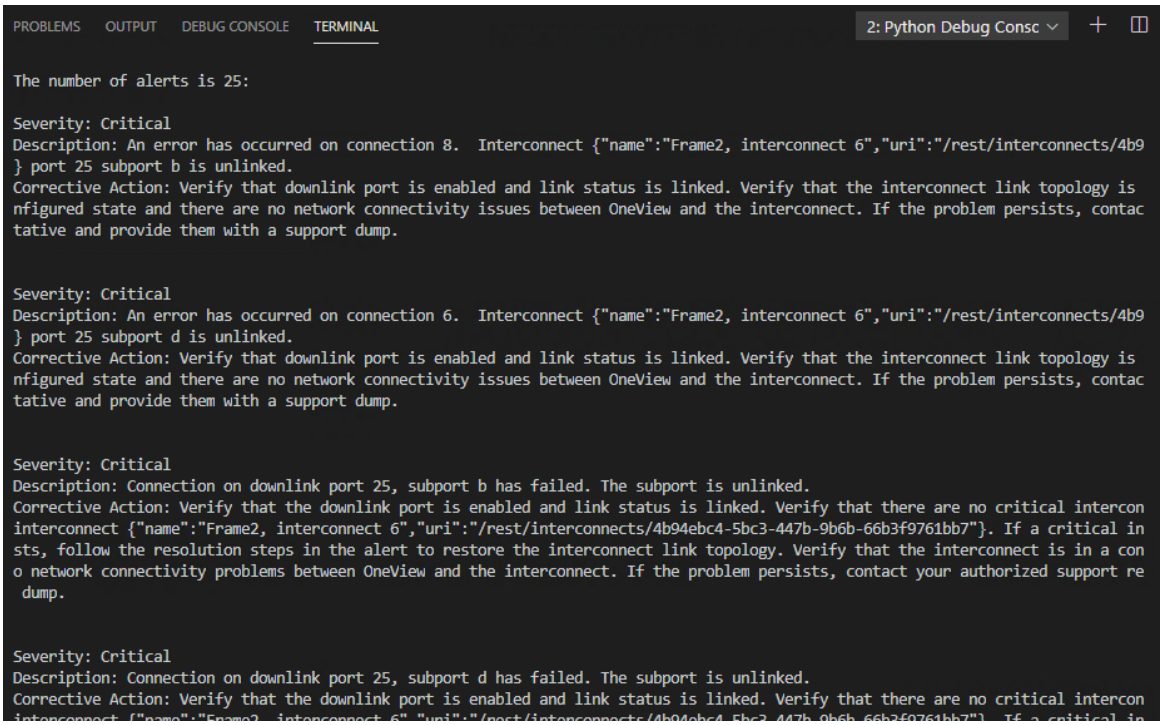
And finally, to display the number of alerts and the alerts in a nice format, you can start by removing all `print()` functions (or commenting each line with a #) in the script and enter the following commands:

```
print("\n\nThe number of alerts is " + str(len(alerts)) + ":")

for alert in alerts:
  print("\nSeverity: " + alert['severity'])
  print("Description: " + alert['description'])
  print("Corrective Action: " + alert['correctiveAction'])
```

> **WARNING**: do not copy/paste the 4 lines above from the PDF version of this guide as this will lose the tab indentation that Python relies on for its syntax. You must copy/paste each line of the loop at a time.

> **Note**: `\n` is used to insert a space (a newline) between each alert for better clarity

This loop is used to display all alerts in the list but using only the `severity`, `description` and `correctiveAction` fields as illustrated below:



Note: *members* is a list so we can get the first alert by entering `[0]`:

```
print(json.dumps(alerts[0], indent=6))
```

