Spring Cloud Tutorial



# What is Spring Cloud?

Spring Cloud is a framework for building robust cloud applications. Spring Cloud provides a solution to the commonly encountered patterns when developing a distributed system.

# Why is Spring Cloud used?

Spring Cloud framework provides tools for developers to build a robust cloud application quickly. We can also build the microservice-based applications, for example, **configuration management, service discovery, circuit breakers, intelligent routing, cluster state, micro-proxy, a control bus, one time tokens, etc**. Using Spring Cloud, a developer can quickly develop services and applications that implement the design patterns. These patterns work well in any distributed environment, including the **bear metal data centers, developer's laptop,** and managed platform such as **Cloud Foundry**.

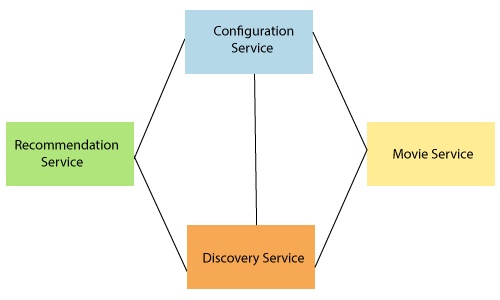
# Features of Spring Cloud

Spring cloud is built upon some of the common building blocks of Spring framework which are as follows:

* Intelligent routing and service discovery
* Service-to-Service Call
* Load Balancing
* Leadership Election
* Global Locks
* Distributed Configuration
* Distributed Messaging

### **Intelligent Routing and Service discovery**

When building a microservices on Spring Cloud, there is a primary concern to deal with the first two primary microservices: **configuration service** and the **discovery service**.



The above figure represents the set of **four microservices**. The connection between each service indicates dependency. All services are dependent on each other. The configuration service lies at the top, and the discovery services at the bottom. There are two microservices in-between which are **Recommendation Service** and **Movie Service**.

### **Service-to-Service calls**

It is the process of "how a microservice communicates with other dependent microservices via service registry or Eureka server." There is a sequence which follows in the service-to-service call.

* Registering the service
* Fetching the Registry
* Finding the downstream service
* Resolving the Underlying IP address
* Call the rest Endpoint

### **Load Balancing**

Load balancing efficiently distributes network traffic to multiple backend servers or server pool. The objective of load balancing is to maximize throughput, minimize response time, increase efficiency, and optimize resource uses. It **avoids overload** of any single resource. Using multiple components with load balancing may increase **reliability** and **availability** through redundancy.

### **Global Locks**

Global locks are used to ensure that no two thread simultaneously access the same resource at the same time. The programmer uses a mechanism to remove such situation, is called a **lock**. Each thread first **cquires the lock, operate on the resources,** and **release the lock** for other thread.

### **Distributed Configuration**

Distributed configuration is to configure every instance of all microservices. "Spring cloud config server" provides client-side support for externalized configuration in a distributed system. With the distributed configurations, we have a central place to manage external properties for applications across all environment.

### **Distributed Messaging**

The distributed messaging system provides the benefit of reliability, scalability, and persistence. The messaging pattern follows the **Publish-Subscribe** (Pub-Sub) model. In the Pub-Sub model, the sender of the message is called publisher and receiver of the message is called subscribers. **Apache Kafka** and **RabbitMQ** are the popular high throughput messaging system.

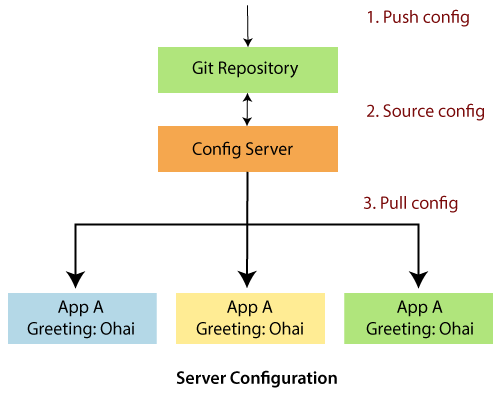
Spring Cloud Components

There are the following components:

* Configuration
* Service Discovery
* Circuit Breakers
* Routing and Messaging
* API Gateway
* Tracing
* CI Pipeline and Testing

## **Configuration**

Spring Cloud configuration components provide server-side and client-side support for externalized configuration in a distributed system. We can manage the external properties with config server for applications across all environments. Spring Cloud config server can use Git, SVN (Apache Subversion), filesystem, and Vault to Store config. Config clients (microservice app) retrieve the configuration client from the server on startup.



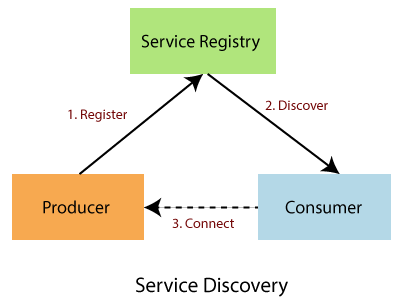
## **Service Discovery**

The service discovery is the automatic detection of devices and services over the network. In other words, service discovery is how an application and microservices connect in the distributed environment. Service discovery implementations include both:

* The **central server** that maintains a global view of the address.
* The **clients** that connect to the central server can update and retrieve the address.

There are **two** discovery patterns: **Client-side discovery** and **Server-side discovery**.

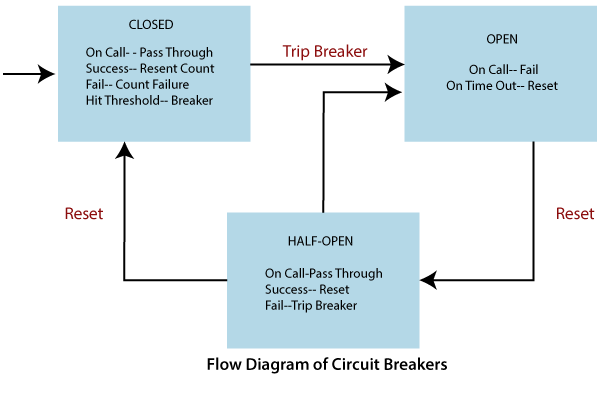
* **Client-side discovery:** In the Client-side discovery, client is responsible for determining the network location of available services. The client uses a **load-balancing algorithm** to select one of the available services and make a request. **Netflix OSS** is an example of a client-side discovery pattern.
* **Server-side discovery:** In the server-side discovery, the client makes an HTTP request to a service through a load balancer. The load balancer contacts to service registry and route each request to an available service instance. Similar to client-side discovery, service instances are registered and deregistered with the service registry. The **AWS ELB** (Elastic Load Balancer) is an example of server-side discovery. ELB balances the external traffic from the internet.



In the above figure producer is a software that sends a message to a message broker (Service Registry). A consumer is also a software that receives the message and processes it.

## **Circuit Breakers**

Netflix has created a library called **Hystrix**. It implements the circuit breakers pattern. Circuit breakers calculate when to open and close the circuit and what to do in case of failure. When all services fail at some point, the circuit breaker handles these failures gracefully. The circuit breakers have three states: **OPEN, CLOSED,** and **HALF-OPEN** State.



## **Routing and Messaging**

The cloud application made up of many microservices so the communication will be critical. Spring Cloud supports communication via messaging or HTTP request. Routing uses **Netflix Ribbon** and **Open Feign while** messaging uses Kafka or Rabbit MQ.

## **API Gateway**

API Gateway allows us to route API request (external or internal) to connect services. It also provides a library for building an API gateway on the top of Spring MVC. Its aims to provide cross-cutting concerns to them, such as **security** and **monitoring**.

## **Tracing**

Spring Cloud's other functionality is **distributed tracing**. Tracing is a single request to get data from the application. Tracing results in an exponentially larger number of requests to various microservices.

