

MicroServer Python API documentation

1. Description

MicroServerAPI.py is the Microservices.MicroServer client module for Python. This client unleashes the potential of the main edition of the server almost completely (with the exception of supporting functions for working with external storage).

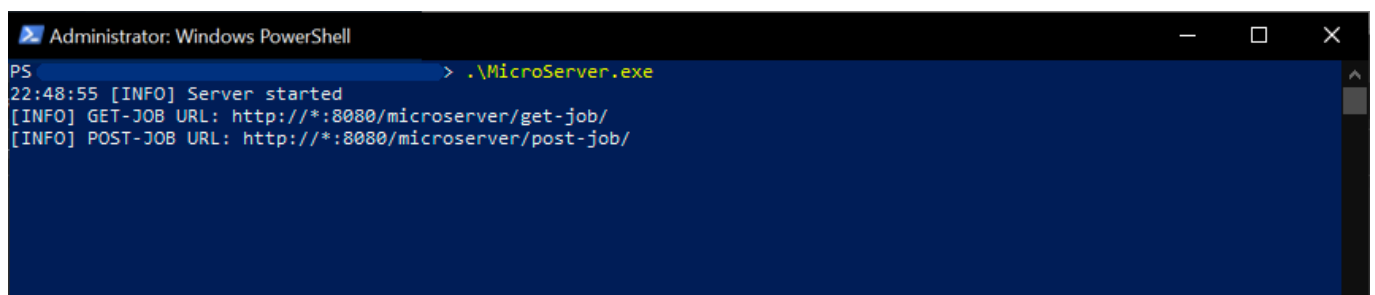
2. Module composition

The "MicroServerAPI" module consists of the following classes:

- MicroService
- ResponseAddress

The "MicroService" class is the main one and is intended, in fact, for turning an ordinary script into a service for MicroServer.

To create an object of this class, you must pass at least two arguments to the constructor: the GET-JOB address and the POST-JOB address that the client class will use. Both addresses can be found from the output of the debug edition of the server immediately after the start:



```
Administrator: Windows PowerShell
PS > .\MicroServer.exe
22:48:55 [INFO] Server started
[INFO] GET-JOB URL: http://*:8080/microserver/get-job/
[INFO] POST-JOB URL: http://*:8080/microserver/post-job/
```

Here the strings "http:// *:8080/microserver/get-job/" and "http://*:8080/microserver/post-job/" mean GET-JOB and POST-JOB addresses, respectively. However, these are not complete addresses, but address templates. To get the final address:

instead of an asterisk, you need to substitute the IP address or domain name; if the server is running on the same computer as this client, the asterisk must be replaced with "localhost".

The result will be the following: «http://localhost:8080/microserver/get-job/» and «http://localhost:8080/microserver/post-job/».

The final touch is to remove the slash from the end of the line.

The final result looks like this: «http://localhost:8080/microserver/get-job» and «http://localhost:8080/microserver/post-job».

It may happen that there will be no asterisk in the address, then you should skip step 1. There may be situations when the asterisk is in different places, then you should create a request that matches the pattern (step 1 gets more complicated).

From build to build, the server can expect a connection on different ports and with different query strings, so in order to enter the correct addresses, it is recommended to first run the debug version of the server, which will show the ports and query strings (as in the screenshot above).

In addition to addresses, the constructor also accepts an optional argument "persistentMode", which is responsible for enabling "persistent mode". In this mode, the client does not issue connection errors with the server, but persistently tries to create a connection. By default, this mode is disabled so that the client will generate an error when specifying incorrect addresses. It is **recommended to enable** this mode in ready-made services, since it will allow you not to worry about the order of starting this service and the server (i.e., that the service should be started only after the server).

An example of creating an object specifying constructor arguments (arguments are specified by name):

```
serv = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',  
    url_post='http://localhost:8080/microserver/post-job')  
  
serv2 = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',  
    url_post='http://localhost:8080/microserver/post-job', persistentMode=True)
```

In the case of an object retrieved in the "serv" variable, the "persistentMode" argument is set to "False" by default.

