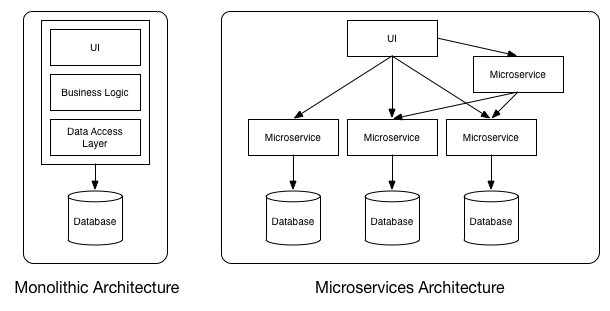
using [Amazon AWS](https://aws.amazon.com/) you can build a cloud application with minimum effort. It’s a good example of what microservices can do.



**Monolithic vs MicroServices Architecture**

As you can see in above diagram, each microservice has it’s own business layer and database. By doing so, changes to one microservice do not impact others.

In general, microservices communicate with each other using widely adopted lightweight protocols, such as HTTP and REST, or messaging protocols, such as JMS or AMQP. In specific scenarios, they can go for more specialized protocols as well.

**Monitor Microservices with Hystrix, Eureka admin and Spring boot admin:**

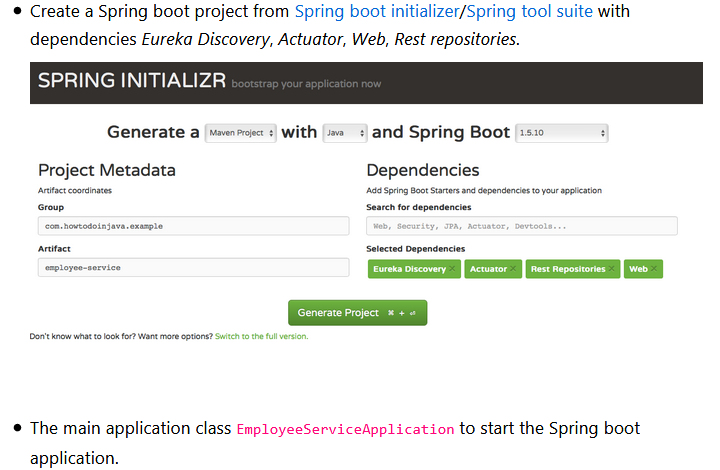
There are many tools available to monitor different health states of these microservices. In this post, we will learn to use three such monitoring tools i.e. **Hystrix dashboard**, **Eureka admin dashboard** and **Spring boot admin dashboard**.

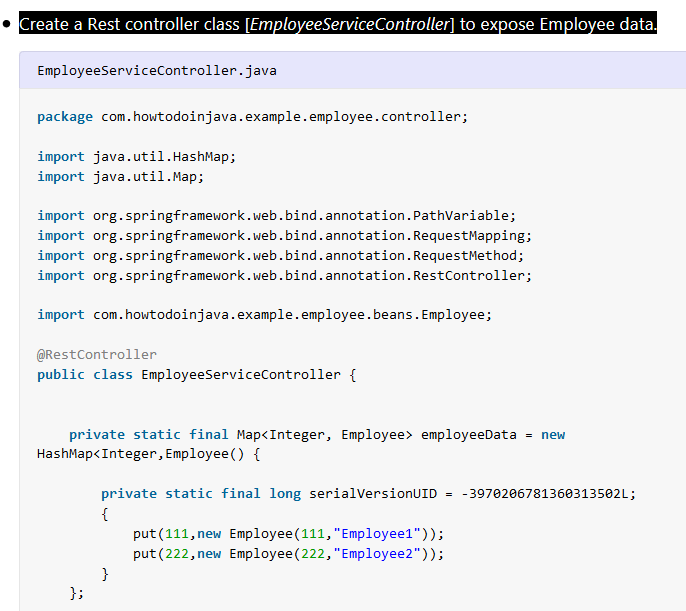
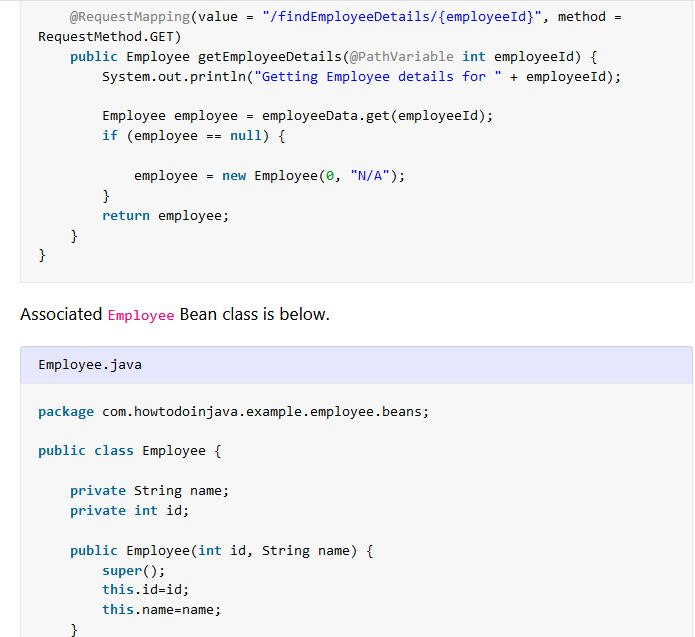
**Description of sample applications**:

we will create three applications. As you may already know that Netflix Eureka server is for building service registry server and associated Eureka clients which will register themselves to lookup other services and communicate through REST API’s.