We can add **Spring Cloud Sleuth** library in our project to enable tracing. Sleuth is responsible for recording **timing**, which is used for **latency analysis**. We can export this timing to Zipkin.

Zipkin is a distributed tracing tool specially designed for **analyzing latency problem** inside the microservice architecture. It exposes HTTP endpoint used for collecting input data. If we required to add tracing in our project, we should add the **spring-cloud-starter-zipkin** dependency.

In the microservices, the input traffic volume is so high, so we cannot collect an only certain amount of data. For that purpose, the Spring Cloud Sleuth provides a **sampling policy**. The sampling policy allows us how much input traffic is sent to Zipkin for analysis. To enable this feature, we have to add the **spring-cloud-sleuth-stream** dependency.

# Main projects of Spring Cloud

**Spring Cloud OpenFeign:** Spring Cloud OpenFeign is a Java to HTTP binder. It reduces the complexity of binding.

**Spring Cloud AWS:** It is a part of Spring Cloud Umbrella project. It provides easy integration with Amazon Web Services. The developer can build their project around the AWS without having to care about maintenance.

**Spring Cloud Task:** It allows us to develop short-lived microservices using Spring Cloud and run them locally in the cloud or even in Spring Cloud Data Flow.

**Spring Cloud Stream:** It is used for building a highly saleable event-driven microservices.

**Spring Cloud Cluster:** It provides tools for building a cluster feature in a distributed system. For example, global locks and leadership election.

**Spring Cloud Data Flow:** It provides tools to create complex topologies for streaming batch data and pipeline. It supports data processing use cases.

**Spring Cloud Config:** It provides client and server-side support for externalized configuration in a distributed environment. We get a central place to manage external properties for applications across the distributed environment.

**Spring Cloud Netflix:** It provides integration with various Netflix OSS components like Eureka, Zuul, Hystrix, etc.

**Spring Cloud CloudFoundry:** It integrates the application with pivotal cloud foundry. It also provides service discovery and easy implementation of SSO (Single-Sign-On) and OAuth2 (OAuth2 is the method of authenticating access to the API) protected resources. It allows authentication without the external application getting the user email-address or password.

**Spring Cloud CLI:** It provides a command-line feature for sparing cloud. We can launch services like Edureka, Zipkin, Config server conveniently from the CLI.

**Spring Cloud Starter:** Spring Cloud Starters eases the curated set of dependency management for the consumers of Spring Cloud.

**Spring Cloud Bus:** It is a lightweight message broker. It can be used to broadcast state changes or other management instructions.

**Spring Cloud Sleuth:** It implements a distributing tracing solution for spring cloud.

# [Microservices Tutorial](https://www.javatpoint.com/microservices)

**Microservice Architecture** is a Service Oriented Architecture.

In the microservice architecture, there are a large number of **microservices**.

By combining all the microservices, it constructs a big service.

In the microservice architecture, all the services communicate with each other.

In the **Microservices** tutorial, we will understand how to implement microservices using **Spring Cloud**.

We will learn how to establish communication between microservices, **enable** **load balancing**, **scaling up and down of microservices**.

We will also learn to **centralize the configuration of microservices**with **Spring Cloud Config Server**.

We will implement **Eureka Naming Server** and **Distributed tracing** with **Spring Cloud Sleuth** and **Zipkin**.

We will create fault tolerance microservices with **Zipkin**.

Our **microservices** tutorial discusses the basic functionalities of **Microservice Architecture**along with relevant examples for easy understanding.

## **What are Microservices**

**Definition**: "Microservices are the small services that work together."

"The microservice architectural style is an approach to develop a single application as a suite of small services.

Each microservice runs its process and communicates with lightweight mechanisms. These services are built around business capabilities and independently developed by fully automated deployment machinery."

