MicroServer C# API documentation

1. Description

"MicroServerAPI.cs" is the file with the namespace of the "Microservices.MicroServer" client for the C # language. 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).

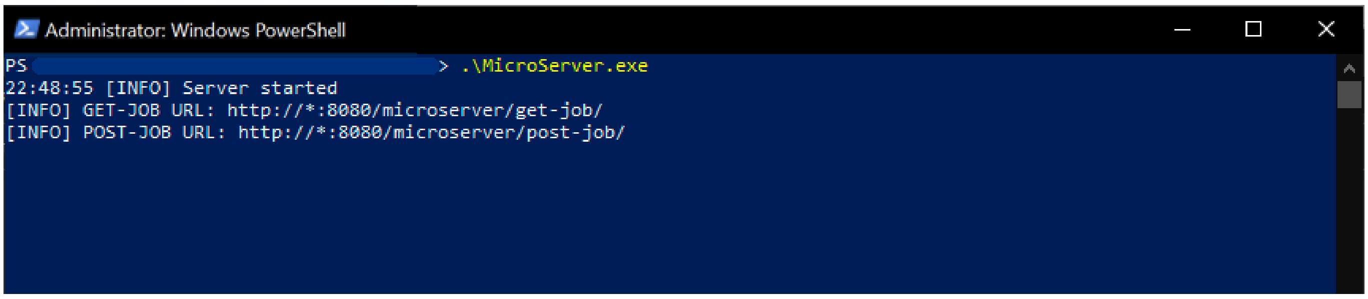
2. Module composition

The module file contains the namespace "MicroServerAPI", which consists of:

- class MicroService
- struct ResponseAddress

The "MicroService" class is the main one and is intended, in fact, for turning a program 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:



```
PS [redacted]> .\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:

1. 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/».
2. 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 the optional “persistentConnections” argument, which is responsible for enabling persistent connection 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 with specifying constructor arguments:

```
var serv = new MicroServerAPI.MicroService("http://localhost:8080/microserver/get-job",
                                            "http://localhost:8080/microserver/post-job",
                                             true);
```

In the case of an object obtained in the "serv" variable, the "persistentConnections" argument is passed the value "true". The default is "false" (this is an optional argument).

3. MicroService class methods

The following methods are implemented in the MicroService class:

- GetJob
- PostJob
- PostIntermediateResult
- PostFinalResult
- ProcessAsFunction

The "GetJob", "PostJob" and "ProcessAsFunction" methods have a number of overloads for most situations.

3.1. GetJob – getting a task

This method can be used to receive a task from the server by the service. The call might look something like this:

```
var  data = serv.GetJob<T>("ServiceTest.CSapi.01", out ResponseAddress addr);
```

The method returns 2 values, one of which is data (placed in the "data" variable), the second (via an argument) - the return address of the data packet (placed in the "addr" variable). The variable "addr" will be needed later when sending the processing result, so **it is important to save it and not change it**.

The template parameter (T) is the object type "data". The example says "T", but in reality it will be the name of a structure or class.

The method accepts only one argument as input - the first is the name of the task (the name of the data type), which the service accepts for processing. In this case, the data type name is "ServiceTest.CSapi.01". In fact, **the type name is just a string that was thought up** by the creator of the other service when sending the packet to the server.

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

It is important to note that the **MicroService class is thread-safe**.

As a result of the function execution, an object of the type specified in the template parameter must be placed in the "data" variable. If the data transferred from the server cannot be converted to the specified type, an exception will be thrown.

This method has several overloads, which will be viewed later.

3.2. PostFinalResult

This method can be called as follows:

```
serv.PostFinalResult<T>("ServiceTest.CSapi.01.Result", addr, data);
```

This method does not return anything, but it takes as the last arguments an object of type "T" - the result of processing the previously received data, and the second - the packet return address, which in the case of this example is stored in the "addr" variable.

The first argument is the type of the result. In this argument, you must give a type name to the content "data". 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 likely to cause the final micro-service system to malfunction.

The first argument **must not (!)** coincide with the first argument of the GetJob method from the same service, otherwise this service delegates processing to itself, thereby entering an endless processing cycle of the same package (*starting from the second iteration, it starts corrupting the package data*).

This method is implemented based on the "PostJob" method.

3.3. PostIntermediateResult

This method is similar to PostFinalResult, and its call looks like this:

```
serv.PostIntermediateResult<T>("ServiceTest.CSapi.01.Result", addr, data);
```

The functionality of this method is also very similar, so this method, like the previous one, is intended 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 PostFinalResult 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).

This method is implemented based on the "PostJob" method.

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

```
using MicroServerAPI;

namespace SomeMicroservice_1 {
    class Program {
        static void Main(string[] args) {
            MicroService serv = new("http://localhost:8080/5-semestr/compiler/get-job",
                                    "http://localhost:8080/5-semestr/compiler/post-job",
                                    true);

            while (true) {
                var data_input = serv.GetJob<string[]>(
                                              "ArrayConcatenator.A42.ConcatenateStringArrayToString",
                                              out ResponseAddress addr);
                //Begin of data processing area

                string result_string = "";

                foreach (var item in data_input) {
                    result_string += item;
                }

                //End of data processing area
                 serv.PostFinalResult<string>(
                                       "ArrayConcatenator.A42.ConcatenateStringArrayToString.Result",
                                       addr,
                                       result_string);
            }
        }
    }
}
```

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 packages with a type name are processed «ArrayConcatenator.A42.ConcatenateStringArrayToString».

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 human.

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"). Note that the type name in the PostFinalResult method is **different** from the type name in GetJob.

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.

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.

A call to this method might look like this:

```
string result = serv.ProcessAsFunction<string, string[]>(
                                    "ArrayConcatenator.A42.ConcatenateStringArrayToString",
                                    string_array);
```

The first argument should be thought of as "the name of the function that will process my data".

The second argument contains the data to be processed.

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 first argument in order for the data to reach the desired service (and to get anywhere at all).

ProcessAsFunction method also has the following overload (call example):

```
string result = serv.ProcessAsFunction<string, string[]>(
                                  "ArrayConcatenator.A42.ConcatenateStringArrayToString",
                                   string_array,
                                  "ArrayConcatenator.A42.ConcatenateStringArrayToString.Result");
```

The last argument in this overload is the type name for the object to receive. Using this overload is optional, but desirable for faster server-side packet processing. It is important to note that **this is only provided that the data processing services specify the type of this result when placing the result, otherwise using this overload will not allow you to get the result!** As you can imagine, using the first overload (presented at the beginning of this section) is a universal (but often less efficient) solution.

Is it possible to call the ProcessAsFunction method to process some data inside the block of code between GetJob 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 first argument of the ProcessAsFunction method does not match the first argument of the GetJob method from the same service.

5. Micro service system example

A small example consists of 2 projects. The code for two "Program.cs" of them is shown below. It is important to note that each of the projects includes one more file - "MicroServerAPI.cs", which are the same and are connected to the projects.

The first part of this example has already been presented earlier - in the section "Typical service template for data processing", where a complete example of a service for data processing is presented.

The second part of the example (the file of the data provider to the system) is shown below:

```
using System;
using System.Collections.Generic;
using MicroServerAPI;

namespace SomeMicroservice_2 {
    class Program {
        static void Main(string[] args) {
            MicroService serv = new("http://localhost:8080/5-semestr/compiler/get-job",
                                    "http://localhost:8080/5-semestr/compiler/post-job",
                                    true);

            while (true) {
                List<string> data_input = new();
                while (true) {
                    Console.Write("Enter string (leave empty to exit): ");
                    string single_string = Console.ReadLine();

                    if (single_string == "") {
                        break;
                    } else {
                        data_input.Add(single_string);
                    }
                }
                Console.Write("Entered strings:");
                foreach (var item in data_input) {
                    Console.Write(" " + item);
                }
                Console.WriteLine();

                var result = serv.ProcessAsFunction<string, string[]>(
                                    "ArrayConcatenator.A42.ConcatenateStringArrayToString",
                                    data_input.ToArray(),
                                    "ArrayConcatenator.A42.ConcatenateStringArrayToString.Result");

                Console.WriteLine("Result: " + result);
                Console.WriteLine();
            }
        }
    }
}
```

The first service from the example deals with data processing, so it immediately connects to the server. However, if the server is not running, "persistentConnections" comes into play: the service is waiting for the server to start, so the order of launching for this service is not important (obviously, the order of launching is not important to the server 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 "persistentConnections").

6. Fields of the MicroService class

The following properties exist in the MicroService class:

- PersistentConnections

The value of the "PersistentConnections" property can be changed after the object of the "MicroService" class has been created. This allows you to create a class with the "persistentConnections" constructor argument equal to "false", and after executing the necessary code, set the "PersistentConnections" property to "true", thereby enabling persistent mode.

7. Overloads of GetJob and PostJob

The basic (advanced) methods of the client class include:

- GetJob
- PostJob

All other methods are implemented on the basis of the above methods.

7.1. GetJob

In addition to the previously considered overload of this method, there are the following (call example):

```
var  data = serv.GetJob<T>("ServiceTest.CSapi.01",
                           "81fe8bfe87576c3ecb22426f8e57847382917acf");
```

And this one (call example):

```
string received_type = serv.GetJob<T>("81fe8bfe87576c3ecb22426f8e57847382917acf",
                                      out data);
```

As you can see, the first overload is similar to the previously given for GetJob: it returns an object, the first argument takes a type name. However, there is also a difference: there is a second argument, which contains the package identifier. The second overload also contains an identifier, but already as the first argument, and returns the result through the second argument.

Why does a task need an identifier?

First, you need to understand that the missing (not specified) argument **becomes null** (null is a keyword), that is, there is no difference between using the overload with the missing argument, and using the overload with the specified argument to pass the null value (the key word). However, it should be noted that most of the overloads are designed in such a way that null substitution occurs inside the method, so in most of the overloads passing null is prohibited (instead of passing null to this overload, it is recommended to use another overload).

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.

7.2. PostJob

This method has 2 overloads. Example of calling the first:

```
serv.PostJob<T>("ServiceTest.CSapi.01", data);
```

The first of the overloads is similar to PostFinalResult, except that this method does not accept a return address as one of the arguments: it only contains the type name and data.

On the next overload, you need to dwell in more detail:

```
serv.PostJob<T>("ServiceTest.CSapi.01",
                "81fe8bfe87576c3ecb22426f8e57847382917acf",
                 true,
                 data);
```

Из уже известных аргументов здесь имя типа, идентификатор, а также данные. Однако, здесь есть bool-аргумент (третий по счёту), который называется «visibleId».

This function has a nullable argument - the first one (type name). This means that "null" (keyword) can be passed to this argument.

There is one limitation for the combination of method arguments, which will be discussed later.

The second 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 is on the basis of this overload that the PostFinalResult and PostIntermediateResult methods are implemented.

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

```
public O ProcessAsFunction<O, I>(string requested_function, I content)
{
    string uid = "MyServiceUniqueName." + DateTime.Now.ToString();

    serv.PostJob<I>(requested_function, uid, false, content);
    serv.GetJob<O>(uid, out O result);
    return
}
```

As you can see in the example, the identifier is generated by the program at the beginning of the function from the service name ("MyServiceUniqueName.") And the current time (obtained as a string by the combination "DateTime.Now.ToString ()"). 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.

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. In this case, using PostJob, a package is created with the initial data, type and identifier, but without specifying the "visibleId" argument (which is basically equivalent to visibleId=true). Next, using GetJob, a batch request is made, but only by its identifier. 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.