IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol. 10, Issue.7, July 2022 E-ISSN: 2347-2693

Debugging Microservices with Pandas, PySpark using Actuators and Logs at Runtime

Sameer Shukla

Lead Software Engineer, Texas, USA

Author's Mail ID: sameer.shukla@gmail.com, Tel.: +1-480-754-9793

DOI: https://doi.org/10.26438/ijcse/v10i7.2730 | Available online at: www.ijcseonline.org

Received: 28/Jun/2022, Accepted: 15/Jul/2022, Published: 31/Jul/2022

***Abstract*—** Microservices architecture is distributed in nature and the expectation is the services in the architecture must be highly available and responsive. Services in the architecture can scale from 1 to 100s and the distributed architecture is complex, and the chances of failure are higher when services communicate to each other. The main advantage of microservice architecture is we can easily mix technologies depending upon the nature of service, if the service is CPU or IO bound then we can develop the service based on the language or framework of our choice, similarly if we have hundreds of services in our architecture than we can build a proper debugging system for our microservices using any platform / frameworks two such libraries are Pandas or PySpark. This paper focuses on creating our own debugging tool in the Microservices architecture using python-based libraries PySpark and Pandas and the concept of Actuators.

***Keywords*—**Microservice, Pandas, Spark, Actuator, SpringBoot, PyActuator, DataFrames

I. Introduction

Modern world applications are expected to be highly available and responsive, in the microservices architecture services are designed using Single Responsibility Principle and services are expected to have strictly one responsibility. The various Patterns within the microservices architecture forces the services to be atomic at all cost, like one database per microservice [3][4], Aggregator, CQRS etc. The major advantage which comes with this complexity is the scalability and faster development cycles as team can develop services in isolation. In the microservices architecture most of the time services communicates to each other, if services are developed using Choreography SAGA, then one service will produce an event which other service consumes and act. Similarly, if the services are REST based then one service will depends on the response of another. To ensure all the services in the architecture are alive microservice [3][4] are mostly comprises of cross-cutting pattern known as Circuit Breaker [2], the circuit breaker sits between request and response mechanism as proxy, when the service experiences failure the circuit breaker trips for some duration. To avoid the cascading failures in the architecture services even decorated with Bulkhead patterns [2] , but what if all the services are up but the responses are wrong or comprises of incorrect data or we need to debug why the service is responding with 404 but the data exists we simply need to take a count of total Not founds and total number of Ok's from the service. Or we need to examine the thread dump from the service programmatically as it's not easy to go through the thread dump manually. Imagine a log file generated by the individual service comprises of 1 GB in size it's not easy to examine the entire log file, we can utilize the PySpark DataFrame and keep applying filters on the DataFrame to check whether the specific failure or response code we are looking for exists or not, this makes debugging a lot easier we can create a Log Analyzer utility which consists of such programs and can be executed on-demand basis. If we want to find how many requests are served successfully by the service or how many are unable to, we can simply execute the Actuator endpoint and hand-over the JSON response to Pandas DataFrame and identify the count individually. In the up-coming sections we are going to understand what Actuators are, how to work with Pandas and PySpark to execute the responses as a DataFrame. If the

II. Related Work

Our Mission in this paper is to create an efficient debugging tool for the microservices architecture. The below figure showcases two services A [3][4][5] and B, A is dependent on B but for some reason B is returning in-correct response or not behaving as expected to debug microservices on the fly we need to rely on either Actuators or Logs, but debugging should be performed programmatically. First, we need to understand the concept of Actuators which comes in built-in with SpringBoot framework but if your services are written in Python based frameworks, please explore the library called PyActuators.

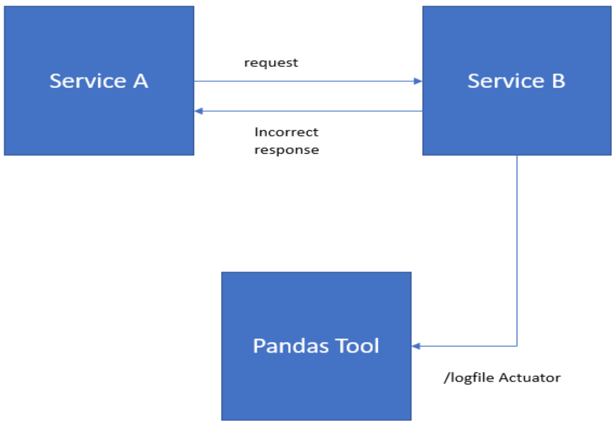
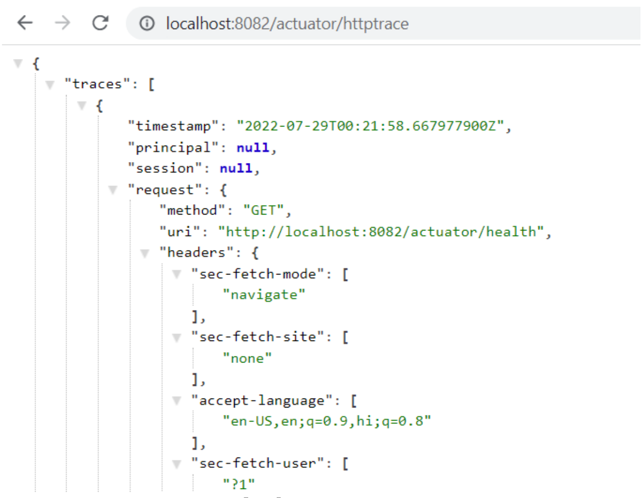


Figure 1. The tool is querying the actuator endpoint

Our microservice debugging tool will be using Actuators, Pandas, Spark, and simple Log file.

A. Actuators: Actuators[8][9] are the endpoints with-in the service that helps in monitoring the application, in nutshell Actuators helps in identifying what's going on with the service. With Actuators we will be able to check application health status, Metrics of all sorts, etc. Imagine we have a Cache in a service and we need to check the status of Cache like how much memory is free, how much is occupied, uptime etc or if we want to evict the data from the cache etc. The /cache endpoint provides the access to application cache, in a similar way there are many such useful actuator endpoints that helps in monitoring, some of them are /httptrace that provides information about HTTP request-response exchanges and /heapdump provides a heap dump from the JVM. Keep in mind these important information we get at runtime by directly making a REST calls to the actuators. The screenshot represents the information retrieved using /httptrace endpoint, where each section represents a meaningful detail, the "request" section gives details about the Type of rquest whether it is 'GET', 'PUT', 'POST' or 'DELETE', it also consists of 'timetaken' in the exchange which is extremely important, status gives info on response status like 200, 404, 420 etc.


localhost:8082/actuator/httptrace

```
{
  "traces": [
    {
      "timestamp": "2022-07-29T00:21:58.667977900Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8082/actuator/health",
        "headers": {
          "sec-fetch-mode": [
            "navigate"
          ],
          "sec-fetch-site": [
            "none"
          ],
          "accept-language": [
            "en-US,en;q=0.9,hi;q=0.8"
          ],
          "sec-fetch-user": [
            "?1"
```

Figure 2. Sample /httptrace actuator response

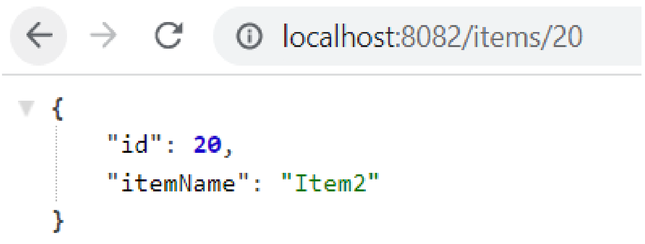
B. ***Pandas****:* Pandas is a Data Analysis Library which is fast as it's uses numpy [8], convinient and contains collections of lot of useful functions for data analysis purposes. Pandas efficiently handles large dataset and we can very efficiently customize or analyze the data from the dataset. In the debugging tool we are going to invoke an Actuator Endpoint and the response will be handed over to Pandas to analyze the response. Imagine a service is running in production and we need to analyze the http request-response exchange of entire day, it's not going to be easy to do it manually but with Pandas we need to perform the analysis on the DataFrame but not on the data.

C. ***PySpark****:* PySpark is an interface for Apache Spark [6] in Python, it's an open source framework [1] which is distributed [14] in nature and it's a library for real-time and used heavily for large data-set processing. PySpark is robust and can perform operations on billion records of rows in distributed clusters and it performs 100X times faster than anyother framework. In our Debugging tool we are going to invoke the Actuator for downloading the log file and handing over the entire logfile to PySpark for further processing. Our debugging tool we will be filtering all the '404 NOT FOUND' responses as well as filtering the clients which are calling our service, for ex: identifying how many requests received by the service from the 'localhost' or identifying how many of them is invoked from other Host.

III. MEHODOLOGY

In the paper we are going to explore two use-cases with the help of two kinds of actuators the /httptrace and /logfile, the request-response exchange analysis [10] we will be done using Pandas and the entire logfile analysis will be done using PySpark as the logfile will be much heavy in size.

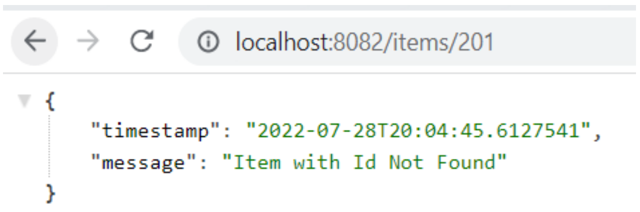
Debugging Service using Actuators and Pandas**:** The /httptrace provides details about HTTP request / response exchanges and this information we are going to give it to Pandas for further response analysis but programmatically, the other advantage working with Pandas is certain plotting libraries like Matplotlib seamlessly integrated with Pandas and we can visualize our dataset received from /httptrace. The service is running locally which returns a valid response and status as 200 if the data found.


localhost:8082/items/20

```
{
    "id": 20,
    "itemName": "Item2"
}
```

Figure 3: Successful call to a service returning data

If data not found, then it returns 404 with Data Not Found message.



```
{
    "timestamp": "2022-07-28T20:04:45.6127541",
    "message": "Item with Id Not Found"
}
```

Figure 4: Successful call but not data found

Below is the Python program which evaluates the dataset.

```python
import requests
import json
import pandas as pd
from matplotlib import pyplot as plt
response = requests.get('http://localhost:8082/actuator/httptrace')
json = json.dumps(response.json())
df = pd.read_json(json,orient ='columns')

for data in df.items():
    okList = []
    notFoundList = []
    for traces in data[1]:
        status = traces['response']['status']
        uri = traces['request']['uri']
        if(status == 200 and 'actuator' not in uri):
            okList.append(status)
        if(status == 404 and 'actuator' not in uri):
            notFoundList.append(status)


print('Successful Count:', len(okList))
print('Not Found List:', len(notFoundList))
```

```
Successful Count: 2
Not Found List: 3
```

The above code is simple more can be done it, we can even plot this data for better analysis and visualization, below code depicts that

```python
status_codes = ['200', '404']
exchange_counts = [len(okList), len(notFoundList)]
plt.bar(status_codes,exchange_counts)
plt.show()
```

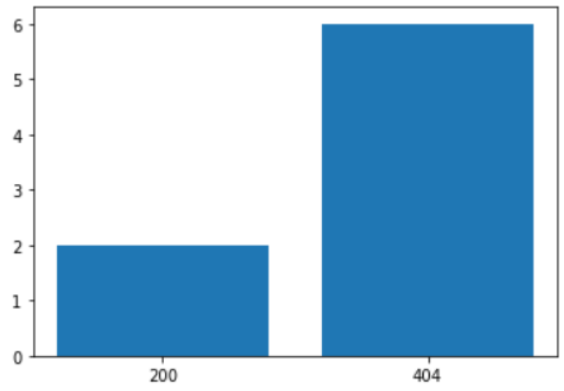


Figure 5: Request/Response exchange Data Visualization

***Debugging Service using Actuators, PySpark and Log file*:** This use case is PySpark [6] specific, again the approach remains the same in this use case /logfile actuator endpoint will be used. The /logfile provides access to the details of the application's log file. The /logfile endpoint upon invocation gives the contents of the log file and we are going to give this file PySpark.

```python
#---- SPARK ----
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, size

appName = "Microservices Logs Analyzer"
master = "local"

# Create Spark session
spark = SparkSession.builder \
    .appName(appName) \
    .master(master) \
    .getOrCreate()

spark.sparkContext.setLogLevel("DEBUG")

# Create DF by reading from log files
df = spark.read.text('app.log')
```

Figure 6: PySpark log analysis program

The application name is "Microservices Logs Analyzer" once the DataFrame is created we are ready to apply filters to analyze our logs. First, let's view the top 5 'NOT_FOUND' responses in a service

```python
df.filter(col('value').contains('404 NOT_FOUND')).head(5)
```

Figure 7: Block of code filtering 404 NOT_FOUND

```
Row(o.s.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND'),
Row(o.s.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND'),
Row(o.s.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND'),
Row(o.s.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND'),
Row(o.s.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND')
```

Figure 8: Response returned by the filter

Other analysis is to explore the requests from 'LocalHost'

```python
df.filter(col('value').like('%Host:%')).show(5)
```

```
+--------------------+
|               value|
+--------------------+
|Host: localhost:8082|
|Host: localhost:8082|
|Host: localhost:8082|
|Host: localhost:8082|
|Host: localhost:8082|
+--------------------+
only showing top 5 rows
```

Figure 9: Filtering request received by localhost

Requests received by the service other than Local host.

```python
df.filter(col('value').like('%Host:%')).filter(~col('value').like('%localhost%')).show()
```

```
+-----+
|value|
+-----+
+-----+
```

Figure 10: Filtering request received not by localhost

IV. Results and Discussion

The Actuators [8] provides lot of useful information, for example the /httptrace actuator is extremely handy it also gives us an ability to check the performance of every request and in case if we want to check the number of requests that are successful, but the performance is not up to the mark, say identify the number of requests taking more than a second to respond

```
number_of_requests = 0
for data in df.items():
    performance = []
    for traces in data[1]:
        timeTaken = traces['timeTaken']
        if(timeTaken > 1000):
            number_of_requests = number_of_requests + 1

print('Number of Requests taking more than a second to respond:', number_of_requests)
```

The /httptrace endpoint has an important parameter 'timeTaken' clubbed with every request and response which returns the time taken by that specific request. In the above program all we are doing is parsing the information and identifying how many requests have taken more than a second to respond. Program seems simple but the number of request/response exchanges handled by the service will be huge, but it's made extremely efficient by Pandas.

V. Conclusion and Future Scope

In this paper, the debugging is wired with the help of Actuators, but the tool can be enhanced to a greater extent. The paper also showcased a sample request / response exchanged can be visualized using the bar and graph charts. The ideal scenario would be to analyse the heap dump and thread dump programmatically as Pandas and PySpark DataFrame object provides various convenient functions for analysis purposes. The tool can be enhanced even further, In the microservice [3] we can also add a custom metrics like Gauge, Counter and Timer these are Prometheus specific metrics, the Gauge helps us in identifying the number of running threads within the application, the number of messages sitting inside a queue. The Counter metrics provided the information on fields like Total, the total number of requests processed, total number of items processed by service and Time provides information on the time taken by the method for request execution, these are all helpful metrics which should be analysed time and again. The tool is dependent on the endpoints, and it is completely decoupled, and it never sends a request to the business endpoint, but it sends a request to the actuator endpoint. Tool has endless possibilities for extension

Acknowledgment

I would like to express sincere appreciation to all those who contributed developing the Actuators / Pandas toolkit. Grateful for spring.io team for providing such valuable help and resources online which simplifies application creation to a great extent.

References

[1] Badidi, E. (2013) "A Framework for Software-As-A-Service Selection and Provisioning". In: International Journal of Computer Networks & Communications (IJCNC), **5 (3): 189-200, 2013.**

[2] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," ArXiv160905830 Cs, **Sep. 2016**

[3] Kratzke, N. (2015) "About Microservices, Containers and their Underestimated Impact on Network Performance". At the CLOUD Comput. 2015, 180, **2015.** https://arxiv.org/abs/1710.04049

[4] Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. (2009). Systematic literature reviews in software engineering–a systematic literature review. Information and software technology, **51(1):7–15, 2009.**

[5] Zimmermann, O. (2009). An architectural decision modeling framework for service oriented architecture design. PhD thesis, Universitat Stuttgart. **2009**.

[6] Nick Pentreath, Machine Learning with Spark, Beijing, **pp. 1-140, 2015.**

[7] Bryant, P. G. and Smith, M (1995) Practical Data Analysis: Case Studies in Business Statistics. Homewood, IL: Richard D. Irwin Publishing: **1995**.

[8] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. Information and Software Technology, **64:1–18, 2015.**

[9] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, pages 38:1–38:10, New York, NY, USA, 2014. ACM

[10] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, ¨ and A. Wesslen. ´ Experimentation in Software Engineering. Computer Science. Springer, **2012**.

[11] B. A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University and University of Durham, **2007**

[12] P. Kruchten. What do software architects really do? Journal of Systems and Software, **81(12), 2008**

[13] Kornacker, M. et al. Impala: A modern, open-source SQL engine for Hadoop. In Proceedings of the Seventh Biennial CIDR Conference on Innovative Data Systems Research, Asilomar, CA, **Jan. 4–7, 2015**

[14] Isard, M. et al. Dryad: Distributed data-parallel programs from sequential building blocks. In Proceedings of the EuroSys Conference (Lisbon, Portugal, Mar. 21–23). ACM Press, New York, **2007**.

AUTHORS PROFILE

Sameer Shukla has done Masters in Computers from Bangalore University, India in 2004. He is having 15 years of experience in Software Design and Development, Currently Working as a Lead Software Engineer in USA and his current expertise/interests are Distributed Computing, Data Analytics, Microservices, Functional Programming, Cloud Computing, Deep Learning, SQL, NoSQL, Big Data, Spark, Data Science