Jules Davoust
Vadim Bernard
Robin Lucas

PROJECT REPORT

Automated Container Log Management with Kafka and Spark for API Integration

Jules Davoust
Vadim Bernard
Robin Lucas

Table des matières

Introduction

In the current context of digital transformation, managing large volumes of data, also known as "Big Data," has become a major challenge for businesses and organizations. The ability to efficiently collect, process, and analyze massive amounts of data is essential for making informed decisions, optimizing operational processes, and delivering personalized, high-quality services. The project we have undertaken fits perfectly within this framework by addressing real-world issues related to the collection and processing of logs generated by containerized applications, a growing need in modern infrastructures based on microservices and containers.

With the rise of containerization technologies like Docker, applications are now deployed in isolated and reproducible environments, facilitating their scalability and portability. However, this approach also introduces new challenges, particularly in log management. Logs are records of events produced by applications and are essential for troubleshooting, performance monitoring, and security. Managing logs in a containerized environment requires robust solutions to collect, centralize, and analyze this data effectively.

In this project, we have implemented a comprehensive infrastructure to automatically retrieve logs from an Nginx container, send them via Kafka, and process them with Spark before exposing them to an API. This approach not only centralizes the logs but also extracts valuable insights in real-time, thanks to Spark's distributed processing power.

This project demonstrates how to integrate and orchestrate various advanced technologies to solve concrete problems related to Big Data. By using Vagrant for environment management, Docker for containerization, Kafka for real-time data stream management, and Spark for data processing and analysis, this project illustrates a complete and modern solution for log management in a microservices context.

First, we will discuss the tools used, and then we will explain the implementation of the project.

Tools used

Filebeat

Filebeat is a lightweight open-source log shipper developed by Elastic. It is designed to read log files and send this data to systems like Elasticsearch or Kafka. In our project, Filebeat collects logs from Docker containers and routes them to Kafka for further processing. We chose Filebeat for its ease of configuration, its ability to handle container logs, and its simple integration with Kafka, ensuring efficient real-time log transmission.

Docker

Docker allows for the creation, deployment, and execution of applications in containers. These containers are lightweight, portable environments that include everything an application needs to run. In our project, we use Docker to run an Nginx container, enabling us to generate logs from a standard web server.

Vagrant

Vagrant is a tool used to create and manage virtual development environments. It uses configuration files to automate the deployment of virtual machines. In our project, Vagrant is used to create a virtual machine with the necessary configurations to run Spark, the API, and Docker.

Apache Spark

Apache Spark is a data processing engine designed for fast, large-scale analytical applications. In this project, Spark is used to consume the logs sent by Kafka, transform them, and analyze them in real-time. Spark was chosen for its compatibility with real-time data streams through its integration with Kafka.

Apache Kafka

Apache Kafka is a distributed streaming platform used to build real-time data pipelines. It allows for publishing, subscribing, storing, and processing data streams in a distributed manner. In our project, Kafka is used to transport logs collected by Filebeat from Docker containers to Spark.

Flask

Flask is a lightweight web microframework for Python, allowing for the development of web applications. In our project, Flask is used to create an API that exposes the processed logs and statistics derived from these logs. We chose Flask for its simplicity, flexibility, and easy integration with other Python libraries.

By using these tools, we were able to set up a complete solution for log management and analysis in a containerized environment. We will now explain how we implemented this project and made it operational.

Jules Davoust
Vadim Bernard
Robin Lucas

Explanation

Part 1 : Environment Configuration and Setup – Robin

Robin was responsible for setting up the development environment using Vagrant and Docker. His work was essential in creating a solid foundation upon which the rest of the project could be built.

Vagrant Configuration

Robin started by configuring Vagrant to create a virtual machine with the necessary resources. The following Vagrantfile was used to specify the details of the virtual machine, including the box used, port forwarding, and allocated resources:

See Appendix 1

Docker and Filebeat Configuration

Next, Robin configured Docker to run an Nginx container and set up Filebeat to collect logs from this container. The configuration of Filebeat was crucial to ensure that the logs from the Nginx container were correctly routed to Kafka.

The Filebeat configuration used was:

See Appendix 2

Bash Script to Launch Spark

Robin also created a bash script to facilitate the launch of the Spark application. This script starts Spark and submits the Spark job automatically, storing Spark logs in the file "spark_log.txt":

```bash
#!/bin/bash
source ~/.bashrc
cd /vagrant_data
echo "Starting Spark at $(date)" >> /vagrant_data/spark_log.txt
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.1
/vagrant_data/spark_app2.py >> /vagrant_data/spark_log.txt 2>&1
```

To launch the Spark application, the user simply needs to connect to the VM and execute the following command :

```
screen -dmS spark /vagrant_data/start_spark.sh
```

This command uses screen to run the script in a detached session, allowing the Spark application to run in the background.

This part played a crucial role by setting up the development environment, ensuring that all necessary dependencies and configurations were in place for the project to be properly developed. Additionally, the creation of the bash script simplified the launch and management of the application, making the entire system more accessible and easier to use.

