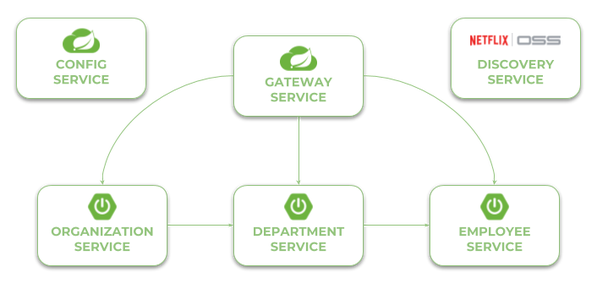
# Quick Guide to Microservices With Spring Boot 2.0, Eureka, and Spring Cloud

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 at the following diagram. It illustrates the architecture of our sample system. We have three independent microservices, which register themselves in service discovery, fetch properties from the configuration service, and communicate with each other. The whole system is hidden behind an 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 version="1.0" encoding="UTF-8"?>

<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 further steps to be taken in order to create a working microservices-based system using Spring Cloud. We will begin from the **Configuration Server**.

The source code of sample applications presented in this article is available in this [GitHub 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.

<?xml version="1.0" encoding="UTF-8"?>

<dependency>

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

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

</dependency>

Then enable running an embedded configuration server during application boot using the @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 stores the configuration data inside the Git repository. This is a very good choice in production mode, but for the sample file system backend, it 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 the service. For example, the YAML file for discovery-service will be located here: src/main/resources/config/discovery-service.yml.

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

**Step 2. Building Service Discovery With Spring Cloud Netflix Eureka**

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

<?xml version="1.0" encoding="UTF-8"?>

<dependency>

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

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

</dependency>

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

@SpringBootApplication

@EnableEurekaServer

public class DiscoveryApplication {

public static void main(String[] args) {

new SpringApplicationBuilder(DiscoveryApplication.class).run(args);

}

}

The application has to fetch a property source from the 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 the config-service module. However, I must say a few words about the configuration visible below. We have changed the Eureka running port from the default value (8761) to **8061**. For a standalone Eureka instance, we have to disable registration and fetching registry.

server:

port: 8061

eureka:

instance:

hostname: localhost

client:

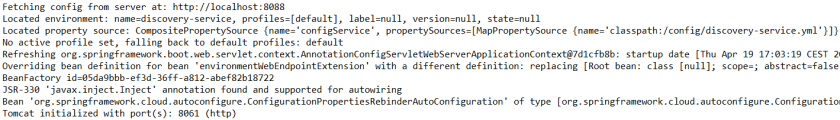
registerWithEureka: false

fetchRegistry: false

serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

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



Once you have successfully started the application, you may visit the Eureka Dashboard, available at the address <http://localhost:8061/>.

## Step 3. Building a Microservice Using Spring Boot and Spring Cloud

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

</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 the 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.services.employee.controller"))

.paths(PathSelectors.any())

.build()

.apiInfo(new ApiInfoBuilder().version("1.0").title("Employee API").description("Documentation Employee API v1.0").build());

}

...

}

The application has to fetch configuration from a remote server, so we should only provide a bootstrap.ymlfile with the service name and server URL. In fact, this is an example of the **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 are not many configuration settings. Here's the application's configuration file stored on a remote server. It stores only the HTTP running port and Eureka URL. However, I also placed the file employee-service-instance2.yml on the remote config server. It sets a different HTTP port for application, so you can easily run two instances of the same service locally based on remote properties. Now, you may run the second instance of employee-service on port **9090** after passing the argument spring.profiles.active=instance2during 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 the implementation of the REST controller class. It provides an implementation for adding new employees and searching for employees using different filters.

