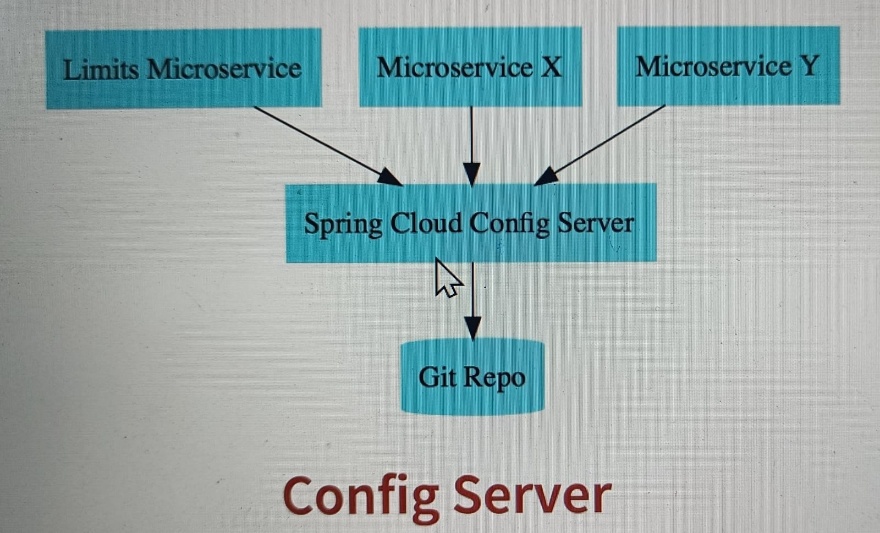
**MICROSERVICES SOLUTIONS**

Spring Cloud is actually a set of projects, its an Umbrella project that provides essential microservice needs. Lets look at the important features that provided by spring cloud.

1. **Centralized Configuration –**

Spring Cloud config server helps you to centrally manage configurations for multiple microservices in central git repository.



So assume that you have multiple microservices, each of these microservices have multiple environments and each microservice might have different configuration in different environments, and therefor managing configuration for the application like which database to connect, which queue to connect so managing this is very, very difficult and that’s why one of the approaches is used in microservices architecture is that to create a central git repository, and you will store multiple configurations of multiple environments in this git repo. Spring cloud config server enables that.

1. **Load Balancing –**

Spring cloud load balancing will helps you to distributes requests across active instances of microservices dynamically. You might have n no. of instances of each microservice, you would want to be able to automatically distribute the load between the active instances of microservices.

For example microservice A might right now have 10 instances, when 10 instances are active you would want to load balance between the existing 10 instances. So this should be dynamic

1. **Service Discovery –**

You might have 100 of microservices and you don’t want to hard code the URLs of each of the microservices, in other microservices. This will make it very, very difficult to make changes, and that’s why in microservices architecture we make use of a feature called service discovery. It enables automatic discovery of microservices, if microservice A needs to talk to microservice B, it needs to talk something called naming server.

For example a Eureka naming server, it will get the address of microservice B and it will send the request to microservice B.

1. **Distributes Tracing –**

In microservice architecture a single request might go through multiple microservices, lets say a single request went through microservice one, microservice two, microservice three, four, and five. Lets say there is a bug, then how do you find out which microservice caused the problem? If you don’t have distributed tracing or if you don’t have tracing of requests across microservices, it becomes very, very difficult to solve problems. Spring Cloud provides really good integration with distributed tracing solutions like Zipkin, Jaeger and lot of others.

1. **Edge Server -**

Spring cloud gateway is one of the popular edge server implementations. Why do you need an edge server? Lets say you would want to implement authentication. Then you don’t want to implement authentication in every microservice, it’s a common feature and that’s why what you can do is you can implement it on the edge server, you can create an API gateway, and ensures that all the requests will only go through API gateway. So you would implement authentication at central place in common to all microservices. So Spring cloud gateway can act as a Edge server, you can implement a single entry point for all your microservices where you can implement a lot of common features.

1. **Fault Tolerance –**

You don’t want a failure in one microservice to cascade and make other microservices to fail, lets say a microservice 1 is calling microservice 2 and microservice 2 is calling microservice 3, if microservice 3 fails, both microservice one and microservice 2 will fail and that’s why your services should be fault tolerant. Spring cloud provides good integration with frameworks like Resilience Forge to make your services fault tolerant.

**Introduction to Microservices with Spring Cloud**

1. Spring Cloud Config Server and Bus
2. Load Balancing with Ribbon and Feign
3. Implement Naming Server with Eureka
4. Implementing API Gateway with Zuul
5. Distributed Tracing with Zipkin
6. Fault Tolerance with Hystrix

Microservices are a services which are exposed by a REST, in addition to that you have small deployable units with very well through out boundaries, small well Chosen Deployable units and this should be cloud enabled.

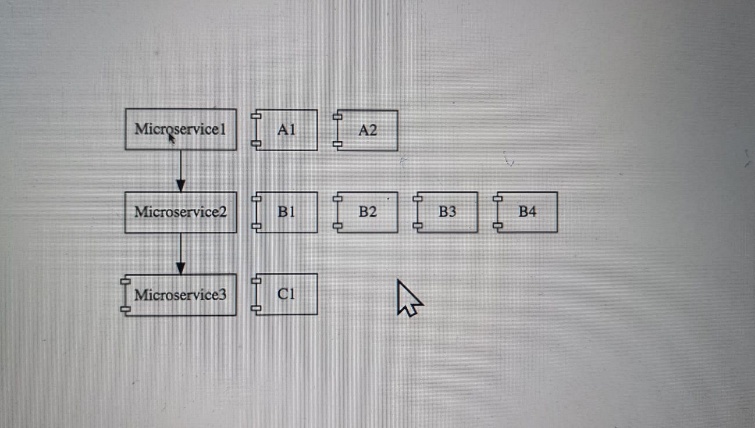
Microservices4

Microservices3

Microservices2

Microservices1

When you build microservices this is how it would look, instead of building one big monolithic application you would be building a set of smaller microservices, I am showing 4 in the picture it might be 10 , 15, 100 or thousand so it’s a set of micro services with w will define boundaries which are interacting with each other.



This are cloud enabled, that means I would be able to have multiple instances for each of the microservices, by cloud enabled I mean that if there is more load on microservice 3 I should be easily bring up another instance of microservice 3 that is mean by cloud enabled.

**Challenges with Microservices –**

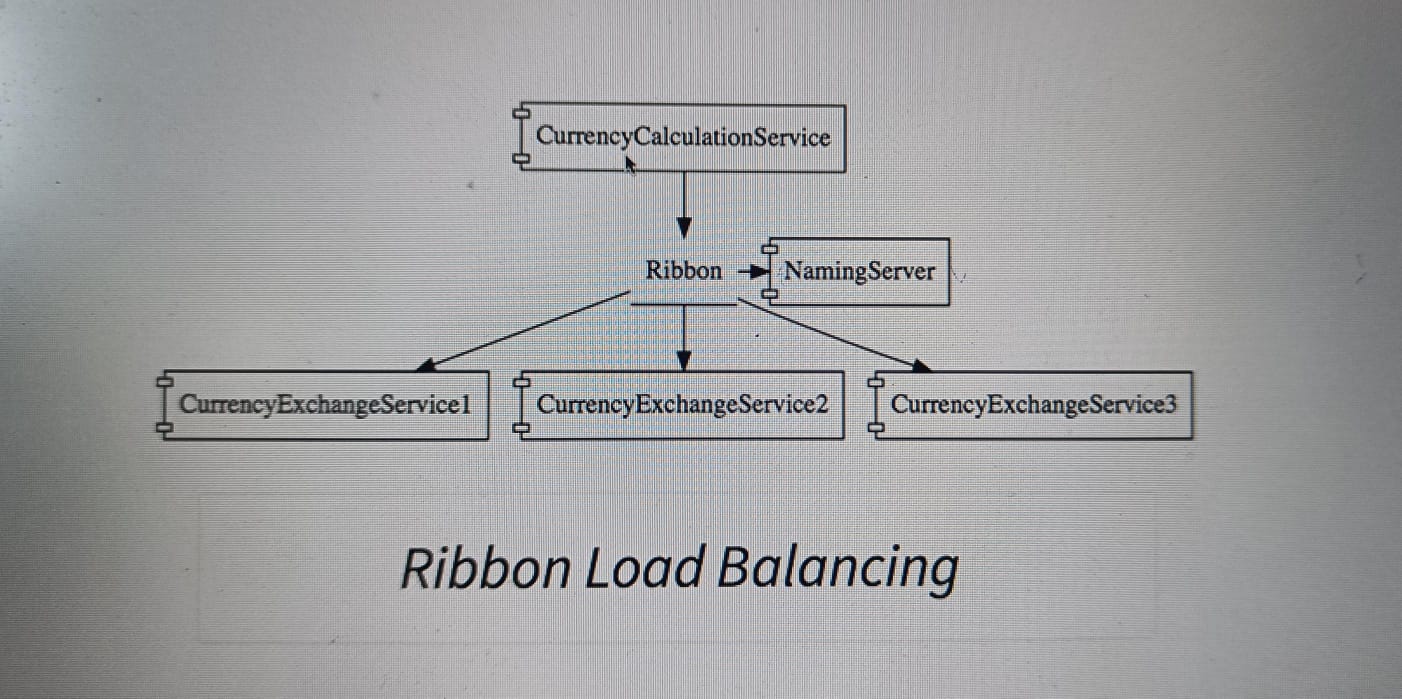
1. BOUNDED CONTEXT - earlier we said instead of one big monolith, application we would building about 5 small microservices or 10, or 20 or 100, but then how do you identify the boundary for each of this microservices , how do you identify what to do in each of this microservices how do you decide what you should do and what you should not do, the thing is for new application this is especially much more difficult because probably you don’t have the business knowledge to be able to establish the right boundaries between these microservices, what I found in my experiences deciding the boundaries of microservices is an evolutionary process its not something you get right at the first time, something which you need to play around with try and follow domain driven design, try to identify the right boundaries for the microservices based on the knowledge you have at that point in time.

The important thing you need to understand as you keep gaining knowledge you should put that knowledge back into the microservices.

1. CONFIGURATION MANAGEMENT – we said we have 5 or 50 microservices, these microservices will have multiple instances in each environment and there are multiple environments, so lets say there are 10 microservices with 5 environment and lest say 50 instances so we are talking about tons of configuration and that’s a lot of work for the operations team to maintain.
2. DYNAMIC SCALE UP AND SCALE DOWN – the loads of the microservices will be different at the different instances of time and at particular instance I might need 2 instances of microservice 2 but later at different point I might needing 10 instances of this, so I should be able to bring new instances of microservices up and bring down older instances of microservices when they are not needed, all this is because dynamic load balancing because when there is a 1 instance of microservice 1 and there is 4 instance of microservice2 then I would want to distribute the load between all the instances of microservice2 and if there are 4 instances of microservice2 coming up then I would want to ensure that all the new ones are also being used to the fullest extends, so we need the ability to dynamically bring new instances and also to distribute the load among the new instances.
3. VISIBILITY – one of the most important challenges is visibility if I say the functionality is distributed among 10 microservices and there is a bug, then how do you identify where the bug is you need to have centralised log where I can go and find out what happened for a specific request which microservice cause the problem not just that we also need monitoring around the microservices because we have 100s of microservices we need to be able to identify the microservices which are down, we would want to be able to automatically identify servers where there is not enuf disk space all this kind of things need to be automated, so we need grate visibility to what’s happening with these microservices.
4. PACK OF CARDS – in microservices architecture you have an one microservice calling another, and another calling another so there would be certain microservices which would be the fundamental for the whole thing and if that microservice goes down then the entire application might go down so its like an pack of cards you building one over the top of the other an so and so on and there for they can collapse very easily , and there fore its very important that you to have fault tolerance into your microservices.

