International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V73I6P104

Volume 73 Issue 6, 26-33, June 2025

Original Article

Advanced Techniques for Microservices Debugging

Madhuri Kolla[1], Krishna Vinnakota[2]

[1] *AT&T Bothell, WA, USA*
[2] *Microsoft, Redmond, USA.*

Corresponding Author : kolla.madhuri1989@gmail.com

Received: 25 April 2025 Revised: 29 May 2025 Accepted: 13 June 2025 Published: 28 June 2025

Abstract *- Microservices architecture is critical in building applications that can grow and be easily maintained. Most industries like e-commerce, finance, telecom, and insurance have chosen microservices architecture due to its scalability and robustness. However, microservice brings significant operational challenges regarding debugging, even if it is scalable. These debugging complexities often worsen as the number of services and their distribution increase. Unlike Monolithic, microservices are minimal and easily deployable individual units as microservices have continuous delivery, faster development cycles, and improved fault isolation. Several companies have used open-source technology, such as Spring Boot. SpringBoot is easy to use and offers several useful features. However, several challenges are associated with debugging microservices if we do not choose the right approaches. This article presents some important debugging techniques that are followed and used by several industries to maintain the stability of microservices.*

Keywords *- Spring Boot, debugging, distributed systems, observability, logs, tracing, microservices, architecture, scalability, Spring Boot, containerization, service discovery, Resilience, Reliability.*

1. Introduction

The rapid adoption of microservices architecture has fundamentally transformed the contemporary software engineering landscape. The microservice architecture promotes the decomposition of monolithic systems into smaller, self-contained, and independently deployable services, each implementing a specific business capability and communicating through protocols such as HTTP or messaging queues. Continuous delivery, faster development cycles, and improved fault isolation towards more agile and resilient software systems are achieved through the adoption of microservices. Spring Boot framework is used to build microservices as it is easy and has many significant features.

The asynchronous nature of microservices brings many debugging challenges. Root cause analysis is complex to know as the different errors can occur with multiple services, and the significant use of event-driven models can obscure call stacks, complicating the process of following the flow of execution. Dynamic configurations, often environment-specific, can lead to unpredictable behavior that is challenging to diagnose, while hidden exceptions may be improperly handled or swallowed, preventing precise error detection. Microservices-based systems' sustained success and stability can be achieved effectively using advanced debugging techniques. This article presents a cohesive and practical framework of advanced debugging techniques tailored explicitly for Spring Boot microservices

2. Literature Review

This article explains the challenges and debugging techniques for commonly used frameworks like spring boot-based microservices and also highlights its relevance to contemporary software engineering practices.

2.1. The Paradigm Shift to Microservices and Inherent Debugging Complexities

Modern IT ecosystems have evolved significantly using microservices architecture instead of monolithic applications. This article explains the significant benefits of microservices, such as enhanced scalability, agility, fault isolation, and technological diversity. Microservices facilitate continuous delivery and faster development cycles compared to monolithic-based applications.

However, Debugging complexities have been increased with the adoption of microservices architecture due to its distributed and often asynchronous nature. Due to the distributed nature, the traditional debugging methods are insufficient. This article identifies several common challenges associated with microservices, such as distributed architecture, asynchronous communication, dynamic configuration, and operational overhead.

2.2. Debugging Techniques for Spring Boot Microservices

The article explains the five core debugging techniques relevant to Spring Boot microservices.

2.2.1. Local Debugging

Early issue identification is detected using local debugging techniques like leveraging IDE features (breakpoints, variable inspection), enabling remote debugging, utilizing Spring Boot DevTools for rapid iteration, and effectively configuring logging levels. The inclusion of Spring Actuator endpoints and profile-specific configurations demonstrates an understanding of Spring Boot's native capabilities for runtime introspection, which is a standard recommendation in Spring development guides (e.g., [10]).

2.2.2. Centralized Logging

Recognizing the distributed nature of microservices, the paper advocates for centralized logging as an indispensable tool for visibility and rapid incident response. It recommends deploying robust logging stacks (ELK, EFK, Grafana Loki) and stresses the importance of standardized log formats, contextual metadata (e.g., requestId/correlationId), and asynchronous logging for production environments. The emphasis on correlation IDs aligns with best practices for tracing requests across service boundaries, a critical aspect of observability in distributed systems (e.g., [6]).

