Microservices Introduction

A microservice is a small, loosely coupled distributed service. Microservice Architectures evolved as a solution to the scalability and innovation challenges with Monolith architectures (Monolith applications are typically huge – more 100, 000 line of code). It allows you to take a large application and decompose or break into easily manageable small components with narrowly defined responsibilities.

Reasons for using Microservice:

In monolith application, there are few challenges:

For a large application, it is difficult to understand the complexity and make code changes fast and correctly, sometimes it becomes hard to manage the code.

Applications need extensive manual testing to ensure the impact of changes.

For small change, the whole application needs to be built and deployed.

The heavy application slows down start-up time.

Benefits of Microservices:

Small Modules –

Application is broken into smaller modules which are easy for developers to code and maintain.

Easier Process Adaption –

By using microservices, new Technology & Process Adaption becomes easier. You can try new technologies with the newer microservices that we use.

Independent scaling –

Each microservice can scale independently via X-axis scaling (cloning with more CPU or memory) and Z-axis scaling (sharding), based upon their needs.

Unaffected –

Large applications remain largely unaffected by the failure of a single module.

DURS –

Each service can be independently DURS (deployed, updated, replaced, and scaled).

Restrictions of Microservices:

Configuration Management –

As it becomes granular the headache comes for configuring the services and monitoring those. You need to maintain configurations for hundreds of components across environments.

Debugging –

Tracking down the service failure is painstaking job. You might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.

Automation – Because there are a number of smaller components instead of a monolith, you need to automate everything – Builds, Deployment, Monitoring etc.

Testing –

Needs a greater effort for end to end testing as it needs all the dependent services to be up and running.

Microservice Frameworks for Java:

There are several microservices frameworks that you can use for developing for Java. Some of these are:

Spring Boot –

This is probably the best Java microservices framework that works on top of languages for Inversion of Control, Aspect Oriented Programming, and others.

Dropwizard –

Dropwizard pulls together stable, mature libraries from the Java ecosystem into a simple, light-weight package that lets you focus on getting things done.

Restlet –

Restlet Framework helps Java developers build better web APIs that follow the REST architecture style.

Spark –

A micro-framework for creating web applications in Kotlin and Java 8 with minimal effort.

Others that you can consider include: Ninja Web Framework, Play Framework, RestExpress and Restx Framework.

Restrictions of Microservices:

Configuration Management –

As it becomes granular the headache comes for configuring the services and monitoring those. You need to maintain configurations for hundreds of components across environments.

Debugging –

Tracking down the service failure is painstaking job. You might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.

Automation – Because there are a number of smaller components instead of a monolith, you need to automate everything – Builds, Deployment, Monitoring etc.

Testing –

Needs a greater effort for end to end testing as it needs all the dependent services to be up and running.

Microservice Frameworks for Java:

There are several microservices frameworks that you can use for developing for Java. Some of these are:

Spring Boot –

This is probably the best Java microservices framework that works on top of languages for Inversion of Control, Aspect Oriented Programming, and others.

Dropwizard –

Dropwizard pulls together stable, mature libraries from the Java ecosystem into a simple, light-weight package that lets you focus on getting things done.

Restlet –

Restlet Framework helps Java developers build better web APIs that follow the REST architecture style.

Spark –

A micro-framework for creating web applications in Kotlin and Java 8 with minimal effort.

Others that you can consider include: Ninja Web Framework, Play Framework, RestExpress and Restx Framework.

Microservices in Java Spring Boot, Microservices in Java, How to build microservices in java ?, Eureka, Resilience4j, Feign Client, Hystrix, Slueth, Zipkin, Spring Boot Admin Dashboard, Spring Boot Actuator, Java Microservices, Microservices in Java Tutorial etc.

Microservices In JavaWhen a new architecture for developing an application comes into the market, it makes Solution Architects more busy in finding out which language to use to fulfill customer’s need. No surprise!, we are just talking about Microservices Architecture that we have covered at length in another article. However, there are many languages which started using Microservices Architecture. The top five languages that are using Microservices are Java, Python, C++, Ruby and Golang. Although this data is based on the number of companies using it. Again Microservices in Java is leading the table. Needless to say, our focus will only be on Microservices in Java that you are expecting from us to discuss here. Of course, selection of a particular language is all about what works for you and what is relevant to your particular problem.

