When should Automated API testing be performed?

Automated API Testing should be performed ideally throughout the SDLC life cycle in from early project Development stage to deployment.

**During Development:** developers usually perform unit testing to validate each individual component behaves as expected, these tests ensure that the individual API functions as expected.

**Continuous Integration:** As the different developers merge the code to central repo to the main branch automated API tests should be executed as a part of CI pipeline, this API tests ensures the new changes merged to main branch does not break the existing/working functionality which ensures the APIs remain constant.

**Before Deployment:** before deploying a new version of API from lower environment to the production or higher environment, this automated API tests should run and all the tests should pass to validate that the existing functionality, this covers all the existing functionality and integrity of API’s, which helps in catching the issues at earlier stages of development.

**Regression Testing:** Regression Tests include a bunch of test cases for the existing functionality from previous releases which should not affect existing functionality/behavior of the application because of new changes.

**Monitoring and production:** once all the Automated tests are completed these tests will be running continuously in production environment as a part of monitoring, which helps to identify issues if any have been introduced which helps to quickly fix the defects with a quick turnaround time and resolution.

By adding automated API tests at each stage of SDLC , teams can ensure the reliability, consistency, and the quality of Automated API’s throughout the its life cycle.

# Executing automated tests

executing automated test through JMeter.

Setting up Apache JMeter

The first step is to go to JMeter folder 🡪 go inside bin folder🡪select JMeter .bat file🡪 double click on this (.bat) file.

A computer screen shot of a computer

Description automatically generated

This might take some time to open and once it opens JMeter window will open.

Click on file 🡪 New to add a new test plan.

Here Test Plan is renamed as Performance Testing

A screenshot of a computer

Description automatically generated

Adding A Thread Group

Right click on the Performance Testing

Go to add 🡪 Thread (Users) and select the Thread Group.

Below snapshot is renamed as setup Thread Group

A screenshot of a computer

Description automatically generated

Here, the thread group can be understood as users, i.e., the number of users.

It has several options, like

Action to be taken after there is an error: you can choose it the way you want.

Then, there is a “number of threads” where you can choose the number of users you want.

Then, there is a “ramp-up period in seconds” that states the time gap between the users' hits.

There is a “loop count” option where you can choose how many times the test will run for the number of users. You can also choose the test to run infinite times.

There is also a scheduler here to help you schedule the start time and the test’s end time.

For now, make the number of threads to be 5. And loop count to be 1.

# Adding HTTP Request

Right-click on the Thread group, go to add option and go to the sampler option.

In the menu that appears, select the HTTP request option.

You can see a Server Name or IP box; it will give the server’s name or the IP in that.

This demo will be used on Metadata Semaphore API, here.

Get all the necessary details mentioned in the below snapshot.

Come back to the JMeter window.

Paste the URL in the Server Name box.

Don't give http or HTTPS since these are protocols that will come in the other box and that will be automatically taken in the http request case.

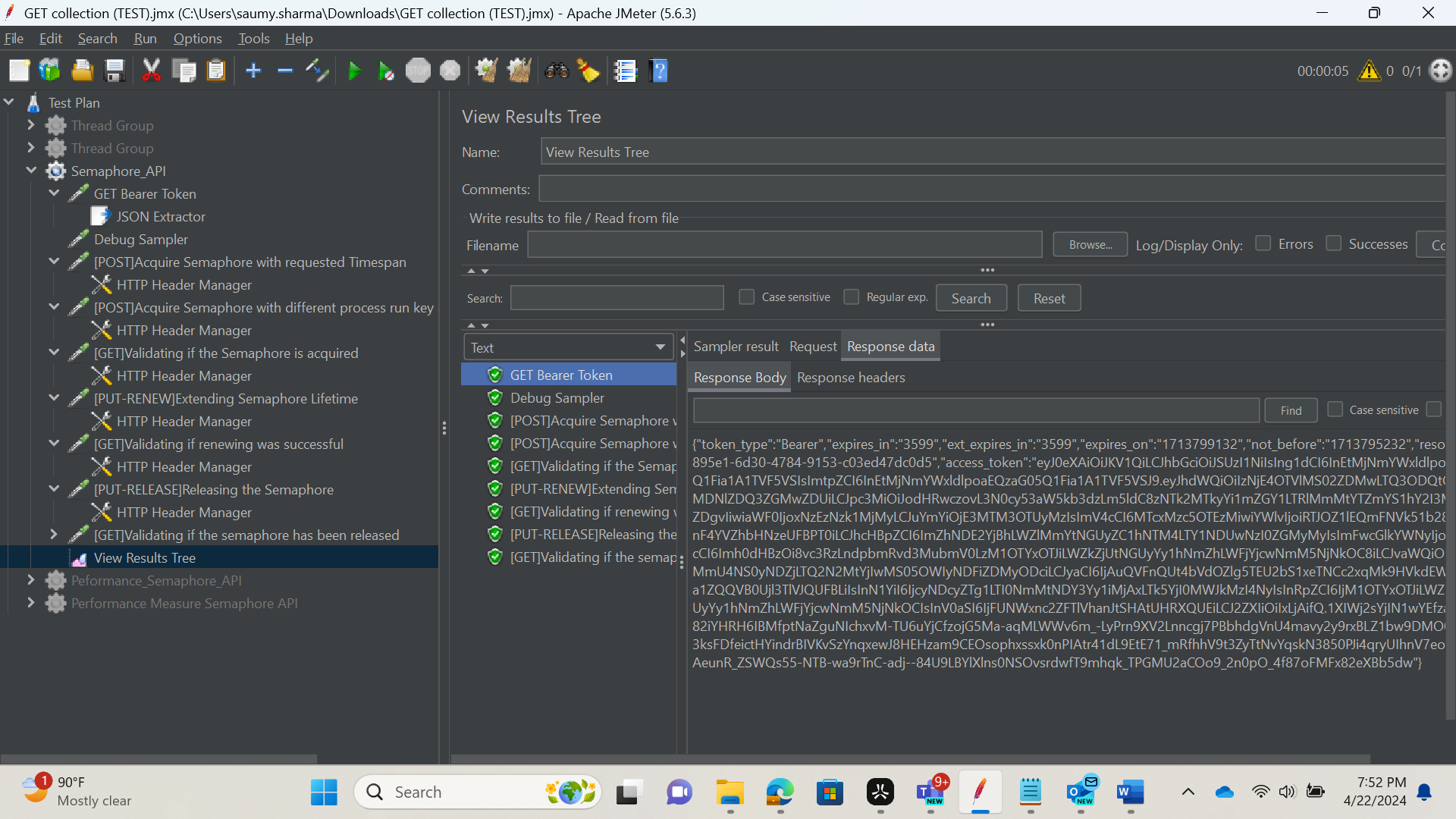
**Dynamically Obtaining the Bearer Token**

