**MICRO-SERVICES NOTES**

**Running Application on many ports:**

1. Right Click on Project ->Run Configurations -> Under Main Tab: Right Click on the Project -> Duplicate -> Go to the Arguments -> Provide -Dserver.port=8089 or whatever new port you want to run the project on.
2. **npx kill-port 3000**: Find and kill port
3. Request through API: localhost:8765/{app-name}/{uri}
4. [**http://localhost:8765/currency-converter-service/currency-converter-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-converter-service/currency-converter-feign/from/USD/to/INR/quantity/10)
5. **Removing Spring boot Actuator and Devtools will help run the application faster.**

A screenshot of a cell phone

Description automatically generated

1. Create a limit-service microservice

Dependencies: Toolkit, Web, Lombok, cloud, web, actuator

Port: 8081

-Two controllers:

1) Fetches hard coded values

2) Fetches values from application.properties.

URL: http://localhost:8081/limits-hardcoded and http://localhost:8081/limits/app-prop

2. Create a spring-cloud-config server with only devtools and config server.

Dependencies: Devtools, Config Server

Port: 8888

- Annotate with @EnableConfigServer

2.1 Connect Config Server to the git local repository

- Create a folder under the project location and do git init on it

- Add data to it and commit

- Connect the project to the local git folder

File -> Project Structure -> Modules -> Sources -> Add Content Root

- Create a file called limit-service.properties and mention minimum and maximum [This will be the default ]

limits-service.minimum = 8

limits-service.maximum = 888

2.2 We have provided the git folder info to the config server by the following on application.properties on the spring-config project:

spring.cloud.config.server.git.uri = file://Users/suzan/Desktop/INTELLIJ\_WORKSPS/spring\_boot\_micro\_services\_learn/git-local-config-repo

2.3 Run the app: http://localhost:8888/limits-service/default [port/name\_file\_that\_contains\_limits/default]

- Until now we have created 2 projects: limit-service and spring-cloud-config-server.

- We have made a connection of config server and the local git repo.

2.4. Create limit-service-dev.properties

limits-service.minimum = 2

limits-service.maximum = 222

2.5 Create limit-service-qa.properties

limits-service.minimum = 1

limits-service.maximum = 111

2.6 http://localhost:8888/limits-service/dev => shows 1, 111 and 8, 888 and low priority

2.7 http://localhost:8888/limits-service/qa => shows 2, 222 and 8, 888 and low priority

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+ SO FAR WE ARE JUST CONNECTING THE CONFIG SERVER TO THE GIT. +

+ - AND WE SET UP MULTIPLE ENVIRONMENTS: DEFAULT, DEV AND QA. +

+ NEXT PART IS TO CONNECT THE LIMIT-SERVICE APP TO CONFIG SERVER +

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

LETS CONNECT LIMITS-SERVICE TO THE CONFIG SERVER

- We want limits-service to talk to the config-server

- Rename the application.properties on limits-service to bootstrap.properties

-Inside the bootstrap.properties of limit-service set up:

spring.cloud.config.uri= http://localhost:8888

- Run both the app and go to http://localhost:8081/limits/app-prop

It will fetch the default values.

-> To pickup specific profile we need to add configuration on bootstrap.properties

#This will pick up values from dev

#spring.profiles.active = dev

#This will pick up values from qa

spring.profiles.active = qa

-> If we only provide the min or max the next one will be fetched from the default.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

3. Create a currency-exchange-service microservice:

Dependencies: Config client, Web, Lombok, Devtools, actuator

Port: 8000

Create a class that has fields required: id , from, to, conversionMultiple and port

3.1 Create a service method that will pull data from in-memory database as required.

3.2 Create a variable Port (currency-exchange-service) that will figure out which instance is coming from.

3.2 Generate getters and setters for the port.

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

public ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

ExchangeValue exchangeValue = new ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));

exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));

return exchangeValue;

}

3.3 Lets create two instances of the app running.

- Right click on the project and Run Configuration

- Go to the project (Main tab) on the side and duplicate it

- Click On the Arguments tab and create a new port using this: -DServer.port=8001

3.4 Run the app. Now the application will be running on both the ports: 8000 and 8001

3.5 Now we set up a JPA to in-memory database

3.5.1 Put in 2 jars on pom.xml

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

3.6 Set up application.properties:

#JPA

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

spring.data.jpa.repositories.bootstrap-mode=default

#Shows sql query in the console

spring.jpa.show-sql=true

spring.h2.console.enabled=true

3.7 Now we can add entity and Id value to the ExchangeValue class.

3.8 Create an interface for the repo and a method for query to query the database:

public interface ExchangeValueRepository extends JpaRepository<ExchangeValue, Long>{