What is Microservices in Java?

It’s not just the Microservices developed using the Java language. There are still many things behind the scene that need our attention. When it comes to developing Microservices, an essential question is which language to choose. JVM is a notable platform in Java that offers developers a great opportunity to use a different language abroad. The strength of the Java-based Microservice architecture is that it makes easier for developers to analyze with other languages or frameworks also without having a very heavy risk.

As we all know, one extraordinary thing about the Java ecosystem(JVM) is that we write our Java code once, and we can run it fundamentally on any other operating system. But there is a condition that you didn’t compile your code with a newer Java version than your target JVM’s versions. Otherwise, you will face java.lang.UnsupportedClassVersionError at runtime.

Moreover, particularly frameworks such as Spring Boot are well recognized and provide great help in building Microservices. Spring Boot provides you embedded servers such as Tomcat, Jetty, Undertow. It helps you to build .jar files very easily with an embedded web server and you can run it anywhere straight away. You don’t need to set up any server separately to run the application. No matter, if you are developing application for configuration, security, REST API, Batch Processing, mobile or even Big Data, there is a Spring Boot Project for you to make development easier & faster.

What is Spring Boot Microservices ?

Generally, we use the term Spring Boot Microservices where we develop a Microservices based application using Java. Spring Boot in nothing but a framework to develop a ready to run application in Java. Moreover, Spring Boot framework also supports to develop Microservices in Java. Furthermore, we have a variety of starters in Spring Boot that help us to easily develop a microservices based application. Hence, When we develop microservice based applications using Spring Boot, we collectively call it Spring Boot Microservices.

How are Microservices Flexible with Spring Cloud?

Spring Cloud can help developers with service configurations, service discovery, circuit-breaking, load-balancing, distributed tracing & logging, and monitoring. It can even sometimes act as an API gateway. Further, let’s discuss about the terminologies that we use in the development of Microservices including the java microservices tools.

What is Service Registration & Discovery?

When you have multiple services running together within an application, they need to detect each other to communicate. This the first and the most important feature of Microservices. In order to make it possible, there should exist one medium where all microservices register themselves. Subsequently, when any service wants to communicate, it can connect to that medium and discovers other service to communicate. This medium is nothing but ‘Service Registry & Discovery’ which is itself a microservice. Moreover, this is similar to Java’s RMI mechanism, where you work with a central registry so that RMI processes could find each other. Microservices has the same obligation.

Netflix Eureka

For this purpose, we use Netflix Eureka to make a kind of server which can register the microservices & discover them when required by the other microservice. We call it Eureka Server or Discovery server. Every Microservice will register into the Eureka server with a service Id and Eureka server will have information(port, IP addresses etc.) of all the microservices running as client applications. Further, in order to implement it we create a microservice using Spring Boot project and include the below dependency mandatorily. A service (Spring Boot Application in our case) annotated with @EnableEurekaServer will work as Eureka server.

<dependency>

<!-- Eureka for service registration -->

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

Additionally, you can visit our detailed article on ‘How To Register & Discover Microservices Using Netflix Eureka?‘.

What is Microservices Intra-communication?

A published microservice communicates with the other microservice with the help of Eureka Server itself. We use one Client to get details of a microservice from Eureka Server. A service(Spring Boot Application in our case) annotated with @EnableEurekaClient will work as a Eureka Client application. These possible clients can be :

1)DiscoveryClient (Legacy client) : This is a basic client. It supports fetching Service Instances from Eureka server based on Service Id as List type. Developer has to choose one instance with less load factor manually (no auto-balance concept supported) from the list of Service Instances.

2)LoadBalancerClient (new client) : It will fetch only one Service Instance from Eureka based on Service Id that has less load factor. LoadBalancerClient is an interface. Its implementation is provided by ‘Spring Cloud Netflix-Ribbon‘ i.e. RibbonLoadBalancerClient(C)

3)FeignClient/Open Feign (Abstract client) : We also call it as Abstract Client or Declarative Rest Client. FeignClient is an interface, at runtime one class gets generated using Dynamic Proxy Pattern. This concept offers two annotations : @EnableFeignClients at starter class and define interface for one Consumer with annotation @FeignClient(name=”ServiceId”).

DiscoveryClient and LoadBalancerClient communicate with Eureka to get Service Instance details, but they don’t support HTTP calls. But, Feign Client acts as a combination client. In fact, it gets Service Instance from Eureka Server, and supports making the HTTP call.

