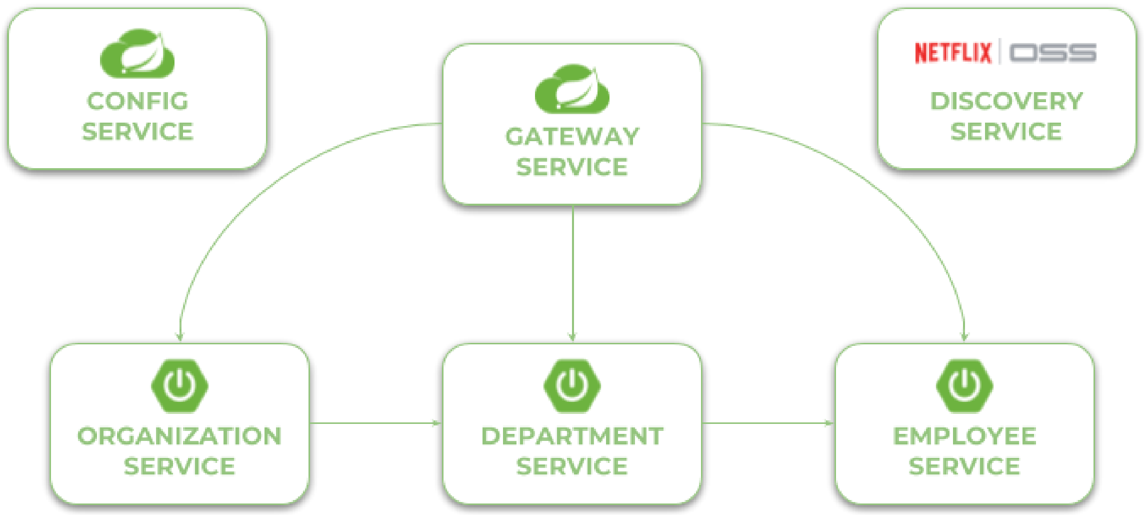
Piotr's TechBlog

Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud

There are many articles on my blog about microservices with Spring Boot and Spring Cloud. The main purpose of this article is to provide a brief summary of the most important components provided by these frameworks that help you in creating microservices. The topics covered in this article are:

- Using **Spring Boot 2.0** in cloud-native development
- Providing service discovery for all microservices with Spring Cloud Netflix **Eureka**
- Distributed configuration with **Spring Cloud Config**
- API Gateway pattern using a new project inside Spring Cloud: **Spring Cloud Gateway**
- Correlating logs with **Spring Cloud Sleuth**

Before we proceed to the source code, let’s take a look on the following diagram. It illustrates the architecture of our sample system. We have three independent microservices, which register themself in service discovery, fetch properties from configuration service and communicate with each other. The whole system is hidden behind API gateway.



Currently, the newest version of Spring Cloud is `Finchley.M9`. This version of `spring-cloud-dependencies` should be declared as a BOM for dependency management.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.M9</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Now, let’s consider the further steps to be taken in order to create working microservices-based system using Spring Cloud. We will begin from **Configuration Server**.

> *The source code of sample applications presented in this article is available on GitHub in repository https://github.com/piomin/sample-spring-microservices-new.git.*

Step 1. Building configuration server with Spring Cloud Config

To enable Spring Cloud Config feature for an application, first include `spring-cloud-config-server` to your project dependencies.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Then enable running embedded configuration server during application boot use `@EnableConfigServer` annotation.

```
@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(ConfigApplication.class).run(args
    }

}
```

By default Spring Cloud Config Server store the configuration data inside Git repository. This is very good choice in production mode, but for the sample purpose file system backend will be enough. It is really easy to start with config server, because we can place all the properties in the classpath. Spring Cloud Config by default search for property sources inside the following locations: `classpath:/, classpath:/config, file:./, file:./config`.

We place all the property sources inside `src/main/resources/config`. The YAML filename will be the same as the name of service. For example, YAML file for discovery-service will be located here: `src/main/resources/config/discovery-service.yml`.

And last two important things. If you would like to start config server with file system backend you have activate Spring Boot profile native. It may be achieved by setting parameter `--spring.profiles.active=native` during application boot. I have also changed the default config server port (8888) to **8061** by setting property `server.port` in `bootstrap.yml` file.

Step 2. Building service discovery with Spring Cloud Netflix Eureka

More to the point of configuration server. Now, all other applications, including discovery-service, need to add `spring-cloud-starter-config` dependency in order to enable config client. We also have to include dependency to `spring-cloud-starter-netflix-eureka-server`.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId
</dependency>
```

Then you should enable running embedded discovery server during application boot by setting `@EnableEurekaServer` annotation on the main class.

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder(DiscoveryApplication.class).run(a
    }

}
```

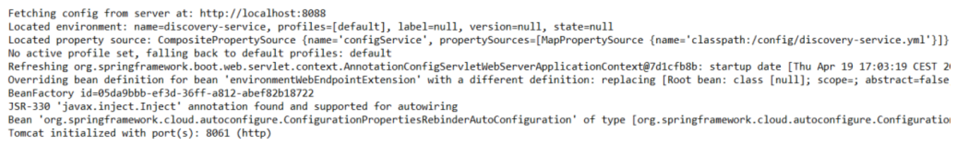
Application has to fetch property source from configuration server. The minimal configuration required on the client side is an application name and config server's connection settings.

```
spring:
  application:
    name: discovery-service
  cloud:
    config:
      uri: http://localhost:8088
```

As I have already mentioned, the configuration file `discovery-service.yml` should be placed inside `config-service` module. However, it is required to say a few words about the configuration visible below. We have changed Eureka running port from default value (8761) to **8061**. For standalone Eureka instance we have to disable registration and fetching registry.

```
server:
  port: 8061

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/
```

Now, when you are starting your application with embedded Eureka server you should see the following logs.



```
Fetching config from server at: http://localhost:8088
Located environment: name=discovery-service, profiles=[default], label=null, version=null, state=null
Located property source: CompositePropertySource {name='configService', propertySources=[MapPropertySource {name='classpath:/config/discovery-service.yml'}]}
No active profile set, falling back to default profiles: default
Refreshing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@7d1cfb8b: startup date [Thu Apr 19 17:03:19 CEST 2
Overriding bean definition for bean 'environmentWebEndpointExtension' with a different definition: replacing [Root bean: class [null]; scope=; abstract=false
BeanFactory id=05da9bbb-ef3d-36ff-a812-abef82b18722
JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration' of type [org.springframework.cloud.autoconfigure.Configuratio
Tomcat initialized with port(s): 8061 (http)
```

Once you have succesfully started application you may visit Eureka Dashboard available under address http://localhost:8061/.

Step 3. Building microservice using Spring Boot and Spring Cloud

Our microservice has te perform some operations during boot. It needs to fetch configuration from `config-service` , register itself in discovery-service, expose HTTP API and automatically generate API documentation. To enable all these mechanisms we need to include some dependencies in `pom.xml` . To enable config client we should include starter `spring-cloud-starter-config` . Discovery client will be enabled for microservice after including `spring-cloud-starter-netflix-eureka-client` and annotating the main class with `@EnableDiscoveryClient` . To force Spring Boot application generating API documentation we should include `springfox-swagger2` dependency and add annotation `@EnableSwagger2`.

Here is the full list of dependencies defined for my sample microservice.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactI
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.8.0</version>
</dependency>
```

And here is the main class of application that enables **Discovery Client** and **Swagger2** for the microservice.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableSwagger2
public class EmployeeApplication {

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

    @Bean
    public Docket swaggerApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
                .apis(RequestHandlerSelectors.basePackage("pl.piomin.
                .paths(PathSelectors.any())
            .build()
            .apiInfo(new ApiInfoBuilder().version("1.0").title("Emplo
    }

    ...

}
```

Application has to fetch configuration from a remote server, so we should only provide `bootstrap.yml` file with service name and server URL. In fact, this is the example of **Config First Bootstrap** approach, when an application first connects to a config server and takes a discovery server address from a remote property source. There is also **Discovery First Bootstrap**, where a config server address is fetched from a discovery server.

```
spring:
  application:
    name: employee-service
  cloud:
    config:
      uri: http://localhost:8088
```

There is no much configuration settings. Here's application's configuration file stored on a remote server. It stores only HTTP running port and Eureka URL. However, I also placed file `employee-service-instance2.yml` on remote config server. It sets different HTTP port for application, so you can esily run two instances of the same service locally basing on remote properties. Now, you may run the second instance of `employee-service` on port **9090** after passing argument `spring.profiles.active=instance2` during an application startup. With default settings you will start the microservice on port **8090**.

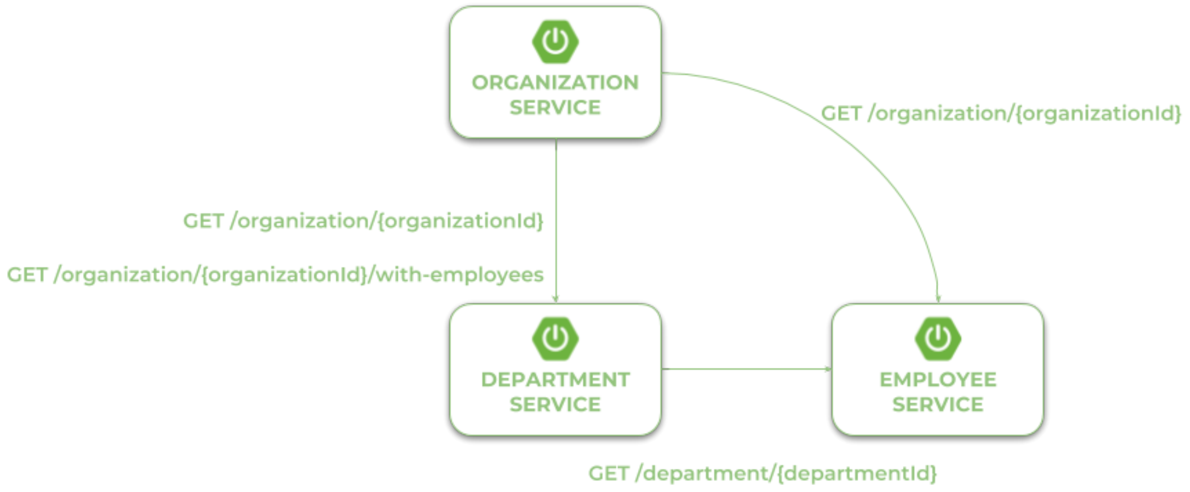
```
server:
  port: 9090

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8061/eureka/
```

Here's the code with implementation of REST controller class. It provides an implementation for adding new employee and searching for employee using different filters.

```
@RestController
public class EmployeeController {

    private static final Logger LOGGER = LoggerFactory.getLogger(Empl

    @Autowired
    EmployeeRepository repository;

    @PostMapping
    public Employee add(@RequestBody Employee employee) {
        LOGGER.info("Employee add: {}", employee);
        return repository.add(employee);
    }

    @GetMapping("/{id}")
    public Employee findById(@PathVariable("id") Long id) {
        LOGGER.info("Employee find: id={}", id);
        return repository.findById(id);
    }

    @GetMapping
    public List findAll() {
        LOGGER.info("Employee find");
        return repository.findAll();
    }

    @GetMapping("/department/{departmentId}")
    public List findByDepartment(@PathVariable("departmentId") Long d
        LOGGER.info("Employee find: departmentId={}", departmentId);
        return repository.findByDepartment(departmentId);
    }

    @GetMapping("/organization/{organizationId}")
    public List findByOrganization(@PathVariable("organizationId") Lo
        LOGGER.info("Employee find: organizationId={}", organizationI
        return repository.findByOrganization(organizationId);
    }

}
```

Step 4. Communication between microservice with Spring Cloud Open Feign

Our first microservice has been created and started. Now, we will add other microservices that communicate with each other. The following diagram illustrates the communication flow between three sample microservices: `organization-service`, `department-service` and `employee-service`. Microservice `organization-service` collect list of departments with (`GET /organization/{organizationId}/with-employees`) or without employees (`GET /organization/{organizationId}`) from `department-service`, and list of employees without dividing them into different departments directly from `employee-service`. Microservice `department-service` is able to collect list of employees assigned to the particular department.


ORGANIZATION SERVICE
GET /organization/{organizationId}
GET /organization/{organizationId}
GET /organization/{organizationId}/with-employees
DEPARTMENT SERVICE
EMPLOYEE SERVICE
GET /department/{departmentId}

In the scenario described above both `organization-service` and `department-service` have to localize other microservices and communicate with them. That's why we need to include additional dependency for those modules: spring-cloud-starter-openfeign. Spring Cloud Open Feign is a declarative REST client that used Ribbon client-side load balancer in order to communicate with other microservice.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

The alternative solution to Open Feign is Spring `RestTemplate` with `@LoadBalanced`. However, Feign provides more ellegant way of defining client, so I prefer it instead of `RestTemplate`. After including the required dependency we should also enable Feign clients using `@EnableFeignClients` annotation.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableSwagger2
public class OrganizationApplication {

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

    ...

}
```

Now, we need to define client's interfaces. Because `organization-service` communicates with two other microservices we should create two interfaces, one per single microservice. Every client's interface should be annotated with `@FeignClient`. One field inside annotation is required – `name`. This name should be the same as the name of target service registered in service discovery. Here's the interface of the client that calls endpoint `GET /organization/{organizationId}` exposed by `employee-service`.

```
@FeignClient(name = "employee-service")
public interface EmployeeClient {

    @GetMapping("/organization/{organizationId}")
    List findByOrganization(@PathVariable("organizationId") Long organ

}
```

The second client's interface available inside `organization-service` calls two endpoints from `department-service`. First of them `GET /organization/{organizationId}` returns organization only with the list of available departments, while the second `GET /organization/{organizationId}/with-employees` return the same set of data including the list employees assigned to every department.

```
@FeignClient(name = "department-service")
public interface DepartmentClient {

    @GetMapping("/organization/{organizationId}")
    public List findByOrganization(@PathVariable("organizationId") Lo

    @GetMapping("/organization/{organizationId}/with-employees")
    public List findByOrganizationWithEmployees(@PathVariable("organi

}
```

Finally, we have to inject Feign client's beans to the REST controller. Now, we may call the methods defined inside `DepartmentClient` and `EmployeeClient`, which is equivalent to calling REST endpoints.

```
@RestController
public class OrganizationController {

    private static final Logger LOGGER = LoggerFactory.getLogger(Orga
```

```java

    @Autowired
    OrganizationRepository repository;
    @Autowired
    DepartmentClient departmentClient;
    @Autowired
    EmployeeClient employeeClient;

    ...

    @GetMapping("/{id}")
    public Organization findById(@PathVariable("id") Long id) {
        LOGGER.info("Organization find: id={}", id);
        return repository.findById(id);
    }

    @GetMapping("/{id}/with-departments")
    public Organization findByIdWithDepartments(@PathVariable("id") L
        LOGGER.info("Organization find: id={}", id);
        Organization organization = repository.findById(id);
        organization.setDepartments(departmentClient.findByOrganizati
        return organization;
    }

    @GetMapping("/{id}/with-departments-and-employees")
    public Organization findByIdWithDepartmentsAndEmployees(@PathVari
        LOGGER.info("Organization find: id={}", id);
        Organization organization = repository.findById(id);
        organization.setDepartments(departmentClient.findByOrganizati
        return organization;
    }

    @GetMapping("/{id}/with-employees")
    public Organization findByIdWithEmployees(@PathVariable("id") Lon
        LOGGER.info("Organization find: id={}", id);
        Organization organization = repository.findById(id);
        organization.setEmployees(employeeClient.findByOrganization(o
        return organization;
    }

}
```

Step 5. Building API gateway using Spring Cloud Gateway

Spring Cloud Gateway is relatively new Spring Cloud project. It is built on top of Spring Framework 5, Project **Reactor** and **Spring Boot 2.0**. It requires the **Netty** runtime provided by Spring Boot and Spring Webflux. This is really nice alternative to Spring Cloud Netflix Zuul, which has been the only one Spring Cloud project providing API gateway for microservices until now.

API gateway is implemented inside module `gateway-service`. First, we should include starter `spring-cloud-starter-gateway` to the project dependencies.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

We also need to have discovery client enabled, because `gateway-service` integrates with Eureka in order to be able to perform routing to the downstream services. Gateway will also expose API specification of all the endpoints exposed by our sample microservices. That's why we enabled Swagger2 also on the gateway.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableSwagger2
public class GatewayApplication {

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

}
```

Spring Cloud Gateway provides three basic components used for configuration: routes, predicates and filters. **Route** is the basic building block of the gateway. It contains destination URI and list of defined predicates and filters. **Predicate** is responsible for matching on anything from the incoming HTTP request, such as headers or parameters. **Filter** may modify request and response before and after sending it to downstream services. All these components may be set using configuration properties. We will create and place on the confiration server file gateway-service.yml with the routes defined for our sample microservices.

But first, we should enable integration with discovery server for the routes by setting property `spring.cloud.gateway.discovery.locator.enabled` to true. Then we may proceed to defining the route rules. We use the Path Route Predicate Factory for matching the incoming requests, and the RewritePath GatewayFilter Factory for modifying the requested path to adapt it to the format exposed by downstream services. The uri parameter specifies the name of target service registered in discovery server. Let's take a look on the following routes definition. For example, in order to make `organization-service` available on gateway under path

`/organization/**` , we should define predicate `Path=/organization/**` , and then strip prefix `/organization` from the path, because the target service is exposed under path `/**` . The address of target service is fetched for Eureka basing uri value `lb://organization-service` .

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
      routes:
      - id: employee-service
        uri: lb://employee-service
        predicates:
        - Path=/employee/**
        filters:
        - RewritePath=/employee/(?.*), /$\{path}
      - id: department-service
        uri: lb://department-service
        predicates:
        - Path=/department/**
        filters:
        - RewritePath=/department/(?.*), /$\{path}
      - id: organization-service
        uri: lb://organization-service
        predicates:
        - Path=/organization/**
        filters:
        - RewritePath=/organization/(?.*), /$\{path}
```

Step 6. Enabling API specification on gateway using Swagger2

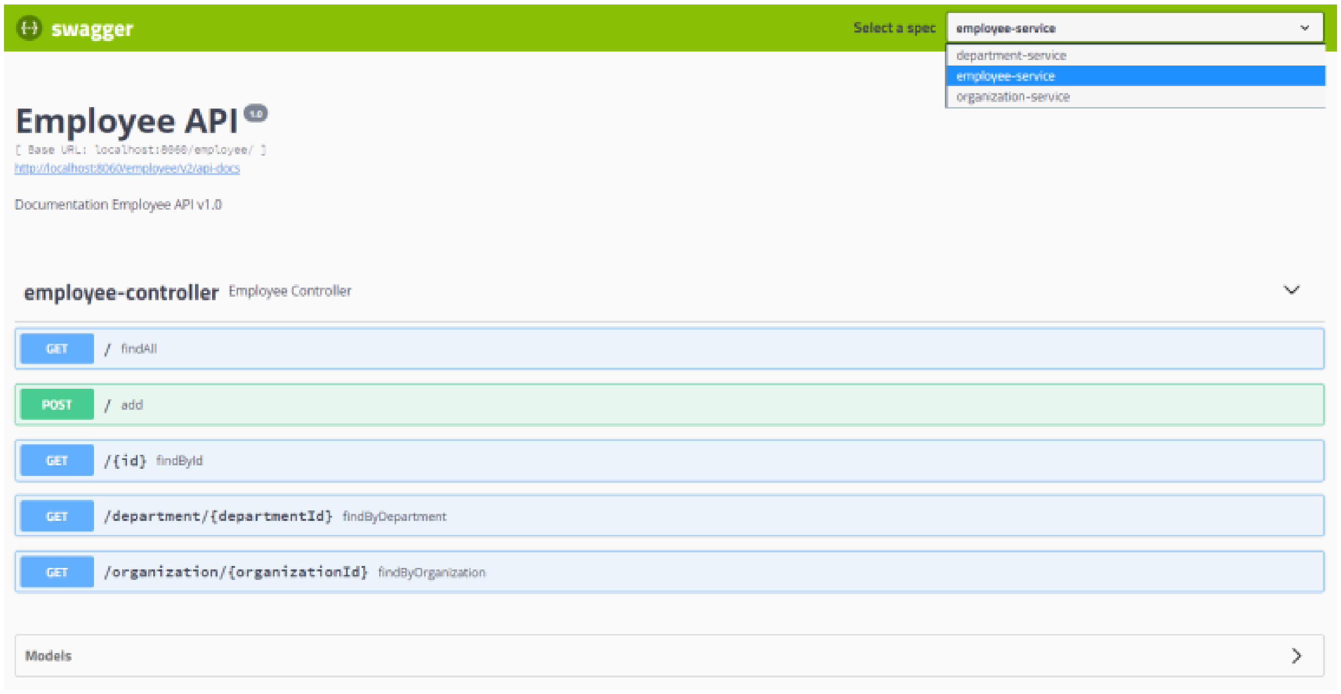
Every Spring Boot microservice that is annotated with `@EnableSwagger2` exposes Swagger API documentation under path `/v2/api-docs` . However, we would like to have that documentation located in the single place – on API gateway. To achieve it we need to provide bean implementing `SwaggerResourcesProvider` interface inside `gateway-service` module. That bean is responsible for defining list storing locations of Swagger resources, which should be displayed by the application. Here's the implementation of `SwaggerResourcesProvider` that takes the required locations from service discovery basing on the Spring Cloud Gateway configuration properties.

Unfortunately, SpringFox Swagger still does not provide support for Spring WebFlux. It means that if you include SpringFox Swagger dependencies to the project application will fail to start… I hope the support for WebFlux will be available soon, but now we have to use Spring Cloud Netflix Zuul as a gateway, if we would like to run embedded Swagger2 on it.

I created module `proxy-service` that is an alternative API gateway based on Netflix Zuul to `gateway-service` based on Spring Cloud Gateway. Here's a bean with SwaggerResourcesProvider implementation available inside `proxy-service`. It uses `ZuulProperties` bean to dynamically load routes definition into the bean.

