***Step 01 Introduction to Microservices***

It's a small autonomous service that work together.

It says it's a style to develop a single application as a suite of small services each running its own

process communicating with very lightweight mechanisms.

Text

Description automatically generated

MicroServices are basically services which are exposed by REST in addition to that you have

well-chosen deployable units and these should be cloud enabled.

REST having small deployable units which can be cloud enabled.



This is how it would look instead of building one big monolith we would be building a set of smaller microservices

It's a set of microservices with well-defined boundaries which are interacting with each other and

these are cloud enabled.

Diagram

Description automatically generated

That means I would be able to have multiple instances for each of these micro services for example we're

looking at the diagram of a production deployment for a set of microservices.

We can see that at that at this moment

There are two instances of MicroService1

Four instances for MicroService2 and

One instance of MicroService3 by cloud enabled

I mean that if there is more load on MicroService3 I should be able to easily bring up another instance of

MicroService3.

This should not involve a lot of configuration I should be able to bring up an instance of MicroService3

or take down a instance of MicroService2 without having huge problems.

**That is what I mean by cloud enabled.**

So as far as this course is concerned, we already looked at RESTful web services what we look at in the subsequent

sections in this course is how to cloud enable them how to set up an architecture such that it would be

able to dynamically adjust and be able to bring new instances up and take the older instances down.

***Step 02 Challenges with Microservices***

**Bounded Context**

One Big monolith application we'd be building about five small microservices or 10 or 20 or hundred.

How do we identify the boundary for each of these microservices?

How do we identify what to do in each of these microservices?

How do we decide what We should do and what We should not do?

Thing is for new applications this is especially much more difficult because probably we don't really

have the business knowledge to be able to establish the right boundaries between these microservices.

What I found in my experience is deciding the boundaries of microservices.

This is an evolutionary process it's not something we would get it right the first time.

It's something which we need to play around with.

Try and follow different domain design try to identify the right boundaries for the microservices based

on the knowledge We have at that point of time.

The important thing to understand is as we keep gaining knowledge, we should put that knowledge back

into the microservices into deciding what is the right boundaries for these.

**Configuration management**

Suppose we have five or 50 microservices these microservices have multiple instances in each environment and there are multiple environments, and we don't mind.

So, let's say there are 10 microservices with 5 environments and let's say 50 instances to be talking about basically tons of configuration and that’s a lot of work for the operation team to maintain.

**Dynamic scale up and scale down**

Dynamically scale up and scale down and dynamically distribute load among the active instances.

Dynamic scale up and scale down establishing the technology to be able to do that.

The loads on different microservices will be different at different instances of time and at particular instance I might need two instances of Microservice2.

But later at a different point in time I might me needing 10 instances of this.

So, I should be able to bring the new instances of Microservice is up and bring down older instances of

Microservice when they are not really needed.

All this with dynamic load balancing because when there is one instance of Microservice1 and there

are four instances of Microservice2 then I would want to distribute the load between all the instances

of Microservice2, and if there are four instances of Microservice2 coming up then

I would want to ensure that all the new ones are also being used to the fullest extent.

So, we need the ability to dynamically bring in new instances and also to distribute the load among the

new instances.

**Visibility**

If I say the functionality is now distributed among 10 microservices and there is a **bug How do We identify where the bug is**, we need to have a centralized log where I can go and find out.

**What happened for a specific request which Microservice caused the problem** not just that?

we also need monitoring around these microservices because we have hundreds of microservices we need to be able to identify the microservices which are down.

We would want to be able to automatically identify those where there is not enough disk space.

All these kinds of things need to be automated.

So, we need great visibility into what's happening with these microservices.

**Pack of cards**

**If it's not well-designed microservices**

We have one microservice is calling 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, it's like a pack of cards We're building one over the top of the other and so on and so on, and therefore they get collapse very easily and therefore it's very important for we to have fault tolerance in our microservices.

***Step 03 Introduction to Spring cloud***

[***https://spring.io/projects/spring-cloud#overview***](https://spring.io/projects/spring-cloud#overview)

Graphical user interface, text, application, email

Description automatically generated

Go to [***https://spring.io/projects/spring-cloud#overview***](https://spring.io/projects/spring-cloud#overview)

It says spring cloud provides tools to quickly build some of the common patterns in distributed systems to the typical problems which are present for distributed systems in the cloud spring cloud provides a range of solutions.

