ISSN: 3064-996X | Volume 2, Issue 1

Open Access | PP: 31-36

DOI: https://doi.org/10.70315/uloap.ulete.2025.0201006

Using Spring Boot to Simplify the Development of Multi-Module Services

Shyrobokov Valentyn

Senior Java Developer in SAPIENS, Holon, Israel.

Abstract

This article examines the role of the Spring Boot framework in simplifying the development and maintenance of multi-module (microservice) applications. The focus is placed on analyzing contemporary scientific and professional literature, highlighting how Spring Boot addresses key challenges of microservice architecture: simplifying configuration, standardizing dependency management, and providing high flexibility for scaling individual modules. Particular attention is given to the Spring Cloud ecosystem, which enables centralized management of configurations and security, as well as containerization (Docker) and orchestration (Kubernetes) tools that are closely integrated with Spring Boot.

Microservice patterns such as API Gateway, Circuit Breaker, and Config Server are explored, along with resilience and observability mechanisms supported by Spring Boot Actuator and service mesh architectures (e.g., Istio). A comparative analysis of deployment approaches for monolithic and microservice systems is presented, emphasizing the advantages of modular structure in the context of continuous integration and delivery (CI/CD). The study consolidates findings from various research efforts, including experimental studies on the performance and stability of Java applications built with Spring Boot, underscoring the practical value of the proposed solutions. This article will be useful for IT professionals, architects, and researchers involved in designing and implementing microservice architecture in industrial environments.

Keywords: *Spring Boot, Microservice Architecture, Multi-Module Services, Auto-Configuration, Devops, Docker, Kubernetes, Monitoring.*

INTRODUCTION

The modern software industry increasingly transitions from monolithic applications to microservice (multi-module) architecture, driven by the growing demand for scalability, flexibility, and faster time-to-market for new features. This approach involves breaking a system into independent modules (services), each of which is developed and deployed separately. However, this transition introduces several challenges related to dependency management, data consistency, secure inter-service communication, and maintaining overall system stability.

The relevance of this topic is underscored by the rising demand for high-load yet adaptive software solutions capable of seamless expansion to meet evolving business requirements. Under the traditional monolithic approach, scaling often necessitates extensive restructuring of the application, and changes to one module can affect the entire system. In contrast, microservice architecture emphasizes module isolation, enabling faster feature delivery, independent development, and testing. Nevertheless, this approach remains non-trivial in terms of environment configuration and development workflow organization.

The Spring Boot framework and its associated tools (Spring Cloud, Spring Security, Spring Actuator, etc.) have established themselves in recent years as one of the most popular technology stacks for implementing microservice systems in Java. Automated configuration, pre-packaged "starter" dependencies, and deep integration with containerization (Docker) and orchestration (Kubernetes) significantly reduce the time required for initial module deployment while enhancing service reliability and manageability. This allows developers to focus more on business logic rather than getting bogged down in excessive technical details.

Despite the widespread adoption of Spring Boot, questions remain regarding how the framework simplifies multi-module development, what its key advantages are, and which implementation patterns are most effective. A systematic analysis of existing scientific literature and professional guides on microservice architecture is necessary to form a comprehensive understanding of why Spring Boot facilitates the adoption of multi-module solutions, which patterns and tools are in demand, and what major challenges need to be addressed.

Citation: Shyrobokov Valentyn, "Using Spring Boot to Simplify the Development of Multi-Module Services", Universal Library of Engineering Technology, 2025; 2(1): 31-36. DOI: https://doi.org/10.70315/uloap.ulete.2025.0201006.

MATERIALS AND METHODS

The preparation of this article involved the study of works by various authors addressing different aspects of microservice architecture and the advantages of using Spring Boot to simplify the development of multi-module applications. M. Abbadi and M. Debnath [1] provide a practical case study on creating a cloud-native microservice application, highlighting the importance of integrating Spring Boot with Kubernetes. Similarly, N. Alshuqayran, N. Ali, and R. Evans [2] systematize the primary challenges and solutions associated with transitioning to microservices and emphasize the significance of specialized frameworks, including Spring Boot, in implementing distributed systems. M. Heckler [3] focuses on practical techniques for using Spring Boot in Java and Kotlin applications, noting its ability to lower the entry barrier for building multi-module services.