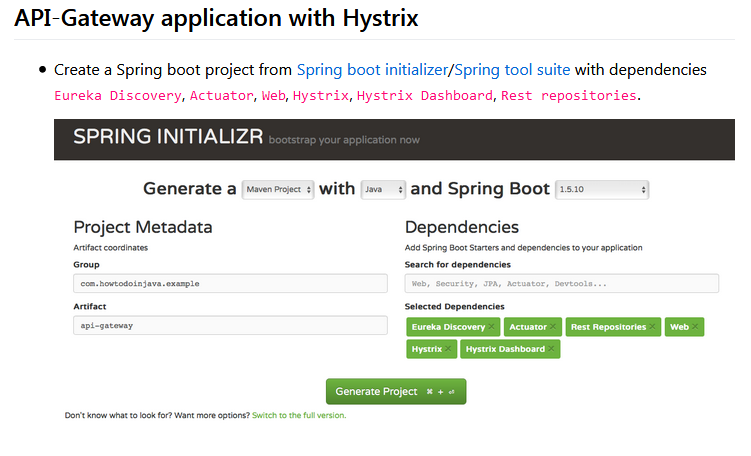
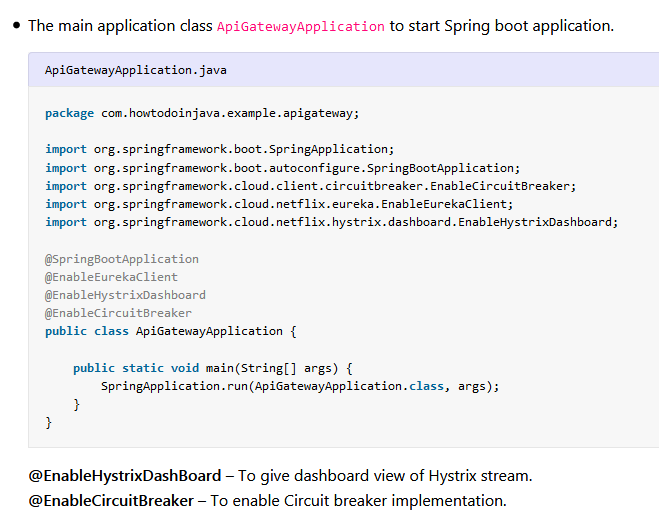
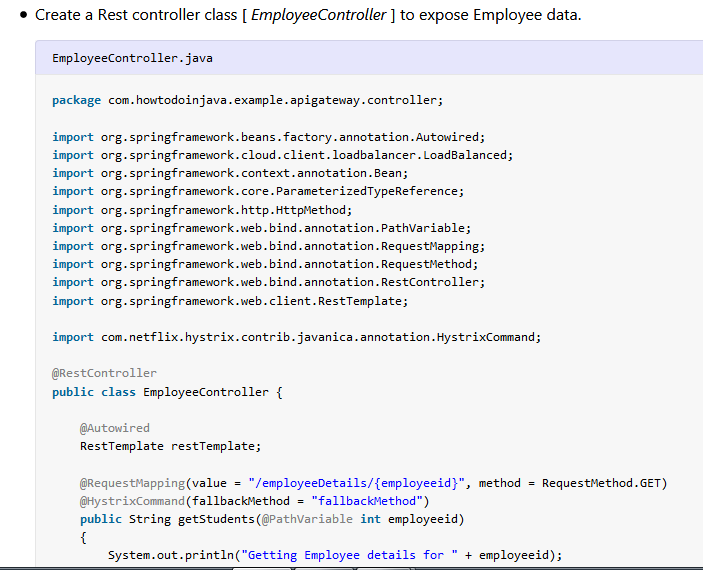
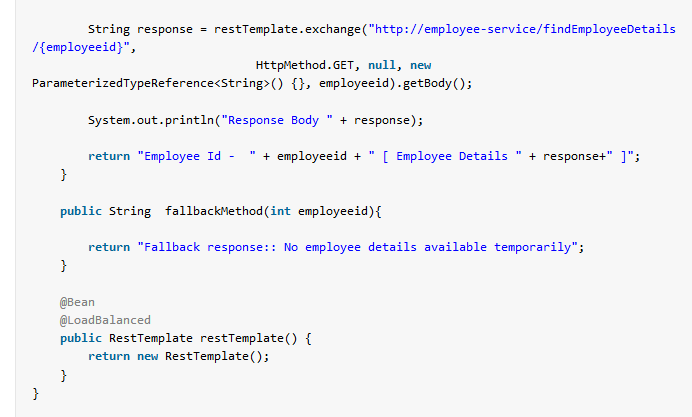
1. **Employee Service** – This microservice application is responsible to fetch data of Employees.
2. **Api-Gateway** – This application is to provide common gateway while accessing different microservices. In the following example it will act as a gateway to Employee Service above.
3. **Eureka Server** – This microservice application will provide service discovery and registration of above microservices.

## Employee Service application:

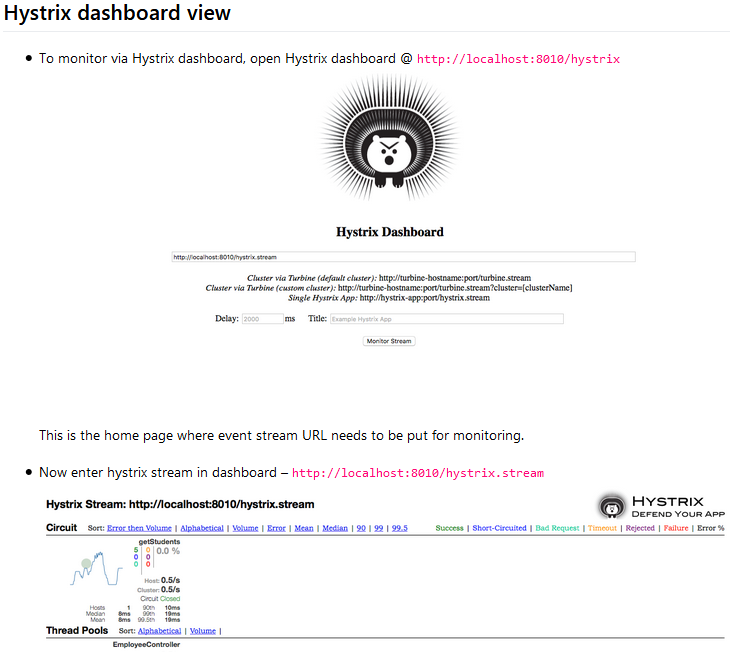


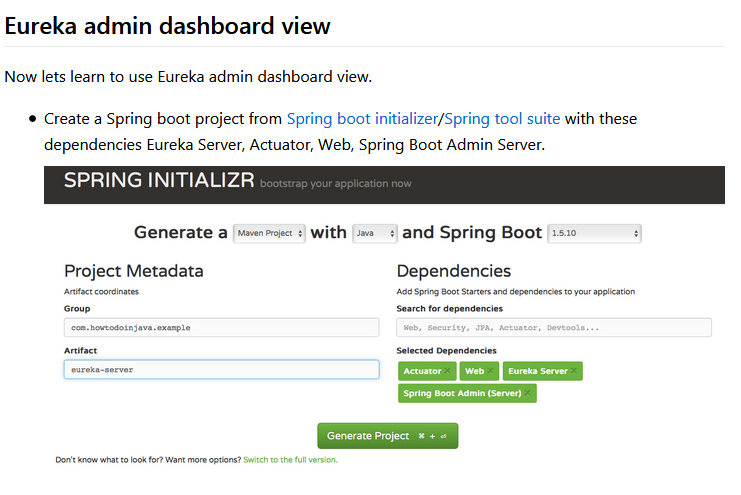


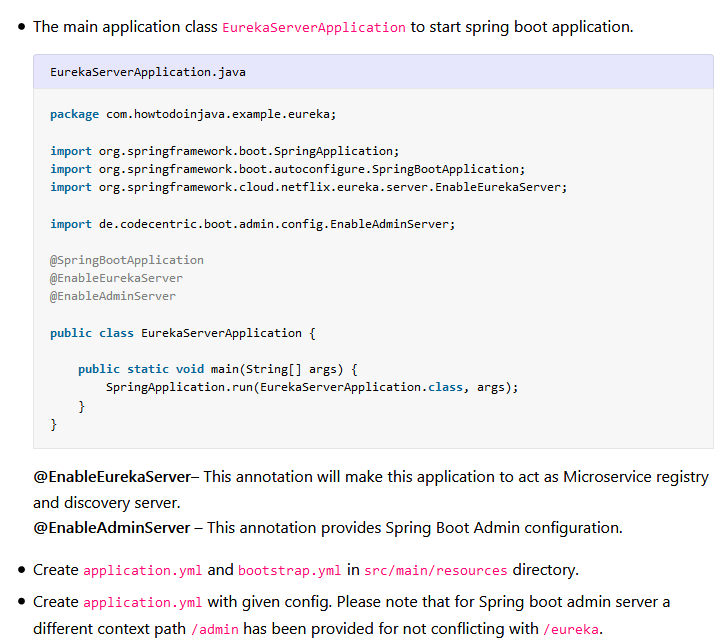
   





This provides realtime information of all Hystrix commands and thread pools.







# Spring Cloud Config Server with Git Integration:

# Config server is where all configurable parameters of microservices are written and maintained. It is more like externalizing properties / resource file out of project codebase to an external service altogether, so that any changes to that property does not necessitate the deployment of service which is using the property. All such property changes will be reflected without redeploying the microservice.

## Why to Use Spring Cloud Config Server:

The idea of config server has come from the [12-factor app](https://12factor.net/config) manifesto related to best practice guideline of developing modern cloud native application. It suggests **to keep properties / resources in the environment of the server** where the values of those resources vary during run time – usually different configurations that will differ in each environment.

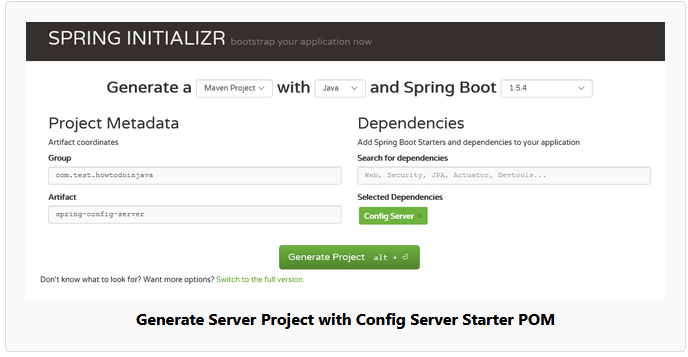
To start with we will develop two Microservices using spring boot.

1. one is the **config server service**, providing the configuration in runtime
2. one is the **config client service**, using the configuration exposed as config server.

## Config Server – Server Side Configuration:

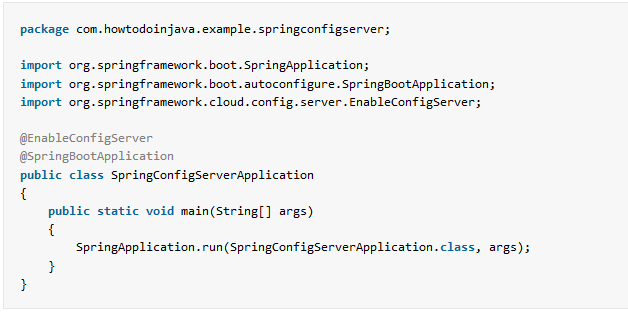
#### 1.Generate the project structure:

#### Here we will choose only Config server starter pom.

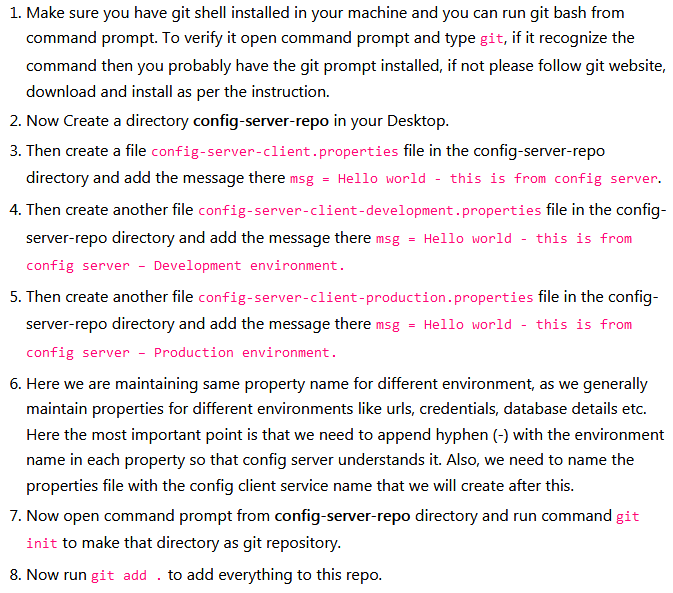


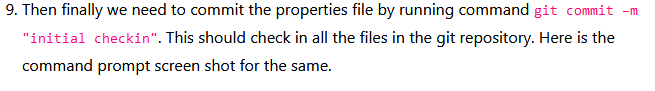
#### Add Config Server Annotation

Now open the Spring Application class that spring already has provided and add the [EnableConfigServer](https://github.com/spring-cloud/spring-cloud-config/blob/master/spring-cloud-config-server/src/main/java/org/springframework/cloud/config/server/EnableConfigServer.java) annotation before the class and build the project. With this annotation, this artifact will act like a spring config server.



#### Create the Git repository:





Note: Do any change in the value of any environment’s property and check-in that file and then run that specific environment’s endpoint, and verify that changed value should be reflected immediately without restarting the server – **that is the magic of Spring Config Server**.

# Spring Cloud Service Discovery with Netflix Eureka:

# We will use Netflix Eureka server for building the service registry server and Eureka clients which will register themselves and discover other services to call REST APIs.

## Overview

We will create three microservices for this **Netflix Eureka example**.

1. **Eureka Service Registry Server** – This microservice will provide the service registry and discovery server.
2. **Student Microservice** – Which will give some functionality based on Student entity. It will be a rest based service and most importantly it will be a eureka client service, which will talk with eureka service to register itself in the service registry.
3. **School Microservice** – Same type as of Student service – only added feature is that it will invoke Student service with service look up mechanism. We will not use absolute URL of student service to interact with that service.

# 

## What is Netflix Eureka Server and Clients?

The transition from Monolithic to Microservice based architecture gives many benefits for future in terms of maintainability, scalability, high availability etc. However at the same time, there are many challenges also while doing this migration. One of them is to maintain individual Microservices addresses.

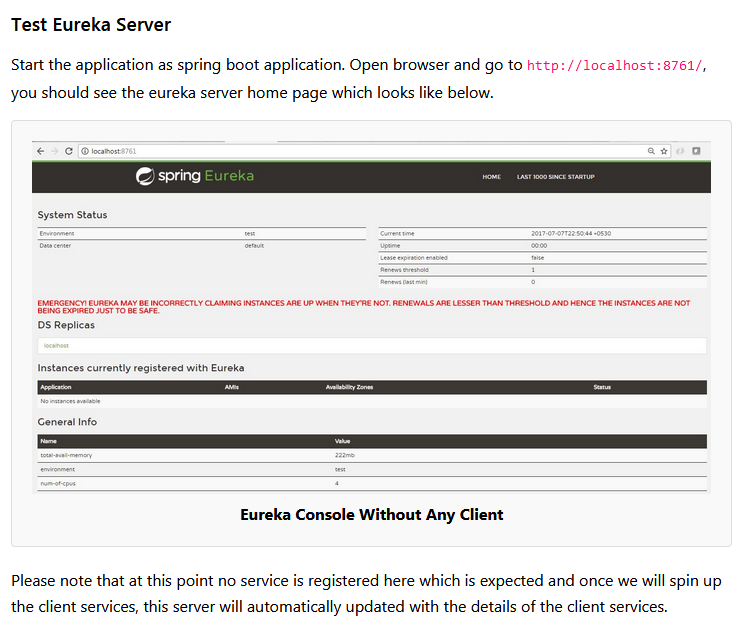
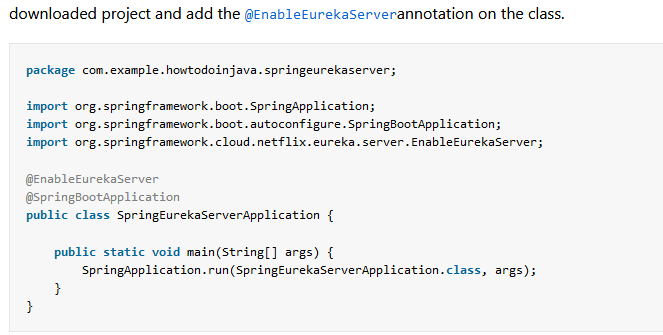
There is a concept called ‘Service registration and discovery’ where one dedicated server is responsible to maintain the registry of all the Microservice that has been deployed and removed. This will act like a phone book of all other applications/microservices.

It is as a lookup service where microservices (clients) can register themselves and discover other registered microservices. When a client microservice registers with Eureka it provides metadata such as host, port, and health indicator thus allowing for other microservices to discover it. The discovery server expects a regular heartbeat message from each microservice instance. If an instance begins to consistently fail to send a heartbeat, the discovery server will remove the instance from his registry. This way we will have a very stable ecosystem of Microservices collaborating among each other, and on top of it we don’t have to manually maintain address of other Microservice.

## Eureka Service Registry Server

#### Create Eureka Server

Create a Spring boot project from [Spring Boot initializer portal](https://start.spring.io/) with two dependencies i.e. Eureka server and Actuator



## Eureka Client – Student Service

Follow these steps to create and run Eureka client running student service.

#### Create Eureka Client Project

Create a Spring boot project from initializer portal with four dependencies i.e. Actuator, Web, Rest Repositories, Eureka Discovery.

#### Add REST API

Now add one RestController and expose one rest endpoint for getting all the student details for a particular school. Here we are exposing /getStudentDetailsForSchool/{schoolname} endpoint to serve the business purpose. For simplicity, we are hard coding the student details.

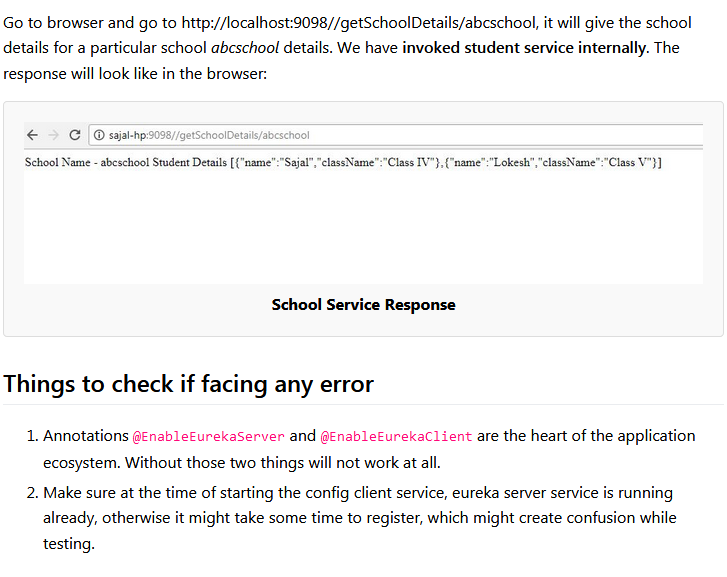
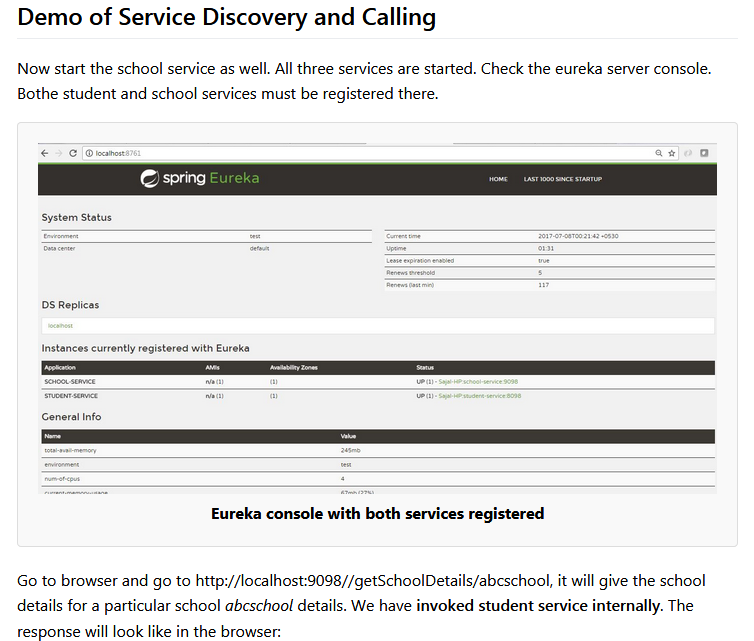
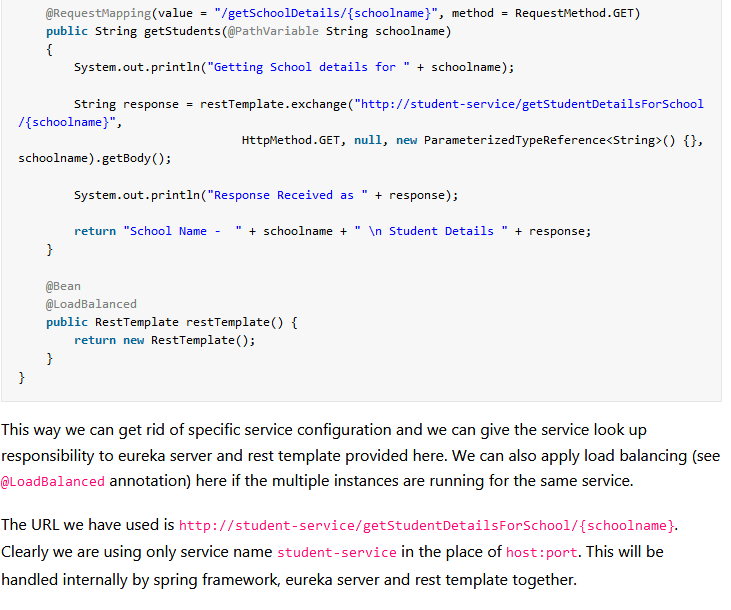
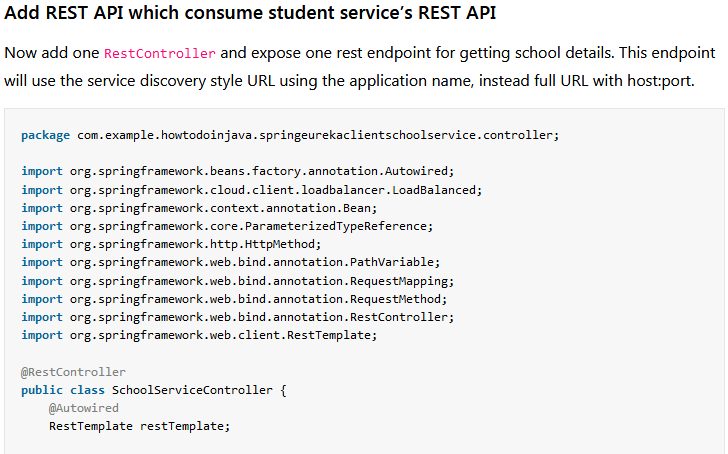
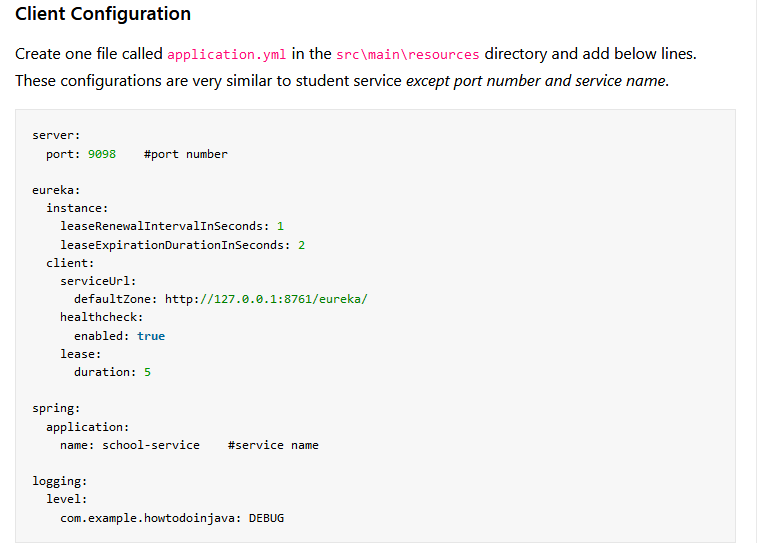
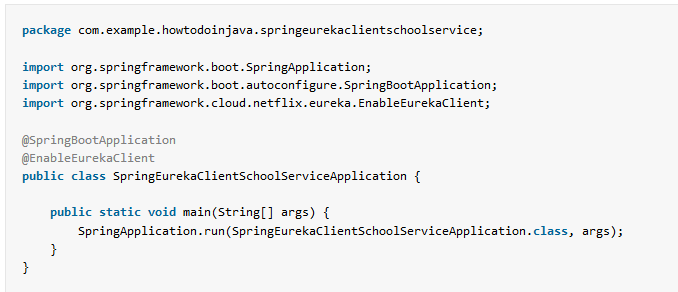
## Eureka Client – School Service

Now we will create school service which will register itself with eureka server – and it will discover and invoke student-service without hardcoded URL path.

#### Create Eureka Client Project

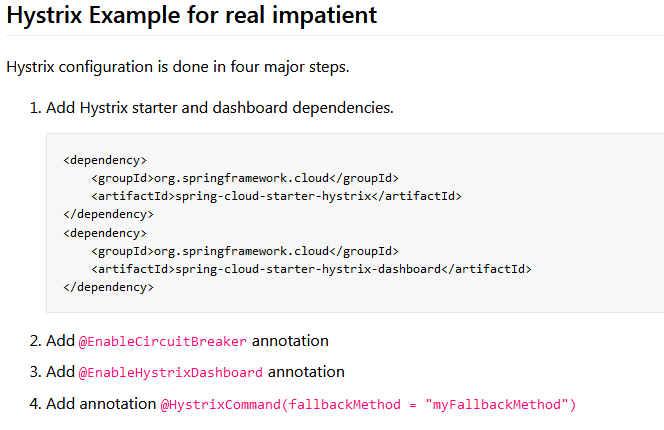
Create a Spring boot project from initializer portal with four dependencies i.e. Actuator, Web, Rest Repositories, Eureka Discovery.

Now add the @EnableEurekaClient annotation on Spring boot application class present in src folder. With this annotation, this artifact will act like a spring discovery client and will register itself in the eureka server attached to this service.



# Hystrix Circuit Breaker Pattern – Spring Cloud:

[Hystrix](https://github.com/Netflix/Hystrix) is used to implement **circuit breaker** while invoking underlying [microservice](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/). It is generally required to enable fault tolerance in the application where some underlying service is down/throwing error permanently, we need to fall back to different path of program execution automatically.



## What is Circuit Breaker Pattern?

If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.

Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.

## Hystrix Circuit Breaker Example

To demo circuit breaker, we will create following two microservices where first is dependent on another.

* **Student Microservice** – Which will give some basic functionality on Student entity. It will be a REST based service. We will call this service from School Service to understand Circuit Breaker. It will run on port 8098 in localhost.
* **School Microservice** – Again a simple REST based microservice where we will implement circuit breaker using Hystrix. Student Service will be invoked from here and we will test the fall back path once student service will be unavailable. It will run on port 9098 in localhost.

## Create Student Service

Follow these steps to create and run Student Service – a simple REST service providing some basic functionality of Student entity.

#### Create spring boot project

Create a Spring boot project from [Spring Boot initializer portal](https://start.spring.io/) with three dependencies i.e. Web, Rest Repositories and Actuator.

#### Server Port Settings

Open application.properties and add port information.

|  |
| --- |
| server.port = 8098 |

#### Create REST APIs

Now add one REST controller class called StudentServiceController and expose one rest endpoint for getting all the student details for a particular school. Here we are exposing /getStudentDetailsForSchool/{schoolname} endpoint to serve the business purpose. For simplicity, we are hard coding the student details.

## Create School Service – Hystrix Enabled

Similar to Student service, create another microservice for School. It will internally invoke already developed Student Service.

#### Generate spring boot project

Create a Spring boot project from [Spring Boot initializer portal](https://start.spring.io/) with those dependencies mainly.

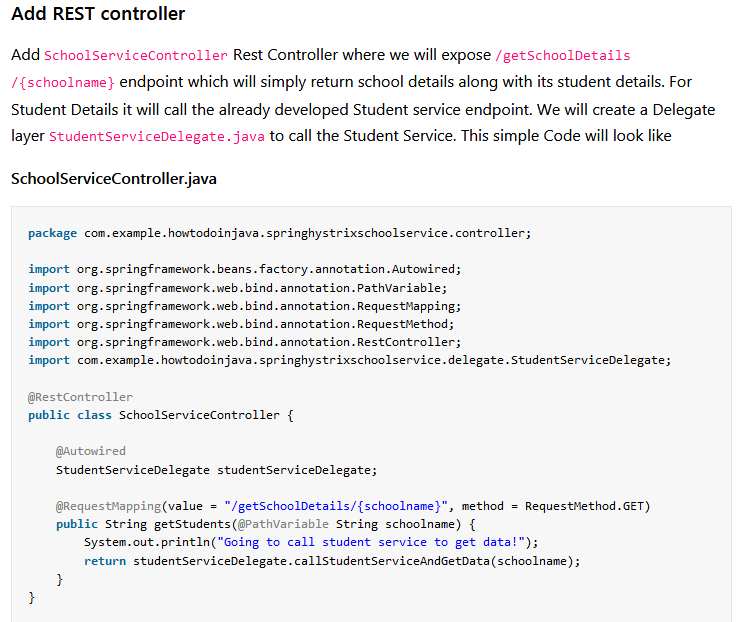
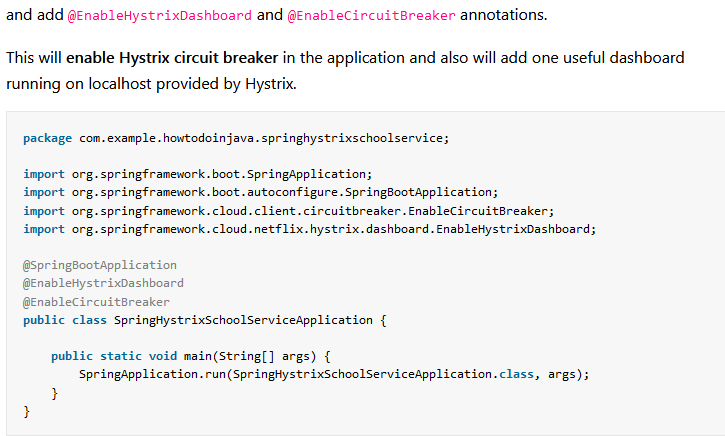
* **Web** – REST Endpoints
* **Actuator** – providing basic management URL
* **Hystrix** – Enable Circuit Breaker
* **Hystrix Dashboard** – Enable one Dashboard screen related to the Circuit Breaker monitoring

#### Server Port Settings

Open application.properties and add port information.