The most important thing that you need to understand is spring cloud is not really one project as such.

There are a wide variety of projects under the umbrella of spring cloud on the spring cloud homepage.

If you scroll down further, you'll be able to see huge variety of projects which are related to spring

cloud.

One of the important projects in spring cloud is spring cloud Netflix.

Netflix was one of the organizations which started playing around with the microservices architecture very early.

There are a wide range of components that Netflix has Open-Source under the project spring Cloud Netflix

let's take a quick look at the challenges which we discussed earlier and the projects in spring cloud

that provide solutions to those challenges. One of the challenges we talked about earlier was

**Configuration management.**

Text

Description automatically generated



We talked about the fact that there would be multiple microservices multiple environments for each of

these microservices and multiple instances in many of those environments.

This would mean that there is a lot of configuration for these microservices that the operations team

needs to manage.

**Spring cloud config server** provides an approach where we can store all configuration for all

the different environment of all the microservices in a Git repository.

So, we can store all the configuration for different environments of different microservices in just one place in a centralized 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 and that makes it very easy to maintain the configuration for all the MicroServices.

Diagram

Description automatically generated

Text

Description automatically generated

Diagram

Description automatically generated

In the example you are looking at there is MicroService is called currency calculation service which is

talking to the currency exchange service.

As you can see in the diagram there are multiple instances of the currency exchange service and it's

possible that at any point in time new instances can be added in or removed out.

And we would want the currency calculations that we'd want to be able to distribute the load between all

the instances of the currency exchange service.

Who would want to be able to dynamically check **what are the available instances of the currency exchange**?

service and make sure that the load is distributed among all of them.

The solution which we would be discussing in this course would be using a **naming server which is Eureka**.

**So, all the instances of all MicroServices would register with the naming server.**

Naming server has two important features.

One is **service registration so all microservices**, can register with the microservic.

Second one is **service discovery**.

So, in this example the currency calculation service can ask the Eureka naming server Hey naming server

give me the current instances of currency exchange service and the naming service would provide those

URLs to the currency calculations.

This helps to establish dynamic relationship between the currency calculation service and the instances

of the currency exchange service.

We will use **Ribbon for client-side load balancing**.

That means the currency calculation service 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'll also use.

**Feign** in the currency calculation service as a mechanism to write simple RESTful clients.

Text

Description automatically generated

visibility and monitoring are the zipkin distributing Tracing Server.

We would use Spring cloud Slot to assign ID to request across multiple components and we would use the Zipkin distributed tracing to trace a request across multiple components.

One of the important things about microservices is these microservices have a lot of common features.

For example

logging security analytics and things like that you don't want to implement all these common

features in every microservice. API Gateways provide great solutions to these kinds of challenges.

We will use a **Netflix Zuel API gateway** in this could.

Text

Description automatically generated with low confidence

We will also implement fault tolerance using hystrix if a service is down. Hystrix help us to configure a default response.

***Step 04 Advantages of microservice architectures***

The most important advantage of the MicroServices architecture is that it enables us to adapt new technology

and processes very easy.

When we build applications as a combination of microservices which can communicate with each other

using simple messages each of these microservices can be built in different technologies.

For example

Microservice one might be java MicroService two might be Nodejs Microservice three might be

written in Kotlin and tomorrow there might be a language X Y Z which is really doing well and which

provides a lot of benefits to us and we can easily create a micro service in that specific language.

And also, for the new Microservices that we create we can bring in new processors as well.

***Dynamic scaling***

Dynamic scaling considers an online shopping application like Amazon.

They don't really have the same amount of load or same amount of traffic or same number 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 our micro-Services are cloud enabled, they can scale dynamically, and you can procure hardware and release that dynamically as well.

So, we can scale up our applications and scale them down based on the Load.

***Faster Release Cycles***

It's much easier to release micro services compared to monolith applications.

This means that you can bring new features faster to market and that's a big advantage to have in the modern world.