S. Newman [4], in a broader context of microservice architecture, confirms that leveraging pre-configured tools simplifies the structuring and maintenance of high-load systems. C. Pautasso et al. [5] explore the design of microservices based on fine-grained services, addressing the role of Spring Boot in their deployment, while C. Posta and B. Sutter [6] describe the integration of Spring Boot applications with service mesh architectures (e.g., Istio). Containerization (Docker) and orchestration (Kubernetes) are discussed in the works of A. Mhatre [7], as well as I. Singh and V. Bhatnagar [9], where the authors present experimental results on the performance of microservices deployed in Docker containers.

S. Richardson [8] focuses on patterns such as API Gateway, Circuit Breaker, and Saga, emphasizing their efficient implementation with Spring Boot and Spring Cloud. K. Walls [10] highlights practical best practices for structuring multi-module Java applications and configuring CI/CD processes.

Thus, the study examined the theoretical and practical aspects of using Spring Boot during the transition to microservice architecture, considering issues of configuration, monitoring, containerization, and scaling.

The following methods were employed in preparing this work:

- **Comparative analysis** – comparing approaches to designing monolithic and microservice systems.
- **Data synthesis from scientific publications and professional guides** – forming a comprehensive understanding of the role of Spring Boot in simplifying the development, testing, and scaling of multi-module services.
- **Systematization** – structuring information about the key challenges of microservice architectures and corresponding solutions based on the Spring ecosystem.
- **Critical evaluation** – analyzing the advantages and limitations of Spring Boot, including the need for proper service decomposition and the complexity of organizational changes during microservice architecture implementation.

The application of these methods made it possible to identify the main factors influencing the successful use of Spring Boot in industrial projects and to determine directions for future research in automating configuration, monitoring, and ensuring the resilience of multi-module applications.

RESULTS

The modern development of microservice architectures is closely associated with the need to simplify and accelerate the creation of multi-module services. The use of Spring Boot in the design and deployment of services enables flexibility, scalability, and manageability of applications under changing business requirements. The analysis of theoretical materials [1–10] demonstrates that Spring Boot simplifies project structure and introduces ready-made mechanisms for the development, testing, and operation of multi-module systems.

From a cloud-native perspective, M. Abbadi and M. Debnath [1] emphasize the importance of combining Spring Boot with orchestration systems (e.g., Kubernetes) in building microservice architectures. Their study presents a practical deployment case where Spring Boot streamlines the configuration of service components (REST interfaces, database settings, security) and facilitates faster integration of services. The authors [1] highlight that Spring Boot reduces the need for boilerplate code and, through pre-configured auto-configuration mechanisms, simplifies the assembly of multi-module projects.

This view is supported by the work of N. Alshuqayran, N. Ali, and R. Evans [2], which provides a systematic mapping study of challenges and solutions associated with microservice architectures. The researchers [2] classify the difficulties encountered during the transition to microservices and emphasize the importance of specialized frameworks (such as Spring Boot) in overcoming various architectural, organizational, and technological barriers. In particular, their review highlights the role of ready-made solutions for configuration, dependency management, and the organization of multi-module projects, forming a foundation for rapid development.

In the context of multi-module Java applications, M. Heckler [3] notes that Spring Boot significantly lowers the entry barrier for building distributed systems. Developers can use pre-configured starter dependencies and switch between modules with a unified configuration model, thereby increasing productivity and reducing errors associated with incorrect library integration or incompatible dependency versions. S. Newman [4], in the broader context of microservices, also highlights similar advantages and recommends Spring Boot as one of the most effective tools for designing high-load systems.