**Spring Cloud –**

One of the challenges we talked about earlier was configuration management, we talked about the fact that the multiple microservices multiple environment for each of this microservices and multiple instances in many of those environment this would means there is a lot of configuration for these microservices that the operations team needs to manage, spring cloud config server provides an approach where you can store all the configuration for all the different environments of all the microservices in a git repository, so you can store all the configurations for different environments of different microservices in just one place in a centralised location and spring cloud config server can be used to expose that configuration to all the microservices this helps us to keep the configuration in one place that makes us very each to maintain the configuration for all microservices.



The next challenge we talked about the dynamic scale up and scale down, in the example you are looking at there is a microservice called CurrencyCalculationService which is talking to the CurrencyExchangeService1 as you can see in the diagram there are multiple exchange service and its possible that at any point in time that new instances can be added in or removed out and we would want that the CurrencyCalculationService to be able to distribute the load between all the instances of the CurrencyExchangeService, we would want to be able to dynamically check what are the available instances of the CurrencyExchangeService and make sure that the load is distributed the load among all of them.

The solution we would be discuss is the naming server which is (Eureka) so all the instances of all microservices would register with the naming server, so the naming server has the two important features, one is service registration so all microservices can register with the microservice.

The second one is the service discovery, in this microservice the CurrencyCalculationService can ask the Eureka naming server, hey naming server give me the current instances of the CurrencyExchangeService and the naming server would provide those URL to the CurrencyCalculationService this helps to establish the dynamic relationship with the CurrencyCalculationService and the instances of the CurrencyExchangeService, we would use Ribbon for client side load balancing that means the CurrencyCalculationService will host Ribbon and it would make sure that the load is evenly distributed among the existing instances that it gets from the naming server, we will also use Feign in the CurrencyCalculationService as a mechanism to write simple REST ful clients.

The solution for VISIBILITY AND MONITORING are a Zipkin Distributed Tracing server we would use spring cloud slouth to assign a id to a request across multiple components and we would use Zipkin Distributed Tracing to trace a request across multiple components one of the important things about the microservices is this microservices have a lot of common features, for example logging, security, analytics and things like that you don’t want to implement all those common features in every microservice API GATEWAYS provide solutions to this kind of grate challenges we will use a NETFLIX zool API Gateway.

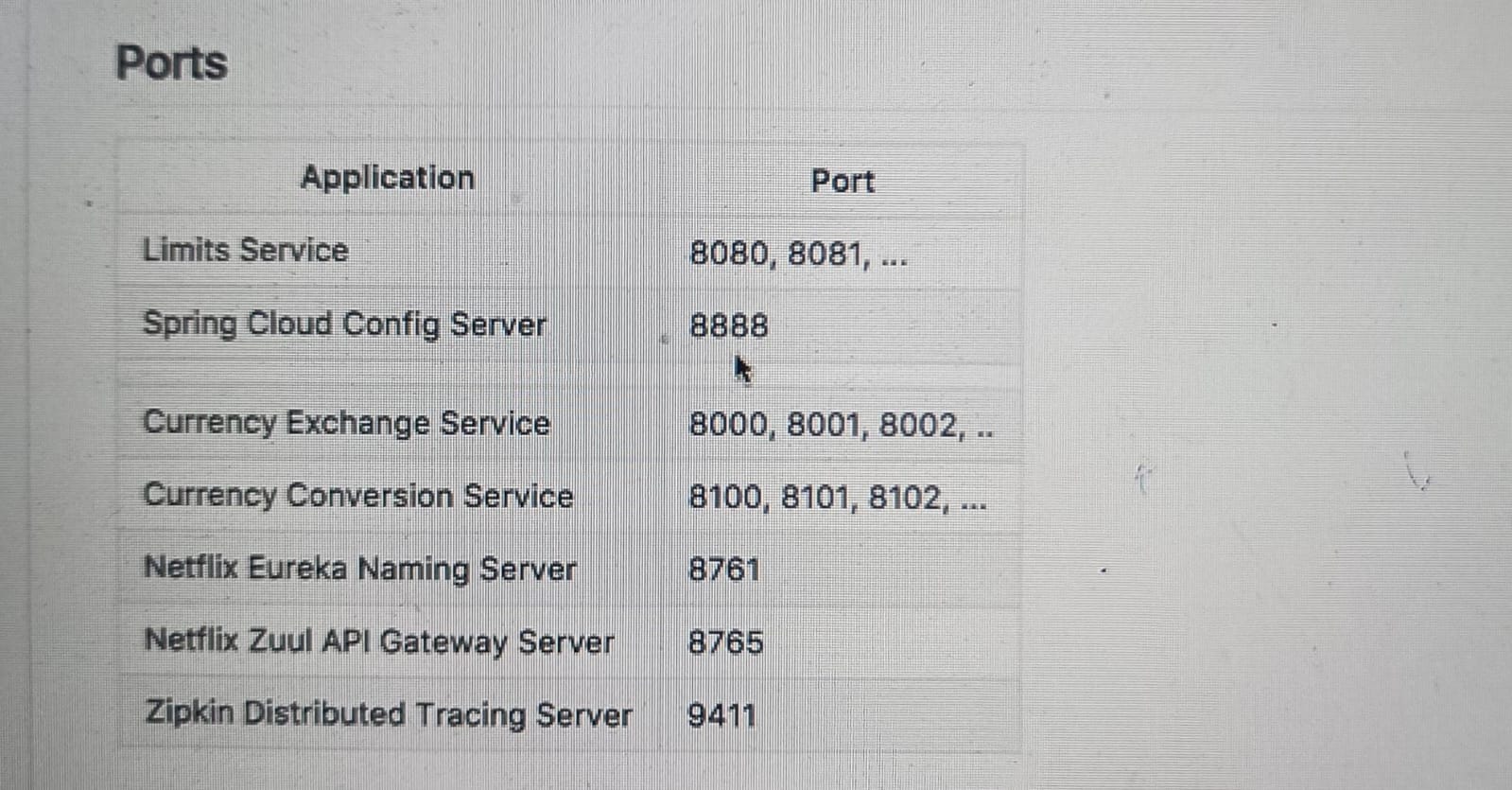
We will also implement FAULT TOLERANCE using Hystrix, if a service is down Hystrix helps us to configure a default response.

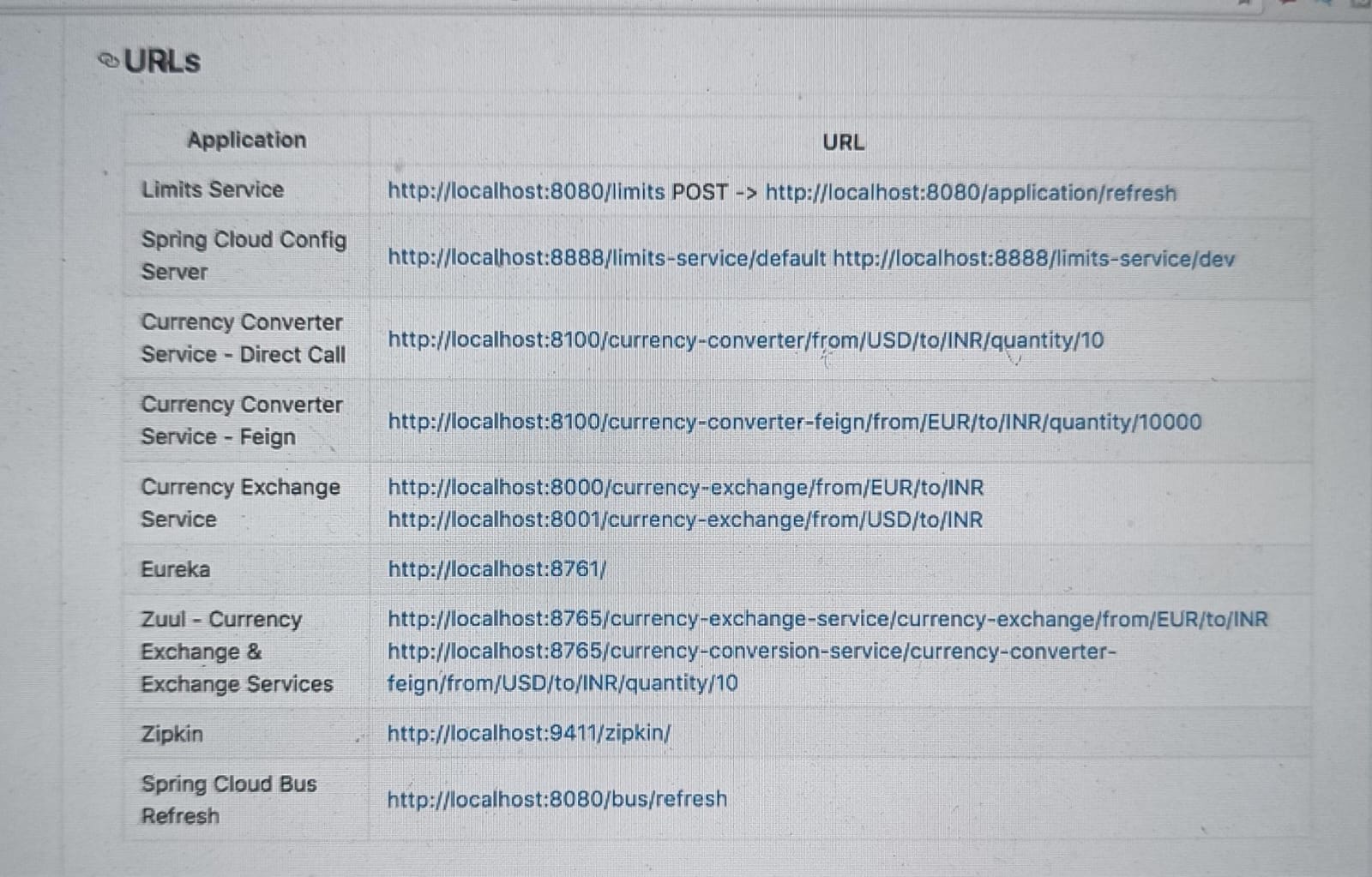
**Advantage Of Microservices –**

1. It enables you to ADAPT NEW TECHNOLOGY AND PROCESSES VERY EASILY – when we build application with the combination of microservices which can communicate each other using simple messages each of this microservices can be built in different technologies, in the typical monolith applications we would not have that flexibility for example microservice1 might be in java microservice might be in nodejs microservice3 might be in Kotlin and tomorrow there might be a language xyz which is really doing well and which provides a lot of benefits to you, and you can easily create a microservice in that specific language and also for the new microservice that we create we can bring in new processes as well.
2. The other important advantage of microservices is dynamic scaling, consider an online shopping application like amazon they don’t really have the same amount of load or same amount or traffic or same amount of users throughout the year especially during the holiday season the load on the application will be a lot and during the rest of the year there might not be so much load, during the black Friday there might be a huge amount of load if your microservices are cloud enabled they can scale dynamically and you can procure hardware and release it dynamically as well. So you can scale up and scale down application based on the load.
3. Because you are developing smaller components its much easier to release microservices compared to monolith applications, this means you can bring new features faster to market and that’s the big advantage you have in the modern world .

Microservice Components – Standardizing Ports and URL

One of the important things in the microservice that we would be develop lot of components, we will be installing at least seven different projects and therefore its very important to standardize the ports on which we would run these applications.