***Docker:*** Language Neutral, Cloud Neutral deployable units

***Kubernetes:*** Orchestrate Thousands of Microservices

***Microservices with Spring Cloud -V2***

***Step 1 Setting up Limits Microservices***

[**https://start.spring.io/**](https://start.spring.io/)

For initialize the project

Graphical user interface, text, application, email

Description automatically generated

Make sure that you don't have any spaces because spaces can give you problems with microservices, especially when we are talking about Cloud config server

When we add in spring-cloud-starter-config what we would need to do is to configure

How spring-cloud-starter-config need to connect to spring cloud, config server?

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

**optional means: spring cloud config server is not been configured then use optional else remove it**

***Reference***

CODE BACKUP FILES and STEP BY STEP CHANGES: For Reference

**Help for Debugging Problems:**

1. Here's the code backup at the end of Step 07: <https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/step07.md>
2. Step by Step changes are detailed here:<https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#step-01>

**Two Recommended Activities:**

**Activity - 1 :**Explore other backups for this section (Steps 08,10,13,15,21,25,29, final) - <https://github.com/in28minutes/spring-microservices-v2/tree/main/03.microservices>

**Activity - 2 :**Get Familiar with the structure of Step by Step changes file - <https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#step-01>

Diagram

Description automatically generated

***Step 2 Creating a hard coded limit service***

***pom.xml***

<dependency>

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

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

        </dependency>

        <dependency>

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

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

        </dependency>

Graphical user interface, application

Description automatically generated

**@RestController**

**public class LimitController {**

**@GetMapping("/limits")**

**public Limits retrieveLimits() {**

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

**}**

**}**

**@SpringBootApplication**

**public class SpringCloudConfigClient {**

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

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

**System.*out*.println("SpringCloudConfigClient has been started successfully!!");**

**}**

**}**

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

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

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

**public** Limits() {

}

**public** Limits(**int** minimum, **int** maximum) {

**super**();

**this**.minimum = minimum;

**this**.maximum = maximum;

}

//Setter and getter

}

Graphical user interface, text, application, email

Description automatically generated

***Step 3 Enhace MS-cloud-config-client by reading the values from property file***

**application.properties**

limits-service.minimum=3

limits-service.maximum=997

@Component

@ConfigurationProperties("limits-service")

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

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

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

**public** **int** getMinimum() {

**return** minimum;

}

**public** **void** setMinimum(**int** minimum) {

**this**.minimum = minimum;

}

**public** **int** getMaximum() {

**return** maximum;

}

**public** **void** setMaximum(**int** maximum) {

**this**.maximum = maximum;

}

}

@RestController

**public** **class** LimitController {

@Autowired

**private** Configuration configuration;

@GetMapping("/limits")

**public** Limits retrieveLimits() {

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

}

}

Graphical user interface, text, application, email

Description automatically generated

***Step 4 Setting up Spring cloud Config Server v2***

***Diagram

Description automatically generated***

Problem: After updating properties in configuration file services needs to be restarted to load configuration files again.

Text

Description automatically generated

**Add below dependency in pom.xml**

**<dependency>**

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

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

**</dependency>**

A picture containing text

Description automatically generated

**Application.property**

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

server.port=8888

@SpringBootApplication

**public** **class** SpringCloudConfigServer {

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

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

System.***out***.println("SpringCloudConfigServer has been started successfully!!");

}

}

***Step 5 Installing Git and Creating Local Git Repository***

Text

Description automatically generated

**Go to desire location and execute the below command**

**first init**

**Then place the property file in our current location and execute the remaining three command.**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Debugging problems with Spring Cloud Config Server - V2

**Debugging** microservices problems can be **difficult** as there are multiple components involved.

**Step by Step instructions** is provided in the **troubleshooting guide** to help you troubleshoot frequently occurring problems.

Using the **Chrome Browser** is recommended.

