Microservices Tutorial

**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**: According to **Sam Newman**, "Microservices are the small services that work together."

According to **James Lewis and Martin Fowler**, "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."

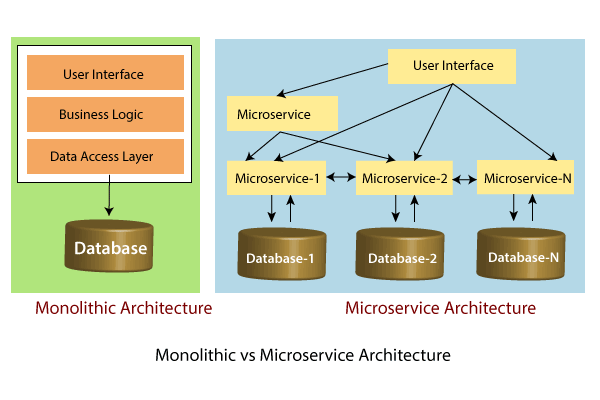
There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.

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. It is next to **Service-Oriented Architecture (SOA)**. The most important feature of the microservice-based architecture is that it can perform **continuous delivery** of a large and complex application.

Microservice helps in breaking the application and build a logically independent smaller applications. For example, we can build a cloud application with the help of Amazon AWS with minimum efforts.



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.

**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.

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.

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 |

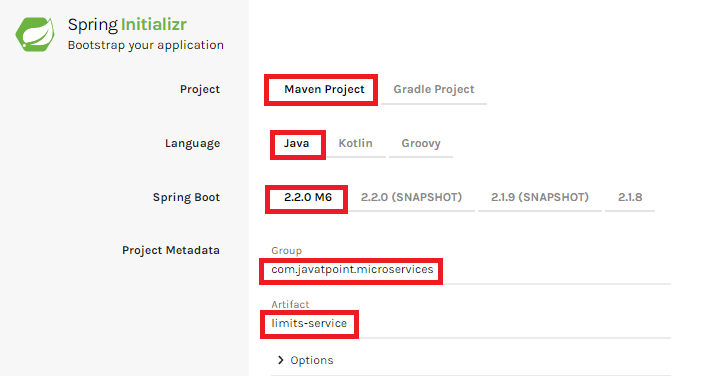
Creating a Simple Microservice

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

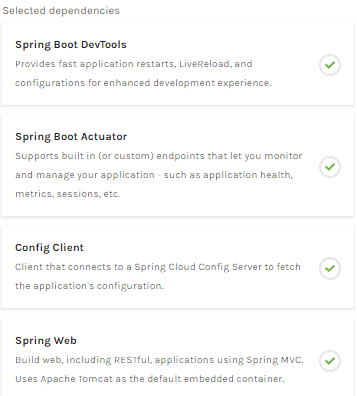
**Step 2**: Choose the Spring Boot version **2.2.0 M6** or higher version. Do not choose the snapshot version.

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

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



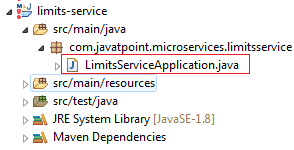
**Step 5**: Add the following dependencies: **Spring Web, Spring Boot DevTools, Spring Boot Actuator, Config Client**.



**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.

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

}

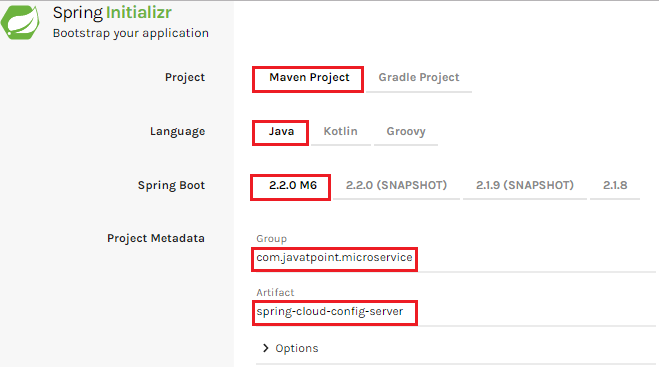
Setting up Spring Cloud Config Server

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

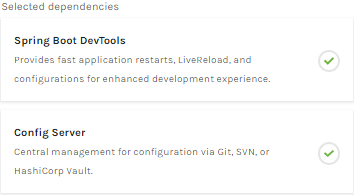
**Step 2:** Choose the Spring Boot version **2.2.0 M6** or higher version. Do not choose the snapshot version.