3. MicroService class methods

The following methods are implemented in the MicroService class:

- GetJob
- GetNextJob
- PostJob
- PostIntermediateResult
- PostFinalResult
- ProcessAsFunction

The "GetJob" and "PostJob" methods are fundamental, since the functionality of the other methods is implemented on their basis. These methods will be considered at the end, since most of the problems are more convenient to solve using the remaining methods.

3.1. GetNextJob

This method should be used for the service to receive the task from the server. The call to this method looks like this:

```
data, addr = serv.GetNextJob(job_type='PyTest.01')
```

The method returns 2 values, one of which is data (placed in the "data" variable), the second is the return address of the data packet (placed in the "addr" variable). It is important that the order of specifying the variables "data" and "addr" is exactly the same. The variable "addr" stores an object of the "ResponseAddress" type (the second of the existing classes in the module) and will later be needed when sending the processing result, so **it is important to save it and not change it**.

The method takes only one argument - "job_type" - the name of the task (name of the data type) that the service accepts for processing. In this case, the datatype name is "PyTest.01". In fact, the **type name is just a string that was invented** by the creator of another service.

This method is blocking, so the thread that called this method will wait until a data packet of the required type arrives.

What exactly is in the "data" variable? In theory, you should know this even before calling "GetNextJob", because you yourself specify the argument "job_type" (give a name to the type). However, generally speaking, the data variable will contain a combination of dictionary, list,

or string, or boolean, or number, or None types. If the data will contain a dictionary or a list, then it can consist of everything that was listed earlier.

The structure of the received data will largely resemble JSON, since the returned object is nothing more than JSON, converted into a set of nested Python objects.

3.2. PostFinalResult

This method takes the following arguments:

```
serv.PostFinalResult(content='some string content', responseAddress=addr,  
    result_type='PyTest.01.Result')
```

This method returns nothing, but accepts "content" - the result of processing the received data and "responseAddress" - the return address of the packet, which in the case of this example is stored in the variable "addr". Both of the above arguments are required.

You may notice that in the case of the example, the result of data processing does not depend on what entered the service (via "GetNextJob"). This is fundamentally wrong in most cases.

The last argument is "result_type" - the type of the result, which is "null" by default. In this argument, you can give a type name to the content "content". It is **highly recommended that you create type names that include a unique service name**. This is necessary in order to avoid unnecessary coincidences of type names for packages, the data in which have a different structure (for example, one stores a string, and the other stores an array of numbers). Such coincidences are highly likely to cause the final micro-service system to malfunction.

It is not necessary to give a name to the type in this method, but it is highly desirable, as this will speed up the processing of packets on the server side.

3.3. PostIntermediateResult

This method has a set of arguments similar to PostFinalResult:

```
serv.PostIntermediateResult(content='some string content', responseAddress=addr,  
    result_type='PyTest.01.Result')
```

The functionality of this method is also very similar, since this method is also intended, like the previous one, to send the results of processing the received data, upon completion of processing (not during processing at all), but there is an important difference.

The PostFinalResult method sends a packet with the result of data processing directly to the service that requested this processing. We can say that this method returns a packet at the previously received address.

In contrast, the PostIntermediateResult method does send the packet not to the previously received address, but simply places it on the server as another task for another service. Thus, the current service can "delegate" the task of processing **this** package to another service and no longer return to **this** package. Now this service can deal with another package, "shifting responsibility" for the old package to another service, which can also delegate part of the processing task to another service, and so on, but, as a result, the last service in the processing chain must call the PostFinalResult method, returning that the result is the service that initiated this entire "pipeline" for processing the packet (that is, the service that sent the completely unprocessed packet to the first service in the chain).

Precisely because PostIntermediateResult delegates the task to another service, the "result_type" argument is required in it, and this argument **must not (!)** coincide with the "job_type" argument of the GetNextJob method from the same service, otherwise this service

delegates processing to itself by entering thus, into an endless processing cycle of the same package (*from the second iteration, starting to damage the package data*).