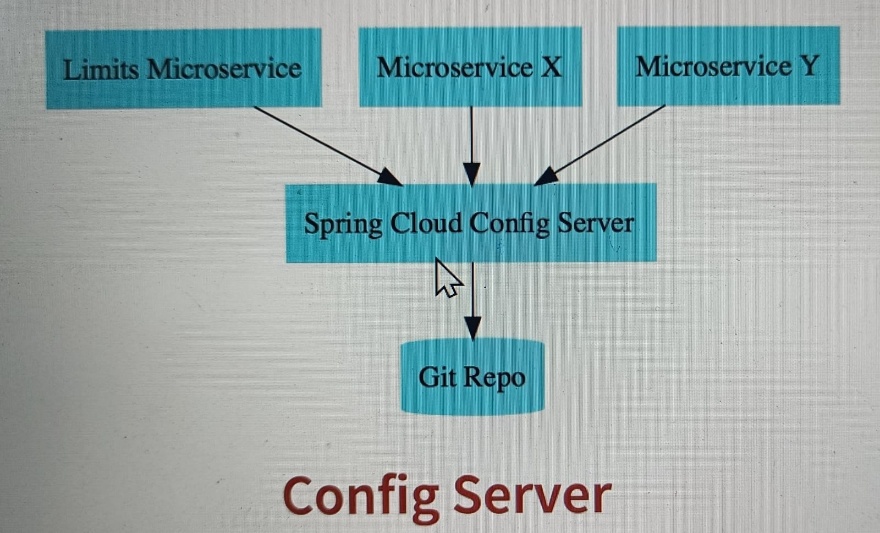




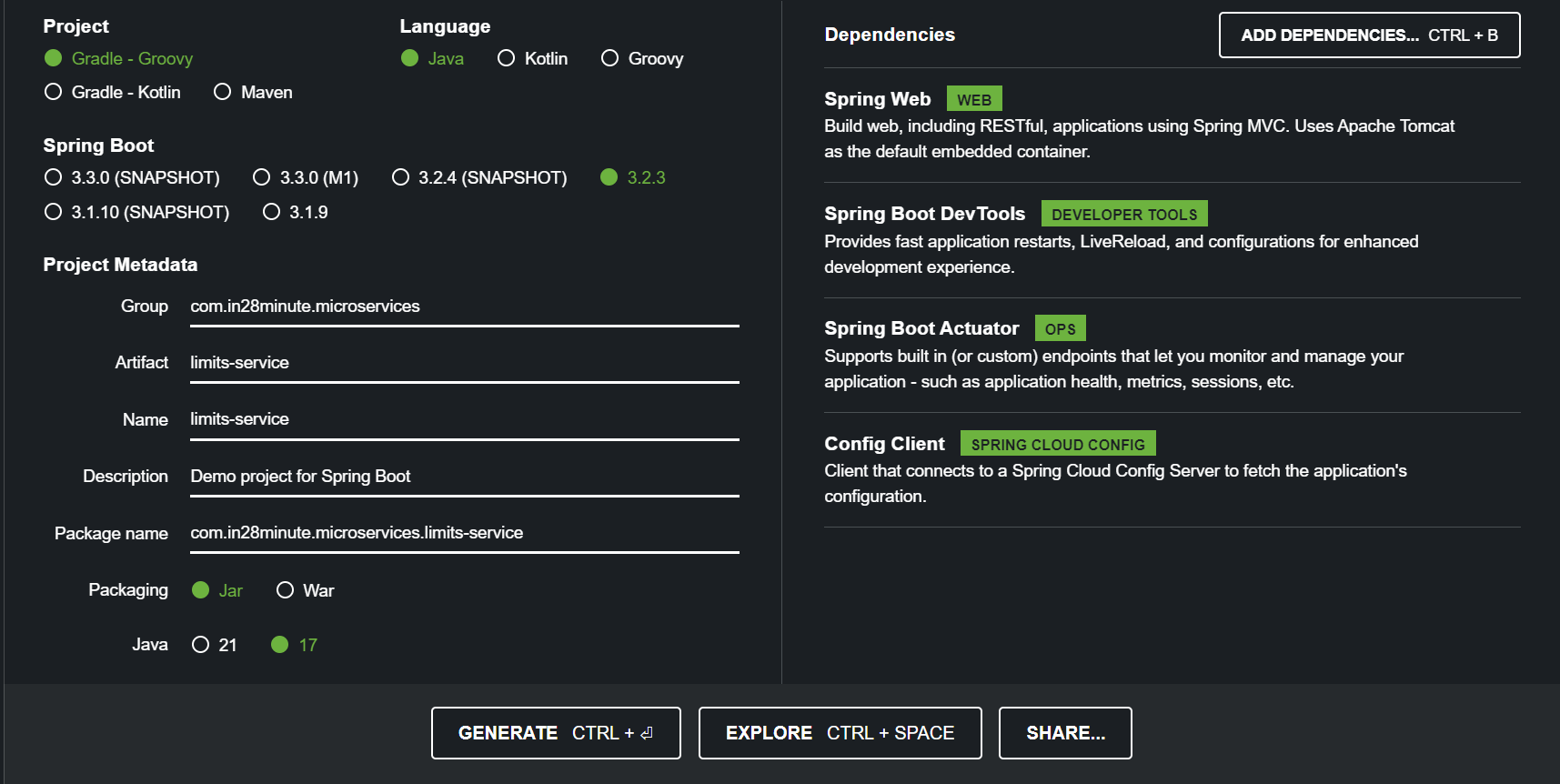
**Microservices – V2 What’s New**

* Microservices Evolve Quickly
* Important Updates:
  + Latest Versions of Spring Boot and Spring Cloud
  + Spring Cloud LoadBalancer instead of Ribbon in the earlier version of spring boot and spring cloud version 2.3 ribbon is the load balancer
  + We will be using Spring Cloud Gateway instead of Zuul
  + Resillience4j instead of Hystrix
* In the next section we would be use Docker to containerize the microservices
  + We will be run the microservices using docker and the docker compose
* And in the next section we will be jump in to Kubernetes – Orchestrate all your Microservices with Kubernetes.

**Setting up Limits Microservice – V2**



We will create a microservice which is a limits microservice, and then will set up a spring cloud config server which would be connected to a git repository lets get started with setting up the limits microservice.



Now import the maven project into the eclipse.

One of the recommendations is that, not to have any spaces in the folder path in the project properties. Because spaces can cause problem to you microservices especially when you use the spring cloud config server.

Before we go further there is one simple configuration that we need to make in our application.properties, when we add spring-cloud-starter-config dependency in to the pom.xml file what we need to do is configure how spring cloud starter config needs to connect to spring cloud config server.

spring.config.import=optional:configserver:http://localhost:8888

write now we do not have config server and that is the reason why we would say this is optional for now. Once the config server is available then you would see that the limit service automatically talk to that limit server.

After all the configuration now you can start the limits service application server.

**LimitsController-**

@RestController

**public** **class** LimitsController {

@GetMapping("/limits")

**public** Limits retrieveLimits() {

**return** new Limits(1, 1000);

}

}

**Limits –**

Create it in the com.in28minute.microservice.bean package

**public** **class** Limits {

**private** **int** minimum;

**private** **int** maximum;

}

Create constructor and getter setter methods.

Output –

http://localhost:8085/limits

{

"minimum": 4,

"maximum": 996

}

In next step we will use application configuration means first we read the values from the application.properties file and then we would actually connect it to a centralised configuration.

**LimitsController-**

@RestController

**public** **class** LimitsController {

@Autowired

**private** Configuration configuration;

@GetMapping("/limits")

**public** Limits retrieveLimits() {

**return** **new** Limits(configuration.getMinimum(),configuration.getMaximum());

//return new Limits(1, 1000);

}

}

**Application.Properties –**

limits-service.minimum=2

limits-service.maximum=998

**Configuration –**

Create the Configuration class inside the com.in28minute.microservice.configuration package

@Component

@ConfigurationProperties("limits-service")

**public** **class** Configuration {

**private** **int** minimum;

**private** **int** maximum;

}

Create the constructor and the getter setter methods.

Output –

http://localhost:8085/limits

{

"minimum": 2,

"maximum": 998

}

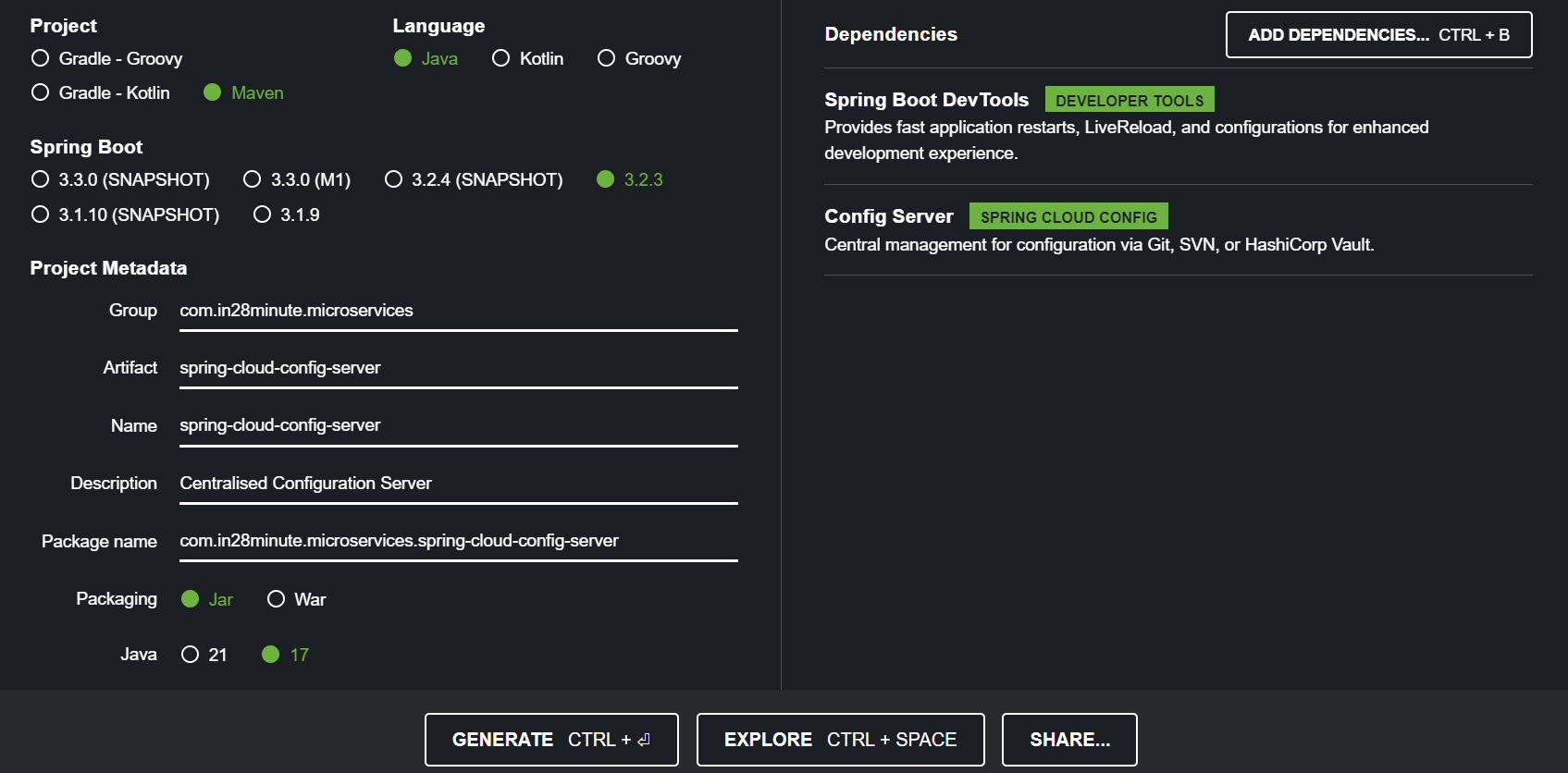
Now we can see our application is picking up the values from the application.properties file.

