Building microservices with Spring Cloud and Netflix OSS, part 1

10 APRIL 2015 // MAGNUS LARSSON

In the previous blog post we defined an operations model for usage of microservices. In this blog post we will start to look at how we can implement the model using Spring Cloud and Netflix OSS. From the operations model we will cover the core parts: *service discovery*, *dynamic routing*, *load balancing* and to some extend an *edge server*, leaving the other parts to upcoming blog posts.

We will use some core components from Spring Cloud and Netflix OSS to allow separately deployed microservices to communicate with each other with no manual administration required, e.g. keeping track of what ports each microservice use or manual configuration of routing rules. To avoid problems with port conflicts, our microservices will dynamically allocate free ports from a port range at startup. To

allow simple access to the microservices we will use an edge server that provides a well known entry point to the microservice landscape.

After a quick introduction of the Spring Cloud and Netflix OSS components we will present a system landscape that we will use throughout the blog series. We will go through how to access the source code and build it. We will also make a brief walkthrough of the source code pointing out the most important parts. Finally we wrap up by running through some test cases on how to access the services and also demonstrate how simple it is to bring up a new instance of a microservice and get the load balancer to start to use it, again without any manual configuration required.

1. SPRING CLOUD AND NETFLIX OSS

Spring Cloud is a new project in the spring.io family with a set of components that can be used to implement our operations model. To a large extent Spring Cloud 1.0 is based on components from Netflix OSS. Spring Cloud integrates the Netflix components in the Spring environment in a very nice way using auto configuration and convention over configuration similar to how Spring Boot works.

The table below maps the generic components in the operations model to the actual components that we will use throughout this blog series:

Operations Component	Netflix, Spring, ELK
Service Discovery server	Netflix Eureka
Dynamic Routing and Load Balancer	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitoring	Netflix Hystrix dashboard and Turbine
Edge Server	Netflix Zuul
Central Configuration server	Spring Cloud Config Server
OAuth 2.0 protected API's	Spring Cloud + Spring Security OAuth2
Centralised log analyses	Logstash, Elasticsearch, Kibana (ELK)

In this blog post we will cover *Eureka*, *Ribbon* and to some extent *Zuul*:

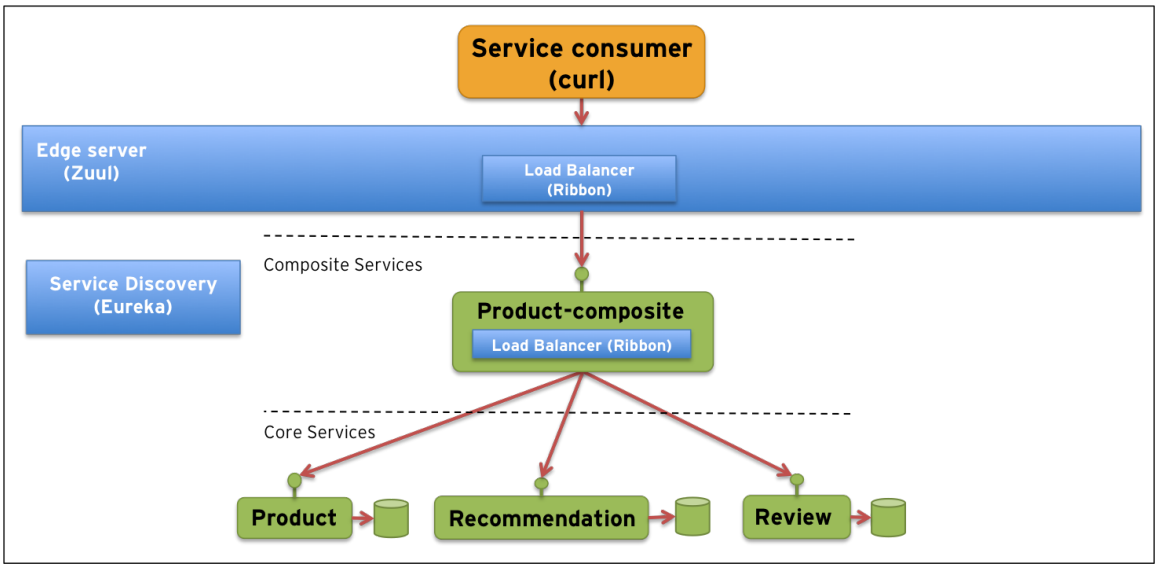
- **Netflix Eureka** - Service Discovery Server Netflix Eureka allows microservices to register themselves at runtime as they appear in the system landscape.
- **Netflix Ribbon** - Dynamic Routing and Load Balancer Netflix Ribbon can be used by service consumers to lookup services at runtime. Ribbon uses the information available in Eureka to locate appropriate service instances. If more than one instance is found, Ribbon will apply load balancing to spread the requests over the available instances. Ribbon does not run as a separate service but instead as an embedded component in each service consumer.
- **Netflix Zuul** - Edge Server Zuul is (of course) our gatekeeper to the outside world, not allowing any unauthorized external requests pass through. Zulu also provides a well known entry point to the microservices in the system landscape.