To obtain the Bearer token dynamically, create a new sampler with a HTTP request under your thread. Use the details as given below. The client\_id and client\_secret are sensitive information, hence hidden.

A screenshot of a computer

Description automatically generated

The Bearer token request gives the output in the form of a JSON body which contains the Bearer Token, which now cannot be used for the other requests under your thread.



Now to extract out the token from this JSON response and make it usable, we need to create a JSON extractor under the request used earlier. Enter the details as shown in the screenshot below.

A screenshot of a computer

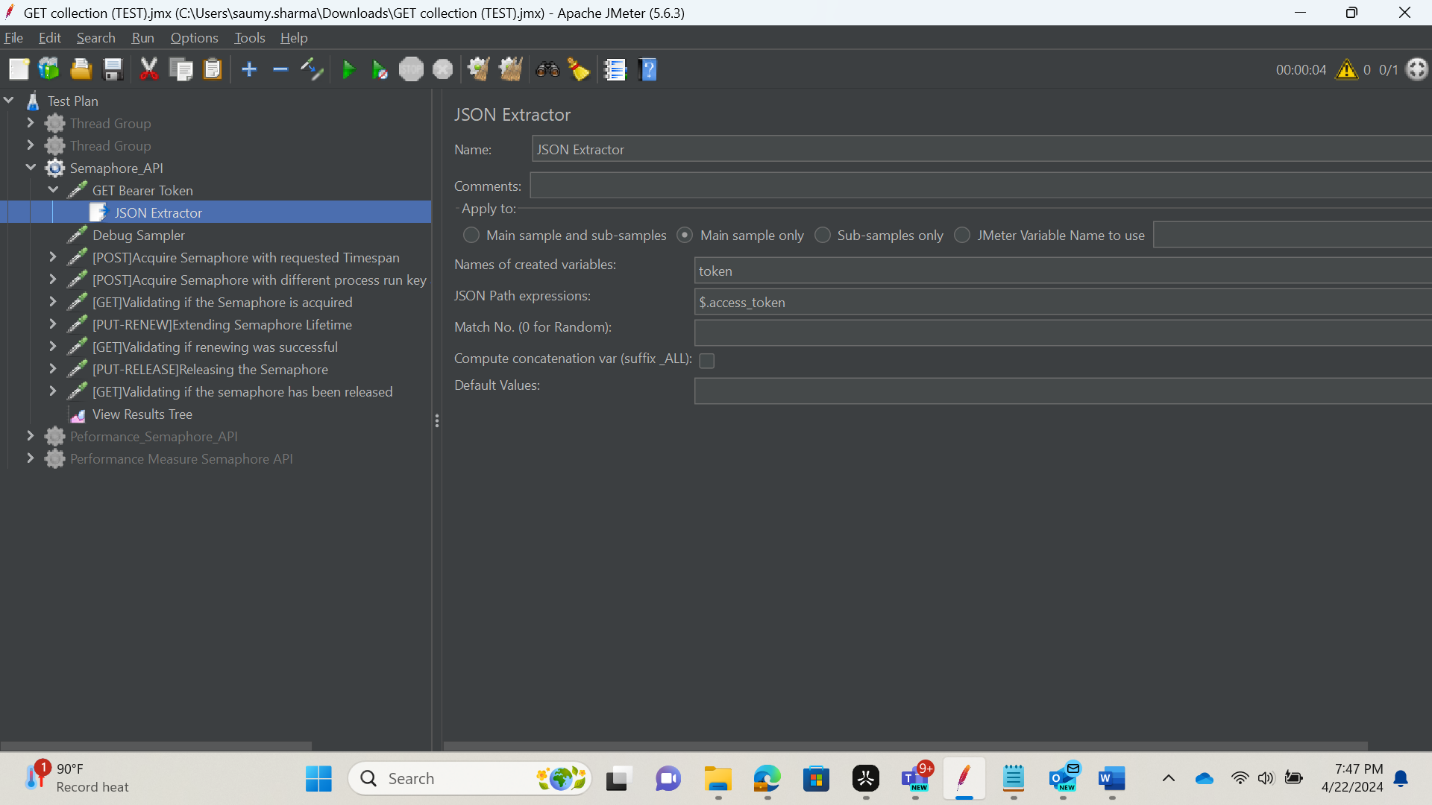
Description automatically generated

This allows for the Bearer token to be extracted out of the JSON and be stored in a variable name entered.