```
@Configuration
public class ProxyApi {

    @Autowired
    ZuulProperties properties;

    @Primary
    @Bean
    public SwaggerResourcesProvider swaggerResourcesProvider() {
        return () -> {
            List resources = new ArrayList();
            properties.getRoutes().values().stream()
                    .forEach(route -> resources.add(createResource(ro
            return resources;
        };
    }

    private SwaggerResource createResource(String name, String locati
        SwaggerResource swaggerResource = new SwaggerResource();
        swaggerResource.setName(name);
        swaggerResource.setLocation("/" + location + "/v2/api-docs");
        swaggerResource.setSwaggerVersion(version);
        return swaggerResource;
    }

}
```

Here's Swagger UI for our sample microservices system available under address http://localhost:8060/swagger-ui.html.



Employee API 1.0

[Base URL: localhost:8060/employee/]

http://localhost:8060/employee/v2/api-docs

Documentation Employee API v1.0

employee-controller Employee Controller

GET / findAll

POST / add

GET /{id} findById

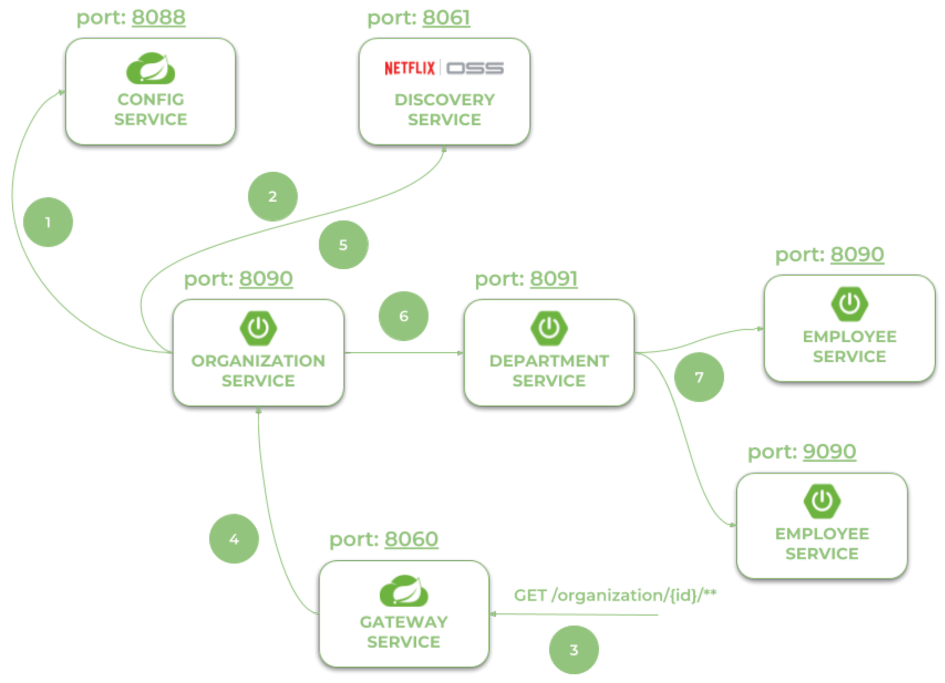
GET /department/{departmentId} findByDepartment

GET /organization/{organizationId} findByOrganization

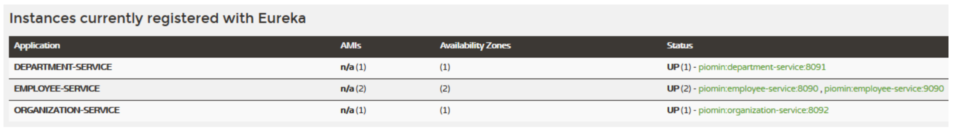
Models

Step 7. Running applications

Let’s take a look on the architecture of our system visible on the following diagram. We will discuss it from the `organization-service` point of view. After starting `organization-service` connects to `config-service` available under address localhost:8088 (1). Basing on remote configuration settings it is able to register itself in Eureka (2). When the endpoint of `organization-service` is invoked by external client via gateway (3) available under address localhost:8060, the request is forwarded to instance of `organization-service` basing on entries from service discovery (4). Then `organization-service` lookup for address of `department-service` in Eureka (5), and call its endpoint (6). Finally `department-service` calls endpont from `employee-service`. The request as load balanced between two available instance of `employee-service` by **Ribbon** (7).



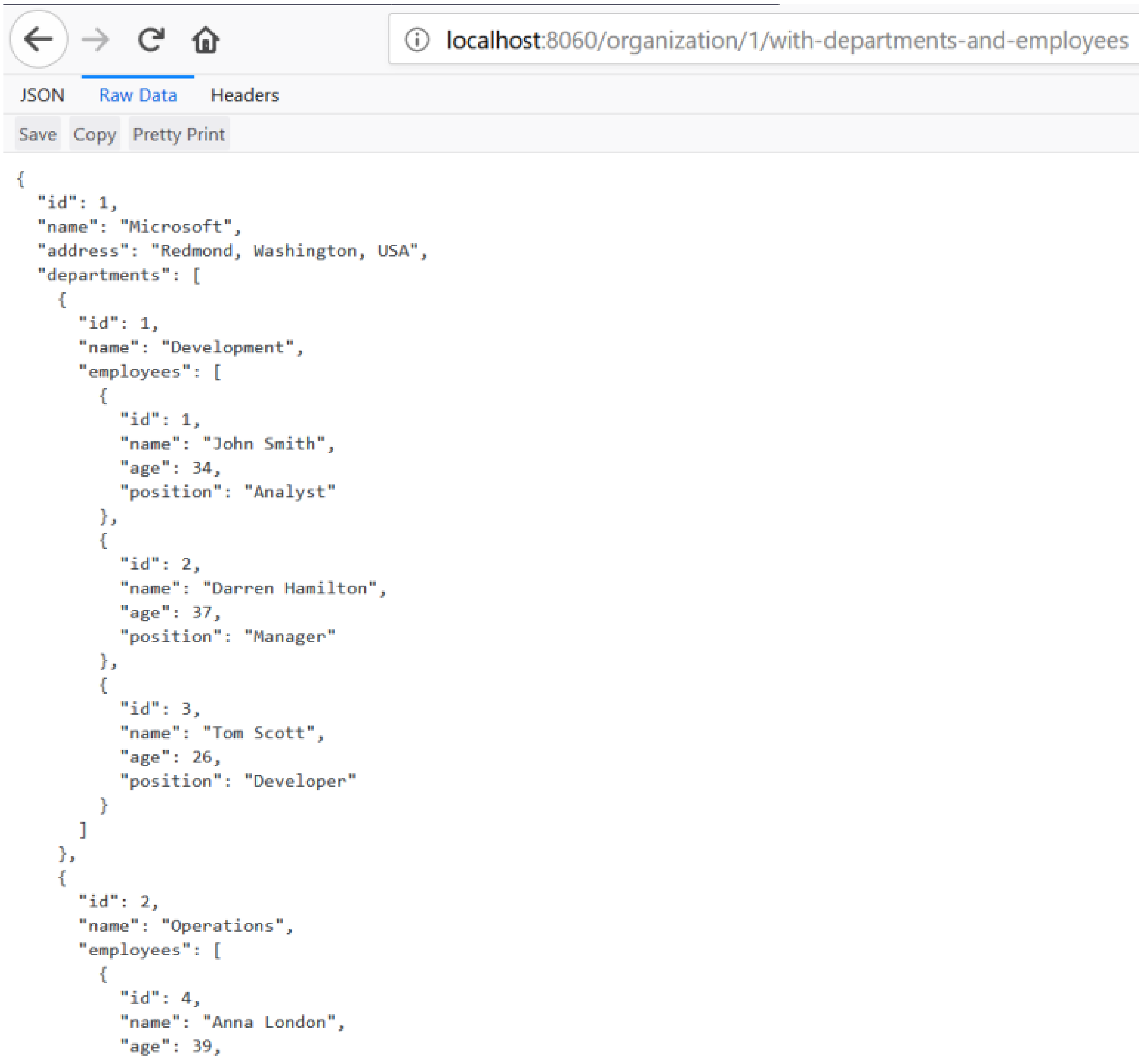
