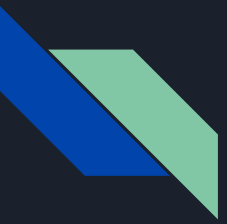
# Building Cloud-Native Java Application in Microservice architecture and managing with Kubernetes

**- Abdullah Al Mamun Oronno**
https://www.linkedin.com/in/oronno
Email: mr.oronno@gmail.com



# Topics

Building **Cloud-Native** Java Application in **Microservice architecture** and managing with **Kubernetes**

# Microservice architecture

# What are microservices?

“Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.”

- Martin Fowler
https://martinfowler.com/articles/microservices.html

# Advantages of microservices

- Fault isolation
- Smaller and faster deployments
- Scalability
- Eliminate vendor or technology lock-in
- Support for Two-Pizza Development Teams

# Example: **Ride hailing application like Uber/Grab**

- Authentication and User Profile Service
- Location Service
- Booking Service
- Pricing Service
- Payment Service
- History Service
- Promotion

# Disadvantages of Microservices

- Communication between services can be complex
- Debugging becomes more challenging with microservices
- While unit testing may be easier with microservices, integration testing is not
- Microservice API need to be consistent and backward compatible; if breaking changes needed, need to support older version for a period of time
- Up-front costs is higher with microservices

# Cloud Native Application

# Defining cloud native

“Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

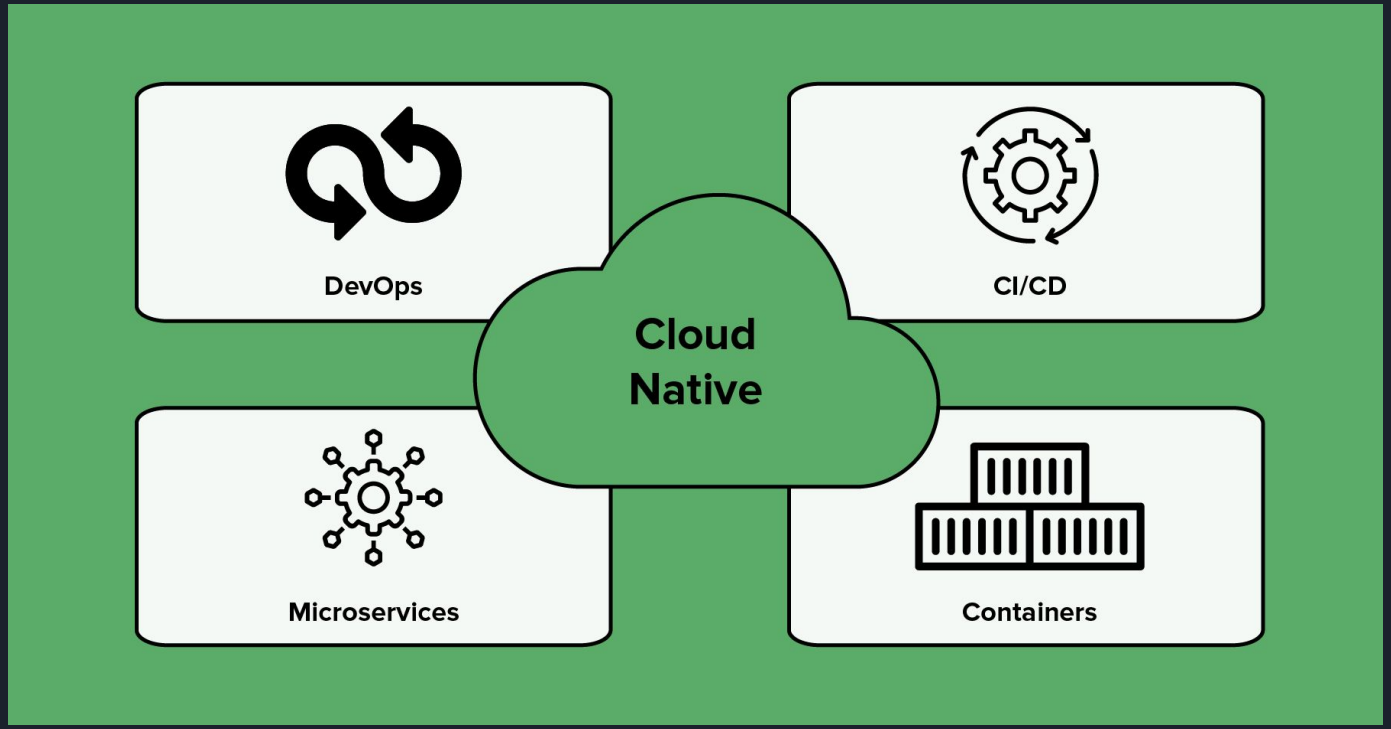
These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.”