Using dynamically allocated ports is convenient to avoid port conflicts and to minimize administration but it makes it of course harder for any given service consumer. Zuul uses Ribbon to lookup available services and routes the external request to an appropriate service instance. In this blog post we will only use Zuul to provide a well known entry point, leaving the security aspects for coming blog posts.

Note: The microservices that are allowed to be accessed externally through the edge server can be seen as an API to the system landscape.

2. A SYSTEM LANDSCAPE

To be able to test the components we need a system landscape to play with. For the scope of this blog post we have developed a landscape that looks like:



It contains four business services (*green boxes*):

- Three core services responsible for handling information regarding **products**, **recommendations** and **reviews**.
- One composite service, **product-composite**, that can aggregate information from the three core services and compose a view of product information together with reviews and recommendations of a product.

To support the business services we use the following infrastructure services and components (*blue boxes*):

- **Service Discovery Server** (Netflix Eureka)
- **Dynamic Routing and Load Balancer** (Netflix Ribbon)
- **Edge Server** (Netflix Zuul)

> *To emphasize the differences between microservices and monolithic applications we will run each service in a separate microservice, i.e. in separate processes. In a large scale system landscape it will most probably be inconvenient with this type of fine grained microservices. Instead, a number of related services will probably be grouped in one and the same microservice to keep the number of microservices at a*

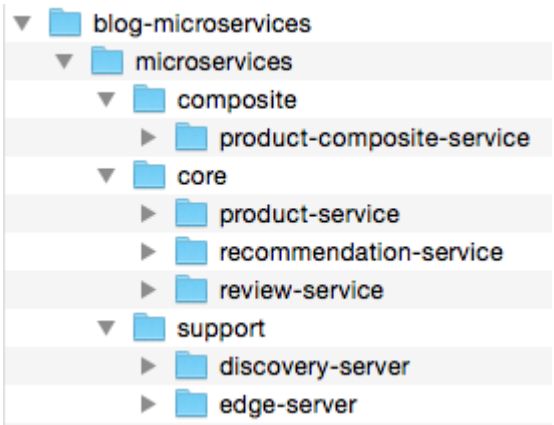
> *manageable level. But still without falling back into huge monolithic applications…*

3. BUILD FROM SOURCE

If you want to check out the source code and test it on your own you need to have Java SE 8 and Git installed. Then perform:

```
$ git clone https://github.com/callistaenterprise/blog-microservices.git
$ cd blog-microservices
$ git checkout -b B1 M1.1
```

This will result in the following structure of components:



Each component is built separately (remember that we no longer are building monolithic applications :-) so each component have their own build file. We use Gradle as the build system, if you don't have Gradle installed the build file will download it for you. To simplify the process we have a small shell script that you can use to build the components:

```
$ ./build-all.sh
```

> *If you are on **Windows** you can execute the corresponding bat-file `build-all.bat`!*

It should result in six log messages that all says:

```
BUILD SUCCESSFUL
```

4. SOURCE CODE WALKTHROUGH

Let's take a quick look at some key source code construct. Each microservice is developed as standalone Spring Boot application and uses Undertow, a lightweight Servlet 3.1 container, as its web server. Spring MVC is used to implement the

REST based services and Spring RestTemplate is used to perform outgoing calls. If you want to know more about these core technologies you can for example take a look at the following blog post.

Instead let us focus on how to use the functionality in Spring Cloud and Netflix OSS!

Note: We have intentionally made the implementation as simple as possible to make the source code easy to grasp and understand.

4.1 GRADLE DEPENDENCIES

In the spirit of Spring Boot, Spring Cloud has defined a set of starter dependencies making it very easy to bring in the dependencies you need for a specific feature. To use Eureka and Ribbon in a microservice to register and/or call other services simply add the following to the build file:

```
    compile("org.springframework.cloud:spring-cloud-starter-eureka:1.0.0
```

For a complete example see product-service/build.gradle.

To be able to setup an Eureka server add the following dependency:

```
    compile('org.springframework.cloud:spring-cloud-starter-eureka-serve
```

For a complete example see discovery-server/build.gradle.

4.2. INFRASTRUCTURE SERVERS

Setting up an infrastructure server based on Spring Cloud and Netflix OSS is really easy. E.g. for a Eureka server add the annotation `@EnableEurekaServer` to a standard Spring Boot application:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

For a complete example see EurekaApplication.java.

To bring up a Zuul server you add a `@EnableZuulProxy` - annotation instead. For a complete example see ZuulApplication.java.

With these simple annotations you get a default server configurations that gets yo started. When needed, the default configurations can be overridden with specific

settings. One example of overriding the default configuration is where we have limited what services that the edge server is allowed to route calls to. By default Zuul set up a route to every service it can find in Eureka. With the following configuration in the `application.yml` - file we have limited the routes to only allow calls to the composite product service:

```
zuul:
  ignoredServices: "*"
  routes:
    productcomposite:
      path: /productcomposite/**
```

For a complete example see edge-server/application.yml.

4.3 BUSINESS SERVICES

To auto register microservices with Eureka, add a `@EnableDiscoveryClient` - annotation to the Spring Boot application.

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProductServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}
```

For a complete example see ProductServiceApplication.java.

To lookup and call an instance of a microservice, use Ribbon and for example a Spring RestTemplate like:

```
    @Autowired
    private LoadBalancerClient loadBalancer;
    ...
    public ResponseEntity<List<Recommendation>> getReviews(int productId

            ServiceInstance instance = loadBalancer.choose("review");
            URI uri = instance.getUri();
    ...
            response = restTemplate.getForEntity(url, String.class);
```

The service consumer only need to know about the name of the service (`review` in the example above), Ribbon (i.e. the `LoadBalancerClient` class) will find a service instance and return its URI to the service consumer.

For a complete example see Util.java and ProductCompositeIntegration.java.

5. START UP THE SYSTEM LANDSCAPE

> *In this blog post we will start the microservices as java processes in our local development environment. In followup blog posts we will describe how to deploy microservices to both cloud infrastructures and Docker containers!*

To be able to run some of the commands used below you need to have the tools cURL and jq installed.

Each microservice is started using the command `./gradlew bootRun`.

First start the infrastructure microservices, e.g.:

```
$ cd .../blog-microservices/microservices

$ cd support/discovery-server;  ./gradlew bootRun
$ cd support/edge-server;       ./gradlew bootRun
```

Once they are started up, launch the business microservices:

```
$ cd core/product-service;                ./gradlew bootRun
$ cd core/recommendation-service;         ./gradlew bootRun
$ cd core/review-service;                 ./gradlew bootRun
$ cd composite/product-composite-service; ./gradlew bootRun
```

> *If you are on **Windows** you can execute the corresponding bat-file `start-all.bat`!*

Once the microservices are started up and registered with the service discovery server they should write the following in the log:

```
DiscoveryClient ... - registration status: 204
```

In the service discovery web app we should now be able to see our four business services and the edge server (http://localhost:8761):

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EDGESERVER	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local
PRODUCT	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:product:eb83f38215c5936a6f8cd3c1c820fc83
PRODUCTCOMPOSITE	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:productcomposite:a74c76b3fcdaa693e046528a52f93ad4
RECOMMENDATION	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:recommendation:fdc59a3a42dcb0b0074668cc7ada15ec
REVIEW	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:review:99a9024f3a1cc3415d146e783a0f6bf2

To find out more details about our services, e.g. what ip addresses and ports they use, we can use the Eureka REST API, e.g.:

```
$ curl -s -H "Accept: application/json" http://localhost:8761/eureka/app
{
  "service": "PRODUCT",
  "ip": "192.168.0.116",
  "port": "59745"
}
{
  "service": "REVIEW",
  "ip": "192.168.0.116",
  "port": "59178"
}
{
  "service": "RECOMMENDATION",
  "ip": "192.168.0.116",
  "port": "48014"
}
{
  "service": "PRODUCTCOMPOSITE",
  "ip": "192.168.0.116",
  "port": "51658"
}
{
  "service": "EDGESERVER",
  "ip": "192.168.0.116",
  "port": "8765"
}
```

We are now ready to try some test cases, first some happy days tests to verify that we can reach our services and then we will se how we can bring up a new instance of a microservice and get Ribbon to load balance requests over multiple instances of that service.

Note: In coming blog posts we will also try failure scenarios to demonstrate how a circuit breaker works.

5.1 HAPPY DAYS

Start to call the composite service through the edge server, The edge server is found on the port 8765 (see its application.yml file) and as we have seen above we can use the path `/productcomposite/**` to reach the product-composite service through our edge server. The should return a composite response (shortened for brevity) like:

```
$ curl -s localhost:8765/productcomposite/product/1 | jq .
{
    "name": "name",
    "productId": 1,
    "recommendations": [
        {
            "author": "Author 1",
            "rate": 1,
            "recommendationId": 1
        },
        ...
    ],
    "reviews": [
        {
            "author": "Author 1",
            "reviewId": 1,
            "subject": "Subject 1"
        },
        ...
    ],
    "weight": 123
}
```

…if we are on the inside of the microservice landscape we can actually call the services directly without going through the edge server. The problem is of course that we don't know on what ports the services runs on since they are allocated dynamically. But if we look at the output from the call to the eureka rest api we can find out what ports they listen to. To call the composite service and then the three core services we should be able to use the following commands (given the port information in the response from the call to the eureka rest api above)

```
$ curl -s localhost:51658/product/1 | jq .
$ curl -s localhost:59745/product/1 | jq .
$ curl -s localhost:59178/review?productId=1 | jq .
$ curl -s localhost:48014/recommendation?productId=1 | jq .
```

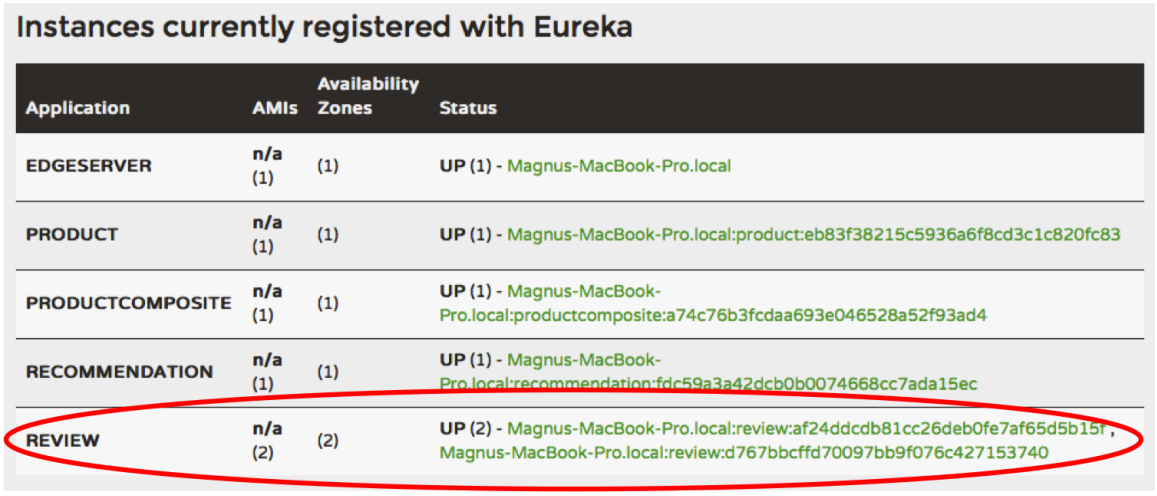
Try it out in your own environment!

5.2 DYNAMIC LOAD BALANCING

To avoid service outage due to a failing service or temporary network problems it is very common to have more than one service instance of the same type running and using a load balancer to spread the incoming calls over the instances. Since we are using dynamic allocated ports and a service discovery server it is very easy to add a new instance. For example simply start a new review service and it will allocate a new port dynamically and register itself to the service discovery server.

```
$ cd .../blog-microservices/microservices/core/review-service
$ ./gradlew bootRun
```

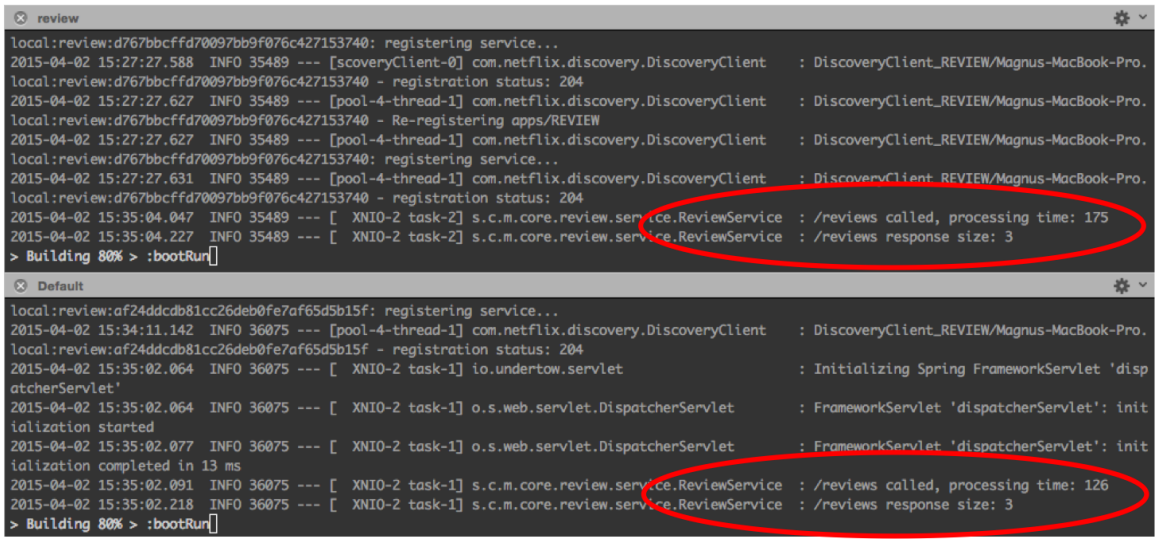
After a short while you can see the second instance in the service discovery web app (http://localhost:8761):



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EDGESERVER	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local
PRODUCT	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:product:eb83f38215c5936a6f8cd3c1c820fc83
PRODUCTCOMPOSITE	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:productcomposite:a74c76b3fcdaa693e046528a52f93ad4
RECOMMENDATION	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:recommendation:fdc59a3a42dcb0b0074668cc7ada15ec
REVIEW	n/a (2)	(2)	UP (2) - Magnus-MacBook-Pro.local:review:af24ddcdb81cc26deb0fe7af65d5b15f , Magnus-MacBook-Pro.local:review:d767bbcffd70097bb9f076c427153740

If you run the previous curl command (`curl -s localhost:8765/productcomposite/product/1 | jq .`) a couple of times and look into the logs of the two review instances you will see how the load balancer automatically spread in calls over the two instances without any manual configuration required:



```
review
local:review:d767bbcffd70097bb9f076c427153740: registering service...
2015-04-02 15:27:27.588  INFO 35489 --- [scoveryClient-0] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740 - registration status: 204
2015-04-02 15:27:27.627  INFO 35489 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740 - Re-registering apps/REVIEW
2015-04-02 15:27:27.627  INFO 35489 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740: registering service...
2015-04-02 15:27:27.631  INFO 35489 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740 - registration status: 204
2015-04-02 15:35:04.047  INFO 35489 --- [  XNIO-2 task-2] s.c.m.core.review.service.ReviewService  : /reviews called, processing time: 175
2015-04-02 15:35:04.227  INFO 35489 --- [  XNIO-2 task-2] s.c.m.core.review.service.ReviewService  : /reviews response size: 3
> Building 80% > :bootRun

Default
local:review:af24ddcdb81cc26deb0fe7af65d5b15f: registering service...
2015-04-02 15:34:11.142  INFO 36075 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:af24ddcdb81cc26deb0fe7af65d5b15f - registration status: 204
2015-04-02 15:35:02.064  INFO 36075 --- [  XNIO-2 task-1] io.undertow.servlet                       : Initializing Spring FrameworkServlet 'disp
atcherServlet'
2015-04-02 15:35:02.064  INFO 36075 --- [  XNIO-2 task-1] o.s.web.servlet.DispatcherServlet         : FrameworkServlet 'dispatcherServlet': init
ialization started
2015-04-02 15:35:02.077  INFO 36075 --- [  XNIO-2 task-1] o.s.web.servlet.DispatcherServlet         : FrameworkServlet 'dispatcherServlet': init
ialization completed in 13 ms
2015-04-02 15:35:02.091  INFO 36075 --- [  XNIO-2 task-1] s.c.m.core.review.service.ReviewService  : /reviews called, processing time: 126
2015-04-02 15:35:02.218  INFO 36075 --- [  XNIO-2 task-1] s.c.m.core.review.service.ReviewService  : /reviews response size: 3
> Building 80% > :bootRun
```

6. SUMMARY

We have seen how Spring Cloud and Netflix OSS components can be used to simplify making separately deployed microservices to work together with no manual administration in terms of keeping track of what ports each microservice use or manual configuration of routing rules. When new instances are started they are automatically detected by the load balancer through the service discovery server and can start to receive requests. Using the edge server we can control what microservices that are exposed externally, establishing av API for the system landscape.

7. NEXT STEP

Ok, so happy days scenarios looks great!

But a lot of questions remains unanswered, e.g.:

- What about something goes wrong, like a failing microservice?
- How do we prevent unauthorized access to our API ()through the edge server)?
- How do we get a good consolidated picture of what is going on in the microservice landscape, e.g. why isn't order #123456 delivered yet?

In upcoming blog posts in the Blog Series - Building Microservices we will look at how to increase resilience using a circuit breaker, use OAuth 2 to restrict external access. We will also look into how we can user the ELK stack to collect and present log entries from all the microservices in a centralized and consolidated way and more.

Stay tuned!

KOMMENTARER

Tushar Chowdhury • 3 years ago

Hello Magnus,

Greetings. Can we get an article on topics like Message/Event driven microservices (RabbitMQ, Kafka) and CQRS patterns (Axon). Liked all your articles end to end. Excellent work.

Best Regards,

Magnus Larsson ➚ Tushar Chowdhury • 2 years ago

Hello Tushar!

Great to see that you like the blog posts!

Your suggestions are really interesting and important topics, I hope to get some time to write about these technologies :-)

Best Regards,
Magnus.

^ | v • Reply • Share ›

Knel Soeng • 2 years ago

Thanks god, for direct me to found this article.
Thanks author.

1 ^ | v • Reply • Share ›

Magnus Larsson ➚ Knel Soeng • 2 years ago

Hello Knel!

Thank you for the feedback :-)

Best Regards,
Magnus.

^ | v • Reply • Share ›

Manuel Silva • 6 months ago

Best Regards From Mexico City. Manuel Silva

^ | v • Reply • Share ›

Ramesh Babu Y • 2 years ago

when am trying to run the apps , apps stuck at

Building 80% > :bootRun

^ | v • Reply • Share ›

Magnus Larsson ➚ Ramesh Babu Y • 2 years ago

Hello Ramesh!

I guess you are stuck on the command "./gradlew bootRun"?

Since Gradle actually not only builds the microservice but also starts it, it will not finish until you stop the running microservice (eg press CTRL/C).
Therefore it reports "Building 80% > :bootRun" when the microservice actually is up and running.

If the startup completes successfully you should be able to see the following message in the console:

"DiscoveryClient ... - registration status: 204"

Does that answer your question?

Best Regards,
Magnus.

Reply • Share ›

Kent Johnson • 3 years ago • edited

How do you get Eureka and Ribbon to recognize two instances of the review service? I can only get it to recognize one instance at a time of my service. I can get it to work with your services, though with my own I am struggling to get Eureka to pick up my two instances.
I see how the code in the composite service is using Ribbon to do client-side load balancing. Do I have to make a separate API service that uses Ribbon directly in order to take advantage of load balancing in Spring Cloud? Or does Zuul automatically load-balance requests with multiple registered instances of a service?

Reply • Share ›

Kent Johnson ➚ Kent Johnson • 3 years ago

I figured it out. It is a config property name change. It goes from eureka.instance.metadataMap.instanceId to eureka.instance.instance-id instead. It was as simple as that. Thanks for the great article, Magnus!
I am having another trouble getting the Eureka Dashboard to display.

Reply • Share ›

abcd • 3 years ago

HI magnus:

I have a Q on the file https://github.com/callista...

Around line 32, I see this there: private RestTemplate restTemplate = new RestTemplate();

I think since you use ProductCompositeIntegration... as a bean as I can see @Component annotation, I believe this can interfere if there are multiple requests in progress as the ProductCompositeIntegration bean is a singleton by default. Can you confirm if my understanding is correct and if I am not correct please help me understand why

Reply • Share ›

Priyal Walpita • 3 years ago

Hi Larsson, Thanks a lot for the nice blog about micro services. I like to do some tweek into this. How we can pass in couple of values in to a service when we run them? (ie : when executing the gradlew bootRun). And how we can capture them in the

application ?

Reply • Share ›

Magnus Larsson ➚ Priyal Walpita • 3 years ago

Hello!

Great to see that you like the blog post!

Regarding injecting variables at startup I guess you can use vanilla Spring properties.

E.g. define your variable at startup by adding something like:

-Dmyvariable=myvalue

And then in the Spring Bean where you want to use the value inject it like:

@Value("${myvariable}")
private String var;

...or you could use the excellent Spring Cloud Configuration Server, that I still need to find time to write about...

Regards,
Magnus.

Reply • Share ›

Priyal Walpita ➚ Magnus Larsson • 3 years ago

Hi Magnus,

Thanks for your tip and I did the coding as follows .

In Start-all.bat file : start /D microservices\api\product-api-service gradlew bootRun -DmyUser=mario

and in the ProductApiService class as follows :

@Value("${myUser}")
private String user;

The class is compiling fine but I am getting following error when it trying to test the application. It would be great if you can let me know how can I change the test class accordingly to support this new param. I am not a java guy but I am interested in Micro services.

see more

Reply • Share ›

Igor • 3 years ago

Hello, I have not yet worked through the whole example, but I already wanted to thank you very much for this tutorial as I am trying to learn how to use and create microservices - and this tutorial is very detailed.

In order to import the projects into Eclipse Neon.1 I had to install the official gradle plugin, but even then I had to import the projects separately, which was a bit cumbersome.

The only the question that I have (at the moment) is:

does one really have to add product-service and review-service as dependencies to the product-composite-service? I am not sure whether those dependencies are necessary when actually running the services, but they seem necessary when developing in Eclipse. So I just added them.

It is also worth noting, that one has to actually INSTALL the lombok.jar (by dropping it onto Eclipse). The gradle-dependency is already included, but it is apparently not enough if one wants to import the projects into Eclipse like I do.

Thank you!

 • Reply • Share ›

Magnus Larsson ➚ Igor • 3 years ago

Hello Igor!

Yes, in this early version of the code base (branch B1) I had a dependency from the composite service to the core services. I got early feedback on that and moved the common code to a util project that all microservices depend on (each microservice decide independently on what version of the util project they depend on).

Regarding separate import I agree with you that it is cumbersome from a blog post/lab perspective. But from a real world perspective I like to see the different microservices independent from each other (with some more code in each microservice, they are very small in my example). E.g. I would like to be able to establish one independent build pipeline per microservice, that they need to be separate projects.

Regarding lombok.jar I have no clue, from what I know I don't use it (not intentionally at least). Maybe I have som dead code (currently not used by the blog posts) in the github-repo that depends in lombok.jar?

Hope it helps,
Regards Magnus.

Regards Magnus.

Reply • Share ›

Aakash Shah • 4 years ago

Hi Mangnus,

It's an informative endeavor and I got quite a lot of insights on microservice architecture. I'd like to know if I can build it in STS. I have downloaded the Gradle and Groovy Extensions. But the Gradle-Tasks shows an ampty list. I am not sure where am i going wrong. It works fine with command line. And in STS what are the steps to start a gradle-based microservices project?

Reply • Share ›

Magnus Larsson ➚ Aakash Shah • 3 years ago

Hello!

I'm sorry but I no longer use neither Eclipse nor STS, hopefully you have found out a solution already. Otherwise I can strongly recommend IntelliJ :-)

Reply • Share ›

Jianeng Xu • 4 years ago

That's great! Your discussions about Oauth2 help me a lot!

Reply • Share ›

Magnus Larsson ➚ Jianeng Xu • 3 years ago

Hi, happy to find out that it helped you!

Reply • Share ›

Neider Tapia • 4 years ago

Thanks. Why the ${server.random} in eureka.instance.metadataMap.instanceId is 0 (cero) always. Eureka register only one instance for de same name.

Instance 1
2016-04-02 11:43:27.829 INFO 8412 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_MANUALTARIFFSERVICE/E-0616.sprc.com.co:ManualTariffService:0 - registration status: 204

Instance 2
2016-04-02 11:43:58.889 INFO 9704 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_MANUALTARIFFSERVICE/E-0616.sprc.com.co:ManualTariffService:0 - registration status:

Reply • Share ›

Magnus Larsson ➚ Neider Tapia • 4 years ago

That's strange!

As you can see i the section "5.2 DYNAMIC LOAD

As you can see i the section "5.2 DYNAMIC LOAD BALANCING" each instance gets its own unique instance id.

Are you using the same versions of Spring Boot and Spring Cloud as I did when I wrote this Blog Post?
(i.e. as defined by the Git tag M1.1 as described in section "3. BUILD FROM SOURCE")

I think I have seen problems in later versions of the Spring Boot/Cloud with how instances are named but I don't remember the exact versions.

Hope it helps,
Magnus.

Reply • Share ›

Neider Tapia ➚ Magnus Larsson • 4 years ago

Hi, my project use spring-cloud-dependencies version Brixton.RC1. Change to Angel.SR6 and work!

Thanks.

Reply • Share ›

Eddy Zhu ➚ Neider Tapia • 4 years ago

I looked at the Spring cloud code. Adding spring.application.instance_id: ${random.value} to your application.yml file does the trick.

Reply • Share ›

letal • 4 years ago

Great article. I have a question. Do spring cloud and eureka-server need internet to work.
If i already have the libraries and it dependencies in my development environment, do i need internet to deploy it in the production environment ?

Reply • Share ›

Magnus Larsson ➚ letal • 4 years ago

Hello Letal!

No, you don't need, from what I know, access to Internet to use Spring Cloud, Netflix Eureka et al.
Is there any specific concern that makes you raise the question?

Regards,
Magnus.

Reply • Share ›

letal ➚ Magnus Larsson • 4 years ago

I want to install or deploy spring cloud and netflix ,eureka on a server not connected to internet. Will i encounter any difficulties doing so ?

^ | v • Reply • Share ›

Magnus Larsson ➚ letal • 4 years ago

I can't see any upfront :-)
Give it a try!

^ | v • Reply • Share ›

taruni nandu • 4 years ago

Thanks for the update...
The post is good to read and useful for the people to meet their success. We IT Hub Online Training supplying
Spring Online Training services.

^ | v • Reply • Share ›

Nguyen Duong Vu • 4 years ago • edited

Thanks a lot for a great article. It's awesome!

^ | v • Reply • Share ›

MadGeek • 4 years ago

HI i'm getting this error can anyone give me an idea about this?

java.lang.NullPointerException: null
at
com.netflix.eureka.resources.StatusResource.isReplicaAvailable(St
at
com.netflix.eureka.resources.StatusResource.getStatusInfo(Status
at
org.springframework.cloud.netflix.eureka.server.EurekaController
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcce
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMet

at java.lang.reflect.Method.invoke(Method.java:497)

at

see more

^ | v • Reply • Share ›

Wang Zhipeng ➚ MadGeek • 4 years ago

I got the same erro when I click the 'localhost' link which redirects to localhost:8761/eureka. Have you already found what's going wrong here?

Reply • Share ›

Rajesh P • 4 years ago

Hey , i am getting an error while using the composite microservice code , the issue is ServiceInstance instance = loadBalancer.choose(serviceId); , instance is evaluated as null. Could you please guide on what could be wrong ?

Reply • Share ›

Magnus Larsson ➚ Rajesh P • 4 years ago

Hello Rajesh!

From what you are describing I guess that either is there a problem with the communication between the composite service and the discovery service (i.e. Eureka) or between the core services and the discovery service.

Can you take a look into Eureka web app, http://localhost:8761, to see what services that are registered with the discovery service?

Regards,
Magnus.

Reply • Share ›

Jagan • 4 years ago

Magnus, Thank you for such a great article. I haven't come across an article like this which ties Microservices to the deployment landscape. Very well written. I am wondering if there are articles that describe a more complex infrastructure setup with Spring Cloud and OSS where multiple teams are contributing to the microservices and operational teams managing Zuul and Eureka installations/processes.

Reply • Share ›

Magnus Larsson ➚ Jagan • 4 years ago

Hello and thanks!

No, not of what I'm aware of.

We plan to, over time, cover more complex setups (e.g. multiple teams and multiple regions) so if you have specific questions that you would like to see covered please let us know!

Regards,
Magnus.

Reply • Share ›

Christian Huening • 4 years ago • edited

First of all: Great Article! And a great example! I really appreciate your work here, since we're venturing into the world of

your work here, since we're venturing into the world of microservices ourselves right now!
I've got a question regarding the application.yml files. Are these generated by some magic at startup or do I have to write them all by myself? Is the standard setup in place, even though I don't have the files at all?

Reply • Share ›

Magnus Larsson ➚ Christian Huening • 4 years ago

Hello Christian!

Thanks a lot!

The application.ml files are standard Spring Boot configuration files (see http://docs.spring.io/sprin... for details).

There are a lot of very good default values. But over time you will learn that you need to override some properties to get the behaviour that you want, for example when it comes to Eureka, Ribbon and Hystrix.

Regards,
Magnus.

Reply • Share ›

arthur autohandle • 4 years ago

i get stuck on startup:
cd support/discovery-server; ./gradlew bootRun

2015-08-06 22:12:45.335 INFO 1550 --- [Thread-3] c.n.eureka.PeerAwareInstanceRegistry : Got 1 instances from neighboring DS node
2015-08-06 22:12:45.335 INFO 1550 --- [Thread-3] c.n.eureka.PeerAwareInstanceRegistry : Renew threshold is: 1
2015-08-06 22:12:45.335 INFO 1550 --- [Thread-3] c.n.eureka.PeerAwareInstanceRegistry : Changing status to UP
2015-08-06 22:12:45.336 INFO 1550 --- [Thread-3] com.netflix.eureka.InstanceRegistry : Finished initializing remote region registries. All known remote regions: []
2015-08-06 22:12:45.345 INFO 1550 --- [Thread-3] com.netflix.eureka.EurekaBootStrap : Replica node URL: http://localhost:8761/eureka/
2015-08-06 22:12:45.345 INFO 1550 --- [Thread-3] e.s.EurekaServerInitializerConfiguration : Started Eureka Server
> Building 80% > :bootRun

either that or it is very (very) slow.

Reply • Share ›

arthur autohandle ➚ arthur autohandle
• 4 years ago • edited

i got it to run by running each "app" in a separate teminal (tab) and they are all stuck at:
> Building 80% > :bootRun

but they did all seem to start & register.

and curl seems to work

ubuntu@ip-10-0-3-240:~$curl -s localhost:8765/productcomposite/product/1
{"productId":1,"name":"name","weight":123,"recommendat
[{"recommendationId":1,"author":"Author 1","rate":1},
{"recommendationId":2,"author":"Author 2","rate":2},
{"recommendationId":3,"author":"Author 3","rate":3}],"reviews":[{"reviewId":1,"author":"Author 1","subject":"Subject 1"},{"reviewId":2,"author":"Author 2","subject":"Subject 2"},{"reviewId":3,"author":"Author 3","subject":"Subject 3"}]}

i got an error on the other one:
curl -s -H "Accept: application/json" http://localhost:8761/eureka/apps | jq '.applications.application[] | {service: .name, ip: .instance.ipAddr, port: .instance.port."$"}'
error: syntax error, unexpected QQSTRING_START, expecting IDENT
.applications.application[] | {service: .name, ip: .instance.ipAddr, port: .instance.port."$"}

but without the jq part - it seems to be working, as well
ubuntu@ip-10-0-3-240:~$ curl -s -H "Accept: application/json" http://localhost:8761/eureka/apps
{"applications":
{"versions__delta":1,"apps__hashcode":"UP_5_","applicat
[{"name":"PRODUCT","instance":{"hostName":"ip-10-0-3-240","app":"PRODUCT","ipAddr":"10.0.3.240","status":"U
{"@enabled":"true","$":"57303"},"securePort":
{"@enabled":"false","$":"443"},"countryId":1,"dataCenterI
{"@class":"com.netflix.appinfo.InstanceInfo$DefaultDataC
...

Tel: +46 8 21 41 01 Tel: +46 31 13 17 67