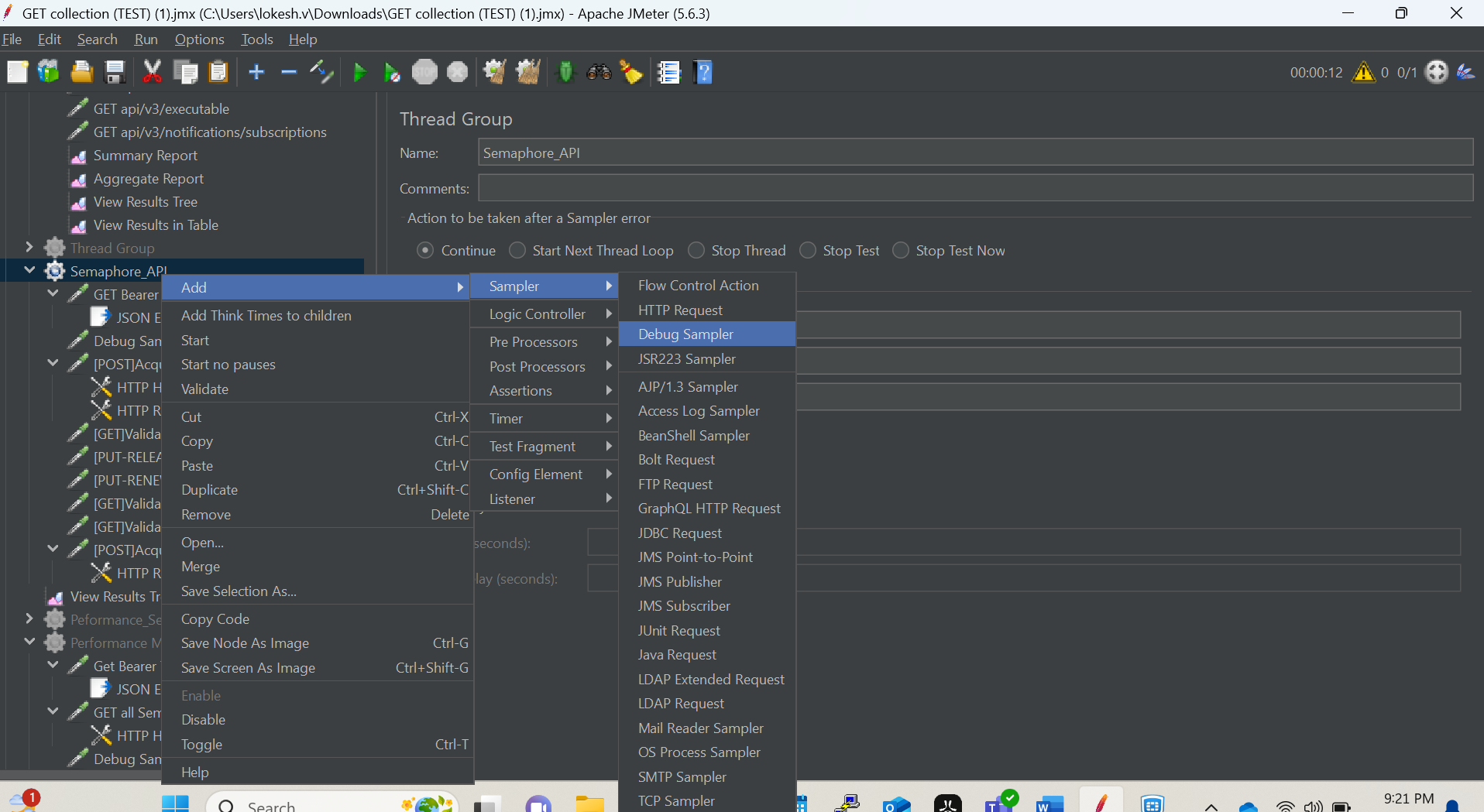
In this example : name of the created variable is **token**

JSON Path expression is **$.{access\_token}**

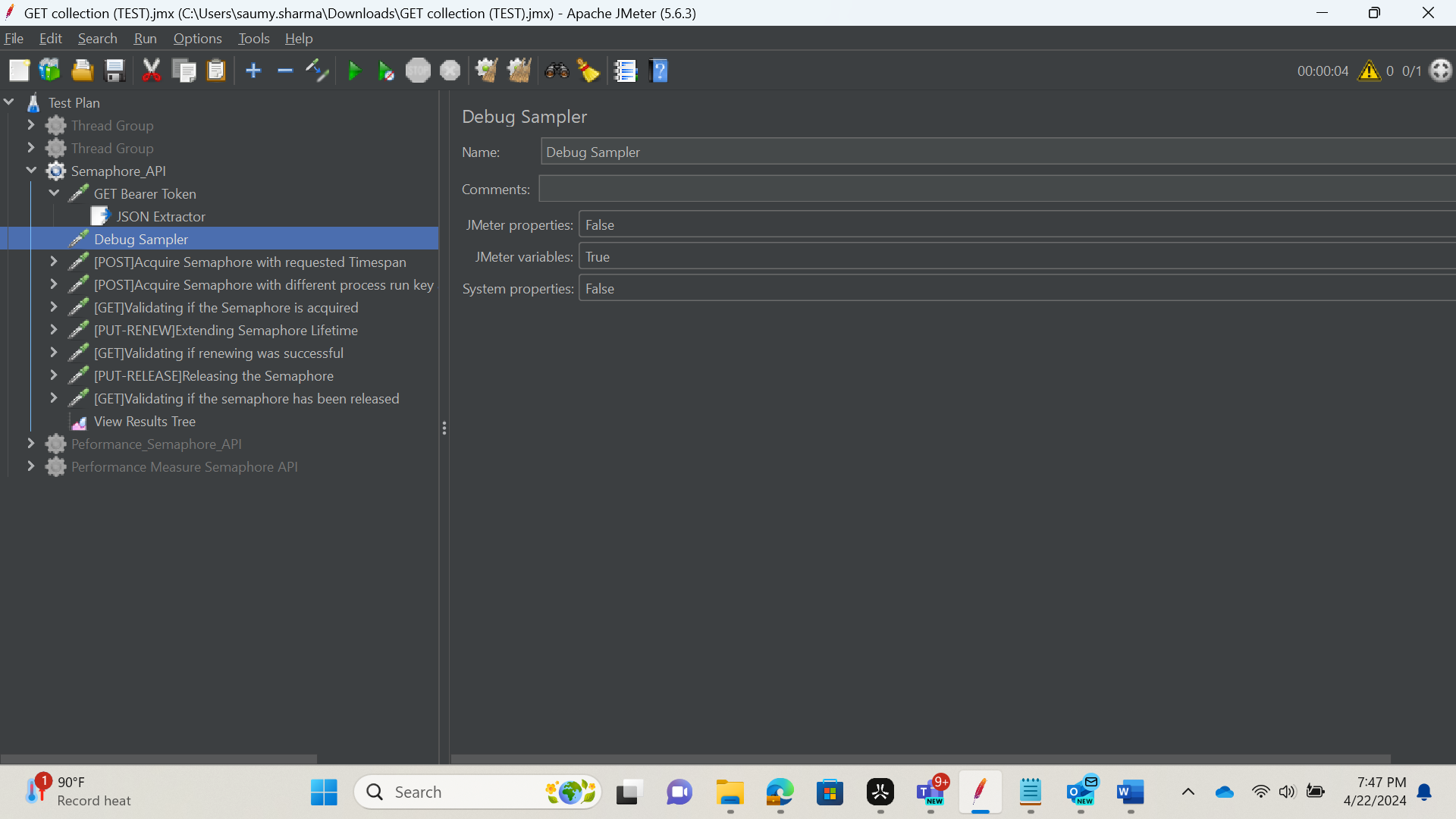
$.{access\_token} 🡪 $ with one dot is the first level response body and access\_token is the name of the key for which we need the value**.**