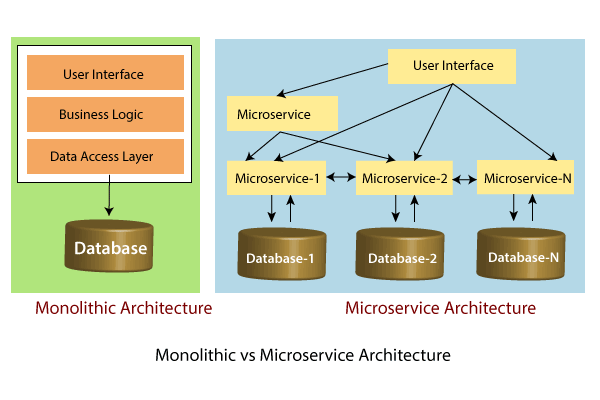
There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.

## **Points to remember**

* These are the services which are exposed by REST.
* These are small well-chosen deployable units.
* The services must be cloud-enabled.

The microservice defines an approach to the architecture that divides an application into a pool of loosely coupled services that implements business requirements.

The most important feature of the microservice-based architecture is that it can perform **continuous delivery** of a large and complex application.



In the above figure, each microservice has its own business layer and database. If we change in one microservice, it does not affect the other services. These services communicate with each other by using lightweight protocols such as HTTP or REST or messaging protocols.

## **Principles of Microservices**

There are the following principles of Microservices:

* Single Responsibility principle
* Modelled around business domain
* Isolate Failure
* Infrastructure automation
* Deploy independently

### **Single Responsibility Principle**

The single responsibility principle states that a class or a module in a program should have only one responsibility. Any microservice cannot serve more than one responsibility, at a time.

### **Modeled around business domain**

Microservice never restrict itself from accepting appropriate technology stack or database. The stack or database is most suitable for solving the business purpose.

### **Isolated Failure**

The large application can remain mostly unaffected by the failure of a single module. It is possible that a service can fail at any time. So, it is important to detect failure quickly, if possible, automatically restore failure.

### **Infrastructure Automation**

The infrastructure automation is the process of scripting environments. With the help of scripting environment, we can apply the same configuration to a single node or thousands of nodes. It is also known as configuration management, scripted infrastructures, and system configuration management.

### **Deploy independently**

Microservices are platform agnostic. It means we can design and deploy them independently without affecting the other services.

# Advantages of Microservices

* Microservices are self-contained, independent deployment module.
* The cost of scaling is comparatively less than the monolithic architecture.
* Microservices are independently manageable services. It can enable more and more services as the need arises. It minimizes the impact on existing service.
* It is possible to change or upgrade each service individually rather than upgrading in the entire application.
* Microservices allows us to develop an application which is organic (an application which latterly upgrades by adding more functions or modules) in nature.
* It enables event streaming technology to enable easy integration in comparison to heavyweight interposes communication.
* Microservices follows the single responsibility principle.
* The demanding service can be deployed on multiple servers to enhance performance.
* Less dependency and easy to test.
* Dynamic scaling.
* Faster release cycle.

## **Disadvantages of Microservices**

* Microservices has all the associated complexities of the distributed system.
* There is a higher chance of failure during communication between different services.
* Difficult to manage a large number of services.
* The developer needs to solve the problem, such as network latency and load balancing.
* Complex testing over a distributed environment.

# Challenges of Microservices Architecture

Microservice architecture is more complex than the legacy system. The microservice environment becomes more complicated because the team has to manage and support many moving parts. Here are some of the top challenges that an organization face in their microservices journey:

* Bounded Context
* Dynamic Scale up and Scale Down
* Monitoring
* Fault Tolerance
* Cyclic dependencies
* DevOps Culture

**Bounded context**: The bounded context concept originated in Domain-Driven Design (DDD) circles. It promotes the Object model first approach to service, defining a data model that service is responsible for and is bound to. A bounded context clarifies, encapsulates, and defines the specific responsibility to the model. It ensures that the domain will not be distracted from the outside. Each model must have a context implicitly defined within a sub-domain, and every context defines boundaries.

