Server (shell) Exercise

General

Submission date: 15.12.22 **Submission mode: Single**

In this exercise, you will implement an HTTP server that can respond to incoming (HTTP) requests.
As part of the exercise you will be given a description of how the server operates, what are the endpoints and their structure (both request and response), and how they should behave.

You can write your code in every tech spec you wish. You will need to supply an executable file through which your application (i.e. the server) will be invoked.

As part of this exercise, you will need to work with a server shell framework. You need to investigate this topic on your own and you can select any framework that will suit your needs and the exercise requirement

Some of the data you will need to send/receive to/from the server will be in JSON format. We recommend seeking and using 3rd party libraries that help in the process of consuming and producing JSON data format.

There is no need to consider or support multiple threading behaviors in this exercise. You can assume requests will be invoked serially.

Your code will be tested by an automation tool that will invoke HTTP requests and will expect proper responses back from your server.
As such it is crucial to stick to the instructions and follow them precisely.

You are encouraged to test your own implementation using Postman or any other HTTP Client you like… (e.g. ex2 ?)

The Exercise

You will need to write a server that exposes calculator capabilities.
The server will support two modes of operation:

1. Independent calculation:
   In this mode, the server will receive all the information it needs to perform the calculation: the arguments and the operation to perform

2. Stack-based:
   In this mode, the server will receive the arguments on which the operation will work and will push them onto a stack.
   When the operation executes, it pops the amount of stacked arguments that it needs (different operations will need different amounts of arguments), performs the operation, and returns the result.
   The result is not pushed onto the stack; the stack is only for the arguments.
   If not enough arguments are stored in the stack it will return an error and the arguments won't be popped.

The operations you will need to support are (the names are case **in**sensitive):

1. **Plus:** binary operator ; x + y
2. **Minus**: binary operator; x - y
3. **Times**: binary operator; x * y
4. **Divide**: binary operator; x / y
5. **Pow**: binary operator; x ^ y
6. **Abs**: unary operator; |x|
7. **Fact**: unary operator; x!

All arguments are integer numbers only; all results are integer numbers only (in division take the integer part)

All endpoints will return the same result in a common json structure:

```
{
        result: <result of operation> : number
        error-message: <message in case of error> : string
}
```

For any response, only one of these fields are filled with data; the second one is ignored and can be omitted.

Special operations results

- Division: in case y = 0 the message will be: "***Error while performing operation Divide: division by 0***"; response code: 409 (conflict)
- Division: in case the result is a fraction, return only the integer part (e.g. 4 / 3 = 1)
- Factorial: in case X < 0 the message will be: "**Error while performing operation Factorial: not supported for the negative number**"; response code 409 (conflict)

In the rest of this document, the result will mean the result attribute on the response object; the error-message means the error-message on the response object.

The signs <...> represent placeholders for data that you should put. They are NOT part of the expected result.

The server will listen on port **8496 (i.e. localhost:8496/…)**
The automation expects your server to come up in a reasonable time: 10 seconds.
You should test and verify the uptime of your server process.

The endpoints:

1. **Independent calculation**
   Performs a full independent calculation.

   Endpoint: **/independent/calculate**
   Method: **POST**
   Body: json object:
   {
   arguments: [3, 4] // array of numbers
   operation: <operation name> // string
   }

   If the operation can be invoked (enough arguments): the response will end with 200; The result will hold the data.
   If there is an error the response will end with 409 (conflict); the error-message will be set according to the error:
   - No such operation:
     "Error: unknown operation: <operation name>"
   - Not enough arguments:
     "***Error: Not enough arguments to perform the operation <operation name>***"
   - Too many arguments:
     "***Error: Too many arguments to perform the operation <operation name>***"

2. **Stack: get stack size**
   returns the current size of the stack

   Endpoint: **/stack/size**
   Method: **GET**

   The response will end with 200; The result will hold the actual number of arguments currently in the stack (the stack size)

3. **Stack: add arguments**
   Pushes a list of arguments on the stack for future operations. The arguments are given in a list and are pushed to the stack according to their order (that is, the last item on the list will be the first item to pop from the stack)

   Endpoint: **/stack/arguments**
   Method: **PUT**
   Body: json object:
   ```
   {
       arguments: [3, 4] // array of integer numbers;
                         // after the operation, 4 will be at the head of the stack
   }
   ```

   The response will end with 200; The result will hold the actual number of arguments currently in the stack (the stack size)

4. **Stack: perform the operation**
   Performs an operation on the stack's arguments.
   The arguments are consumed from the stack. For operations that are defined with more than one argument (e.g. minus: x - y) the topmost item in the stack will be the first argument (here 'x') and the next item after it will be the second argument (here 'y').

   Endpoint: **/stack/operate**
   Method: **GET**
   Query Parameter: **operation**. Value <operation name>

   If the operation can be invoked (enough arguments): the response will end with 200; The result will hold the calculation result. The stack will pop out the number of arguments that were used.

   If there is an error the response will end with 409 (conflict); the error-message will be set according to the error:
   - No such operation:
     "Error: unknown operation: <operation name>"
   - Not enough arguments:
     *"**Error: cannot implement operation <operation name>. It requires <operation arguments count> arguments and the stack has only <total arguments in the stack> arguments***"

   Note: in that case, the arguments ARE NOT popped out of the stack

5. **Stack remove arguments**
   Pops the given amount of arguments from the stack.

   Endpoint: **/stack/arguments**
   Method: **DELETE**
   Query Parameter: **count**. Value: total arguments to pop from the stack

   If the operation can be invoked (enough arguments): the response will end with 200; The result will hold the actual number of arguments currently in the stack (the stack size after performing the deletion)

   If the operation cannot be invoked (not enough arguments): the response will end with 409 (conflict); the error-message will be:
   *"**Error: cannot remove <arguments count> from the stack. It has only <total arguments in the stack> arguments**"*

   Note: In that case, the arguments WON'T BE removed from the stack!

What to Submit

You should submit a zipped file that holds your implementation along with a file called **run.bat**. The file should hold all the commands needed to run and execute your code.
Please follow these guidelines when writing your **run.bat** file

If your code relies on other 3rd parties to run, you should prepare them beforehand, and place them on the zipped file itself (e.g. node_modules in Node JS).
DO NOT assume that as part of the automation there will be time to download 3rd parties or any internet connection at all.

You can zip using ONLY zip, or rar (DO NOT use 7z, arj, gzip, cpbackup or any other means)
Please note that your zip is expected to be 'flat', that is all files are located directly inside it, not within an inner nested folder or something. (that is, after unzipping your submission, the folder will contain the **run.bat**)