How to add a debug sampler please see snapshot below.



Debug sample is used to debug the variables used in a particular request, for example here we are using the token as a variable to get the access token from the GET Bearer Token POST Request.



The above snapshot shows how to setup the debug sampler with default values.

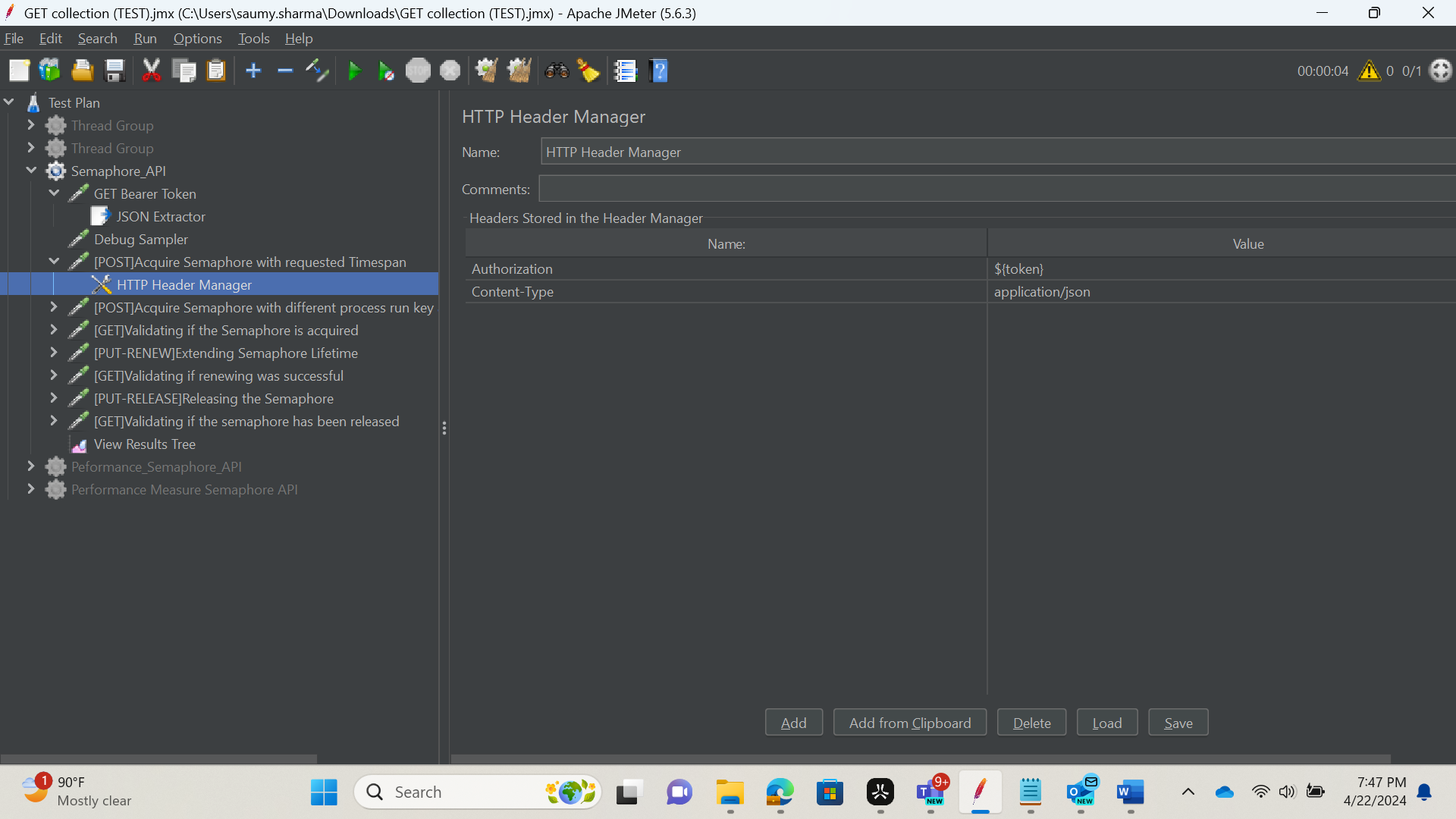
Below is an example for debug sampler, after we run a thread or test case, in the view result tree we see the Debug sample and variable used as token and its value.

A screenshot of a computer program

Description automatically generated

Under the existing Header Manager, change the authorization field to the value as shown in the screenshot. This ensures that the access token which is stored in a variable is used wherever “Authorization” is needed.