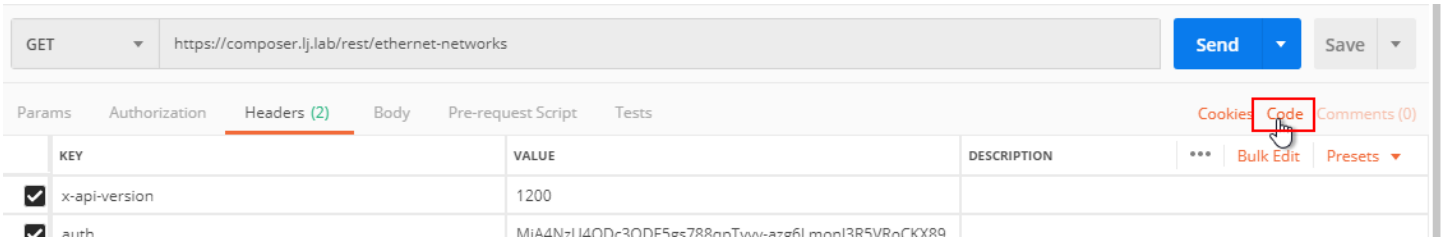
or `[-1]` to get the last one:

```
print(json.dumps(alerts[-1], indent=6))
```

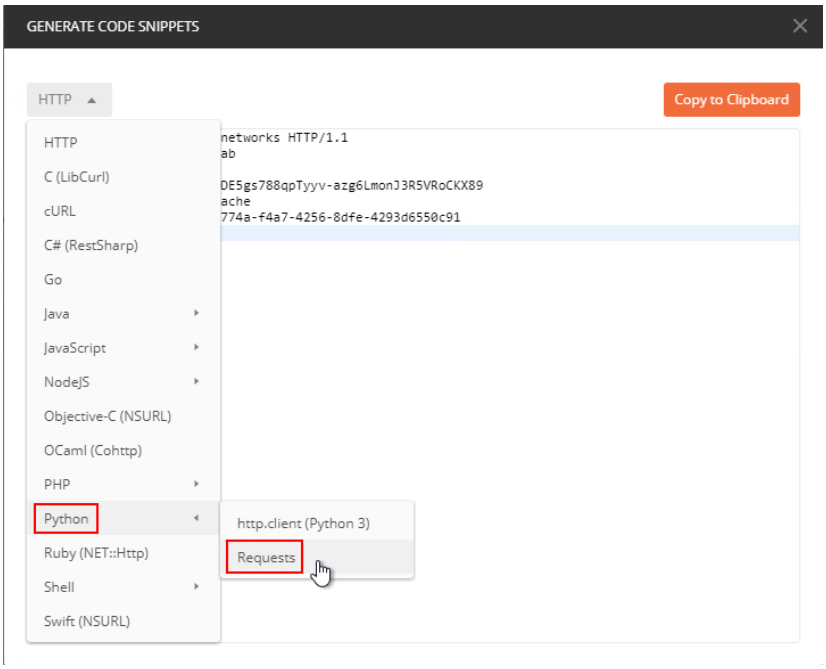
Just for information

One of the cool features available in Postman is the Generate code snippets. This feature allows you generate snippets of code in various languages and frameworks, including Python.

So, after having built an API request, you can simply click on **Code** under the blue Send button to open the Generate code snippets modal.



Use the dropdown menu to select a language - some languages have multiple options. This lets you select different frameworks from which to make your request.



This is an example of the Python code generated for a `get /rest/ethernet-networks` call:

This code looks familiar, doesn't it?

This concludes the Python Lab.

Summary

During this lab, you have been introduced to the HPE OneView and HPE OneView Global Dashboard RESTful API. You have learned about REST API concepts, how to login and start to leverage some of the API methods, how to find and decrypt the right resource method and how the OneView resource model is built. You also learned how to use Postman and how to decrypt JSON responses. Finally, you discovered how to use PowerShell and Python to send web request tasks against the REST API, how to translate the JSON response and how to manipulate objects.

If you want to go further and automate your infrastructure management and eliminate complex manual processes, I would recommend you to visit the HPE OneView Composable Infrastructure Ecosystem, see https://www.hpe.com/us/en/solutions/developers/composable.html