**Microservices:**

**Monolithic Architecture**

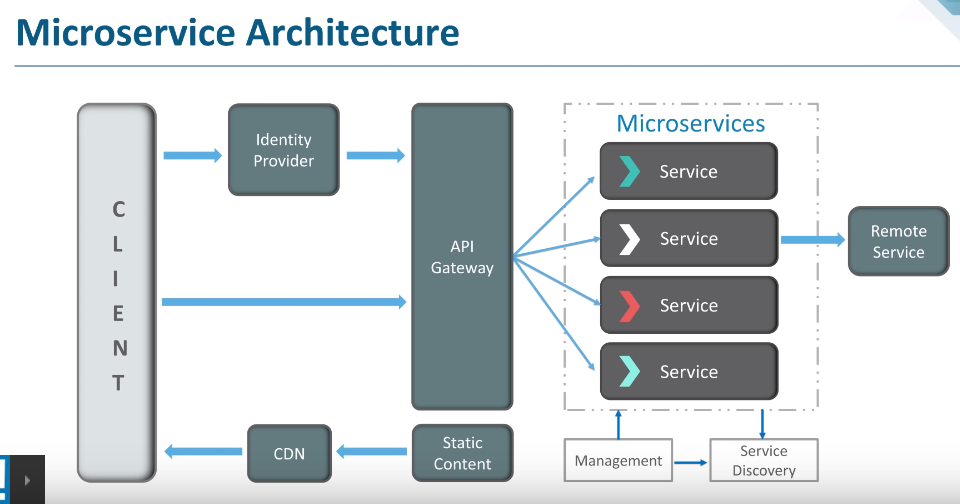
Monolithic Architecture is like a big container wherein all software components of an application are assembled together and tightly packed.

Disadvantages:

* Large & complex Applications
* Slow Development – Because we have to bring the whole code base. Development and deployment becomes time consuming task as the code base grow.
* Blocks continuous Development
* Unscalable – You can’t scale up a single service as such. You got to scale the whole application.
* Unreliable- If one of the service fails entire app fails
* Inflexible – Not flexible when you want to rewrite the whole application.

**Microservices Architecture**

This is an architectural style that structures an application as a collection of small autonomous services modelled around a business domain. Each service is self-contained and implements a single business capability



Identity Provider – Authenticates the request

API Gateway – Entry point

Service Discovery – Tells mode of communication between Service and then the service communicates via HTTP Rest or Message Bus.

Services has to deploy their static content to a cloud based storage service that can deliver its content via Content Delivery Network

Management – Places the services on respective nodes. Resolves any failures

Features of Microservices Architecture:

1. Loosely coupled
2. Componentization
3. Business capabilities
4. Autonomy
5. Continuous Delivery
6. Responsibility
7. Decentralized Governance(since the services are working individually, they should be managed individually)
8. Agility(Build or discard features)
9. Language Neutral
10. Bounded context – There is no need for 1 microservice to understand other microservice.

Advantages of microservices:

1. Independent Development
2. Independent Deployment
3. Fault Isolation – one service goes down. Entire application doesn’t go down
4. Mixed Technology Stack
5. Granular scaling – scale up the service that is only needed not all the service or application as such

Characteristics of microservices:

1. Organized on Business Capabilities
2. Products not projects
3. Smart Endpoints & Dumb pipes – Smart communication endpoints should be there. So the app integrity is the best
4. Decentralized Governance
5. Decentralized Data management – each service will have its own db
6. Infrastructure Automation
7. Design for failure

Best Practices:

1. Separate datastore for each service
2. Keep code at a similar level of maturity
3. Separate build for each microservice
4. Deploy into containers
5. Treat servers as stateless

**Domain Driven Design(DDD) :**

To ease the development of complex applications. Connects the related piece of software into an ever evolving model. Focuses on core domain & domain logic. Find complex domain designs on models of domain, constantly collaborate with domain experts to improve the application model and resolve any domain related issues.

**Ubiquitous language :**

Language used by developers and users for common specification of the domain they are working on. Language that the domain can be explained easily. Brings all team members on the same page. Architecture that is built of this language is called DDD.

**Why there is a need for DDD:**

Mapping to Domain 🡪 Reduced complexity 🡪 Testability 🡪 Maintainability 🡪 Knowledge Rich Design 🡪 Brings business and service together 🡪 Context focused 🡪 Ubiquitous Language

**Cons of a microservice:**

* Increases troubleshooting challenges
* Increases delay due to remote calls
* Increased effort for configuration and other operations
* Difficult to maintain transaction safely
* Tough to track data across various service boundaries

**Difference between monolithic, SOA and Microservices**

Monolithic – Single Unit

SOA – Coarse grained, Modularity to smaller extent, More sharing of resources across modules, Middleware services for inter communication between modules

Microservices – Fine grained, complete modularity, No sharing, No middleware services for inter communication, they communicate through REST APIs

**Challenges:**

Automating the components.

Perceptibility

Configuration Management

Debugging

**Cohesion & Coupling**

Cohesion : Intra bond. Degree to which the elements inside a module belong together.

Coupling : Inter bond between modules

Good Design should have low coupling and high cohesion

**Spring Boot Actuator:**

Access current state of running application in production environment. Provides restful webservices endpoints which you use and check various metrics. This can monitor the app in production environment

**Problems solved by Spring Cloud**

* Complexity associated with Distributed System
* Service Discovery
* Redundancy
* Load Balanced
* Performance

**Difference between REST and Microservices:**

Microservice is an approach for software development. REST is a medium to build microservices

**Differences Types of test for Microservices:**

* Unit Testing
* Exploratory Testing
* Acceptance Testing
* Performance Testing

Distributed Transaction : Any situation where a single event results in the mutation of 2 or more separate sources of data which cannot be committed atomically

Idempotence : Endresult should be same no matter how many times the app executes. It is used at the remote service or data source so that, in cases where it receives the instruction more than once, it processes only once

Client certificates : Types of digital certificates that is used by client systems to a make authenticated request to a remote server

Types of credentials in 2 factor authentication system:

* Something you know’
* Something you have’
* Something you are

PACT :Open source tool to allow testing interactions between service providers & consumers in isolation against a contract. To test consumer driven contract between provider and consumer of a microservice.

Conways law : Organizations which design systems are constrained to produce designs following the organization’s communication structure

Contract Testing vs End to end testing

Container : You can encapsulate your microservice along with its dependencies in a container.Gives modularity

Consumer Driven Contract : In Consumer-Driven Contracts, each consumer captures their expectations of the provider in a separate contract. All of these contracts are shared with the provider so they gain insight into the obligations they must fulfill for each individual client.

Semantic Monitoring/Synthetic Monitoring : Running an automated test in live production on a regular basis and pushing the result into the monitoring service which triggers alert in case of failures

Stub or mock is a dummy object for any microservice test

Docker : Container which can be used to host any application. Software application and its dependencies are tightly packaged here. This packaged product is called a container. Since its done by docker, its called docker container. Main components are docker client, docker daemon or server, docker containers.

Canary Releasing : Relaese software update to specific set of people and once this version works. You will be releasing it to everyone.

Reactive Extensions : A design approach in which we collect results by calling multiple services and then compile a combined response. Also known as Rx

