When Cloud-Config is used, each microservice gets two properties files with the name 'application.properties'. One in GIT and the other one present locally. Which one will take precedence?

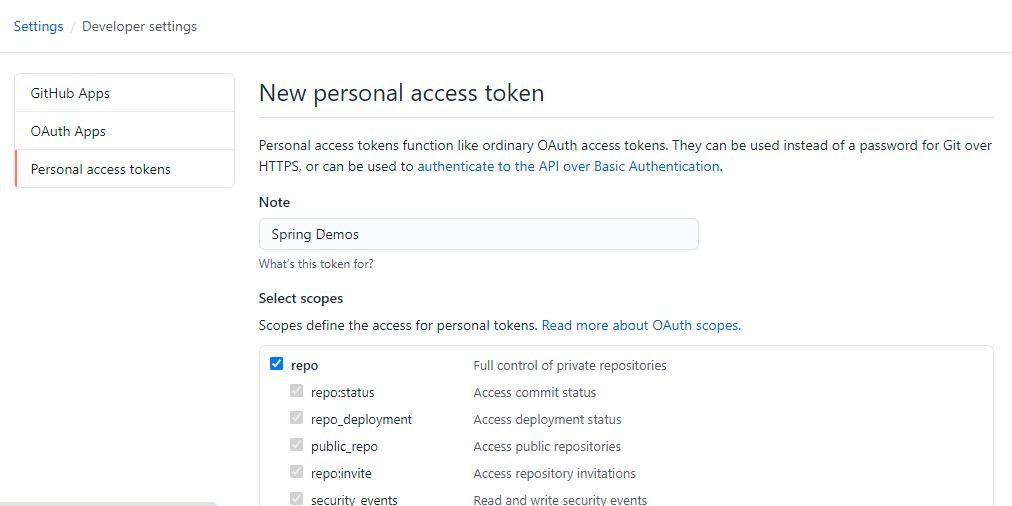
To ensure that the properties file from GIT takes higher priority, it needs to be accessed first before the local properties file for which the bootstrap context is used.

Spring Cloud creates a parent context to the spring application context called the ‘bootstrap’ context. This context takes precedence over the application context. This context is responsible for loading configuration details from an external source. Both these contexts share the Spring Environment, thus making configuration usage seamless.

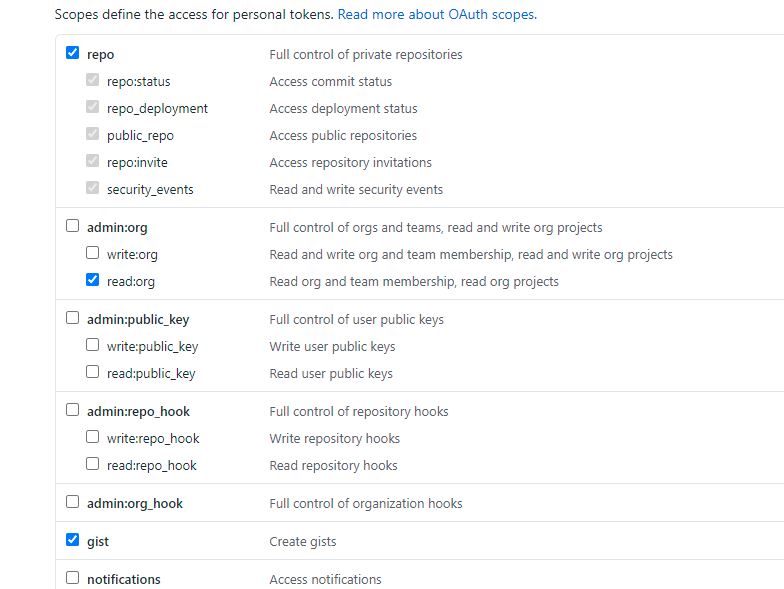
Since the bootstrap context takes precedence the URI of the config-server must be mentioned in bootstrap.properties/YAML for the clients.