Analyzing practical implementations of microservice patterns (such as API Gateway, Circuit Breaker, Saga), C. Richardson [8] underscores that Spring Boot and related projects (Spring Cloud, Spring Cloud Netflix) enable inter-service interaction and observability without excessive complexity. The design patterns used, coupled with Spring Boot's built-in tools (e.g., Spring Actuator, pre-configured REST controllers, and support for asynchronous data streams), provide the ability to rapidly scale the number of modules without compromising coherence [8].

From an architectural perspective, C. Pautasso and co-authors [5] emphasize in their work the role of fine-grained services, where Spring Boot is well-suited for hosting each module as a standalone unit. According to their research [5], services must support independent deployment and scaling, which, in the case of Spring Boot, is achieved through the combination of a self-contained application package and an auto-configuration mechanism. In other words, each module can use its port, dependencies, and configurations, operating independently of other modules.

B. Sutter and C. Posta [6], in their book on service meshes and Istio, note that Spring Boot services easily integrate into Istio environments, where features such as request tracing, load balancing, and centralized security policies play a critical role. For multi-module architectures, the compatibility of Spring Boot with monitoring systems (Prometheus, Grafana) and logging frameworks (ELK stack) is particularly valuable. Each microservice application can include standard metrics (health checks, usage stats) that are automatically collected and analyzed [6].

The works of P. Raj and S. Patel [7], as well as I. Singh and V. Bhatnagar [9], highlight the importance of containerization (Docker) and orchestration (Kubernetes) when working with Spring Boot (see Fig. 1).

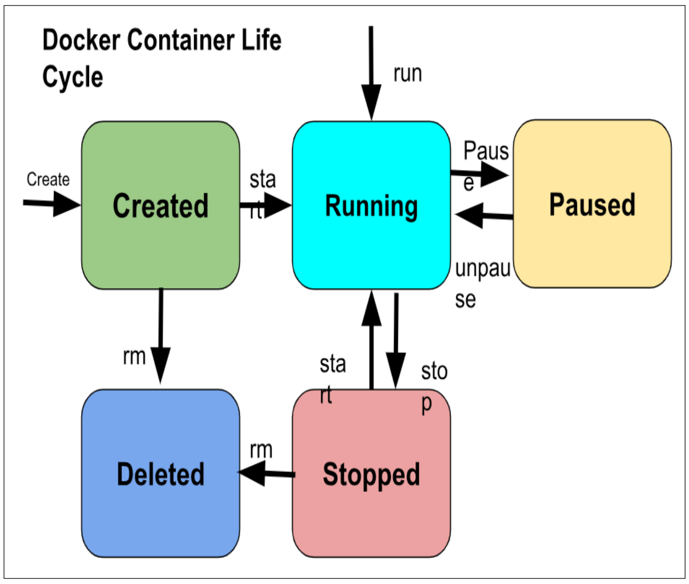

Figure 1. Kubernetes Architecture Diagram [7]

Specifically, discusses the experience of packaging a multi-module application into Docker containers, noting that Spring Boot enables each module to be self-sufficient and requires minimal manual configuration during deployment. Meanwhile, the authors of [9] describe an experiment evaluating the performance of Spring Boot applications in Docker, demonstrating that the efficient distribution of services and their environments (CPU/Memory) within containers ensures a high level of scalability. This approach allows individual modules (services) to scale independently as load increases while maintaining the integrity of the overall system [9].

C. Walls [10] emphasizes that developing multi-module services relies on a set of best practices, including:

- structured storage of source code by modules (domain-driven design, DDD approach);
- the use of Spring Boot starters to standardize configurations;
- organizing CI/CD processes with isolated testing of each module;
- centralized configuration management (Spring Cloud Config) to simplify maintenance.