In other words, the service owns its data and is responsible for its integrity and mutability. It supports the most important feature of microservices, which is independence and decoupling.

**Dynamic scale up and scale down**: The loads on the different microservices may be at a different instance of the type. As well as auto-scaling up your microservice should auto-scale down. It reduces the cost of the microservices. We can distribute the load dynamically.

Backward Skip 10sPlay VideoForward Skip 10s

**Monitoring**: The traditional way of monitoring will not align well with microservices because we have multiple services making up the same functionality previously supported by a single application. When an error arises in the application, finding the root cause can be challenging.

**Fault Tolerance**: Fault tolerance is the individual service that does not bring down the overall system. The application can operate at a certain degree of satisfaction when the failure occurs. Without fault tolerance, a single failure in the system may cause a total breakdown. The circuit breaker can achieve fault tolerance. The circuit breaker is a pattern that wraps the request to external service and detects when they are faulty. Microservices need to tolerate both internal and external failure.

**Cyclic Dependency**: Dependency management across different services, and its functionality is very important. The cyclic dependency can create a problem, if not identified and resolved promptly.

**DevOps Culture**: Microservices fits perfectly into the DevOps. It provides faster delivery service, visibility across data, and cost-effective data. It can extend their use of containerization switch from Service-Oriented-Architecture (SOA) to Microservice Architecture (MSA).

## **Other challenges of microservices**

* As we add more microservices, we have to be sure they can scale together. More granularity means more moving parts, which increase complexity.
* The traditional logging is ineffective because microservices are stateless, distributed, and independent. The logging must be able to correlate events across several platforms.
* When more services interact with each other, the possibility of failure also increases.

Difference between Microservices Architecture (MSA) and Services-Oriented Architecture (SOA)

|  |  |
| --- | --- |
| **Microservice Based Architecture (MSA)** | **Service-Oriented Architecture (SOA)** |
| Microservices uses **lightweight protocols** such as **REST**, and **HTTP**, etc. | SOA supports **multi-message protocols**. |
| It focuses on **decoupling**. | It focuses on application service **reusability**. |
| It uses a **simple messaging system** for communication. | It uses **Enterprise Service Bus** (ESB) for communication. |
| Microservices follows "**share as little as possible**" architecture approach. | SOA follows "**share as much as possible architecture**" approach. |
| Microservices are much better in **fault tolerance** in comparison to SOA. | SOA is not better in fault tolerance in comparison to MSA. |
| Each microservice have an **independent** database. | SOA services share the **whole** data storage. |
| MSA used **modern** relational databases. | SOA used **traditional** relational databases. |
| MSA tries to **minimize** sharing through bounded context (the coupling of components and its data as a single unit with minimal dependencies). | SOA **enhances** component sharing. |
| It is better suited for the **smaller** and **well portioned**, web-based system. | It is better for a **large** and **complex** business application environment. |

# Microservices Monitoring

Monitoring is the control system of the microservices. As the microservices are more complex and harder to understand its performance and troubleshoot the problems. Given the vivid changes to software delivery, it is required to monitor the service. There are **five** principles of monitoring microservices, as follows:

* Monitor container and what's inside them.
* Alert on service performance.
* Monitor services that are elastic and multi-location.
* Monitor APIs.
* Monitor the organizational structure.

These principles allow us to address technological changes associated with the microservices and organizational changes related to them.

## **Microservices Monitoring Tool**

There are three monitoring tools are as follows:

* Hystrix dashboard
* Eureka admin dashboard
* Spring boot admin dashboard

## **Microservice Virtualization**

Microservices virtualization is the method to simulate the behavior of specific components in various component-based application like cloud-based application, SOA, and API driven architecture. Service virtualization also reduces cost and save time. By combining service virtualization, an organization can develop the application which can be delivered from various locations and dissimilar environments.

