**Eureka for registration and discovery**

So far, we have achieved externalizing configuration parameters as well as load balancing across many service instances. Ribbon-based load balancing is sufficient for most of the MicroServices requirements. However, this approach falls short in a couple of scenarios:

* If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.
* When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.
* In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually update the Config server. Though the configuration changes will be automatically propagated to all required instances, the manual configuration changes will not work with large scale deployments. When managing large deployments, automation, wherever possible, is paramount.

To fix this gap, the MicroServices should self-manage their life cycle by dynamically registering service availability, and provision automated discovery for consumers.

**What is Dynamic Service Registration?**

Dynamic Registration means, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.

**What is Dynamic Discovery**

Dynamic discovery is applicable from the service consumer's i.e.client point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services, and then invoke the services accordingly. In this approach, **instead of statically configuring the service URLs, the URLs are picked up from the service registry**. Here the clients may keep a local cache of the registry data for faster access. In this approach, the state changes in the registry server will be propagated to the clients connected to it to avoid using stale data.

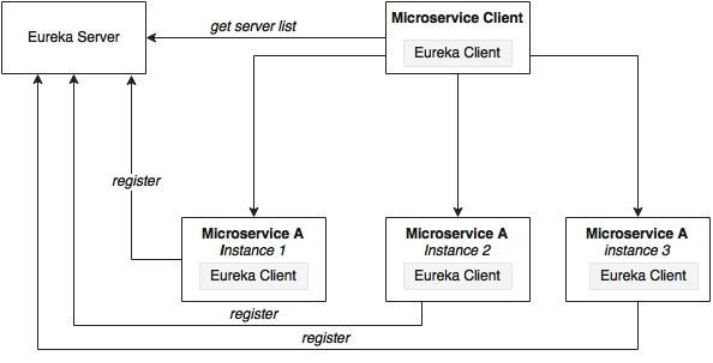
There are a number of options available for dynamic service registration and discovery. **Netflix Eureka, ZooKeeper, and Consul** are available as part of Spring Cloud, as shown in the http://start.spring.io/ screenshot below screenshot. **Etcd** is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery. we will focus on the Netflix Eureka implementation:



**What is Eureka?**

Spring Cloud Eureka comes from Netflix OSS.Eureka is primarily used for self-registration, dynamic discovery, and load balancing. **Eureka uses Ribbon for load balancing internally.**

Lets consider the following diagram:



As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability.**The microservices use the Eureka client for registering their availability**. The consuming components will also use the Eureka client for discovering the service instances.

**How Registry will work?**

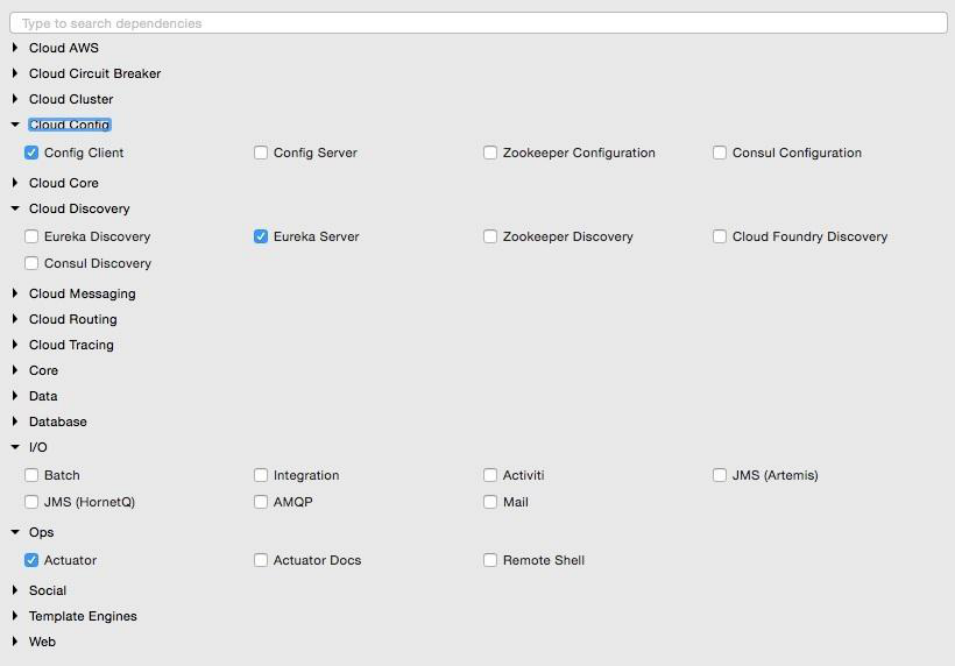
When a microservice is bootstrapped, it reaches out to the Eureka server, and advertises its existence with the binding information. Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease. If a service endpoint cannot renew its lease in a few attempts, that service endpoint will be taken out of the service registry.