**Course Name : Spring Cloud Fundamentals by Dustin Schultz in pluralsight**

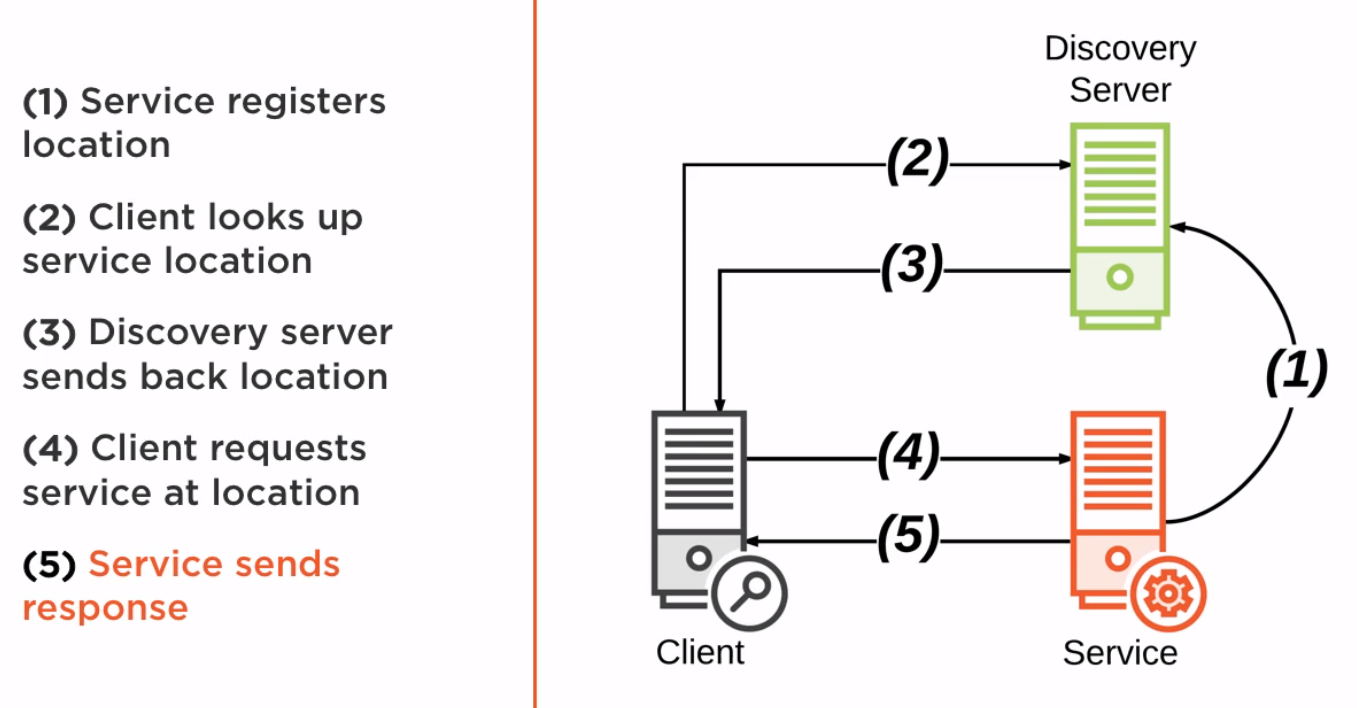
**1.Finding services using Service Discovery**

There will be service Discovery server. It will provide a way for a service to register itself, re-register itself, a way for a client to find other services, a way to check the health of the service and remove unhealthy services.

Services can be discovered by the following projects in Spring cloud:

* Spring Cloud Consul
* Spring Cloud Zookeeper
* Spring Cloud Netflix – Netflix Open source Project + Spring + Spring Boot

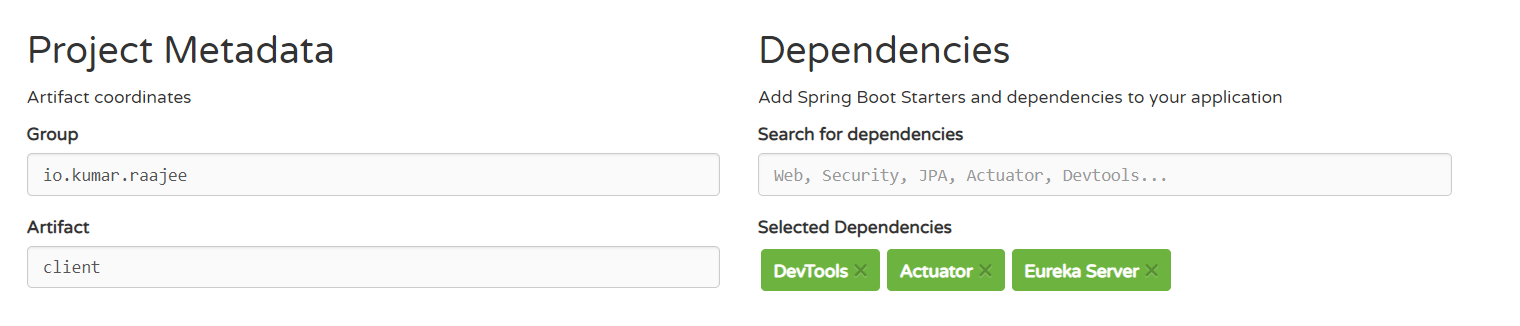




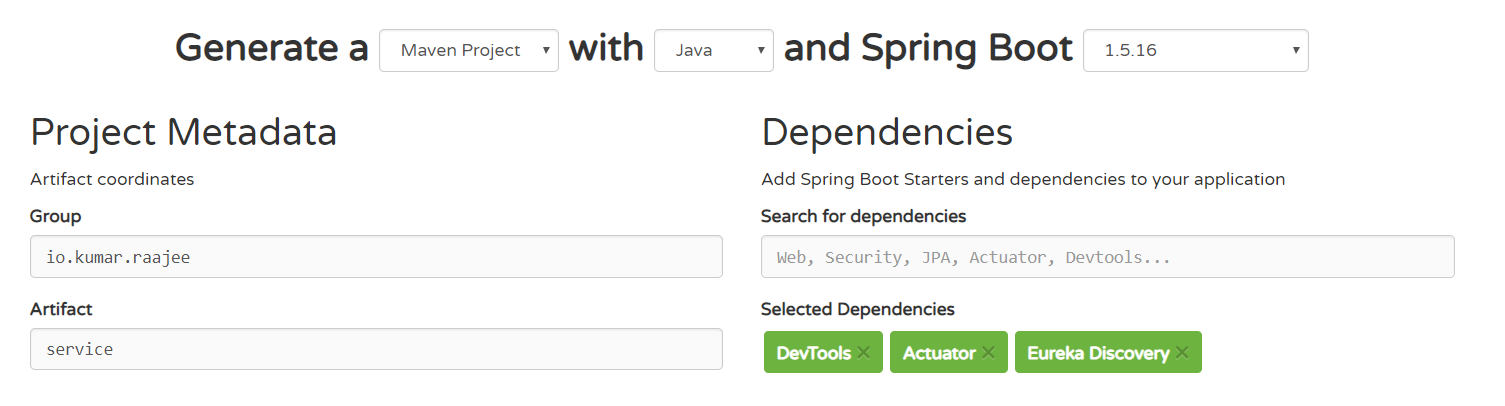
Demo : C:\Users\stkra\Documents\Spring\spring\_cloud\_workspace

Go to <https://start.spring.io/> and create projects for Service Discovery Server, Service and Client. Import all the 3 projects in STS through import -> Existing Maven Projects.

Discovery Server –

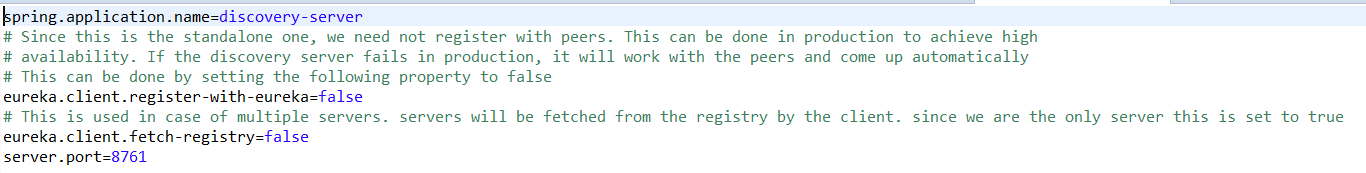


Service & Client



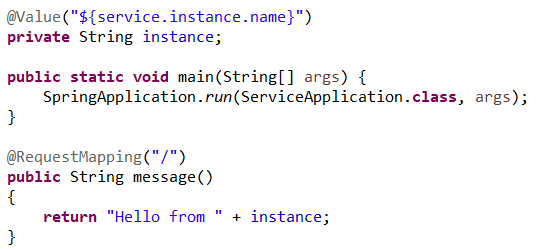
Discovery Server :

1. Open up the main application in Discovery Project and add the annotation @EnableEurekaServer
2. Open up the application .properties and add the following entries,

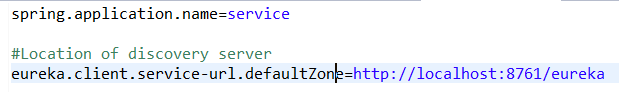


Service :

1. Open up the main application in Service Project and add the annotation @EnableDiscoveryClient[This enables the service to register itself. At the same, it can search for a service in the registry],@RestController and update the class as follows,



