Best Practices for Managing Java-Based Production Systems

Omar Al-Farsi
Department of Computer Science, University of Qatar
Fatima El-Sayed
Department of Computer Science, University of Cairo

check for updates

Article history:
Received:
Jan/12/2024
Accepted:
Mar/08/2024

.

Abstract

Java has long been a cornerstone technology in enterprise computing, known for its robustness, portability, and scalability. From web applications to large-scale enterprise systems, Java provides a versatile platform that can adapt to various business needs. Among the various frameworks available for building Java applications, Spring Boot has gained significant popularity due to its ability to simplify development, streamline configurations, and accelerate time-to-market. However, managing Spring Boot applications in a production environment presents unique challenges that require a well-structured approach to ensure they remain reliable, secure, and efficient. Spring Boot Actuator is a powerful tool that provides production-ready features such as monitoring, metrics, health checks, and more. It integrates seamlessly with Spring Boot applications, offering endpoints that allow administrators to monitor and manage their applications effectively. This paper aims to explore best practices for managing Java-based production systems, with a particular emphasis on leveraging Spring Boot Actuator alongside other tools and strategies. The structure of this paper includes several sections: continuous monitoring using Spring Boot Actuator, scalability considerations, security best practices, and performance optimization techniques. Each section provides detailed insights and recommendations for managing Spring Boot applications in a production environment, ensuring they can meet the demands of modern enterprise systems while minimizing operational risks.

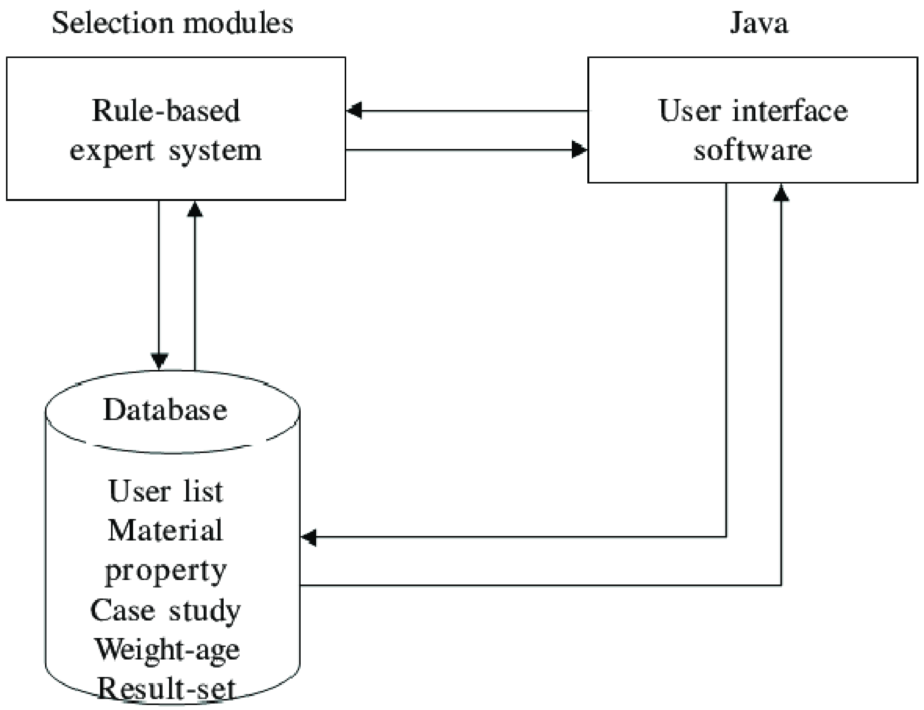
Keywords: Java application monitoring, performance surveillance, Spring Boot Actuator, JVM optimization, holistic monitoring, application performance management, real-time analytics, enterprise Java, network monitoring, memory management, thread activity, distributed tracing, anomaly detection, microservices architecture, continuous integration and deployment (CI/CD), automated testing, containerization, Docker, Kubernetes, horizontal scaling, vertical scaling, load balancing, service discovery, circuit breaker pattern, distributed caching.

Introduction

Java has long been a cornerstone technology in enterprise computing, known for its robustness, portability, and scalability. Since its inception, Java has evolved into a versatile and powerful programming language that is used to build a wide range of applications, from small-scale desktop programs to large-scale, mission-critical enterprise systems. Its platform independence, backed by the Java Virtual Machine (JVM), allows developers to write code that can run on any system that supports Java, making it a favored choice for businesses seeking long-term solutions that are adaptable to changing technological landscapes.

Among the various frameworks available for building Java applications, Spring Boot has gained significant popularity due to its ability to simplify development, streamline configurations, and accelerate time-to-market. Spring Boot is built on the comprehensive Spring Framework, which provides a range of functionalities such as dependency injection, transaction management, and aspect-oriented programming. Spring Boot, by embracing convention over configuration, reduces the complexity typically associated with configuring Spring applications, allowing developers to focus on building business logic rather than dealing with boilerplate code and setup.

However, managing Spring Boot applications in a production environment presents unique challenges that require a well-structured approach to ensure they remain reliable, secure, and efficient. As organizations deploy more complex applications that serve a growing number of users, the demands on these systems increase, necessitating robust management strategies to ensure seamless operation. This includes not only maintaining the performance and availability of the applications but also ensuring they are secure from potential threats and capable of scaling to meet increasing loads.



Spring Boot Actuator is a powerful tool that provides production-ready features such as monitoring, metrics, health checks, and more. It integrates seamlessly with Spring Boot applications, offering endpoints that allow administrators to monitor and manage their applications effectively. This paper aims to explore best practices for managing Java-based production systems, with a particular emphasis on leveraging Spring Boot Actuator alongside other tools and strategies. The comprehensive approach includes examining continuous monitoring, scalability considerations, security best practices, and performance optimization techniques. Each section provides detailed insights and recommendations for managing Spring Boot applications in a production environment, ensuring they can meet the demands of modern enterprise systems while minimizing operational risks.

Monitoring Java-Based Production Systems with Spring Boot Actuator

Monitoring is the cornerstone of effective system management. Without proper monitoring, it's challenging to understand system performance, identify bottlenecks, or troubleshoot issues in a timely manner. In the context of Java-based production systems, particularly those built with Spring Boot, monitoring involves tracking a wide range of metrics, from CPU and memory usage to application-specific parameters like thread count, garbage collection, and response times. Effective monitoring not only provides visibility into the health of the system but also plays a crucial role in proactive issue detection, allowing for preemptive action before problems escalate.

The Role of Spring Boot Actuator in Monitoring

Spring Boot Actuator is an integral part of the Spring ecosystem, offering a wide array of production-ready features that make monitoring and managing Spring Boot applications more straightforward. It provides numerous built-in endpoints that expose application metrics, health information, environment properties, and more. These endpoints can be accessed via HTTP, JMX, or even through a custom UI, making it easier to keep track of the application's status in real-time.

One of the key features of Spring Boot Actuator is its ability to expose detailed metrics about the application. These metrics can include information about the JVM (like memory usage and garbage collection), HTTP request statistics, and custom application metrics. JVM metrics are particularly valuable in understanding how the underlying system resources are being utilized, which is crucial for optimizing performance and avoiding issues like memory leaks or excessive garbage collection pauses. By integrating with popular monitoring tools like Prometheus, Micrometer, or Grafana, Spring Boot Actuator allows for the collection and visualization of these metrics, enabling administrators to monitor the application continuously. [1]

Automated Monitoring with Spring Boot Actuator

Spring Boot Actuator can be configured to automatically collect and expose metrics and health information, which is crucial for maintaining the operational health of production systems. For instance, the /metrics endpoint provides a detailed view of various metrics that are important for understanding how the application is performing under load. This includes detailed insights into how different components of the application are behaving, such as database connections, thread pool usage, and response times for various services. These insights are critical for diagnosing performance issues and understanding where optimization efforts should be focused.

The Actuator’s /health endpoint is particularly valuable as it provides a comprehensive health check of the application. It aggregates the status of various system components, such as the database, message brokers, or any external services the application relies on. This aggregated health status can be configured to include custom health indicators that reflect the specific needs of the application. For instance, a custom health indicator might check the availability of a third-party API that the application depends on, ensuring that any issues with external dependencies are detected and reported immediately. This health information can be used to trigger alerts or automate responses when certain components fail or degrade in performance, ensuring that administrators can take action before the user experience is impacted.

Moreover, Spring Boot Actuator supports custom metrics and health indicators, allowing developers to expose specific metrics relevant to their application. For example, if an application processes orders, developers might create a custom metric to track the number of orders processed per minute or a health indicator that checks the status of the payment gateway. These custom metrics can provide deeper insights into application performance and are essential for monitoring business-critical functions. [2]

Log Management with Spring Boot Actuator

In addition to metrics and health checks, Spring Boot Actuator provides the /logfile endpoint, which can be used to access the application’s log files. This is particularly useful for troubleshooting issues in production without having to access the server directly. Logs are a vital source of information when diagnosing issues, as they provide a detailed record of application behavior over time. Administrators can view and analyze logs through this endpoint, helping them quickly identify and resolve problems. This can be particularly useful in distributed systems, where accessing logs from multiple servers can be cumbersome.

Spring Boot Actuator also integrates with logging frameworks like Logback and Log4j2, allowing for dynamic changes to logging levels at runtime. This means that in a production environment, logging levels can be adjusted without restarting the application, providing greater flexibility in managing log verbosity during different operational conditions. For instance, logging can be increased temporarily to debug a specific issue, then reduced again to minimize performance overhead once the issue has been resolved.

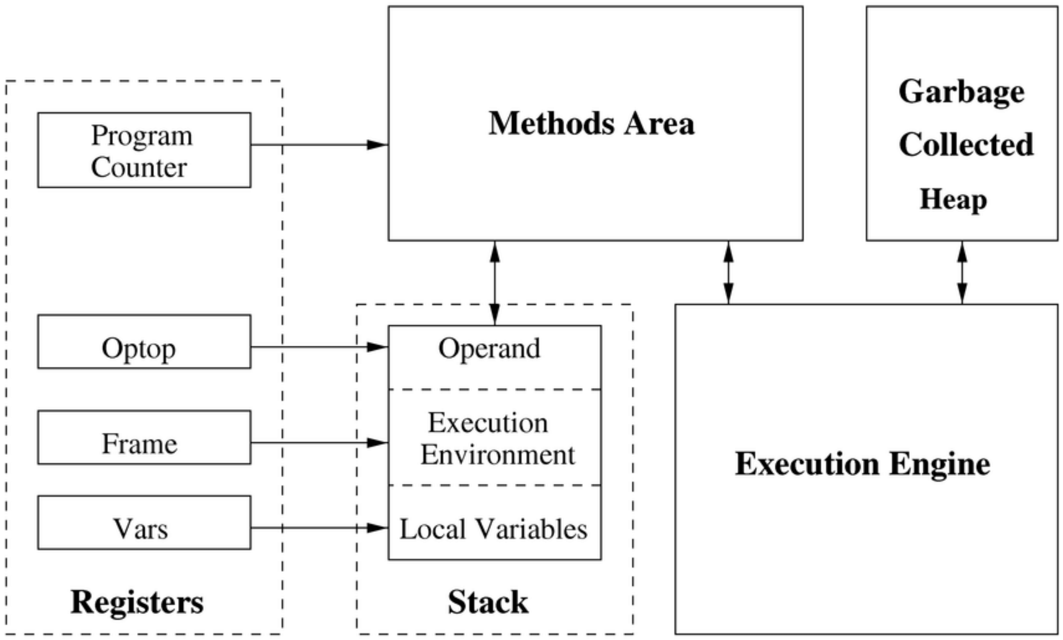
Moreover, logs can be centralized using tools like ELK (Elasticsearch, Logstash, Kibana) or Splunk, which allow for advanced querying, visualization, and alerting based on log data. Spring Boot Actuator’s integration with these logging frameworks enhances the ability to monitor, analyze, and respond to issues in real-time, ensuring that the application remains stable and performant. [3]

Alerting Mechanisms and Spring Boot Actuator

Spring Boot Actuator’s integration with monitoring systems enables robust alerting mechanisms. By forwarding Actuator metrics to systems like Prometheus or Graphite, organizations can set up alerts based on specific conditions. For instance, an alert could be triggered if the memory usage exceeds a certain threshold, if the CPU load is consistently high, or if the response time of a critical endpoint becomes unacceptably high. These alerts can be configured to notify administrators through various channels, such as email, SMS, or integration with incident management tools like PagerDuty or OpsGenie.

The flexibility of Spring Boot Actuator in exposing custom metrics also means that organizations can create highly tailored alerting rules that reflect the specific needs of their application. For instance, if an application handles financial transactions, an alert might be set up to trigger if the number of failed transactions exceeds a certain threshold, indicating a potential issue with the payment gateway. This level of customization ensures that alerts are both meaningful and actionable, reducing the risk of alert fatigue while improving response times to actual issues.

Additionally, Spring Boot Actuator supports integration with advanced monitoring and alerting platforms like Grafana and Prometheus, where custom dashboards can be created to visualize application health in real-time. These dashboards can aggregate data from multiple instances of the application, providing a comprehensive view of system performance across the entire infrastructure. Alerts can be configured based on trends observed in the dashboards, such as a gradual increase in response time or a slow but steady increase in memory usage, allowing administrators to address issues before they impact users.



Scaling Java-Based Systems

Scalability is a critical consideration in the design and management of modern software systems. As businesses grow, their IT systems must scale to accommodate increasing loads and ensure continuous availability and performance. Spring Boot applications, like other Java-based systems, benefit from several strategies to achieve scalability, including horizontal scaling, load balancing, and the use of cloud-native features. Ensuring that an application can scale effectively is key to maintaining its performance and availability as demand increases.

Horizontal Scaling with Spring Boot

Horizontal scaling, or scaling out, involves adding more instances of an application to distribute the load. This approach is particularly effective for handling increased traffic, as it allows the application to process more requests concurrently. Spring Boot applications are particularly well-suited for horizontal scaling because they are typically designed to be stateless or to externalize state management, making it easier to run multiple instances in parallel. [4]

In a microservices architecture, Spring Boot applications can be deployed as independent services that can be scaled independently. This modular approach to application design allows organizations to scale specific services based on demand without having to scale the entire application. For example, a microservice handling user authentication might need to scale differently than a service handling product catalog searches, depending on the load patterns.

Kubernetes, a popular container orchestration platform, is commonly used to manage the deployment and scaling of Spring Boot applications. By packaging Spring Boot applications as Docker containers, organizations can leverage Kubernetes' powerful scaling features, such as the Horizontal Pod Autoscaler, to automatically adjust the number of application instances based on CPU usage, memory consumption, or custom metrics exposed by Spring Boot Actuator. This automated scaling ensures that the application can dynamically adapt to changes in demand, optimizing resource usage and maintaining performance.

Load Balancing in Spring Boot Applications

Load balancing is essential for distributing incoming traffic across multiple instances of a Spring Boot application. Effective load balancing not only helps in managing traffic efficiently but also plays a crucial role in ensuring high availability and fault tolerance. By using load balancers like NGINX, HAProxy, or cloud-native load balancers provided by AWS, Azure, or Google Cloud, organizations can ensure that traffic is evenly distributed and that no single instance becomes a bottleneck.

Spring Boot applications can be configured to work seamlessly with load balancers. For instance, by using Spring Cloud, developers can implement client-side load balancing with Netflix Ribbon or leverage the service discovery features of Spring Cloud Netflix to dynamically route requests to different instances of a service. This dynamic routing is particularly useful in cloud environments where application instances may come and go based on scaling policies.

Additionally, by integrating Spring Boot Actuator with a load balancer, organizations can implement intelligent routing decisions based on the health status of individual application instances. For example, if an instance is marked as unhealthy by the Actuator's /health endpoint, the load balancer can automatically redirect traffic away from that instance until it recovers. This ensures that users are always directed to healthy instances, improving the overall reliability and user experience of the application.

Furthermore, advanced load balancing strategies such as session affinity (also known as sticky sessions) can be implemented if necessary. While Spring Boot applications are typically stateless, there may be scenarios where maintaining session state is important. Load balancers can be configured to route all requests from a specific user session to the same backend instance, ensuring consistency in user experience. However, this approach should be used with caution, as it can lead to uneven load distribution.

Auto-Scaling with Spring Boot and Spring Cloud

Auto-scaling is a key feature in cloud environments that allows applications to scale automatically based on demand. This capability is crucial for handling sudden spikes in traffic, such as those experienced during a product launch or a seasonal sale, without the need for manual intervention. Spring Boot applications can take full advantage of auto-scaling capabilities provided by cloud platforms like AWS, Azure, or Google Cloud, ensuring that they can scale in real-time to meet user demand.

Spring Cloud, a set of tools that builds on top of Spring Boot, provides comprehensive support for auto-scaling. For example, Spring Cloud's integration with Netflix OSS (such as Eureka for service discovery and Ribbon for client-side load balancing) makes it easy to implement auto-scaling in a microservices architecture. Eureka allows for dynamic registration and discovery of services, enabling the system to automatically adjust as new instances are brought online or taken offline.

By using Spring Cloud with Kubernetes, organizations can set up auto-scaling policies that respond to metrics exposed by Spring Boot Actuator, such as request latency, CPU usage, or memory consumption. This ensures that the application scales in real-time to meet demand, optimizing resource usage and maintaining performance. Kubernetes also supports vertical scaling (increasing the resources allocated to a single instance) and horizontal scaling (increasing the number of instances), providing flexibility in how scaling is approached. [5]

Moreover, cloud-native features like autoscaling groups in AWS, Azure Scale Sets, or Google Cloud's Instance Groups can be used to automatically scale the infrastructure underlying the Spring Boot application. These features monitor the load on individual instances and automatically adjust the number of instances based on predefined rules, ensuring that the application remains responsive even under heavy load. This integration between Spring Boot Actuator, Spring Cloud, and cloud-native tools provides a robust solution for managing the scalability of Java-based systems in production environments.

Security in Java-Based Production Systems

Security is a top priority in any production environment, especially for applications that handle sensitive data or provide critical services. As cyber threats become more sophisticated and regulations around data protection more stringent, securing Java-based production systems requires a multi-faceted approach that includes securing the application code, protecting data in transit and at rest, and ensuring that access to the system is tightly controlled. Spring Boot applications come with built-in security features that can be enhanced and extended to meet the specific security requirements of an organization.

Securing Spring Boot Applications

Spring Security, the de facto security framework for Spring applications, provides comprehensive security features that can be easily integrated into Spring Boot applications. It offers support for authentication, authorization, and protection against common security vulnerabilities such as cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection. These vulnerabilities, if left unchecked, can be exploited by attackers to gain unauthorized access to the system, steal data, or disrupt operations.

By default, Spring Security can be configured to secure application endpoints, ensuring that only authenticated users can access protected resources. This is crucial for protecting sensitive parts of a Spring Boot application, such as administrative interfaces or APIs that expose critical functionality. Spring Security supports a wide range of authentication mechanisms, including form-based login, OAuth2, and single sign-on (SSO) via integration with identity providers like Okta, Auth0, or Active Directory. This flexibility allows organizations to implement the authentication strategy that best fits their security requirements.

In addition to securing endpoints, Spring Security provides features for securing the application against session fixation attacks, securing cookies, and enforcing secure communication channels. Developers can also implement fine-grained access controls using Spring Security's powerful authorization features, which allow for the definition of complex access rules based on user roles, permissions, and attributes. For instance, access to certain endpoints can be restricted based on the user's department, job title, or other attributes, ensuring that users only have access to the resources they need. [6]

Encryption and Secure Communication

Securing communication channels is essential for protecting data in transit between the client and the server. Data transmitted over the network is vulnerable to interception by attackers, making it imperative to encrypt sensitive information to prevent unauthorized access. Spring Boot makes it easy to implement HTTPS by configuring SSL/TLS certificates in the application's properties file. This ensures that all communication between clients and the Spring Boot application is encrypted, protecting it from eavesdropping and man-in-the-middle attacks. SSL/TLS also ensures data integrity, preventing attackers from tampering with the data during transmission.

For data at rest, Spring Boot applications can leverage the Java Cryptography Architecture (JCA) to encrypt sensitive data stored in databases or file systems. This is particularly important for protecting sensitive information such as user credentials, payment information, or personally identifiable information (PII) that may be stored in the application's database. Encryption of data at rest ensures that even if an attacker gains access to the underlying storage, they cannot read or modify the data without the encryption keys.

Additionally, Spring Security provides mechanisms for securely storing and managing user credentials, including integration with secure hashing algorithms like bcrypt. Passwords are hashed and salted before being stored, making it significantly more difficult for attackers to retrieve the original passwords even if they gain access to the hashed values. Spring Security also supports advanced security features like multi-factor authentication (MFA), which adds an additional layer of protection by requiring users to provide multiple forms of verification before accessing sensitive resources.

Access Controls and Security Auditing

Access control is another critical aspect of securing Spring Boot applications. Controlling who has access to what parts of the application is essential for protecting sensitive data and ensuring that users can only perform actions that they are authorized to perform. With Spring Security, developers can implement fine-grained access controls using annotations like @PreAuthorize and @Secured to restrict access to specific methods or endpoints based on the user's roles or permissions. These annotations allow for declarative security, where the access rules are defined at the code level and enforced automatically by the framework.

Spring Boot Actuator also plays a role in security by providing endpoints that can be secured using Spring Security. For example, the Actuator's /shutdown endpoint, which allows administrators to gracefully shut down the application, should be protected to prevent unauthorized access. Similarly, the /logfile endpoint should be secured to prevent unauthorized access to sensitive logs, which could contain information that might be useful to an attacker, such as error messages, stack traces, or system configuration details. [7]

Security auditing is also supported by Spring Boot, which can be configured to log security events such as login attempts, access denials, and other significant security-related actions. These logs provide a detailed record of security-related activities within the application, which can be invaluable for detecting potential security threats or for investigating incidents after they occur. For example, by analyzing audit logs, administrators can identify suspicious patterns of behavior, such as repeated failed login attempts, which might indicate a brute-force attack. Integrating Spring Boot with security information and event management (SIEM) systems can further enhance the ability to detect and respond to security threats in real-time.

Performance Optimization

Performance optimization is critical for ensuring that Spring Boot applications can handle the demands of production environments. As the user base grows and the complexity of the application increases, it becomes essential to optimize both the application code and the infrastructure to ensure that the application remains responsive, efficient, and cost-effective. By optimizing the performance of these applications, organizations can improve user experience, reduce operational costs, and ensure that the system can scale effectively.

Code Optimization in Spring Boot Applications

The performance of a Spring Boot application is heavily influenced by the quality of its code. Developers should follow best practices for writing efficient Java code, such as minimizing object creation, using appropriate data structures, and avoiding expensive operations in performance-critical paths. For instance, using collections with optimal performance characteristics (like ArrayList for fast iteration or HashMap for quick lookups) can have a significant impact on the overall performance of the application.

Spring Boot's auto-configuration feature simplifies development by automatically configuring many aspects of the application based on its dependencies. However, this convenience can sometimes lead to suboptimal configurations if not carefully managed. Developers should review the auto-configuration settings and disable any unnecessary features that could impact performance. For example, if an application does not use an embedded database, the default DataSource configuration can be disabled to save resources.

Spring Boot also provides support for caching through the @Cacheable annotation, which can be used to cache the results of expensive operations. Caching can dramatically improve performance by reducing the need to repeatedly perform resource-intensive operations, such as database queries or complex computations. Developers should carefully analyze which parts of the application would benefit most from caching and ensure that the cache is configured to expire entries appropriately to prevent stale data from being served.

Caching and Performance Tuning

Caching is a powerful technique for improving the performance of Spring Boot applications. In addition to method-level caching, Spring Boot supports HTTP caching, which can be used to cache static content such as images, stylesheets, and JavaScript files. This reduces the load on the server and speeds up the delivery of static assets to clients. HTTP caching can be implemented using response headers such as Cache-Control and ETag, which instruct browsers and intermediate proxies to cache resources and reduce the number of requests that reach the server.

Spring Boot Actuator provides metrics that can be used to monitor the effectiveness of caching strategies. For example, developers can track cache hit and miss rates to determine whether the cache is being used effectively. If the cache miss rate is high, it may indicate that the cache is not being populated correctly, or that the cache size is too small to be effective. Monitoring cache performance metrics can help developers fine-tune the cache configuration to maximize its benefits. [8]

Tuning the garbage collection (GC) settings of the JVM is another important aspect of performance optimization. Garbage collection is the process by which the JVM reclaims memory that is no longer in use, and it can have a significant impact on application performance if not properly configured. Spring Boot applications can benefit from selecting the appropriate GC algorithm and tuning the JVM's memory settings based on the application's workload. For example, in applications with large heaps or long-running processes, the G1 garbage collector may be more effective than the default Parallel GC, as it can help reduce the occurrence of long GC pauses.

By monitoring GC metrics exposed by Spring Boot Actuator, administrators can identify performance issues related to garbage collection and make necessary adjustments. For example, if GC pauses are causing latency spikes, increasing the heap size or adjusting the GC algorithm's parameters may help mitigate the issue. Additionally, JVM profilers such as VisualVM, YourKit, or JProfiler can be used to analyze memory usage and identify memory leaks, which can further improve application performance.

Load Testing and Performance Profiling with Spring Boot

Load testing is essential for understanding how a Spring Boot application performs under stress. Load testing involves simulating high levels of traffic and measuring the application’s response to identify bottlenecks, performance degradation, and resource utilization under peak conditions. Tools like Apache JMeter, Gatling, or LoadRunner can be used to simulate high levels of traffic and measure the application’s performance under various load conditions. These tools can generate detailed reports that highlight how the application performs as the number of users increases, allowing organizations to identify potential bottlenecks and optimize the application to handle increased demand.

Performance profiling is another critical aspect of performance optimization. Profiling involves analyzing the application’s runtime behavior to identify areas where performance can be improved. Spring Boot applications can be profiled using tools like YourKit, JProfiler, or VisualVM to analyze CPU usage, memory allocation, and thread activity. Profiling helps developers identify inefficient code paths, such as methods that consume excessive CPU time or create large numbers of objects, and optimize them to improve overall performance.

Spring Boot Actuator provides the /metrics endpoint, which can be used to gather detailed performance metrics during load testing and profiling. These metrics can be visualized using tools like Grafana, allowing administrators to monitor the application’s performance in real-time and make data-driven decisions about optimizations. For example, if load testing reveals that the application’s response time degrades significantly under heavy load, the metrics can help identify whether the issue is related to CPU saturation, memory exhaustion, or another factor, guiding the optimization efforts.

Moreover, integrating performance testing and profiling into the continuous integration/continuous deployment (CI/CD) pipeline can ensure that performance regressions are detected early in the development process. Automated performance tests can be run as part of the build process, and profiling data can be collected and analyzed after each build, providing continuous feedback on the impact of code changes on application performance.

Conclusion

Managing Java-based production systems, particularly those built with Spring Boot, is a complex but essential task that requires a comprehensive approach to ensure reliability, security, and performance. These systems are at the core of many enterprise applications, and their effective management is critical to the success of the business. By leveraging Spring Boot Actuator alongside other best practices in monitoring, scalability, security, and performance optimization, organizations can maintain their systems in peak condition and reduce operational risks.

Spring Boot Actuator plays a pivotal role in providing the insights and tools needed to monitor, manage, and optimize Spring Boot applications. It offers a wide range of endpoints that expose valuable metrics, health information, and logs, making it easier to maintain the operational health of production systems. By continuously monitoring these metrics, organizations can gain real-time visibility into the performance and health of their applications, allowing them to detect and address issues before they impact users. [9]

By following the guidelines outlined in this paper, organizations can build and maintain Spring Boot applications that are robust, scalable, secure, and optimized for performance. This not only enhances the user experience but also contributes to the overall success of the business by ensuring that critical systems are always available and operating at their best. As the demand for more reliable, scalable, and secure applications continues to grow, adopting these best practices will be essential for staying competitive in today's fast-paced digital landscape.

References

[1] Liu Y., "A survey on ai for storage.", CCF Transactions on High Performance Computing, vol. 4, no. 3, 2022, pp. 233-264.

[2] Liu J., "Sora: a latency sensitive approach for microservice soft resource adaptation.", Middleware 2023 - Proceedings of the 24th ACM/IFIP International Middleware Conference, 2023, pp. 43-56.

[3] Jani, Y. "Spring boot actuator: Monitoring and managing production-ready applications." European Journal of Advances in Engineering and Technology 8.1 (2021): 107-112.

[4] Chen Y., "A survey on industrial information integration 2016–2019.", Journal of Industrial Integration and Management, vol. 5, no. 1, 2020, pp. 33-163.

[5] Furrer F.J., "Safety and security of cyber-physical systems: engineering dependable software using principle-based development.", Safety and Security of Cyber-Physical Systems: Engineering dependable Software using Principle-based Development, 2022, pp. 1-537.

[6] Varghese B., "A survey on edge performance benchmarking.", ACM Computing Surveys, vol. 54, no. 3, 2021.

[7] Xiong W., "Advances in security analysis of software-defined networking flow rules.", Xi'an Dianzi Keji Daxue Xuebao/Journal of Xidian University, vol. 50, no. 6, 2023, pp. 172-194.

[8] Meiklejohn C., "Method overloading the circuit.", SoCC 2022 - Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 273-288.

[9] Raj P., "Cloud-native computing: how to design, develop, and secure microservices and event-driven applications.", Cloud-native Computing: How to Design, Develop, and Secure Microservices and Event-Driven Applications, 2022, pp. 1-331.