ExchangeValue findByFromAndTo(String from, String to);

}

3.8 Create a method in the controller

@Autowired

private ExchangeValueRepository repository;

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

public ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to) {

ExchangeValue exchangeValue = repository.findByFromAndTo(from, to);

exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));

return exchangeValue;

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

4. Create another service currency-conversion-service with:

Dependencies: Config client, Web, Devtools, actuator

Port: 8100

4.1 Create a class with the follwing:

public class CurrencyConversion {

private Long id;

private String from;

private String to;

private BigDecimal conversionMultiple;

private BigDecimal quantity;

private BigDecimal totalCalculateAmount;

private int port;

}

4.2 Generate Constructor with all fields and no args constructor

4.3 Use the rest template to invoke other service

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

public CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to,

@PathVariable BigDecimal quantity) {

// Feign - Problem 1

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

uriVariables.put("from", from);

uriVariables.put("to", to);

ResponseEntity<CurrencyConversionBean> responseEntity = new RestTemplate().getForEntity(

"http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.class,

uriVariables);

CurrencyConversionBean response = responseEntity.getBody();

return new CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity,

quantity.multiply(response.getConversionMultiple()), response.getPort());

}

--> Here we are fetching data from the exchange server to get data from the database to multiply

the rate by the amount.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

---------------

|SPRING CLOUD:|

---------------

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

1. **Feign**: **Used for calling other microservices**.

++++++++++++++++++++++++++++++++

- Makes it easier to invoke other microservices.

- Provides integration with Ribbon that focuses on client-side load balancer

1.1 To upgrade code on 4.3 we are adding openfeign jars on pom.xml to the currency-exchange

1.2 To enable feign got to main application class on currency-converter and do this:

@EbableFeignClients("com/curr/conv") //The package where currency-exchange is located.

@SpringBootApplication

@EnableFeignClients("com/curr/conv")

public class CurrencyConversionServiceApplication {

public static void main(String[] args) {

SpringApplication.run(CurrencyConversionServiceApplication.class, args);

System.out.println("Currency Convertor is running!!!!");

}

}

1.3 Create a Feign Proxy to talk to outside microservice.

1.4 This is just like creating a JPA repository.

- create an interface called Proxy and create a method like this:

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

public interface CurrencyExchangeServiceProxy {

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

public CurrencyConversionBean retrieveExchageValue

(@PathVariable String from, @PathVariable String to);

}

1.5 Then on controller create this method to call the currency-exchange service.

@Autowired

private CurrencyExchangeServiceProxy proxy;

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

public CurrencyConversionBean convertCurrencyFeign(@PathVariable String from, @PathVariable String to,

@PathVariable BigDecimal quantity) {

CurrencyConversionBean response = proxy.retrieveExchageValue(from, to);

return new CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity,

quantity.multiply(response.getConversionMultiple()), response.getPort());

}

Ribbon: **Distribute the load: Used to call several instances of a micro-service**

A picture containing screenshot

Description automatically generated

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

RIBBON: To Balance the load: Ribbon can help us call different instances of CurrencyConverterService

--------------------------------------

1. Add maven dependecy of ribbon

<dependency>

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

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

2. Enable Ribbon on Proxy [ExchangeCurrencyServiceProxy]

3. go to application.properties of currency-converter-service and do this:

currency-exchange-service.ribbon.listOfServers = http://localhost:8000, http://localhost:8001

//This will connect this service to 2 instances of services of exchange running on port 8000 and 8001

4. On ProxyL

@FeignClient(name="currency-exchange-service") //calling another servie

@RibbonClient(name="currency-exchange-service") // distributing the load

public interface CurrencyExchangeServiceProxy {

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

public CurrencyConversionBean retrieveExchageValue

(@PathVariable String from, @PathVariable String to);

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Based on the previous setup, it is not easy to hardcode the instances of services running. We can use naming Server to dynamically reduce and increase the number of instances of servers.**

**Let’s use naming server to fix this issue.**

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**NAMING SERVER:**

1. **EUREKA:** When an instance of service comes up it will register itself with Eureka server (Service Registration).
2. When a service wants to talk to another server, it will ask the naming server what are the instances of the servers that it wants to talk to (Service Discovery).

**Here**: All services (exchange, limit, converter) register themselves to the naming server (Eureka). If any of them want to connect to the next they will talk to the Eureka Server.