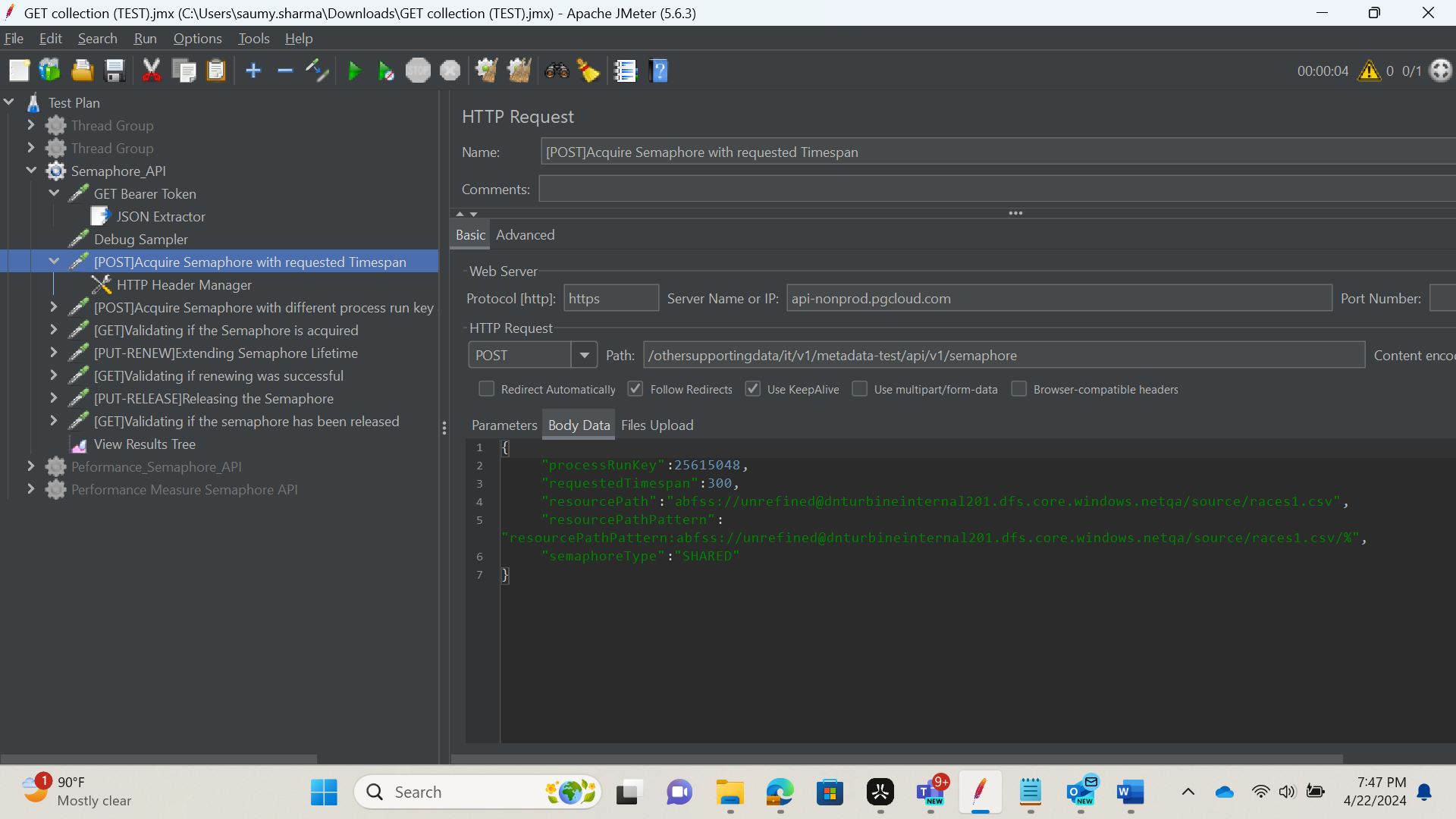
Here we are using $.{token} $. 🡪 represents the first level key, $.. 🡪 represents the second level key.