Thus, a comprehensive analysis of the literature [1–10] indicates that using Spring Boot provides the following benefits:

1. Reducing development time by offering basic configurations, templates (starters), and auto-configuration tools.
2. Ensuring modularity and isolation, where each service (or module) becomes a standalone application that is easy to debug and scale independently of other system components.
3. Simplifying DevOps practices, as integration with containerization (Docker) and orchestration (Kubernetes) environments is facilitated by standard Spring Boot mechanisms for building, metrics, and health checks.
4. Enhancing the manageability of multi-module systems through built-in monitoring (Spring Actuator), request routing patterns (API Gateway), and dynamic configuration tools (Spring Cloud Config).

The following tables (Tables 1–3) summarize part of the systematized data and illustrate key trends in using Spring Boot to simplify the development and maintenance of multi-module services.

Table 1. Classification of Key Issues in Microservice Architecture

Issue	Description	Possible Solution with Spring Boot
Deployment and Configuration	Difficulties in environment setup, multiple configuration files	Spring Boot starters and Spring Cloud Config enable centralized and standardized configuration
Dependency Management	Library version conflicts, code duplication	Auto-configuration and curated dependencies (starters) reduce the likelihood of conflicts
Monitoring and Logging	Multiple services complicate data collection and analysis	Integration of Spring Boot Actuator with monitoring systems (Prometheus, Zipkin, Grafana)
Networking and Security	Access control and data encryption between services	Spring Security and OAuth2 capabilities in Spring Boot simplify the implementation of secure interactions

(Source: compiled by the author based on [2])

This classification provides a general overview of typical barriers faced by organizations transitioning to microservice architecture [2]. The Spring Boot ecosystem addresses these challenges in a comprehensive manner.

Table 2. Deployment Scheme Comparison for Multi-Module Applications

Criterion	Monolith + Manual Configuration	Microservices on Spring Boot + Docker/Kubernetes
Scalability	Vertical, requiring separate environment setup	Horizontal, each microservice can be scaled independently (ReplicaSet, HPA)
Isolation Level	Internal modules are not fully isolated; updating one module can affect others	Full independence of service modules: each service is containerized with its dependencies
Continuous Integration/ Delivery (CI/CD)	Making conflicts possible, it harder to verify the functionality of individual modules	Easier CI/CD pipeline implementation: each microservice is built and tested independently, reducing systemic risk
Configuration Management	Extensive manual setup, large central configuration file	Use of Spring Cloud Config, centralized management of environment variables, and settings passed during container startup
Monitoring	Complex integration, requiring numerous custom solutions	Spring Boot Actuator, built-in health check, and metric endpoints, seamless integration with Istio, Prometheus, etc.

(Source: compiled by the author based on [1], [7])

As illustrated in Table 2, the approach using Spring Boot-based multi-module microservices is particularly advantageous for scaling individual components as load increases, provided DevOps tools are utilized [1], [7].

Table 3. Examples of Common Microservice Patterns and Their Implementation in Spring Boot

Pattern	Description	Implementation in Spring Boot
API Gateway	Centralized entry point, request routing, hiding the internal structure of microservices	Spring Cloud Gateway or Netflix Zuul for routing, filters, and load balancing between services
Circuit Breaker	Isolates failures of dependent services	Spring Cloud Netflix Hystrix (legacy), Resilience4j implemented via annotations, monitored with Spring Boot Actuator
Config Server	Centralized storage of configuration data for multiple services	Spring Cloud Config Server: unified configuration repository (Git), services retrieve settings on startup
Service Registry	Service registration and discovery (Eureka, Consul, ZooKeeper)	Spring Cloud Netflix Eureka Server for service metadata storage, @ EnableEurekaClient for simple microservice integration
Saga	Distributed transaction management to ensure consistency across a series of calls	Implemented through orchestration (e.g., BPMN tools like Camunda) or choreography (event-driven), simplified by Spring Boot through Spring Events

(Source: compiled by the author based on [8], [10])

These patterns [8], [10] are widely used in the development of multi-module microservice systems. Spring Boot and its associated tools help automate many routine tasks related to routing, configuration, and ensuring service resilience.