The registry information will be replicated to all Eureka clients so that the clients have to go to the remote Eureka server for each and every request. Eureka clients fetch the registry information from the server, and cache it locally. After that, the clients use that information to find other services. This information is updated periodically (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one.

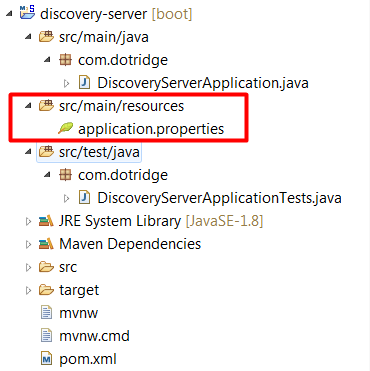
When a client wants to contact a microservice endpoint, the Eureka client provides a list of currently available services based on the requested service ID. The Eureka server is zone aware. Zone information can also be supplied when registering a service. When a client requests for a services instance, the Eureka service tries to find the service running in the same zone. The Ribbon client then load balances across these available service instances supplied by the Eureka client. The communication between the Eureka client and the server is done using REST and JSON.

**Setting up the Eureka server**

**Step-1:** Start a new Spring Starter project discovery-service, and select Config Client, Eureka Server, and Actuator:

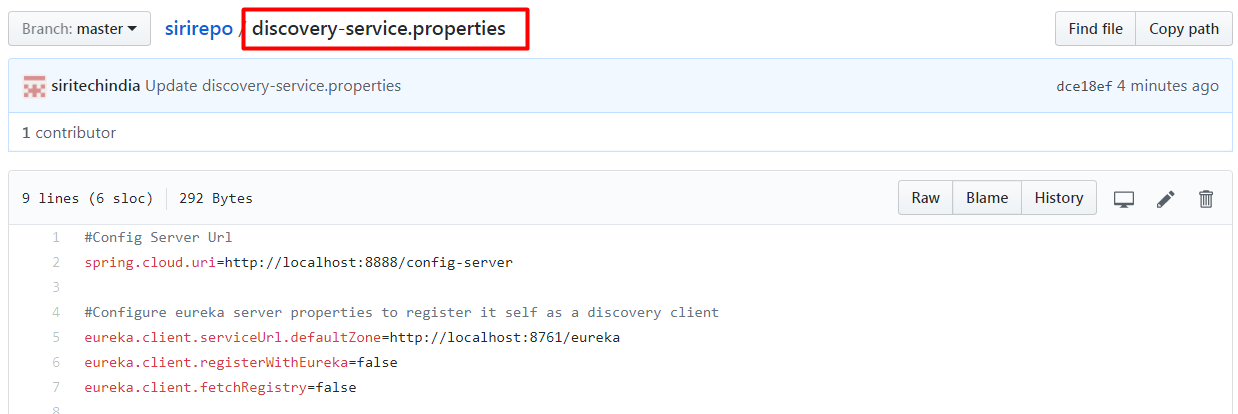


The project structure of the Eureka server i.e,discovery-server is shown in the following image:



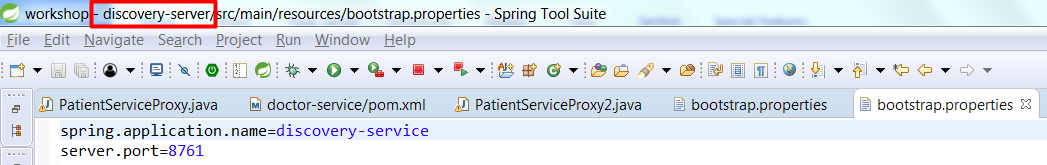
Note that the main application is named DiscoveryServerApplication.java

**Step-2:** Create a <spring.application.name>.properties I.e. discovery-server.properties file and commit to the git repository as discovery-service uses config-server to fetch the configuration properties. This file should have the config server url and eureka server properties as shown below:



The Eureka server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing state from the other Eureka servers. The Eureka client is taken to its peers by configuring the **eureka.client.serviceUrl.defaultZone** property.In the standalone mode, we point **eureka.client.serviceUrl.defaultZone** back to the same standalone instance. Even it is not necessary to configure the above eureka properties when it is running in standalone mode.