In the next step we would want to connect our limits service to a centralised configuration server.

**Setting up Spring Cloud Config Server –**

Create the new project for spring cloud config server using spring initializer

In the previous step we setup the limits service, what we want to do now is that we want to set up the spring cloud config server and connect it to a Git Repository.



Now import the maven project into the eclipse.

Now we have to assign a standardized port , so we have to assign a port 8888 to a spring cloud config server.

**Application.Properties –**

spring.application.name=spring-cloud-config-server

server.port=8888

After all the configuration now, you can start the limits service application server.

**Installing Git and Creating Local Gite Repository**

Now open the command prompt or terminal and check if the git is installed or not.

git –version

Now we will cd to the folder where we want to create the git repository, my present working directory is

D:/CODE/springMicroservices

And over here I actually create the new directory

mkdir git-localconfig-repo

make sure that you do not have any spaces in the entire path.

cd git-localconfig-repo

and over here we want to create our git repository so we initialize it here

git init

so, this will initialize the empty git repository in this specific folder, we would want to store all our configuration files in here in this git-localconfig-repo, so you can open up any text editor and go over to that specific folder, I would use visual studio code which is one of text editor that I have in here.

And over here I will create a new file

**Limits-service.properties –**

Limits-service.minimum=4

Limits-service.maximum=996

Now that I have a limits-service.properties file created, now go ahead and commit the changes in the local git repository.

Now we want to add this file into the git repository

git add \*

now I can say

git commit -m “adding limits-service.properties”

**Connect Spring Cloud Config Server to Local Git Repository**

Go to the application.Properties of the spring cloud config server and configure the folder.

**Application.Properties –**

spring.cloud.config.server.git.uri = file:///D:/CODE/springMicroservices/git-localconfig-repo

before we run the spring cloud config server we need to add another annotation to the SpringCloudConfigServerApplication and that is @EnableConfigServer

**SpringCloudConfigServerApplication-**

@EnableConfigServer

@SpringBootApplication

**public** **class** SpringCloudConfigServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringCloudConfigServerApplication.**class**, args);

}

}

Now run this application and go to chrome and type below url

<http://localhost:8888/limits-service/default>

now we have connected our spring cloud config server to the git repository

**Connect Limits Service to Spring Cloud Config Server –**

In the last step we have connected or config server with the git repository now we want to connect our limits server **with** the spring cloud config server, this would complete our chain.

There are 2 very important configurations that we have already made which are important to connect limits service to the spring cloud config server.

The first one is to add a dependency of a spring-cloud-starter-config how does this came, while the project creation step we added the Config Client dependency.

And where do we configure the url of the cloud config server? We have already done that in the application.properties file of the limits service.

If you would want to make the connection with the config server then you can remove the optional from the url.

**Configuring Profiles for Limits Service**

Let us say there are multiple environments for Limits Service, like dev, prod, qa then how do you store separate configurations for these environments and how do you make use of them using spring cloud config server lets look at that.

The first thing that we will start with is how do you store configurations related to multiple environments in a GitHub repository, earlier we created the limits-service.properties in the git local config repo, what I do now I will create multiple copies of it.

Like – limits-service-dev.properties

limits-service-qa.properties

This is how actually you can create property files for different environments.

Now configure the different values for the different environments –

limits-service-dev.properties –

limits-service.minimum=5

limits-service.maximum=995

Now we would want to pick these values from the config server –

<http://localhost:8888/limits-service/dev>

So now we have the values for multiple environments, picked up from the github repository how do we make the application take the values from the config server?

The way you can do that actually going into application.properties file of the limit service, typically whenever you want to support multiple environments in a spring application you would go for a concept called profiles.

**Application.Properties –**

Spring.profiles.activce = dev

Spring.cloud.config.profile = dev

Now you can check the values at the url -

Output –

http://localhost:8085/limits

{

"minimum": 5,

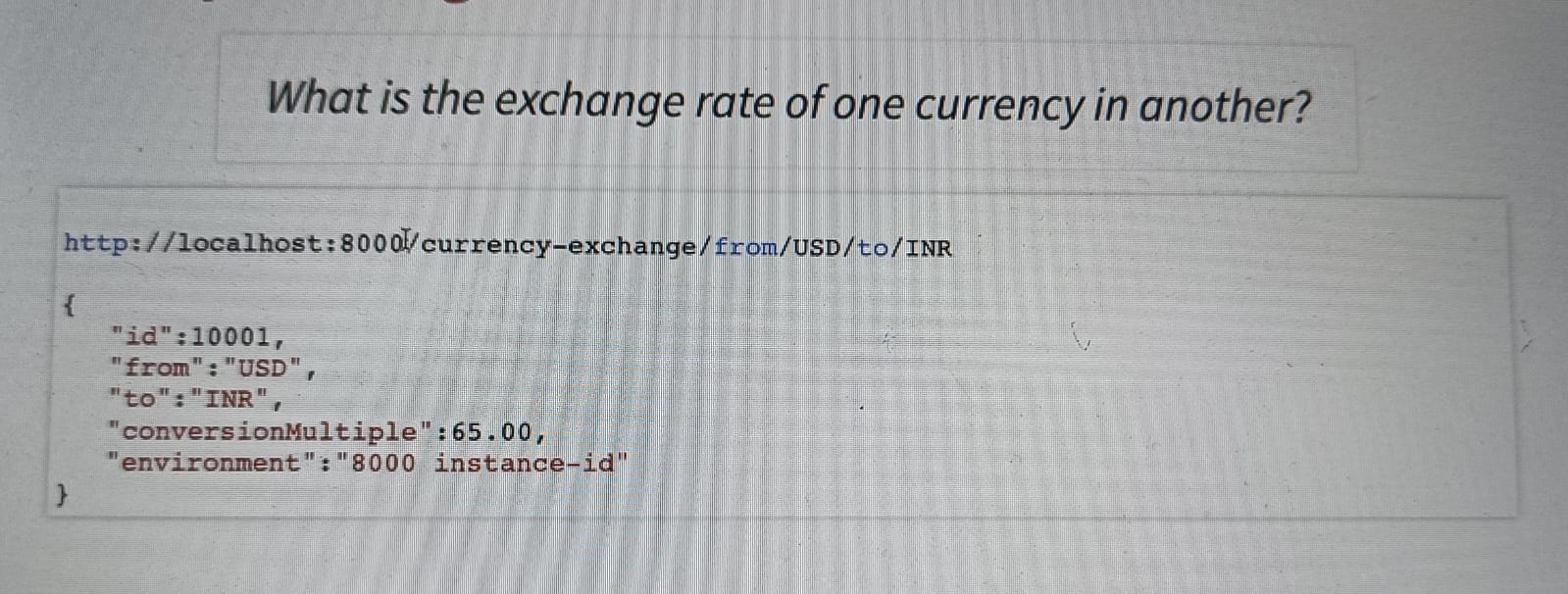
"maximum": 995

}

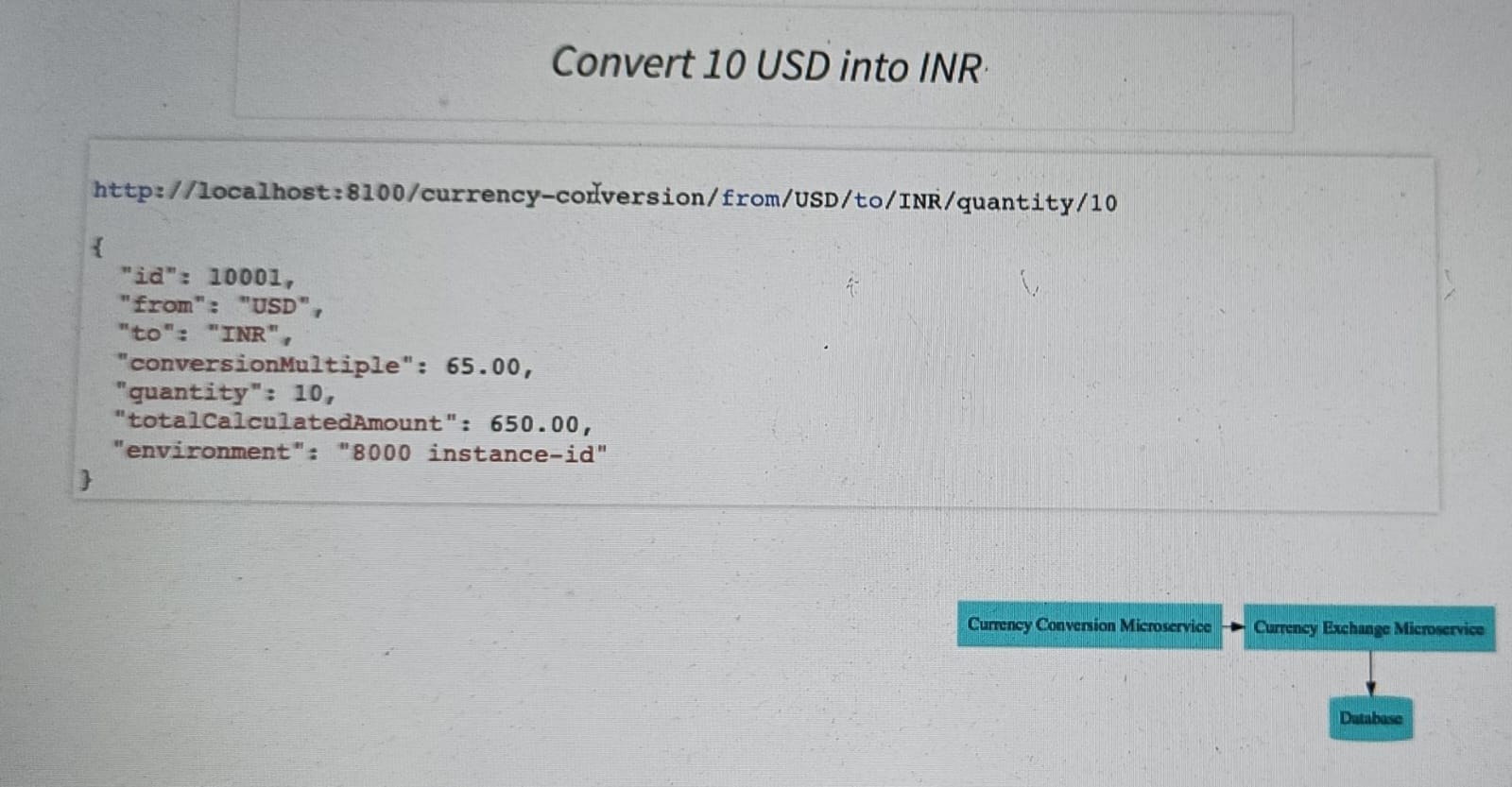
**Introduction to Currency Conversion & Exchange Microservices –**

In the next few steps we would want to play with api gateway, distributed tracings, circuit backers and a lot of other things, and to be able to play with these we need a couple more microservices.