**Step** **3:** Provide the **Group** name. In our case, **com.javatpoint.microservices.**

**Step 4:** Provide the **Artifact id**. We have provided **spring-cloud-config-server.**



**Step 5:**Add the **Spring Boot DevTools**and**Config Server**dependencies**.**



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

**Step 7:** Now, open the **eclipse.** Import the downloaded maven project. It will download the required files.

In the next step, we will create a simple Git repository and configure the spring cloud config server to pick up the values from the particular Git repository. We need to install the local Git.

Installing Git and creating a local repository

**Step 1:**Download Git from <https://git-scm.com/> and install it.

**Step 2:**Create a Git repository and store the files that we want to be able to configure a limits-service. We will try to access them from the spring-cloud-config-server. Open the Git bash and type the following commands:

Creating a new directory:

1. mkdir git-localconfig-repo
2. cd git-localconfig-repo/

Initializing a new Git repository:

1. git init

It initializes an **empty**git repository.

**Step 3:**Now move to the **spring-cloud-config-server** project and add a link to the specific folder.

1. Right-click on the **spring-cloud-config-server**project**.**
2. Click on **Build Path**->**Configure Build Path**…
3. Select the **Source** tab.
4. Click on **Link Source** and browse the folder **git-localconfig-repo**.
5. Right click on the folder-> **New** -> **Other** -> **File** -> **Next** -> Provide the file name**: limits-service-properties**-> **Finish**.
6. Now write the following code in the properties file:
7. limits-service.minimum=8
8. limits-service.maximum=888

**Step 4:**Configure the user name and user email:

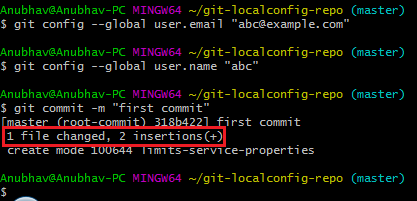
1. git config -global user.email abc@example.com
2. git config -global user.name "abc"

The command commits any file we have added with the git add command and also commits any files we have changed since then.

1. git add -A

Now execute the command to commit the changes in the repository. It records or snapshots the file permanently in the version history.

1. git commit -m "first commit"



We can see that a file is changed with two new instructions. These instructions are changed in the local repository.

Connect Spring Cloud Config Server to Local Git Repository

In this section, we are going to learn how to connect spring-cloud-config-server to the local git repository. First, we will find the folder path.

Right-click on **git-localconfig-repo** -> **Properties** -> copy the **Location** label address and paste it into the **application.properties** file.

Add the annotation **@EnableConfigServer**in the SpringCloudConfigServerApplication.java file.

Type the following URL in the browser:

**localhost:8888/limits-service/default**

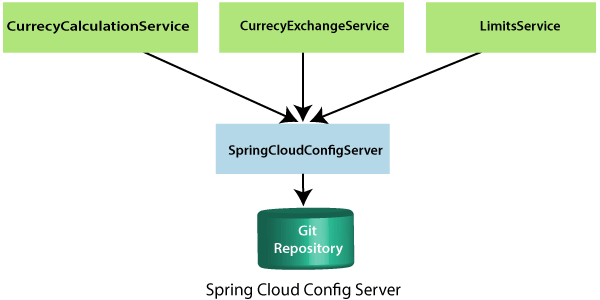
**Output**

1. {
2. name: "limits-service",
3. -profiles: [
4. "default"
5. ],
6. label: **null**,
7. version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",
8. state: **null**,
9. -propertySources: [
10. - {
11. name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties",
12. -source: {
13. limits-service-minimum: "8",
14. limits-service-maximum: "88"
15. }
16. }
17. ]
18. }

In this we have establish the connection between **SprinCloudConfigServer** and the **Git repository**.

We can see that it displays a set of property and values. It also retrieves the file name of the property file from where these values (minimum and maximum) are retrieved.

The important thing about SpringCloudConfigServer is that **it stores configuration for multiple services.**It can also store configuration for each of the services for different environments.



In the above figure, there are three services **CurrencyCalculationService**, **CurrencyExchangeService**, and **LimitsService**. The LimitsService has four environment services **Dev, QA, Stage,** and **Production**. We can configure these three services in SpringCloudConfigServer.