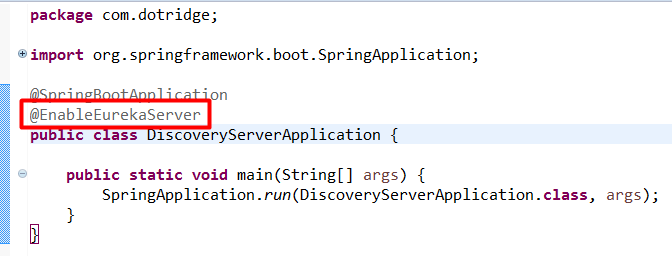
**Step-3:** Rename application.properties to bootstrap.properties since this is using the Config server. It should have application name and port as shown below:



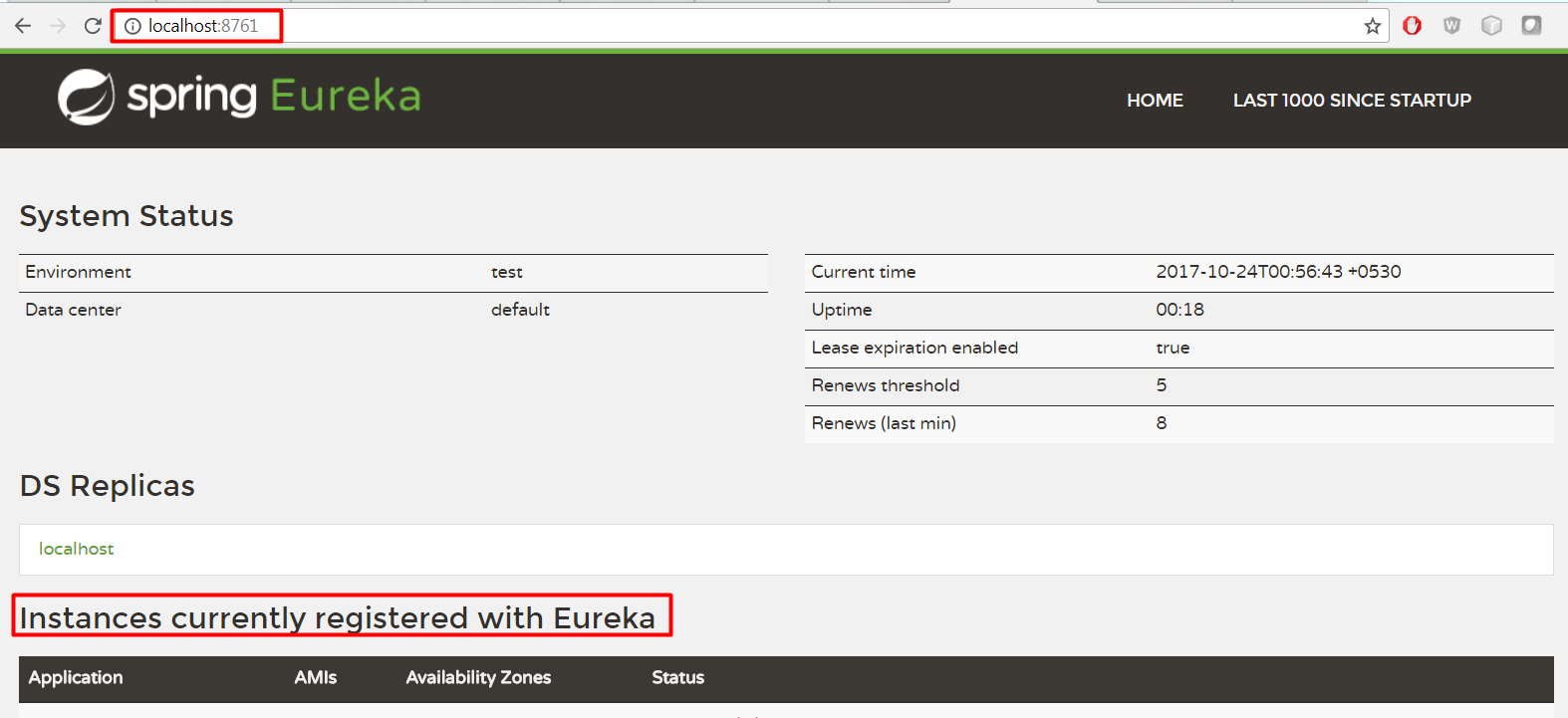
**Note:**

Renaming application.properties file to bootstrap.properties file is not necessary. But when we go for high availability of config-server and discovery-server we might need this.

