Eureka implementation example details.

To be informed about the presence of a client, they must send a heartbeat signal to the registry.

To achieve the goal of this, we'll implement three microservices:

* a service registry (Eureka Server)
* a REST service, which registers itself at the registry (Eureka Client, currency-exchange-service)
* a web application, which is consuming the REST service as a registry-aware client (Spring Cloud Netflix Feign Client, currency-conversion-service)

Implementing a *Eureka Server* for service registry is as easy as:

1. adding [spring-cloud-starter-netflix-eureka-server](https://search.maven.org/search?q=spring-cloud-starter-netflix-eureka-server) to the dependencies
2. enabling the Eureka Server in a [*@SpringBootApplication*](https://www.baeldung.com/spring-boot-application-configuration) by annotating it with *@EnableEurekaServer*
3. configuring some properties

Implementing a *Eureka Client:*

For a @SpringBootApplication to be discovery-aware, we have to include a Spring Discovery Client (for example, [spring-cloud-starter-netflix-eureka-client](https://search.maven.org/search?q=spring-cloud-starter-netflix-eureka-client)) into our classpath.

Then we need to annotate a @Configuration with either @EnableDiscoveryClient or @EnableEurekaClient. Note that this annotation is optional if we have the spring-cloud-starter-netflix-eureka-client dependency on the classpath.

The latter tells Spring Boot to use Spring Netflix Eureka for service discovery explicitly. To fill our client application with some sample-life, we'll also include the [spring-boot-starter-web](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-web%22) package in the pom.xml and implement a REST controller.

Now we'll run the client and point our browser to [*http://localhost:8761*](https://localhost:8761/) again to see its registration status on the Eureka Dashboard. By using the Dashboard, we can do further configuration, like link the homepage of a registered client with the Dashboard for administrative purposes. The configuration options, however, are beyond the scope of this article:

Implement *Feign Client:*

We'll now implement a *REST*-consuming web application using *Spring Netflix Feign Client*.

**Think of***Feign***as a discovery-aware**[*Spring* *RestTemplate*](https://www.baeldung.com/rest-template)**using interfaces to communicate with endpoints. These interfaces will be automatically implemented at runtime, and instead of***service-urls***, it's using***service-names***.**

Without *Feign,* we would have to autowire an instance of *EurekaClient* into our controller with which we could receive service-information by *service-name* as an *Application* object.

The *Feign Client* is located in the [spring-cloud-starter-feign](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-feign%22) package. To enable it, we have to annotate a *@Configuration* with *@EnableFeignClients*. To use it, we simply annotate an interface with *@FeignClient(“service-name”)* and auto-wire it into a controller.

A good method for creating such *Feign* *Clients* is to create interfaces with [*@RequestMapping*](https://www.baeldung.com/spring-requestmapping) annotated methods and put them into a separate module. This way they can be shared between server and client. On the server-side, we can implement them as *@Controller*, and on the client-side, they can be extended and annotated as *@FeignClient*.

Furthermore, the [*spring-cloud-starter-eureka package*](https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-starter-eureka%22) needs to be included in the project and enabled by annotating the main application class with *@EnableEurekaClient*.

While running Eureka servers, we often run into exceptions like:

com.netflix.discovery.shared.transport.TransportException: Cannot execute request on any known server

Basically, this happens due to the wrong configuration in application.properties or application.yml. Eureka provides two properties for the client that can be configurable:

* registerWithEureka: If we set this property as true, then while the server starts, the inbuilt client will try to register itself with the Eureka server.
* fetchRegistry: If we configure this property as true, the inbuilt client will try to fetch the Eureka registry.

Now when we start up the Eureka server, we don't want to register the inbuilt client to configure itself with the server.

If we mark the above properties as true (or just don't configure them, as they're true by default), then while starting the server, the inbuilt client tries to register itself with the Eureka server and also tries to fetch the registry, which isn't available yet. As a result, we get TransportException.

So we should never configure these properties as true in the Eureka server applications.

We recommend you to watch all the properties files of each one application of the example and also classes to get a better understanding.

The order to execute the programs is:

1. naming-server
2. currency-conversion-service
3. currency-exchange-service