Note:

* In the recent releases of Spring Boot versions >2.4.x, the bootstrap context initialization using property sources like bootstrap.properties/bootstrap.yml is deprecated.
* As the demos are using Spring Boot 2.5.3 version we can add these configuration data to the application.properties file itself. No need for additional bootstrap.properties.
* Also, the property to be added in the microservices is modified in recent releases as spring.config.import=optional:configserver:http://localhost:1111. Refer to the downloaded demos for the complete code.
* To connect to the Infygithub/Github instead of the password we can also use the Personal Access Token. The access token can be generated from -> Settings -> Developer Settings -> Personal Access Tokens



Select the scope configurations based on the requirement and generate the token:



Spring uses Environment to get the configuration details from various sources such as the environment variables, properties files, YAML files, etc. These are built in property sources. When we use cloud config, it adds an additional property source to the Environment such that it takes the properties from the cloud config server.

When cloud config is used, the additional property source takes a higher priority. It means that, any duplicate properties in other property sources are ignored.

We can access the configuration files by using endpoint on the config server in any one of the below patterns:

1. http:*//<config\_server\_host>:<port>:/<application>-<profile>.yml*
2. http:*//<config\_server\_host>:<port>/<application>-<profile>.properties*

The properties file used in the GIT must have the same name as that of the client's spring.application.name

We can have multiple profiles as well and if a profile is not mentioned it will load the default profile, which is the same name as that of the spring.application.name.

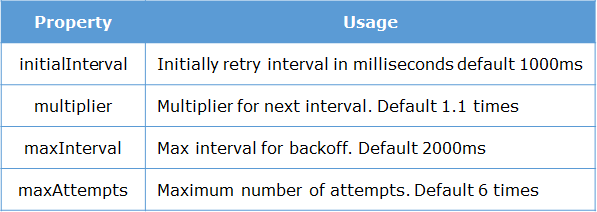
If we try to access a microservice property for a given profile, the order of files used will be:

1. Yaml file for that profile
2. Properties file for that profile
3. application.yml file
4. application.properties file

For example, if we have a property called x=10 in application.yml and x=20 in the application.properties, the final value used will be 10

The config-server is contacted by the clients only once, during the start of the project. Therefore any changes made to the configuration after the application starts will not be reflected in the application.

We can also configure the clients retry attempts to contact the config server using properties like



If a port is not specified for the config server, it runs in its default port 8888. Also, if the Cloud-Config server is down, then the client will throw an error not during startup, but while trying to access the property at runtime. To avoid this we can have the failFast property set to true. By this, the client will fail at startup time rather than at the run time.

1. spring.cloud.config.failFast=true

Also, in order to avoid Config-server to be a single point of failure, we usually deploy multiple instances of it to ensure high availability. If the cloud config server is unavailable, it will use the properties files in the individual applications as a fallback

When we make any changes to the properties file in GIT, the config-clients automatically do not update themselves with the modified values. **This is because the configurations are taken only once at the time of startup.** For example, if we modify the property of CustomerMS, we have to restart CustomerMS so that it can again fetch the properties from the config server. However, this is not a practical approach.

When we make any changes to the properties file in GIT, the config-clients automatically do not update themselves with the modified values. **This is because the configurations are taken only once at the time of startup.** To overcome this we need to :

* Add @RefreshScope annotation on the bean which is using the property
* Add spring-boot-actuator endpoint dependency

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

* In order to access the actuator endpoints using HTTP, we need to both enable and expose them by adding the below property in the relevant microservices:
  1. management.endpoints.web.exposure.include=\*

* Send a **POST** request to the /refresh endpoint of the service. This refreshes the microservice without restarting/redeploying it.