**Step-4:** In DiscoveryServerApplication, add @**EnableEurekaServer** annotation as shown below



**Step-5:** We are now ready to start the Eureka server. Ensure that the Config server is also started. Right-click on the application and then choose Run As | Spring Boot App. Once the application is started, open http://localhost:8761 in a browser to see the Eureka console.

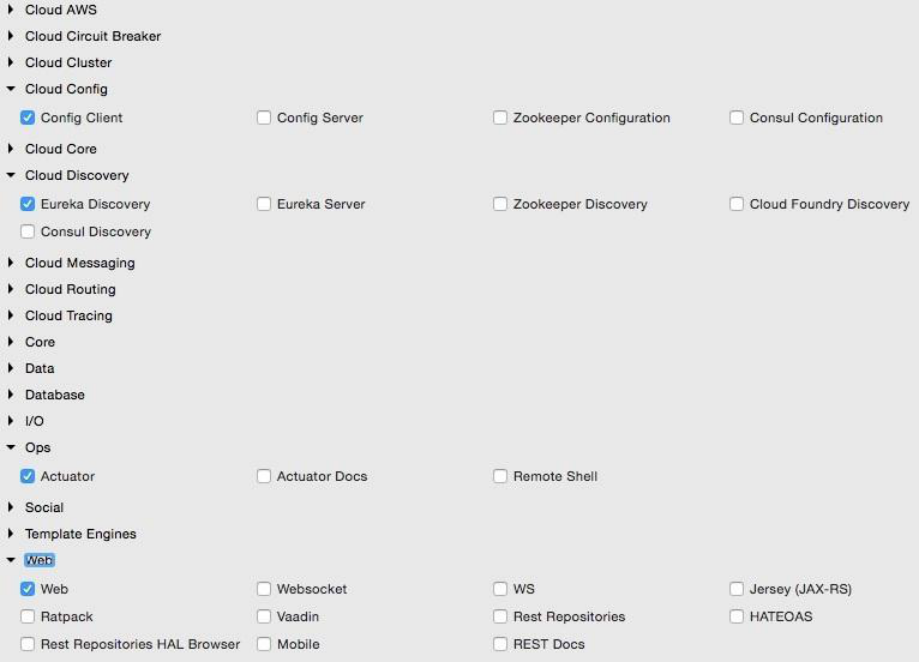


In the console, note that there is no instance registered under Instances currently registered with Eureka. Since no services have been started with the Eureka client enabled, the list is empty at this point.

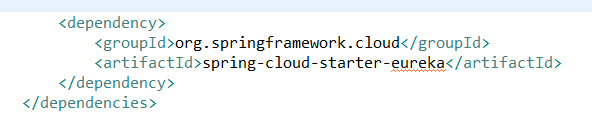
**Dynamic Discovery**

Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service.

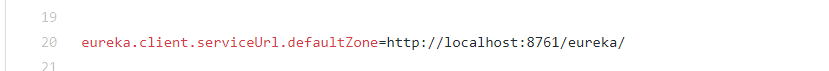
**Step-1:** add the Eureka dependencies to the pom.xml file. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web** as well as **Eureka discovery** client as follows:



But as we are modifying our existing MicroServices, add the following additional dependency to all MicroServices in their pom.xml files:

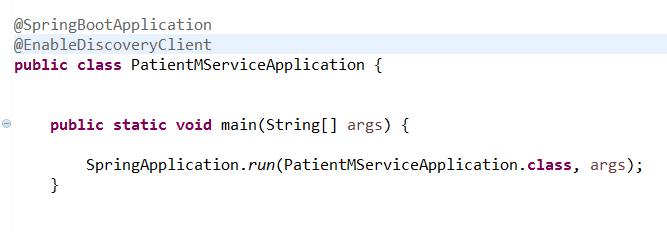


**Step-2:** The following property has to be added to all microservices in their respective configuration files in config-server i.e.git repo. This will help the microservices to connect to the Eureka server. Commit to Git once updates are completed:



In our case add in doctor-service.properties, patient-service.properties files of config-server I.e.available in git repo.

**Step-3:** Add @EnableDiscoveryClient to all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start up to advertise their availability. Start all servers except doctor-service. Since we are using the Ribbon client on the doctor-service micro service, the behavior could be different when we add the Eureka client in the class path. We will fix this soon.



**Step-4:** Going to the Eureka URL (http://localhost:8761), we can see that patient-service is up and running:

