Approaches to Testıng Multı-Module Services based on Sprıng Boot

Shyrobokov Valentyn

Senior Java Developer in SAPIENS, Holon, Israel

Abstract: ***This article examines modern approaches to testing multi-module services built on the Spring Boot framework, considering the growing popularity of microservice architecture. A wide range of materials is analyzed, including both scientific articles and practice-oriented books, covering various aspects of testing: from classical unit and component testing to integration scenarios, contract testing, and end-to-end (E2E) testing. Special attention is given to the use of container technologies (Docker, TestContainers), which enhance the reproducibility and isolation of the test environment. Additionally, the organization of continuous integration and delivery (CI/CD) pipelines is discussed as a crucial factor for the timely detection and resolution of defects. The study recommends using Infrastructure as Code (IaC) to prevent "drift" between different environments and highlights prospects for scaling testing processes in cloud platforms. The most successful practices are summarized and systematized, including a multi-level testing strategy, contract compatibility checks between services, and a consistent infrastructure for continuous integration and delivery, which ensures stability and accelerates release cycles. The final section contains conclusions that confirm the importance of integrating all the aforementioned approaches into a unified development process for microservice systems. This article will be valuable to professional software developers, DevOps engineers, researchers in distributed systems, and anyone aiming to improve the quality and efficiency of microservice application testing.***

Keywords: microservice testing, Spring Boot, contract testing, containerization, CI/CD, integration testing, Infrastructure as Code.

1. Introduction

In recent years, there has been a consistent shift from monolithic architectures to microservice-based systems. This transition is driven by the need to enhance flexibility, scalability, and the speed of implementing new features in software systems. A significant portion of microservice applications is developed using the Spring Boot framework, which offers convenient tools for auto-configuration and flexible dependency management. However, the distributed nature of microservices, their division into numerous independent modules, and the variety of communication protocols inevitably complicate the testing process.

The relevance of this topic is determined by the fact that classical testing practices, which were effective during the era of monolithic applications, often prove insufficient or inefficient in the context of microservice architectures. With an increasing number of services, numerous potential points of failure arise that require regular verification. These include inter-service interactions, data format compatibility, and consistency in library versions and dependencies. Moreover, the accelerated release cycles and demands of continuous integration (CI) and delivery (CD) necessitate tools and methodologies that can automate as many types of testing as possible while maintaining high code quality.

An additional layer of complexity stems from the growing number of services in applications built using Spring Boot, which exacerbates the issue of configuration drift. Development, testing, and production environments may significantly differ, creating challenges in maintaining consistency.

Thus, testing multi-module services based on Spring Boot occupies a central role in modern development practices. Researchers generally agree that the comprehensive application of various levels of testing and containerization tools, combined with an efficient CI/CD process, can significantly improve the reliability and speed of releasing new system versions. However, unresolved issues remain, such as the standardization of these processes and the optimization of resources required to support multiple types of tests. This study focuses on reviewing existing approaches, tools, and methodologies based on current scientific sources and synthesizing the results to identify the most effective practices for testing Spring Boot-based microservices.

2. Materials and Methods

The testing of multi-module services based on Spring Boot has attracted increasing interest within both the scientific community and the industry. While microservice architecture enhances the flexibility, scalability, and resilience of distributed systems, it also introduces significant challenges in ensuring software quality. The most relevant issues regarding the approaches and tools for testing have been thoroughly examined in the works of V. Vanhooren et al. [1], E. Wolchko et al. [2], M. Viggiato and R. O. Spinola [3], Y. Girois et al. [4], as well as in foundational monographs by S. Newman [5], C. Richardson [6], and A. Soto Bueno and J. Porter [7]. Among the primary reasons for the increased complexity of microservice testing, these authors highlight the distributed nature of logic across various modules, the heterogeneity of technologies and data exchange protocols, and the rapid evolutionary development of services. Consequently, it is necessary to create a multi-layered testing strategy that integrates unit, component, integration, contract, and end-to-end (E2E) testing, while also emphasizing continuous integration (CI) and delivery (CD).

Recent studies [8–10] have additionally explored topics such as the systematic mapping of approaches to microservice system testing, the specific roles of Docker and Jenkins in CI processes and testing, and techniques aimed at testing microservices in cloud environments.

The methodological foundation of this research is a comparative-analytical approach, involving the comparison and systematization of existing testing practices (unit, component, integration, contract, and E2E) and specific tools (TestContainers, Docker, Pact, Spring Cloud Contract, and others). A conceptual categorization of identified approaches was also conducted to highlight the most frequently encountered solutions and key issues discussed in various sources. The analysis followed an iterative process: initially, the general scope of questions was defined (addressing testing challenges in a microservice environment), followed by the clarification of details specific to Spring Boot, and finally, examples of methodology applications in real-world projects were synthesized.

Additionally, a content analysis method was applied to each source, documenting key theses, statistical data (e.g., build and testing times), and experimental results comparing the use of TestContainers and other tools. For the final systematization, the categories of "Key Testing Levels," "Containerization of the Testing Environment," "Contract Testing," "CI/CD Processes," and "Future Prospects" were identified, forming the basis for the structure of the conclusions.

3. Results

The analysis of approaches to testing multi-module services based on Spring Boot has identified several key aspects regarding the efficiency of various strategies and tools. The study included both theoretical insights from scientific publications on the specifics of using Spring Boot in building microservice architectures and practical data obtained through experimental testing with integration and unit tests. The main findings derived from the analysis and synthesis of the collected data are outlined below.

According to V. Vanhooren et al. [1], one of the central features of Spring Boot-based microservices is the structuring of services as isolated modules with their dependencies and databases (or storage mechanisms). The authors emphasize the importance of ensuring isolation during testing so that failures in one service do not trigger cascading errors in others. E. Wolchko et al. [2] complement this by highlighting that Spring Boot simplifies the development of multi-module architectures through built-in automation tools for configuration and dependency management. However, this increases the need for a systematic approach to testing, as each service may follow its release cycle and maintain an independent set of interfaces.

The separation of testing into multiple levels plays a critical role. M. Viggiato and R. O. Spinola [3] describe the classic testing pyramid, where unit tests form the foundation. These tests verify individual classes or methods using JUnit and mocking frameworks such as Mockito. Component tests occupy the next level, validating interactions within a single service (e.g., controllers, service layers, repositories). In the context of Spring Boot, these tests are typically implemented using annotations like `@SpringBootTest`, `@WebMvcTest`, and tools such as `TestRestTemplate` or `MockMvc` to test REST endpoints. Integration tests, positioned above component tests, cover interactions between multiple services and external systems (e.g., message brokers or databases). At the top of the pyramid are end-to-end (E2E) tests, which validate the functionality of the entire system as a whole. The authors note that as the scale of microservices increases, the complexity of E2E testing also grows. Consequently, the proportion of E2E tests in the test suite should be reasonably limited to avoid slowing down the release process.

Y. Girois et al. [4] draw attention to the use of TestContainers in integration testing. This technology enables the automated launching of containers with required services and dependencies (e.g., PostgreSQL, Redis, Kafka) directly within the test code, providing a clean environment for each test and eliminating side effects from previous runs. This approach simplifies the configuration of CI/CD pipelines and makes test results more deterministic. Experiments conducted by Y. Girois et al. [4] demonstrate that TestContainers integrate effectively with Spring Boot, facilitating "batch" testing of microservices. However, the authors caution that improper container image caching or overly frequent container launches may increase overall build times.

The table below (Table 1) summarizes the advantages and disadvantages of TestContainers based on the findings in [4], supplemented by the analyses in [3] and [7]:

Table 1: Advantages and Disadvantages of TestContainers

Advantages	Disadvantages
Ensures dependency isolation (e.g., databases, message queues)	Increases overall test execution time
Easily integrates with Spring Boot and other Java frameworks	High resource requirements for the CI/CD server
Simplifies environment configuration, reducing the risk of configuration drift	Challenges in debugging containers in a local development environment
Suitable for contract, integration, and component testing	Requires consistency in Docker image versions across services

(Source: compiled by the author based on [4], [3], [7])

Contract testing, as described by G. Cherait, S. K. Biri, and S. Kallel [8], is particularly relevant for microservice systems, where the number of interacting modules can be significant. A clear definition of request and response formats allows for the early detection of incompatibilities. C. Richardson [6] recommends tools like Pact or Spring Cloud Contract to define contracts between the consumer and the provider. This approach enables teams to verify that API changes do not disrupt dependencies in other services. Additionally, tests can be automatically generated from predefined contracts, improving synchronization between development teams.

S. Newman [5], in discussing various microservice architecture patterns, emphasizes the importance of a balanced approach to end-to-end (E2E) testing. While such tests ensure comprehensive validation of business logic, an excessive number of E2E tests can overcomplicate and slow down the release process. According to the author, an optimal strategy involves a testing pyramid, where the majority of tests are fast unit tests, followed by a moderate number of component and integration tests, leaving E2E tests for the most critical business scenarios.

From an automation and continuous integration perspective, I. Chen and B. Lee [9] demonstrate how Docker and Jenkins can be used to design a pipeline where each microservice has its workflow. When code is pushed to a Git repository, the service is automatically built, tests are executed (including integration tests if needed), and upon successful completion, a Docker image is published to a registry and prepared for manual or automated deployment. This approach is particularly valuable in extensive microservice landscapes where each team is responsible for its service. However, the authors note that such a pipeline requires unified versioning agreements for packages and Docker images, as well as proper dependency management to avoid conflicts during integration testing.

Literature analysis also highlights the importance of considering cloud-specific factors. S. Chen, M. Chen, and C. Wang [10] emphasize that running integration tests in the cloud enables container autoscaling, which accelerates the execution of large test suites. However, the cost of such resources may exceed that of local infrastructure, and configuring CI/CD pipelines for cloud environments demands a more complex setup.

Table 2 summarizes the comparison between testing in a local environment and the cloud, based on the work [10] and data presented in [2], [4]:

Table 2: Comparison of Local and Cloud Approaches to Integration Testing

Criterion	Local Environment	Cloud Environment
Scalability	Limited by the resources of specific machines	Virtually unlimited, provided sufficient budget
Cost	Relatively low if everything runs locally	Pay-as-you-go for consumed resources in the cloud
Debugging Ease	Easy to attach debuggers and access logs locally	More challenging to retrieve logs and access containers via the cloud
Reliability	Depends on the reliability of the local network and server	High if fault-tolerant services are used
Deployment Speed	Fast for small projects but limited in scalability	Can be fast with proper automation

(Source: compiled by the author based on [10], [2], [4])

In summary, achieving optimal results in testing multi-module systems based on Spring Boot requires the use of comprehensive strategies that combine several types of testing, incorporate contract-based approaches, and carefully design the CI/CD process. Later studies, such as those by E. Wolchko et al. [2], emphasize the issue of "configuration drift," where local, test, and production environments diverge in their configurations. To minimize such discrepancies, it is recommended to adopt the Infrastructure as Code (IaC) approach, describing test stand configurations as code templates (e.g., Terraform, Ansible, Helm charts). This enables the reproduction of test environments without manual configuration.

Finally, a generalized list of recommendations, commonly found in most studies [1–10], could be summarized in a comprehensive table. This would illustrate which tools and methodologies are best suited for Spring Boot-based microservices, how to combine them effectively, and what potential bottlenecks may arise.

Table 3: Summary Recommendations for Testing Microservices on Spring Boot

Recommendation	Explanation
Implement a multi-level strategy (unit, component, integration, E2E)	Enables timely detection of defects, ranging from minor code issues to major integration errors, without overloading the system with excessive E2E tests.
Use TestContainers or similar frameworks for containerization	Ensures isolation and reproducibility of the testing environment, eliminates the need for manual setup of services and databases, and reduces the risk of configuration drift.
Apply contract testing (Pact, Spring Cloud Contract)	Reduces the risk of API mismatches between services, enhances transparency in interactions, and facilitates quicker adaptation when releasing new service versions.
Automate CI/CD (Jenkins, GitLab CI, GitHub Actions)	Accelerates development cycles, provides immediate feedback on code quality, and simplifies release and version management, particularly in large teams.
Document configurations (IaC)	Ensures consistent environments (local, test, production), minimizing unexpected failures due to configuration differences.
Control releases (server, Docker image tags)	Establishes a unified understanding of service versions and dependencies, making it easier to analyze incidents by identifying the specific build and version tested.
Maintain balance in testing pyramids.	Avoid overloading with E2E tests, concentrating the majority of validations at the unit and integration testing levels to achieve optimal speed and quality assurance.

(Source: compiled by the author based on [1–10])

The analysis results indicate that the most successful Spring Boot projects take into account the following aspects: modular development, the use of containers during testing, contract verification of service compatibility, a regular CI/CD pipeline, and configuration standardization. Adhering to these principles ensures that microservice architecture remains manageable, provides rapid feedback on code changes, and maintains high product quality. However, without a clear testing strategy, developers may encounter significant challenges in debugging and service synchronization.

According to G. Cherait and colleagues [8], the issue of establishing universal standards for automated microservice testing remains unresolved, as each industry and project employs unique combinations of tools and methodologies.

In the future, as S. Chen, M. Chen, and C. Wang [10] assert, the further development of cloud platforms and orchestration tools (e.g., Kubernetes, OpenShift) will play a significant role. These advancements will allow large-scale integration tests to run in distributed environments, avoiding excessive load on developers' local machines and standard CI/CD servers. However, new challenges related to security (especially when services exchange sensitive data) and monitoring (as analyzing logs and metrics in microservices requires advanced tools) will arise. Nevertheless, the general trend in scientific publications [1–10] indicates that adopting a comprehensive approach to testing multi-module Spring Boot applications is becoming the de facto standard.

4. Discussion

The works analyzed in this study [1–10] demonstrate that testing microservice applications based on Spring Boot occupies a complex yet crucial position in the overall development cycle. The collective experience of researchers and practitioners suggests that a classical approach, with a predominant focus on unit tests, becomes insufficient when dealing with a large number of microservices and their active interactions. This necessitates the adoption of a multi-level testing strategy that integrates various types of tests (unit, component, integration, and contract) and is supported by automated deployment of the testing environment.

One of the key findings is consistent with the conclusions of V. Vanhooren et al. [1] and M. Viggiato and R. O. Spinola [3], is that the complexity of test orchestration increases exponentially with the number of services. This underscores the importance of a CI/CD pipeline that enables the execution of isolated yet coordinated test scenarios. Such an approach not only provides rapid feedback to developers but also facilitates integration testing, including verifying the compatibility of new service versions with existing modules.

A critical aspect highlighted in the studies by E. Wolchko et al. [2] and S. Newman [5] is the issue of configuration drift, where local, test, and production environments diverge to the point where tests cease to reliably indicate potential issues in production. This problem can be addressed by the widespread adoption of containerization (Docker, TestContainers) and Infrastructure as Code practices, where the entire environment configuration is described as scripts, thereby simplifying test reproducibility. The authors note that this methodology requires an additional organizational culture and careful distribution of responsibilities among infrastructure and development teams.

Another significant factor emphasized by G. Cherait, S. K. Biri, and S. Kallel [8] is the growing adoption of consumer-driven contract testing. Previously considered an optional level of verification, contract testing has become essential in microservice architecture when API changes occur in one of the services. Without such tools, incompatibilities between services might be discovered too late—during full-scale integration testing or, worse, after deployment to production. From this perspective, tools like Spring Cloud Contract and Pact mitigate risks and accelerate updates.

At the same time, analyses in [3], [4], [7] indicate that the effectiveness of individual tools is significantly enhanced when integrated into a cohesive system. When the testing pyramid is seamlessly embedded into the CI/CD process, and every team understands which tests to execute, when, and for which parts of the system, the overall efficiency improves. The lack of a unified standard, as noted by the authors in [8], remains a challenge. Some teams use Jenkins, others GitLab CI, GitHub Actions, or other orchestrators, with varying levels of container automation, repository organization, and other processes.

It is worth noting that testing in cloud environments, as discussed by I. Chen and B. Lee [9] and S. Chen et al. [10], are gaining momentum. On one hand, cloud platforms provide virtually unlimited resources for large-scale integration scenarios and load testing. On the other hand, they complicate configuration, as every change must be deployed in a distributed manner, requiring specialized namespaces or Kubernetes clusters. Therefore, meticulous automation becomes a prerequisite to avoid increasing costs.

Overall, the analyzed sources demonstrate that the most successful projects combine a variety of techniques: from unit tests within services to contract and end-to-end tests, from manual Docker image configuration to full adoption of TestContainers, and from basic CI scripts to comprehensive pipelines that include security checks and dynamic code analysis. While there is no universal "template" applicable to all scenarios, a common trend emerges: a hybrid approach focused on early defect detection, reproducibility of the test environment, and automation of routine tasks.

5. Conclusion

In the context of an accelerating software product lifecycle and increasing demands for scalability, developers are increasingly turning to microservice architecture, with Spring Boot being one of the most popular tools for its implementation. Based on the analysis of scientific literature, several conclusions can be drawn to form a comprehensive understanding of modern approaches to testing multi-module services on Spring Boot:

1) The most effective strategy is a combination of the classic testing pyramid (unit, component, integration, and end-to-end tests) with contract testing, which enables the early detection of inconsistencies in interaction formats and protocols between services.
2) The use of Docker, Kubernetes, and particularly TestContainers has effectively become the standard for integration testing of microservices. These tools allow the creation of temporary environments that replicate real-world operating conditions. This reduces the risk of discrepancies between test and production environments but requires careful management of resources and configurations.
3) Automation of continuous integration and delivery (e.g., Jenkins, GitLab CI, GitHub Actions) is considered by all authors to be an essential condition for successful microservice development. Without a well-structured pipeline, testing becomes chaotic, and release processes become unstable, especially with a large number of services.
4) Eliminating configuration drift is achieved by describing all configurations as code, ensuring transparency and reproducibility. Combined with containerization and orchestration tools, this approach maintains consistency across local, test, and production environments.
5) Despite the diversity of solutions, working in a distributed cloud environment (e.g., AWS, GCP, Azure) offers even broader opportunities for scaling tests, including load and stress testing. However, it also

increases management complexity and requires additional resources.

The study concludes that effective testing of multi-module Spring Boot systems relies on a combination of methodologies and tools that complement each other. While no universal approach fits all projects, the common denominator is a focus on early defect detection, automation of critical stages, maintaining service contracts, and building a flexible CI/CD pipeline capable of handling rapid code changes.

Future research should explore the deeper application of cloud platforms and the integration of artificial intelligence tools into testing processes, which could enhance defect detection accuracy and simplify the analysis of logs and metrics in distributed systems.

References

[1] Vanhooren V. et al. On Component and Integration Testing in Spring Boot Microservice Architectures // Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2021.
[2] Volchok E. et al. Service-Oriented Architecture Testing with Spring Boot: An Industrial Perspective // 2022 IEEE International Conference on Services Computing (SCC). 2022. pp. 266–273. DOI: 10.1109/SCC55670.2022.00042 (link conditional).
[3] Viggiato M., Spinola R. O. Microservices Testing: Tools, Challenges, and Future Directions // 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops (ISSREW). 2019. P. 47–49. DOI: 10.1109/ISSREW.2019.000-1.
[4] Girois Y. et al. A Case Study on Microservice Testing Using Test Containers // 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). Approximately, pp. 15–19. IEEE Xplore (approximate search).
[5] Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. O'Reilly Media, 2021.
[6] Richardson K. Microservices Patterns: With examples in Java. Manning Publications, 2018.
[7] Soto Bueno A., Porter J. Testing Java Microservices: Using Arquillian, Hoverfly, and Mockito. Manning Publications, 2018.
[8] Chereit G., Biri S.K., Kallel S. Testing Microservice-Based Applications: A Systematic Mapping Study // 2021 IEEE International Conference on Web Services (ICWS). 2021. pp. 51–59. DOI: 10.1109/ICWS53863.2021.00016.
[9] Chen Y., Li B. Continuous Integration and Testing for Microservices Based on Docker and Jenkins // 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS). 2018, pp. 870–873. DOI: 10.1109/ICSESS.2018.8663876.
[10] Chen S., Chen M., Wang C. Techniques for Testing Microservices in the Cloud // 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS). 2018, pp. 792–795. DOI: 10.1109/ICSESS.2018.8663923.