Configuration for Multiple Environment in Git Repository

services **Dev, QA, Stage,** and **Production**. We can configure these three services in SpringCloudConfigServer.

**Configuration for Multiple Environment in Git Repository**

In the spring-cloud-config-server project, we have added a link to git-localconfig-repo, which contains the limits-service.properties file. It becomes the default configuration for the limits-service.

However, we can overwrite them for a specific environment. To overwrite these values, copy the **limits-service.properties** and paste in the folder **git-localconfig-repo**rename it with **limits-service-dev.properties**. Now update the minimum and maximum values.

1. limits-service.minimum=1
2. limits-service.maximum=111

Again copy the same file and paste it in the same folder. Rename it with **limits-service-qa.properties**. Now update the minimum and maximum values.

1. limits-service.minimum=2
2. limits-service.maximum=222

If we want to pick the default value of the maximum instead of modified value, put a **introduction-to-currency-conversion-and-currency-exchange-service** symbol at the starting of the statement. Now the second statement becomes a comment.

1. limits-service.minimum=1
2. introduction-to-currency-conversion-and-currency-exchange-servicelimits-service.maximum=111

When we execute it, it picks up the maximum value 888 from the default properties file instead of maximum value 111. Whenever we make the changes in the file, commit the changes in the local repository.

Now open the Git Bash and execute the following commands:

Create the directory in which we want to add files.

1. cd git-localconfig-repo

Add the files into the Git repository.

1. git add -A

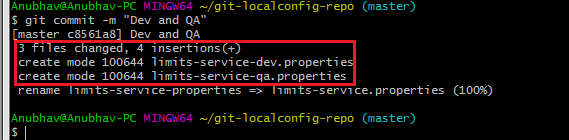
Now check the status of the files that have to be committed.

1. git status



Now commit the changes

1. git commit -m "Dev and QA"



Now we can access the properties Dev and QA.

Type the following in the address bar of the browser.

1. localhost:8888/limits-service/qa

**Output**

1. {
2. name: "limits-service",
3. -profiles: [
4. "qa"
5. ],
6. label: **null**,
7. version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",
8. state: **null**,
9. -propertySources: [
10. - {
11. name: C:\Users\Anubhav\git-localconfig-repo\limits-service-qa.properties",
12. -source: {
13. limits-service-minimum: "2",
14. limits-service-maximum: "222"
15. }
16. },
17. -{
18. name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties?,
19. -source: {
20. limits-service-minimum: "8",
21. limits-service-maximum: "888"
22. }
23. }
24. ]
25. }

We can observe that it is retrieving the property sources. These list of property are in the list of priority. The heights priority is whatever values are configured in the QA file.

If there is a value that is not present in the QA file, then the value from the default file will be picked up. So whatever is in the QA file gets the highest property.

Connect limits-service to Spring Cloud Config Server

In this section, we will connect limits-service to pick up the configuration from the spring-cloud-config-server. We do not need to configure values in the application.properties file. Move to the **limits-service** project and rename the **application.properties** file to **bootstrap.properties**. We do not need to configure values in the bootstrap.properties. All the configuration values picked from the spring-cloud-config-server. Specify the URI in the bootstrap.properties.

1. spring.application.name=limits-service
2. spring.cloud.config.uri=http://localhost:8888

**limits-service** is the critical path of the bootstrap.properties. Based on the application name, we are going to pick up values from the local Git repository. Now restart the **LimitsServiceApplication.java.**

1. Fetching config from the server at http://localhost:8888
2. Located environment: name=limits-service, profiles=[**default**], label= **null**,  version="0898c54ae1deb62733728e37e4c7962f529ee9ad", state=**null**,

Configuring profiles for Limit Service

The point to understand here is that all the configuration for the limits-service is coming from the Git repository. We did not configure anything in the limits-service. The advantage of configuring stuff in the Git repository is that the entire configuration of limits-service is separated from the deployment of the limits-service. It will automatically pick up from the Git repository.

Now open the **bootstrap.properties** and add the **dev** profile into it.