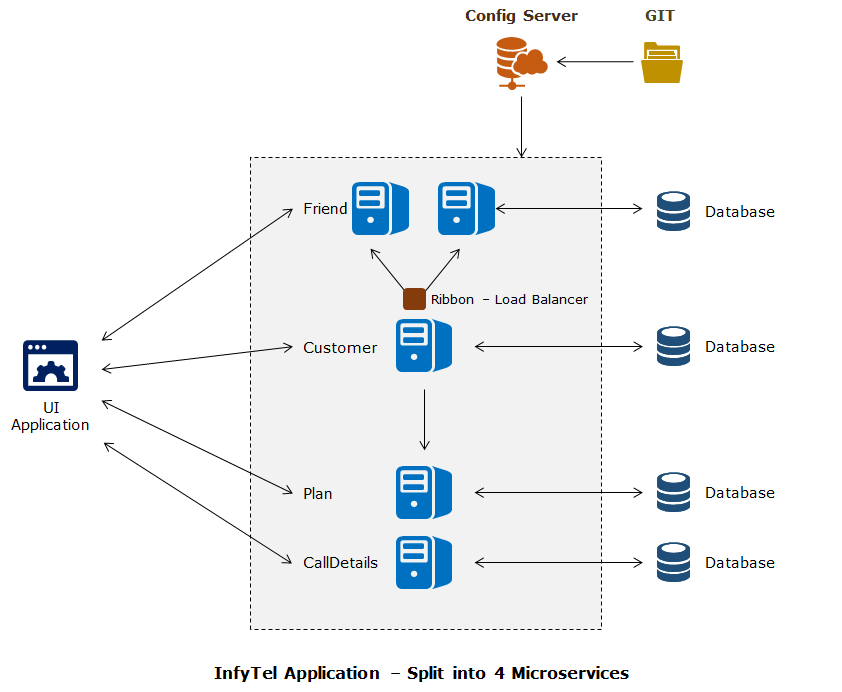
In this case, we would add @RefreshScope on the CustomerController, add actuator dependency on the CustomerMS and send a POST request to **http://localhost:8001/actuator/refresh** which is where the CustomerMS is running

The @RefreshScope annotation is needed only if we are using @Value. If we are getting the values from the spring environment variable directly, we don’t need this annotation. Every time we use getProperty() on the spring environment variable, it fetches the latest value. However, we would still need to have the /refresh actuator endpoint and send a POST request to it.

The problem with using the /refresh endpoint is that we have to manually fire a POST request to it. It is not automatically done when the configuration changes. Also, we need to fire a request to /refresh on all the services which might get affected by the change in the property. That means we have to keep track of which property is used in which application. If we have 100 microservices using the property, we need to fire /refresh on all those microservices.

The solution is to use Spring-Cloud-Bus. This, along with a Queuing service like RabbitMQ, will trigger refresh events on all dependent microservices. However, Spring-Cloud-Bus is beyond the scope of the curriculum.

* Spring Cloud project allows us to write cloud compatible code
* One of its sub-project is Spring Cloud Config
* Cloud-Config allows keeping all configurations in a central place
* Since this uses the Bootstrap context, we need to use bootstrap.properties file
* A config server contacts the GIT repo.
* The config clients contact the config server at startup to gather the configurations
* Changes made to configurations don’t automatically reflect in the clients
* The configuration files use a specific order of priority.
* The below illustration shows our application using Ribbon.


* 

1. Add the dependency in the infytel-customer
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
5. </dependency>
6. Create a Configuration class with the below bean
7. @Bean @LoadBalanced
8. public RestTemplate restTemplate() {
9. return new RestTemplate();
10. }
11. Autowire the RestTemplate as @Autowired RestTemplate template; in the CustomerController
12. Add @RibbonClient(name="custribbon") annotation on the CustomerController class
13. Add the below properties in the properties file:
14. custribbon.ribbon.eureka.enabled=false
15. custribbon.ribbon.listOfServers=http:*//localhost:8001,http://localhost:8002*
16. Autowire the rest template and update the infytel-friend-family invocation in CustomerController as
17. List<Long> friends = template.getForObject("http://custribbon/customers/" + phoneNo+"/friends", List.class);
18. Run the application, with the two instances of FriendFamilyMS running in two different ports

Notes:

 If you want to start an instance of Friend FamilyMS on a different port use the following command after the maven build of your project is done.

1. java -jar -Dserver.port=9797 target/FriendFamilyMS-0.0.1-SNAPSHOT-exec.jar *// java -jar -Dserver.port=<<port no>> target/<<jar name>>-exec.jar*

By default, the ribbon uses the **NoOpPing**strategy for checking if the services are up. However, NoOpPing is a dummy strategy. It assumes that all services are up. Thus it will keep pinging the services even if they are down. We can configure the Ping strategy so that we stop sending requests to services that are down.

Also, Ribbon by default uses the Round Robin load balancing strategy.

These things can be modified by adding a configuration file.

Note: In recent releases by default, the IPing strategy is used. But in a similar way, we can configure any strategy that is needed.

