# Microservices.MicroServer documentation

## 1. Addresses

To access the server, 2 addresses are used, originally set in the code and requiring recompilation to change.

The first address is the address for GET requests, the second is for POST requests. This separation is not necessary, however, it was made to further differentiate between post requests and data requests. If necessary, these addresses can be combined, thereby simplifying the request validation code.

For example, the following addresses will be used:

- GET: http://localhost:8080/microserver/get-job
- POST: http://localhost:8080/microserver/post-job

## 2. Server editions

The server is available in two editions: main and debug. The main version is designed to work in the final product, the debug version is intended for use in the product development process, since it has an extended set of commands that helps with debugging the interaction of micro-services. Due to the debugging functionality, this edition is somewhat less productive than the main edition.

## 3. Server goal

It is assumed that a certain service will request some data from the server (for processing), and then send the result to the server. Each individual service does not have to worry about the global route of its data, since the server takes over the transfer to the required service. The transfer to a particular service is carried out mainly depending on the type of data of interest to the service (the type is reported by the service). Thanks to this scheme, it is possible to increase the performance of the final product by increasing the number of instances of a slow service connected to the server.

Services send data to the server, and also request data from it by attribute (type or identifier) - this is how the interaction with the server is built.

## 4. MicroServer's communication protocol

Only JSON (JavaScript Object Notation) is used to exchange information with the server. Since JSON is a text format, the messages from this server are quite human-readable.

All data is sent to the server via a POST request. Any POST request to the server must be based on the following JSON structure (called "basic content"):

```
{
    "id":<string>,
    "visibleId":<bool>,
    "type":<string>,
    "content":<object>
}
```

- The "content" field contains directly the data that needs to be transferred. Any JSON type can be here (array, null, true/false, object, number).
- The "type" field contains a string indicating the type. This line tells the server and the service how to work with the data in the "content" field (there is a situation when the server still processes the "content" field, and does not blindly pass it to the service, but more on that later). It should be said that the **server works with UTF-8**, so the set of allowed characters is quite large.
- The "id" field is necessary to indicate the belonging of one or another part of the data (basic content) to any request. Please note that it is a REQUEST, not a service. There could potentially be an unlimited number of basic content with the same id, but this is wrong. Id (identifer) is needed so that the service, after placing a task (sending data to the server), can later request (by "id") the result of processing the task that it posted, and not another with the same type ("type"). Essentially, "id" is the return address.
- The "visibleId" field is required to prohibit the retrieval of published data by id. Thus, if the field "visibleId" for some data is false, the service can request this data only by its type ("type"). This behavior is needed to implement "functions". The thing is that the server does not store data about where what data came from: it just stores the information that came in. You can just as freely request information from the server. Let some service N want to request processing of some type of data, and then continue working with already processed data. To do this, the service generates an id in order to find exactly its own result (and not someone else's). After that, it sends data to the server and waits for the required id to appear on the server. But stop! After all, such an id already exists and is in the basic content just sent from this service (N) to the server! As a result, the service receives as a result what it sent. A pointless waste of time. It is in this situation that specifying the "visibility" of the id helps. If service N specifies that the id of the packet should not be visible, then no one (and he himself) will be able to receive this particular data packet by id. But the processing service R will easily receive this package for the requested type. After processing, it will form a new data packet with the same id, but already visible; will send this packet to the server. And after that service N will receive its processed data. By the way, the R service could "delegate" the processing of received packets further - to another W service (make a kind of pipeline), specifying the required type in the basic content, leaving the old id invisible. Then the W service would have finished processing and sent basic content with a visible id to the server.

It is very important to choose a unique type name ("type"). Since the types are "global" for the whole system (they can be used by any services), it is very important to avoid collisions (when the same name is used by different micro-services for different data types). Therefore, it is advisable to add the name of the service that creates data of this type to the type name, or something else. There are no restrictions on the type string: it can even be empty, however, for the stable operation of the system, it is important to ensure that there are no collisions.

Also an important point: JSON-value null for the server is equivalent to the string "null". This is a special value that can be used both when sending data and when requesting it, however, when sending such a value is normal (for type or id), but with a restriction: a packet

cannot have "type" equal to "null" at the same time and "visibleId" is "false". This is due to the fact that when requesting data, the string "null" has a special role - it means that the service does not care about the value of the packet field, if the value "null" is given in the corresponding field during a GET request. For example, if a type is specified during a request, but the id is "null", then any packet with the same type will be returned in response (id will not be checked) - *this is the most optimized type of request in MicroServer.*

If both the type and id are equal to "null" at the same time, then during the request it will be considered that "null" in the id field is a full-fledged identifier, and the server will look for a package whose id field will also be "null". This is necessary so that the response from the server is at least a little meaningful, and not the first packet in the list of types.

## 5. Server commands and "external" package storage

Sometimes a situation may arise in which there may be too many packages on the server. This can happen, for example, if the sent packets are not used further by any service, but simply accumulate. The server does not perform the removal of "surpluses", but sometimes it is still necessary, therefore there is a set of 3 commands that allows a certain service to download the surplus from the server, becoming an external storage of packages. Such a service periodically executes requests for a list of overflowed types, as well as requests for surplus.
The following commands have been implemented for the operation of external storage:

- [GET]: MicroServer.25367be645.ExternalStatus
- [GET]: MicroServer.25367be645.FetchOverflow
- [POST]: MicroServer.25367be645.CompensateUnderflow

How is the command sent to the server? This happens in the same way as a request for a data packet - via a GET or POST request, but with a strictly defined (special) value of the "type" field, and is processed by the server directly (without transferring to any service). The value of the "id" field will be ignored.
It is important to understand that services cannot use reserved type names for their own purposes. A list of reserved type names will be given towards the end of the documentation.
All fields are passed through a query string, for example: http://localhost:8080/microserver/get-job?type=MicroServer.25367be645.ExternalStatus
It is important to note that the query string has a limited set of allowed characters, so the rest of the characters must be URL-encoded.
The first two commands from the list are executed via a GET request, the last one via a POST request. The command executed via a POST request is written not in the query string, but in the basic content structure like this:

```
{
    "id":null,
    "visibleId":false,
    "type":"MicroServer.25367be645.CompensateUnderflow",
    "content":<object>
}
```

In this case, the content is almost the same as a GET request: id is ignored, visibleId too (since this command is processed directly by the server). The "type" field contains the command, but the "content" field is an array of basic content.
Now we come to, directly, the functions of the previously given commands.

## 5.1. MicroServer.25367be645.ExternalStatus

A command with this name (in the text we will call it ExternalStatus). As you may have noticed, part of the type (command) name is the server name ("MicroServer"), then pseudo-random bytes follow, and only then is the command (type) name. All characters in the name are allowed in the URL, so no encoding is required.
As a result, the server returns a JSON array of the following plan:

```
[
    {
        "type":<string>,
        "underflow":<bool>,
        "overflow":<bool>
    },
    <...>
]
```

The array consists of elements called "external status". The array can contain several elements, or it can be empty. The symbol "<...>" means that in its place there may be several more external status elements. The comma is placed only between elements: no comma is placed after the last element (according to the JSON rules).
The "type" field indicates which type is being considered.
The "underflow" field contains a boolean value indicating whether to send additional 32 elements (you can send less or more, but ideally - 32). It is not worth sending more than 127, since the server will definitely reject such a request. The sent items are saved in the internal storage of the server.
The "overflow" field contains a boolean value indicating whether the surplus of the last items in the queue should be taken to external storage. The server can send any number of items, and the external storage must accept all of them, since at the time of sending these items have already been deleted from the server's internal storage. If the external storage cannot accept all the basic content elements, the unaccepted ones will be irretrievably lost (this should not be for the stable operation of the system).
The underflow and overflow fields cannot be true at the same time.
After receiving a response to the ExternalStatus request, external storage, depending on the underflow/overflow indicators, must execute other commands, and which is the story in the next subsection.
If the external storage will execute ExternalStatus requests to the server quite often (for example, once every 5 seconds), then the number of elements in the surplus should be close to 32 (not less than 33, but not much more).

## 5.2. MicroServer.25367be645.FetchOverflow

Execution of this command is permissible only if the collection of surplus from the server is allowed (makes sense) for a specific type. To call it, the query string must have a “type” field with the value “MicroServer.25367be645.FetchOverflow” and an “id” field with the name of the type, the surplus of which needs to be collected. That's right: the id parameter must be specified, while the type name must be placed in it. For example: http://localhost:8080/microserver/get-job?type=MicroServer.25367be645.FetchOverflow&id=InterestedType
If in response to such a request a status code of 200 is received, then the content will be a JSON array of the following plan:

```
[
    {
        "id":<string>,
        "visibleId":<bool>,
        "type":<string>,
        "content":<object>
    },
    <...>
]
```

As you can see, the array consists of an arbitrary number of basic content elements. This data will be taken from the *front* of the queue (in order to more quickly give way to newer (relevant) packages).

### 5.3. MicroServer.25367be645.CompensateUnderflow

As stated earlier, this command is invoked by a POST request. It has the following form:

```
{
    "id":null,
    "visibleId":false,
    "type":"MicroServer.25367be645.CompensateUnderflow",
    "content": [
        {
            "id":<string>,
            "visibleId":<bool>,
            "type":<string>,
            "content":<object>
        },
        <...>
    ]
}
```

The id field is ignored, visibleId too (since this command is processed directly by the server). The "type" field contains the command, the "content" field is an array of basic content, which need to be moved to the *end* of the queue (so that newer (actual) ones can pass earlier) of the server's internal storage. It is important that the types of all packets match, otherwise the request will be rejected by the server. Also, the request can be rejected if as a result of its execution there is an excess of packets of this type. That is why it is recommended to send no more than 32 packets in one CompensateUnderflow call.

## 6. Debug commands (for debug edition)

It is important to note that any debug command is inherently diagnostic, that is, it cannot affect the operation of the server or data in the internal storage in any way. Debugging commands help in debugging micro-services, showing the developer which requests (for which packages) are relevant at the moment, what is stored on the server at the time of the command execution, and what data had arrived/left by the time the command was executed. All debug commands ignore the is in the query line. List of commands with descriptions is below.

- MicroServer.25367be645.DebugEdition.getInternalStorageSnapshot

This command queries the complete contents of the server's internal storage. In response, all packages that are currently stored on the server are received (they are not deleted from the server). It is not guaranteed that the resulting array will reflect the actual order of packets

in the queue (you should not look at the order of packets relative to each other). The answer has the following format:

```
[
    {
        "id":<string>,
        "visibleId":<bool>,
        "type":<string>,
        "content":<object>
    },
    <...>
]
```

Note that **there are more economical commands** for the amount of data to transfer (listed below).

- MicroServer.25367be645.DebugEdition.getLocallyAvailibleTypes

This command asks for a list of types ("type") that are in the internal storage. Types located only in external storage are not counted. The answer has the following format:

```
[
    <string>,
    <...>
]
```

- MicroServer.25367be645.DebugEdition.getTypesStatistic

This command asks for a complete list of types that are available from internal storage and those that may be available from external storage, but are not available internally. The answer has the following format:

```
{
    <type string>:<unsinged integer>,
    <...>
}
```

Note that the answer to this command is not an array, but a key-value object, where key is a type and value is the number of stored objects of that type. At its core, it is a dictionary format.

- MicroServer.25367be645.DebugEdition.getPendings

This command asks for a list of requests that are active at the time the command is executed. The service sends the request to the server and waits for a response. With this command, you can see what type of data and with what id is expected by the services connected to the server. The answer has the following format:

```
[
    {
        "type":<string>,
        "id":<string>
    },
    <...>
]
```

- MicroServer.25367be645.DebugEdition.retrivePostHistory

This command asks for a list of completed POST requests. The debug version of the server saves all completed GET and POST requests. After executing this command, the north will return the list of completed requests, after which clears the list of POST requests on its side, so that the next similar command will not return what the previous one returned. The server has a limitation on the POST request log: no more than 512 of the latest received packets are stored. If there are more packets, the server removes the first 128 received packets from the log.

```
[
    {
        "datetime":<DateTime>,
        "content": {
            "id":<string>,
            "visibleId":<bool>,
            "type":<string>,
            "content":<object>
         }
    },
    <...>
]
```

As you can see, the array in the response contains an object that contains basic content (directly what the service sent to the server), as well as the date and time of receipt in the following format: «YYYY-MM-DDTHH:mm:ss.fffffff+HH:mm». Example: «2021-10-04T10:35:40.9449944+03:00». The "+" sign is followed by the time zone offset relative to UTC. The date and time itself are displayed for the current zone, not UTC.

**It is important to note that server commands are not logged, although they are essentially requests to receive or send data.**

- MicroServer.25367be645.DebugEdition.retriveGetHistory

This command asks for a list of completed GET requests. The debug version of the server saves all completed GET and POST requests. After executing this command, the north will return the list of completed requests, after which clearing the list of GET requests on its side, so that the next similar command will not return what the previous one returned. The server has a limitation on the GET request log: no more than 512 last packets received by services are stored. If there are more packets, the server removes the first 128 packets sent from the log.

```
[
    {
        "datetime":<DateTime>,
        "requestedType":<string>,
        "requestedId":<string>,
        "content": {
            "id":<string>,
            "visibleId":<bool>,
            "type":<string>,
            "content":<object>
         }
    },
    <...>
]
```

As you can see, the array in the response contains an object that contains basic content (directly what the service sent to the server), as well as the date and time of receipt in the following format: «YYYY-MM-DDTHH:mm:ss.fffffff+HH:mm». Example: «2021-10-04T10:35:40.9449944+03:00». The "+" sign is followed by the time zone offset relative to UTC.

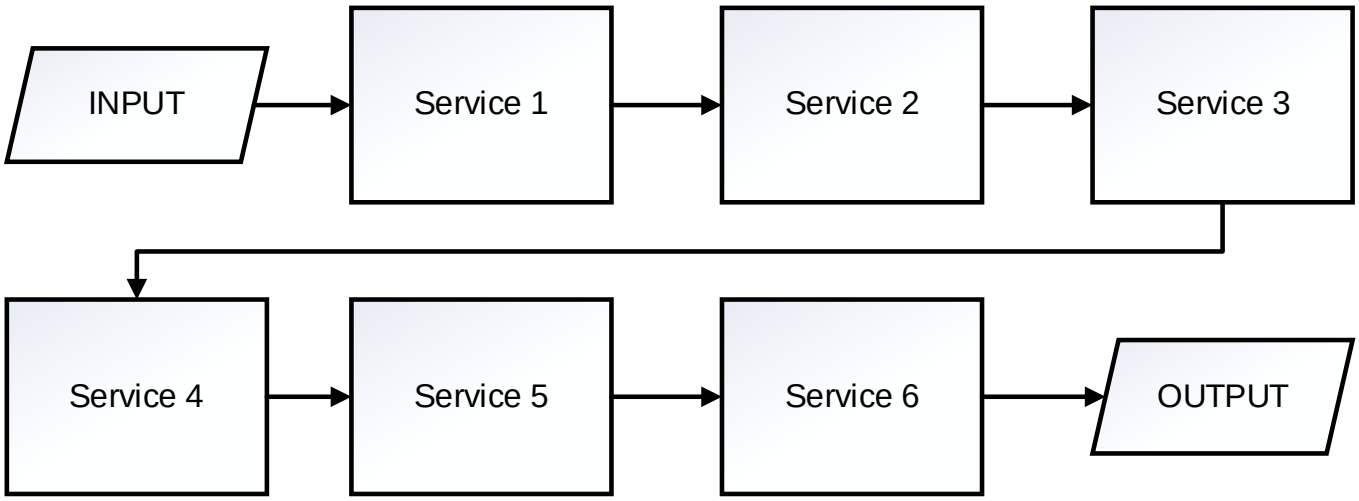
This command has a response similar to the response of the retrivePostHistory command, but contains 2 more additional fields: "requestedType" and "requestedId". These are exactly the "type" and "id" respectively, which were sent in the packet request to the server. If the request contained only a type, then "requestedId" will contain "null".

**It is important to note that server commands are not logged, although they are essentially requests to receive or send data.**

## 7. Client creation guidelines

- When creating client classes, it is important to distinguish between two main modes of system operation: conveyor and functional.

The first mode is simpler, but scaling it can be problematic. Within this mode, all services receive packages exclusively by type. It turns out that the services work as a conveyor, to the beginning of which one type entered, and another came out from the end, while each of the services (for one specific type) worked no more than once. When processing a data stream, this mode is significantly superior to the functional one. It can be schematically represented as follows:

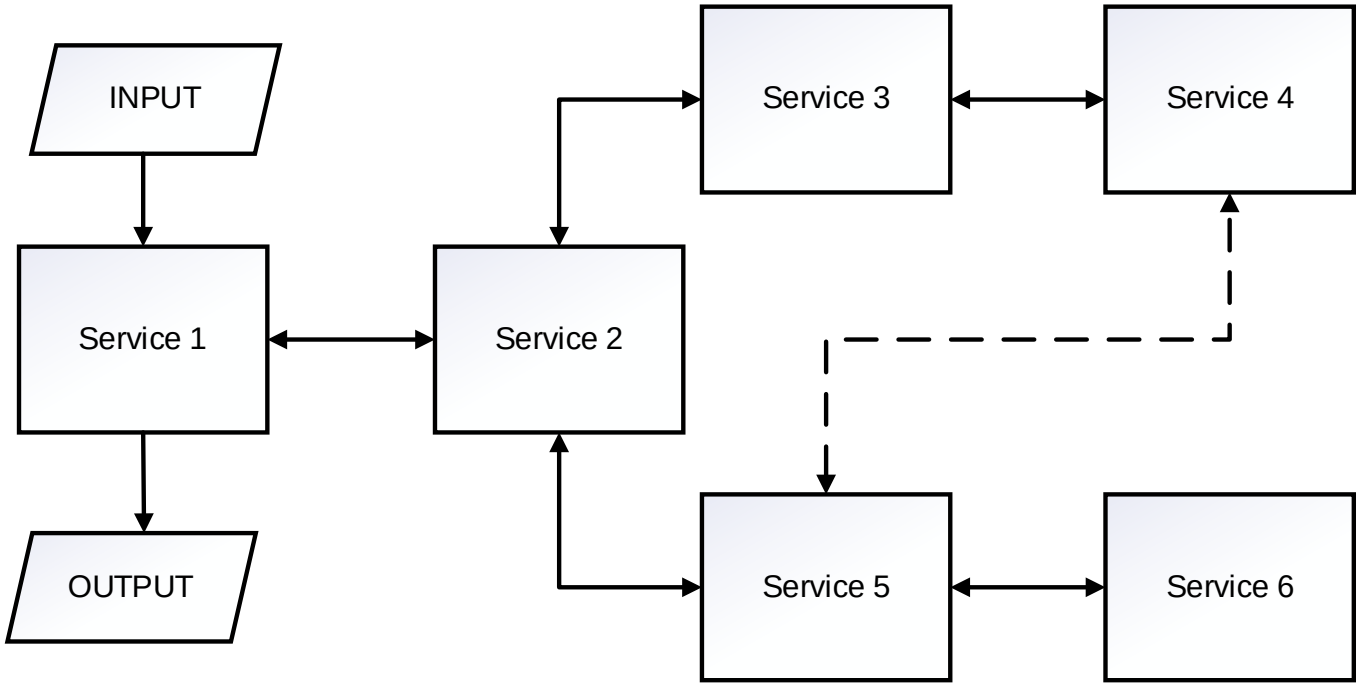



The functional mode works the other way around: data goes out and returns to the same service. If in the pipeline mode during processing the data always passed through, then in the functional mode the data passes through only in the final services (those that themselves do not call other services). In other services, at one of the moments, data would be sent to the server and waiting for a response, because of which these services would be unavailable (waiting for a response to return - like a normal function).

This mode is shown schematically in the figure below.

An important pole of this mode is its easier scalability and less cohesion of micro-services with each other. It also becomes possible to use the same service at different stages of data processing, which can also be convenient. This is shown in the figure with a dashed line. As you can see, input and output are performed from the same service, moreover, within the same request. Further, the first service calls the second and waits for its result, but the second does not do everything itself, therefore it calls first the third, and then the fourth,

which also do not do everything themselves, etc. As a result, it turns out that the execution time of the request by the first service is the sum of the execution time of all services.




It is also important not to confuse the user with the order of sending and receiving, because if the service first receives and then sends, then this service processes the data, but if the order is the other way around (first sends and then receives), then the data is processed by another service, and this one is "Customer".

- It's important to keep track of the mime types being sent. The fact is that the server does not accept POST requests in which the CONTENT-TYPE header does not contain "application/json". This is necessary because the server only accepts data in JSON format. In addition, the server also only sends data in JSON format, so the client can also set a request for "application/json" in the CONTENT-TYPE.
- It is important to keep track of the sending and receiving encoding. Server only works with UTF-8.
- It is important to encode the values of the "type" and "id" fields in accordance with the rules for the URL.
- It is important not to allow (throw an error) when the combination "type" is "null" and "visibleId" is "false". The server will not pass a packet with this combination (it will throw a 4xx error), so it is better to do client side validation and give a clear error message.
- It is important to follow the rules of the process of receiving data from the server. As part of this process, the client makes requests with a large TIMEOUT value (preferably about 60 seconds), to which the server responds with a delay of 25 to 50 seconds or less. The server responds with either 2xx or 408. A successful code means that the server has sent data, and a 408 code means that the connection timed out and a new request needs to be made. Such an organization will allow the service to react much faster to the appearance of new data on the server. If the client drops the connection before 25 seconds from the moment the connection was created, it can cause irrecoverable packet loss. If the client drops the connection after 25 seconds (but still, preferably 30 because the time measurements on the client and server may differ slightly due to delays), then nothing bad will happen - you can make a new request. However, it is much easier to trust the server to break the old connection.
- It is important not to use special types for custom tasks. The following types are considered special:
  - MicroServer.25367be645.ExternalStatus
  - MicroServer.25367be645.FetchOverflow

- MicroServer.25367be645.CompensateUnderflow
- MicroServer.25367be645.DebugEdition.getInternalStorageSnapshot
- MicroServer.25367be645.DebugEdition.getLocallyAvailibleTypes
- MicroServer.25367be645.DebugEdition.getTypesStatistic
- MicroServer.25367be645.DebugEdition.retrivePostHistory
- MicroServer.25367be645.DebugEdition.retriveGetHistory
- MicroServer.25367be645.DebugEdition.getPendings
- MicroServer.25367be645.GET_TIMEOUT_25

Any of the special types are NOT treated as a regular user-defined type, even if the revision is not debug. The last type in the list serves for the internal needs of the server and is not processed by the client in principle.

- It is important that the client treats all 2xx status codes as successful (not just 200), as the server returns other codes as well (for example 201).
- It is advisable to avoid the combination of "type" is equal to "null" and "id" is equal to "null", as this leads to unintuitive server behavior (the "null" string in the "id" field is considered to be a normal id for which the package is searched for).

It is allowed not to specify in the query string a field whose value is null (the server substitutes null by default), but you cannot skip both fields ("type" and "id"), since a combination of two unspecified fields is undesirable.

- It is desirable to formulate the functional request directly as a function that takes in the arguments the type of the input object, the type of the output object (optional), the input object, and returns the object.
- It is advisable to implement some protection and transfer of the id that came to the service from the server in a batch for processing to the function of sending the work result.
- It is also desirable to implement two functions for sending the result: one for sending the final result (with a visible id so that the service client picks up this package), and the second for pipelining (with a visible or invisible id, depending on how he came to this service).
- It is desirable to implement the "persistent" mode of the client operation, in which the client will not generate an error in the event of a connection failure (for example, if the server has not been started yet). This mode will allow you not to worry about the order of launching services and the server.

## 8. Protections and alternative servers

The MicroServer functionality is simple enough to be implemented as another server with expanded or reduced capabilities. In general, the server can be divided into three functional parts: package storage, package request handler, package receive handler.

In the simplest case, the storage can be represented by a simple list (not even a queue), in which structures of the form basic content will be stored. The store must be thread safe, so in the simplest case any operation on a list of items must be surrounded by a mutex block. The operation of inserting an item into the store is a simple addition to the end of the list. The operation of getting an element from the list – iterating over all the elements in the list until it finds one that matches all conditions ("type", "id", "visibleId"). It is also necessary to take into account the case when the required element is not found.

In the simplest case, the data query part checks to see if the URL query string parameters are specified and, if not, sets the fields to "null". In addition, this part contains a loop for requesting data from the storage. This cycle should be performed until either an item matching the request is found, or 25 seconds have passed. After exiting the loop, if an element

is found, this element is returned to the client, otherwise a 408 status code is returned to the client.

The part responsible for receiving data by the server, in the simplest case, should only check that the client has sent a packet in which "type" is not equal to "null" or "visibleId" is not equal to "false". If this is not the case, then the server should reject the packet, since a packet of this configuration cannot be requested from the server in any way.

Protections associated with checking the CONTENT-TYPE header (for the content of application/json) are highly desirable. It is also desirable for the server to set the CONTENT-TYPE header when sending packets.

The rest of the protections are set depending on the need according to the description of the protocol given above. MicroServer implements the following checks for requesting data from the server:

- Parameters in the query string are
- Parameters in the query string are either "type" or "id"
- The specified type name does not match any reserved special types

If a special type is specified, then the following checks are performed:

- The specified special type (command) is allowed

The following checks are implemented for sending data to the server:

- Is the title CONTENT-TYPE set
- Does this header contain the string "application/json"
- Whether the received packet has the expected signature (a set of fields with a specific data type)
- Is there a situation "type" is "null" and "visibleId" is "false"
- Type name in package does not match reserved special types

If a special type is specified, then the following checks are performed:

- The specified special type (command) is allowed
- Does the given content field match a certain expected signature
- Array has more than zero elements
- Item type names do not match reserved names