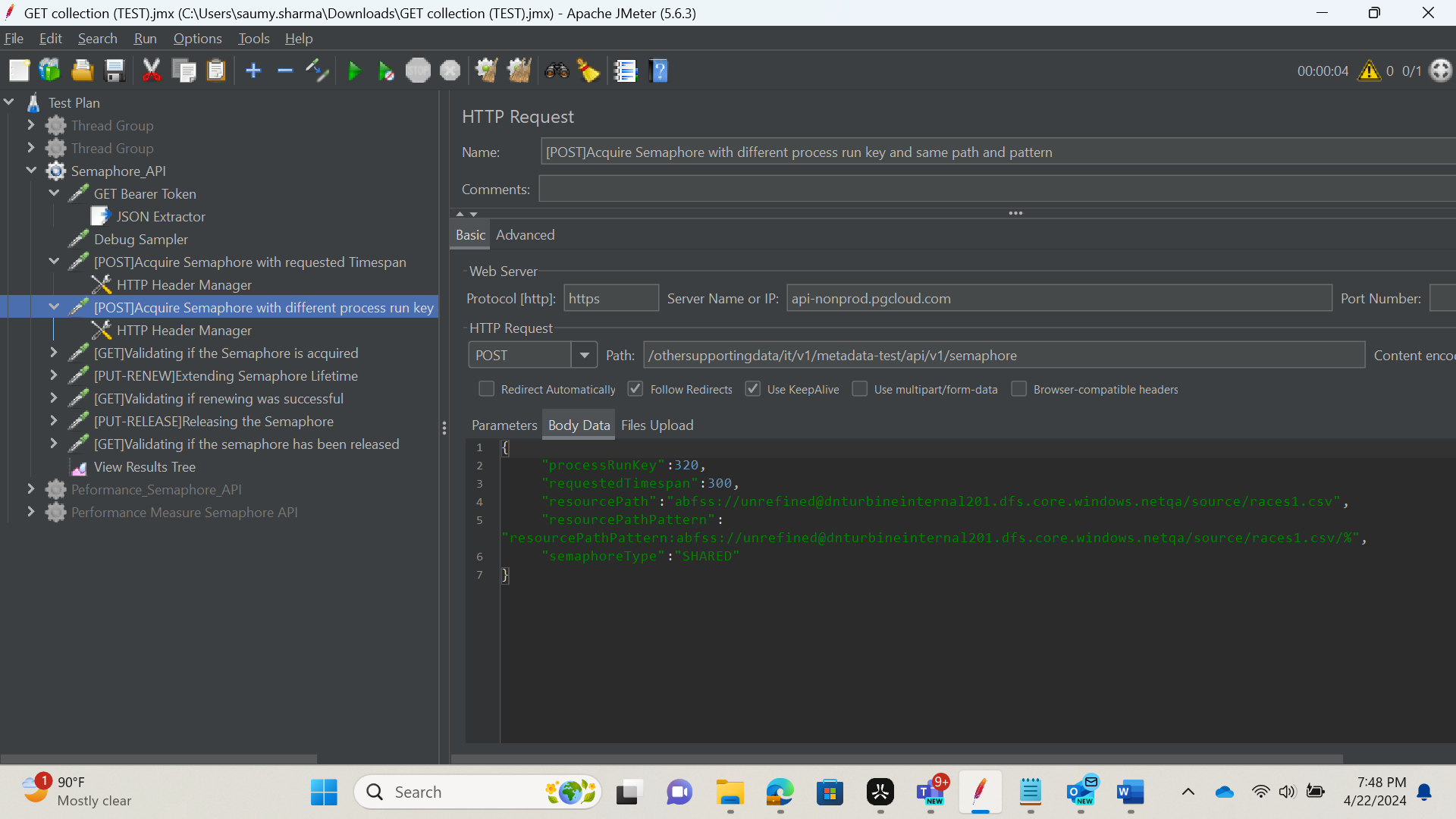
**Semaphore Flow**

Create respective sampler HTTP requests for each step of a semaphore flow under your thread. Setup each carefully by entering relevant request type, server name and path.

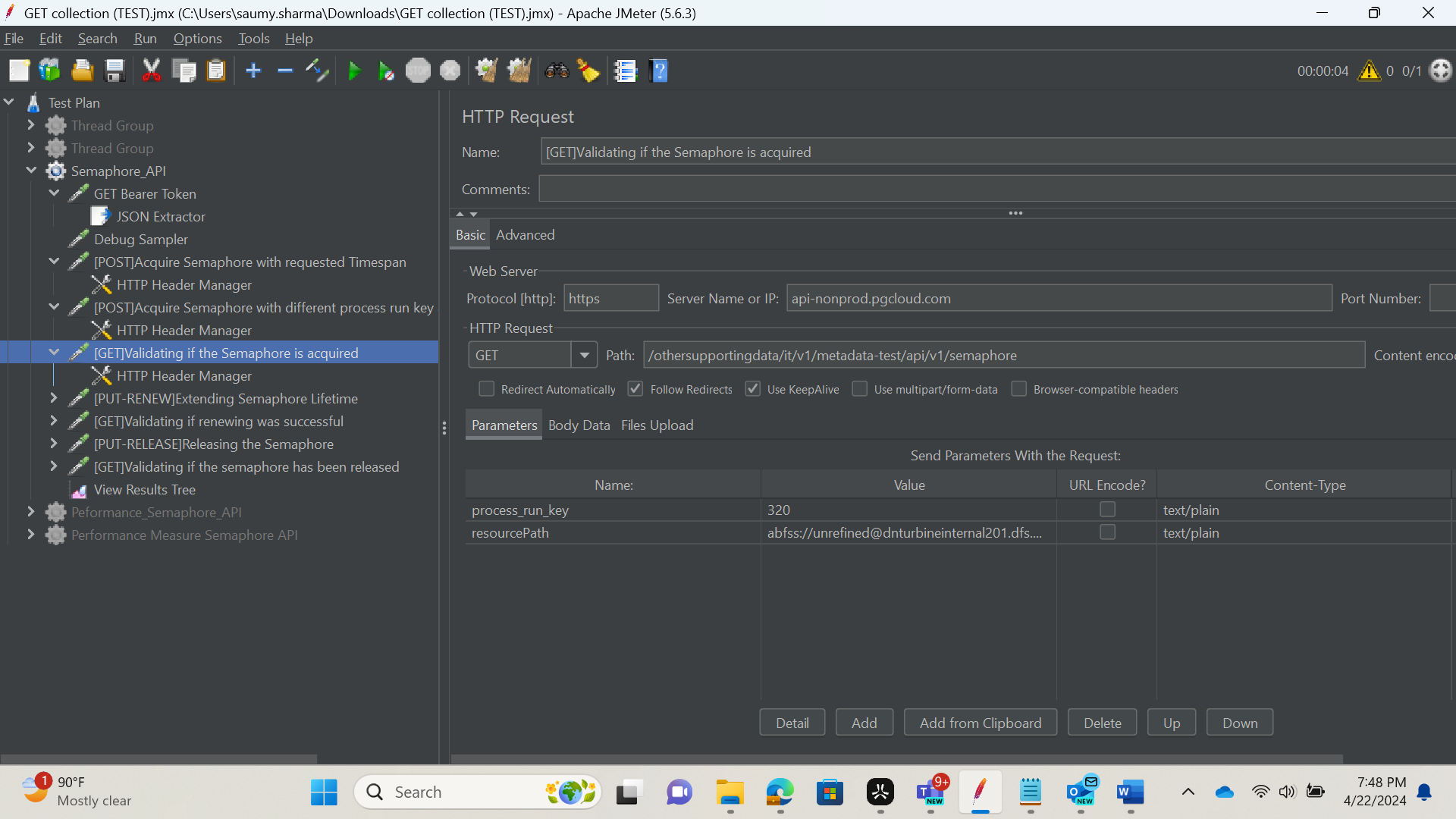
1. Creating/acquiring Semaphore using POST with *requestedTimespan* passed