1. spring .profile.active=dev

When we run the limits, it shows the following output:

1. {
2. maximum: 111,
3. minimum:1
4. }

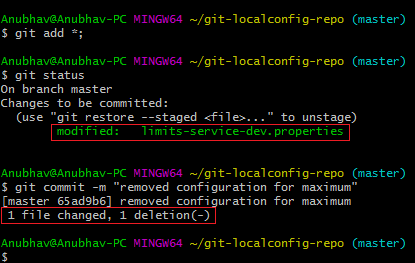
If we look at the limits-service-dev.properties file, the values are fetching from there.

Suppose we want to pick a maximum value from the **limits-service.properties** and minimum value from **limits-service-dev.properties** then remove the maximum value from the **limits-service-dev.properties**. The limits-service-dev.properties file looks like this:

1. limits-service-minimum: 1

Now commit the changes by using the following commands:

1. git add \*;
2. git status
3. git commit -m "removed configuration for maximum "



Now start the **LimitsServiceApplication.java**. When we start the LimitsServiceApplication, it picks values from the SpringCloudConfigServer. We can observe that it picks the maximum value from the limits-service.properties (default service) that is**888**and the minimum value from the **limit-service-dev.properties**that is **1.**However, we have overwritten the minimum value of the default service.

Let's see what happens when we change the profile **dev** to **qa.**Open **bootstrap.properties**and write **qa** in place of **dev**. The application will start and pick up the changes. Now execute the **limits.**

**Output**

{

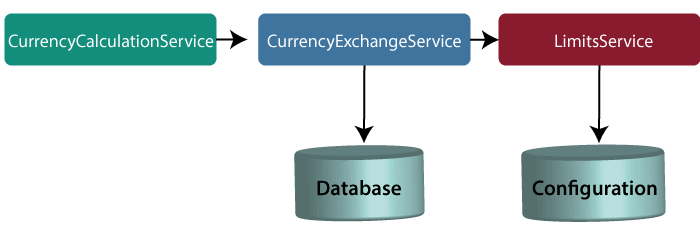
maximum: 222,

minimum: 2

}

# Introduction to Currency Conversion and Currency Exchange Service

In this section, we will create a couple of microservices: **CurrencyCalculationService** and **CurrencyExchangeService**.



#### Note: In this tutorial, we have quoted currency conversion service as a currency calculation service. Both the services have the same meaning, so don't be confused.

Let's understand the functionality of these services.

In the above figure, the CurrencyExchangeService uses JPA to talk to the database and returns the exchange value of the specific currency. For example, USD to INR conversion.

When we invoke CurrencyExchangeService, we need to pass two parameters: **from**(convert from), and **to** (convert to). For example, if we want to convert currency from **USD** to **INR**.

Consider the URL **http://localhost:8000/currency-exchange/from/USD/to/INR**. It retunes the following response:

1. {
2. id: 101,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 72,
6. port: 8000
7. }

The currency exchange service will return what the conversion multiple is. The conversion multiple means **1 USD** is equal to **72 INR**. The currency converter service uses a currency exchange service. Suppose the currency converter service wants to convert 100 USD to INR. So it will call the currency exchange service and will convert the specified amount that we have provided in the path parameter. For example:

**http://localhost:8100/currency-converter/from/USD/to/INR/quantity/100**

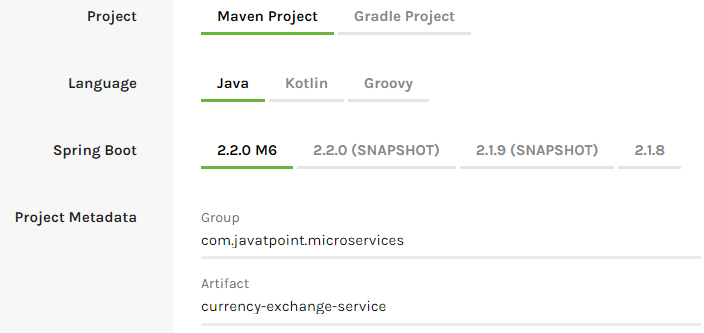
1. {
2. Id: 101,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 72,
6. quantity: 100
7. totalCalculatedAmount: 7200,
8. port: 8000
9. }

We will implement these two services in our example using Spring Cloud.