|  |
| --- |
| server.port = 9098 |

#### Enable Hystrix Settings

**StudentServiceDelegate**

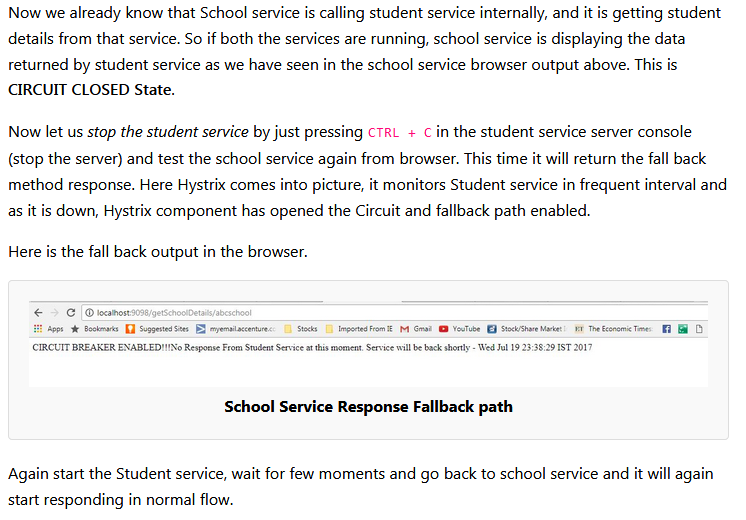
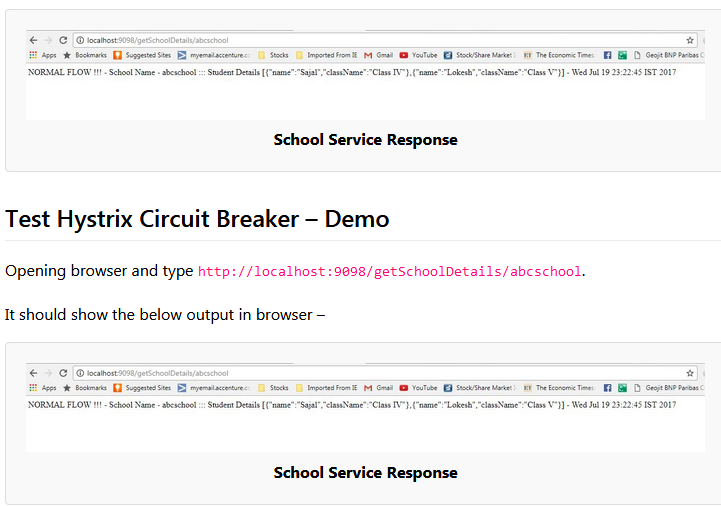
We will do the following things here to enable Hystrix circuit breaker.

* Invoke Student Service through spring framework provided RestTemplate
* Add Hystrix Command to enable fallback method – @HystrixCommand(fallbackMethod = "callStudentServiceAndGetData\_Fallback") – this means that we will have to add another method callStudentServiceAndGetData\_Fallback with same signature, which will be invoked when actual Student service will be down.
* Add fallback method – callStudentServiceAndGetData\_Fallback which will simply return some default value.

#### Build and Test of School Service

Now do a final build using mvn clean install and run the server using command java -jar target\spring-hystrix-school-service-0.0.1-SNAPSHOT.jar. This will start the school service in default port **9098**.

Start the student service as described above and then test school service by opening browser and type http://localhost:9098/getSchoolDetails/abcschool. It should show the below output in browser :



# Netflix zuul example – zuul api gateway pattern – spring cloud tutorial:

## Zuul Components

Zuul has mainly four types of filters that enable us to intercept the traffic in different timeline of the request processing for any particular transaction. We can add any number of filters for a particular url pattern.

* **pre filters** – are invoked before the request is routed.
* **post filters** – are invoked after the request has been routed.
* **route filters** – are used to route the request.
* **error filters** – are invoked when an error occurs while handling the request.

## Overview of netflix zuul example

Now let’s make our hands dirty by creating a simple yet meaningful ecosystem using Zuul proxy. We will create below artifacts to demonstrate the whole thing:

* **Student Microservice** – a spring boot based microservice which will just expose a single url to enable some search functionality. For simplicity we will just return hardcoded values, but in real world we can connect to anywhere from this service to get the data.
* **Zuul gateway service proxy** – It would be again a spring boot based, which will basically intercept all the traffic of student service and apply series of request filter and then route to the underlying service and again at the time of response serving, it will apply some response filtering. Since it is a gateway, we can literally take many interesting and useful action using the filters effectively.

Some of the common responsibility of gateway service are –

* + Apply **microservice authentication and security** in the gateway layer to protect the actual services

## Create Student Microservice

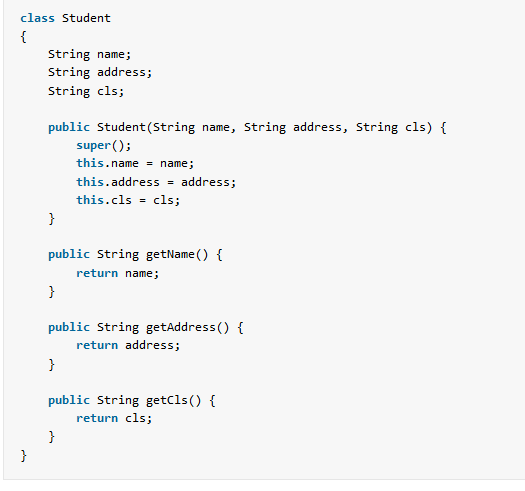
Follow these steps to develop student microservice which will expose couple of REST endpoints which would be later accessed via zuul proxy. We will look into the zuul part later, let’s now create the student service first.

#### 4.1. Create Spring Boot Project

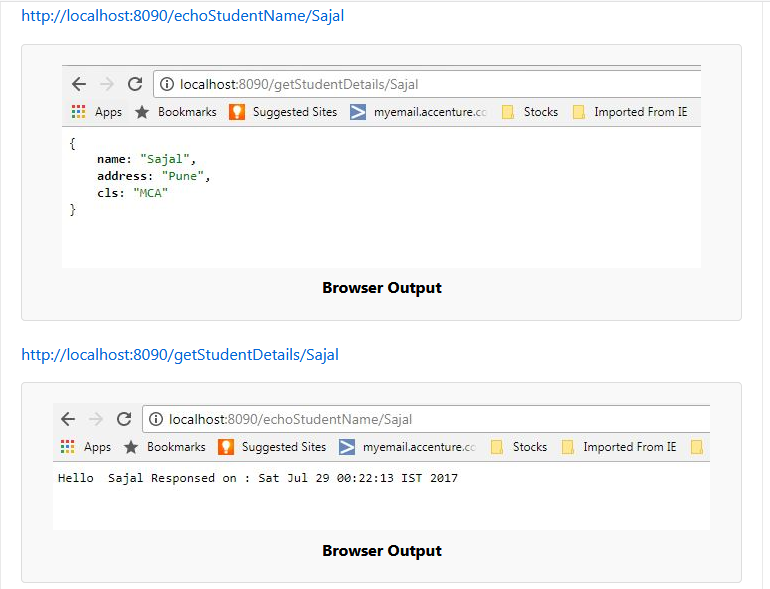
Create a Spring boot project from [spring initializer portal](https://start.spring.io/) with dependencies i.e.Web and Rest Repositories.