<https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#spring-cloud-config-server---steps-01-to-08>

***Step 6 Connect Spring cloud Config Server to Local Git Repository***

Text

Description automatically generated

***For windows –*** [***file:///followed***](file:///followed) ***by path with forward slashes!***

**spring.cloud.config.server.git.uri=file:///D:/config**

Text

Description automatically generated

#Access URL

#http://localhost:8888/centralized/default

#http://localhost:8888/centralized/dev

#http://localhost:8888/centralized/prod

Graphical user interface, text, email

Description automatically generated

***Default, dev, prod is the profile.***

***Created Git Repo***

Graphical user interface, text, application

Description automatically generated

*application.property file*

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

server.port=8888

spring.cloud.config.server.git.uri=file:///D:/config

***Step 7 Connect MS-cloud-config-cliet to Spring cloud config server***

<dependency>

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

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

</dependency>

***Above dependency is used to connect with spring cloud config server***

Graphical user interface, text, application

Description automatically generated

*application.property*

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

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

#optional means : spring cloud config server is not been configured then

# use optional else remove it.

spring.application.name=centralized

#Centralized is the property file name

limits-service.minimum=3

limits-service.maximum=997

*property which are there in* ***application.property*** *file having the less priority then property which are there in centralized property file.*

***Centralized.property***

***welcome.message.start=welcome to micro-service dev profile world start***

***limits-service.minimum=3***

***limits-service.maximum=996***

***welcome.message.end=welcome to micro-service dev profile world start end***

Graphical user interface, text, application

Description automatically generated

***Step 8 Configuring Profiles for Limits Service***

If there are multiple environments for the limits-service.

**a dev environment,**

**a QA,**

**a stage, and**

**a production environment,** how do we store separate configuration for limits-service for these four environments and how do we make use of it from the Spring Cloud Config Server?

Graphical user interface, text, application

Description automatically generated

Earlier, we created the centralized.properties in the git-localconfig-repo.

Now will create multiple copies of it, as shown in the above pic

Centralized-dev.properties

Centralized-prod.properties

Like wise we can create another environment property file as required.

So now we have values for multiple environments picked up from the GitHub repository.

**How do we make the application take the values from the Spring Cloud Config Server?**

Add in below property in application.property file

Spring.profiles.active=dev

Spring.cloud.config.profile=dev

*#By adding above two property it will take the dev profile as automatically*

*#and it will pic the values from the centralized dev property file*

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

***Step 9 Introduction to Currency conversion and Exchange Microservices***

Diagram

Description automatically generated

We would call these microservices as Currency Conversion Microservice and the Currency Exchange Microservice.

**What is the Currency Exchange?**

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

***Step 10 Setting up Currency Exchange Microservices***

Graphical user interface, text, application

Description automatically generated

**application.property**

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

#optional means: spring cloud config server is not been configured then use optional else remove it.

spring.application.name=currency-exchange

server.prot=8000

***Step 11 Create a simple hard coded currency exchange service.***

URL and Response Structure for Currency Exchange Service

**We will make use of these in the next lecture!**

URL

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

Response Structure

1. {
2. "id":10001,
3. "from":"USD",
4. "to":"INR",
5. "conversionMultiple":65.00,
6. "environment":"8000 instance-id"
7. }

URL Of currency exchange service will be

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

@RestController

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

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

**public** CurrencyExchange retrieveExchnageValue(

@PathVariable String from,

@PathVariable String to) {

**return** **new** CurrencyExchange(1000L, "USD", "INR", BigDecimal.*valueOf*(50));

}

}

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

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal conversionMultiple;

**public** CurrencyExchange(Long id, String from, String to, BigDecimal conversionMultiple) {

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

**this**.conversionMultiple = conversionMultiple;

}

// setter and getter method

}

**application.property**

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

#optional means : spring cloud config server is not been configured then

# use optional else remove it.

spring.application.name=currency-exchange

server.port=8000

pom.xml

**<dependency>**

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

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**<dependency>**

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

**<artifactId>spring-cloud-starter-config</artifactId>**

**</dependency>**

***Step 12 Setting up Dynamic port in the Response.***

Diagram

Description automatically generated

**import org.springframework.core.env.Environment;**

**@RestController**

**public class CurrencyExchangeController {**

**@Autowired**

**private Environment environment;**

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

**public CurrencyExchange retrieveExchnageValue(**

**@PathVariable String from,**

**@PathVariable String to) {**

**CurrencyExchange currencyExchange = new CurrencyExchange(1000L, "USD", "INR", BigDecimal.*valueOf*(50));**

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

**currencyExchange.setEnvironment(port);**

**return currencyExchange;**

**}**

**}**

**In** CurrencyExchange class add

**private** String environment.

and add the respective setter and getter method

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

***How to run multiple instances of above application on different port?***

***We need to configure it in eclipse.***

***Click on highlighted in yellow color then click on Run configuration***

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

***So once after clicking on Run – it will launch new instance of same application on different port***

***Now there are three instances are running for same application on different port.***

***Step 13 Configure JPA and initialized Data***

One of the things we can see in here is the name of the table.