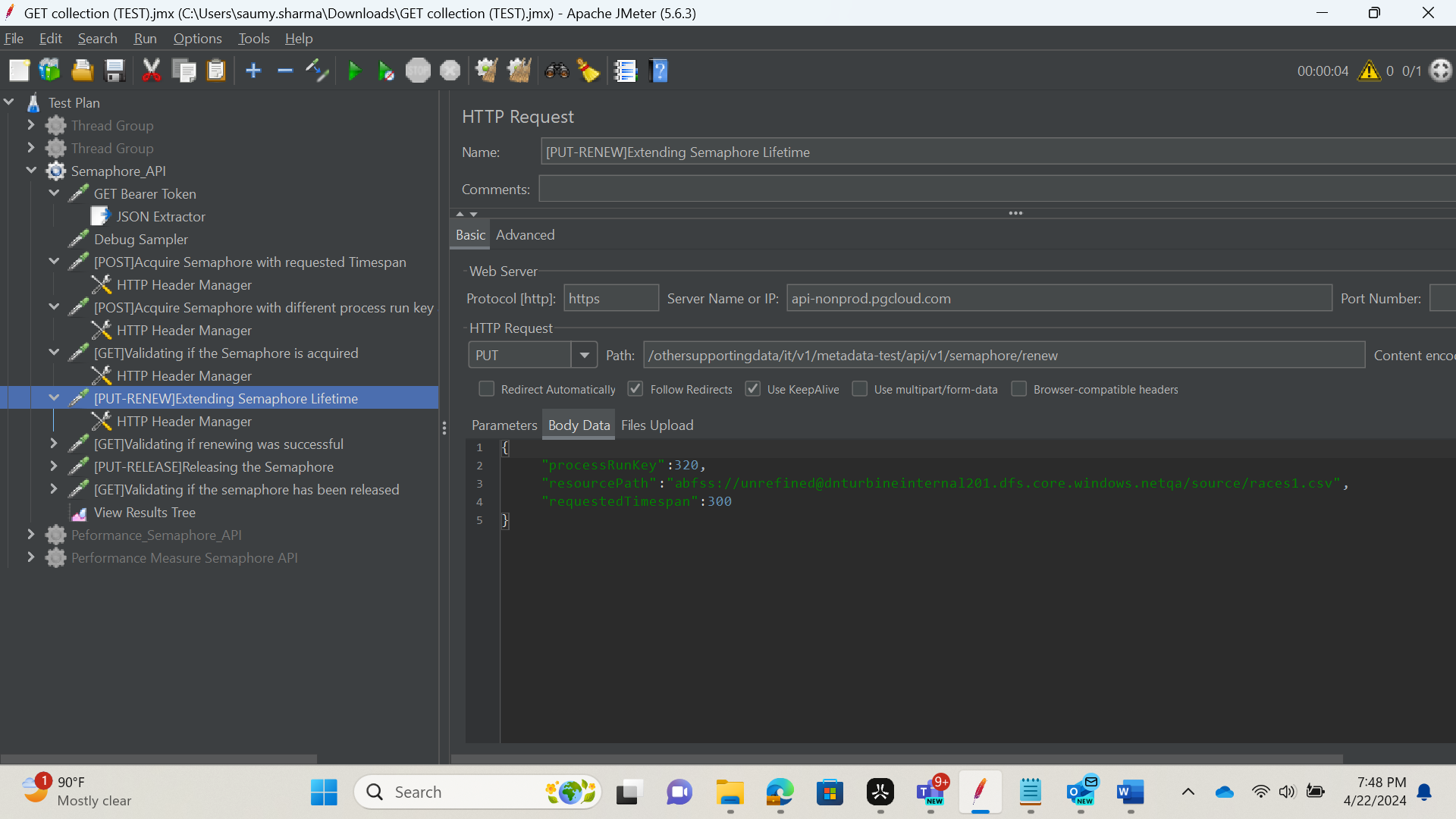
1. Attempting to create/acquire Semaphore again using another POST request and different *processRunKey (resource path and pattern are the same as the previous request)*



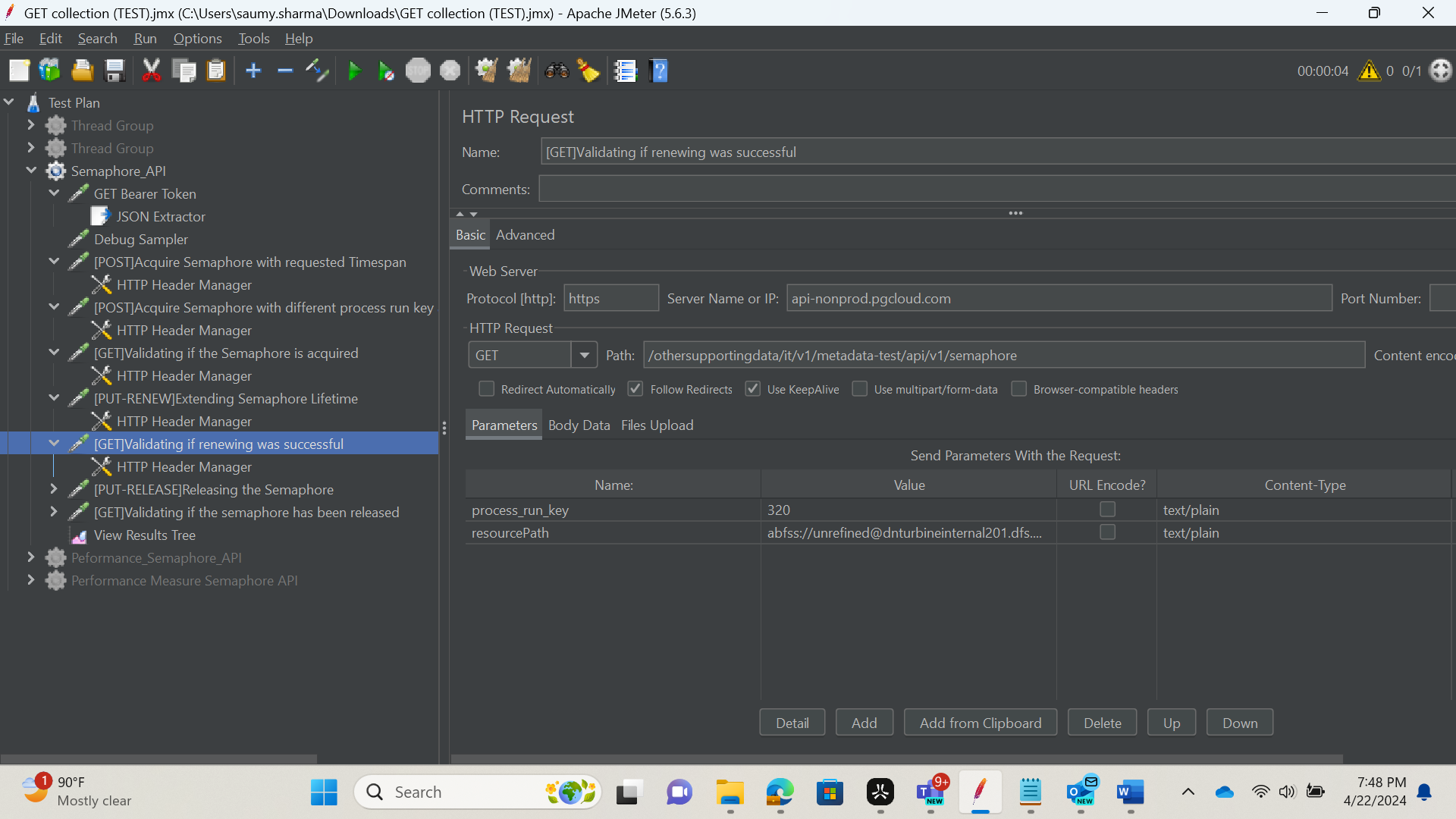
1. Validating if Semaphore was in fact created using GET request