@RestController

public class EmployeeController {

private static final Logger LOGGER = LoggerFactory.getLogger(EmployeeController.class);

@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 departmentId) {

LOGGER.info("Employee find: departmentId={}", departmentId);

return repository.findByDepartment(departmentId);

}

@GetMapping("/organization/{organizationId}")

public List findByOrganization(@PathVariable("organizationId") Long organizationId) {

LOGGER.info("Employee find: organizationId={}", organizationId);

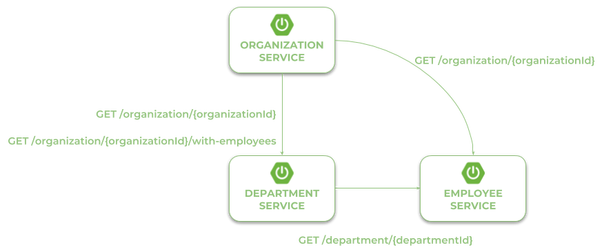
return repository.findByOrganization(organizationId);

}

}

## Step 4. Communication Between Microservices 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 a list of employees assigned to the particular department.



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 an additional dependency for those modules: spring-cloud-starter-openfeign. Spring Cloud Open Feign is a declarative REST client that uses a Ribbon client-side load balancer in order to communicate with other microservices.

<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 a more elegant way of defining clients, so I prefer it instead of RestTemplate. After including the required dependency, we should also enable Feign clients using the @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 the 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 in the annotation is required - name. This name should be the same as the name of the target service registered in service discovery. Here's the interface of the client that calls the endpoint GET /organization/{organizationId} exposed by employee-service.

@FeignClient(name = "employee-service")

public interface EmployeeClient {

@GetMapping("/organization/{organizationId}")

List findByOrganization(@PathVariable("organizationId") Long organizationId);

}

The second client's interface available inside organization-service calls two endpoints from department-service. The first of them, GET /organization/{organizationId}, returns the organization only with the list of available departments, while the second,GET /organization/{organizationId}/with-employees, returns 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") Long organizationId);

@GetMapping("/organization/{organizationId}/with-employees")

public List findByOrganizationWithEmployees(@PathVariable("organizationId") Long organizationId);

}

Finally, we have to inject the Feign client's beans into 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(OrganizationController.class);

@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") Long id) {

LOGGER.info("Organization find: id={}", id);

Organization organization = repository.findById(id);

organization.setDepartments(departmentClient.findByOrganization(organization.getId()));

return organization;

}

@GetMapping("/{id}/with-departments-and-employees")

public Organization findByIdWithDepartmentsAndEmployees(@PathVariable("id") Long id) {

LOGGER.info("Organization find: id={}", id);

Organization organization = repository.findById(id);

organization.setDepartments(departmentClient.findByOrganizationWithEmployees(organization.getId()));

return organization;

}

@GetMapping("/{id}/with-employees")

public Organization findByIdWithEmployees(@PathVariable("id") Long id) {

LOGGER.info("Organization find: id={}", id);

Organization organization = repository.findById(id);

organization.setEmployees(employeeClient.findByOrganization(organization.getId()));

return organization;

}

}

## Step 5. Building API Gateway Using Spring Cloud Gateway

Spring Cloud Gateway is a 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 a really nice alternative to Spring Cloud Netflix Zuul, which has been the only Spring Cloud project providing API gateways for microservices until now.

The API gateway is implemented inside the module gateway-service. First, we should include the 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. The gateway will also expose the API specification of all the endpoints exposed by our sample microservices. That's why we also enabled Swagger2 on the gateway.

@SpringBootApplication

@EnableDiscoveryClient

@EnableSwagger2

public class GatewayApplication {

public static void main(String[] args) {

SpringApplication.run(GatewayApplication.class, args);

}

}

The Spring Cloud Gateway provides three basic components used for configuration: routes, predicates, and filters. A**route** is the basic building block of the gateway. It contains the destination URI and list of defined predicates and filters. The**predicate** is responsible for matching on anything from the incoming HTTP request, such as headers or parameters. The**filter** may modify a 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 it on the configuration server file gateway-service.yml with the routes defined for our sample microservices.

But first, we should enable integration with the discovery server for the routes by setting the property spring.cloud.gateway.discovery.locator.enabled to true. Then we may proceed to define 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 at the following routes definition. For example, in order to make organization-service available on the gateway under the path /organization/\*\*, we should define the predicate Path=/organization/\*\*, and then strip the prefix /organization from the path, because the target service is exposed under path /\*\*. The address of target service is fetched for Eureka based on the 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>.\*), /$\{path}

- id: department-service

uri: lb://department-service

predicates:

- Path=/department/\*\*

filters:

- RewritePath=/department/(?<path>.\*), /$\{path}

- id: organization-service

uri: lb://organization-service

predicates:

- Path=/organization/\*\*

filters:

- RewritePath=/organization/(?<path>.\*), /$\{path}

## Step 6. Enabling API Specification on Gateway Using Swagger2

Every Spring Boot microservice that is annotated with @EnableSwagger2 exposes the Swagger API documentation under the path /v2/api-docs. However, we would like to have that documentation located in the single place - on the API gateway. To achieve it, we need to provide a bean implementing the SwaggerResourcesProvider interface inside the gateway-service module. That bean is responsible for defining the 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 based 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 in the project, the application will fail to start... I hope 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 the 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 the SwaggerResourcesProvider implementation available inside proxy-service. It uses theZuulProperties bean to dynamically load route definitions 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(route.getServiceId(), route.getId(), "2.0")));

return resources;

};

}

private SwaggerResource createResource(String name, String location, String version) {

SwaggerResource swaggerResource = new SwaggerResource();

swaggerResource.setName(name);

swaggerResource.setLocation("/" + location + "/v2/api-docs");

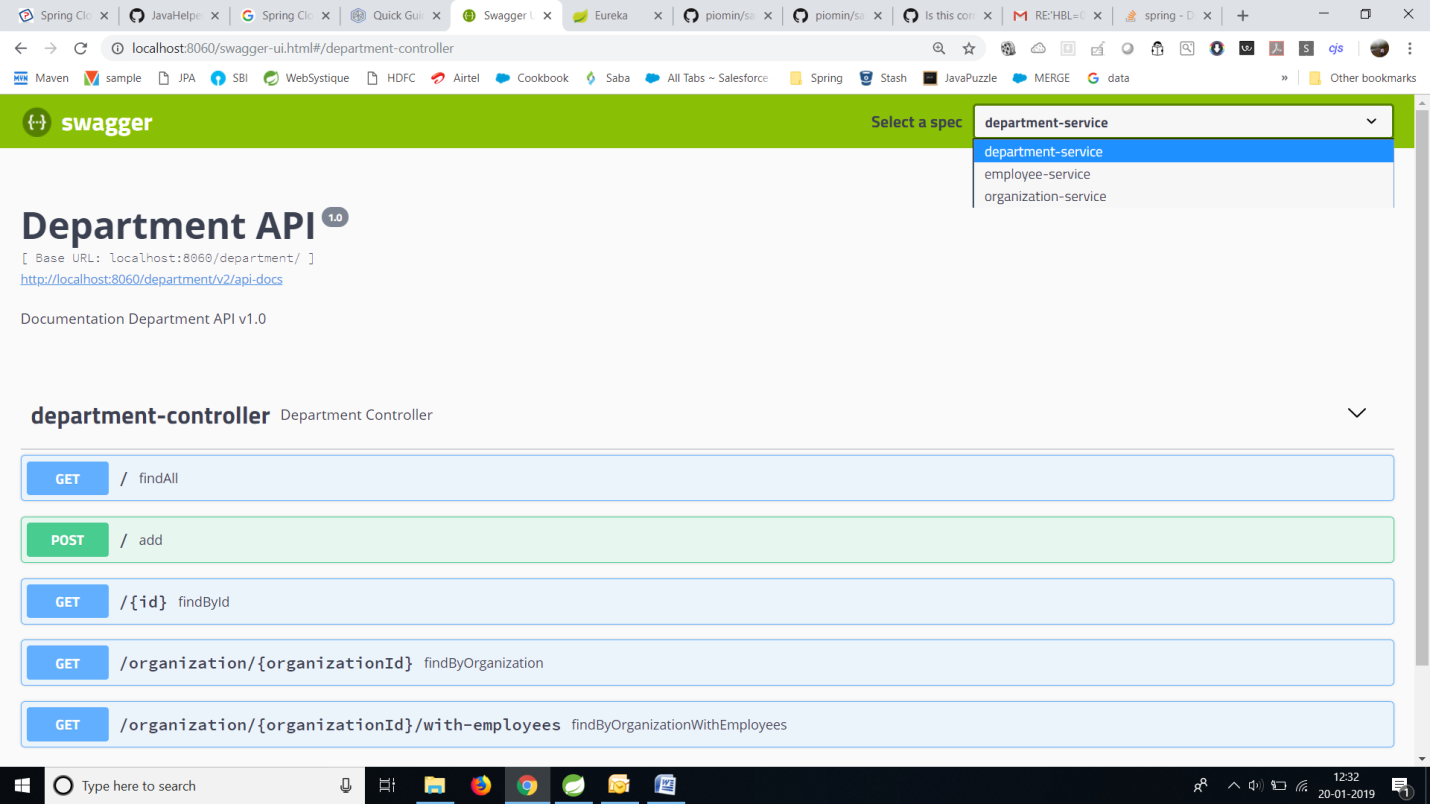
swaggerResource.setSwaggerVersion(version);