By using above clients we can get Service Instance details of microservice from Eureka Server using Service Id as input. Once we get Service Instance data, then we use HTTP Client: RestTemplate (C) to make an HTTP Request for service provider application. IP and PORT numbers may get changed based on the System and Deployment Instance count. So, we should read these details from Eureka Server using Service Instance.

How to enable Load Balancer?

Although the previous versions of Spring Boot were using the ‘Ribbon’ to enable the load-balancing feature. However, we can get the load balancing feature enabled if we are using ‘Feign Client’ & ‘Eureka’. Below is the dependency to include ‘Feign Client’.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

What is API Gateway?

API Gateway is the single entry & exit point of all the microservices in the application. Since every microservice has its own IP & port, and we can’t provide multiple IP & port details to the client / end user, there must be a single entry and exit point. It is also a type of microservice, that calls all other microservices using Eureka and it should also be registered with the Eureka Server like other microservices. It generates a class(proxy) based on the service Id provided with the path(URL) using a load balancer client. Then it selects one Service Instance from Eureka and makes the HTTP call. This is obviously required because Eureka Server itself can’t communicate with any microservice.

Eureka Server is only to register and discover microservices. One microservice can communicate with another with the help of Eureka Server only. Eureka never supports making HTTP call to any microservice. API Gateway helps in implementing Security, applying filters, SSO(Single Sing On), dynamic routing etc.

Zuul Proxy Server

In order to implement API Gateway, we can use Zuul Proxy Server that handles all the requests and does the dynamic routing of microservices. Dynamic routing is nothing but choosing one microservice instance and make HTTP call based on the load. We also sometimes call it as Zuul Server or Edge Server. We add the @EnableZuulProxy annotation on our main application class to make our Spring Boot application act as a Zuul Proxy server.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

Note : Newer versions of Spring Boot suggests us to use Spring Cloud Gateway in place of Zuul.

Spring Cloud Gateway

Spring Cloud Gateway is a simple, yet an effective way to route to APIs. It also offers implementation of various cross cutting concerns such as Security, Logging, Monitoring/metrics etc. It is built on top of Spring Webflux (A reactive programming approach), We need to add the dependency as given below in order to get features of Spring Cloud Gateway.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

Some important features of the Spring Cloud Gateway are:

1) Match routes on any request attribute

2) Define Predicates & Filters

3) Integrates with Spring Cloud Discovery Client (Load Balancing)

4) Path Rewritting

What is Circuit Breaker?

If the actual method of a microservice is continuously throwing exception, then stop executing actual method and redirect each request to a fallback method, such concept is called Circuit Breaker. In this situation, we configure a dummy method that will execute and give a response back to the client such as ‘Service Not Working’, ‘Unable to Process request at this time’, ‘try after some time’…etc. We call such a dummy method as Fallback method. There are two types of circuit : open circuit & closed circuit. In Open Circuit, client request directly moves to Fallback method. But In closed Circuit, client request moves to the actual service method only.

Spring Cloud Hystrix

In order to implement Circuit Breaker mechanism, we use Spring cloud Hystrix. We add the @EnableHystrix annotation on our main application class to make our Spring Boot application act as a Circuit Breaker. In addition, @HystrixCommand(fallbackMethod = “DUMMY METHOD NAME”) at RestController method level. Additionally, you can visit our detailed article on ‘How To Implement Hystrix Circuit Breaker In Microservices Application?‘.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

What is Fault Tolerance in Microservices?

In order to implement complete Fault Tolerance, even including circuit breaker, we use Resilience4j API. It has multiple separate modules such as Rate Limiter, Time Limiter, Bulkhead, Circuit Breaker, Retry etc. We have separate annotations for each functionality as @RateLimiter, @TimeLimiter, @Bulkhead, @CircuitBreaker, @Retry respectively. However, we can also implement these functionalities programmatically by using Decorator pattern. Moreover, we require below dependencies to implement Resilience4j. Additionally, you can visit our detailed article on ‘How To Implement Fault Tolerance In Microservices Using Resilience4j?‘.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

What is Spring Cloud Config Server?

In short, we also call it Config server. If we have same ‘key=value’ properties in every microservice, then we can define all of them in one common properties file outside of all microservices projects. In order to do it, we create one microservice which will have this common properties file and call it as Config Server. That common properties file will have association with every microservice using Config Server. Common ‘key=value’ properties are like, DB Connection, Email, Security etc. However, we can handle it in two ways : External Config Server and Native Config Server. We can implement External Config Server by using GitHub, GitLab, Bitbucket etc. We can implement Native Config Server just by using local drives of our system which is appropriate only for development environment.

Moreover, we add the @EnableConfigServer annotation on our main application class to make our Spring Boot application act as a Config Server.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

What is Distributed Tracing & Logging?

In real time one application can have multiple microservices. Further, one request can involve multiple microservices till the completion of the request. Then the manual tracing of all microservices involved in a request becomes a lengthy task. Here, tracing is the process of finding an execution path or flow of multiple microservices involved in serving a request. Also, Logging is not going to be a simple process as each microservice will have its own log file. Therefore, implementation of a Distributed Tracing & Logging mechanism becomes mandatory. When it implemented one time, developers need not to check code again and again for the flow of a request whether it is testing purpose, debug purpose, or for the development. It provides execution order of microservices and related log lines.

Sleuth

In order to implement Distributed Tracing & Logging, Spring Cloud API offers two cloud components : Sleuth & Zipkin. Sleuth provides unique IDs for request flows. The developer uses this ID to find the execution flow of a request. There are two types of IDs : Trace ID and Span ID. Trace ID is a unique Id is for a complete flow (from Request till Response). Using this ID, developer can find out logs of all microservices involved in the flow. Span Id is a unique ID for one microservice flow. Using this ID, developer can find out log messages for a particular microservice.

Zipkin

Zipkin works in a client server model. In every microservice, we should also add this dependency along with Sleuth. It contains Sampler (Collect data from microservice using Sleuth and provide to Zipkin Server). There must be only one centralized Zipkin Server that collects all data from Zipkin Client and displays it as a UI. Now, Developer should make a request and goto Zipkin Server to find Trace ID, Span ID and flow as well. Then the developer should open Log files to see Log lines related to current Trace Id.

Additionally, you can visit our detailed article on ‘How To Implement Distributed Logging & Tracing Using Sleuth & Zipkin?‘.

ELK Stack

ELK Stack (Elasticsearch, Logstash, Kibana) is one of the most popular tools to monitor our application via log analysis. Further, to get a deeper view on it, kindly visit our article on ‘How To Monitor Spring Boot Microservices Using ELK Stack?‘.

What is Spring Boot Admin Dashboard?

Spring Boot Admin is a web application that manages and monitors multiple Spring boot applications(microservices in our case) and shows the results in the form of a single dashboard. Generally, microservices developers use it for monitoring of webservices. If we add Spring Actuator to the Spring Boot applications, we can get multiple endpoints to monitor and deal with Spring Boot applications. Each Spring Boot application acts as a client and registers to the Spring Boot Admin Server. Spring Boot Actuator endpoints provides the magic behind the scene. We apply Actuator to Spring Boot Admin Client Applications and expect the results in Spring Boot Admin Server in the form of a dashboard.

Moreover, we add the @EnableAdminServer annotation on our main application class to make our Spring Boot application act as a Spring Boot Admin Server.

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-server</artifactId>

</dependency>

Spring Boot Actuator

Spring Boot Actuator provides endpoints for managing and monitoring of your Spring Boot application. All actuator endpoints secure by default. Endpoints are nothing but details to access a web service such as path(/emp/data), http method(GET), input(String), output(JSON) etc. In order to enable Spring Boot Actuator in your application, you need to add the Spring Boot Starter Actuator dependency in your pim.xml file as given below.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

Some of the popular & important actuator endpoints are as given below. You can enter them in your web browser and monitor the behavior of your application.

env : in order to know the environment variables used in the application.

beans : in order to view the Spring beans and its types, scopes and dependency used in the application.

health : in order to view the application health. Moreover, this endpoint says ‘IS OUR APPLICATION STARTED PROPERLY OR NOT?’.Additionally, we can get the data like Memory for Disk Space, PING STATUS etc.

info : in order to get information of current microservice to other Clients/Users/Dev etc.

trace : in order to view the list of traces of your Rest endpoints.

metrics : in order to view the application metrics such as memory used, free memory, classes, threads, system uptime etc.