1. Open up the application .properties and add the following entries,

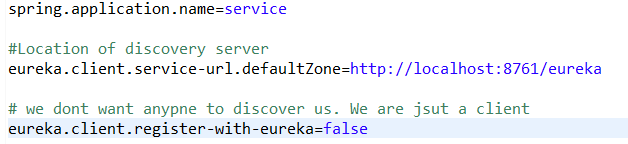


Client:

1. Open up the main application in Client Project and add the annotation @EnableDiscoveryClient,@RestController and update the class as follows,



1. Open up the application .properties and add the following entries,



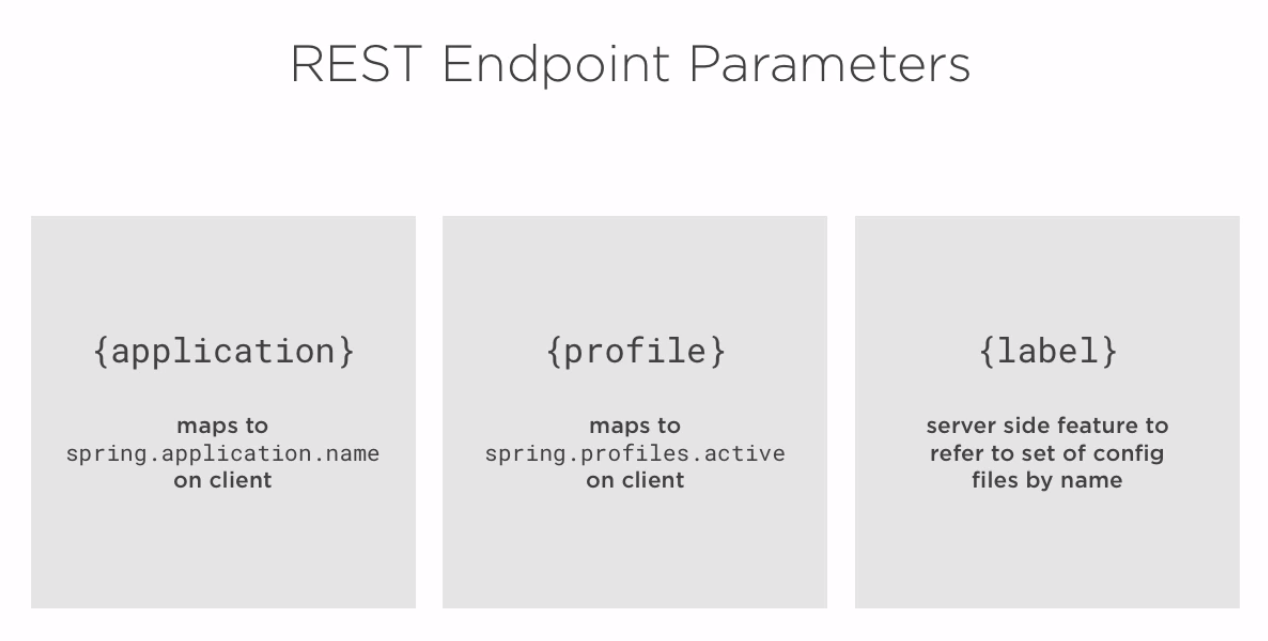
Execution :

1. Start the Discovery Server
2. You have to create 2 instances of the service through Run -> Run As Configurations and start both the instance in different ports 8081 & 8082. Provide the service.instance.name in override properties
3. Start the client
4. Go to the browser and give localhost:8080. You will get response from both the instances.
5. To go to the Spring Cloud Eureka Dashboard, go to localhost:8761

**2.Configuring services using Distributed Configuration:**

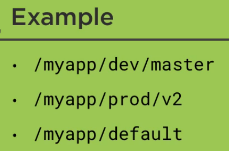
To set up the configuration for a native cloud application(distributed Application), we go for a dedicated configuration server(Spring Cloud Config Server is an implementation of Configuration Server). Spring Cloud Config provides server and client side support for externalized configuration in a distributed system.

1. Config Files accessed by HTTP REST access
2. No writing of config files. It will only serve the config files already stored
3. No need of DB to store the config files. It stores in GIT, SVN, Filesystem
4. Configuration scopes : Global, application specific, profile



REST Endpoints:

* Get/{application}/{profile}/[{label}] – Label is optional

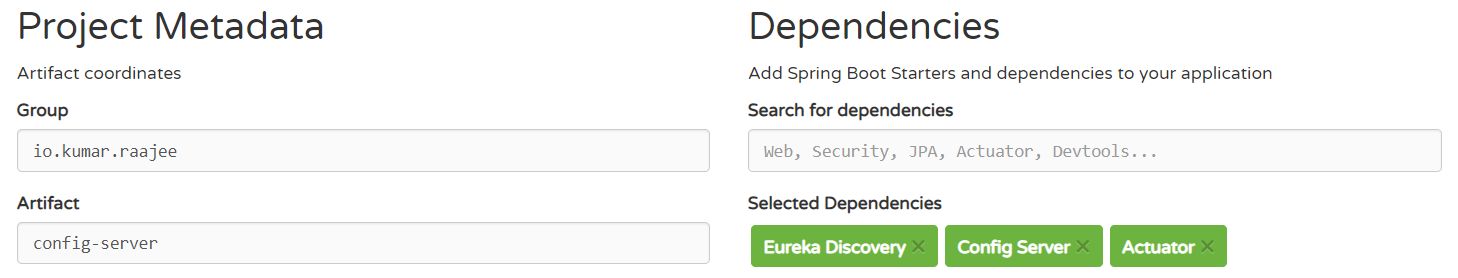


* Get/{application}-{profile}.(yml|properties)
* Get/{label}/{application}-{profile}.(yml|properties)

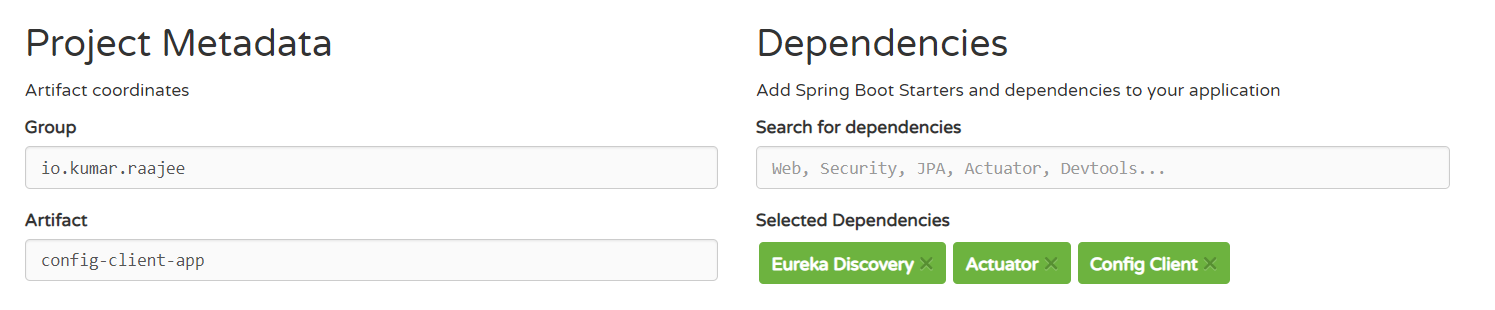
Demo : C:\Users\stkra\Documents\Spring\spring\_cloud\_config\_workspace

Go to <https://start.spring.io/> and create projects for config Server. Import the project in STS through import -> Existing Maven Projects.

Config Server:

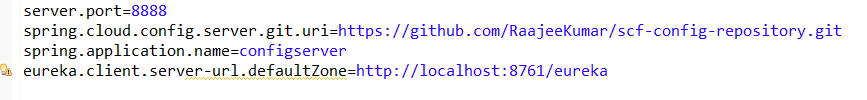


Config Client App:



Config Server:

1. Open up the main application in Config Server Project and add the annotation @EnableConfigServer,@EnableDiscoveryClient(This enables clients to search for the config server in the discovery server)
2. Open up the application .properties and add the following entries,



Where are the actual configurations stored.?

GIT. Go to your github account and fork dustinschultz/scf-config-repository. Copy the clone url and open up the STS GIT perspective and try to clone the url. Once its cloned, expand the folder, right click on the working tree and import maven projects. You can see the configuration files in your workspace.