- Cloud Native Computing Foundation (CNCF)
  https://github.com/cncf/foundation/blob/master/charter.md

# Why cloud native

| Company | Experience |
| --- | --- |
| Netflix | Has 600+ services in production. Deploys hundred times per day. |
| Uber | Has 1,000+ services in production. Deploys several thousand times each week. |
| WeChat | Has 3,000+ services in production. Deploys 1,000 times a day. |

# Major aspects of Cloud Native Architecture



# Aligning with the cloud native patterns

- Aligning with the microservices patterns
- Using containerization
- Following declarative communication pattern
- Deploying container orchestration
- Continuous Integration and Delivery (CI & CD)
- Exposing health check
- Collecting telemetry data
- Writing code according to **12-factor application principles**

https://12factor.net

# The Twelve Factors

**I. Codebase**
One codebase tracked in revision control, many deploys

**II. Dependencies**
Explicitly declare and isolate dependencies

**III. Config**
Store config in the environment

**IV. Backing services**
Treat backing services as attached resources

**V. Build, release, run**
Strictly separate build and run stages

**VI. Processes**
Execute the app as one or more stateless processes

**VII. Port binding**
Export services via port binding

**VIII. Concurrency**
Scale out via the process model

**IX. Disposability**
Maximize robustness with fast startup and graceful shutdown

**X. Dev/prod parity**
Keep development, staging, and production as similar as possible

**XI. Logs**
Treat logs as event streams

**XII. Admin processes**
Run admin/management tasks as one-off processes

# Example of few patterns use with microservices in cloud native application

# Centralized config management

1. Support multiple environments such as dev, test and prod.
2. Transparent config fetching
3. Automatic property refresh when property changes
4. Maintain change history and easily revert to older version

# Service Discovery

The problem seems simple at first: how do clients know the IP and port for a service that could exist on multiple hosts? Things get more complicated as you start deploying more services in versatile cloud environment.

Several aspects should be considered:

- Fault Tolerance
- Load Balancing
- Integration Effort
- Availability Concerns

# Message-driven architecture for Asynchronous Communication

- Avoid synchronous communication if possible
- Scale with ease

# Distributed tracing

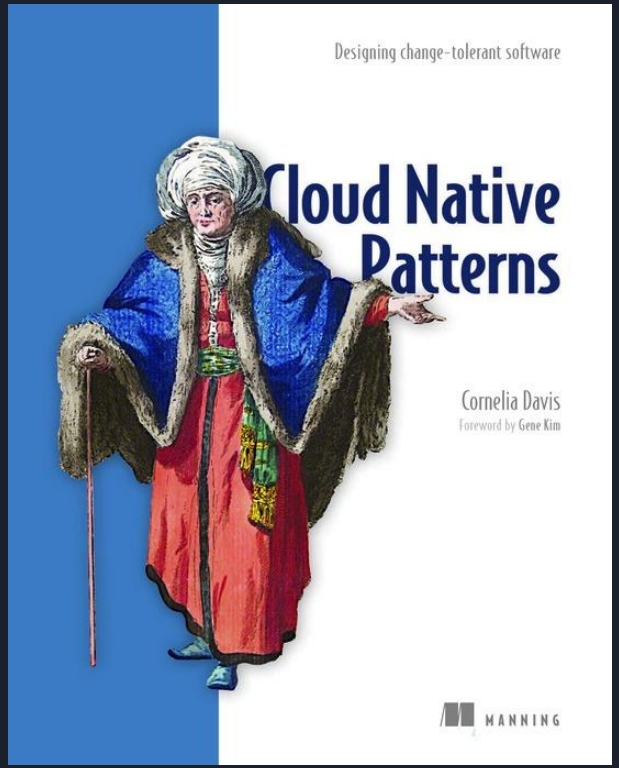
Under microservice architecture, one external request might involve several internal service calls, and these services might spread over many machines. Although most solutions have implemented centralized logging storage and search, it is still hard to trace end-to-end transactions spanning across multiple services. Figuring out how a request travels through the application also means manually searching log keywords many times to find clues. This is really time-consuming and error-prone, especially when you may not have enough understanding of microservice topology. Actually, what we need is to correlate and aggregate these logs in one place.