Let’s take a look on the Eureka Dashboard available under address *http://localhost:8061*. There are four instances of microservices registered there: a single instance of `organization-service` and `department-service` , and two instances of `employee-service` .



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
DEPARTMENT-SERVICE	**n/a** (1)	(1)	**UP** (1) - piomin:department-service:8091
EMPLOYEE-SERVICE	**n/a** (2)	(2)	**UP** (2) - piomin:employee-service:8090 , piomin:employee-service:9090
ORGANIZATION-SERVICE	**n/a** (1)	(1)	**UP** (1) - piomin:organization-service:8092

Now, let’s call endpoint ***http://localhost:8060/organization/1/with-departments-and-employees***.


localhost:8060/organization/1/with-departments-and-employees
JSON
Raw Data
Headers
Save
Copy
Pretty Print

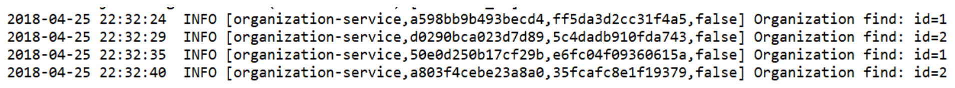
```
{
  "id": 1,
  "name": "Microsoft",
  "address": "Redmond, Washington, USA",
  "departments": [
    {
      "id": 1,
      "name": "Development",
      "employees": [
        {
          "id": 1,
          "name": "John Smith",
          "age": 34,
          "position": "Analyst"
        },
        {
          "id": 2,
          "name": "Darren Hamilton",
          "age": 37,
          "position": "Manager"
        },
        {
          "id": 3,
          "name": "Tom Scott",
          "age": 26,
          "position": "Developer"
        }
      ]
    },
    {
      "id": 2,
      "name": "Operations",
      "employees": [
        {
          "id": 4,
          "name": "Anna London",
          "age": 39,
```

Step 8. Correlating logs between independent microservices using Spring Cloud Sleuth

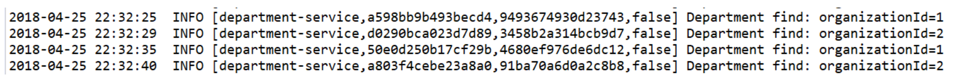
Correlating logs between different microservice using Spring Cloud Sleuth is very easy. In fact, the only thing you have to do is to add starter `spring-cloud-starter-sleuth` to the dependencies of every single microservice and gateway.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

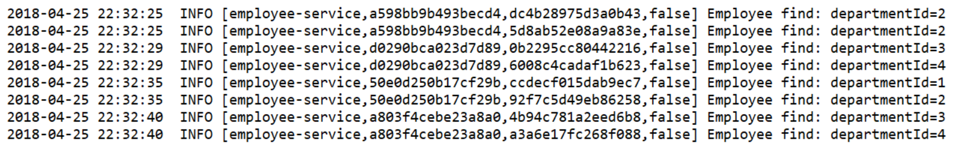
For clarification we will change default log format a little to: `%d{yyyy-MM-dd HH:mm:ss} ${LOG_LEVEL_PATTERN:-%5p} %m%n` . Here are the logs generated by our three sample miccroservices. There are four entries inside braces `[]` generated by Spring Cloud Stream. The most important for us is the second entry, which indicates on `traceId` , that is set once per incoming HTTP request on the edge of the system.