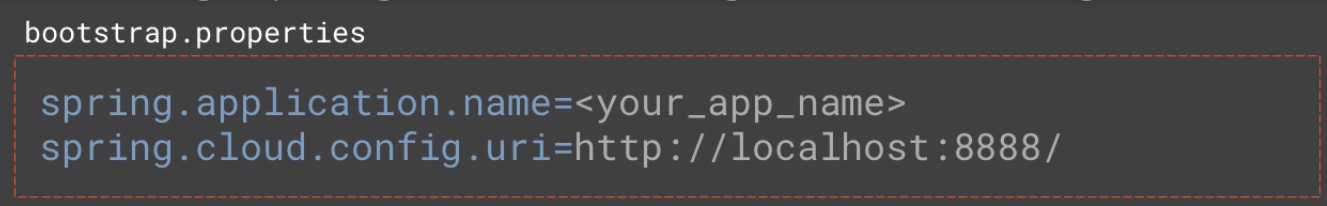
Discovery server:

This is the place where the clients will look for the config server. Go to <https://github.com/dustinschultz/scf-discovery-server> and copy the clone url. Copy the clone url and open up the STS GIT perspective and try to clone the url. Once its cloned,right click on the folder and say import maven projects. You can see the Discovery server project in the workspace.

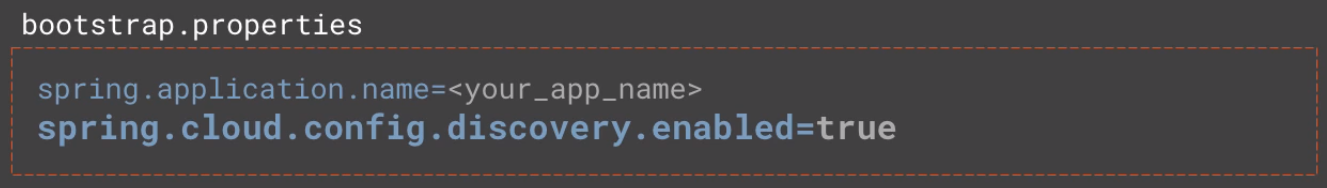
Config Client App : This will bootstrap(It has to fetch the configuration from config server before the spring context is initialized to resolve things like resolving all property placeholders).This can be done by Bootstraping with bootstrap.properties/bootstrap.yaml file

Two ways :

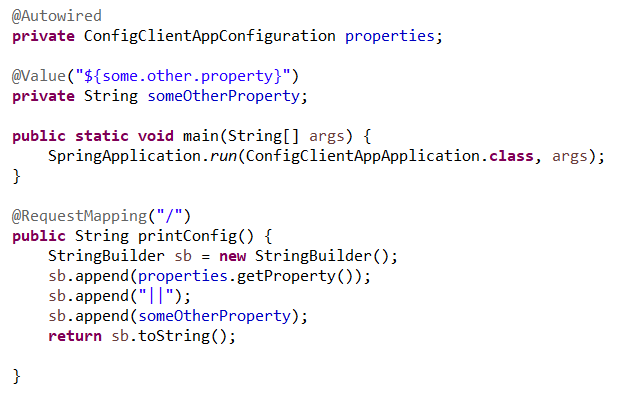
Config first – application name, url to configuration server



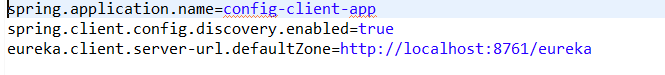
Discovery first – application name, location service discovery server



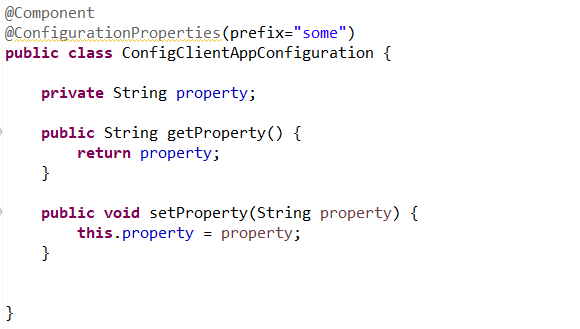
1. Open up the main application in Client Project and add the annotation @EnableDiscoveryClient,@RestController and update the class as follows,



1. Open up the bootstrap .properties and add the following entries,



1. Create a class ConfigClientAppConfiguration



Execution :

1. Start the Discovery Server
2. Start the Config Server
3. Start the client
4. Go to the browser and give localhost:8080. You will see the properties from config server

Updating configuration at runtime :

1. Update the configuration and push it to repository.
2. Then call /refresh endpoint on actuator project.

Actuator Refresh : In postman, method = post, url =http://localhost:8888/refresh

This has to be called on all services using this configuration. Other option is to use /bus/refresh



1. @RefreshScope – This is to refresh a @Bean or @Value that only gets its configuration during initialization.

**3.Mapping services using Intelligent Routing:**

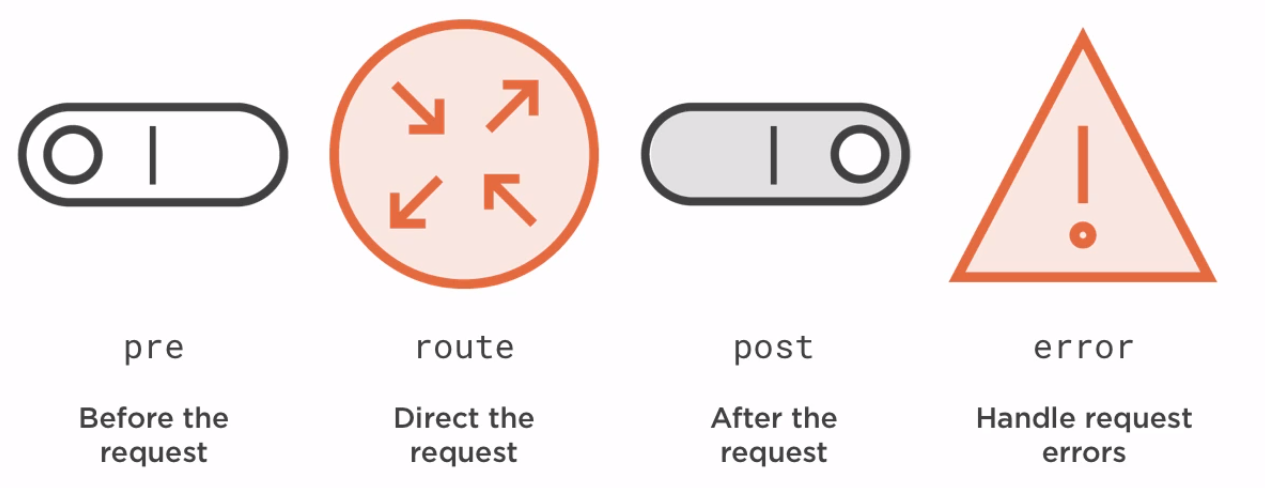
Intelligent Routing – If you have multiple services in cloud with different IP, port. It will be difficult for the client to identify the services.so intelligent routing came into picture. Intelligent Routing is done through gateway. Gateway service provides the following,

* Dynamic Routing & Delivery
* Security & Filtering
* Auditing & Logging
* Request Enhancement
* Load Balancing
* Different APIs for different clients