From the currency exchange microservice we will be exposing a simple url –

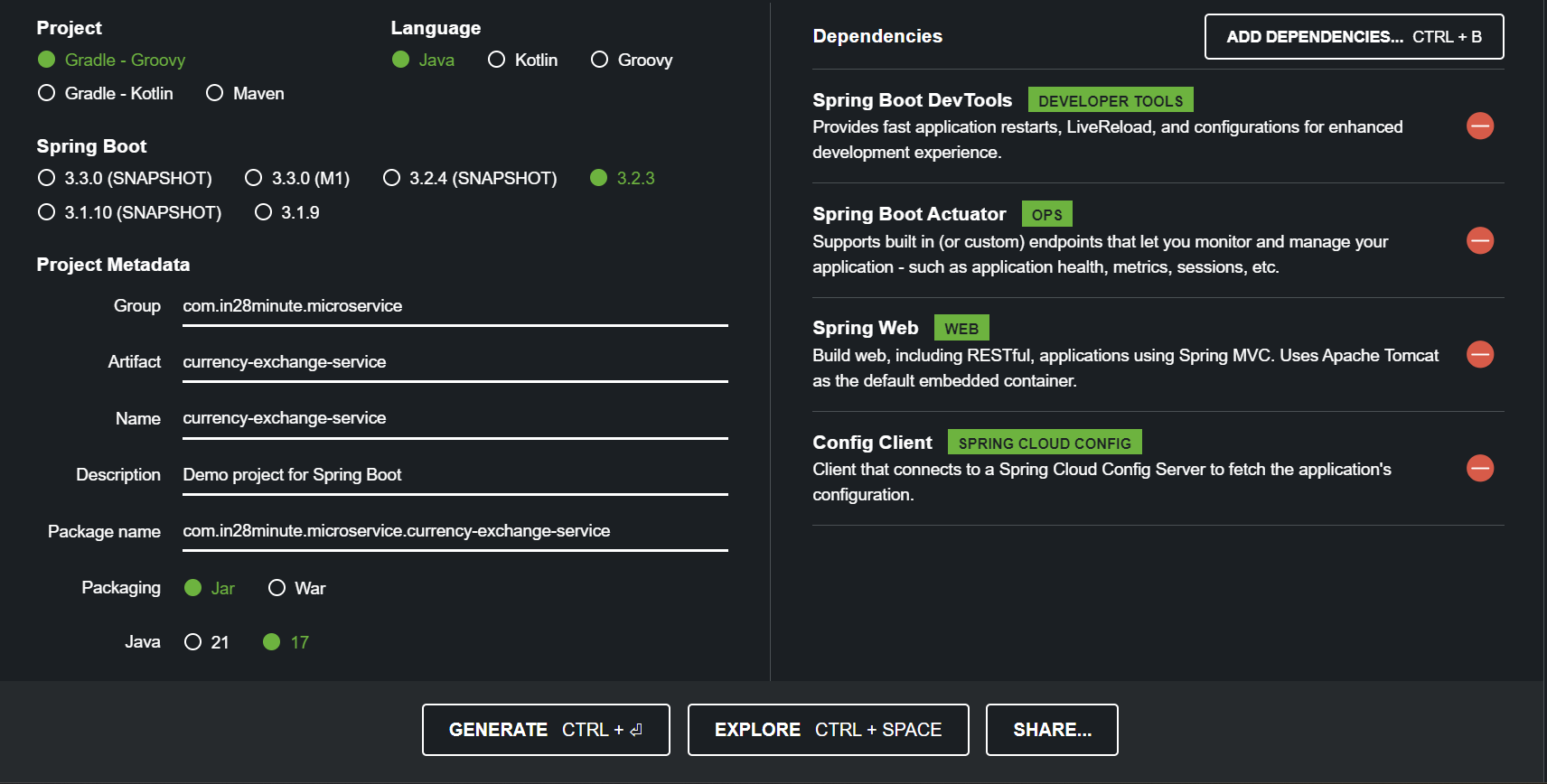


The other microservice we would want to setup is the currency conversion microservice-



This is responsible for 10 USD into INR, for the currency conversion microservice to provide its features what it would do it will call the currency Exchange microservice, so the currency conversion microservice call the currency exchange microservice and ask what is the value of USD in INR for today and then it takes the return value that would be 65, and it would take it multiply with it 10 and it will return the response back.

**Setting up Currency Exchange Microservice –**



Now import the maven project into the eclipse.

Now we have to assign a standardized port , so we have to assign a port 8000 for the currency-exchange-service.

**Application.Properties –**

spring.application.name=currency-exchange

server.port=8000

**Create a simple hard coded currency exchange service –**

**CurrencyExchangeController –**

@RestController

**public** **class** CurrencyExchangeController {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyExchange retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

CurrencyExchange currencyExchange = new CurrencyExchange(1000L,from, to, BigDecimal.valueOf(50));

**return** currencyExchange;

}

}

**CurrencyExchange –**

**public** **class** CurrencyExchange {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal conversionMultiple;

}

Create the constructor and the getter setter methods.

Now run this application and go to chrome and type below url

<http://localhost:8000/currency-exchange/from/USD/to/INR>

output –

{

Id: 1000,

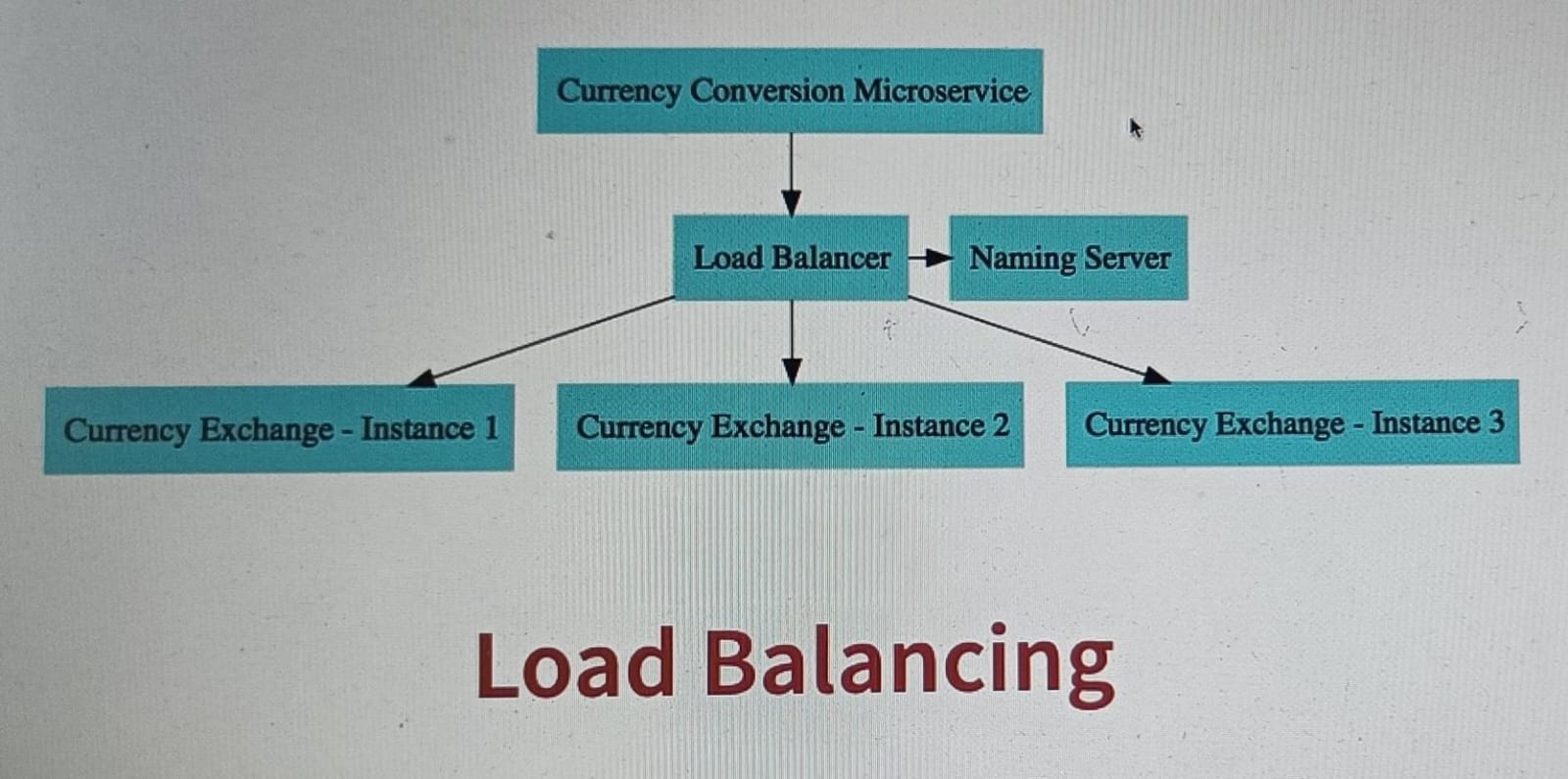
From: “USD”,

To: “INR”,

conversionMultiple: 50,

}

**Setting up Dynamic Port in the Response –**



In next few steps you will see that we would have multiple instances of currency conversion microservice, and the currency exchange microservice, and when I am calling from currency conversion microservice I would need to know which instance of the currency exchange microservice providing the response.

**CurrencyExchange –**

**public** **class** CurrencyExchange {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal conversionMultiple;

**private** String environment;

}

Create the constructor and the getter setter methods.

**CurrencyExchangeController –**

@RestController

**public** **class** CurrencyExchangeController {

@Autowired

**private** Environment environment;

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyExchange retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

CurrencyExchange currencyExchange = new CurrencyExchange(1000L,from, to, BigDecimal.valueOf(50));

String port = environment.getProperty("local.server.port");

currencyExchange.setEnvironment(port);