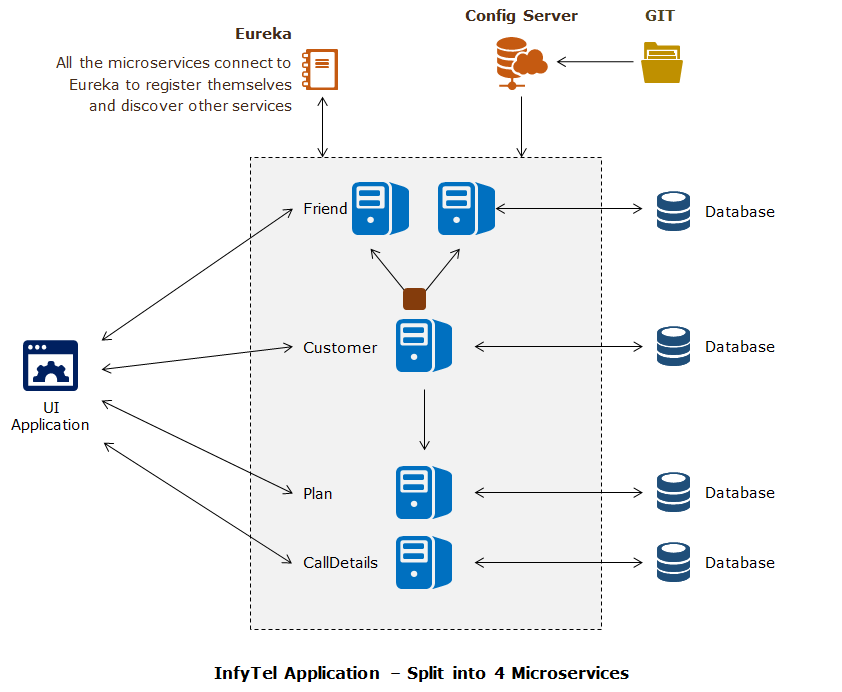
Ribbon can be configured as:

1. <clientName>.<nameSpace>.<propertyName>=<value>

For example:

1. custribbon.default.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule

In the third stage of the application, we will register our microservices with a Eureka Service Discovery server. The details of the Eureka are also stored in the GIT repo which can be accessed using the ConfigServer



1. Create a Spring Starter project with the name infytel-eureka
2. Add the below dependencies:
   1. <dependencyManagement>
   2. <dependencies>
   3. <dependency>
   4. <groupId>org.springframework.cloud</groupId>
   5. <artifactId>spring-cloud-dependencies</artifactId>
   6. <version>Greenwich.RELEASE</version>
   7. <type>pom</type>
   8. <scope>import</scope>
   9. </dependency>
   10. </dependencies>
   11. </dependencyManagement>
   13. <dependency>
   14. <groupId>org.springframework.cloud</groupId>
   15. <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
   16. </dependency>
3. Add the below properties in the application.properties file of infytel-eureka
   1. spring.application.name=Eureka1
   2. server.port=5555
   3. eureka.client.fetch-registry=false
   4. eureka.client.register-with-eureka=false
   5. eureka.client.service-url.defaultZone=http:*//localhost:5555/eureka*
4. Add @EnableEurekaServer annotation in the application file of infytel-eureka
5. Add the below dependencies in the microservices:
   1. <dependency>
   2. <groupId>org.springframework.cloud</groupId>
   3. <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
   4. </dependency>
6. Add @EnableDiscoveryClient in all microservices application file
7. Add the below property in the application.properties file in git
   1. eureka.client.service-url.defaultZone=http:*//localhost:5555/eureka*
8. Autowire Discovery client in CustomerController class as @Autowired DiscoveryClient client;
9. Remove the String friendUri; from CustomerController and update the code in accessing the friend-family-service as:
   1. List<ServiceInstance> instances=client.getInstances("FriendFamilyMS");
   2. ServiceInstance instance=instances.get(0);
   3. URI friendUri = instance.getUri();

Note: The service discovery happens through the spring.application.name value of the services. Hence they should not change.

1. Run the application

@EnableDiscoveryClient annotation makes an application a Eureka instance as well as a Eureka Client. Every application registered with Eureka is a Eureka instance. Since every Eureka instance can also get details about other registered Eureka instances, it also becomes a client.

register-with-eureka property when set to true ( which is by default ), will register an application with the Eureka Sever. Such an application is also called a Eureka Instance. The Eureka Instance will start sending heartbeats to the Eureka Server. If the Eureka server does not receive heartbeats from an Instance within a configurable time limit ( every 30 secs by default), it considers the Instance to be down and deregisters it from the registry.