A picture containing screenshot

Description automatically generated

STEPS:

1. Create a component for Eureka Naming Server.
2. Update currency-calculation-service to connect to Eureka Naming Server.
3. Connect currency-calculation-service to talk to Eureka Naming Server.
4. Configure Ribbon (currently installed on currency-converter-service).
5. Use Ribbon to find those details.

EUREKA SERVER:

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

STEPS:

1. Create a component for Eureka Naming Server.

name: netflix-eureka-naming-server

Dependencies: devtools, eureka server, config client, actuator

port: 8761

1.1 Under the main class:annotate with @EnableEurekaServer

1.2 Under application.properties set up the following:

spring.application.name = netflix-eureka-naming-server

server.port = 8761

eureka.client.register-with-eureka = false //Not registering with eureka server itself

eureka.client.fetch-registry = false //not fetching the registry

1.3 Run the application and go to localhost:8761. You will see the details of the Eureka server.

2. Update currency-calculation-service to connect to Eureka Naming Server.

2.1 Go to pom.xml of currency-calculation-service add dependency of eureka

<dependency>

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

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

</dependency>

2.2 Annotate @EnableDiscoveryClient so that its discovered.

@SpringBootApplication

@EnableFeignClients("com/curr/convert")

@EnableDiscoveryClient

public class CurrencyConvertionServiceApplication {

public static void main(String[] args) {

SpringApplication.run(CurrencyConvertionServiceApplication.class, args);

System.out.println("Currency Convertor is running!!!!");

}

}

2.3 Go to the appliction.properties of converter service and do the following:

eureka.client.service-url.default-zone = "http://localhost:8761/eureka"

3. Connect currency-exchange-service to talk to Eureka Naming Server.

3.1 Go to pom.xml of currency-calculation-service add dependency of eureka

<dependency>

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

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

</dependency>

3.2 Annotate @EnableDiscoveryClient so that its discovered.

2.3 Go to the appliction.properties of converter service and do the following:

eureka.client.service-url.default-zone = "http://localhost:8761/eureka"

4. Configure Ribbon (currently installed on currency-converter-service).

4.1 Go to Proxy class on Currency Converter and set up ribbon

@FeignClient(name = "currency-exchange-service")

@RibbonClient(name = "currency-exchange-service")

public interface CurrencyExchangeServiceProxy {

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

public CurrencyConversionBean retrieveExchageValue(@PathVariable String from, @PathVariable String to);

}

5. Use Ribbon to find those details.

@FeignClient(name = "currency-exchange-service")

@RibbonClient(name = "currency-exchange-service")

public interface CurrencyExchangeServiceProxy {

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

public CurrencyConversionBean retrieveExchageValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++

GATEWAY API: (ZUUL) instead of letting services calling directly, call through API Gateway: Also, for debugging and analytics purpose.

++++++++++++++++++++++++++++++++++++++++++++++

AUTHENTICATION: whether user has right permission

RATE LIMITS: certain number of calls per hour

FAULT TOLERATION: If something is wrong, default response is needed

SERVICE AGGREGATION: External consumer needs to call 15 services, better aggregate all the calls and provide 1 call.

++++++++++++++++++++++++++++++++++++++++++++++

**THREE STEPS FOR ZUUL:**

1. CREATE A COMPONENT.
2. WHAT SHOULD IT DO WHEN IT INTERCEPTS A REQUEST?
3. MAKE SURE ALL IMPORTANT REQUESTS ARE CONFIGURED TO PASS THROUGH API GATEWAY.

1. CREATE A COMPONENT.

- Created a service with the following dependencies: zuul, devtools, eureka discovery, actuator

- Enable @EnableZuulProxy on the main class

- Enable @EnableDiscoveryClient as well

- Configure on application.properties: name of service and the port number = 8765

2. WHAT SHOULD IT DO WHEN IT INTERCEPTS A REQUEST?

- Add logging

- Create a component class ZuulLoggingFilter and make it extend ZuulFilter

package com.zuuls.filter;

import java.util.logging.Logger;

import javax.servlet.http.HttpServletRequest;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import com.netflix.zuul.ZuulFilter;

import com.netflix.zuul.context.RequestContext;

import com.netflix.zuul.exception.ZuulException;

@Component

public class ZuulLoggingFilter extends ZuulFilter{

private org.slf4j.Logger logger = LoggerFactory.getLogger(this.getClass());

@Override

public boolean shouldFilter() {

return true; //true: it should filter everything

}

@Override

public Object run() throws ZuulException {

HttpServletRequest request = RequestContext.getCurrentContext().getRequest();

logger.info("request -> {} request uri -> {}", request, request.getRequestURI());

return null;

