Volume 9, Issue 11, November– 2024 International Journal of Innovative Science and Research Technology
ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/IJISRT24NOV1065

Config Client Refresh Approach on Containerized Spring Boot Microservices: System Scalability and Reliability

Shreeson Shrestha Er.
Infinite Computer Solutions

Abstract:- Spring boot microservices require the application runtime to pause until the services restart for the configuration refresh. Services linked to a centralized config server, or preferably, a config client, need this. Config client refresh played a critical role in maintaining the consistency and responsiveness of distributed microservices by dynamically updating client-side configurations without application downtime. This approach explored various strategies and optimizations for config client refresh to enhance system reliability. This Study investigated different available approaches of config client refresh mechanisms for impact on system behavior. This study experiment utilized the available spring packages, bash-scripts, and docker for the containerization of applications and collected performance metrics for analysis. Further, this approach identified key factors affecting refresh efficiency and proposed optimized techniques to mitigate potential challenges. Findings contribute a deeper understanding of configuration client refresh and offer pragmatic insights into designing robust and efficient scalable systems.

Keywords:- *Configuration Management, Config Refresh, Containerization, Distributed System, Microservice Architecture, Spring Cloud Config*

I. INTRODUCTION

In the Current Spring boot Microservices, consistently maintaining updated configurations across config-clients ensured system reliability and adaptive dynamic system environments. Config client refresh is the process by which applications retrieve and apply updated configurations from a centralized config server and is a mechanism for achieving this objective. This paper considered various config-client refresh mechanisms to optimize the performance and reliability of microservices and suggested an approach for config-client refresh dynamically.

➢ *Background and Context*

Latter-day software architectures rely on centralized configurations to control the application process and functionality, including environment-specific settings, features, etc. In addition, microservices have moved from regular deployments to a containerized approach. These configs change concerning growing requirements, scalability demands, operational changes, etc. So, the challenge for these client applications was to synchronize configuration efficiently from central sources like config servers for consistent behavior across distributed microservices.

➢ *Problem Statement*

Challenges to this consistency among the distributed microservices relate to the system's resilience, reliability, scalability, and timeliness. In large-scale systems with numerous clients, configuration inconsistency can impact system functionality. Additionally, distributed microservices can have unpredictable behavior dependent on these configurations, affecting the system's reliability. Springboot microservices, config-client requires restarting just making a downtime for normal functioning. This study aims to solve these challenges by instigating an optimized approach to the config client refresh process.

➢ *Objectives of Study*

The Objectives of this study are:

- Analyze different mechanisms for config client refresh
- Factors affecting the scalability and reliability of the system
- Propose a strategy for config client refresh in the microservice system

➢ *Significance of Config Refresh*

The reliability of the system design relies upon the config client refresh process. Improved config client refresh can enhance the system reliability and streamline deployment practices.

In further sections, this paper probes into the literature around config client refresh mechanism outlines the experiment for this study, presents an experiment of use cases, and discusses implications for practice.

This section prepares for an approach emphasizing the significance of config client refreshes, framing the research problem, and setting out the study objectives.

II. LITERATURE REVIEW

In this section, we review the existing strategies related to configuration management and synchronization of configuration dynamically. This dynamic refresh of config clients was a core aspect of maintaining system integrity and reliability in spring boot microservices.

Several approaches have already been explored where spring boot provides config refresh from the centralized configuration for microservices in practice. Spring community approaches to config refresh using a centralized config server and config clients as standalone microservices. This config server resembles the configuration for distributed microservices either on the file system or remote repositories, especially git repositories, and, the microservices, a config client should sync with the updated configurations on the config server periodically. For Spring Boot microservices with these latter system designs having a centralized config server, all the config clients are responsible for consistently updating configurations dynamically. Additionally, it impacts deployment by making the config-client restart after the updated configurations to the config-server.

Industries use various approaches like webhooks, packages, tools, and technologies to sync these configurations between the config server and client. Config-clients synchronize these configurations either polling with the centralized config server or needing a restart of the service itself which caused a downtime for the application functionality. Additionally, Spring Cloud provides packages: Spring Cloud Config, Spring Actuator, Spring Monitor, and Spring Cloud Bus for efficient configuration distribution and simultaneity between the configurations. These specific packages maintain a role for each step involved in the synchronization process. Here, spring Cloud config provides an approach for centralizing the config server, spring actuator enables restful service **/actuator/refresh** API for config client to sync configuration for specific single microservice, Spring Cloud bus integrates with the Message brokers like RabbitMQ or Kafka for configuration synchronization. This Cloud Bus, allows services to broadcast events like management instructions and state change of configuration to entire config client services and also a powerful approach for synchronization of Configuration without restarting of service. Again, webhooks are utilized in use cases such as cloud repository applications that maintain configurations in git repositories. These applications set up webhooks on the config server, which leads to an event of configuration updates and refreshes all of the services linked to Spring Cloud Bus. In this case, the /monitor API that Spring Monitor exposes was set up to be triggered by SCM (Source Control Management) webhooks, activating the bus refresh endpoint that Spring Cloud Bus Integrations provides.