The name of the table is currency underscore exchange.

However, the bean that we have created is currency exchange.

And even the column names if we look at them.

Multiple maps to conversion underscore multiple in Java.

We use camelCase.

However, in databases, typically under scores are used to separate worlds and spring data.

JPI understands that, and that's the reason why tables and columns are created with underscores.

**What would happen is whenever the application is restarted, did the data from this file would be loaded into your database?**

Yes

We have just created the data.sql file.

We want this to be loaded after the tables are created by default with the latest version of Springboot.

The load of data.sql, is done before the tables are created.

Yes, that sounds strange, but that's how it is.

And that's why we would need to make a configuration to defer the execution the data.sql.

**How can we do that?**

spring.jpa.defer-datasource-initialization=true

CODE BACKUP FILES and STEP BY STEP CHANGES: For Reference

#### **Help for Debugging Problems:**

* Here's the code backup at the end of Step 13: <https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/step13.md>
* Step by Step changes are detailed here: <https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#step-13>

#### **Two Recommended Activities:**

**Activity - 1** : Explore other backups for this section (Steps 08,10,13,15,21,25,29, final) - <https://github.com/in28minutes/spring-microservices-v2/tree/main/03.microservices>

**Activity - 2** : Get Familiar with the structure of Step by Step changes file - <https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#step-13>

***Step 14 Create a JPA Repository***

Now REST API to connect to the database and to be able to do that, we'd need to create a Spring Data class called Repository.

We must create a repository to manage this specific entity.

We have to create a interface. So, it should be a public interface CurrencyExchangeRepository and we need to make it extend a specific interface called JpaRepository.

I have to pass two things, which is the entity to be managed and what is the primary key.

The entity we would want to manage is CurrencyExchange and type of primary key Long

A picture containing timeline

Description automatically generated

**package** com.amit.microservices.bean;

**import** java.math.BigDecimal;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@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;

**public** CurrencyExchange(Long id, String from, String to, BigDecimal conversionMultiple) {

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

**this**.conversionMultiple = conversionMultiple;

}

//develop setter and getter method of above property

}

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.amit.microservices.bean.CurrencyExchange;

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

CurrencyExchange findByFromAndTo(String from,String to);

}

@RestController

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

@Autowired

**private** CurrencyExchangeRepository repository;

@Autowired

**private** Environment environment;

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

**public** CurrencyExchange retrieveExchnageValue(

@PathVariable String from,

@PathVariable String to) {

//CurrencyExchange currencyExchange = new CurrencyExchange(1000L, "USD", "INR", BigDecimal.valueOf(50));

//Getting currencyExchange from the db

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;

}

}

**application.properties**

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

#optional means : spring cloud config server is not been configured then

# use optional else remove it.

spring.application.name=currency-exchange

server.port=8000

#-Dserver.port=8001

spring.jpa.show-sql=true

# sql query generated will be shown on the console

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

#To connect with db2 URL

spring.h2.console.enabled=true

#To enable h2 database

spring.jpa.defer-datasource-initialization=true

Create **data.sql** in the same directory where application.properties file are there

*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',25,'');*

***To launch h2 console hit below URL***

[***http://localhost:8000:h2-console***](http://localhost:8000:h2-console)

***Every restart of server it will swap the existing data which are there in db2 database.***

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

***Step 15 Setting up Currency Conversion Microservices***

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

***application.properties***

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

#optional means : spring cloud config server is not been configured then

# use optional else remove it.

spring.application.name=currency-conversion

server.port=8100

URL and Response Structure for Currency Conversion Service

**We will make use of these in the next lecture!**

#### URL

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

#### Response Structure

1. {
2. "id": 10001,
3. "from": "USD",
4. "to": "INR",
5. "conversionMultiple": 65.00,
6. "quantity": 10,
7. "totalCalculatedAmount": 650.00,
8. "environment": "8000 instance-id"
9. }