## Setting up a currency-exchange-service

**Step 1:**Open the spring initializer [http://start.spring.io](http://start.spring.io/).

**Step 2:**Select the **Project**: Maven Project, **Language:**Java, and Spring Boot version **2.2.0 M6**or above. Provide the **Group name** and **Artifact ID.**We have provided**com.javatpoint.microservices**and **currency-exchange-service,**for group name and Artifact id respectively.



**Step 3:**Add the dependencies **Web, DevTools, Actuator,**and **Config Client**.

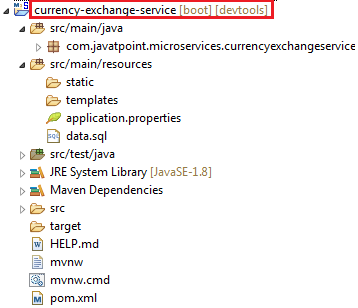
**Step 4:**Click on the **Generate Project** button. It will download the **zip** file of the project.

**Step 5: Extract** it in the local disk.

**Step 6: Import** the project.

Click on File menu-> Import -> Existing Maven Projects -> Next -> Browse ->Select the project ->Finish

It takes some time to import. When the project import is done, it shows the following project directory. Do not consider the data.sql file in the directory, because we will create it later.



**Step 7:**Open the **application.properties** file and configure the **application name** and **port** number.

**application.properties**

1. spring.application.name=currency-exchange-service.
2. server.port=8000

When we run the currency-exchange-service, it runs but does not perform any service. In the next step, we will implement code in the currency-exchange-service.

## Hardcoded the currency-exchange-service

Now we will create a service that converts the currency from USD to INR.

**Step 1:**Create a class file (REST Controller) with the name **CurrencyExchangeController** in the package **com.javatpoint.microservices.currencyexchangeservice.**

**CurrencyExchangeController.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **import** org.springframework.boot.SpringApplication;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.RestController;
8. @SpringBootApplication
9. @RestController
10. **public** **class** CurrencyExchangeController
11. {
12. @GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable
13. **public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)  //from map to USD and to map to INR
14. {
15. **return** **new**  ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));
16. }
17. }

**Step 2:**Create a class file with the name **ExchangeValue.**

**ExchangeValue.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **public** **class** ExchangeValue
4. {
5. **private** Long id;
6. **private** String from;
7. **private** String to;
8. **private** BigDecimal conversionMultiple;
10. **public** ExchangeValue()
11. {
12. }
13. //generating constructor using fields
14. **public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {
15. **super**();
16. **this**.id = id;
17. **this**.from = from;
18. **this**.to = to;
19. **this**.conversionMultiple = conversionMultiple;
20. }
21. //generating getters
22. **public** Long getId()
23. {
24. **return** id;
25. }
26. **public** String getFrom()
27. {
28. **return** from;
29. }
30. **public** String getTo()
31. {
32. **return** to;
33. }
34. **public** BigDecimal getConversionMultiple()
35. {
36. **return** conversionMultiple;
37. }
38. }

**Step 3:**Run the **CurrencyExchangeServiceApplication.java.**It runs on the port **8000** that we have configured in the application.properties file.

We get the following response on the browser:

1. {
2. id: 101,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 72,
6. port: 8000
7. }

## Setting up Dynamic port in the Response

The CurrencyExchangeService determines the exchange value of the currency. The CurrencyCalculationService uses the CurrencyExchangeService to determine the value of one currency in other currency. We will create multiple instances of the **CurrencyExchangeService** later in next topic.

At present, the service is running on port **8000**. Later we will run it on port **8001, 8002,** and so on.  In the next step, we will set a port to the currency-exchange-service.

**Step 1:** Open the **ExchangeValue.java** file and add a **port**variable. Generate getters and setters for the port variable only.

**ExchangeValue.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **public** **class** ExchangeValue
4. {
5. **private** Long id;
6. **private** String from;
7. **private** String to;
8. **private** BigDecimal conversionMultiple;
9. **private** **int** port;
10. **public** ExchangeValue()
11. {
12. }
13. //generating constructor using fields
14. **public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {
15. **super**();
16. **this**.id = id;
17. **this**.from = from;
18. **this**.to = to;
19. **this**.conversionMultiple = conversionMultiple;
20. }
21. //generating getters
22. **public** **int** getPort() {
23. **return** port;
24. }
25. **public** **void** setPort(**int** port) {
26. **this**.port = port;
27. }
28. **public** Long getId()
29. {
30. **return** id;
31. }
32. **public** String getFrom()
33. {
34. **return** from;
35. }
36. **public** String getTo()
37. {
38. **return** to;
39. }
40. **public** BigDecimal getConversionMultiple()
41. {
42. **return** conversionMultiple;
43. }
44. }

We have already configured the application name and port number in the application.properties file, so need not to configure again.