#### 4.2. Add few REST Endpoint

We will now just add few REST endpoints to this service for testing the proxy later. To do that we need to add one REST controller by adding annotation @RestController. For simplicity, we will add one model class Student.



#### 4.4. Verify Student Service



Now we will create the actual proxy service using Zuul.

## 5. Create Zuul Gateway Service Proxy

This will be again a spring boot based microservice, but it has a special feature. It will use zuul to create a API gateway proxy which will proxy the student service. Later we can add any number of microservices like student service and able to create a strong microservice ecosystem.

#### 5.1. Create Spring Boot Project

Create a Spring boot project from [spring initializer portal](https://start.spring.io/) with Zuul dependency

#### 5.2. Enable Zuul Service Proxy

Now add the @**EnableZuulProxy** annotation on Spring boot application class present in src folder. With this annotation, this artifact will act like a Zuul service proxy and will enable all the features of a API gateway layer as described before. We will then add some filters and route configurations.



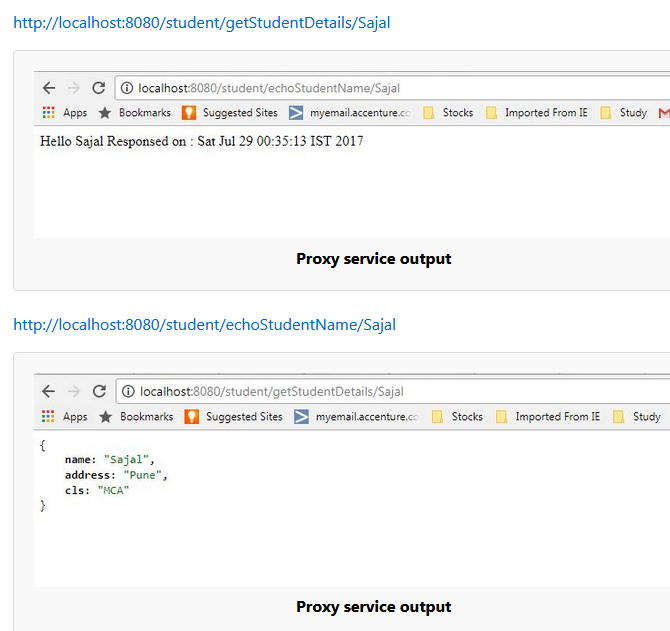
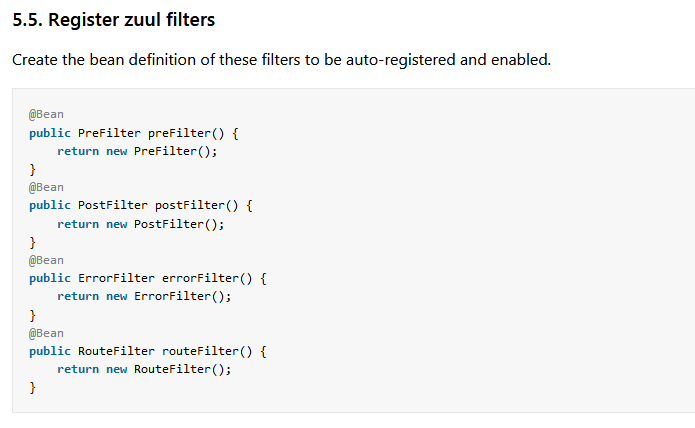
#### 5.4. Add Zuul Filters

We will now add few filters as we have already described, Zuul supports 4 types of filters namely pre,post,route and error. Here we will create each type of filters.

To write a filter we need to do basically those steps:

* Need to extend com.netflix.zuul.ZuulFilter
* Need to override filterType, filterOrder, shouldFilter and run methods. Here filterType method can only return any one of four String – pre/post/route/error. Depedending on this value the filter will act like a particular filter.
* run method is the place where our filter logic should be placed depending on our requirement.
* Also we can add any number of any particular filter based on our need, this case filterOrder will come into place to determine the order of that filter at the phase of execution of that type of filter.

**pre filter code** – We will add the below pre filter. Currently filters are doing nothing apart from a println for testing purpose. But actually those are powerful enough to do many important aspects as mentioned before.



# Spring cloud ribbon with eureka – Client side load balancer example:

# We use **ribbon** as client side load balancer and **eureka** as registry service. We will Learn how we can dynamically add new instances of microservices under the load balancer.

## 1. Traditional server side load balancing

Server side load balancing is involved in monolithic applications where we have limited number of application instances behind the load balancer. We deploy our war/ear files into multiple server instances which are basically a pool of server having the same application deployed and we put a load balancer in front of it.

#### 1.1. Problems in microservices architecture

# Mostly server side load balancing is a manual effort and we need to add/remove instances manually to the load balancer to work. So ideally we are loosing the today’s on demand scalability to auto-discover and configure when any new instances will be spinned of.

## Client side load balancing

To overcome the problems of traditional load balancing, client side load balancing came into picture. They reside in the application as inbuilt component and bundled along with the application, so we don’t have to deploy them in separate servers.

In microservice architecture, we will have to develop many microservices and each microservice may have multiple instances in the ecosystem. Now if one microservice wants to communicate with another microservice, it generally looks up the service registry using discovery client and Eureka server returns all the instances of that target microservice to the caller service. Then it is the responsibility of the caller service to choose which instance to send request.

Here the client side load balancing comes into picture and automatically handles the complexities around this situation and delegates to proper instance in load balanced fashion. Note that we can specify the load balancing algorithm to use.

## 3. Netflix ribbon – Client side load balancer

Netflix ribbon from Spring Cloud family provides such facility to set up client side load balancing along with the service registry component. Spring boot has very nice way of configuring ribbon client side load balancer with minimal effort. It provides the following features

1. Load balancing
2. Fault tolerance
3. Multiple protocol (HTTP, TCP, UDP) support in an asynchronous and reactive model
4. Caching and batching

|  |
| --- |
| pom.xml |
| <dependency>      <groupId>com.netflix.ribbon</groupId>      <artifactId>ribbon</artifactId>      <version>2.2.2</version>  </dependency> |

We will create the following components and see how the whole eco system coordinates in distributed environment.

* Two microservices using Spring boot. One needs to invoke another as per business requirement
* Eureka service registry server
* Ribbon in the invoking microservice to call the other service in load balanced fashion WITH service discovery
* Invoking service in load balanced manner WITHOUT service discovery

#### 4.2. Create backend microservice