Furthermore, while existing works of literature and organizational practices provide various valuable approaches to enable client microservices to refresh configurations with likely zero downtime, ensuring operating with consistent configurations and reducing discrepancies, there are still gaps and limited approaches were researched and fallbacks for use cases where numerous containerized services interacting with the shared configurations configured not to use message brokers which too restricts the use of Spring Cloud Bus and Monitor approach.

This literature review provides a base for understanding current scenarios over an approach to config-client refresh and highlights the gap for further areas and use cases. The following Sections on the methodology used in the study address these gaps.

III. METHODOLOGY

Tools and techniques used for optimizing the config-client refresh process in distributed spring-boot microservices applications are explained in this section. The principal focus was to suggest an approach to refresh configuration in a containerized spring-boot microservices environment using docker and bash scripts to automate refresh operations.

This paper followed an experimental approach, polishing a practical implementation of a configuration client refresh process in a dockerized Spring Boot application. The configuration refresh process was automated using a Bash script inside a Dockerfile, which triggered configuration updates and monitored their effect on system performance and responsiveness. The configuration update process uses the REST API provided by the **spring-boot-starter-actuator** package (i.e. **/actuator/refresh).**

➢ *Tools and Techniques used*

To carry out this experimental approach, the following tools were utilized.

- **Docker**: Docker was used to containerize the Spring Boot application. This containerization of the application ensured that all required dependencies for an application were packaged along with the docker image creating an isolated environment with easier deployment and test configuration updates across different environments.

- **Spring Boot**: The application was built using Spring Boot, a popular Java framework for building microservices and standalone applications. Spring Boot provided a range of non-functional features for large-scale applications like externalized and centralized config-server inclusive of libraries for dynamically updating their configurations during runtime without requiring restarts.

- **Spring Cloud Config**: The application used Spring Cloud config to support externalized configuration for microservices. These provided support for a centralized config server for an application and client-side support for microservices for dynamic configurations.

- **Spring Boot Actuator:** The REST API (/actuator/refresh) was made available to us by the application's use of the spring-boot-starter-actuator package. A health check and refresh are a part of the extra services and features offered in this sub-package.

- **Bash Script**: A Bash script was created to automate the configuration refresh procedure. For the Config client Spring Boot application, the script initiated a procedure to update its configuration from the centralized configuration server. This made it possible for us to synchronize configuration updates effectively through automation.

➢ *Config-Client Configuration Refresh Process*

To conduct this experiment, the following procedures were followed.

- **Initial Setup:** A microservices architecture-based spring boot application was created. This application used spring cloud config for the centralized config server and numerous config-client setups and actuator packages for providing API for config-client refresh for configurations.

- **Containerize Application:** This application was containerized using docker. The Dockerfile contains the bash scripts for refreshing configurations for the config clients. Each config client had a separate Dockerfile which made a separate container replicating a distributed system with numerous config client applications.

- **Automated configuration refresh:** The bash script for refreshing configuration for config client from centralized config server which was placed inside a docker container itself. This bash script periodically triggers a REST API an actuator provides (i.e. /actuator/refresh).

- **Monitoring of Performance:** This dockerized application was monitored for performance metrics such as refresh time and resource utilization. These collected metrics helped system reliability measures for handling refresh operations without downtime.

➢ *Data Collection and Analysis*

Data related to performance metrics were collected throughout the process. The analysis focused on the following.

- **Refresh time:** The response time for the refresh trigger was captured
- **Resource Utilization:** CPU and memory utilization was monitored in the process of frequent refresh operations. These metrics were collected with the hosting environment with this containerized Application. We used the Openshift cluster.
- **Reliability of system:** Errors, Exceptions, and downtime as a result of periodic refresh configurations were observed to ensure system reliability

IV. RESULTS AND DISCUSSION

From the above experimental approach on config refresh on containerized config clients of spring boot-microservices, we have the following findings. Also, this focused on analyzing the performance metrics after the implementation of the approach.

A. Results and Findings

This approach for dynamic configuration refresh on containerized spring boot microservices, also, focused on the performance metrics for scalability and reliability of the system and the impact of periodic refresh. The following findings are presented.

➢ *Resource Utilization:*

The container's resource utilization was observed closely during the configuration refresh process. Resource consumption was seen as in patterns as it was a periodic trigger for refresh. The experiments revealed that:

- **CPU and Memory Utilization**: Utilization in this aspect appeared to be on pattern. This periodic refresh affected only acceptable limits and did not significantly notice a choke over the container's limit.
- **Network Overhead**: As the bash script was used to trigger refresh API using the curl POST method, overhead on a network was expected due to periodic refresh. Only a minimal increase in network traffic was observed due to this operation.