**return** currencyExchange;

}

}

<http://localhost:8000/currency-exchange/from/USD/to/INR>

output –

{

Id: 1000,

From: “USD”,

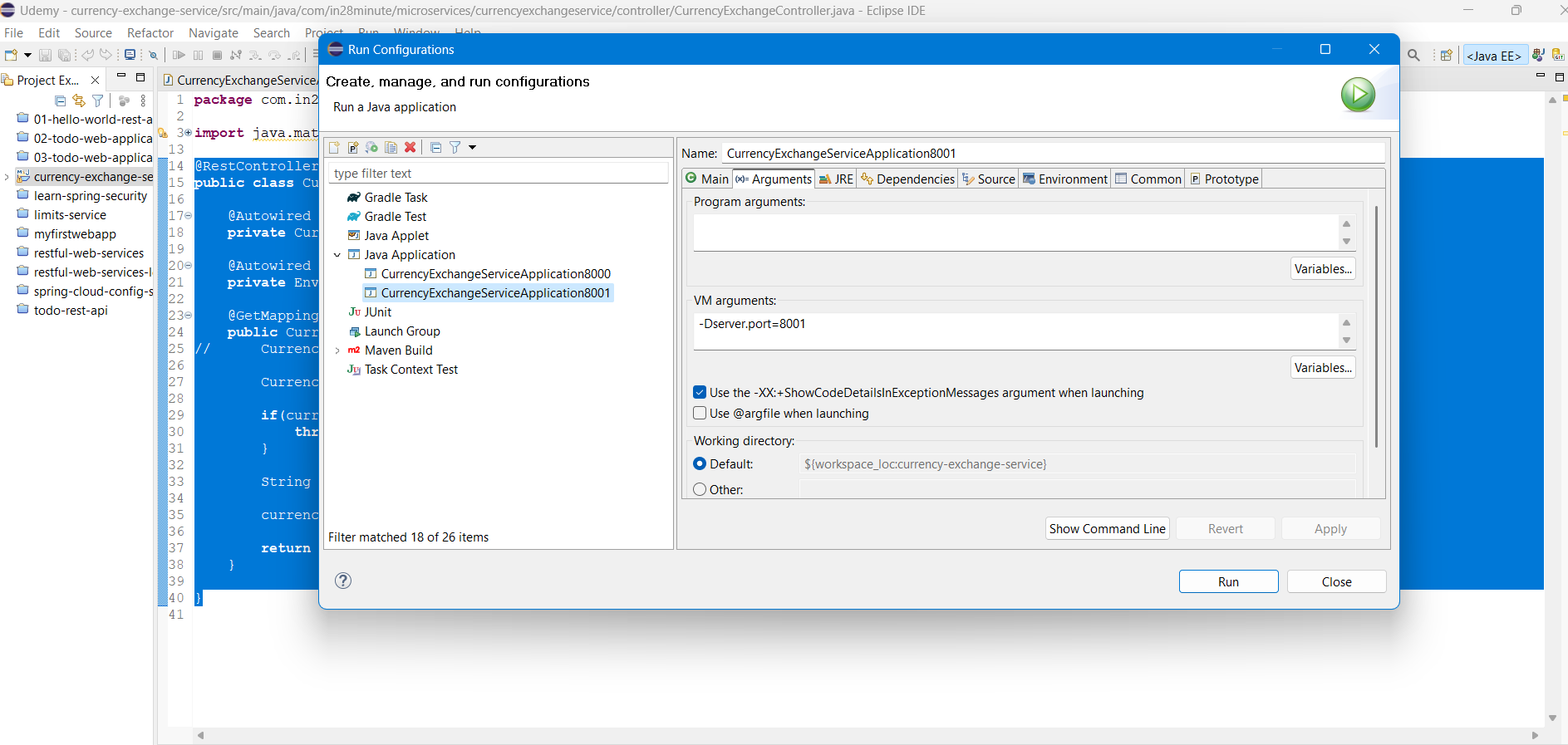
To: “INR”,

conversionMultiple: 50,

environment: “8000”

}

Now our application returning a port which is 8000 in response, now I want to run 2nd instance of this application on the port 8001, so I will go to the and right click on the project and click on the run Configurations, and inside the application I will create a duplicate copy of the same application and for the duplicate application in the arguments tab I will provide the port 8001.



**Configure JPA and Initialized Data –**

Right now, we have hard coded data, lets now fetch data from the database and for this we need to add the dependences in the pom.xml file.

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

Now stop the server and restart it again.

Now go to the application.properties and configure the database configurations –

**Application.Properties –**

spring.jpa.show-sql=true

spring.datasource.url=jdbc:h2:mem:testdb

spring.h2.console.enabled=true

#TO LOAD THE DATA AFTER THE TABLE CREATION

spring.jpa.defer-datasource-initialization=true

Url to go to the h2-console –

<http://localhost:8000/h2-console>

Now we need to create the table, for this we need to mark our class with the entity annotations.

**CurrencyExchange –**

@Entity

**public** **class** CurrencyExchange {

@Id

**private** Long id;

@Column(name="currency\_from")

**private** String from;

@Column(name="currency\_to")

**private** String to;

**private** BigDecimal conversionMultiple;

**private** String environment;

}

Now to the src/main/resources folder and create the new file as data.sql and write the insert query into the table.

**Data.sql**

insert into currency\_exchange(id,currency\_from,currency\_to,conversion\_multiple,environment)values(10001,'USD','INR',65,'');

insert into currency\_exchange(id,currency\_from,currency\_to,conversion\_multiple,environment)values(10002,'EUR','INR',75,'');

insert into currency\_exchange(id,currency\_from,currency\_to,conversion\_multiple,environment)values(10003,'AUD','INR',75,'');

**Create JPA Repository –**

Lets now update our rest api to connect to the database, and to be able to do that we need to create interface which extends JpaRepository.

**CurrencyExchangeRepository –**

**public** **interface** CurrencyExchangeRepository **extends** JpaRepository<CurrencyExchange, Long>{

CurrencyExchange findByFromAndTo(String from,String to);

}

**CurrencyExchangeController –**

@RestController

**public** **class** CurrencyExchangeController {

@Autowired

**private** CurrencyExchangeRepository repository;

@Autowired

**private** Environment environment;

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyExchange retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

//CurrencyExchange currencyExchange = new CurrencyExchange(1000L,from, to, BigDecimal.valueOf(50));

CurrencyExchange currencyExchange = repository.findByFromAndTo(from, to);

**if**(currencyExchange == **null**) {

**throw** **new** RuntimeException("Unable to find data for " + from + " to " + to);

}

String port = environment.getProperty("local.server.port");

currencyExchange.setEnvironment(port);

**return** currencyExchange;

}

}

Output -

<http://localhost:8000/currency-exchange/from/USD/to/INR>

{

"id": 10001,

"from": "USD",

"to": "INR",

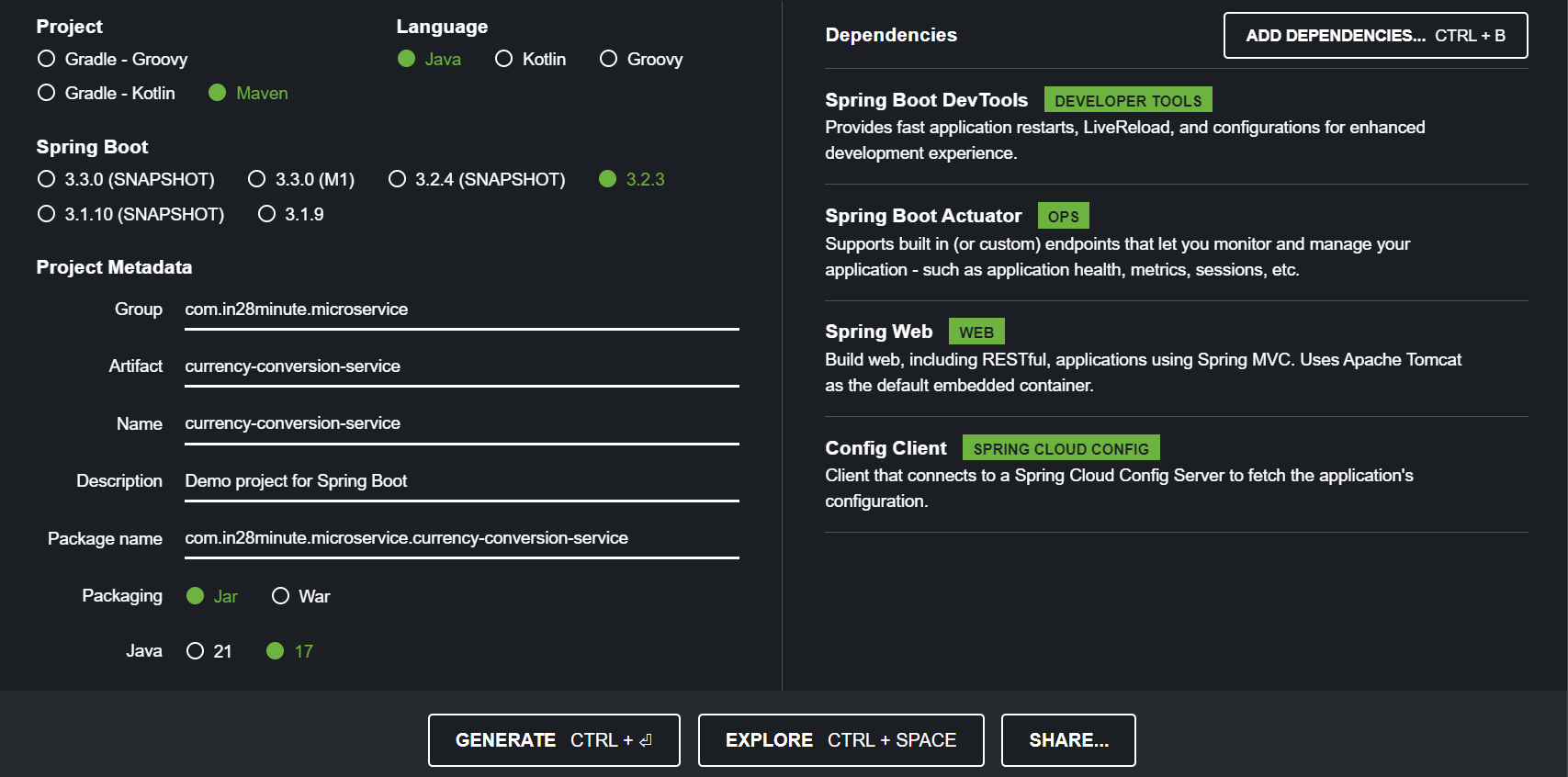
"conversionMultiple": 65.00,

"environment": "8000"

}

**Setting up Currency Conversion Microservice –**

We have now set up currency exchange microservice and we have created a simple rest Api, and now we would shift our attention to currency conversion microservice.



Now import the maven project into the eclipse.

Now we have to assign a standardized port , so we have to assign a port 8100 for the currency-conversion-service.

**Application.Properties –**

spring.application.name=currency-conversion

server.port=8100

**Creating a service for currency conversion –**

**CurrencyConversionController –**

@RestController

**public** **class** CurrencyConversionController {

@GetMapping("/currency-conversion/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversion calculateCurrencyConversion(

@PathVariable String from,

@PathVariable String to,

@PathVariable BigDecimal quantity ) {

**return** **new** CurrencyConversion(10001L,from,to,quantity,BigDecimal.ONE, BigDecimal.ONE,””);

}

}

**CurrencyConversion –**

**public** **class** CurrencyConversion {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal quantity;

**private** BigDecimal conversionMultiple;

**private** BigDecimal totalCalculatedAmount;

**private** String environment;

}

Create the constructors and the getter setters.

Output –

<http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10>

{

"id": 10001,

"from": "USD",

"to": "INR",

"quantity": 10,

"conversionMultiple": 1,

"totalCalculatedAmount": 1,

"environment": ""

}

**Invoking Currency Exchange from Currency Conversion Microservice –**

Let us see how to call the currency conversion microservice to a currency exchange microservice.

**CurrencyConversionController –**

@RestController

**public** **class** CurrencyConversionController {

@GetMapping("/currency-conversion/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversion calculateCurrencyConversion(

@PathVariable String from,

@PathVariable String to,

@PathVariable BigDecimal quantity ) {

HashMap<String, String> uriVariables = **new** HashMap<>();

uriVariables.put("from", from);

uriVariables.put("to", to);

ResponseEntity<CurrencyConversion> responseEntity = **new** RestTemplate().getForEntity

("http://localhost:8000/currency-exchange/from/{from}/to/{to}",

CurrencyConversion.**class**,uriVariables);

CurrencyConversion currencyConversion =responseEntity.getBody();

**return** **new** CurrencyConversion(currencyConversion.getId(),from,to,quantity,currencyConversion.getConversionMultiple(),quantity.multiply(currencyConversion.getConversionMultiple()),currencyConversion.getEnvironment());

}

}

Output –

<http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10>

{

"id": 10001,

"from": "USD",

"to": "INR",

"quantity": 10,

"conversionMultiple": 65.00,

"totalCalculatedAmount": 650.00,

"environment": "8000"

}

**Using Feign REST Client for Service Invocation –**

In the last step we have to write a lot of tedious code around REST template to get the current conversion service to talk with the currency exchange microservice, to make as simple rest api call we need to write about 20 lines of code, and Imagin in a microservices architecture you have a 100s of microservices, they would be calling each other and you need to write this kind of code everywhere and that’s where spring cloud provides you a framework called FEIGN.

FEIGN make it easy to call other microservices, and to make use of feign we need to add specific dependency into currency conversion service.

**Pom.xml –**

<dependency>

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

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

</dependency>

After adding the dependency, you need to restart the server.

Now we need to talk with the currency exchange service from the currency conversion controller, and to be able to do that we need to create a proxy interface.

**CurrencyExchangeProxy-**

@FeignClient(name="currency-exchange", url="localhost:8000")

**public** **interface** CurrencyExchangeProxy {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversion retrieveExchangeValue(@PathVariable String from, @PathVariable String to);

}

Now we will make use this proxy in the CurrencyConversionController –

**CurrencyConversionController-**

@RestController

**public** **class** CurrencyConversionController {

@Autowired

**private** CurrencyExchangeProxy proxy;

@GetMapping("/currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversion calculateCurrencyConversionFeign(

@PathVariable String from,

@PathVariable String to,

@PathVariable BigDecimal quantity ) {

CurrencyConversion currencyConversion = proxy.retrieveExchangeValue(from, to);

**return** **new** CurrencyConversion(currencyConversion.getId(),from,to,quantity,currencyConversion.getConversionMultiple(),quantity.multiply(currencyConversion.getConversionMultiple()),currencyConversion.getEnvironment());

}

}

Output –

<http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/10>

{

"id": 10001,

"from": "USD",

"to": "INR",

"quantity": 10,

"conversionMultiple": 65.00,

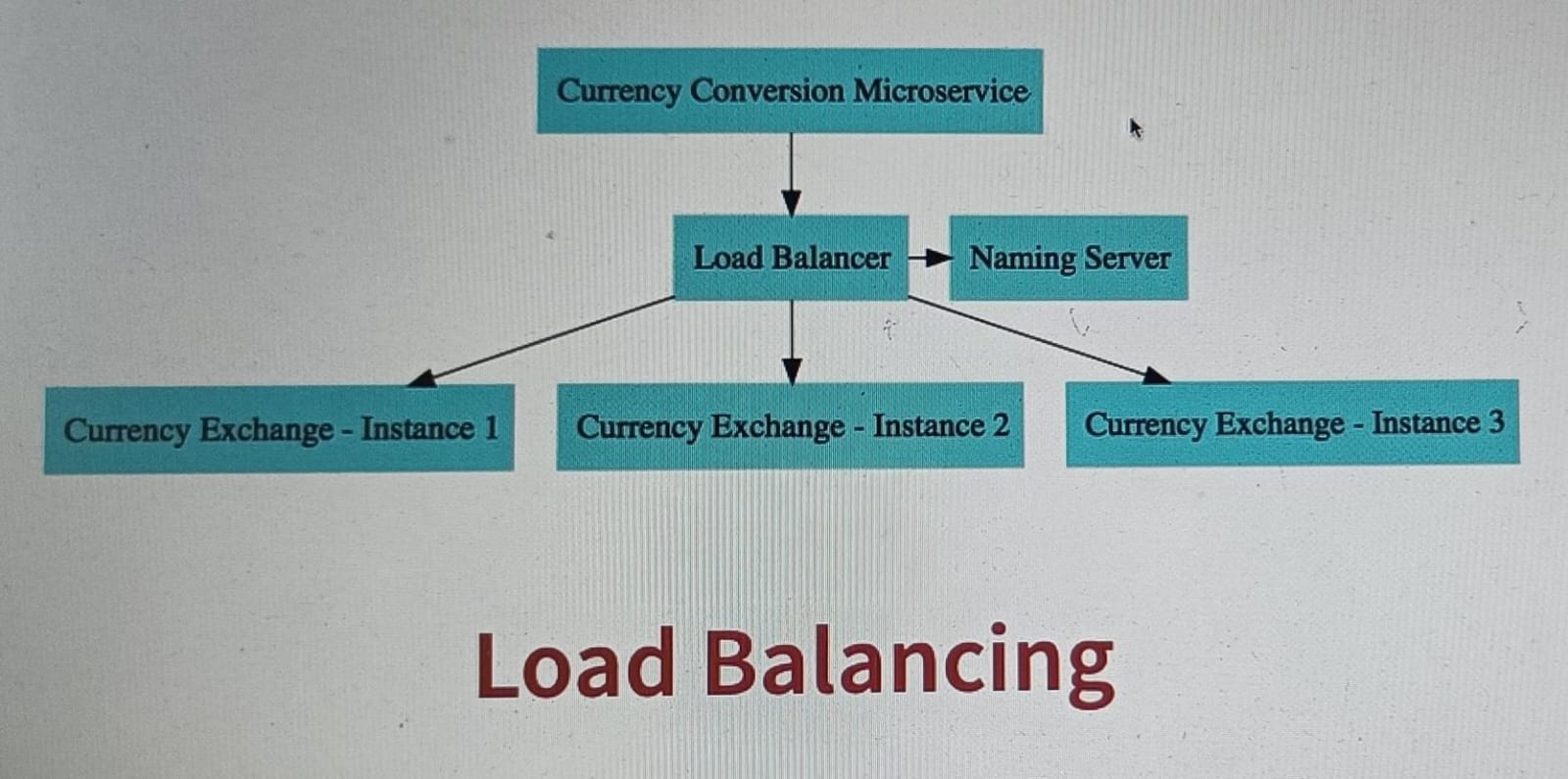
"totalCalculatedAmount": 650.00,

"environment": "8000"

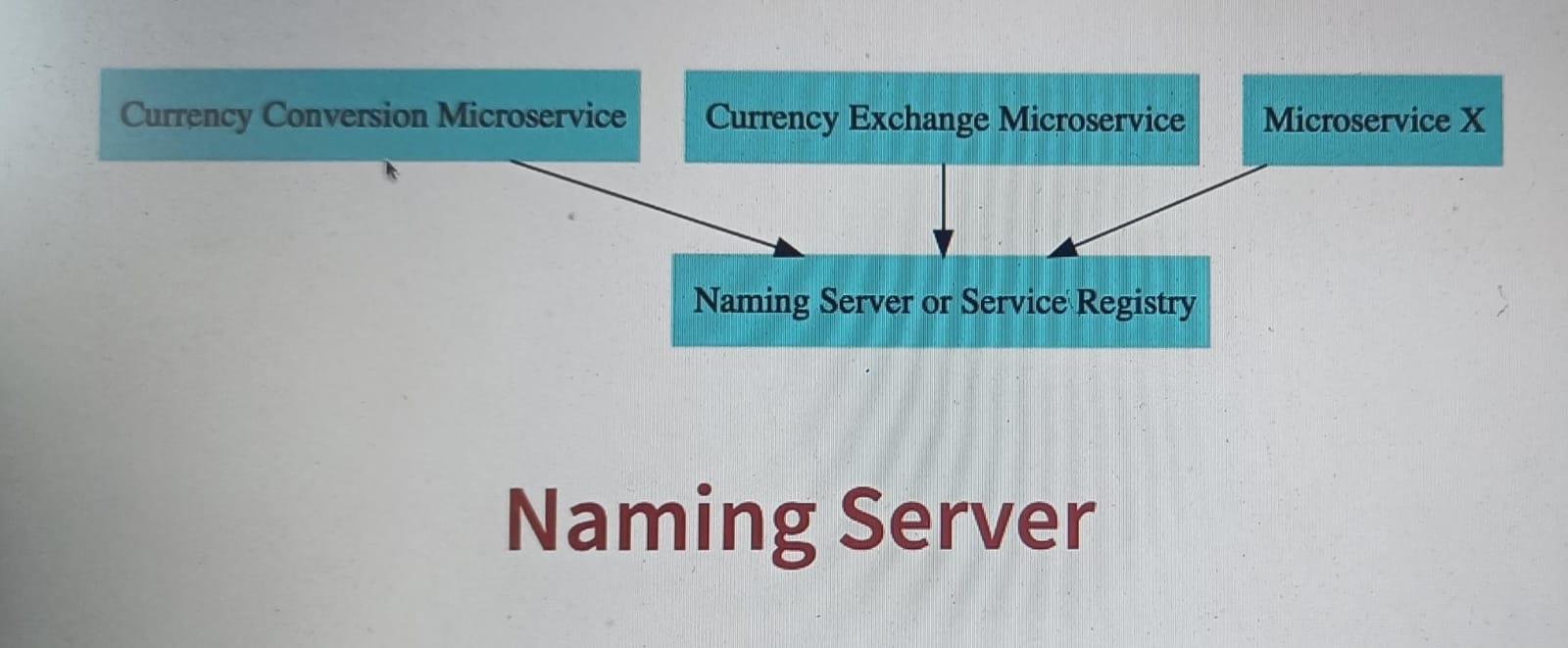
}

**Understand Naming Server and Setting up Eureka Naming Server –**

In the CurrencyExchangeProxy we are hardcoding the url of the currency exchange service, so if I would want to talk with the different instance of the currency exchange service want do I need to do I need to go into proxy interface and change the url, 8000,8001 and so on.

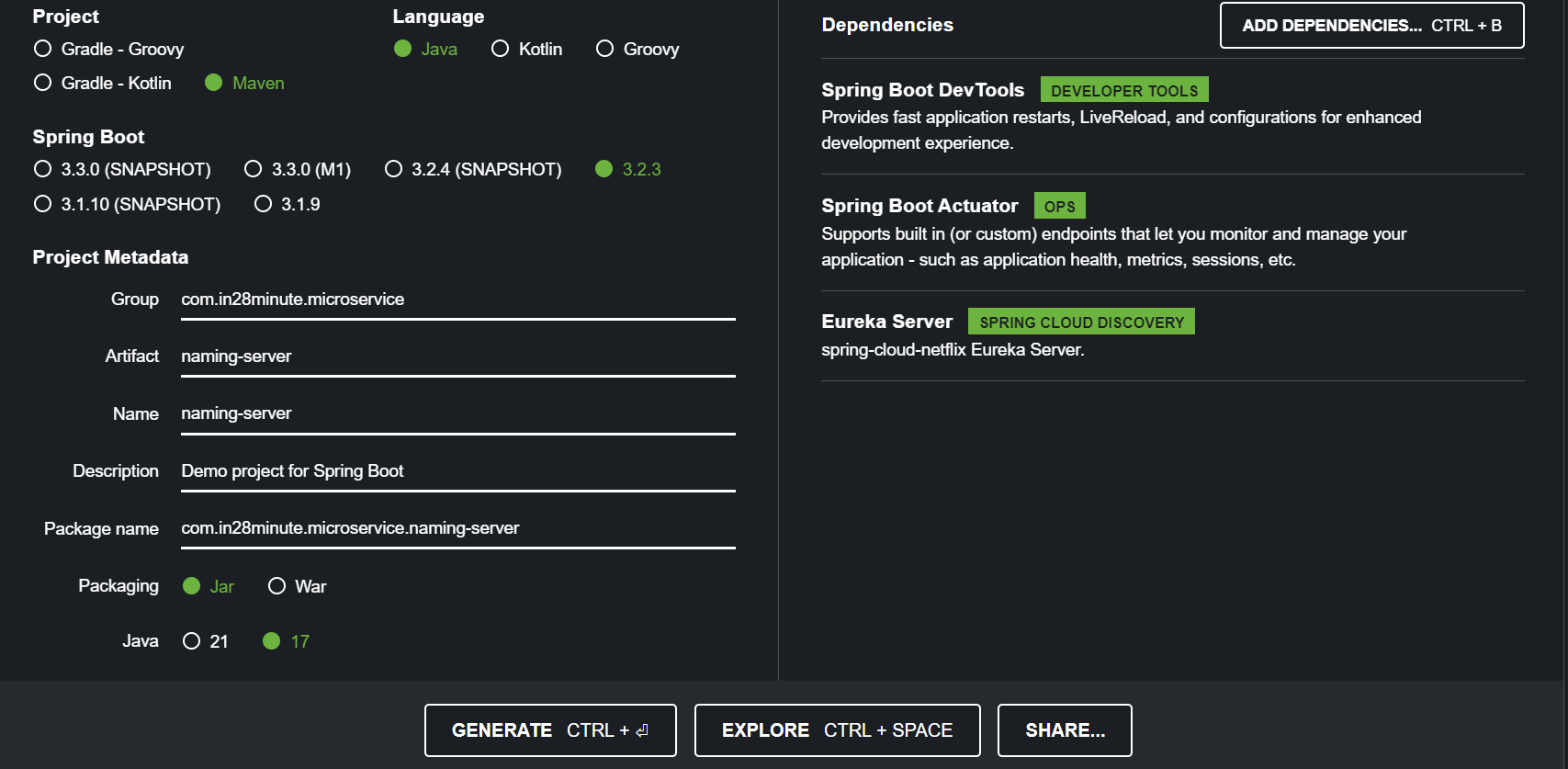
Bu we would want to able to do dynamically launch currency exchange instances and distribute load between them 

Right now lets say 1st instace on port 8000, 2nd is on 8001 and 3rd is on 8002 as instances come uo and go down we would want to automatically discovered them and load balance between them and that’s the reason why we go for service registry or a naming server.