}

@Override

public String filterType() {

return "pre"; //what type of filter or when

}

@Override

public int filterOrder() {

return 1; //priority

}

}

-

3. MAKE SURE ALL IMPORTANT REQUESTS ARE CONFIGURED TO PASS THROUGH API GATEWAY.

- Now make sure that the request is passing throught the API Gateway

- New URL will be the URl of the API localhost:8765/{application-name}/{uri}

- New Urls:

1) Exchange Service: http://localhost:8765/currency-exchange-service/currency-exchange/from/{from}/to{to}

2) Converter Service: http://localhost:8765/currency-converter-service/currency-converter-feign/from/{from}/to{to}/quantity{quantity}

-Now lets make sure that the currency-converter service is not calling currency-exchange not directly

- Go to the Proxy class on converter service and comment on the @FeignClient

- Add @FeignClient(name = "netflix-zuul-api-gateway-server")

- Append currency-exchange-service to the following url.

//@FeignClient(name = "currency-exchange-service")

//This will call the naming server to get the uri for the exchange service

@FeignClient(name = "netflix-zuul-api-gateway-server")

@RibbonClient(name = "currency-exchange-service")

public interface CurrencyExchangeServiceProxy {

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

public CurrencyConversionBean retrieveExchageValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

**OTHER FEATURES**:

|  |  |  |
| --- | --- | --- |
| 1 | Distributed Tracing | One place I would like to go and see what happened to the request. Spring Cloud Sleuth with Zipkin will be used here. |
| 2 | Sleuth | Assign unique id to the request so that we can track them. But still does not provided centralized dashboard. You must go through individual log and check them.  -To solve this problem, we can create a centralized log.  -Then comes Zipkin, Kibana where we have centralized logs and here we go with Zipkin. |
| 3 | Zipkin | Distributed Tracing System: Dashboard. |
| 4 | Rabbit MQ | Queue to store message from microservices |
| 5 | Spring Cloud  Bus | Acts as a bus to transfer messages Need to look at it again |
| 6 | Fault Tolerance with Hystrix | Fault Tolerance. Need to look at it again |
| 7 |  |  |
| 8 |  |  |

**SLEUTH**:

Assigns unique id to the request so that we can track them.

Lets Implement Spring Cloud Sleuth on conversion, converter and api-gateway-server

Steps to implement Sleuth:

1. Add dependency to the pom.xml to all 3 of them above.

<dependency>

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

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

</dependency>

2. Tell all the requests to intercept on netflix-zuul-api-gateway-server

- Go to Application class on the netflix-zuul-api-gateway-server, exchange server and converter-server and create a sample bean

@Bean

public Sampler defaultSampler() {

return Sampler.ALWAYS\_SAMPLE;

}

3. Provide a logger to log it.

**ZIPKIN:**

**(….Like Kibana Dashboard)**

- Provides logs for all the servers to get the consolidated view of what’s happening across all micro- servers.

- Will use UI to see the details of all the services.

- Will use RabbitMQ: when any service has message or logs, it will put them into MQ using RabbitMQ.

- ZipkinSistributedServer will pick up the message from the queue and it will store the message to the database (here it’s an in-memory database).

**A close up of a map

Description automatically generated**

**Steps:**

1. Install RabbitMQ

* Make sure you have latest version of brew
* brew update
* brew install rabbitmq
* Launch rabbitmq by [/usr/local/sbin/rabbitmq-server]
* Or You can add this folder to the bash profile and just run: rabbitmq-server

1. Set up Distributed Tracing with Zipkin Tracing

* Go to google and type “Zipkin Quick Start”

Go to <https://github.com/in28minutes/spring-microservices#installing-tools> ->

* <https://zipkin.io/pages/quickstart> -> java and click on “latest release”
* The jar will be downloaded.
* Copy that jar into any folder and cd into it
* Add configuration to tell Zipkin to tell that RabbitMQ is running & start Zipkin Server.
* RABBIT\_URI=amqp://localhost java -jar zipkin-server-2.21.6-exec.jar
* Go to localhost:9411/Zipkin

1. Connecting microservies to put message into RabbitMQ.

* 1) Lets connect:
* Go to pom.xml and add a dependency of RabbitMQ and Zipkin to exchange, converter and zuul and run it
* <dependency>
* <groupId>org.springframework.cloud</groupId>
* <artifactId>spring-cloud-starter-zipkin</artifactId>
* </dependency>
* <dependency>
* <groupId>org.springframework.amqp</groupId>
* <artifactId>spring-rabbit</artifactId>
* </dependency>

DONE UP TO 112 on spring microservice course