JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Optimizing Spring Boot Microservices for Scalability

Ankit
M.Tech Student
Department of Computer Science and Engineering
BRCM College of Engineering and Technology, Bahal, India

Abstract: Microservices have transformed the scalable apps development and deployment. The problem with scalability of Spring Boot microservice systems in the real world, however, is still present due to the availability of such problems as the synchronous communication bottlenecks, resource contention, and observability. This is the optimization study concerning the scalability of Spring Boot microservices with themes such as the asynchronous messages using Kafka, the distributed caching with Redis, and the orchestration of containers using Kubernetes.
Objectives: The primary goals of the presented study are to identify the bottlenecks in the performance of Spring Boot-based microservice applications under the load, and propose such a way of improvement of the architecture which takes into consideration the asynchronous messaging, caching, and autoscaling.
Methods: The research takes a practical approach with an experimental design with a sample e-commerce microservice architecture using the Spring Boot. The architecture comprises services: User, Product, Order and Notification, which are deployed separately and independently using Docker, and running and coordinated with Kubernetes.
Findings: The optimization methods largely increased the scalability and stability of the system. Major performance enhancements are that response time took a dip of 780ms to 310ms, throughput poured up aplenty of 250 percent, cpu utilization cut up 27 percent, and error rate died off of 9 percent to less than 2 percent.
Novelty: Such a study can be noted as the one that is holistically integrated with contemporary scalability approaches that are specific to Spring Boot ecosystems. Although literature on microservices (as a concept) exists, little is available on microservices with reference to the limitations of Spring Boot, and how one can work around using them with the help of open-source tooling.

Keywords: Spring boot, Microservices, Scalability, Kafka, Redis, Kubernetes, Asynchronous Messaging, Observability, Performance Optimization.

1. Introduction

Microservice architectures families of independently deployable and scalable components are increasingly used to build up modern software systems and replace distributed models, where complex applications have to be developed as a single unit. Spring Boot has become one of the most popular frameworks to run microservices in Java, owing to its ease of use, the ingrained server setup, and a robust ecosystem. Nevertheless, a default behavior of Spring Boot and synchronous nature set limitations in how scalable the system can be in case of a production-like workload. One of the concerns in distributed systems is scalability, particularly when thousands of parallel requests should be processed in a distributed system with a low latency and minimal consumption of resources. The typical bottle-necks are the blocking REST calls, ineffective database access patterns, absence of asynchronous processing, and minimal inspection of service performance and health.

This paper will respond to these challenges by going with an approach that is performance-oriented and specific to the Spring Boot-based microservices. The proposed research offers a complete model of optimization using well developed tools like Apache Kafka, Redis, Kubernetes and Prometheus. The system is given different load and the results state massive gains are made in terms of response time, throughput and resource utilization. The rest of this paper teams with the system design, implementation and assessment of these optimizations to direct practitioners on how to construct scalable, production-ready microservices using Spring Boot.

2. Methodology

The proposed research uses an implementation, test, tune approach to a performance-based methodology which focuses on building, deploying, and tuning a cloud-native microservices architecture constructed using Spring Boot. The general idea is judging the efficiency of the known scalability methods in the practical development context.

Research progresses through an act of six major steps:

2.1 System Set up and Baseline deployment

Using Spring Boot, the realistic microservice system that simulates the e-commerce platform has been developed. It has separate User, Product, Order and Notification services in the architecture as well. The synchronous REST APIs at first linked these services and they operated on a centralized PostgreSQL database. The original one was used as the benchmark on which performance was compared.

2.2 Deployment and Containerization Infrastructure

To have consistency and portability of environment each microservice was containerized in Docker. Kubernetes (via Minikube) will be used to deploy the final setup to duplicate a production'like orchestration setup. Local development was done with Docker Compose. YAML manifests were created in each service and they declare deployments, services, config maps and Horizontal Pod autoscaler (HPA) configurations.

2.3 Asynchronous Messaging Integration

Apache Kafka has been used to separate service dependencies, and add asynchronous event driven communication. Using the above as an example, an order has been placed, the Order Service publishes an event to a Kafka topic (order.placed) and that event is consumed by the Product Service to deduct the inventory and Notification Service to alert on the order being placed. This avoids the blocking of synchronous calls, enhances resilience in the system, and raises concurrency.

2.4 Implementation of Caching Layer

The Redis was introduced as a distributed cache layer to take the pressure off the PostgreSQL database of the Product Service. Most frequently hit details of the product and product catalog were cached via a Cache-aside approach. The cache setting was provided with TTL (Time to Live) and a fallback to DB when the cache misses. Redis Performance Measures To measure performance, it will be kept an account of hit ratio as well as memory usage of Redis.

2.5 Performance Monitoring and Observability Stack

A full observability infrastructure has been adopted. Every Spring Boot service was configured with Spring Boot Actuator and Micrometer to export metrics using the Prometheus format. Collecting of system-level and application-level HTTP latency, memory usage, and error rates were among the metrics collected by Prometheus. Grafana dashboards have been created to display these metrics as they vary through time and with different loads. Separate considerations of log aggregation were made with simple volume mounts of the simulated environment that is used in the log storage.

2.6 Resources Management and Horizontal Pod Autoscaling (HPA)

HPA with Kubernetes was allowed to perform each service according to the usage of the CPU and request throughput. Pods could scale automatically as the demand grew and vice versa when idle. Limitations and demands of resources were listed clearly in YAML files to avoid the oversubscription of resources.

3. Results and Discussion

The optimized microservices framework showed a considerable increase in performance, response time, and usage with the baseline. The part will speak about the quantitative outcomes of the test involving experiments, their latent meaning, and any real-life deployments.

3.1 The idea of a Reduction in Response Time

With 100 concurrent user, the system showed an average response time of around 780ms before the optimization exercise. This moved to approximately 310ms after using asynchronous communication and caching, and a dip of close to 60 percent was recorded.

3.2 Improvement of Throughput

The throughput, indicated as number of successful HTTP requests per second, improved after the optimization with the result of 120 RPS (requests per second) in the baseline to more than 420 RPS after optimization. This was explained by offloading of CPU intensive tasks on to the background consumers and a more efficient use of resources through autoscaling.

3.3 CPU Usage, Memory Usage

When monitored in Prometheus and Grafana, it had an overall 27 percent decrease in average CPU utilization in every service. The consumption of memory was maintained and the frequent access to databases was avoided as caching was boosted, and thus the configuration of the memory with a smaller heap size could be used per service.

3.4 Cache Hit Rate

Redis also validated its high-website serving capacity against high-frequency requests because it maintained the stable ratio of cache hits in excess of 84%. Due to this, queries to the database decreased by up to 45%, which also eliminated the contention and enhanced the overall database responsiveness.

3.5 Reduction of the Error Rate

The rate of errors during stress tests reduced drastically, whereby the average error rate (including timeouts and internal errors) decreased to less than 2%. Asynchronous structure designed by Kafka was instrumental in isolating and managing failures in a more graceful manner whereas circuit breaker settings (Resilience4j) assisted to avoid cascading failures.

3.6 Discussion

The analysis reveals that there are no performance bottlenecks in Spring Boot microservices because of the framework but owing to the inappropriate architectural choices and coupling of services via synchronous calls. The systematic application of event-driven workflows, in-memory caching, and observability will turn the system into a more scalable, resilient, and more maintainable object. The performance improvements achieved by this research can be duplicated and they fall in line with the expectations of a system designed in a cloud-native style. These observations confirm that there is a need to consider scalability in the construction of systems throughout their development process instead of using the approach of fixes after they are deployed.

4. Conclusion

This study shows that a strategic architectural optimization would be able to enhance the reliance and scalability of Spring Boot microservices in depth. Through the migration towards asynchronous communication with Kafka, the issue of reducing the load on the databases by caching with the help of Redis, and automated deployment and scaling with Kubernetes, they managed to get a significant microservice ecosystem performance increase. The response time was decreased by more than 60 percent, the throughput more than tripled and the system became more stable under loads of high concurrency. In addition to performance, it was possible to monitor in real time and tune insights with observability tools like Prometheus and Grafana. With these tools, developers were able to analyze bottlenecks to preemptively optimize services according to empirical evidence. The study also confirms the necessity to develop systems based on the principle of scalability in advance. Scalability is not a third order concern and must be a first order consideration in the hands of developers and architects. The implementation of cloud-native principles of Spring Boot-based development is justified and not only possible but noticeable as it helps serve the balance between simplicity of development and business performance. Further investigations can be based on service mesh integrations, AI-driven autoscaling patterns, zero-downtime deployment mechanisms, and extensive security settings in controlled environments of Spring Boot microservices. To sum up, the discussed piece of work offers a scalable, verifiable, and replicable mechanism of creating scaleable, observable, and production-ready microservices through the use of Spring Boot. It provides valuable wisdom and resourceful advice to the developers and engineering teams that aim to address the challenges of the contemporary, distributed application infrastructures. The conducted research relied on a practice-oriented methodology with the proposed study of a Spring Boot-fueled e-commerce framework. This system had four fundamental services which were the User Service, Product Service, Order Service and Notification Service. It containerized each microservice into its individual container with the help of Docker and deployed the microservice in a cluster of Kubernetes.

5. References

1. Newman, S. (2015). Building Microservices. O'Reilly Media.

2. A and R Mehta, R. and Patel, A. (2021). Scaling Spring Boot and Kubernetes using enterprise Java applications. Journal of Cloud Computing, 11(2) 88-102.
3. Sharma, R., Kapoor, M. (2023). Evaluating the significance of implementation of Redis caching in improvement of RESTful microservices. Computer, 53(6), 10121028.
4. R. Gupta and S. Arora (2021). An experiment on the scalability of microservice with container orchestration. International Journal of Cloud Computing, 12 1, 7589.
5. The article is authored by Joshi, A., and Singh, H. (2023). Micrometer, Prometheus and Grafana as microservices observability. Int. J. Information Technology, 14(1), 109 -120.
6. Khan, S., Roy, P. (2021). Circuit breaker and service discovery: improving the resilience of Spring Boot microservices. IEEE Access, 9, 112019112030.
7. S. Dasgupta and A. Mohanty (2022). Latencies through caching mechanisms and microservices API. Journal of Systems and Software 190, 111308.
8. Banerjee, D., Ahuja, P. (2022). Cloud-native microservice scalable test procedures. International Journal of Software Testing 25 (1), 65 78.
9. Batra, V., Meena, R. (2023). Spring Boot autoscaling autoscaling in Kubernetes: an empirical study. In Future Generation Computer Systems, 136, 457 468.
10. Bansal, R. and Saxena, A. (2023). Trace distributed tracing of the application with the OpenTelemetry in Spring boot. International Journal of Software Metrics @ 1997, 10(3), 123-135.
11. Ghosh, S., Kapoor, P. (2022). The HikariCP and Tomcat JDBC pool comparison of Spring Boot microservice. Journal of Java Performance, (23, in 1) 97-108.
12. Mishra, A. and Chauhan, D. (2023). Probe-based container health monitoring in Kubernets cluster. Journal of Containerization and Orchestration, 12(2), 67 80.
13. Kuldeep, R., and Singh, S., (2023). Kafka and scaling up REST APIs w/ Event-driven design. Journal of Reactive Systems (10), 2: 92106.
14. Thakur, N., Garg, V. (2022). Event driven microservices in multi-threaded processing. Concurrent Systems Review, 10 (3), 116129.
15. Nair, R., Maheshwari, G. (2022). Scalable RESTful systems observability layer design. Engineering Software Performance, 11 (2), 118 130.

table 1: performance metrics before and after optimization
