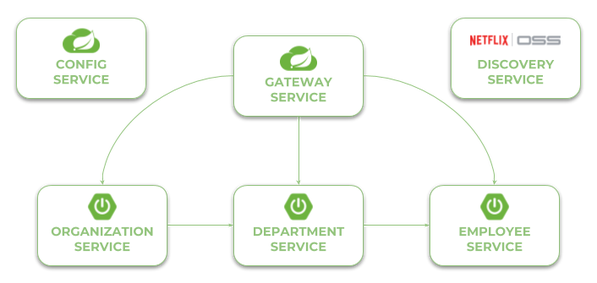
**Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud**

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.

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.SR1</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.

**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-configdependency 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(args);

    }

}

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](http://localhost:8088/)

As I have already mentioned, the configuration file discovery-service.ymlshould 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://](NULL)${eureka.instance.hostname}:${server.port}/eureka/

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-swagger2dependency 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</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 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());

    }

    ...

}

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](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:

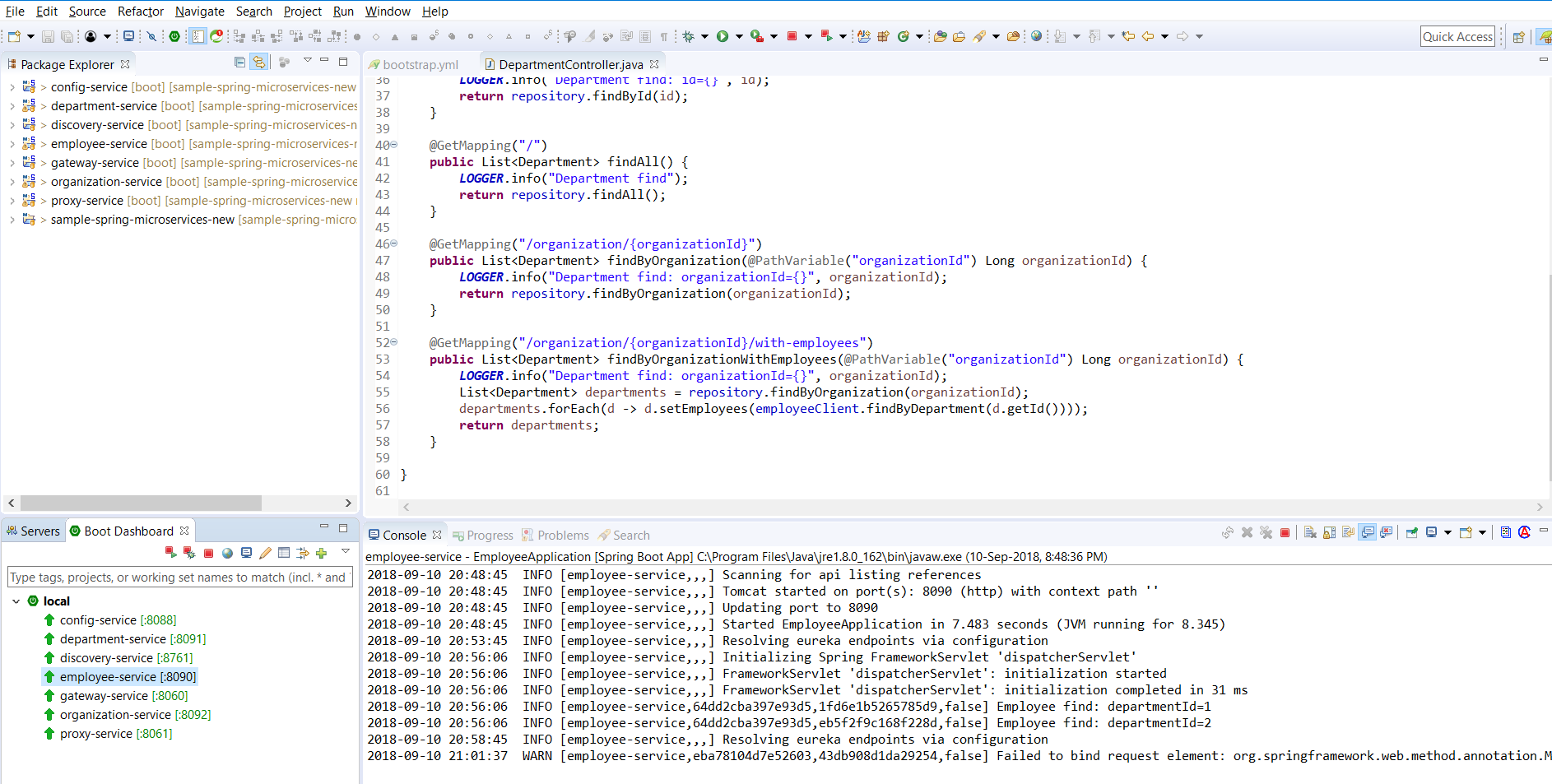
  port: 9090

eureka:

  client:

    serviceUrl:

      defaultZone: <http://localhost:8061/eureka/>



<http://localhost:8761/>