Summarizing the results of the analysis, the following conclusions can be drawn:

1. Spring Boot offers a comprehensive set of solutions to meet the core needs of a microservice project, ranging from dependency management (starters) and configuration to debugging and monitoring (Actuator).
2. The use of Spring Boot in multi-module architecture simplifies system decomposition into services through streamlined module configuration and support for best practices (REST, health checks, integration with logging, and monitoring systems).
3. With Spring Boot Actuator and relevant libraries (Resilience4j, Hystrix), developers can implement mechanisms for service health monitoring, rapid recovery, and error notifications in multi-module systems.
4. Packaging microservices as fat jars and containers allows each module to be integrated into the build pipeline independently while maintaining a unified configuration concept through Spring Boot starter dependencies.

In summary, the analysis indicates that Spring Boot is not merely an auxiliary tool in microservice architecture but serves as a fundamental engine driving the development of multi-module distributed systems [3], [4], [8]. Auto-configuration, a unified development model (convention over configuration), and a broad ecosystem of extensions (Spring Cloud Netflix, Spring Cloud Config, Spring Security) reduce barriers in the design and maintenance of microservice applications. These solutions contribute to faster time-to-market and improved product quality through better service isolation, ease of testing, and centralized lifecycle support for services.

DISCUSSION

The analysis identified three key vectors of Spring Boot's influence on the development and maintenance of distributed systems: standardization, automation, and ecosystem integration.

First, standardization is achieved through the use of Spring Boot starters and the auto-configuration mechanism. According to N. Alshuqayran et al. [2], unifying dependencies simplifies the launch of each new module and enhances component consistency within the system. This is particularly relevant when the number of microservices grows dynamically, making manual configuration increasingly challenging and resource-intensive. This approach aligns with the principles outlined by S. Newman [4], who notes that reducing boilerplate code directly accelerates the design and implementation phases of services.

Second, automation of configuration, deployment, and monitoring processes allows developers to focus on business logic. As noted by M. Abbadi and M. Debnath [1], Spring Boot Actuator provides detailed metrics on module health, while systematic use of Spring Cloud Config simplifies environment parameter management. This approach reduces risks associated with human error and promotes the broader adoption of DevOps practices, such as CI/CD and testing at every stage. These findings are consistent with the conclusions of I. Singh and V. Bhatnagar [9], demonstrated that containerizing Spring Boot services simplifies the transition from development to production by formalizing build and configuration steps.

Third, Spring Boot's ecosystem integration (e.g., Spring Cloud, Istio, Kubernetes) offers solutions for scaling multi-module applications [6], [7]. Service meshes, as described by C. Posta and B. Sutter [6], enable detailed management of network interactions and platform-level security, while Spring Boot provides hooks for integrating with these service meshes. This combination reduces the complexity of monitoring multi-module systems and enables more agile responses to failures.

However, several sources [2], [5], [10] emphasize that Spring Boot alone does not guarantee optimal system decomposition. While it simplifies certain aspects, such as configuration, startup, and deployment, architectural decisions regarding service partitioning (e.g., domain-driven design, transactional boundaries, and inter-module communication strategies) remain the responsibility of architects and developers. Additionally, challenges may arise as the number of services grows rapidly, requiring a standardized approach to dependency management and testing to maintain a consistent technology stack and library versions.

It is also noted that adopting Spring Boot in combination with microservice architecture necessitates revisiting the team's organizational structure (e.g., DevOps culture, responsibility distribution), as reflected in systematic reviews [2], [5]. In other words, even with the high level of abstraction provided by Spring Boot, a thoughtful approach to orchestrating numerous services and their life cycles is essential.

The analysis indicates that the successful implementation of Spring Boot in multi-module projects depends on both technological and organizational factors. Promising research directions include studying (1) optimal methods for implementing SAGA, Circuit Breaker, and API Gateway patterns in high-load systems using Spring Boot, (2) further improvement of auto-configuration mechanisms in alignment with modern DevOps practices, and (3) the impact of service mesh architectures on the performance and resilience of microservices.

CONCLUSION

Spring Boot significantly simplifies the development process for multi-module services and facilitates a faster transition to microservice architecture. The framework addresses key challenges related to dependency management, configuration, monitoring, and scaling of distributed applications. Through pre-configured starters, an auto-configuration mechanism, and seamless integration with containerization (Docker) and orchestration (Kubernetes) ecosystems, developers can introduce new features more quickly while minimizing system downtime.

The most notable advantages of Spring Boot include the reduction in development time achieved through standard configurations and pre-configured modules (starters). It simplifies testing and debugging processes by offering fat jar packaging and built-in health check endpoints. Additionally, the framework ensures flexible scaling by providing complete service isolation and enabling the rapid replication of containers. Spring Boot also enhances centralized configuration management through Spring Cloud Config and offers ready-to-use security solutions, such as Spring Security and OAuth2. Furthermore, it supports DevOps practices by integrating seamlessly with CI/CD systems, containers, and service meshes.

However, Spring Boot does not eliminate the complexities associated with designing microservice architecture itself. Proper decomposition into services, defining context boundaries, and aligning data requirements remain the responsibility of architects and developers. Nevertheless, the available mechanisms (Spring Cloud Config, Spring Cloud Netflix, Spring Boot Actuator, and others) help lower technical barriers and accelerate implementation.

Thus, Spring Boot proves to be an effective tool for transitioning from monolithic to microservice applications. Its use is particularly relevant in dynamic business environments where there is a need to scale functionality without compromising system stability. Future research directions in this area include developing methods for optimal integration of Spring Boot with service mesh architectures, as well as exploring automated configuration generation and the adoption of new resilience patterns.

REFERENCES

1. Abbadi, M., Debnath, M. A Cloud-Native Approach to Microservices Implementation using Kubernetes and Spring Boot: A Case Study // 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT). – IEEE, 2020. – pp. 1–6.
2. Alshuqayran, N., Ali, N., Evans, R. A Systematic Mapping Study in Microservice Architecture // 2016 9th International Conference on Quality of Information and Communications Technology (QUATIC). – IEEE, 2016. – pp. 267–271. – DOI: 10.1109/QUATIC.2016.071.
3. Heckler, M. Spring Boot: Up and Running: Building Cloud Native Java and Kotlin Applications. – O'Reilly Media, 2021. – URL: https://www.oreilly.com/library/view/spring-boot-up/9781492076957/ (accessed: 18.01.2025).
4. Newman, S. Building Microservices: Designing Fine-Grained Systems. – 2nd ed. – O'Reilly Media, 2019. – URL: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/ (date of access: 18.01.2025).
5. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N. Microservices in Practice, Part 1: Reality Check and Service Design // IEEE Software. – 2017. – T. 34, No. 1. – P. 91–98. – DOI: 10.1109/MS.2017.24.
6. Posta, C., Sutter, B. (eds.). Istio: Up and Running. – O'Reilly Media, 2022. – URL: https://www.oreilly.com/library/view/istio-up-and/9781492043775/ (access date: 01/18/2025).
7. Mhatre, Anand. (2023). Microservices Architecture Using Docker and Kubernetes. International Journal For Multidisciplinary Research. 5.
8. Richardson, C. Microservices Patterns: With Examples in Java. – Manning Publications, 2018. – URL: https://www.manning.com/books/microservices-patterns (access date: 01/18/2025).
9. Singh, I., Bhatnagar, V. Evaluating the Performance of Java-based Microservices using Spring Boot on Docker // 2019 2nd International Conference on Intelligent Communication and Computational Techniques (ICCT). – IEEE, 2019. – pp. 213–217. – DOI: 10.1109/ICCV.2019.00095.
10. Walls, C. Spring Boot in Action. – Reprint. 2019. – Manning Publications, 2016. – URL: https://www.manning.com/books/spring-boot-in-action (accessed: 18.01.2025).