2.2.3. Distributed Tracing

The identification of performance bottlenecks and failures is achieved with this technique. Distributed tracing tracks the end-to-end flow of requests across multiple services. The paper highlights the role of Spring Cloud Sleuth for automatic trace ID generation and its integration with tracing backends like Zipkin, Jaeger, and OpenTelemetry. The discussion on propagating trace context, creating custom spans, and visualizing traces through dashboards reflects a deep understanding of how to gain granular visibility into service interactions, a key tenet of modern observability platforms (e.g., [7], [8]).

2.2.4. Health and Metrics Monitoring

Continuous monitoring of application status and performance indicators is identified as essential for ensuring reliability and efficiency. The authors detail the use of Spring Boot Actuator for exposing health endpoints and Micrometer for exporting metrics to tools like Prometheus and Grafana.

The emphasis on collecting various metrics (JVM, HTTP, DB, custom business metrics) and configuring alerts underscores a proactive approach to operational health. Kubernetes Readiness and Liveness Probes demonstrate an awareness of deployment-specific monitoring needs in containerized environments. This aligns with the principles of robust system monitoring and alerting (e.g., [12], [13]).

2.2.5. Container-Level Debugging

Given the prevalence of containerization (Docker, Kubernetes) in microservices deployments, the paper dedicates a section to debugging within these isolated environments. Techniques such as docker exec, kubectl exec, inspecting container logs, enabling remote debugging within containers, and analyzing environment variables and running processes are discussed.

Advanced strategies for troubleshooting production-like issues like the sidecar debugging containers and Alpine debug images showcased without disrupting the running application, reflecting practical insights into container orchestration challenges (e.g., [14], [15]).

This article effectively articulates debugging practices as critical for microservices-based applications' stability, maintainability, and success. The paper also provides a valuable guide for developers and operations teams navigating the complexities of distributed systems.

3. What is Microservices Architecture?

A microservices-based application is composed of the following key components:

- Service Independence: Each service operates independently and can be deployed, updated, scaled, or restarted without affecting other services.
- Domain-Driven Design (DDD): Services are modeled around business domains and bounded contexts.
- Decentralized Data Management: Each microservice has its database, and they are loosely coupled in nature.
- API Gateway: A central entry point for external clients to access internal microservices.
- Service Discovery: Automatically registers and locates services within the system.

3.1. Advantages of Microservices

- Scalability: They can be scaled independently based on usage and service performance.
- Agility: Teams can develop, deploy, and scale services independently, promoting faster release cycles.
- Fault Isolation: Failures are contained within the affected service, improving system resilience.
- Technology Diversity: Each microservice can be built using the most suitable technology stack.

3.2. Disadvantages of Microservices

- Complex Deployment and DevOps: Requires container orchestration and CI/CD pipelines.
- Service Coordination and Communication: Complex inter-service communication patterns and potential for latency.
- Data Consistency: Ensuring consistency in a distributed environment is complex (e.g., eventual consistency).
- Monitoring and Debugging: Requires distributed tracing, centralized logging, and real-time monitoring tools.

3.3. Common Debugging Challenges

As discussed, some of the disadvantages of microservices below are common debugging challenges faced with microservices.

3.3.1. Distributed Architecture

Root cause analysis is complex as the error can be across multiple services.

3.3.2. Asynchronous Communication

The use of queues and event-driven models obscures call stacks.

3.3.3. Dynamic Configuration

Environment-specific properties can cause unexpected behavior.

3.3.4. Hidden Exceptions

Exceptions may be swallowed or improperly logged.

4. Debugging Techniques

Here are some debugging techniques that can be used for spring boot-based microservices.

- Local Debugging
- Centralized Logging
- Distributed Tracing
- Health and Metrics Monitoring
- Container-level Debugging

4.1. Local Debugging

Local debugging is convenient and necessary for ensuring the stability, correctness, and performance of Spring-based applications before they hit higher environments.

4.1.1. Importance of Local Debugging

Local debugging is very convenient, and it ensures the stability, correctness, and performance of Spring-based applications before they are deployed to production.

4.1.2. Importance of Local Debugging

Local debugging allows developers to identify and fix issues early in the development cycle. It provides complete control to inspect code behavior, set breakpoints, and test changes in real-time. This reduces deployment errors, speeds up development, and ensures higher code quality before integration into larger systems.