# Components of Microservices

There are the following components of microservices:

* Spring Cloud Config Server
* Netflix Eureka Naming Server
* Hystrix Server
* Netflix ZuulAPI Gateway Server
* Netflix Ribbon
* Zipkin Distributed Tracing Server

### **Spring Cloud Config Server**

Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation **@EnableConfigServer**.

### **Netflix Eureka Naming Server**

Netflix Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called **Eureka Client**. Eureka client interacts with the Eureka server for service discovery. The Eureka client also balances the client requests.

### **Hystrix Server**

Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the **Circuit Breaker mechanism**. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.

Backward Skip 10sPlay VideoForward Skip 10s

### **Netflix Zuul API Gateway Server**

Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.

### **Netflix Ribbon**

Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing:

* Load balancing
* Fault tolerance
* Multiple protocols(HTTP, TCP, UDP)
* Caching and Batching

### **Zipkin Distributed Server**

Zipkin is an open-source project m project. That provides a mechanism for sending, receiving, and visualization traces.

One thing you need to be focused on that is port number.

|  |  |
| --- | --- |
| **Application** | **Port** |
| Spring Cloud Config Server | 8888 |
|  |  |
| Netflix Eureka Naming Server | 8761 |
| Netflix Zuul API gateway Server | 8765 |
| Zipkin distributed Tracing Server | 9411 |

# Creating a Simple Microservice

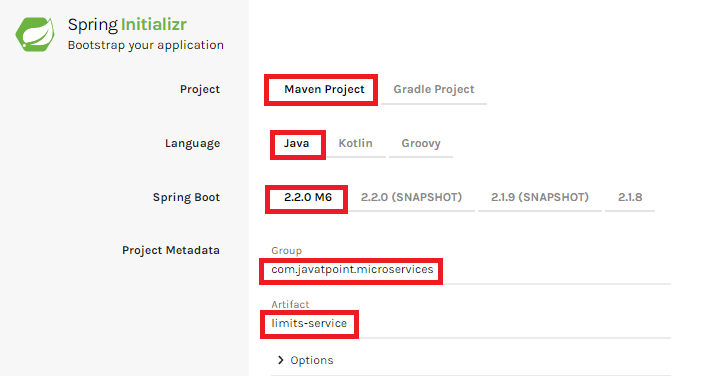
**Step 1**: Create a Maven project using Spring Initializr <https://start.spring.io/>

**Step 2**: Choose the Spring Boot version **2.2.0 M6** or higher version. Do not choose the snapshot version.

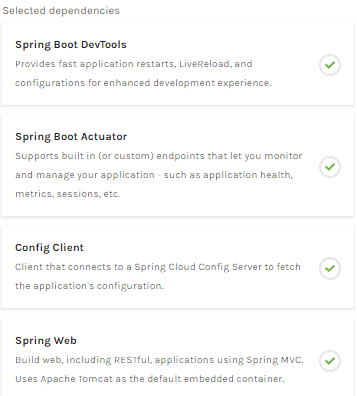
**Step 3**: Provide the **Group** name. In our case **om.javatpoint**

**Step 4**: Provide the **Artifact id**. We have provided **limits-service**.

Backward Skip 10sPlay VideoForward Skip 10s



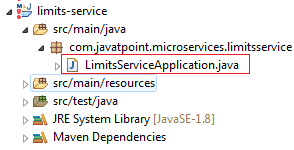
**Step 5**: Add the following dependencies: **Spring Web, Spring Boot DevTools, Spring Boot Actuator, Config Client**.



**Step 6**: Click **on Generate the project** button. A **zip** file will download, extract it into the hard disk.

**Step 7**: Now, open the **eclipse**. Import the created maven project. It takes some time to download the required files.