What would happen in a microservice architecture, all the instances of all the microservices would register with service registry so the currency conversion microservice will register with service registry, currency exchange microservice will register with service registry, and all the other microservices also register with service registry, and lets say currency conversion microservice wants to talk with currency exchange microservice then it would ask service registry hey what are the addresses of the currency exchange microservice, the naming server will return out these addresses and then currency conversion microservice can talk with the currency exchange microservice.

Now let us create the Naming Server –



Now import the maven project into the eclipse.

Now we have to assign a standardized port , so we have to assign a port 8100 for the currency-conversion-service.

**Application.Properties –**

spring.application.name=naming-server

server.port=8761

after adding the port, we need to add more configuration related to eureka which is usally recommended by eureka, we are creating a eureka server and we don’t want to register itself and to do that we need to add couple of properties.

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

after adding the configuiration into application.properties we need to add onw more annotation to the application file.

**NamingServerApplication-**

@EnableEurekaServer

@SpringBootApplication

**public** **class** NamingServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(NamingServerApplication.**class**, args);

}

}

<http://localhost:8761/>

**Connect Currency Conversion & Currency Exchange Microservice –**

In the previous step we set up the naming server, in this step lets connect other microservice with the naming server.

And for this we just need to add one dependency into both currency conversion microservice and the currency exchange microservices.

**Pom.xml –**

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Now stop both the currency conversion and exchange application and restart it.

Next step lets try and load balance between multiple instances of the currency exchange from the currency conversion service.

**Load Balancing with Eureka, Feign and Spring Cloud LoadBalancer –**

**CurrencyExchangeProxy-**

//@FeignClient(name="currency-exchange", url="localhost:8000")

@FeignClient(name="currency-exchange")

**public** **interface** CurrencyExchangeProxy {

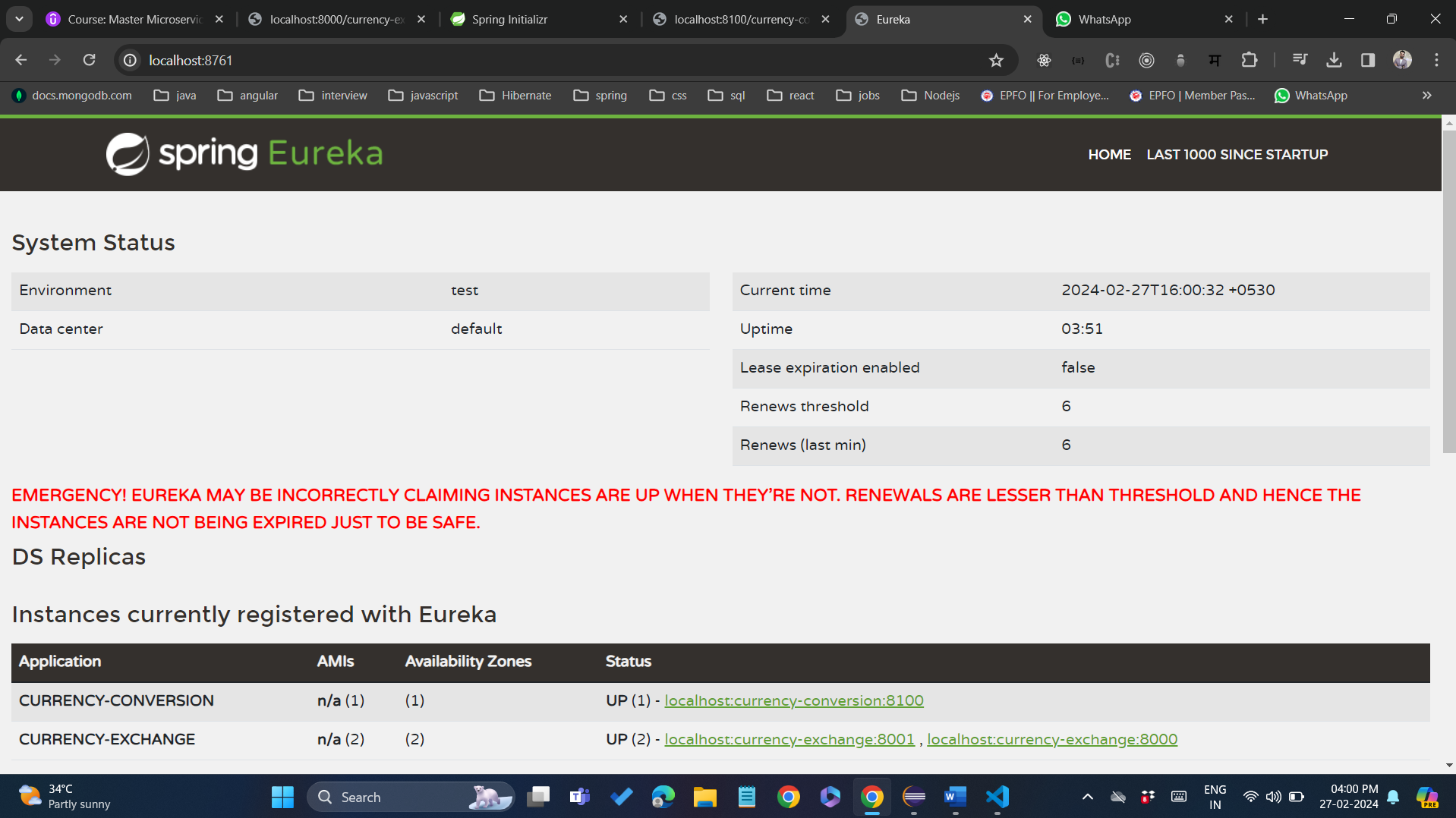
@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversion retrieveExchangeValue(@PathVariable String from, @PathVariable String to);

}

So we just want our feign client to talk to eureka and pickup the instances of currency exchange and do load balancing between them all that magic will happen just by removing the url.

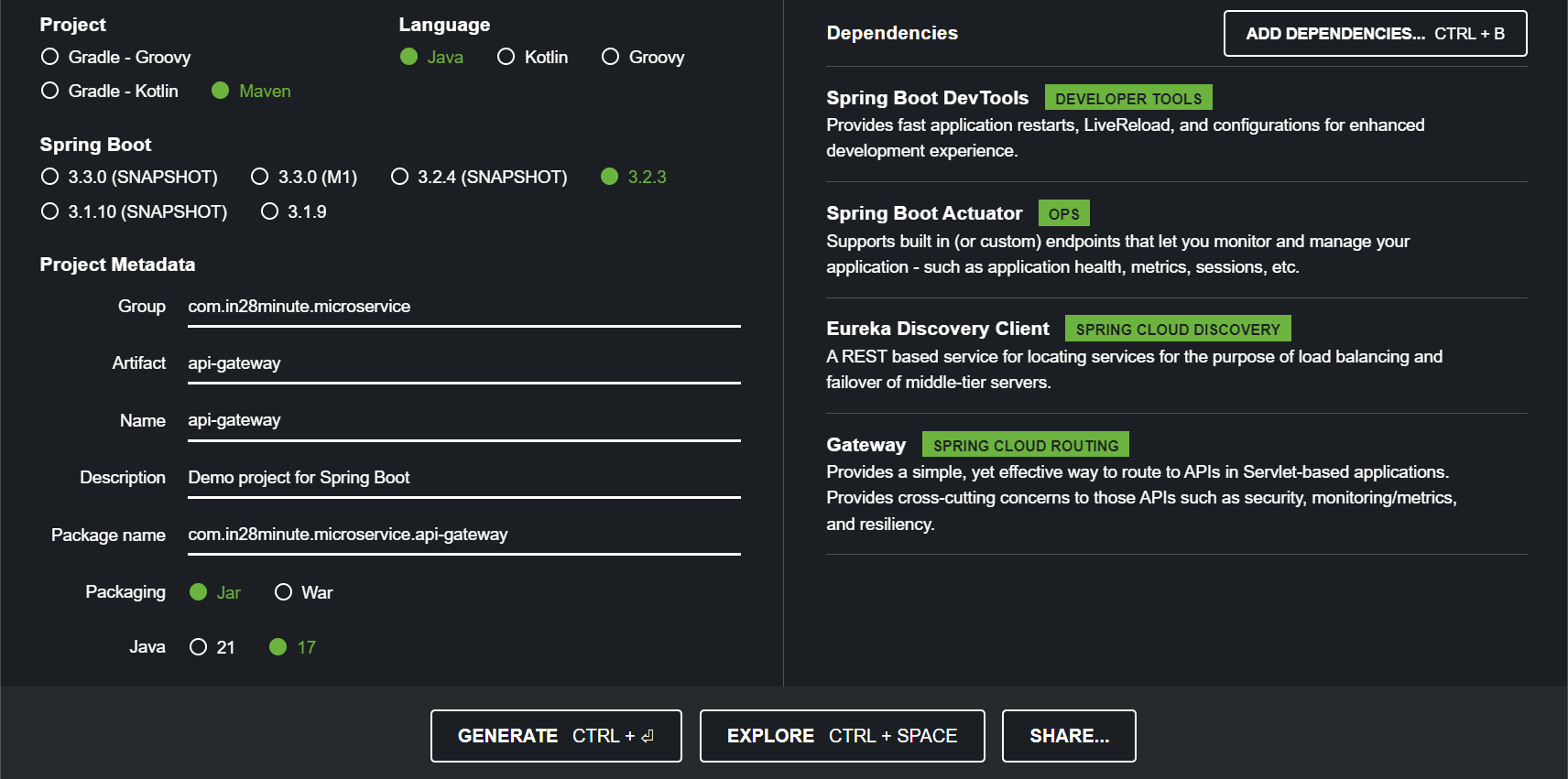
To check load balancing you just need to run currency exchange service on two different ports.



You can see here there are 2 instances of the currency exchange service are registered with the spring eureka naming server.

**Setting up Spring Cloud API Gateway –**

We implemented microservices, and these microservices have a lot of common features authentication, authorization, logging where do you implement all these common features that’s one of the typical questions in the microservice architecture and the typical solution is to go for an api gateway.



Now import the maven project into the eclipse.

Now we have to assign a standardized port , so we have to assign a port 8765 for the api-gateway.

**Application.Properties –**

spring.application.name=api-gateway

server.port=8765

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka