


[REFCARD UPDATE] Java Performance Optimization: Patterns and Anti-Patterns

[Read Now](#) ▶

DZone > Microservices Zone > Microservices Using Spring Boot and Spring Cloud — Part 1: Overview

Microservices Using Spring Boot and Spring Cloud — Part 1: Overview

by Siva Prasad Reddy Katamreddy  MVB · Mar. 04, 18 · Microservices Zone · Tutorial

Microservices is the hot buzzword in software development and many organizations prefer building their enterprise applications using microservices architecture. In the Java community, Spring Boot is the most widely used framework for building both monoliths and microservices. I am planning to write a series of articles covering how to build microservices using Spring Boot and Spring Cloud.

In this article, we are going to learn about the following:

- Monoliths
- What are microservices?
- Advantages of microservices
- Challenges with microservices
- Why Spring Boot and Spring Cloud are a good choice for microservices
- Introducing the application

Traditionally, we are building large enterprise applications in a modularized fashion (!) but finally deploy them together as a single deployment unit (EAR or WAR). These are called monolithic applications.

There are some issues with the monolithic architecture, such as

- Large codebases become a mess over time.
- Multiple teams working on a single codebase becomes tedious.
- It is not possible to scale up only certain parts of the application.
- Technology updates/rewrites become complex and expensive tasks.

However, in my opinion, it is relatively easy to deploy and monitor monoliths compared to microservices.

A microservice is a service built around a specific business capability which can be independently deployed. So, to build large enterprise applications, we can identify the sub-domains of our main business domain and build each sub-domain as a microservice using Domain Driven Design (DDD) techniques. But in the end, we need to make all these microservices work together to serve the end user as if it is a single application.

Advantages of Microservices

- Comprehending a smaller codebase is easy.
- You can independently scale up highly used services.
- Each team can focus on one (or a few) microservice(s).
- Technology updates/rewrites become simpler.

Challenges With Microservices

- Getting the correct sub-domain boundaries, in the beginning, is hard.
- You need more skilled developers to handle distributed application complexities.
- Managing microservices-based applications without the proper DevOps culture is next to impossible.
- A local developer environment setup might become complex to test cross-service communications, though by using Docker/Kubernetes, this can be mitigated to some extent.

Spring Boot is the most popular and widely-used Java framework for building MicroServices. These days, many organizations prefer to deploy their applications in a Cloud environment instead of the headache of maintaining a datacenter themselves. But, we need to take good care of the various aspects to make our applications Cloud Native. There comes the beauty of Spring Cloud.

Spring Cloud is essentially an implementation of various design patterns to be followed while building Cloud Native applications. Instead of reinventing the wheel, we can simply take advantage of various Spring Cloud modules and focus on our main business problem rather than worrying about infrastructural concerns.

Following are just a few Spring Cloud modules that can be used to address distributed application concerns:

- **Spring Cloud Config Server:** Used to externalize the configuration of applications in a

central config server with the ability to update the configuration values without requiring to restart the applications. We can use Spring Cloud Config Server with **git**, **Consul**, or **ZooKeeper** as a config repository.

- **Service Registry and Discovery:** As there could be many services and we need the ability to scale up or down dynamically, we need a Service Registry and Discovery mechanism so that service-to-service communication does not depend on hard-coded hostnames and port numbers. Spring Cloud provides **Netflix Eureka**-based Service Registry and Discovery support with just minimal configuration. We can also use **Consul** or **ZooKeeper** for Service Registry and Discovery.
- **Circuit Breaker:** In microservices-based architecture, one service might depend on another service, and if one service goes down, then failures may cascade to other services as well. Spring Cloud provides a Netflix Hystrix-based Circuit Breaker to handle these kinds of issues.
- **Spring Cloud Data Streams:** We may need to work with huge volumes of data streams using **Kafka** or **Spark**. Spring Cloud Data Streams provides higher-level abstractions to use those frameworks more easily.
- **Spring Cloud Security:** Some microservices need to be accessible to authenticated users only, and most likely, we'll want a **Single Sign-On** feature to propagate the authentication context across services. Spring Cloud Security provides authentication services using OAuth2.
- **Distributed Tracing:** One of the pain points with microservices is the ability to debug issues. One simple end-user action might trigger a chain of microservice calls; there should be a mechanism to trace the related call chains. We can use **Spring Cloud Sleuth with Zipkin** to trace cross-service invocations.
- **Spring Cloud Contract:** There is a high chance that separate teams will work on different microservices. There should be a mechanism for teams to agree upon API endpoint contracts so that each team can develop their APIs independently. **Spring Cloud Contract** helps to create such contracts and validate them by both the service provider and consumer.

These are just a few of Spring Cloud's features. To explore more, visit <https://projects.spring.io/spring-cloud/>.

I strongly believe in learning by example, so let us learn how to build microservices using

I strongly believe in learning by example, so let us learn how to build microservices using Spring Boot and Spring Cloud with a sample application. I will deliberately keep the application business logic very simple so that we focus on understanding the Spring Boot and Spring Cloud features.

We are going to build a simple shopping cart application and assume we are going to start with the following microservices:

- **catalog-service:** It provides a REST API to provide catalog information like products.
- **inventory-service:** It provides a REST API to manage product inventory.
- **cart-service:** It provides a REST API to hold the customer cart details.
- **order-service:** It provides a REST API to manage orders.
- **customer-service:** It provides a REST API to manage customer information.
- **shoppingcart-ui:** It is a customer-facing front-end web application.

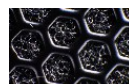
We are going to build various services and REST endpoints as we go through various microservice concepts.

Stay tuned for the next post, where we are going to create a catalog-service and use the spring-cloud-config server to have a centralized configuration for all our microservices.

Like This Article? Read More From DZone



DZone Article
Deploy a Spring Boot Microservice Architecture to Google Cloud and Google Kubernetes Engine



DZone Article
Microservices Architecture With Spring Boot and Spring Cloud



DZone Article
Microservices With Spring Boot, Part 1 — Getting Started



Free DZone Refcard
Designing Microservices With Cassandra

Topics: JAVA, MICROSERVICES, SPRING BOOT, SPRING CLOUD, TUTORIAL

Published at DZone with permission of Siva Prasad Reddy Katamreddy , DZone MVB. [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.