Now pick up port number from the environment.

**Step 3**: Open the **CurrencyExchangeController.java** and get the property of the environment.

**CurrencyExchangeController.java.**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.core.env.Environment;
6. **import** org.springframework.web.bind.annotation.GetMapping;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.RestController;
9. @SpringBootApplication
10. @RestController
11. **public** **class** CurrencyExchangeController
12. {
13. @Autowired
14. **private** Environment environment;
15. @GetMapping("/currency-exchange/from/{from}/to/{to}") //where {from} and {to} are path variable
16. **public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)  //from map to USD and to map to INR
17. {
18. //taking the exchange value
19. ExchangeValue exchangeValue= **new** ExchangeValue (1000L, from, to, BigDecimal.valueOf(65));
20. //picking port from the environment
21. exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));
22. **return** exchangeValue;
23. }
24. }

When we refresh the browser, the URL changes to: **http://localhost:8000/currency-exchange/from/USD/to/INR**.

1. {
2. id: 1000,
3. from: "USD",
4. to: "INR"
5. conversionMultiple: 65,
6. port: 8000
7. }

At present **CurrencyExchangeServiceApplication** is running on port **8000**.

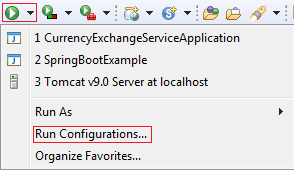
Now we will run **CurrencyExchangeServiceApplication** on a different port number. For this, we have to change the port in the **application.properties**file from 8000 to 8001, 8002, etc. whichever we want.

Suppose we want to create two instances of the **CurrencyExchangeServiceApplication**. For this, we have to set port externally.

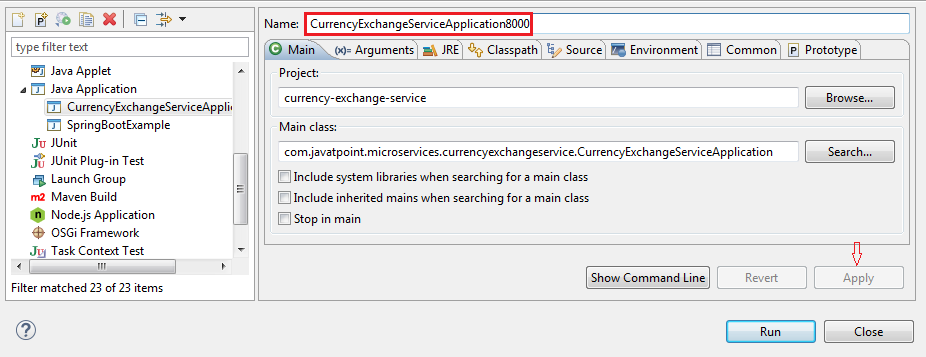
Let's create an instance of the **CurrencyExchangeServiceApplication**that runs on the port **8001**.

**Step 1:**Right-click on the project -> Run As -> Run Configurations.

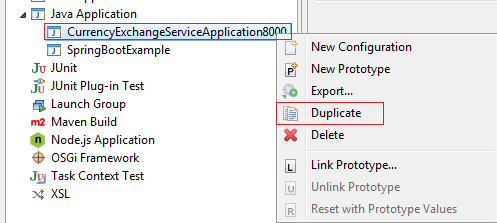
Or click on the highlighted symbol -> Run Configurations.



**Step 2: Rename**the**CurrencyExchangeServiceAppication** to**CurrencyExchangeServiceAppication8000**and click on the**Apply**button.



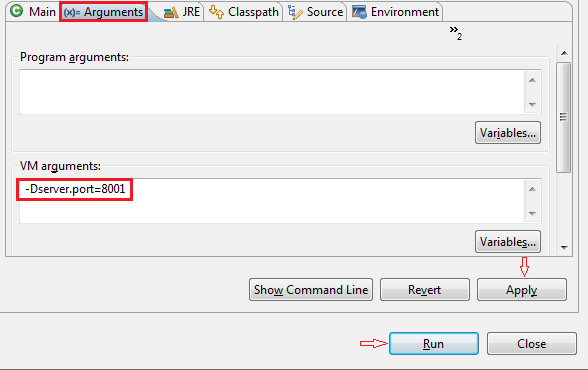
**Step 3:**Right-click on the **CurrencyExchangeServiceApplication8000**-> Duplicate.