Below are some local debugging techniques that can be used for microservice applications:

- Use an IDE with Debug Support: Use IntelliJ IDEA, Eclipse, or VS Code for setting breakpoints and inspecting variables. one can run the application in debug mode using the IDE's built-in debugger.
- Enable Debug Mode Manually: You can start the Spring Boot application with remote debugging enabled:

```
java-
agentlib:jdwp=transport=dt_socket,server=y,suspend=n,add
ress=*:5005 -jar your-app.jar
```

This allows you to attach the debugger from an IDE.

- Spring Boot DevTools: Spring boot provides dev tools for automatic restarts and live reloads during development.

Below is the dependency:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
</dependency>
```

- Use Logs Effectively: Enable appropriate logging levels in application.properties of spring boot application:

```
logging.level.org.springframework=DEBUG
logging.level.com.yourpackage=TRACE
```

- Check Spring Actuator Endpoints: Use Spring Boot Actuator for runtime insights Endpoints: /actuator/health, /actuator/beans, /actuator/env, etc. Helps debug environment variables, loaded beans, configurations, etc.

- Profile-Specific Configuration: Run with specific profiles to debug configuration issues using the below command:

```
java -jar app.jar --spring.profiles.active=dev
```

- Inspect Request Mappings: Use the below command to check all exposed REST endpoints and routes.

```
curl http://localhost:8080/actuator/mappings
```

- Test with Postman or Curl: Validate API behavior independently using tools like Postman or CLI tools like Curl.
- Mock External Services: Use @MockBean or tools like WireMock to simulate external systems for isolated debugging
- Analyze Stack Traces: Carefully read Java stack traces to trace the root cause of runtime exceptions or NullPointerException.
- Database Debugging: Use in-memory databases like H2 during development.

Enable H2 console:

```
spring.h2.console.enabled=true
```

4.2. Centralized Logging

Centralized logging enables traceability for microservices and rapid incident response and is key to maintaining reliability and security in distributed applications. Modern microservices and many cloud-based applications use centralized logging.

4.2.1. Importance of Centralized Logging

Centralized logging is vital for monitoring and troubleshooting in distributed systems. Logs can be viewed in one place as they are collected from multiple applications. Consolidating logs from multiple services helps in quicker root cause analysis and easier debugging, and it also helps in security auditing. It also supports real-time monitoring, alerting, and compliance, improving system observability and operational efficiency.

Here are key points and techniques for implementing and using centralized logging in Spring Boot microservices or any distributed system:

- Deploy a logging stack like: ELK Stack (Elasticsearch, Logstash, Kibana), EFK Stack (Elasticsearch, Fluentd, Kibana), Grafana Loki + Promtail, Splunk, Graylog, or Datadog Logs.
- Standardize Log Format - Ensure all services use a consistent log structure (e.g., JSON format). Helps with parsing, filtering, and querying logs. Spring Boot: Use logback with JSON encoders (e.g., logstash-logback-encoder).
- Add Contextual Metadata- Include important fields in logs like timestamp, service name, environment, requestId or correlationId, userId (if applicable), and log level.
- Use Correlation IDs - Pass a correlation ID through HTTP headers or MDC (Mapped Diagnostic Context). Enables tracing a request across multiple services.

Spring Boot Example:
MDC.put("correlationId",
UUID.randomUUID().toString());

- Configure Centralized Log Shipping by setting up log forwarders to send logs from services to the aggregation system, such as Logstash, Fluentd, or Promtail, to collect logs—ship logs from stdout, files, or syslog.
- Enable asynchronous logging by avoiding I/O bottlenecks by using async logging in production:
- (Elasticsearch, Logstash, Kibana), EFK Stack (Elasticsearch, Fluentd, Kibana), Grafana Loki + Promtail, Splunk, Graylog, or Datadog Logs.
- Standardize Log Format - Ensure all services use a consistent log structure (e.g., JSON format). Helps with parsing, filtering, and querying logs. Spring Boot: Use logback with JSON encoders (e.g., logstash-logback-encoder).
- Add Contextual Metadata- Include important fields in logs like timestamp, service name, environment, requestId or correlationId, userId (if applicable), and log level.
- Use Correlation IDs - Pass a correlation ID through HTTP headers or MDC (Mapped Diagnostic Context). Enables tracing a request across multiple services.

Spring Boot Example:
MDC.put("correlationId",
UUID.randomUUID().toString());

- Configure Centralized Log Shipping by setting up log forwarders to send logs from services to the aggregation system, such as Logstash, Fluentd, or Promtail, to collect logs—ship logs from stdout, files, or syslog.
- Enable asynchronous logging by avoiding I/O bottlenecks by using async logging in production:

<appender class=" ch.qos.logback.classic.AsyncAppender">

- Separate Log Levels by Environment Use different log levels (INFO, DEBUG, ERROR) for dev, test, and prod environments. Avoid verbose logs in production unless needed for diagnostics.
- Protect Sensitive Data: Never log passwords, tokens, or personal data. Apply log sanitization to mask sensitive information.
- Use Logging Libraries Consistently. Try to Use SLF4J + Logback in Spring Boot. Example of a logging *statement in Java:*

private static final Logger logger = LoggerFactory.getLogger(MyClass.class);
logger.info("Processing user {}", userId);

- Implement dashboards and alerts by creating dashboards (e.g., in Kibana or Grafana) for Error trends, Request failures, and Service-specific logs. Set up alerts for high error rates, service downtimes, or anomalies.

4.2.2. Helpful Log Message Practices

- Use clear and actionable messages.
- Include method names, parameters (when safe), and exception causes.
- Use structured logging (key-value pairs) for better parsing.

4.3. Distributed Tracing

The end-to-end flow of a request across multiple services in a microservices architecture is easily understood by distributed tracing. It helps identify performance bottlenecks, failures, and latency issues by recording each transaction step across system boundaries. Distributed tracing is a critical

observability practice for microservices. Teams can diagnose, debug, and optimize the performance of a microservice, which can be achieved with distributed tracing. It helps in getting to know the requests for end-to-end visibility.

4.3.1. Importance of Distributed Tracing

Distributed tracing is essential for tracking requests across microservices, enabling end-to-end visibility into system behavior. Performance bottlenecks, trace failures, and analyze latency issues. Connecting service interactions with trace IDs improves debugging, enhances observability, and ensures reliability in complex, distributed architectures.

Below are important points for implementing distributed tracing in distributed systems or Spring Boot microservices.

Below are important points for implementing distributed tracing in distributed systems or Spring Boot microservices.

1. Use tracing libraries in Spring Boot:

- Use Spring Cloud Sleuth to trace/span ID generation automatically.
- Integrates seamlessly with Zipkin, Jaeger, OpenTelemetry, and Grafana Tempo.

Dependency (Maven):

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

2. Export Traces to a Tracing Backend

- Standard options include Zipkin, Jaeger, Elastic APM, AWS X-Ray, and OpenTelemetry Collector. Spring Boot + Zipkin Example:

```
spring.zipkin.base-url=http://localhost:9411/
spring.sleuth.sampler.probability=1.0     # 100% tracing in dev
```

3. Propagate Trace Context Across Services

- Ensure that Trace-ID and Span-ID headers are forwarded in HTTP requests or messaging headers.
- Use filters/interceptors to propagate and log context.

4. Add Custom Spans for Critical Sections

- Use Tracer or @NewSpan to create spans in Spring Boot manually.

Example:

```
@Autowired
private Tracer tracer;
public void customTraceLogic() {
Span newSpan = tracer.nextSpan().name("custom-operation").start();
try (Tracer.SpanInScope ws = tracer.withSpan(newSpan.start())) {
        // code to trace
   } finally {
         newSpan.end();
         }
}
```

5. Use Correlation IDs in Logs

- Combine distributed tracing with centralized logging to correlate logs with trace IDs.
- MDC (Mapped Diagnostic Context) helps include trace data in log messages.

6. Visualize and Analyze Traces

- Use tracing dashboards (e.g., Zipkin UI, Jaeger UI, Grafana Tempo) to:
  - View trace timelines
  - Analyze latency
  - Detect failed spans

7. Integrate with Metrics and Logging

- Combine traces with Prometheus metrics and ELK logs for complete observability.
- Some tools (e.g., OpenTelemetry) unify metrics, logs, and traces.

4.3.2. Best Practices for Distributed Tracing

- Trace all incoming and outgoing calls to ensure complete trace coverage.
- Sample selectively in production reduces overhead by not tracing every request.
- Use meaningful names so that it is easier to understand trace diagrams.
- Enrich spans with custom tags as it improves filtering and querying.
- Monitor trace latency thresholds to alert on slow services automatically.

4.4. Health and Metrics Monitoring

Health and monitoring tools continuously check the status and performance of spring boot-based microservices to ensure the application runs correctly and efficiently. Health and metrics monitoring is essential for ensuring application reliability and performance. The collection of performance metrics can be done using health and monitoring tools. These tools can monitor system health and respond to issues in real time.

4.4.1. Importance of Health and Metrics Monitoring

Health and Metrics Monitoring helps detect issues in the early phase, supports proactive maintenance, and helps scale

efficiently. Availability, performance, and reliability can be achieved by health and monitoring tools like spring boot actuators, Prometheus and Micrometers. Teams can prevent downtime, optimize performance, and ensure a seamless user experience by tracking system health and resource usage.

Here are some Important Points and Techniques for Health and Metrics Monitoring:

1. Spring Boot Actuator can show the health endpoints of a microservice.

<dependency><groupId>org.springframework.boot</groupId><artifactId>spring-boot-starter-actuator</artifactId></dependency>

- Enable endpoints inapplication.properties : *management.endpoints.web.exposure.include=health, info, metrics*

2. Use Built-in and Custom Health Indicators

Spring Boot provides default health indicators (DB, disk space, Redis, etc.). Create custom indicators like the below:

```
@Component
public class MyHealthIndicator implements HealthIndicator
@Override
public Health health() {
        // custom logic
        return Health.up().withDetail("status", "OK").build();
        } }
```

3. Expose Metrics for Monitoring Tools

Metrics exposed at /actuator/metrics Use Micrometer (included in Spring Boot) to export metrics to:

- Prometheus
- InfluxDB
- Graphite
- Datadog

4. Collect Application and System Metrics and common metrics to monitor:

- JVM: memory usage, GC time
- HTTP: request count, error rate, latency
- DB: connection pool usage, query time
- Custom business metrics (e.g., transactions processed)

5. Integrate with Visualization Tools, export metrics to Prometheus, and visualize in Grafana. Dashboards and it should show below things

- CPU/memory usage
- Request response times
- Service Uptime
- Error rate trends

6. Set Up Alerts like below

Configure alert rules (e.g., high error rate, low memory, unhealthy service)

Send alerts via:

- Email
- Slack
- PagerDuty

7. Use Readiness and Liveness Probes (Kubernetes) and

Define your deployment.yaml:

```
readinessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
livenessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
```

8. Tag and Filter Metrics

- Add tags like instance, region, service, and status for easier filtering.
- Useful for multi-instance and multi-region deployments.

9. Use Distributed Tracing with Metrics

- Correlate traces and metrics for complete observability.
- Tools: OpenTelemetry, Spring Sleuth + Zipkin/Jaeger

10. Ensure Security on Monitoring Endpoints

- Do not expose health or metrics endpoints publicly without security.
- Restrict access using roles or firewall rules.

4.5. Container-level Debugging

Container-level debugging involves inspecting and troubleshooting application behavior inside a container (e.g., Docker) to diagnose runtime issues such as crashes, misconfigurations, or resource bottlenecks. Container-level debugging is critical when troubleshooting production-like environments. Teams can identify broken dependencies and runtime issues that don't occur in local development.

4.5.1. Importance of Container-level Debugging:

Container-level debugging is crucial in identifying configuration-related issues, network problems and runtime failures in environments like kubernetes or Docker. Container-level debugging is crucial for diagnosing issues in environments like Docker or Kubernetes. Container-level debugging can access logs, environment variables, and

system states, and developers can resolve issues quickly and ensure application stability in production environments.

Here are some Key Techniques for Container-Level Debugging

1. Use docker exec to access the container. Run an interactive shell inside a running container:

docker exec -it <container_id> /bin/sh

Useful for inspecting logs, config files, environment variables, or running diagnostic commands.

2. Check Container Logs to view application logs from stdout/stderr:

For Docker :

docker logs <container_i

For Kubernetes:

kubectl logs <pod_name>

3. Use kubectl exec for kubernetes pods. Run commands inside a Kubernetes container:

kubectl exec -it <pod_name> -- /bin/bash

Good for checking running processes, files, or container environments.

4. Enable Remote Debugging in Spring Boot:

- Run Spring Boot with remote debugging enabled: *-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005 -jar app.jar*
- Expose port 5005 in Dockerfile or Kubernetes deployment for IDE debugging.

5. Inspect Environment Variables

Inside the container: *printenv* Helps confirm configuration passed via ENV, -e, or Kubernetes ConfigMaps and Secrets.

6. Analyze Running Processes

List running processes inside the container: *ps aux, top*. Detects if the main application process is running or if it is stuck.

7. Use Sidecar Debugging Containers

- In Kubernetes, attach a temporary debug container (sidecar) to access a pod without modifying the running app.

8. Check Network Connectivity

- Use Curl, ping, or telnet to check connectivity to databases, APIs, or services from inside the container.
- It helps diagnose DNS or firewall issues.

9. Volume & File Inspection Inspect mounted volumes or files:

ls /path/to/mount
cat /path/to/config.yaml

- Confirm that the configuration is correctly mounted and accessible.

10. Use Alpine Debug Images:

If your container is minimal (Alpine, distroless), add debugging tools or use a debugging base image:

kubectl debug <pod> --image=busybox --target=app

5. Conclusion

Debugging techniques are critical in microservices architecture due to the distributed and decentralized nature of the system. Unlike monolithic applications, microservices consist of multiple independently deployed components interacting with networks. This complexity makes identifying the root cause of issues more challenging. Effective debugging techniques—such as centralized logging, distributed tracing, health checks, and container-level debugging—are essential for gaining visibility into system behavior and diagnosing problems efficiently. These techniques enable developers and operations teams to isolate failures, trace request flows across services, and monitor real-time performance metrics. They help reduce Mean Time To Resolution (MTTR), which is crucial for maintaining service uptime and user satisfaction.

Debugging also plays a vital role in performance optimization, helping to identify bottlenecks such as slow API responses or resource exhaustion. Proactive debugging strategies ensure reliability, scalability, and resilience in production environments by detecting issues early and allowing graceful degradation. Furthermore, debugging tools support agile and DevOps workflows by enabling rapid diagnosis during continuous integration and delivery cycles. Overall, robust debugging practices are essential for ensuring the stability, maintainability, and success of microservices-based applications in modern cloud-native architectures.

References

[1] Pivotal Software, Spring Boot Documentation. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/
[2] Baeldung, Spring Boot Debugging Techniques. [Online]. Available: https://www.baeldung.com/spring-boot-debugging
[3] JetBrains, Remote Debugging with IntelliJ IDEA, 2024. [Online]. Available: https://www.jetbrains.com/help/idea/remote-debugging.html
[4] Elastic, What is the ELK Stack?. [Online]. Available: https://www.elastic.co/what-is/elk-stack

[5] Grafana Labs, Loki: Like Prometheus, but for Logs. [Online]. Available: https://grafana.com/oss/loki/
[6] Ran Ramati, Best Practices for Centralized Logging in Microservices, Logz.io. [Online]. Available: https://logz.io/blog/logging-best-practices/
[7] OpenZipkin, Distributed Tracing Made Easy. [Online]. Available: https://zipkin.io
[8] OpenTelemetry, Getting Started with Distributed Tracing. [Online]. Available: https://opentelemetry.io/docs/
[9] Spring, Spring Cloud Sleuth. [Online]. Available: https://spring.io/projects/spring-cloud-sleuth
[10] Spring, Spring Boot Actuator Endpoints. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/actuator-api/html/
[11] Micrometer, Application Metrics for JVM-Based Systems. [Online]. Available: https://micrometer.io
[12] Prometheus, Prometheus Documentation. [Online]. Available: https://prometheus.io/docs/introduction/overview/
[13] Grafana Labs, Grafana Documentation. [Online]. Available: https://grafana.com/docs/grafana/latest/
[14] Docker, docker exec. [Online]. Available: https://docs.docker.com/engine/reference/commandline/exec
[15] Kubernetes, Debugging Pods and Containers. [Online]. Available: https://kubernetes.io/docs/tasks/debug/