# Circuit Breaker

- Save your resource if other service down
- Don’t overwhelm when other service recovering
- Fallback logic: from cache or internal logic

# And many more patterns.....

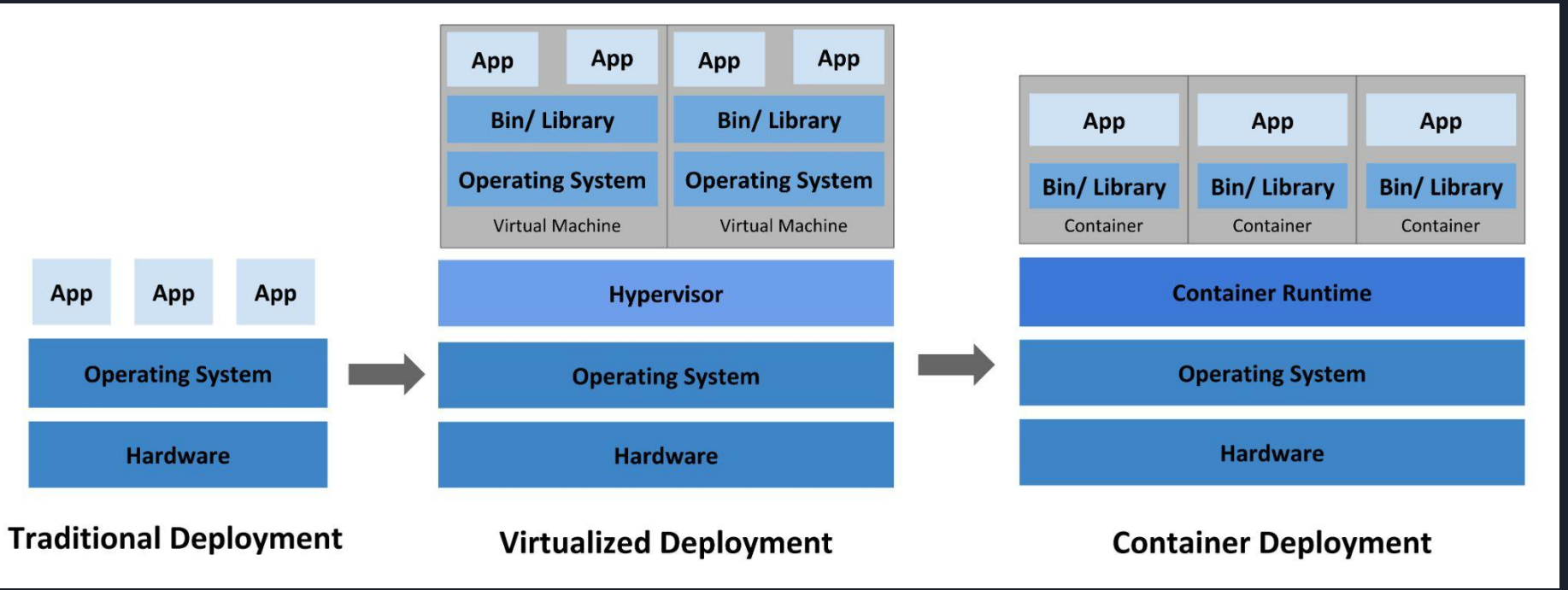
# Read to learn cloud native patterns...



# What is containers?

Containers are executable units of software in which application code is packaged, along with its libraries and dependencies, in common ways so that it can be run anywhere, whether it be on desktop, traditional IT, or the cloud. To do this, containers take advantage of a form of operating system (OS) virtualization in which features of the OS are leveraged to both isolate processes and control the amount of CPU, memory, and disk that those processes have access to.

# Deployment: Traditional vs VM vs Container



# Docker

Docker is a software platform for building applications based on containers. While containers have been used in Linux and Unix systems for some time, Docker, an open source project launched in 2013, helped popularize the technology by making it easier than ever for developers to package their software to “build once and run anywhere.”

Docker enhanced the native Linux containerization capabilities with technologies that enable:

- Improved and seamless portability - Docker containers run without modification across any desktop, data center and cloud environment
- Automated container creation: Docker can automatically build a container based on application source code.
- Container versioning: Docker can track versions of a container image, roll back to previous versions, and trace who built a version and how. It can even upload only the deltas between an existing version and a new one.
- Container reuse: Existing containers can be used as base images—essentially like templates for building new containers.
- Shared container libraries: Developers can access an open-source registry containing thousands of user-contributed containers.