It generates the duplicate file of **CurrencyExchangeServiceApplication8000.**We will run it on port **8001.**

**Step 4:**Click on the **Arguments** tab and write **–Dserver.port=8001**in the **VM arguments** text box. Click on the **Apply**and **Run** button, respectively.

#### Note: Whatever value we are passing in the VM arguments, it overwrites the configuration of the application.properties file.



After clicking on the **Run**button, it starts running on port **8001**.

**Step 5:**Change the port number in the URL **http://localhost:8001/currency-exchange/from/USD/to/INR** and press enter key. We get the following response:

1. {
2. id: 1000,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 65,
6. port: 8001
7. }

Now we have two instances of **CurrencyExchangeServiceApplication**that are running on two different ports **8000** and **8001**.

In the previous section, we have hardcoded the response for the exchange value. It comes from the database. In this section, we will create a connection to the in-memory database.

Let's see how to connect microservice to the H2 database. Follow the following steps to connect microservice to JPA in-memory database.

**Step 1:** Open **pom.xml** of **currency-exchange-service** and add the following two dependencies.

1. <dependency>
2. <groupId>com.h2database</groupId>
3. <artifactId>h2</artifactId>
4. <version>1.4.197</version>
5. <scope>test</scope>
6. </dependency>
7. <dependency>
8. <groupId>org.springframework.boot</groupId>
9. <artifactId>spring-boot-starter-data-jpa</artifactId>
10. <version>2.1.3.RELEASE</version>
11. </dependency>

Once we have added the dependencies, now we have to define **entity**.

**Step 2:**Open the **ExchangeValue.java**file and do the following:

* Add **@Entity** annotation at the class level.
* Define the **table name** by using the annotation **@Table**.
* Define an **Id** for the entity by adding the annotation **@Id.**
* Define columns by adding the annotation **@Column**above each field and also specify the column name.

**ExchangeValue.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.Id;
6. **import** javax.persistence.Table;
7. @Entity
8. @Table(name="Exchange\_Value")
9. **public** **class** ExchangeValue
10. {
11. @Id
12. @Column(name="id")
13. **private** Long id;
14. @Column(name="currency\_from")
15. **private** String from;
16. @Column(name="currency\_to")
17. **private** String to;
18. @Column(name="conversion\_multiple")
19. **private** BigDecimal conversionMultiple;
20. @Column(name="port")
21. **private** **int** port;
22. //default conatructor
23. **public** ExchangeValue()
24. {
25. }
26. //generating constructor using fields
27. **public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple)
28. {
29. **super**();
30. **this**.id = id;
31. **this**.from = from;
32. **this**.to = to;
33. **this**.conversionMultiple = conversionMultiple;
34. }
35. //generating getters and setters
36. **public** **int** getPort()
37. {
38. **return** port;
39. }
40. **public** **void** setPort(**int** port)
41. {
42. **this**.port = port;
43. }
44. **public** Long getId()
45. {
46. **return** id;
47. }
48. **public** String getFrom()
49. {
50. **return** from;
51. }
52. **public** String getTo()
53. {
54. **return** to;
55. }
56. **public** BigDecimal getConversionMultiple()
57. {
58. **return** conversionMultiple;
59. }
60. }

We have created the entity, now we have to insert some data into the database.

**Step 3:** Create a **data.sql**file to insert data into database.

Right-click on the folder **src/main/resources** -> New -> File -> Provide the name **data.sql** -> Finish

**Step 4:**Insert the data into data.sql file. We have inserted the following data:

**data.sql**

1. insert into exchange\_value(id,currency\_from,currency\_to,conversion\_multiple,port)
2. values(10001,'USD', 'INR' ,65,0);
3. insert into exchange\_value(id,currency\_from,currency\_to,conversion\_multiple,port)
4. values(10002,'EUR', 'INR' ,75,0);
5. insert into exchange\_value(id,currency\_from,currency\_to,conversion\_multiple,port)
6. values(10003,'AUD', 'INR' ,25,0);

**Step 5:** Open **application.properties**file and enable **H2 console,**configure **URL** and **datasource**. The default JDBC URL is **testdb**. We can specify our own JDBC URL.