**Step 8**: Once the project is downloaded, go to **src/main/java**. Open the **LimitsServiceApplication**.



**Step 9**: Now run the **LimitsServiceApplication.java** as Java Application.

**It started the Tomcat on port(s) 8080 (http).**

Now we will add couple of services in the above project. For this we will have to follow the following steps:

**Step 1**: Open **application.properties** file and write the following code:

1. spring.application.name=limits-service      //name of application

**Step 2**: Create a class file with name **LimitsConfigurationController.java** in the folder src/main/java under the package **com.javatpoint.microservices.limitsservice** and write the following code:

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. **import** com.javatpoint.microservices.limitsservice.bean.LimitConfiguration;
5. @RestController
6. **public** **class** LimitsConfigurationController
7. {
8. @GetMapping("/limits")
9. **public** LimitConfiguration retriveLimitsFromConfigurations()
10. {
11. **return** **new** LimitConfiguration(1000, 1);
12. }
13. }

**Step 3**: Create a class file with name **LimitConfiguration.java** in the folder **src/main/java** under the package **com.javatpoint.microservices.limitservice.bean** and write the following code:

1. **package** com.javatpoint.microservices.limitsservice.bean;
2. **public** **class** LimitConfiguration
3. {
4. **private** **int** maximum;
5. **private** **int** minimum;
6. //no-argument constructor
7. **protected** LimitConfiguration()
8. {
9. }
10. //generating getters
11. **public** **int** getMaximum()
12. {
13. **return** maximum;
14. }
15. **public** **int** getMinimum()
16. {
17. **return** minimum;
18. }
19. //genetrating constructor using fields
20. **public** LimitConfiguration(**int** maximum, **int** minimum)
21. {
22. **super**();
23. **this**.maximum = maximum;
24. **this**.minimum = minimum;
25. }
26. }

Type the **localhost:8080/limits** in the browser and press enter, we get the JSON response as output.

**Output**

*{*

*maximum: 1000,*

*minimum: 1*

*}*

## **Adding services to the application.properties**

In the previous program, we will modify the code according to the requirement.

Advertisement

Now we call the **limits-service** from the **application.properties** file. In this file, we are configuring a couple of values.

1. limits-service.minimum=99
2. limits-service.maximum=9999

There is a better approach in Spring Boot to read values from the configuration using the annotation **@ConfigurationProperties**.

**Step 1**: Create a class with name **Configuration.java** in the folder **src/main/java** under the package **com.javatpoint.microservices.limitservice**.

**Step 2**: Add the annotations **@Component** and **@ConfigurationProperties**.

**Step 3**: Declare two variables **minimum** and **maximum**.

**Step 4**: If we are using the Configuration file, we need to generate getters and setters.

The Configuration.java file look like this.

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.boot.context.properties.ConfigurationProperties;
3. **import** org.springframework.stereotype.Component;
4. @Component
5. @ConfigurationProperties("limits-service")
6. **public** **class** Configuration
7. {
8. **private** **int** maximum;
9. **private** **int** minimum;
10. **public** **void** setMaximum(**int** maximum)
11. {
12. **this**.maximum = maximum;
13. }
14. **public** **void** setMinimum(**int** minimum)
15. {
16. **this**.minimum = minimum;
17. }
18. **public** **int** getMaximum()
19. {
20. **return** maximum;
21. }
22. **public** **int** getMinimum()
23. {
24. **return** minimum;
25. }
26. }

**Step 5**: Now move to **LimitsConfigurationController.java** file and modify the code. In this we will use Configuration.

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.GetMapping;
4. **import** org.springframework.web.bind.annotation.RestController;
5. **import** com.javatpoint.microservices.limitsservice.bean.LimitConfiguration;
6. @RestController
7. **public** **class** LimitsConfigurationController
8. {
9. @Autowired
10. **private** Configuration configuration;
11. @GetMapping("/limits")
12. **public** LimitConfiguration retriveLimitsFromConfigurations()
13. {
14. //getting values from the properties file
15. **return** **new** LimitConfiguration(configuration.getMaximum(), configuration.getMinimum());
16. }
17. }

Now refresh the browser page. It shows the JSON format of the updated values which are configured in **application .properties** file.

**Output**

*{*

*maximum: 999,*

*minimum: 99*

*}*