Part 2 : Setting up Kafka and Spark – Jules

Jules was in charge of configuring and setting up Kafka and Spark, two key components for log processing and analysis.

Kafka Configuration

Jules installed and configured Kafka on the host machine, ensuring it was ready to receive logs sent by Filebeat. He had to ensure the correct transmission of logs between Filebeat and Kafka so that Spark could then process the logs.

The commands used to configure Kafka were:

```
# Start Zookeeper
.bin\windows\zookeeper-server-start.bat .config\zookeeper.properties

# Start Kafka
.bin\windows\kafka-server-start.bat .config\server.properties

# Create a topic for the logs
.bin\windows\kafka-topics.bat --create --topic logs --bootstrap-server
localhost:9092
```

These commands start the Zookeeper and Kafka services, as well as create a topic for the logs, thus preparing Kafka to receive the data.

Spark Configuration

Jules then configured Spark to consume Kafka messages and process them in real-time.

The following code shows how Spark was configured to read logs from Kafka and process them :

See Appendix 3

This section allowed the configuration of the key components of Kafka and Spark, ensuring the effective transmission and processing of logs in real-time. It ensured that the logs could be analyzed in real-time, providing significant added value to the project.

Part 3 : Developing the Flask API – Vadim

Vadim was responsible for developing the API using Flask. This API allows for retrieving processed logs and providing statistics on these logs.

Flask API Development

Vadim created several endpoints for the API, including an endpoint to retrieve logs and another to get statistics on the logs. The API code is as follows :

See Appendix 4

Vadim developed an API that exposes the data processed by Spark, allowing users to retrieve logs and obtain detailed statistics. His work facilitated access to the data and made analyses more accessible via API endpoints. This API serves as an essential interface for interacting with the data and extracting relevant information.

Conclusion

This project allowed us to set up an infrastructure for managing and analyzing logs in a containerized environment. We leveraged several technologies to achieve this goal. The development environment configuration, managed by Robin, established a solid foundation using Vagrant to create a well-equipped virtual machine, Docker to run Nginx containers, and Filebeat to collect logs. The configuration of these tools enabled centralized log management and prepared the data for further processing.

Jules took charge of configuring Kafka and Spark, two essential components for transporting and analyzing logs in real-time. Kafka was used to ensure reliable transmission of logs from Filebeat, while Spark was used to process and analyze this data in real-time.

Vadim developed a Flask API that exposes the data processed by Spark, allowing users to retrieve logs and obtain detailed statistics. This API offers filtering and grouping functionalities for logs, facilitating data exploration and analysis. Thanks to this API, we were able to make the data accessible and exploitable, providing a user-friendly interface to interact with logs and extract relevant information.

This project gave us a deep understanding of integrating various technologies to solve data management problems. We learned to configure and orchestrate tools such as Vagrant, Docker, Filebeat, Kafka, Spark, and Flask within the context of log retrieval from a container. By combining these technologies, we demonstrated how it is possible to collect, process, and analyze large volumes of data in real-time, thus meeting the growing needs of modern infrastructures based on microservices and containers.

Annexes

Annexe 1

Ensures that the necessary ports (8080 for Spark UI, 7077 for Spark master, 5000 for Flask) are accessible from the host.

```
Vagrant.configure("2") do |config|
  config.vm.box = "paulovn/spark-base64"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "6012"
    vb.cpus = 4
  end

  config.vm.network "forwarded_port", guest: 8080, host: 8080  # Port Spark UI
  config.vm.network "forwarded_port", guest: 7077, host: 7077  # Port Spark master
  config.vm.network "forwarded_port", guest: 5000, host: 5000  # Port Flask

  config.vm.synced_folder "./data", "/vagrant_data"

  config.vm.provision "shell", inline: <<-SHELL
    sudo su
    sudo apt-get update
    sudo apt-get install -y openjdk-8-jdk wget python3-pip screen

    wget https://downloads.apache.org/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz
    tar -xzf spark-3.5.1-bin-hadoop3.tgz
    sudo mv spark-3.5.1-bin-hadoop3 /usr/local/spark
    echo 'export SPARK_HOME=/usr/local/spark' >> ~/.bashrc
    echo 'export PATH=$PATH:$SPARK_HOME/bin' >> ~/.bashrc
    source ~/.bashrc

    pip3 install flask
    pip install flask

    curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-7.13.1-
amd64.deb
    sudo dpkg -i filebeat-7.13.1-amd64.deb

    sudo wget -O get-docker.sh https://get.docker.com/
    sudo sh get-docker.sh
  SHELL
end
```

Automates the installation of Java, Spark, Flask, Filebeat, and Docker to ensure that the environment is ready for use.

The Vagrantfile automates the VM configuration by installing the necessary dependencies, setting up environment variables, and installing Docker and Filebeat. This setup ensures that the environment is ready to use as soon as the VM is launched.

Annexe 2

```
filebeat.inputs:
- type: container
  enabled: true
  paths:
    - /var/lib/docker/containers/*/*.log
  processors:
    - add_docker_metadata: ~
    - decode_json_fields:
        fields: ["message"]
        target: ""
        overwrite_keys: true
  fields:
    log_topic: 'logs'
  fields_under_root: true
```

« container » specifies that Filebeat should collect logs from Docker containers.

