Logs Exercise

General

Submission date: 5.1.23 **Submission mode: Single**

In this exercise you will add a logging facility on top of the calculator server you created in the last exercise.

As part of the exercise, you will be given a description of which logs, at which level, should be plotted to which output.
All logs will have a declared name and output level, and you will need to create an additional endpoint that enables you to change a certain log level.

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 logging facility framework. You need to investigate this topic on your own and you can select any framework that will suit your needs and the exercise requirements.

Again, there is no need to consider or support multiple threads behavior 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 vs the server and will expect proper logging as function of these calls.
As such it is **crucial** to stick to the instructions and follow them precisely.

The Exercise

This exercise logic relies heavily on the *server exercise*. The same logic is preserved and you will need to enhance it with logs that will be added to it.

The server will listen on port **9583**

The server will hold a global request counter that will count how many incoming requests it gets (starting from 1). Each log line will have a reference to which request number it belongs (again, **there is no need** to pay attention to multithreading in this case)

Each log line will hold the below structure:
(**{}** symbols stand for a place holder marker. They are NOT part of the actual data):

{date-time} {log-level}: {log-message} | request #{request-number}

Where:

- date-time: date of the log. Format: dd-mm-yyyy hh:mm:ss.sss
- log-level: the level of the log line. **Capital case** (e.g. INFO, DEBUG)
- log-message: the actual log message (detailed below)
- request-number: the serial number of the request

You will need to define certain loggers that will be in charge of logging data on various places of the code. The loggers will output some files. All files should be placed in a folder called **'logs'** in the working directory folder. Check this guide for detailed verification about the folder and files location.
Also you can view this movie (No audio. Follow the subtitles) to gain another validation of it

Below you will find a list of requested loggers.
Do note:

- The {} marker states for dynamic data that you will need to place inside the log message
- Wherever there are multiple logs per action in different levels, the 'rule' is that the INFO message should be given **before** the DEBUG message (check log files example below)
- The logs messages described below are the content of the macro {log-message} described above

1. **Requests Logger**
   Name: request-logger
   Description: in charge of logging each incoming request of any type to the server
   Targets
      File: requests.log
      Stdout (The screen)
   Default Level: INFO
   Logs:
   INFO: "Incoming request | #{request number} | resource: {resource name} | HTTP Verb {HTTP VERB in capital letter (GET, POST, etc)}"
   DEBUG: "request #{request number} duration: {duration in ms}ms"

2. Stack logger

Name: stack-logger
Description: in charge of logging information on all the stack behavior
Targets
File: stack.log
Default Level: INFO
Logs:
The below list holds all detailed logs:

#	endpoint	Level	Log message
1	Get stack size	INFO	“Stack size is {stack size}”
2		DEBUG	“Stack content (first == top): [{stacks elements comma separated}]
3	Add arguments	INFO	“Adding total of {total args} argument(s) to the stack | Stack size: {stack size after addition}”
4		DEBUG	“Adding arguments: {arguments comma separated} | Stack size before {stack size before} | stack size after {stack size after}”
5	Perform operation	INFO	“Performing operation {op name}. Result is {result} | stack size: {stack size after operation}”
6		DEBUG	“Performing operation: {op name}({arguments separated by comma}) = {result}”
7	Remove arguments	INFO	“Removing total {total args} argument(s) from the stack | Stack size: {stack size after removal}”

In case of error or failure in any of the requests, you should log the below log message
Level: ERROR
Message: “**Server encountered an error ! message: {the error message you place on the resulted json response}**”

3. Independent logger

Name: independent-logger
Description: in charge of logging information on all the independent behavior
Targets
File: independent.log
Default Level: DEBUG
Logs:
The below list holds all detailed logs:

#	endpoint	Level	Log message
1	Independent operation	INFO	"Performing operation {op name}. Result is {result}
2		DEBUG	"Performing operation: {op name}({arguments separated by comma}) = {result}

In case of error or failure in any of the requests, you should log the below log message
Level: ERROR
Message: "**Server encountered an error ! message: {the error message you place on the resulted json response}**"

In addition, you will need to add an additional 2 new endpoints:

#	purpose	endpoint	method	Query param	response
1	Gets the current level of a logger	/logs/level	GET	logger-name : the name of the logger	Simple text. **Success**: The log level in capital case **Failure**: Message (one line): free text to deliver the essence of the failure
2	Sets the level of a logger	/logs/level	PUT	logger-name : the name of the logger logger-level : the requested level: ERROR, INFO, DEBUG	Simple text. **Success**: The log level in capital case **Failure**: Message (one line): free text to deliver the essence of the failure

You can follow the below sequence of requests and verify how each log file should look alike:

req #	verb	resource	Query param	body	stack	request	indepe
1	GET	/stack/size			INFO	INFO	DEBUG
2	PUT	/stack/arguments		{ "arguments": [2,3] }	INFO	INFO	DEBUG
3	POST	/independent/calculate		{ "arguments": [4,2], "operation": "divide" }	INFO	INFO	DEBUG
4	GET	/stack/size			INFO	INFO	DEBUG
5	PUT	/logs/level	logger-name=stack-logger&logger-level=DEBUG		DEBUG	INFO	DEBUG
6	GET	/stack/size			DEBUG	INFO	DEBUG
7	GET	/stack/operate	operation=fact		DEBUG	INFO	DEBUG
8	GET	/stack/operate	operation=minus		DEBUG	INFO	DEBUG
9	PUT	/stack/arguments		{ "arguments": [8,5] }	DEBUG	INFO	DEBUG
10	GET	/stack/operate	operation=minus		DEBUG	INFO	DEBUG
11	PUT	/logs/level	logger-name=request-logger&logger-level=DEBUG		DEBUG	DEBUG	DEBUG
12	PUT	/stack/arguments		{ "arguments": [2,3] }	DEBUG	DEBUG	DEBUG
13	GET	/stack/operate	operation=abs		DEBUG	DEBUG	DEBUG
14	DELETE	/stack/arguments	count=1		DEBUG	DEBUG	DEBUG
15	GET	/stack/size			DEBUG	DEBUG	DEBUG

See attached files for example of output for this sequence of requests:
stack.log | independent.log | requests.log

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**)