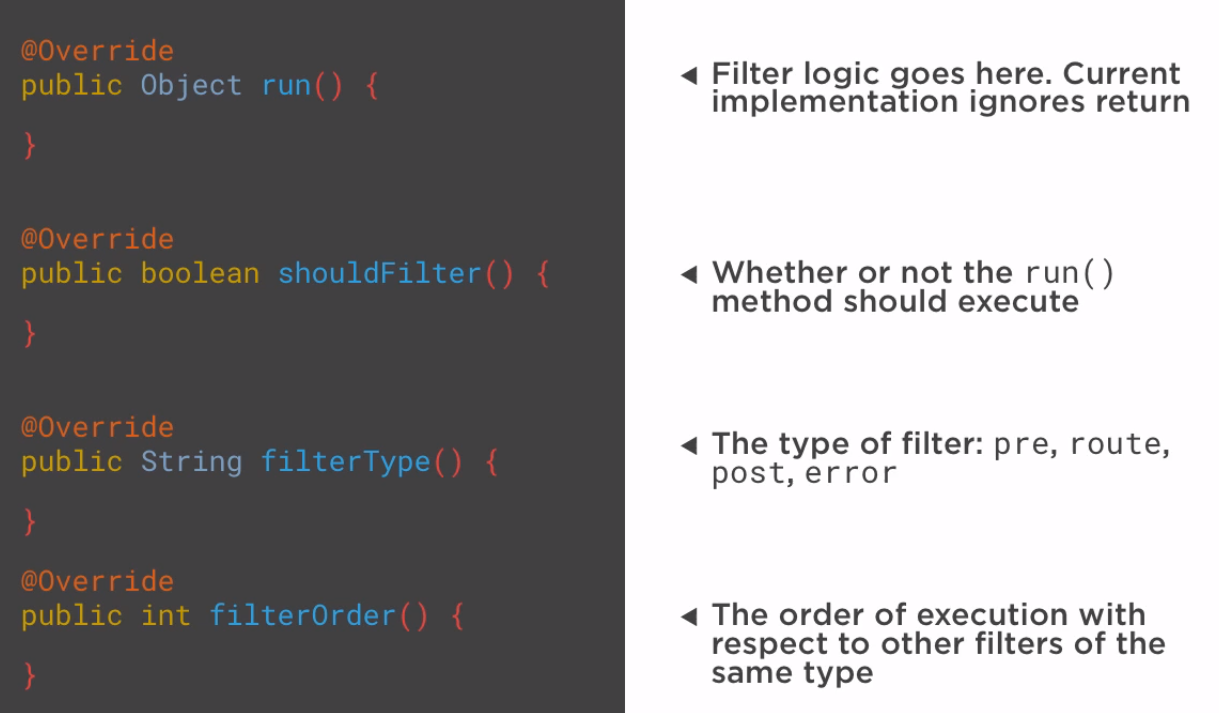
Intelligent Routing is implemented by Netflix Zuul & Spring cloud. You can do it 2 ways. One is through Service Discovery. Other one without it. Ribbon is the load balancing in client side

Zuul allows you to intercept and control requests and responses that pass through the gateway.

Filter Types:



To implement Filter, extend Zuul filter and override 4 methods



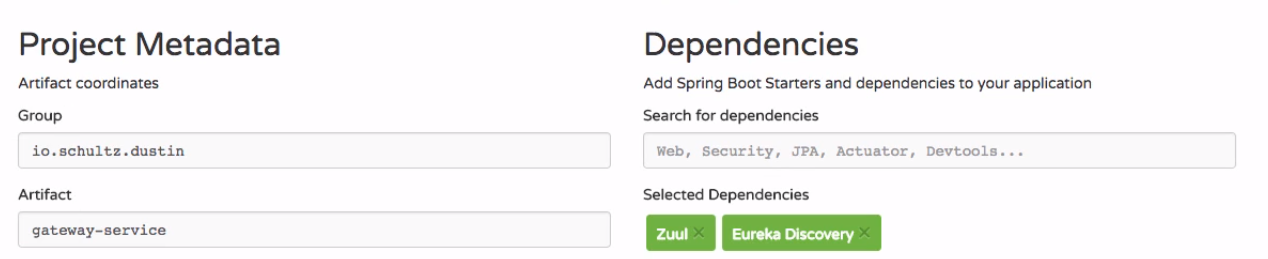
**Demo** - C:\Users\stkra\Documents\Spring\spring\_cloud\_routing\_workspace

Discovery server:

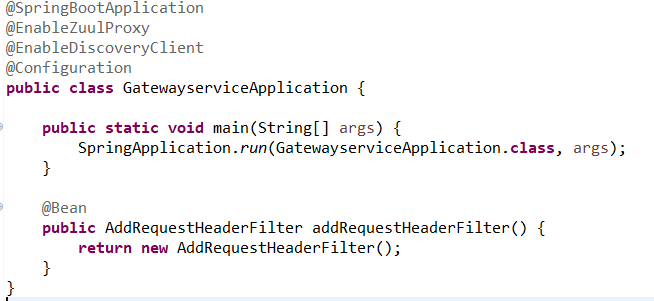
This is the place where the clients will look for the Gateway Service. Go to <https://github.com/dustinschultz/scf-discovery-server> and copy the clone url. Right click on the package Explorer and say import -> Checkout maven projects from SCM. Provide the clone URl,you can see the Discovery server project in the workspace.

Go to <https://start.spring.io/> and create projects for config Server. Import the project in STS through import -> Existing Maven Projects.

Gateway Service:



1. Main Application changes : @EnableZuulProxy turns your class to Gateway service.



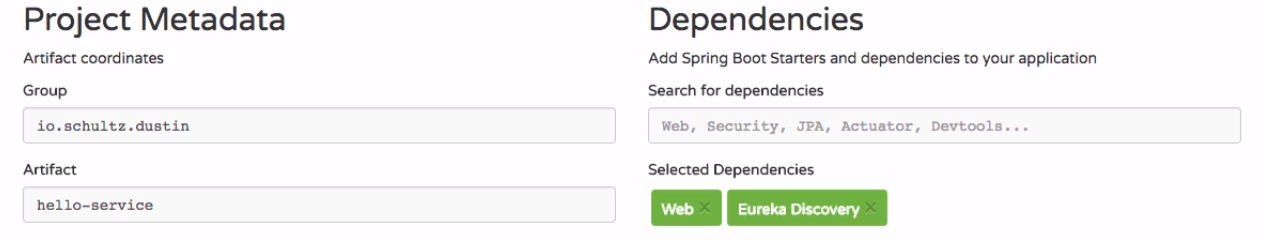
1. Application.properties

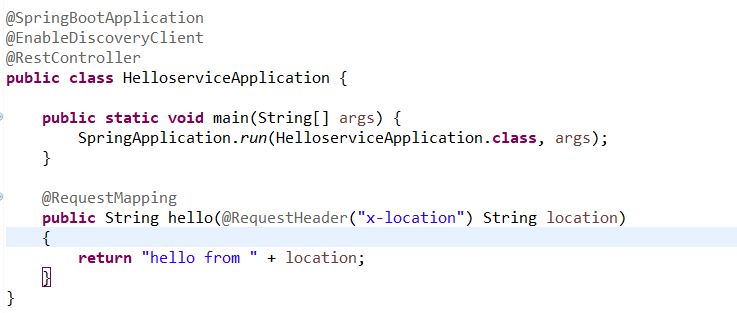


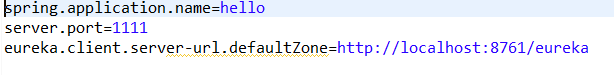
1. Filter code:



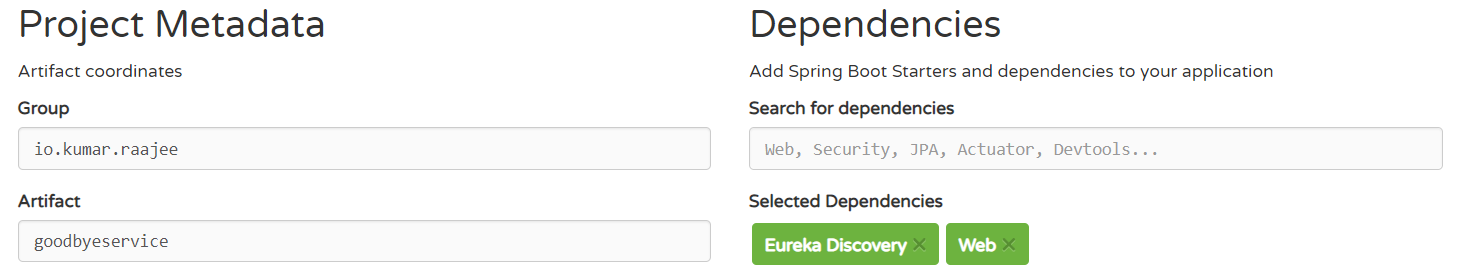
Helloservice:

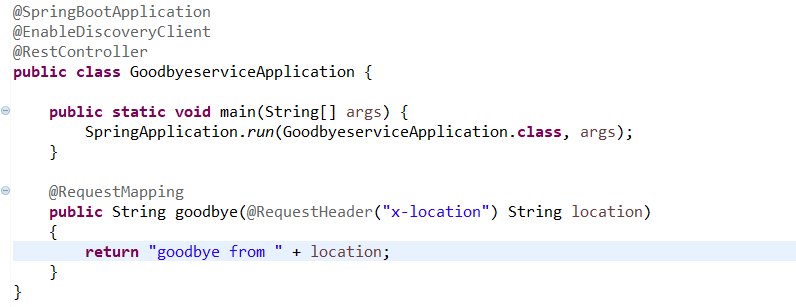


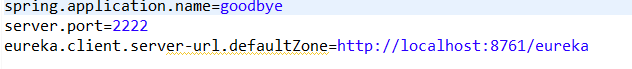




Goodbyeservice:







Execution :

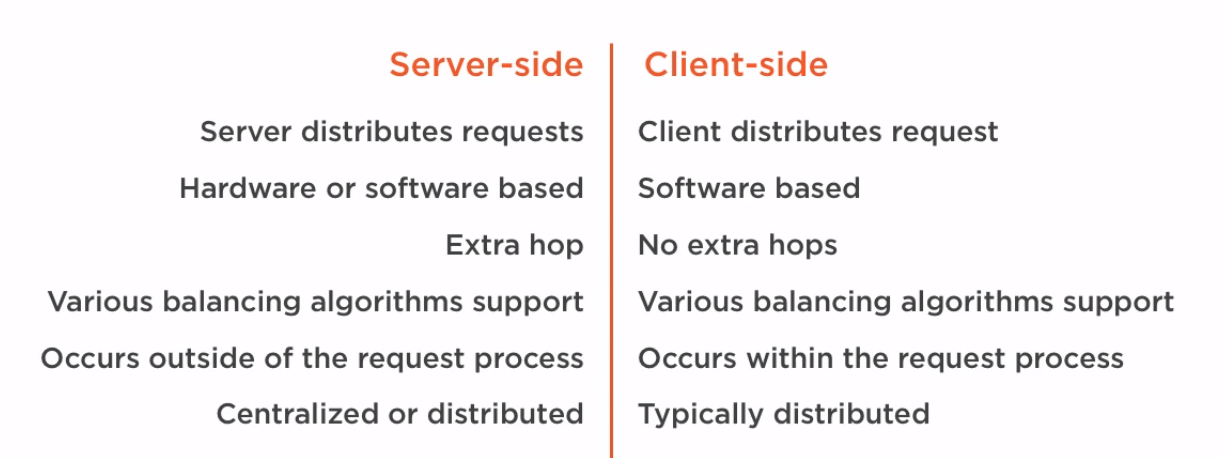
1. Start the Discovery Server, Gateway Service, helloService, goodbyeService
2. Go to the browser and give <http://localhost:8080/goodbye>, <http://localhost:8080/hello>

You will get response from both the services. Request goes through the gateway service.

1. To go to the Spring Cloud Eureka Dashboard, go to localhost:8761

**4.Calling services using Client side Load Balancing:**

Client Load Balancing vs Serverside Load Balancing:



Client side Load Balancing is implemented by Netflix Ribbon + Spring cloud

@LoadBalanced – Marks a Rest template to support Load Balancing

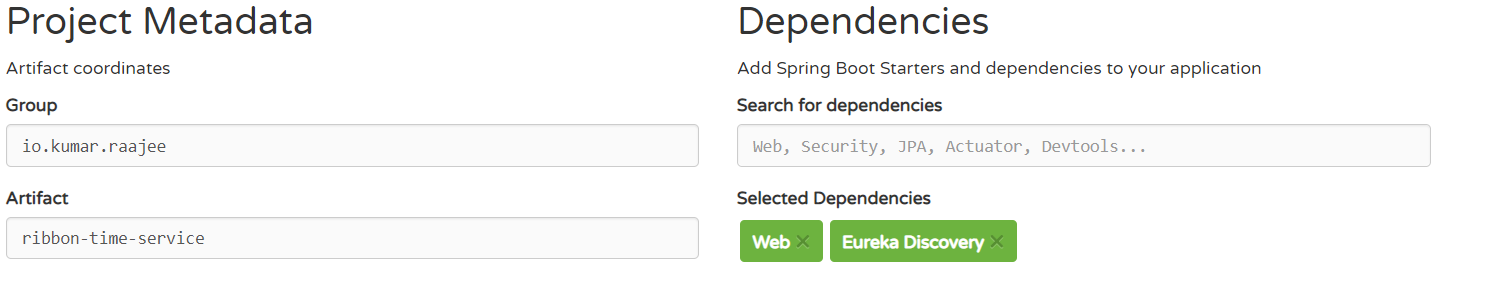
@RibbonClient – Used for custom configuration and when service discovery is absent

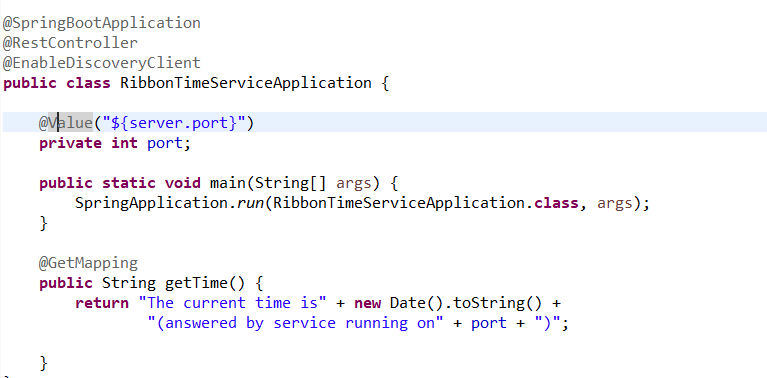
Demo: C:\Users\stkra\Documents\Spring\spring\_cloud\_loadbalancing\_workspace

Discovery server:

This is the place where the clients will look for the Gateway Service. Go to <https://github.com/dustinschultz/scf-discovery-server> and copy the clone url. Right click on the package Explorer and say import -> Checkout maven projects from SCM. Provide the clone URl,you can see the Discovery server project in the workspace.

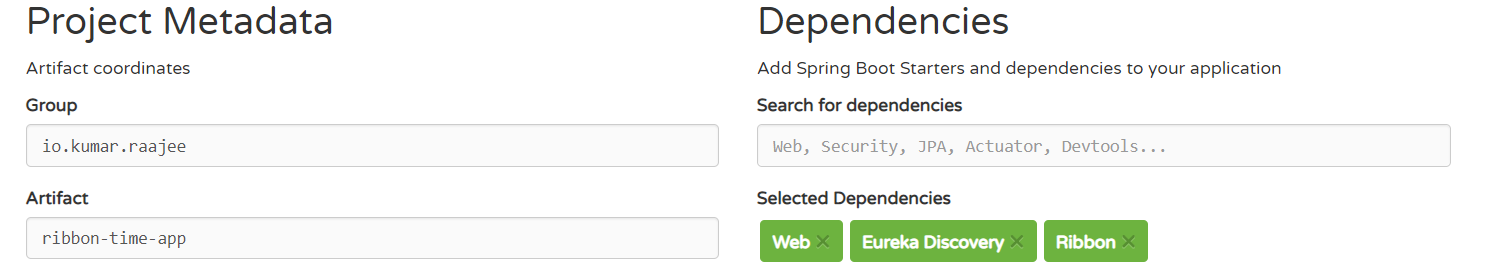
Ribbon-time-service: Create 2 instances of this service by going to Run -> Run As Configurations

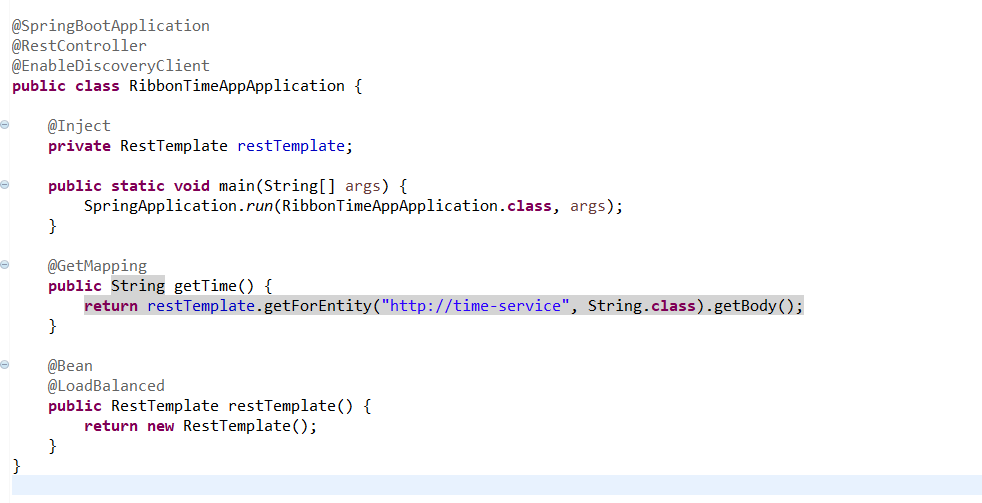




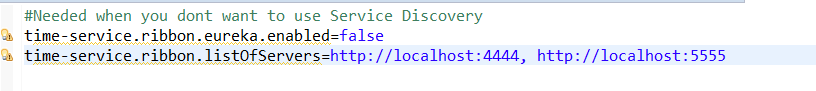


Ribbon-time-app:

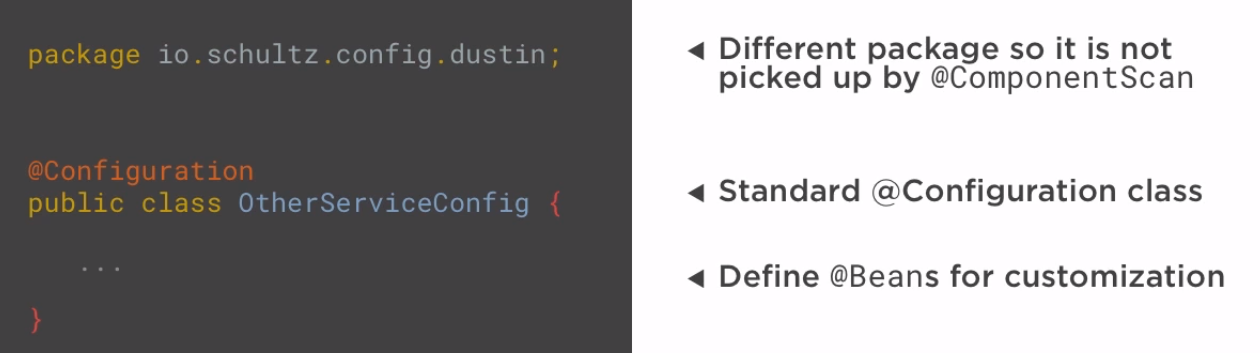




To make use of Ribbon client, In service class remove @EnableDicoveryClient. Then remove the discoveryServer Url from Service Application. Then in client app, remove @EnableDiscoveryClient & add @RibbonClient(name=<Nameof service>). Then add the entries in application.properties



Customizing the Ribbon load balancer through a configuration class



IRule Bean: control load balancing strategy that is used to distribute the load between the instances that are being load balanced

IPing Bean: choose the strategy to check the liveliness or availability of a given instance that is being load balanced

Execution :

1. Start the Discovery Server, ribbon time Service 1 , ribbon time Service 2, ribbon time app
2. Go to the browser and give <http://localhost:8080>. You will get response from both the service instances.
3. To go to the Spring Cloud Eureka Dashboard, go to localhost:8761

**5.Creating self healing services with circuit Breaker:**

Cascading Failures : Failures in one part of system causes the other part to fail.

To avoid cascading failures:

Circuit Breaker Pattern : Used to detect failures and encapsulates logic of preventing a failure to reoccur constantly

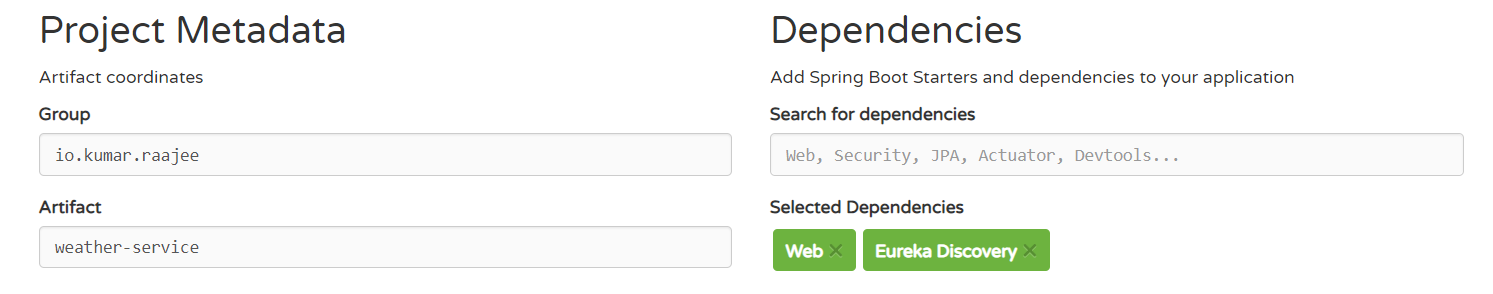
Fault Tolerance is implemented by Netflix Hystrix + Spring Cloud

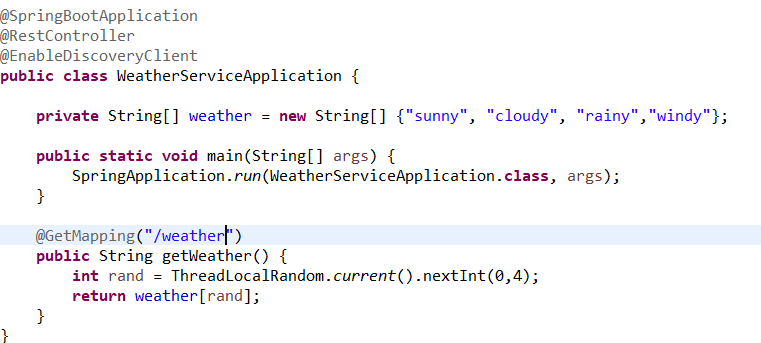
Demo: C:\Users\stkra\Documents\Spring\spring\_cloud\_circuitbreaker\_workspace

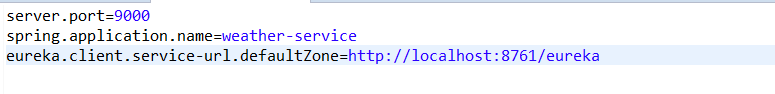
Discovery server:

This is the place where the clients will look for the Gateway Service. Go to <https://github.com/dustinschultz/scf-discovery-server> and copy the clone url. Right click on the package Explorer and say import -> Checkout maven projects from SCM. Provide the clone URl,you can see the Discovery server project in the workspace.

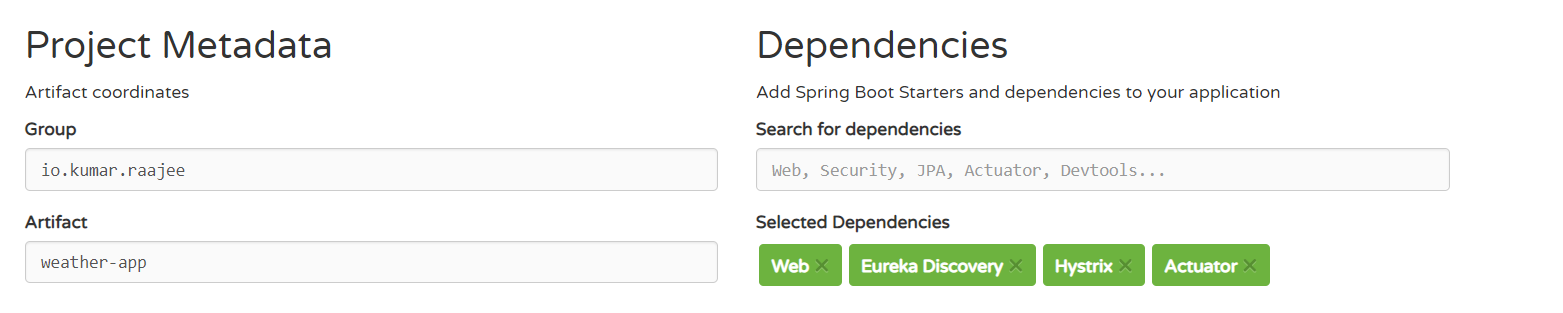
Weather-service:

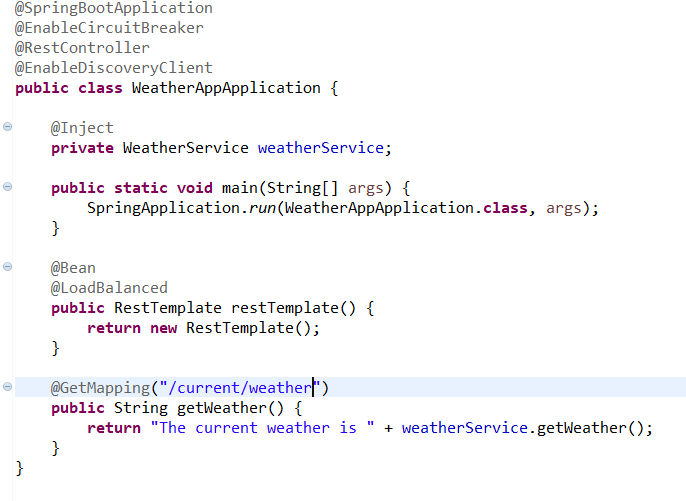




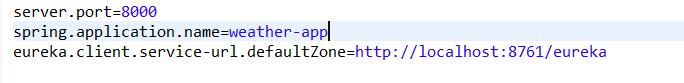


Weather-app:



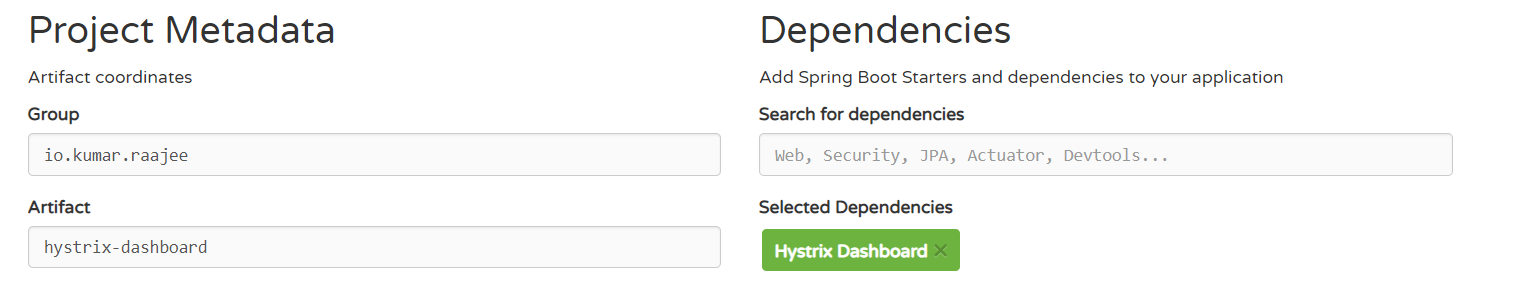


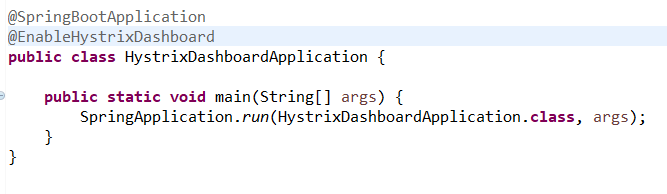




hystrix-dashboard:

Hystrix Dashboard @EnableHystrixDashboard – To enable HystrixDashboard



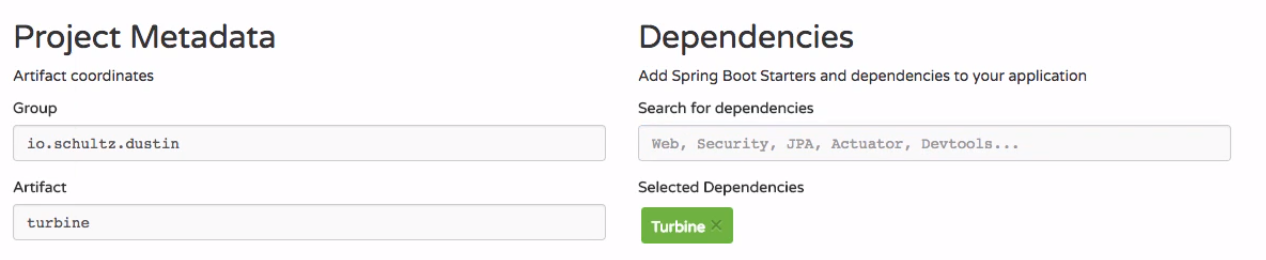


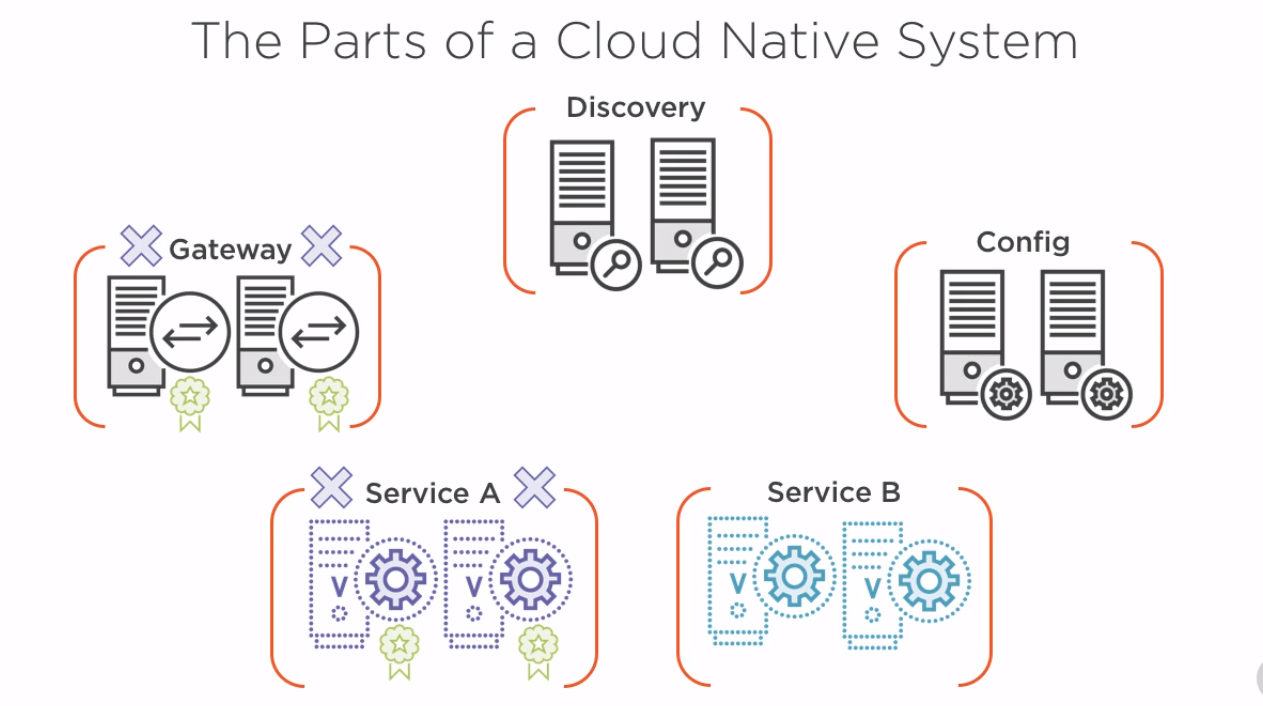
Execution :

1. Start the Discovery Server, weather Service , weather app, hystrix dashboard
2. Go to the browser and give <http://localhost:8000/current/weather>.
3. <http://localhost:8080/hystrix> for hystrix dashboard and provide this url to get the details, <http://localhost:8000/hystrix.stream>
4. To go to the Spring Cloud Eureka Dashboard, go to localhost:8761

Netflix Turbine – Aggregation of many hystrix streams to one

This is implemented by Netflix Turbine + Spring Cloud





1. On start up, services, gateway & config look up at discovery to either register or look up for services
2. For services, its very important as they look up the discovery to find the config server
3. Request comes to the gateway, in our case its Netflix Zuul. Gateway will match the path to a given service id and look it up in the Discovery server.
4. Once the service is located, it will use the hystrix protected call using Ribbon to handle the client side load balancing of which instance it should send the traffic to.
5. If service A is dependent on Service B, it will look up for service B from discovery server. Once its located, it will use the hystrix protected call using Ribbon to handle the client side load balancing and call its instance and return the response to A. A inturn sends to gateway. Gateway returns to the client