« add_docker_metadata » enriches the logs with information about the containers.

```
output.kafka:
  hosts: ["<Host's IP>:9092"] # Replace by your Host's IP
  topic: '%{[log_topic]}'
  codec.json:
    pretty: false
```

Sends the logs to Kafka on the "logs" topic, ensuring they can be processed in real-time.

```
filebeat.config.modules:
  path: ${path.config}/modules.d/*.yml
  reload.enabled: false

logging.level: info
logging.to_files: true
logging.files:
  path: /var/log/filebeat
  name: filebeat
  keepfiles: 7
  permissions: 0644
```

The Filebeat configuration allows for the collection of logs from Docker containers, enriching them with metadata, and sending them to Kafka for further processing.

Jules Davoust
Vadim Bernard
Robin Lucas

Annexe 3

Uses the necessary packages for integration with Kafka.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType
from pyspark.sql.functions import from_json, col

spark = (SparkSession.builder
        .appName("KafkaConnectivityTest")
        .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.1")
        .getOrCreate())

df = (spark
    .readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "<Host's IP>:9092")  # Replace by your Host's IP
    .option("subscribe", "logs")
    .load())
```

Uses readStream to consume messages in real-time.

```
schema = StructType([
    StructField("@timestamp", StringType(), True),
    StructField("log_level", StringType(), True),
    StructField("message", StringType(), True),
    StructField("container", StructType([
        StructField("name", StringType(), True),
        StructField("image", StructType([
            StructField("name", StringType(), True)
        ])),
        StructField("id", StringType(), True)
    ])),
    StructField("host", StructType([
        StructField("name", StringType(), True)
    ])),
    StructField("stream", StringType(), True)
])

logs = df.select(from_json(col("value").cast("string"),
schema).alias("data")).select("data.*")
```

Defines a structured schema for JSON messages, allowing conversion into columns.

```
query_memory = (logs
        .writeStream
        .outputMode("append")
        .format("memory")
        .queryName("logs_table")
        .start())

query_console = (logs
        .writeStream
        .outputMode("append")
        .format("console")
        .start())
```

Uses writeStream to store logs in a temporary table, facilitating subsequent SQL queries.

This code configures Spark to read logs from Kafka, convert them into columns according to a defined schema, and store them in a temporary table for later use. The use of readStream and writeStream allows for real-time data processing.

Annexe 4

```python
from flask import Flask, jsonify, request, Response
import threading

app = Flask(__name__)

def keep_spark_streaming():
    query_memory.awaitTermination()
    query_console.awaitTermination()

thread = threading.Thread(target=keep_spark_streaming)
thread.daemon = True
thread.start()

@app.route('/logs', methods=['GET'])
def get_logs():
    try:
        log_level = request.args.get('log_level', None)
        container_name = request.args.get('container_name', None)
        start_time = request.args.get('start_time', None)
        end_time = request.args.get('end_time', None)
        limit = int(request.args.get('limit', 100))
        offset = int(request.args.get('offset', 0))

        query = "SELECT * FROM logs_table"
        filters = []

        if log_level:
            filters.append(f"log_level = '{log_level}'")
        if container_name:
            filters.append(f"container.name = '{container_name}'")
        if start_time:
            filters.append(f"to_timestamp(`@timestamp`) >= to_timestamp('{start_time}')")
        if end_time:
            filters.append(f"to_timestamp(`@timestamp`) <= to_timestamp('{end_time}')")

        if filters:
            query += " WHERE " + " AND ".join(filters)

        query += f" LIMIT {limit} OFFSET {offset}"

        logs_df = spark.sql(query)
        logs = logs_df.collect()

        logs_list = []
        for row in logs:
            logs_list.append({
                "timestamp": row["@timestamp"],
                "log_level": row.log_level,
                "message": row.message,
                "container": {
                    "name": row.container.name,
                    "id": row.container.id,
                    "image": row.container.image.name
                },
                "host": row.host.name,
                "stream": row.stream
            })

        return jsonify(logs_list)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
```

Uses a separate thread to keep the Spark stream active without blocking Flask.

Allows for retrieving logs with filtering options by log level, container name, time range, limit, and offset.

```
@app.route('/log-stats', methods=['GET'])
def log_stats():
    try:
        stats_df = spark.sql("""
            SELECT
                log_level,
                COUNT(*) as count,
                container.name as container_name
            FROM logs_table
            GROUP BY log_level, container.name
        """)
        stats = stats_df.collect()

        stats_list = []
        for row in stats:
            stats_list.append({
                "log_level": row.log_level,
                "count": row['count'],
                "container_name": row.container_name
            })

        return jsonify(stats_list)
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Provides statistics on logs, grouped by log level and container name.

This code configures Flask to create a RESTful API that allows for retrieving logs and obtaining statistics. The /logs and /log-stats endpoints offer filtering and grouping functionalities for logs, enabling thorough exploration and analysis of the collected data.