This periodic refresh approach was observed within the acceptable limits, this experiment suggested refresh operation using a script was CPU and Memory-efficient but further optimizations on scripts and resources could lead to more efficient results for resource utilization.

➢ *Refresh Time:*

This was one of the major key performance metrics for the refresh operations. In general, refresh time was good but some latency was observed for some services.

- **Normal refresh rate:** Most of the responses on services for curl POST for refresh were under 1 minute and 30 seconds. This ensured the spring boot application dynamic update of configurations to the config-clients very efficiently.
- **Abnormal refresh rate:** Some abnormal response times on the services were recorded. The latency time captured was even more than 20 minutes for some services. However, this latency was seen on random services and not on a specific service every time. This might be due to some network glitches but abnormality was not observed for every trigger.

This occasional latency observed for some random services emphasized the importance of stable network connections while especially relying on API calls. However, some retry mechanisms and limits to response time could mitigate refresh time issues due to network delays or glitches.

➢ *Reliability of System*

The application demonstrated a high level of system reliability despite latency on refresh time and periodic refresh operation.

- **Error and Exception:** No exceptions and errors regarding the configuration refresh were detected on the container logs of containerized Spring Boot applications. This indicates periodic configuration refresh operated smoothly.

- **Application Downtime:** No application downtime was observed during the refresh configuration process. Further, this approach removed the config clients restart (i.e. downtime) for refreshing configurations dynamically which had impacted the system reliability**.**

Handling configuration refresh in containerized Spring Boot application without errors, exceptions, and downtime proves the effectiveness of the current approach for the reliability of the system.

V. LIMITATIONS AND FUTURE CONSIDERATIONS

This approach of using a bash-script on containerized spring-boot microservices provided an efficient way for dynamic refresh of configuration without restarting config-clients in microservices. However, future research and considerations can be made as follows:

➢ *Network Stability*

This approach of configuration refresh was completely dependent on the API calls provided by the spring boot actuator. API request was handled using the CURL bash script. Therefore, implementation of retry mechanisms and response time out could improve the network resilience and consider the stable network for efficiency for refresh operations.

➢ *Frequency of Refresh*

The approach suggested a periodic refresh with a certain interval for dynamically refreshing the configurations. This periodic refresh approach could be altered or reduced with the necessity of refresh and if configuration change was infrequent.

VI. CONCLUSION

This paper examined the different approaches available for dynamic configuration refresh for spring boot microservices and focused on the analysis of performance metrics that could impact system reliability and scalability. Furthermore, an approach to a config refresh mainly with containerized environments for spring boot microservices was also suggested by this paper. Also, performance metrics like CPU and Memory utilization, and impact to the system using the periodic configuration refresh and refresh time were analyzed for its efficacy on the system with the suggested approach.

This experiment used docker for containerization of spring boot applications and triggered an API using CURL script for refreshing the configuration on config clients of microservices architecture using a centralized config server. API was configured and provided by the spring actuator package (i.e. /actuator/refresh) which syncs the configuration from the central config server.

Again, the CPU and memory utilization with the periodic trigger of refresh had a minimal effect within the considerable limits. However, the response time was observed abnormal for some of the services. This abnormality was seen most specifically due to the network instability, This instability could be optimized using failure resilience approaches such as response timeout and retry mechanism for trigger operations. Also, the interval period can be minimized if the frequency of config refresh on the centralized config server is less providing reliable configuration synchronization to the config clients.

REFERENCES

[1]. Pivotal Software, Inc. (n.d.). *Spring Boot 3.3.x reference guide*. Spring.io. Retrieved October 12, 2024, from https://docs.spring.io/spring-boot/docs/3.3.x/reference
[2]. Spring Cloud. (n.d.). *Spring Cloud Config: Centralized External Configuration Management*. Spring.io. Retrieved October 15, 2024, from https://spring.io/projects/spring-cloud-config
[3]. Docker, Inc. (n.d.). *Docker Overview*. Docker Documentation. Retrieved October 15, 2024, from https://docs.docker.com/get-started/overview/
[4]. Spring.io. (n.d.). *Spring Cloud Bus 4.1.x reference guide*. Retrieved October 18, 2024, from https://docs.spring.io/spring-cloud-bus/docs/4.1.x/reference/html/
[5]. Spring.io. (n.d.). *Building a RESTful web service with Spring Boot Actuator*. Retrieved October 18, 2024, from https://spring.io/guides/gs/actuator-service
[6]. Spring.io. (n.d.). *Push notifications and Spring Cloud Bus in Spring Cloud Config*. Retrieved October 19, 2024, from https://cloud.spring.io/spring-cloud-config/multi/multi__push_notifications_and_spring_cloud_bus.html