***Step 16 Creating a service for Currency Conversion.***

Text

Description automatically generated

@RestController

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

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

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

**public** CurrencyConvresion calculateCurrencyConversion(

@PathVariable String from,

@PathVariable String to,

@PathVariable BigDecimal quantity) {

**return** **new** CurrencyConvresion(1000L, from, to, quantity, BigDecimal.***ONE***,BigDecimal.***ONE***, "");

}

}

**public** **class** CurrencyConvresion {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal quantity;

**private** BigDecimal conversionMultiple;

**private** BigDecimal totalCalucatedAmout;

**private** String environment;

**public** CurrencyConvresion(Long id, String from, String to, BigDecimal quantity, BigDecimal conversionMultiple,BigDecimal totalCalucatedAmout, String environment) {

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

**this**.conversionMultiple = conversionMultiple;

**this**.quantity = quantity;

**this**.totalCalucatedAmout = totalCalucatedAmout;

**this**.environment = environment;

}

//Develop setter and getter method.

}

Graphical user interface, application

Description automatically generated

***As of now we have hard coded its values.***

***Step 17 Invoking Currency Exchange from Currency Conversion.***

**How to call the Currency Exchange Microservice from the Currency Conversion Microservice?**

RestTemplate can be used to make REST API calls.

getforEntity(); Used to send a get request and we would want to get a object back.

first argument is what is the URL that you would want to invoke.

localhost:8000/currency-exchange/from/USD/to/INR

**@RestController**

**public class CurrencyConversionController {**

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

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

**public CurrencyConvresion calculateCurrencyConversion(**

**@PathVariable String from,**

**@PathVariable String to,**

**@PathVariable BigDecimal quantity) {**

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

**uriVariables.put("from", from);**

**uriVariables.put("to", to);**

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

**ResponseEntity<CurrencyConvresion> responseEntity =**

**new RestTemplate().getForEntity(**

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

**CurrencyConvresion.class,**

**uriVariables);**

**CurrencyConvresion currencyConvresion = responseEntity.getBody();**

**/\***

**\* return new CurrencyConvresion(1000L, from, to, quantity, BigDecimal.ONE,**

**\* BigDecimal.ONE, "");**

**\*/**

**return new CurrencyConvresion(currencyConvresion.getId(),**

**from, to, quantity,**

**currencyConvresion.getConversionMultiple(),**

**quantity.multiply(currencyConvresion.getConversionMultiple()),**

**currencyConvresion.getEnvironment());**

**}**

**}**

***Step 18 Using Feign Rest Client for Service invocation***

We had to write a lot of tedious code around RestTemplate to get the Currency Conversion

service to talk with the Currency Exchange Microservice.

To make a simple REST API call, we need to write about 20 lines of code. And just imagine **what would happen if we have hundreds of Microservices, they are calling each other?**

So, we need to repeat this kind of code everywhere.

And that's where Spring Cloud provides us with a framework called **Feign**.

Feign makes it really, easy to call other Microservices and to make use of Feign

we need to add a specific dependency into our Currency Conversion service.

<dependency>

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

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

</dependency>

We would want to be able to talk to the currency-exchange-service from the CurrencyConversionController.

What we need to do is to create a proxy?

@**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);

}

name="currency-exchange": can be name as application name.

From currency-conversion-service we'd want to call the currency-exchange.

we want to call currency-exchange using the Currency ExchangeProxy.

**@EnableFeignClients in main class**

@SpringBootApplication

**@EnableFeignClients**

**public** **class** CurrencyConversionServicesApplicationUsingFeign {

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

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

System.***out***.println("CurrencyConversionServicesApplicationUsingFeign"

+ " service has been started successfully!!");

}

}

@RestController

**public class CurrencyConversionController {**

@Autowired

**private** CurrencyExchangeProxy proxy;

@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()+ " " + "rest template");

}

**@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() + " " + "feign");**

**}**

**}**

Graphical user interface, text, application

Description automatically generated

***Step 19 Understand Naming Server and Setting up Eureka Naming Server***

We are hard coding the use of the currency exchange service.

So, if I would want to get the currency conversions, I wish to talk to a different instance of currency exchange, so **what do I need to do**?

I need to go here and change the localhost:8001, localhost:8002, localhost:8003 and so on.

