COSC 3P95 – Assignment 2 – Report

Robert Morabito
7093230
2023-03-25

System Overview

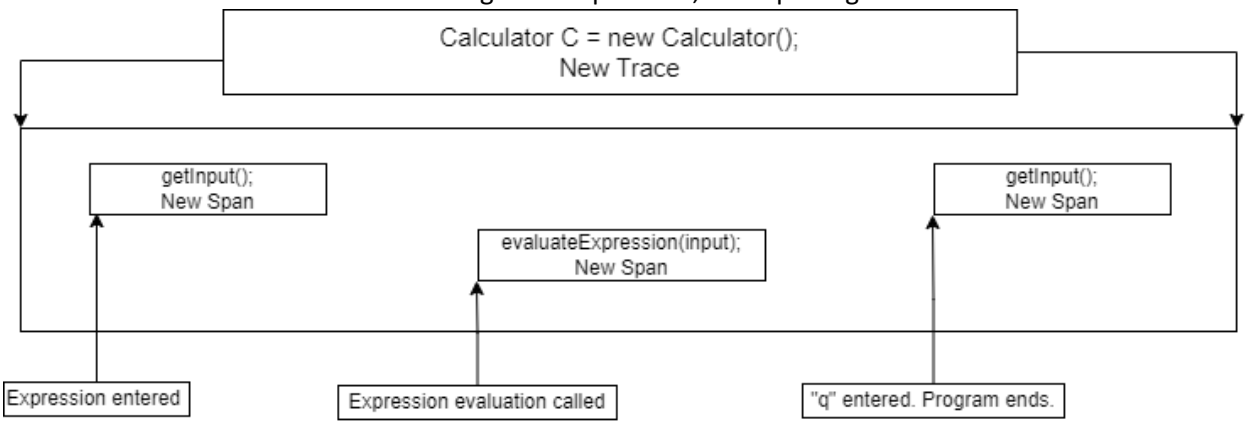
The system in place is one that I have built myself using JAVA and works as a simple calculator program. The system accepts a user input from the console as a string (e.g., “2+2*7-5”, “sin(3)”, “1.5*6.348”), then parses the string an evaluates the operations specified within. It is able to do operations with addition, subtraction, multiplication, division, exponents, modulo, logarithms, natural logarithm, sine, cosine, and tangent (note; currently sin, cos, tan, log, and ln operations overwrite any operations done before them). The results are then presented to the user via the console and the user has the option to exit using the command “q” or enter another operation to be evaluated.

Instrumentation Overview

Unfortunately, I was not able to get Open Telemetry to work as the integration process is expansive and the tutorials/guides are sparse. However, the way that I would have implemented my instrumentation, had I been able to integrate Open Telemetry, would have been the following.

Trace and Span structure

First, I would have created an Open Telemetry Tracer object and set a global tracer within the constructor for the calculator class, that way a trace is set every time the calculator class is called (in this case it would be once every run). Next, I would create Open Telemetry Span objects within each function and call the Open Telemetry span builder to start the span at the beginning of the function and end the span at the end of the function. With this structure, the entire calculator class is captured under a trace and each function call is captured under a separate trace. Below is a diagram that demonstrates how this would work in the case of a user entering in an expression, then quitting.

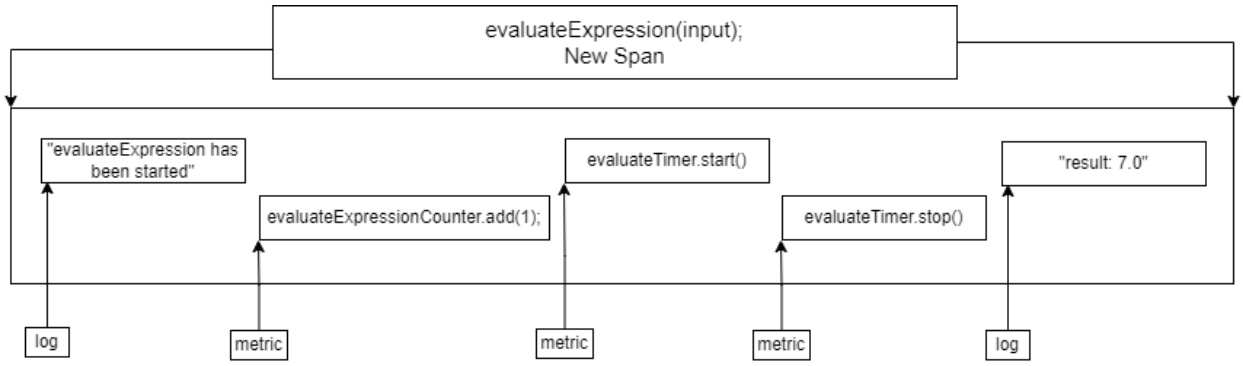


Logs and Metrics

Since the Open Telemetry structure would have been setup, I could integrate logs and metrics. For logs I would use Open Telemetry’s logger at 3 critical points; at the start each function I would have a simple string to outline the function has been called such as “getInput has started”, after the input is received I

would have a log to record what the user input was such as “input: 2+2*3”, and finally after the expression is evaluated I would have a log to record the result of the expression such as “result: 12”. With these 3 logs being recorded for within each span it would allow me as the programmer to evaluate if some functions are not being called or are being called in an incorrect order by evaluating the order in which they appear in the log, as well as being able to evaluate if the inputs that are received result in a correct output. This can be expanded further by creating another program that reads the inputs and outputs of each log and return true or false if the result is correct for the input (using another calculator program for evaluation) so certain problematic operations can be isolated.

For metrics I would focus on 2 key metrics that could effect this program; the number of times a function is being entered, and the execution time taken for each function. To record the number of times a function is entered I would create an Open Telemetry Counter object for each function as global variables and add to each counter with each time the function is entered. This would allow me to see specifically how many times each function was entered and cross reference with how many times it should have been entered (for example, getInput() is entered once per expression, so the counter should equal the number of expressions entered). Making this metric global allows me to see the number of times entered across the entire trace, not each span, which is important and ideally each function call should only be one span. To record the amount of time taken within each function, I would create an Open Telemetry Meter object and use it to create a Timer object. This would be global and would allow me to run Timer.Sample at the start of each function to start a new timer, which can then be stopped at the end of the function (before return) and would allow me to record the amount of time spent within the function. This is an important metric as it can reveal specific expressions that are bogging down the system (such as ones that use sin, cos, or tan as those require more parsing) which allows for refinement as well as revealing potentially poor overall performance which allows for rebuilding parts of the program structure. Below is an example of a span and all of the logs and metrics that are recorded within it.



With this instrumentation structure I believe that the majority of critical information is being captured and allows for a testing system to be built in the future such as one that uses random testing to generate data and statistical debugging to evaluate data.

Visualization

To visualize the information provided by Open Telemetry, I would implement Zipkin into the code. I would do this by first configuring Open Telemetry to write traces to Zipkin by changing the Open Telemetry exporter properties, and I would set up Zipkin to write to a localhost port. Then, to access and visualize the data I would simply go to the webpage that Zipkin is hosted on to view traces and spans. To

visualize logs and metrics, I would setup the Open Telemetry collector and set up the exporters for both metrics and logs within the pipelines to export to JSON files, this way the data is easier to process and import into a variety of different databasing and visualization tools.