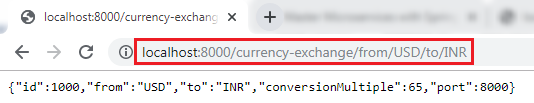
We have specified JDBS URL: **jdbc:h2:mem:javatpoint**

**application.properties**

1. spring.application.name=currency-exchange-service
2. server.port=8000
3. spring.jpa.show-sql=**true**
4. spring.h2.console.enabled=**true**
5. spring.datasource.platform=h2
6. spring.datasource.url=jdbc:h2:mem:javatpoint

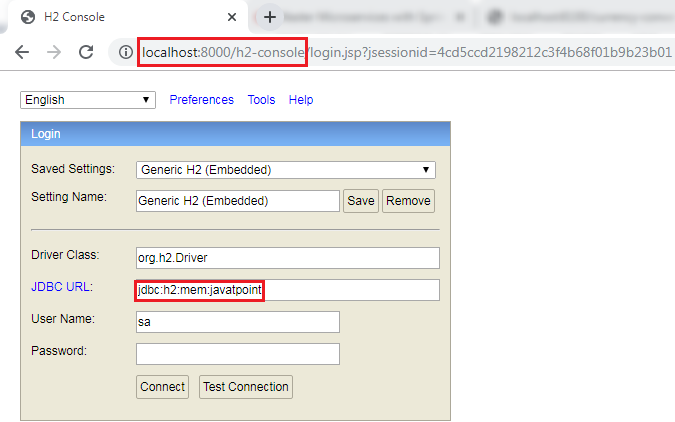
**Step 6:** Restart the application.

**Step 7:**Open the browser and type the URI **http://localhost:8000/currency-exchange/from/USD/to/INR**. It returns the response, as shown below:



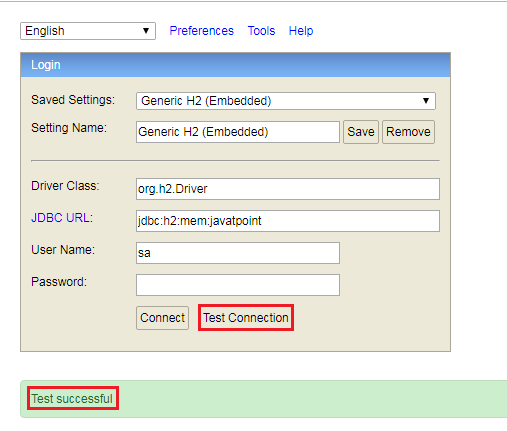
We can also see the data in the database which we have inserted in the **data.sql** file. To open the H2 Console, we have to do the following:

* In the browser type <http://localhost:8000/h2-console>. It displays the following page:

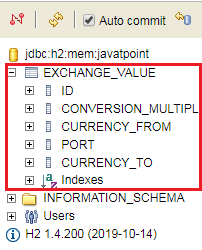


**Remember:** The **JDBC URL** must be the same as you have specified in the **application.properties** file. Do not write anything in the **User Name** and **Password** field. The default User Name is **sa.**

Now click on the **Test Connection**button**;**if the connection is successful, it shows the message **Test Successful.**



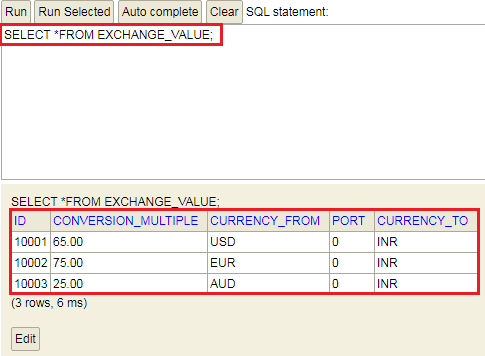
**Step 8:** Click on the **Connect** button. It shows the created table on the left-hand side of the page.



**Step 9:** Run the following query to view the data.

1. SELECT \*FROM EXCHANGE\_VALUE;

It shows the data which we have inserted in the **data.sql** file.



In this section, we have created the in-memory database and inserted some values in the database.