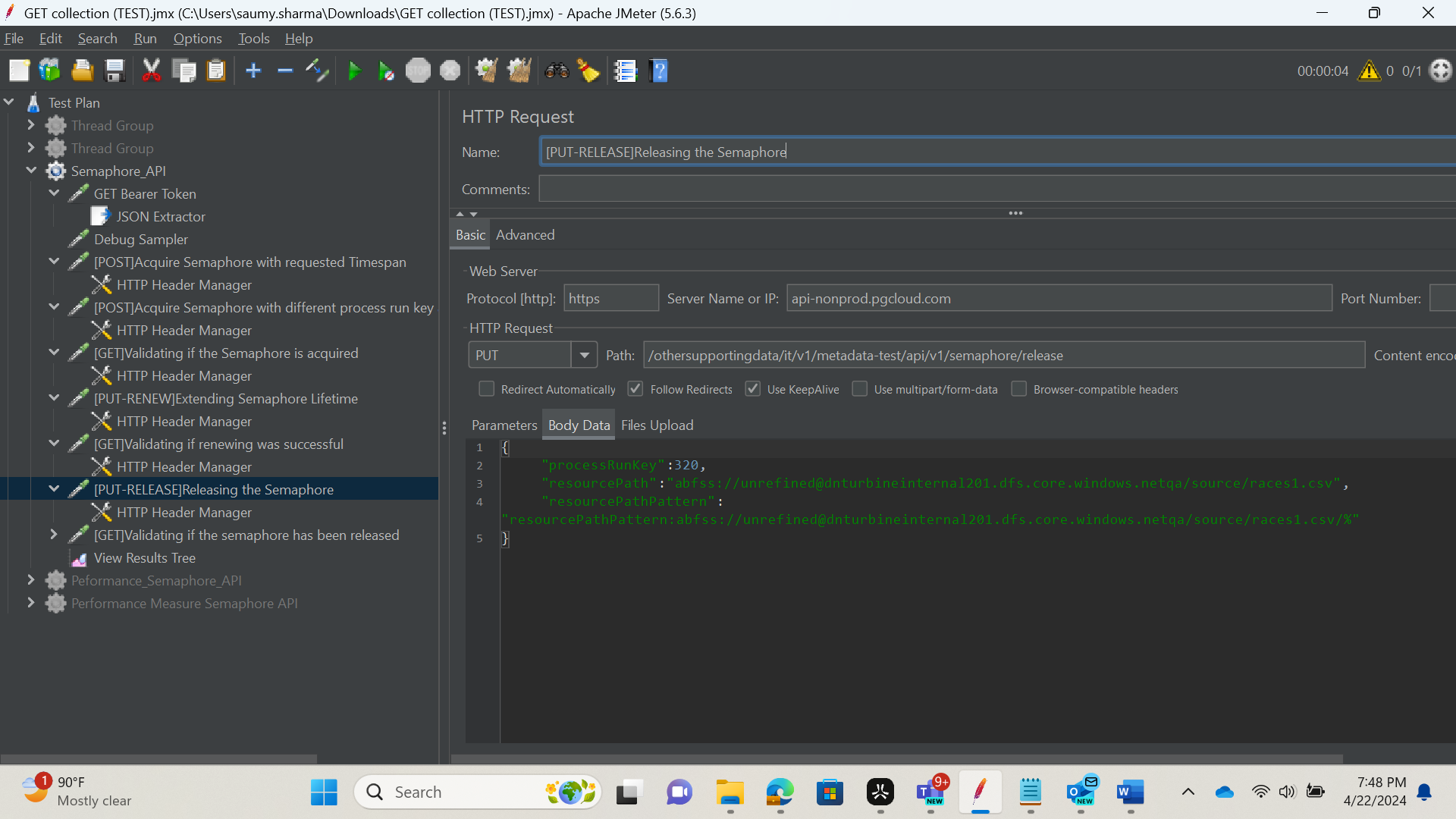
1. Extending Semaphore’s lifetime by sending PUT /renew request,



1. Validating if renewing was completed by sending another GET request and comparing *releaseTime*.



1. Releasing/destroying the Semaphore by sending PUT /release request.



1. Validating if the Semaphore has been released via sending final GET request with resourcePath and processRunKey to check if no semaphores was found.



**Results**

To view the results of the requests run, create a Results Tree listener under your thread. After executing, each test gets displayed in green or red as shown below, denoting success or failure respectively. The response body for each request can also be checked here.