The fetch-registry property will fetch the registry from the Eureka Sever once at startup time and will cache it. It will check the Eureka Server at regular intervals ( by default at every 30 secs) to see if there are any changes. If there are changes, it fetches only the updates and the unchanged parts will be continued to be accessed from the cache.

A Eureka server is also a client. Because it can register itself with other Eureka servers and form a cluster.

Every client has to register itself with Eureka and it will also try to fetch the registry details. If we want to run only a single Eureka Server instance, then these two properties( fetch-registry, register-with-eureka) should be false. Otherwise, it is trying to register itself with itself.

Every registered client gets access to what is known as a Discovery client. The Discovery client is actually a service endpoint, which returns an enum of all ServiceInstance instances of the clients registered with the Service Registry.

You can take a look at all the instances registered with the service registry using this endpoint as follows:

**http://{eureka-host}:{eureka-port}/{eureka}/apps/{spring-application-name}**

* In a cloud environment, we cannot predict the host and port of different microservices.
* By using the service discovery pattern, we can dynamically find the services registered
* Netflix Eureka allows us to create a Service Registry
* Clients can register themselves and discover other instances
* Ribbon is typically used along with Eureka. Earlier we had seen how we can use Ribbon with a static list of servers. Instead of using a static list, we can get a dynamic list of servers by using it with Eureka.
* When used with Eureka, not only will Ribbon get the server list, but it also will depend on Eureka to know if a service is up or not.

1. Remove Autowire of DiscoverClient and Autowire load balanced RestTemplate
2. Remove code for getting URI from discovery client
3. Access the PlanMS and FriendMS through the rest template object and use the names of the service instead of the URI. Since Eureka is used, the URI will be picked automatically based on the service name.
4. PlanDTO planDTO=template.getForObject("http://PLANMS"+"/plans/"+custDTO.getCurrentPlan().getPlanId(), PlanDTO.class);
5. List<Long> friends=template.getForObject("http://FRIENDFAMILYMS"+"/customers/"+phoneNo+"/friends", List.class);

Eureka is rarely run as a single instance, as it would become a single point of failure. Typically we run multiple instances of Eureka forming a cluster. In a cluster, each Eureka server replicates the information in the other servers.

1. Open the hosts file in C:\Windows\System32\drivers\etc
2. Add the below hostnames:
3. 127.0.0.1 Eur1
4. 127.0.0.1 Eur2
5. 127.0.0.1 Eur3
6. Use the below yml file in the infytel-eureka server
7. spring:
8. profiles: Eureka1
9. application:
10. name: Eureka
11. server:
12. port: 2222
13. eureka:
14. instance:
15. hostname: Eur1
16. client:
17. registerWithEureka: true
18. fetchRegistry: true
19. serviceUrl:
20. defaultZone: http:*//Eur2:2223/eureka/,http://Eur3:2224/eureka/*
21. ---
22. spring:
23. profiles: Eureka2
24. application:
25. name: Eureka
26. server:
27. port: 2223
28. eureka:
29. instance:
30. hostname: Eur2
31. client:
32. registerWithEureka: true
33. fetchRegistry: true
34. serviceUrl:
35. defaultZone: http:*//Eur1:2222/eureka/,http://Eur3:2224/eureka/*
36. ---
37. spring:
38. profiles: Eureka3
39. application:
40. name: Eureka
41. server:
42. port: 2224
43. eureka:
44. instance:
45. hostname: Eur3
46. client:
47. registerWithEureka: true
48. fetchRegistry: true
49. serviceUrl:
50. defaultZone: http:*//Eur1:2222/eureka/,http://Eur2:2223/eureka/*
51. Comma separated values in the defaultZone indicate peer awareness. Eureka1, Eureka2 ,and Eureka3 are peers of each other and hence will replicate the details across each other.
52. Update the application.properties file in GIT for the below property:
    1. eureka.client.service-url.defaultZone=http:*//Eur1:2222/eureka,http://Eur2:2223/eureka,http://Eur3:2224/eureka*
53. Run all the three profiles of the Eureka server and restart all the microservices
54. You will get three dashboards in three different Eureka ports. Since we have a cluster, each dashboard will have the same details of microservices as the other two Eureka Servers in the cluster.
55. Bring down a microservice. You will find that since each Eureka server in a cluster replicates itself, all Eureka servers in the cluster will now have the same updated information.