4. (3.3+½) Typical data processing service template

```
import MicroServerAPI

serv = MicroServerAPI.MicroService('http://localhost:8080/microserver/get-job',
                                   'http://localhost:8080/microserver/post-job', persistentMode=True)

while True:
    data, addr = serv.GetNextJob(job_type='ArrayConcatenator.A42.ConcatenateStringArrayToString')
    #Begin of data processing area

    result_string = ""

    for item in data:
        result_string += item

    #End of data processing area
    serv.PostFinalResult(content=result_string,
                        result_type='ArrayConcatenator.A42.ConcatenateStringArrayToString.Result',
                        responseAddress=addr)
```

Above, you can see an example of a typical micro-service. At the beginning, an object is created to interact with the server ("serv"), then an endless loop follows, in which packets with the name of the "ArrayConcatenator.A42.ConcatenateStringArrayToString" type are processed.

Please note that "A42" is a conditionally random combination of characters (invented by the developer of the service), which is needed to avoid unnecessary coincidences of type names (this was discussed earlier). "ArrayConcatenator" is the name of the micro-service itself, "ConcatenateStringArrayToString" is the name of the task to be processed. Please note that the name for this task could be, for example, and this: "62f54e2bd30ad44a6be7d22a7238003892357ed8", which would not change anything functionally, but the first option is much more preferable, since it is much more informative for a person.

After receiving the data and the return address, the script execution enters the data processing area, where the variable "result_string" (which has the "string" type) is declared. Further, it is assumed that "data" is an array of strings (the "assumption" is made based on the fact that the package for processing was selected by the specified type name, and not random). Then this array is looped into one line, after which the script execution goes beyond the scope of data processing.

Here the variable with the result ("result_string") is used, as well as the previously received address ("addr"). The type name is optional here, but it is specified (and it is desirable to specify it). Note that the type name in the PostFinalResult method is **different** from the type name in GetNextJob.

Please note that **after processing the data and sending the result, the service should forget all the data with which it worked**, and then repeat the cycle of receiving and sending again.

Among other things, note that the order of the arguments to the PostFinalResult method in the example is different from what was shown earlier. This is due to the fact that arguments are accessed here by name, without regard to position (see named arguments in python).

3.4. ProcessAsFunction

Previously, the methods used to organize the work of a data-processing micro-service were considered. However, it is currently unclear how the data is “produced” for processing. How will data from somewhere outside get into the micro-service system?

Answer: for example, using the ProcessAsFunction method.

The call to this method looks like this:

```
result = serv.ProcessAsFunction(content='14+24*(1/7)',  
    requested_function='A424.Calculator.CalcExpression',  
    target_content_type='A424.Calculator.CalcExpression.Result')
```

The “requested_function” argument is essentially the same “job_type”, but since we are talking about data processing by other services (the current service only produces data, it does not process any data itself), the arguments are named a little differently. The “requested_function” argument should be thought of as “the name of the function that will process my data.” Аргумент «content» содержит данные, которые нужно обработать.

The current service must know in what form the data should be presented (string, array, dictionary, etc.) so that the desired service can process it. He also needs to know what needs to be written in the requested_function argument in order for the data to get to the required service (and to get anywhere at all).

The last argument (“target_content_type”) is optional, but desirable for faster server side processing of the packet. It is important to note that **this is only provided that the data processing services indicate the type of this result when placing the result, otherwise specifying the “target_content_type” argument will not allow you to get the result!** As you can imagine, omitting the target_content_type argument is a universal (but often less effective) solution.

Is it possible to call the ProcessAsFunction method to process some data inside the code block between GetNextJob and PostFinalResult? Of course, **ProcessAsFunction can be called in the data processing block (and, moreover, an unlimited number of times).** It is only important that the “requested_function” of the ProcessAsFunction method does not match the “job_type” of the GetNextJob method from the same service.

5. Microservice system example

A small example consists of 3 files, the code of two of which is given below. The third file is the client module file (“MicroServerAPI.py”), which must be kept near the script code files (in the examples, this file is in the same folder as the scripts themselves).