If I, let's say Fein, provide an option where you can hardcourt multiple URLs in here, even

that would not be a good solution because.

@**FeignClient**(name="currency-exchange", url="localhost:8000,localhost:8001,localhost:8002")

Let's say 8000 went down and let's say a new instance was brought up on 8000.

Then we must change the configuration of this application of the code, of this application all

the time. And that's the reason **why we go for something called a service registry or a naming server.**

**What would happen is in a microdevices architecture**, all the instances of all the micro services would register with a service registry.

The Currency Exchange Service would register with the service registry and all the other micro services also registered with the service registry.

Currency conversion micro service wants to talk to the currency exchange, make service.

It would ask the service registry, hey, what are the addresses of the currency exchange?

Micro service?

The service registry would return those back to the currency and we should make a service.

And then the currency conversion micro service can send the request out to the currency exchange micro service.

**pom.xml**

<dependency>

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

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

</dependency>

@EnableEurekaServer

@SpringBootApplication

**public** **class** EurekaNamingServerApplication {

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

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

System.***out***.println("EurekaNamingServerApplication application has started!");

}

}

**application.properties**

spring.application.name=naming-server

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

URL: <http://localhost:8761/>

Graphical user interface, text, application

Description automatically generated

Debugging Problems with Eureka - V2

**Debugging** microservices problems can be **difficult** as there are multiple components involved.

**Step by Step instructions** is provided in the **troubleshooting guide** to help you Troubleshoot frequently occurring problems. Using the **Chrome Browser** is recommended.

**COMPLETE DEBUGGING GUIDE**

<https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#eureka---step-19-to-21>

**Top Recommendation From Debugging Guide:**

Give these settings a try **individually** in application.properties of all microservices (currency-exchange, currency-conversion) to see if they help

1. eureka.instance.prefer-ip-address=true

OR

1. eureka.instance.hostname=localhost

***Step 20 Connect Currency Conversion & Currency Exchange Microservices using naming server***

Diagram

Description automatically generated

**How we can connect our Microservices to the Naming Server?**

We have to connect the Currency Conversion Microservice and the Currency

Exchange Microservice connect to Naming server.

and afterwards, we'd want to make the Currency Conversion talk to Currency Exchange through the Naming Server.

Let's get the Currency Exchange Microservice and the **Currency Conversion Microservice to**

**register with the Naming server.**

**What we have to do for registration?**

Add below dependency in pom.xmls of currency-exchange-service and the currency-conversion-service and below dependencies

<dependency>

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

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

</dependency>

Add Eureka client dependency and that's all is needed for us to connect with Eureka.

For safer side configure it to properties file.

So, we can configure the naming server URL in the application.properties as well.

Add the below property in **application.properties** of currency-exchange-service and currency-conversion-service to configure eureka.

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

After adding above property and dependencies, if it is already started then stop and start both the services.

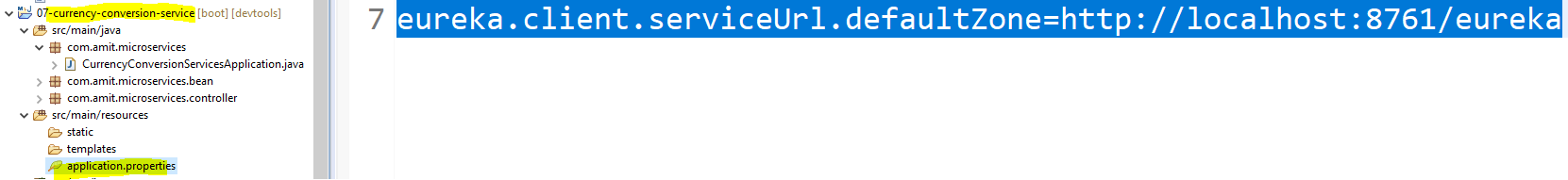
Then check the Eureka console - refresh Eureka we can see that currency-exchange-service

and currency-conversion-service is now registered with Eureka.

Each time an application is registered and unregistered. So, status up is when it's coming up and status down is when it's going down. So, each time when you restart the application, first it would be unregistered and then it would be registered again.

Text

Description automatically generated



<http://localhost:8761/>

Graphical user interface, text, application

Description automatically generated