

MicroServer.Debugger documentation

1. Description

MicroServer.Debugger is a utility for getting debug data from Microservices. MicroServer. This program implements all debugging methods of the server, but is not intended for full participation in the work of the micro-service system (the program allows you to monitor data on the server, but not modify them).

2. Beginning of work

The GET-JOB and POST-JOB addresses are required to start the debug program. After entering the addresses, the debug program tries to connect to the server to make sure the entered data is correct. If one of the addresses is entered incorrectly, the program will display an error and a description of this error, which should help in fixing the problem.

If you have already entered addresses before, the program saves them to files in its working directory. This saves you the trouble of entering addresses every time you start. However, if you enter the address incorrectly, or you need to enter a different address, you can delete files in the program folder named "GET" and "POST", or edit their contents.

After successful completion of address verification, a message will be displayed in the console «SERVER VALIDATED SUCCESSFULLY».

NOTICE: after restarting the server, there is no need to restart this program. The debugger does not check if the server is running or not after the initial check of addresses, so the server may not work while the debugger is running, however, if the server is not active while the debugger command is running, the debugger will generate an error.

Important! The use of this program makes sense only if the debug edition of the server is used. Otherwise, almost all of the commands provided by this program will be unavailable.

3. Data output

All successful command execution results, during which the debug program receives useful data from the server, are written to files with a name corresponding to the command name located in the program's working folder (usually next to the executable file).

The file saves "raw" data, that is, directly those that came from the server, so when creating a structure parser from output files, use the MicroServer documentation.

4. Debug commands

The debug program (read "monitoring") implements the following micro-service server requests:

1. MicroServer.25367be645.ExternalStatus
2. MicroServer.25367be645.DebugEdition.getPendings

3. `MicroServer.25367be645.DebugEdition.getLocallyAvailableTypes`
4. `MicroServer.25367be645.DebugEdition.getTypesStatistic`
5. `MicroServer.25367be645.DebugEdition.getInternalStorageSnapshot`
6. `MicroServer.25367be645.DebugEdition.retrieveGetHistory`
7. `MicroServer.25367be645.DebugEdition.retrievePostHistory`

As you can see, request number 1 is not exclusive to the debug edition of the server, which means it can be executed for the main edition of the server as well.

List of available commands for debugging the program (as you can see, this list is compared with the previous one by item numbers):

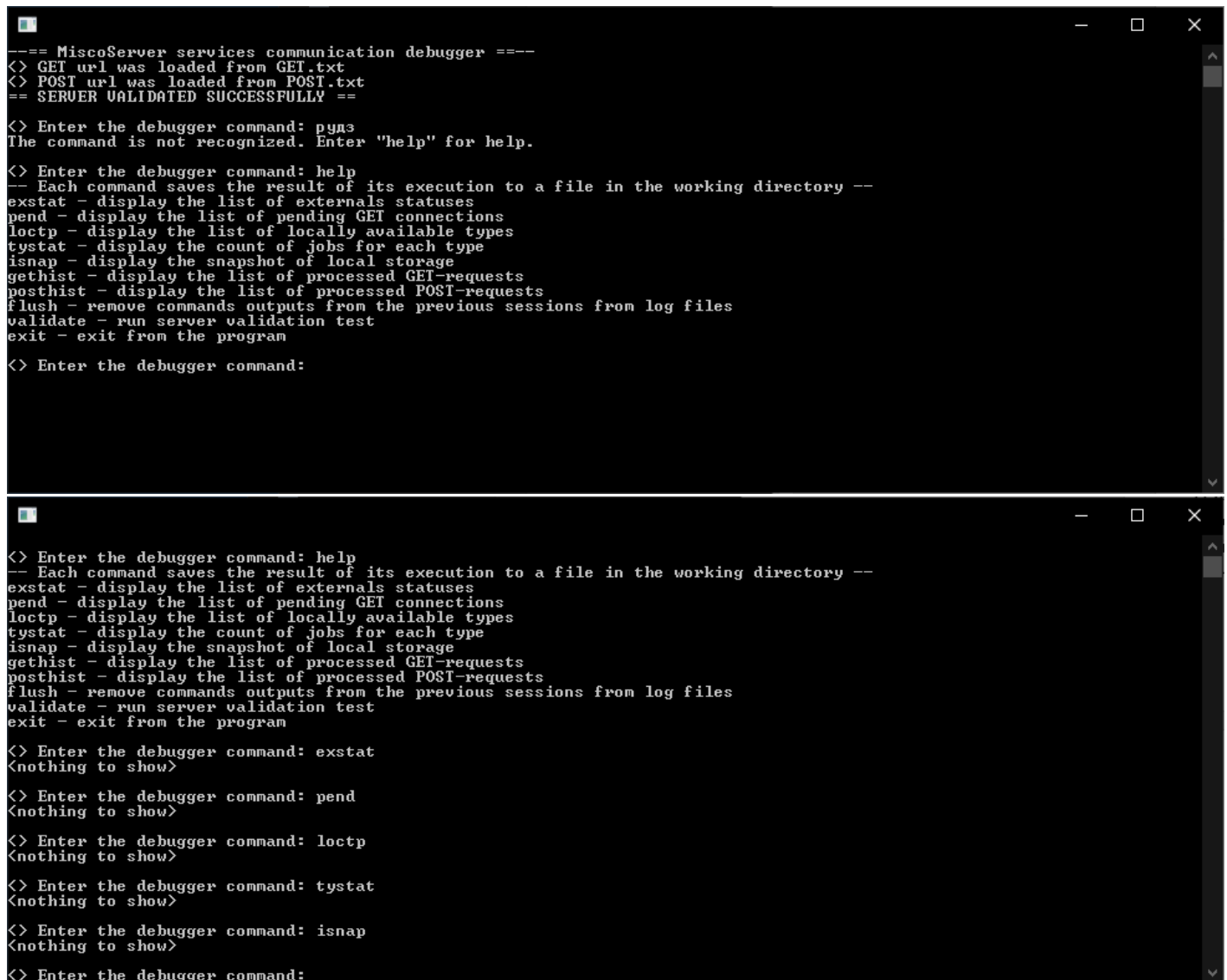
1. **exstat** – get overflow/underflow statuses of "external" types (that is, those stored in external storage as well).
2. **pend** – get a list of requirements, which packets fall under which half-awaited GET requests await at the time of command execution.
3. **loctp** – get a list of locally available types (prints just the type names).
4. **tystat** – get a list of type names with the number of packages of this type in both internal and external repositories.
5. **isnap** – get a **complete** snapshot of the internal storage (gets all packages in full, but *displays only the displayed data on the screen, and saves the full version to a file*). You should use this command only when the functionality of others is not enough, since it may take some time to transfer the data of the entire storage.
6. **gethist** – get a list of packets received by the server from the moment of the last execution of this command to the moment of the current execution of this command (receives all packets in full, as well as the requirements for which the packet was selected and the time of receipt, but *displays only the displayed data on the screen, and the full option saves to file*). If during the interval between executions of this command more than 512 packages for this debug list accumulate on the server, this list will be truncated by removing the first 128 packages.
7. **posthist** – get a list of packets received by services from the server during the period from the last execution of this command to the moment of the current execution of this command (receives all packets in full and the time of sending, but *displays only the displayed data on the screen, and saves the full version to a file*). If during the interval between executions of this command more than 512 packages for this debug list accumulate on the server, this list will be truncated by removing the first 128 packages.

5. Debugger Serving Commands

- **flush** – delete from files all data that was received **not during** the current session. This command can be called any time the debugger is running and will not affect the data received during the current session. Useful for reducing the size of the command output saving files (most often they are located in the program folder), since the size of these files grows over time.
- **validate** – run a server test (as when starting the debugger). This can be useful if, for example, you need to make sure that the server is still running.
- **exit** and **help** do not need additional description.

6. Debugger examples

This is what working with the debugger looks like in general:



```
--== MiscoServer services communication debugger ==--
<> GET url was loaded from GET.txt
<> POST url was loaded from POST.txt
== SERVER VALIDATED SUCCESSFULLY ==

<> Enter the debugger command: pyd3
The command is not recognized. Enter "help" for help.

<> Enter the debugger command: help
-- Each command saves the result of its execution to a file in the working directory --
exstat - display the list of externals statuses
pend - display the list of pending GET connections
loctp - display the list of locally available types
tystat - display the count of jobs for each type
isnap - display the snapshot of local storage
gethist - display the list of processed GET-requests
posthist - display the list of processed POST-requests
flush - remove commands outputs from the previous sessions from log files
validate - run server validation test
exit - exit from the program

<> Enter the debugger command:

<> Enter the debugger command: help
-- Each command saves the result of its execution to a file in the working directory --
exstat - display the list of externals statuses
pend - display the list of pending GET connections
loctp - display the list of locally available types
tystat - display the count of jobs for each type
isnap - display the snapshot of local storage
gethist - display the list of processed GET-requests
posthist - display the list of processed POST-requests
flush - remove commands outputs from the previous sessions from log files
validate - run server validation test
exit - exit from the program

<> Enter the debugger command: exstat
<nothing to show>

<> Enter the debugger command: pend
<nothing to show>

<> Enter the debugger command: loctp
<nothing to show>

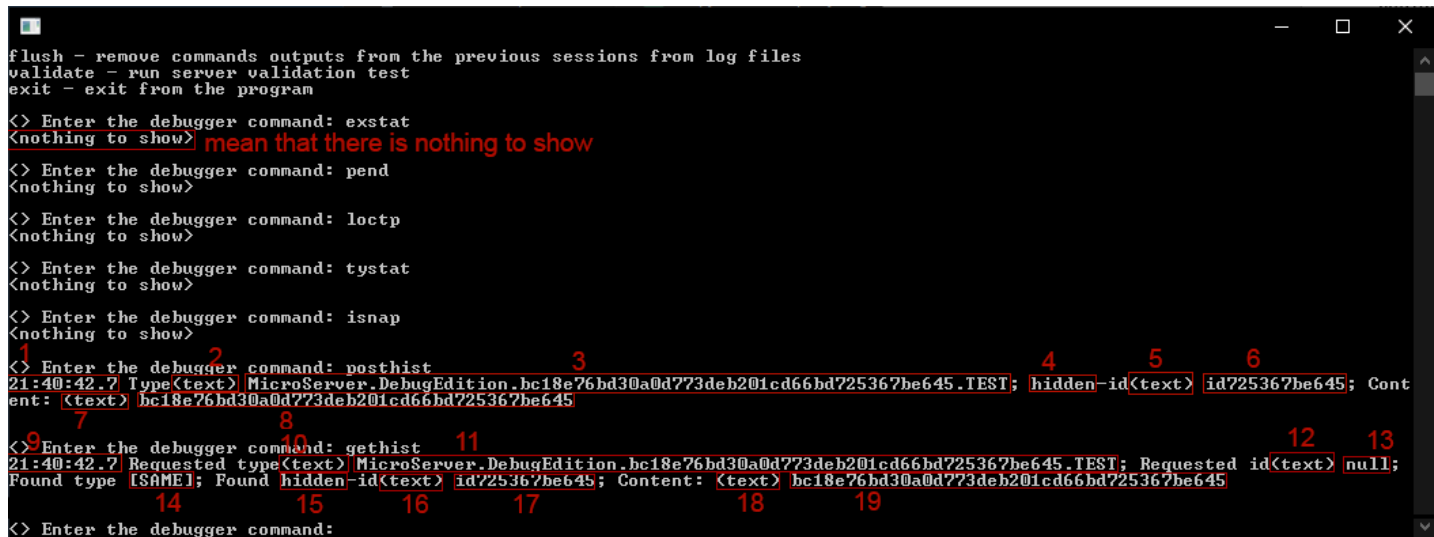
<> Enter the debugger command: tystat
<nothing to show>

<> Enter the debugger command: isnap
<nothing to show>

<> Enter the debugger command:
```

In the top picture at the top, you can see that the GET-JOB and POST-JOB addresses were automatically taken from the files (see Getting Started). It will also be appropriate to mention here that the monitoring program does not distinguish between upper/lower case letters and ignores whitespace characters before and after the command.

Now it's worth considering the output of the posthist and gethist commands.:



```
flush - remove commands outputs from the previous sessions from log files
validate - run server validation test
exit - exit from the program

<> Enter the debugger command: exstat
<nothing to show> mean that there is nothing to show

<> Enter the debugger command: pend
<nothing to show>

<> Enter the debugger command: loctp
<nothing to show>

<> Enter the debugger command: tystat
<nothing to show>

<> Enter the debugger command: isnap
<nothing to show>

1 <> Enter the debugger 2 command: posthist 3
21:40:42.7 Type<text> MicroServer.DebugEdition.bc18e76bd30a0d773deb201cd66bd725367be645.TEST; 4 hidden-id<text> 5 id725367be645; 6 Content: <text> bc18e76bd30a0d773deb201cd66bd725367be645

7 <> Enter the debugger 8 command: gethist 9
21:40:42.7 Requested type<text> MicroServer.DebugEdition.bc18e76bd30a0d773deb201cd66bd725367be645.TEST; Requested id<text> 12 null; 13 Found type [SAME]; Found hidden-id<text> id725367be645; Content: <text> bc18e76bd30a0d773deb201cd66bd725367be645

14 <> Enter the debugger 15 command:
16
17
18
19
```

- It's worth starting with posthist, where (red box numbers):

- 1 - the time the packet arrived at the server (more accurate time is recorded in the file)
- 2 - description of the representation of the type name: in this case - text, but if the type were an unprintable string, in field 3 bytes in hex format would be displayed, and in field 2 - "(hex)".
- 3 - the name of the type in some representation (in the form of a string in this case). The debugger sends a package of this type to the server during validation.
- 4 - means that the id of the package is hidden
- 5 - by analogy with 2, but in relation to field 6
- 6 - the id that the received package has (although it is hidden, but IT IS)
- 7 - by analogy with 5 and 2, but slightly more values: there can be not only "(text)" and "(hex)", but also "[boolean]", and "[NULL]", "JSON.Object" or even "[...].Length=" (after the equal sign is the size of the array), etc. All this diversity is due to the fact that the content field of the basic content structure can contain anything (within the framework of the JSON format), but not everything can be displayed in the terminal without littering it completely. Therefore, sometimes field 8 may be missing, and sometimes the text does not actually show text ("[boolean]", for example). From the parentheses, you can understand what the matter is with, the *text is always preceded by parentheses*. There is a case where field 8 is missing: this happens if content contains a number.
- 8 - in this case, this field contains text. It is this text that the debugger sends to the server for further verification (after receiving).

- Go to gethist, where (red box numbers):

- 9 - the time when the packet was sent from the server to the client. It seems that it coincides with 1, but it is not, just the precision of the number displayed on the screen is not enough to draw such a conclusion, however, the result saved to the file has the maximum precision.
- 10 - by analogy with 5 and 2, but in relation to 11
- 11 is the **requested** type (requirement), that is, the one that the client (debugger) requested from the server. It is the same as 3.

12 - by analogy with 11, 5 and 2, but in relation to 13

13 is the **requested** id (requirement), which is "null" (string), and the string "null" means that the client does not care about the package id (see MicroServer documentation: the very end of 4 section "Communication protocol with MicroServer"). If id 6 had been specified in the request (here), the package would not have been issued, since although the package id and the requested one are the same, the package id is hidden, which means it is not available for requests.

14 is a special word that means that the type of the issued packet matches the requested type (the packet was received by type).

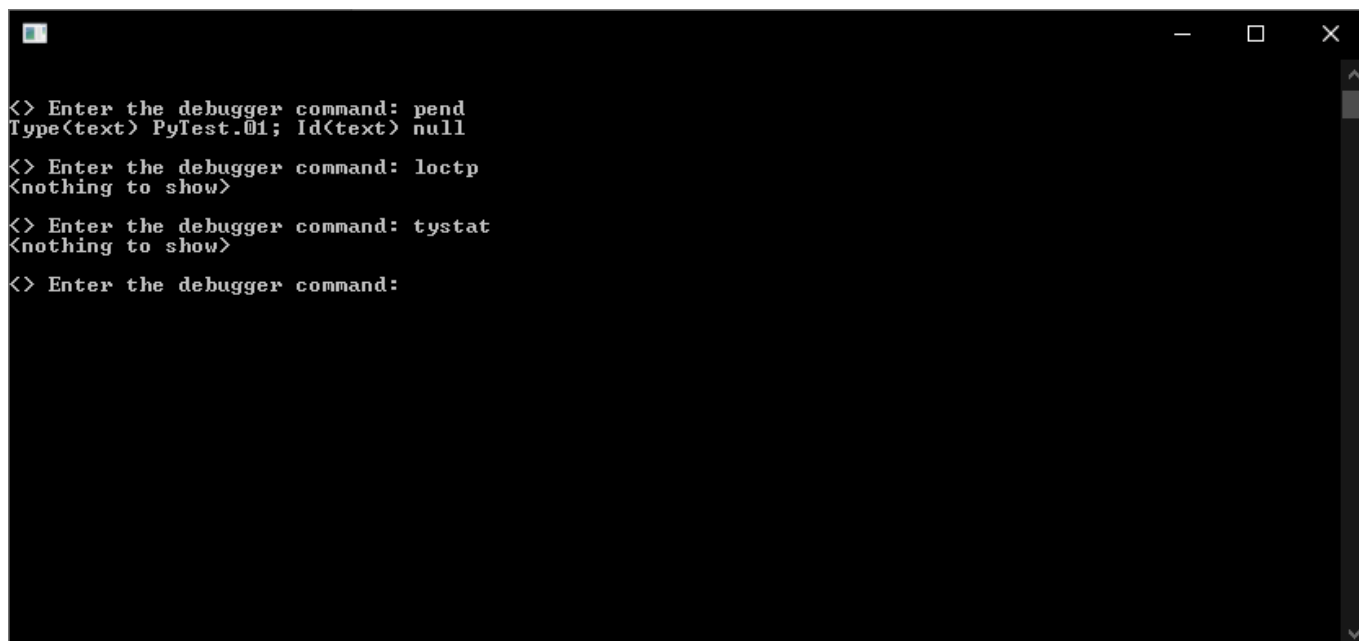
15 is essentially a repetition of point 4.

16 - repeat step 5 (What else can you expect? The package is the same!)

17, 18, 19 - repetitions of points 6, 7 and 8, respectively, since we are talking about the same package.

It should be noted that posthist and gethist, if used correctly, are very powerful tools for debugging service communications: with their help, you can find out exactly when which packet arrived (and with what content), by what request and what time it left the server..

- Now we will give examples of the "states" of the server data, which are closer to real ones:

A screenshot of a terminal window with a dark background. The window has standard window controls (minimize, maximize, close) in the top right corner. The terminal shows a series of debugger commands and their outputs. The first command is 'pend', which outputs 'Type<text> PyTest.01; Id<text> null'. The second command is 'loctp', which outputs '<nothing to show>'. The third command is 'tystat', which also outputs '<nothing to show>'. The fourth command is just 'Enter the debugger command:' without an output yet.

```
<> Enter the debugger command: pend
Type<text> PyTest.01; Id<text> null

<> Enter the debugger command: loctp
<nothing to show>

<> Enter the debugger command: tystat
<nothing to show>

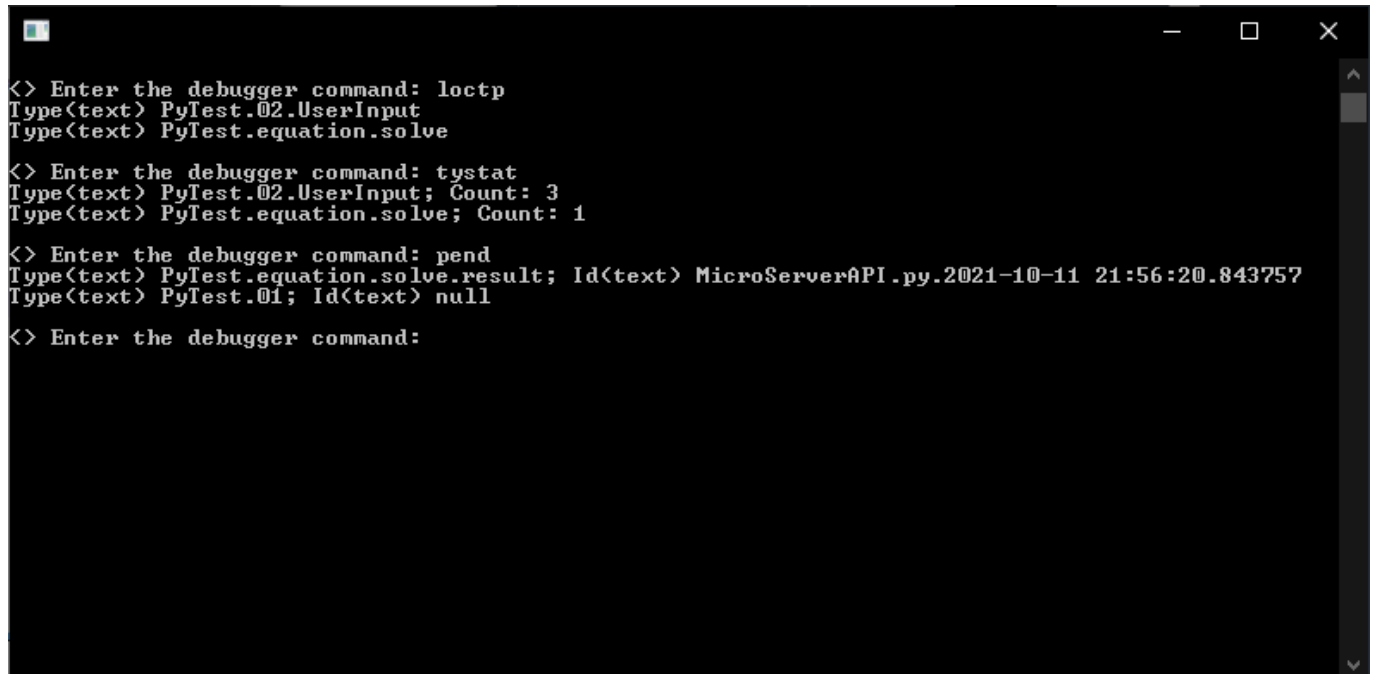
<> Enter the debugger command:
```

As you can see, not a single package is stored on the server, but there is one request (in reality, there can be tens of times more requests). This means that the microservice system is waiting for data to arrive from somewhere outside (possibly from the user). After some service sends a packet for processing, work will begin, but for now, all services are just waiting.

The output of the "pend" command contains information about each active request at the time of the command execution: what type of package and with what id the service is waiting. The example shows that the service does not care about the package id, but it does care about the package type (the type is not "null"). For this particular service, the server expects a packet with the "PyTest.01" type and any id (possibly even hidden).

Note that the `loctp` command was executed following the `pend`, although it is less informative than `tystat`. This is due to the fact that the `"loctp"` request is faster than `"tystat"`, which can be noticeable when many different types are stored on the server (more than 100, for example, and some of them are in external storage). If you don't know at first how many types are stored, it is better to run `"loctp"`.

- And how to understand that something is going “wrong”? In this case, consider a "reactive" system, i.e. one that does nothing by itself, but reacts to external data.

A screenshot of a debugger window with a dark background. The window title bar shows standard OS controls (minimize, maximize, close). The text inside the window shows the following sequence of commands and their outputs:

```
<> Enter the debugger command: loctp
Type<text> PyTest.02.UserInput
Type<text> PyTest.equation.solve

<> Enter the debugger command: tystat
Type<text> PyTest.02.UserInput; Count: 3
Type<text> PyTest.equation.solve; Count: 1

<> Enter the debugger command: pend
Type<text> PyTest.equation.solve.result; Id<text> MicroServerAPI.py.2021-10-11 21:56:20.843757
Type<text> PyTest.01; Id<text> null

<> Enter the debugger command:
```

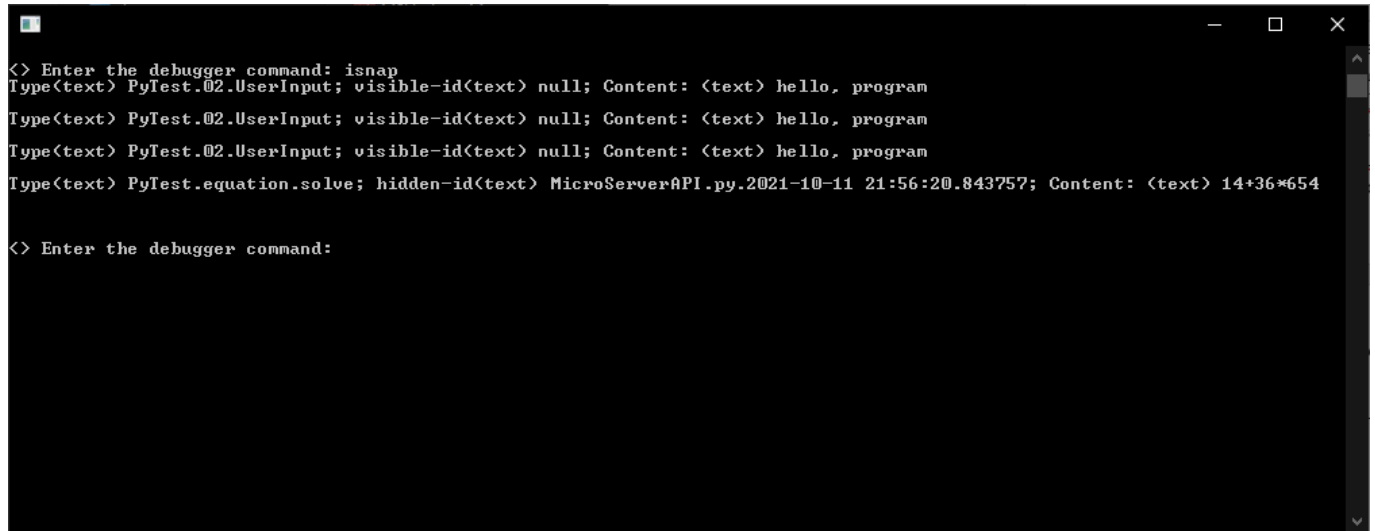
After executing `"loctp"` we see that the types are not 100 or even 50. However, let's say we have only 4 services. If these are not render services (or something else that is loaded), then it is rather unusual that we managed to catch a situation where there are already 2 types on the server at the moment. Let's say you're lucky.

We execute the `"loctp"` command and see something suspicious: there are already 3 packages of the `"PyTest.02.UserInput"` type, which for a "reactive" system in most cases means that some of its services "fell" and more does not process data, therefore, it has already managed to accumulate 3 packages for processing. If it goes on like this, then the number of packets can exceed thousands and, depending on the characteristics of the server host, at one time or another lead to a server “crash”. Here one could use a "crutch" in the form of external storage, which would take the surplus packages and store them somewhere (in special cases, like this, the external storage could start deleting the excess packages).

We also see the type `"PyTest.equation.solve"`, of which there is 1 package. Well, at least not three...

Now it would be nice to check the requests: we execute `"pend"` and see that 2 requests are active, one of which - `"PyTest.01"` - is waiting for data to be processed (the service is working properly), and the other, apparently, from the service that made the functional request: it sent data of type `"PyTest.equation.solve"` for processing, and now waits for the result of processing data with type `"PyTest.equation.solve.result"` and id `"MicroServerAPI.py.2021-10-11 21:56:20.843757"`. If you execute `"isnap"` in this configuration, you can know for sure if this is the case. You can execute the `"pend"` command again (2-5-10 seconds after the first execution, for example), and if `"PyTest.equation.solve.result"` with the same id is again present in the output, then the data processing service like `"PyTest.equation.solve"` also "fell"...

Well, let's execute the isnap command:



```
<> Enter the debugger command: isnap
Type<text> PyTest.02.UserInput; visible-id<text> null; Content: <text> hello, program
Type<text> PyTest.02.UserInput; visible-id<text> null; Content: <text> hello, program
Type<text> PyTest.02.UserInput; visible-id<text> null; Content: <text> hello, program
Type<text> PyTest.equation.solve; hidden-id<text> MicroServerAPI.py.2021-10-11 21:56:20.843757; Content: <text> 14+36*654

<> Enter the debugger command:
```

The guesses are confirmed: firstly, a package of type "PyTest.equation.solve" is indeed part of the functional request. it has an id, which is also hidden, and secondly, we see that this package has not yet been processed (see id - it is still the same), which means that the service that processes these packages is clearly out of order.

And yes, maybe something is wrong in the running service, because it returns an expression (14+36*654) with a type name containing the word "equation" (translated as "equation"), so perhaps the problem is not even in services that have already fallen (in any case, not only in them), but in the one that is currently running, and is waiting for the result of processing its data. This service either delivered the wrong type to the packet (in the type equation, not an expression), or inserted the wrong data into the packet. A similar pipelined incorrect (probably) packet in the past could have caused another service to crash.

In the best traditions of the detective, the one who was less suspected turned out to be guilty...

It is not yet clear why "PyTest.02.UserInput" is needed. As the name suggests, this is some kind of input. The service that generates them is probably working (otherwise why didn't it crash after the first packet?). Many combinations are possible here (however, as in the previous output), so additional data is needed.

And, pay attention, all conclusions are made without any information about the functionality of the system: only based on the number of services (four), information from the debugger and the knowledge that the system is "reactive". Of course, these "conclusions" are, rather, "guesses" (since there is no data about the functionality of the system, about which services were responsible for what, etc. - perhaps the system just needs more running service instances), but such a result quite good (with minimal knowledge of the system).

Now it's time to use heavy artillery (the "gethist" command) to trace the history of receiving packets from the server, but... I won't do that, since I didn't organize a beautiful crash, but wrote a stub to demonstrate how to work with the debugger. But the example is beautiful :)

In a real situation, there will be much more data (since at least there will be more information about services), so the reflections will be more substantive (less guesswork and "deductive method").