Code of data processing service (ExampleService.ArrayConcatenator.py):

```

import MicroServerAPI

serv = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',
                                   url_post='http://localhost:8080/microserver/post-job',
                                   persistentMode=True)

while True:
    data,addr=serv.GetNextJob(job_type='ArrayConcatenator.A42.ConcatenateStringArrayToString')
    #Begin of data processing area

    result_string = ""

    for item in data:
        result_string += item

    #End of data processing area
    serv.PostFinalResult(content=result_string,
                        result_type='ArrayConcatenator.A42.ConcatenateStringArrayToString.Result',
                        responseAddress=addr)

```

Code for receiving data from outside service (ExampleService.TestProgram.py):

```

import MicroServerAPI

serv = MicroServerAPI.MicroService(url_get='http://localhost:8080/microserver/get-job',
                                   url_post='http://localhost:8080/microserver/post-job',
                                   persistentMode=True)

while True:
    st = []
    inp = input('Enter string: ')
    while inp != '':
        st.append(inp)
        inp = input('Enter next string (leave blank to exit): ')

    print("Input array:")
    print(st)
    print()
    print("Requesting concatenation...")

    result = serv.ProcessAsFunction(content=st,
                                   requested_function='ArrayConcatenator.A42.ConcatenateStringArrayToString',
                                   target_content_type='ArrayConcatenator.A42.ConcatenateStringArrayToString.Result')

    print("Result: ")
    print(result)
    print()

```

The first of the above services is engaged in data processing, so it immediately connects to the server. However, if the server is not running, "persistentMode" comes into play: the service is waiting for the server to start, so the order of starting this service is not important (obviously, the server is not important in principle).

The second is no longer involved in processing, but waits for user input. If you start this service while the server is not running, this service will still wait not for the server, but for user input. If, by the time the user has finished entering data, the server is still not running, the service will start waiting for the server (thanks to "persistentMode").

6. Поля модуля MicroService

The following properties exist in the MicroService class:

- PersistentMode

The value of the "PersistentMode" property can be changed after the creation of an object of the "MicroService" class. This allows you to create a class with the "persistentMode" constructor argument equal to "False", and after executing the necessary code, set the "PersistentMode" property to "True", thereby enabling persistent mode.

7. Advanced Techniques

The advanced methods of the client class include:

- GetJob
- PostJob

On the basis of the above methods, all the previously considered methods were implemented.

7.1. GetJob

The call to this method looks like this:

```
data,addr = serv.GetJob(job_type='PyTest.01',  
                        job_id='81fe8bfe87576c3ecb22426f8e57847382917acf')
```

As you can see, this method is similar to "GetNextJob": it returns the same number of variables, in the same order, the "job_type" argument is present. However, there is also a difference: one of the GetJob arguments is optional. You can specify both arguments, or you can specify only "job_type", or you can specify only "job_id". You cannot specify any arguments.

As you might guess, "GetNextJob" is nothing more than GetJob without specifying "job_id".

Why does a task need an identifier?

To begin with, you need to understand that the **unspecified argument becomes "null"**, that is, there is no difference between not specifying an argument, and in order to specify it and pass the value "null".

Then, you need to understand exactly what the meaning of "null" is: it is a special meaning that can simultaneously show both absence and indifference depending on the action. If the packet is sent to the server, then "null" in the packet means the absence of a value for one or another field. If the packet is requested from the server, then "null" in the request means indifference to the value of the corresponding field.

Now we can return to the question "Why does the task need an identifier?"

The identifier for the task is specified when the service needs to receive the result of processing its package. (Doesn't this remind you of the "ProcessAsFunction" method?) The identifier needs to be unique for the entire micro-service system, plus everything, unlike the type, it is one-time and is generated by the client immediately before sending the packet. For all the previously listed properties, it can be considered a full-fledged "name" of the package, but if we speak closer to the previously introduced designations, then the "address", or rather, the "return address". It seems like something was said about the return address earlier, right?

Previously, the return address was considered for granted: it is there, it needs to be saved and transferred. Here, the package attribute in the form of the desired identifier (return address) can be set.

Of course, the variable "addr" will store the same address that was specified during the request (if specified), and, of course, in order to set the address in GetJob, this address still needs to be learned from somewhere. It is impossible to find out the address from the outside, which means that all that remains is to create your own unique identifier (return address) and create a package with such an identifier.

7.2. PostJob

The call to this method looks like this:

```
serv.PostJob(content='data to process',
             job_type='PyTest.01',
             job_id='81fe8bfe87576c3ecb22426f8e57847382917acf',
             visibleId=False)
```

Of the already known arguments, here are "content" and "job_type" (in the PostFinalResult and PostIntermediateResult methods, this argument was called "result_type").

Of the required arguments, only "content" is here. "Job_type" and "job_id" can be specified both or separately, but, like GetJob, they cannot be missing together. *There is one more limitation, which will be discussed later.*

The "job_id" argument is needed to send a packet with the specified identifier for this argument. As mentioned earlier, this identifier will be the return address, so it will need to be specified in GetJob.

It seems that with this information, you can write your own "ProcessAsFunction":

```
def ProcessAsFunction(content, requested_function, target_content_type='null'):
    uid = 'MyModuleUniqueName.' + str(datetime.now())

    serv.PostJob(content=content,
                 job_type=requested_function,
                 job_id=uid,
                 visibleId=False)

    data, addr = serv.GetJob(job_type=target_content_type, job_id=uid)

    return data
```

As you can see in the example, the identifier is generated by the program at the beginning of the function from the module name ("MyModuleUniqueName.") And the current time (obtained as a string by the combination "str (datetime.now ())"). This allows you to create a fairly unique identifier that is stored in the "uid" variable.

Then the example calls the "PostJob" method, to which the created uid is also given. Upon completion of PostJob execution, the package is located on the server. Now it can be received by other services (of course, **one package can only go to one service**).

Now you can wait for the processing result. This is done in the GetJob method, which is passed the expected packet type, and, most importantly, the packet identifier (variable "uid"), by which it is possible to identify the very one among the potential dozens of the same type.

Please note that the created function only returns the "data" variable. The variable "addr" is actually not needed here, since it contains what is already known: "uid".

Haven't you guessed yet what the "visibleId" argument for the "PostJob" method is for? Well, I propose to think about it. If anything, the answer is below.

Well, let's pretend there is no such argument. Consider a situation where "target_content_type" is equal to "null". In this case, using PostJob, a package is created with the initial data, type and identifier, but without specifying the "visibleId" argument (which is equivalent to visibleId=True). Next, using GetJob, a request for a package is made, but only by its identifier (since "target_content_type" was not set). Is there a package with the uid identifier on the server at the time the GetJob command is executed? Of course, there is: this is the same packet that was sent to the server by the last command. As a result, it is this packet that will be received. It turns out that the same data will come as the one left.

The "visibleId" argument with the value "False" solves the problem simply and efficiently: by hiding the identifier in the sent packet. The server will not compare the requested ID with the package ID if the latter is hidden. Thus, a packet with a hidden id cannot be obtained by id, and the only way to get such a packet is to request it with the identifier "null" (that is, indifference to the identifier) and the same type name (this implies the restriction: **type name cannot be "null" when the identifier is hidden**).

After processing the data of a package with a hidden id, it must be returned to the service that created the task for processing. As mentioned earlier, this can be done using the PostFinalResult method (a special case of PostJob), which sends the processing result in a packet with the type name corresponding to the result, the old identifier ("uid" in our example and the "return address" for the processing service), which is denoted visible.

After sending PostFinalResult of a packet with a visible id, in the example above, the GetJob method will receive a packet with the result by id and return the result to the calling code.

8. Recommended reading

It is highly recommended that you familiarize yourself with the MicroServer MicroServer Communication Debug Program: MicroServer.Debugger Documentation. More in-depth literature into the MicroServer device is recommended: "Documentation Microservices.MicroServer". It may be helpful to become familiar with the MicroServer API clients for other programming languages.