```
2018-04-25 22:32:24  INFO [organization-service,a598bb9b493becd4,ff5da3d2cc31f4a5,false] Organization find: id=1
2018-04-25 22:32:29  INFO [organization-service,d0290bca023d7d89,5c4dadb910fda743,false] Organization find: id=2
2018-04-25 22:32:35  INFO [organization-service,50e0d250b17cf29b,e6fc04f09360615a,false] Organization find: id=1
2018-04-25 22:32:40  INFO [organization-service,a803f4cebe23a8a0,35fcafc8e1f19379,false] Organization find: id=2
```



```
2018-04-25 22:32:25  INFO [department-service,a598bb9b493becd4,9493674930d23743,false] Department find: organizationId=1
2018-04-25 22:32:29  INFO [department-service,d0290bca023d7d89,3458b2a314bcb9d7,false] Department find: organizationId=2
2018-04-25 22:32:35  INFO [department-service,50e0d250b17cf29b,4680ef976de6dc12,false] Department find: organizationId=1
2018-04-25 22:32:40  INFO [department-service,a803f4cebe23a8a0,91ba70a6d0a2c8b8,false] Department find: organizationId=2
```



```
2018-04-25 22:32:25  INFO [employee-service,a598bb9b493becd4,dc4b28975d3a0b43,false] Employee find: departmentId=2
2018-04-25 22:32:25  INFO [employee-service,a598bb9b493becd4,5d8ab52e08a9a83e,false] Employee find: departmentId=2
2018-04-25 22:32:29  INFO [employee-service,d0290bca023d7d89,0b2295cc80442216,false] Employee find: departmentId=3
2018-04-25 22:32:29  INFO [employee-service,d0290bca023d7d89,6008c4cadaf1b623,false] Employee find: departmentId=4
2018-04-25 22:32:35  INFO [employee-service,50e0d250b17cf29b,ccdecf015dab9ec7,false] Employee find: departmentId=1
2018-04-25 22:32:35  INFO [employee-service,50e0d250b17cf29b,92f7c5d49eb86258,false] Employee find: departmentId=2
2018-04-25 22:32:40  INFO [employee-service,a803f4cebe23a8a0,4b94c781a2eed6b8,false] Employee find: departmentId=3
2018-04-25 22:32:40  INFO [employee-service,a803f4cebe23a8a0,a3a6e17fc268f088,false] Employee find: departmentId=4
```

Related

[The Future of Spring Cloud Microservices After Netflix Era]

In "microservices"

[Quick Guide to Microservices with Kubernetes, Spring Boot 2.0 and Docker]

In "containers"

[Part 2: Creating microservices – monitoring with Spring Cloud Sleuth, ELK and Zipkin]

In "spring cloud"

Author: Piotr Mińkowski

IT Architect, Java Software Developer [View all posts by Piotr Mińkowski]

Piotr Mińkowski / April 26, 2018 / microservices, spring boot, spring cloud / Eureka, microservices, Netflix OSS, spring boot, spring cloud, Spring Cloud Gateway, spring-cloud-config, swagger2, Zuul

31 thoughts on "Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud"

Albert

April 28, 2018 at 3:19 pm

This is a great tutorial, thanks!
If possible, could you add Auth Server (OAuth2) in? That would make this cover all …
🙂

★ Like

Piotr Mińkowski

April 28, 2018 at 10:42 pm

Hi,
Thanks 🙂 In this tutorial there is no auth sevrer, but there some other articles in my blog about microservices and oauth2:
https://piotrminkowski.wordpress.com/2017/02/22/microservices-security-with-oauth2/
https://piotrminkowski.wordpress.com/2017/12/01/part-2-microservices-security-with-oauth2/

★ Like

Sebas

May 2, 2018 at 12:51 am

Hi, nice tutorial! Thanks!
I think you have a typo in step 2:

"Then you should enable running embedded discovery server during application boot by setting @EnableConfigServer annotation on the main class."

It should say "… by setting @EnableEurekaServer …", shouldn't it?

★ Like

Piotr Mińkowski

May 4, 2018 at 9:51 am

Thanks 🙂 Of course – I changed it

★ Like

sluk3r

May 2, 2018 at 7:23 am

hi, I am new to Spring-cloud, could tell me to how run the application, please?

★ Like

Piotr Mińkowski

May 4, 2018 at 9:53 am

You can run the main class using your IDE or just build the whole project using Maven and then run it like a standard java app with java -jar ….

★ Like

Pingback: 使用Spring Boot 2.0, Eureka, and Spring Cloud快速搭建Spring微服务 – 快速迭代

Stomer

May 3, 2018 at 7:08 am

awesome, that great!

★ Like

Stomer

May 3, 2018 at 7:09 am

If possible, could you show cases configuration to auto scale service to many instances

★ Like

Piotr Mińkowski

May 4, 2018 at 9:58 am