# Container Orchestration

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system? This where container orchestration comes in.

Container orchestration automates the scheduling, deployment, networking, scaling, health monitoring, and management of containers.

# What is Kubernetes

Kubernetes is the most popular container orchestration tool that allows to deploy and manage multi-container applications at scale. While in practice Kubernetes is most often used with Docker, it can also work with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes. Google open sourced Kubernetes in 2014 makes it easy to run applications in the cloud.

# Benefits of Kubernetes

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- Automatic bin packing
- Self-healing
- Secret and configuration management

# Develop Cloud Native Java Application

# What to use?

There are a lots of options. Each one has its own pros and cons. Do your own research before choosing any stack.

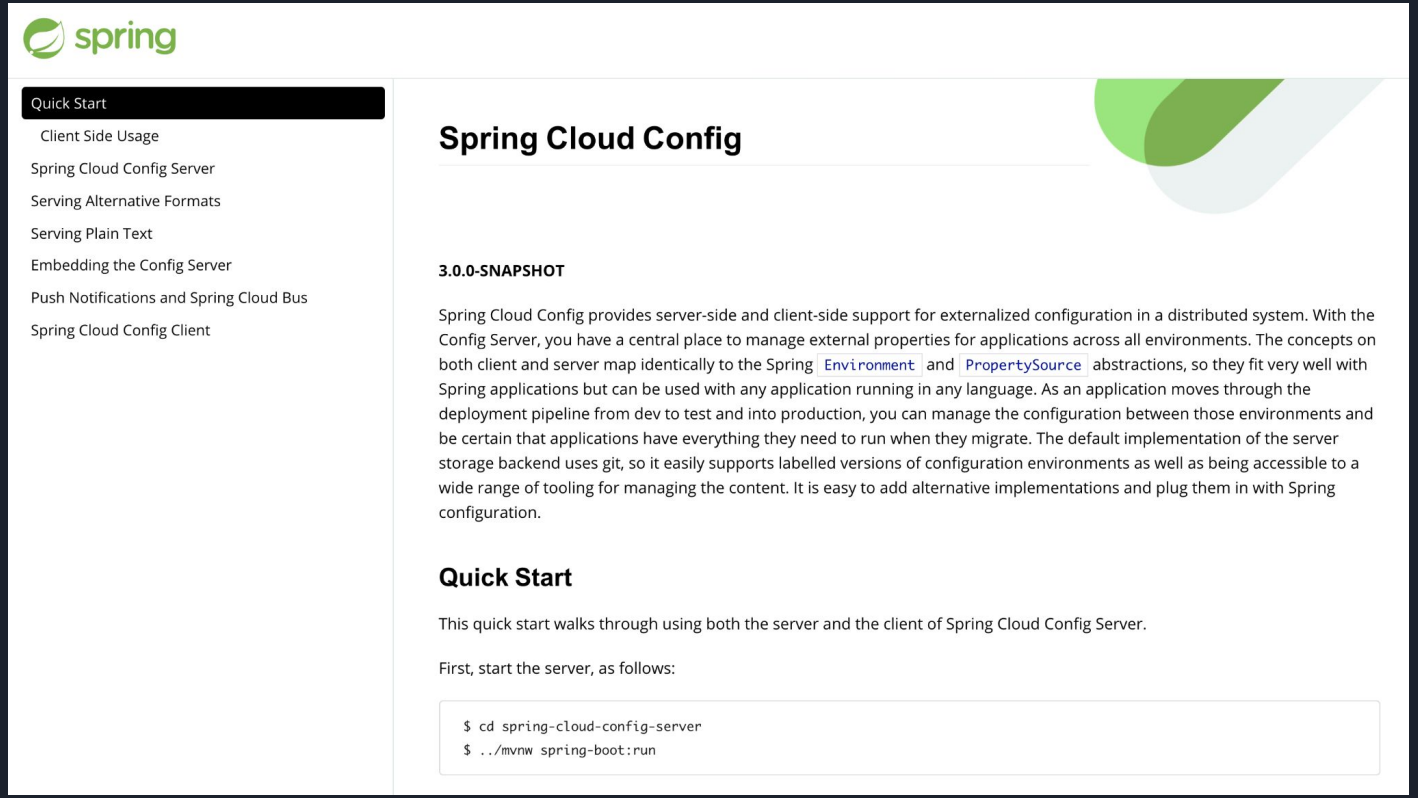
Example recommendation:

RESTful api -> Spring Boot

Build Docker Container with Java App -> Spotify’s dockerfile-maven-plugin | Gradle Jib plugin

Cloud native patterns -> Spring Cloud (check next slides for some example)

# Centralized config management with **Spring Cloud Config**



# Service Discovery via **Spring Cloud Discovery**



## 2. Spring Cloud Commons: Common Abstractions

Patterns such as service discovery, load balancing, and circuit breakers lend themselves to a common abstraction layer that can be consumed by all Spring Cloud clients, independent of the implementation (for example, discovery with Eureka or Consul).

### 2.1 @EnableDiscoveryClient

Spring Cloud Commons provides the `@EnableDiscoveryClient` annotation. This looks for implementations of the `DiscoveryClient` interface with `META-INF/spring.factories`. Implementations of the Discovery Client add a configuration class to `spring.factories` under the `org.springframework.cloud.client.discovery.EnableDiscoveryClient` key. Examples of `DiscoveryClient` implementations include Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery, and Spring Cloud Zookeeper Discovery.

By default, implementations of `DiscoveryClient` auto-register the local Spring Boot server with the remote discovery server. This behavior can be disabled by setting `autoRegister=false` in `@EnableDiscoveryClient`.

`@EnableDiscoveryClient` is no longer required. You can put a `DiscoveryClient` implementation on the classpath to cause the Spring Boot application to register with the service discovery server.

# Message-driven architecture with **Spring Cloud Stream**

spring

## Spring Cloud Stream Reference Documentation

Sabby Anandan · Marius Bogoevici · Eric Bottard · Mark Fisher · Ilayaperumal Gopinathan · Gunnar Hillert · Mark Pollack · Patrick Peralta · Glenn Renfro · Thomas Risberg · Dave Syer · David Turanski · Janne Valkealahti · Benjamin Klein · Vinicius Carvalho · Gary Russell · Oleg Zhurakousky · Jay Bryant · Soby Chacko · Domenico Sibilio

**3.0.10.RELEASE**

The reference documentation consists of the following sections:

| | |
|---|---|
| **Overview** | History, Quick Start, Concepts, Architecture Overview, Binder Abstraction, and Core Features |
| **Rabbit MQ Binder** | Spring Cloud Stream binder reference for Rabbit MQ |
| **Apache Kafka Binder** | Spring Cloud Stream binder reference for Apache Kafka |
| **Apache Kafka Streams Binder** | Spring Cloud Stream binder reference for Apache Kafka Streams |
| **Additional Binders** | A collection of Partner maintained binder implementations for Spring Cloud Stream (e.g., Azure Event Hubs, Google PubSub, Solace PubSub+) |
| **Spring Cloud Stream Samples** | A curated collection of repeatable Spring Cloud Stream samples to walk through the features |

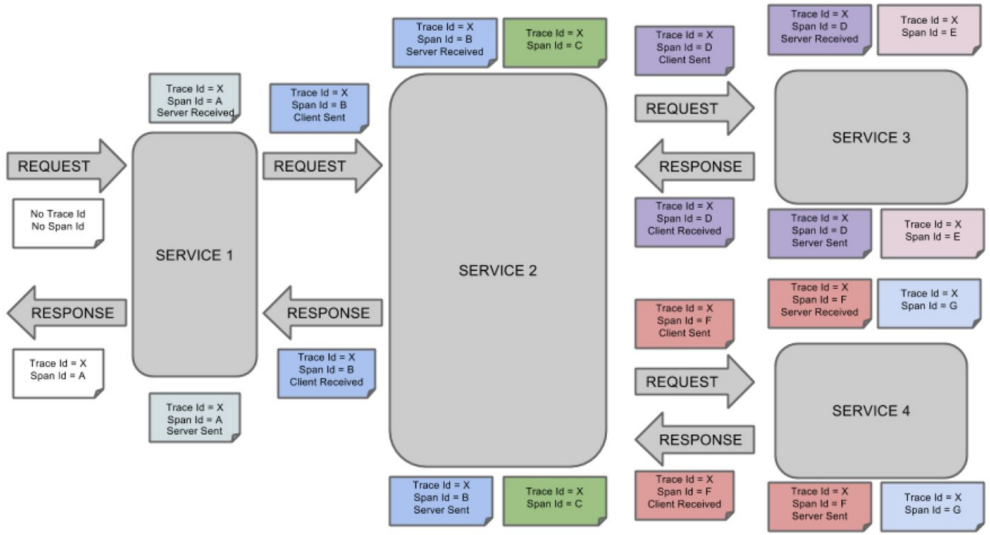
# Distributed tracing via Spring Cloud Sleuth and Zipkin

Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud.

## 1.1 Terminology

Spring Cloud Sleuth borrows Dapper's terminology.

The following image shows how **Span** and **Trace** look in a system, together with the Zipkin annotations:



# Circuit Breaker with Spring Cloud support

and many more....

# Spring Cloud 2020.0.3



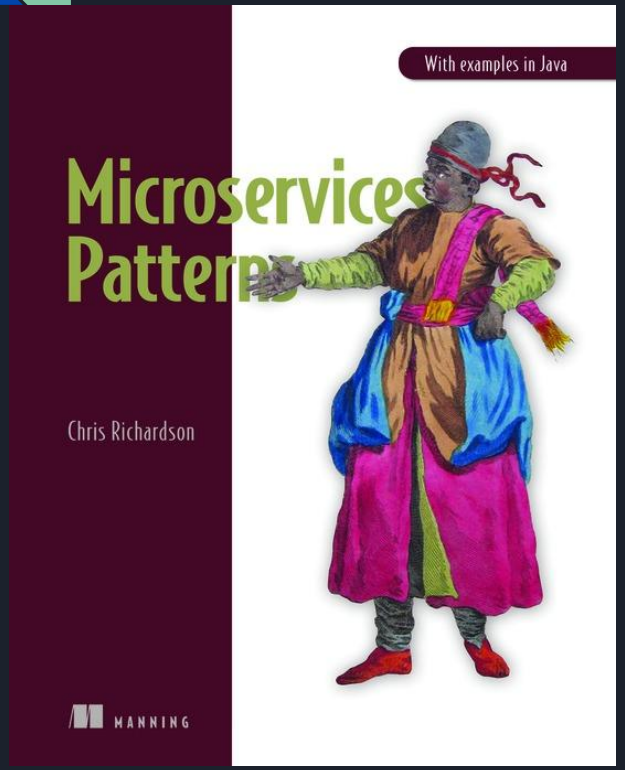
Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

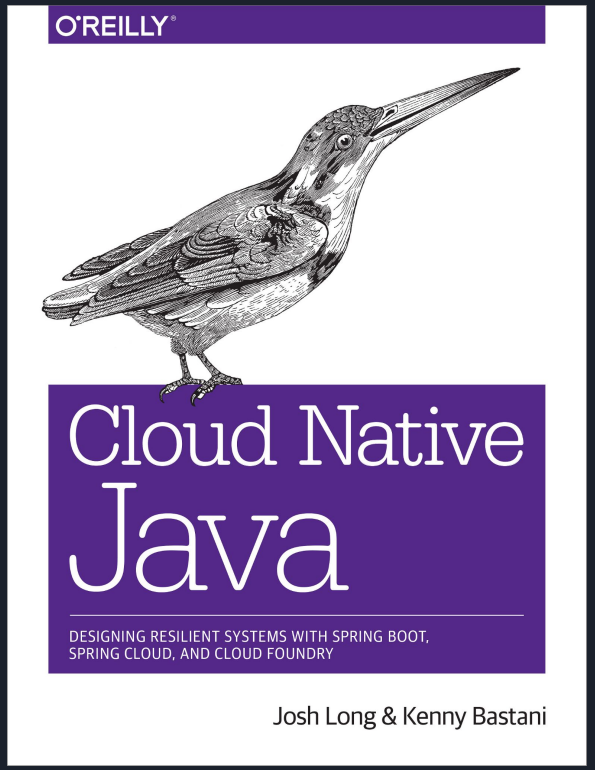
## Features

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers

# Book Recommendation



# Reference links:


- https://spring.io/projects/spring-cloud
- https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
- https://www.bmc.com/blogs/cloud-native-devops/#
- https://www.ibm.com/cloud/learn/docker
- https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition
- https://medium.com/walmartglobaltech/cloud-native-application-architecture-a84ddf378f82
- https://tanzu.vmware.com/cloud-native
- https://www.nginx.com/learn/microservices
- https://www.infoq.com/articles/spring-cloud-azure/