return swaggerResource;

}

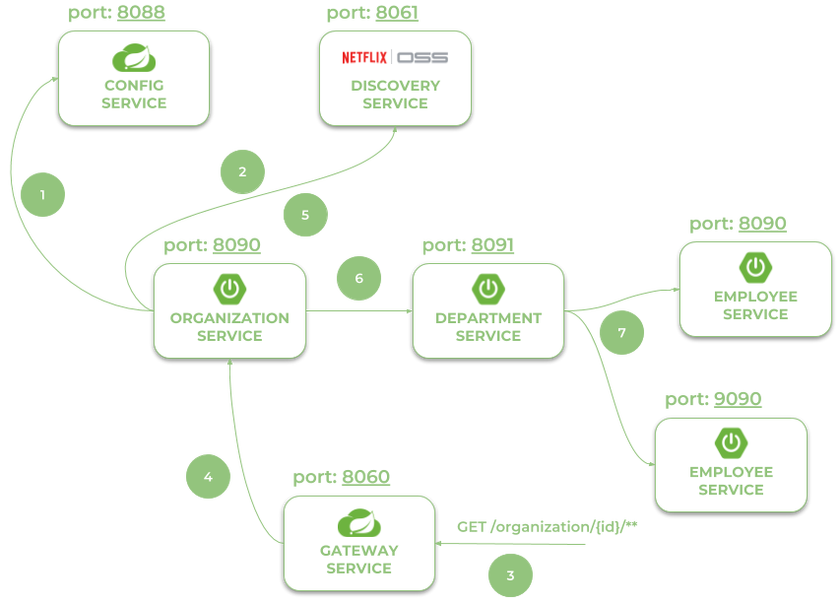
}

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

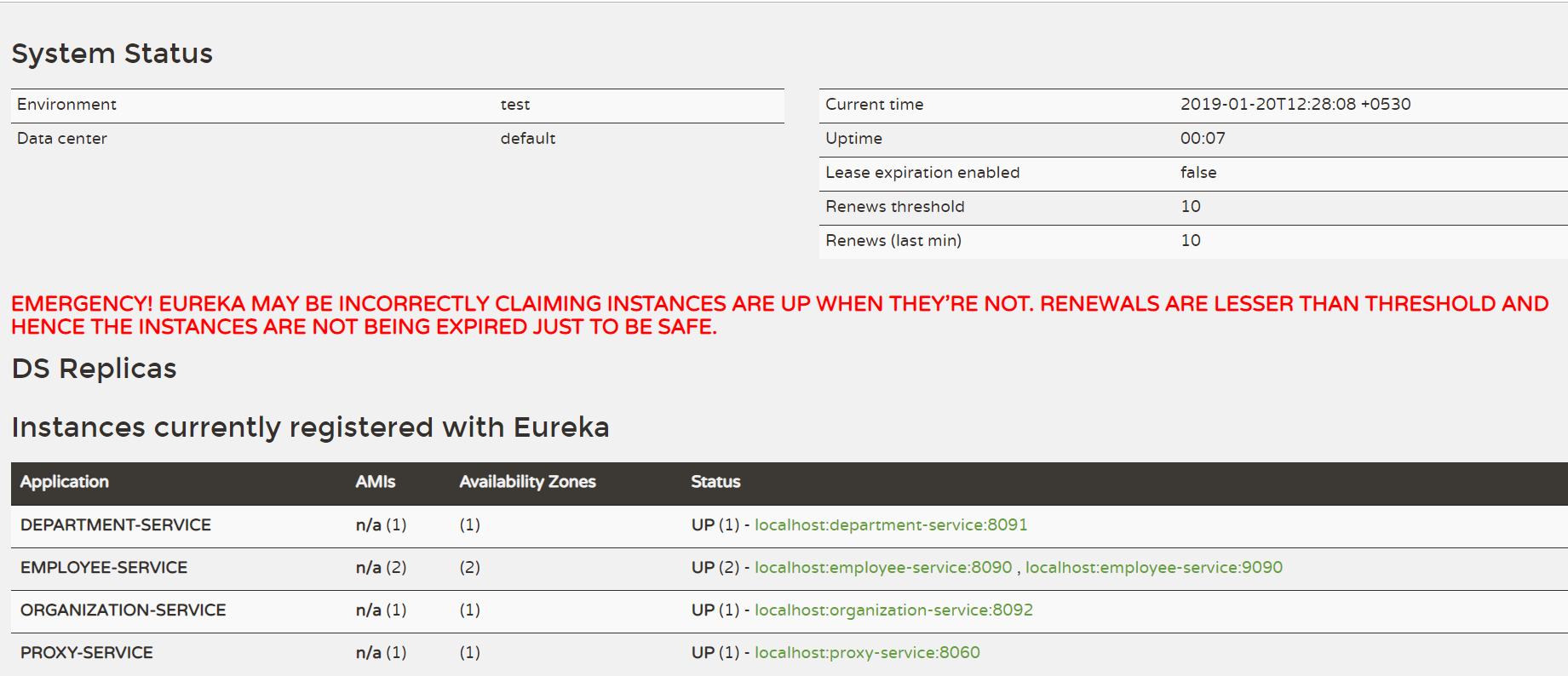


## Step 7. Running Applications

Let's take a look at 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 the address localhost:8088 **(1)**. Based on remote configuration settings, it is able to register itself in Eureka **(2)**. When the endpoint of organization-service is invoked by an external client via the gateway **(3)** available under address localhost:8060, the request is forwarded to an instance of organization-servicebased on entries from service discovery **(4)**. Then organization-service lookup for address of department-service in Eureka **(5)**, and calls its endpoint **(6)**. Finally, department-service calls the endpoint from employee-service. The request is load balanced between two available instances of employee-service by **Ribbon (7)**.



Let's take a look at the Eureka Dashboard available under [*http://localhost:8061*](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.



<http://localhost:8060/organization/1/with-departments-and-employees>

{"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,"position":"Analyst"},{"id":5,"name":"Patrick Dempsey","age":27,"position":"Developer"}]}],"employees":[]}