you can use some third-party tolls. For example HashiCorp's Nomad. Here's my article about it: https://piotrminkowski.wordpress.com/2018/04/17/deploying-spring-cloud-microservices-on-hashicorps-nomad/

★ Like

Pingback: Reactive Microservices with Spring WebFlux and Spring Cloud – Piotr's TechBlog

---

Sridhar

May 6, 2018 at 10:59 am

We are using Spring Cloud Zuul in our production environment. We are planning to upgrade to Spring Boot 2.0 and Spring Cloud Finchley.RELEASE when Spring Cloud Finchley is officially released(GA Release).

Can we consider replacing Spring Cloud Zuul with Spring Cloud Gateway. Since Spring Cloud Gateway is based on a non-blocking api I feel it should give better performance thatn Spring cloud Zuul.

★ Like

Piotr Mińkowski

May 6, 2018 at 5:05 pm

I think it's a good idea 🙂

★ Like

Vivek

July 17, 2018 at 7:46 am

Hi very nice explanation. It hellp me to start writing microservices in easy go. I did not understand the proxy service role yet. can you please explain with the flow of the above example?
Thanks in Advance.

★ Like

Piotr Mińkowski

July 17, 2018 at 8:40 am

Hi,
Thanks 🙂 Proxy service is just an API Gateway, which guarantee that all the microservices are visible outside the system under single address and port. It proxies the request to the downstream services by context path

★ Like

Jeff

August 9, 2018 at 9:59 pm

You response would apply to the gateway-service, right? If I understand correctly, the proxy-service is a workaround to act as a gateway for Swagger UI since it doesn't support WebFlux.

★ Like

Piotr Mińkowski

August 9, 2018 at 8:00 pm

Exactly

★ Like

Cyrano

August 10, 2018 at 10:02 pm

very good doc. thank you very much. it really help me to have a better insight on implementing microservices with spring-boot.

★ Like

Piotr Mińkowski

August 11, 2018 at 8:03 pm

Thanks 🙂

★ Like

Savani

August 15, 2018 at 5:41 pm

Hi Author – Thanks for very nice post, but I 'm getting error mentioned here : https://stackoverflow.com/questions/51863731/caused-by-java-lang-illegalstateexception-you-need-to-configure-a-uri-for-the.
Could you please help me with this?

★ Like

Ivo

August 28, 2018 at 1:45 pm

Hello Savani

I'm not the author but I'll try to help you.
Have you placed

spring:
profiles:
active: native

As the answer to the topic have said?

★ Like

Pingback: Microservices made easy with Spring Boot | This Technology Life

Suzan

October 6, 2018 at 2:49 pm

How do you run this application with the Dockerfile?
Do I need to run profile "native" on Docker too?

★ Like

Piotr Mińkowski

October 30, 2018 at 10:49 pm

For config server? Yes

★ Like

Vitor Saad

October 11, 2018 at 1:31 am

Congratulations on this project.
But I did not understand where Zuul fits.
The gateway project seems to be the single point of entry .. so what is zuul's function?

★ Like

Piotr Mińkowski

October 30, 2018 at 10:46 pm

I used Zuul, because Spring Cloud Gateway didn't support SpringFox Swagger. Maybe they have already fixed it in springfox library.

★ Like

Travis

February 28, 2019 at 6:55 pm

Thank you for this guide.

I was wondering why use Feign for inter-service communication? Couldn't the Organization service contact the Department service through the Gateway service?

★ Like

Piotr Mińkowski

March 1, 2019 at 11:25 am

Why not? Feign is integrated with discovery so why you prefer communication through proxy?

★ Like

swapnil skate

April 9, 2019 at 12:14 pm

It may be achieved by setting parameter –spring.profiles.active=native during application boot

I did not understand this. Does this mean I have to create applications-native.properties file and put it inside config-service project and run the app. Since I am getting error when I rune with spring-boot:run -Dspring.profiles.active=native but still it gives me an error.

Can you please help me on this.

★ Like

Piotr Mińkowski

April 9, 2019 at 6:38 pm

It is not VM arg. You should set –spring.profiles.active=native

★ Like

swapnil skate

April 10, 2019 at 2:53 pm

Sorry to ask it again. But do I need to set
spring:
profiles.active: native

in bootstrap.yml file?

Can you help me to understand how I can run config service.

Thanks

★ Like

This site uses Akismet to reduce spam. Learn how your comment data is processed.