**Spring Microservices, Spring Cloud with Spring Boot**

# **Spring Boot**

Spring Boot is an efficient framework for creating a stand-alone Spring-based application that you can “just run”. It provides non-functional features:

* Embedded servers like Tomcat, Jetty or Undertow directly (no need to deploy WAR files) which are easy to deploy with the containers
* It helps in monitoring the multiples components
* It helps in configuring the components externally
* Create stand-alone Spring applications
* Provide opinionated 'starter' dependencies to simplify your build configuration
* Automatically configure Spring and 3rd party libraries whenever possible
* Provide production-ready features such as metrics, health checks and externalized configuration
* Absolutely no code generation and no requirement for XML configuration

# **What is Microservices?**

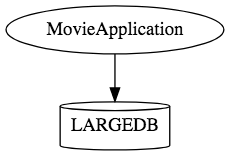
A microservice is an engineering approach focused on **decomposing** applications into **single-function** modules with **well-defined interfaces** which are **independent**ly deployed and operated by **small teams** who own the **entire lifecycle**of the service.

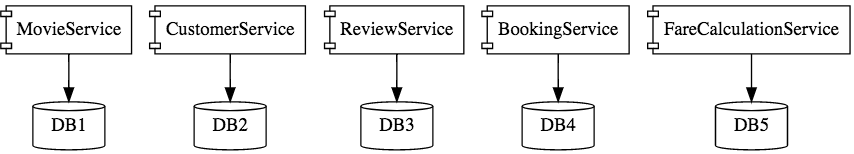
**Microservices** *in a nutshell*allows us to break our large system into number of independent collaborating components.

Developing a single application as a suite of small services each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies

## **Microservice Architecture?**

This is how a monolith would look like. One application for everything.



This is how the same application would look like when developed using Microservices Architecture. 

Microservice Architectures involve a number of small, well designed, components interacting with messages.

http://www.springboottutorial.com/images/Microservices-Chain-Example.png

## **Benefits of Microservice Architecture?**

* **Decomposing -** So, instead of having one large application, we decompose it into separate, different, mini-applications (services).

Each service handles a specific business domain (logging, auth, orders, customers) and provides the implementation for user interface, business logic, and connection to database.

* **Single-function -** Each and every service has a specific function, or responsibility. And yes, a service can do many tasks, but all of them are nevertheless relevant to this single function.
* **Independent -** Independent means services don’t know about each other implementation. They can get tested, deployed, and maintained independently.

It might be the case where services are implemented using different language stacks, and communicate with different databases.

But that doesn’t mean they don’t work together. They do, in order to complete their required operation.

* **Small Teams -** We split the work up and team across the services. Each team focuses on a specific service, they don’t need to know about internal workings of other teams.

Those teams are can work efficiently, communicate easily, and each service can be deployed rapidly as soon as it’s ready.

* **Entire Lifecycle -** The team is responsible for the entire lifecycle of the service; from coding, testing, staging, deploying, debugging, maintaining.

In traditional application, we may have a team for coding, and another one for deployment. In microservices, that’s not the case.

* **Minimizing Communication -** Minimizing communication doesn’t mean that the team members should ignore each other. It means the only essential cross-team communication should be through the interface that each service provides.

They all need to agree on the external interface, so that communication between services is clearly defined.

* **The scope and risk of change -** Services should be changed without breaking other services. And so long as we don’t change the external interface there will be no problem for other services.

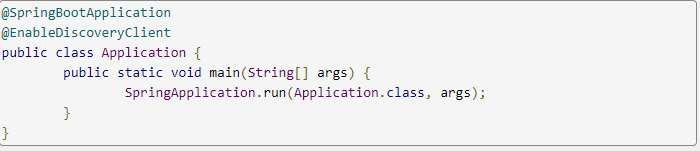
As a result of changes, the versions of services are updating individually, and there is no relationship between them.

* **Scaling with Cloud:** can be scaled microservices with Cloud
* **New Technology & Process Adaption** becomes easier. You can try new technologies with the newer microservices that we create.
* Faster Release Cycles

# **What is Spring Cloud?**

* Spring Cloud is a framework for building robust cloud applications. It facilitates the development of applications by providing solutions to many of the common problems faced when building distributed system.
* Spring Cloud- which builds on top of Spring Boot\*, provides a set of tools to quickly build cloud-based microservices. Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems.
* Spring Cloud takes a very declarative approach, and often you get a lot of features with just a class path change and/or an annotation.

Example application that is a discovery client:



## **Spring Cloud Architecture (Modules)**

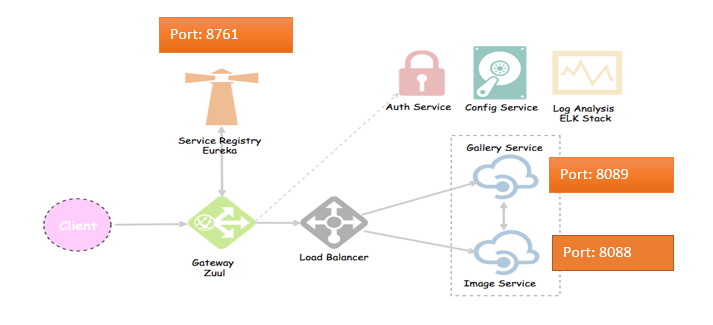


Fig. Cloud Basic Architecture

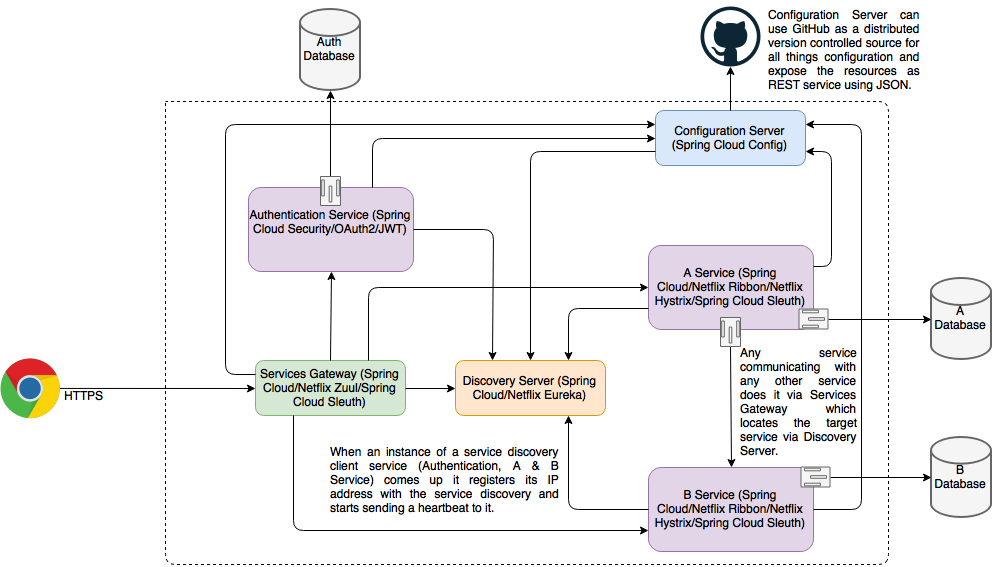


Fig. Cloud Detailed Architecture

### **Spring Cloud Tools**

Spring Cloud provides the following tools to build cloud based microservices:

* + Configuration Management: Spring Config Server (with GIT repository)
  + Service Registration and Discovery - Eureka Server (owned by Netflix)
  + Gateway to client applications- Zuul (owned by Netflix)
  + Rest Client – Feign Client and Rest Template
  + Client-Side Load Balancer – Netflix Ribbon
  + Fault Tolerance – Hystrix
  + Distributed log tracing – Zipkin and Sleuth
  + [Managing, Searching, and Visualizing Logs — ELK (Elasticsearch, Logstash, Kibana)](https://medium.com/oneclicklabs-io/streaming-spring-boot-application-logs-to-elk-stack-part-1-a68bd7cccaeb)

#### **Config Server with Git Integration**

Now-a-days we often come across the term [Microservices](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/) while discussing any new java based server side development (mainly service API). Microservices approach now has become an industry standard for any new API development and almost all the organizations are promoting it.

In this tutorial, we will discuss and do a demo on a specific Microservice feature called **Config Server**. 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.

Table of Contents

[Why to Use Config Server](https://howtodoinjava.com/spring-cloud/spring-cloud-config-server-git/#why-config-server)

[Tech Stack Used in Example](https://howtodoinjava.com/spring-cloud/spring-cloud-config-server-git/#tech-stack)

[Config Server – Server Side Configuration](https://howtodoinjava.com/spring-cloud/spring-cloud-config-server-git/#server-config)

[Config Server - Client Side Configuration](https://howtodoinjava.com/spring-cloud/spring-cloud-config-server-git/#client-config)

[Demo](https://howtodoinjava.com/spring-cloud/spring-cloud-config-server-git/#demo)

[Things to check if facing any error](https://howtodoinjava.com/spring-cloud/spring-cloud-config-server-git/#debugging-tips)

##### 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.

As an example, let’s say one service is dependent on another service (invoked for certain business scenario) and if that dependent service URL got changed to some other service, and then usually we need to build and deploy our service with the updated location is required. Now if we go by 12 factor app approach and if we read those config from external service deployed as different process, then we just need to refresh that config server.

So, the idea is very clear and effective. Let’s now see how we can create config server.

##### Tech Stack Used in Example

We will be using spring-boot based spring-cloud API that is easily available and very popular. It is called Config Server in spring framework nomenclature. Also, we will use git configuration to host the properties file.

So finally, our technology stack for this demo will be:

1. [Java 1.8](https://howtodoinjava.com/java-8-tutorial/)
2. Eclipse IDE
3. Spring cloud
4. [Spring boot](https://projects.spring.io/spring-boot/)
5. Spring Rest
6. GitHub as resource repository
7. [Maven](https://howtodoinjava.com/maven/)
8. REST client

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

Let’s first build the config server part with given steps:

###### Generate the project structure

Start with [spring boot initializer portal](https://start.spring.io/) which is a great starting point for creating any spring boot based application. Here we will choose only **Config server** starter pom. The screen shot is something like this. With this configuration, once we generate the project, one zip file will be downloaded, which we will simply import in eclipse after unzipping.

Generate Server Project with Config Server Starter POM

###### Import the project in Eclipse

Once you have the zip file from the spring initializer portal, we need to unzip it to a directory we choose to and import it to eclipse as maven project

###### Build in eclipse

Next step will be to run mvn clean install from either command prompt or from eclipse whatever you are comfortable with. In this step, all necessary dependencies will be downloaded from maven repo. Make sure you are trying it from any network where no download restriction is present. Successful build in this step is very much required to proceed to the next steps.

###### 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" \t "_blank) annotation before the class and build the project once again. With this annotation, this artifact will act like a spring config server.

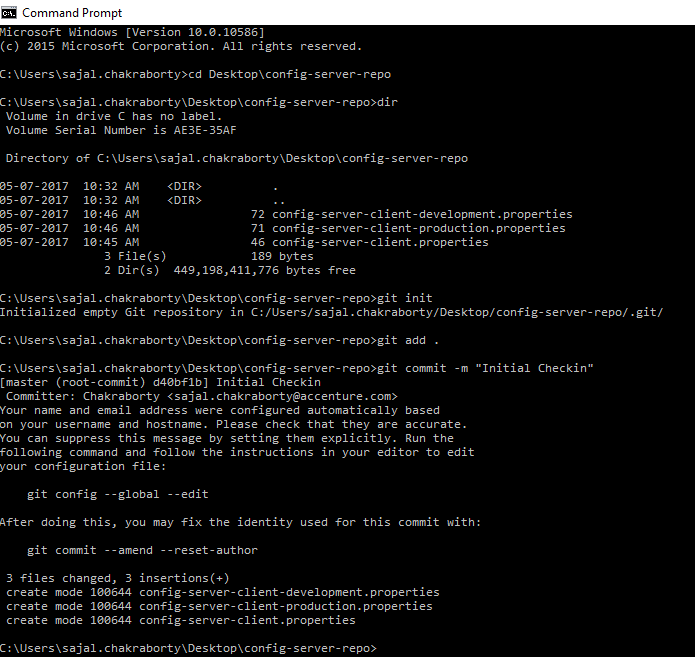
After adding this annotation the class will look like below – the class name can be different depending on the Project name you gave while generating, also you can manually change the class name to a name you like it.

|  |
| --- |
| package com.howtodoinjava.example.springconfigserver;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.config.server.EnableConfigServer;    @EnableConfigServer  @SpringBootApplication  public class SpringConfigServerApplication  {      public static void main(String[] args)      {          SpringApplication.run(SpringConfigServerApplication.class, args);      }  } |

###### Create the Git repository

Next very important step is to create a local git repository. It can easily be converted to a remote repository later by configuring it’s URL in the properties file. We will place the external property file [configuration], which will be used by the Config server Microservice to provide the external configuration of properties. We need to follow the below steps to create a local git repository and check in sample properties files.

* 1. Make sure you have git shell installed in your machine and you can run git bash from command prompt. To verify it open command prompt and type git, if it recognize the command then you probably have the git prompt installed, if not please follow git website, download and install as per the instruction.
  2. Now Create a directory **config-server-repo** in your Desktop.
  3. Then create a file config-server-client.properties file in the config-server-repo directory and add the message there msg = Hello world - this is from config server.
  4. Then create another file config-server-client-development.properties file in the config-server-repo directory and add the message there msg = Hello world - this is from config server – Development environment.
  5. Then create another file config-server-client-production.properties file in the config-server-repo directory and add the message there msg = Hello world - this is from config server – Production environment.
  6. Here we are maintaining same property name for different environment, as we generally maintain properties for different environments like urls, credentials, database details etc. Here the most important point is that we need to append hyphen (-) with the environment name in each property so that config server understands it. Also, we need to name the properties file with the config client service name that we will create after this.
  7. Now open command prompt from **config-server-repo** directory and run command git init to make that directory as git repository.
  8. Now run git add . to add everything to this repo.
  9. Then finally we need to commit the properties file by running command git commit –m "initial checkin". This should check in all the files in the git repository. Here is the command prompt screen shot for the same.

Property Check-in in Git

###### Point the git repo from Config Server

Create one file called bootstrap.properties in the src\main\resources directory of spring-config-sever project and add below lines.

|  |
| --- |
| #Server port  server.port = 8888    #Git repo location.  spring.cloud.config.server.git.uri=${USERPROFILE}\\Desktop\\config-server-repo    #Disable security of the Management endpoint  management.security.enabled=false |

Now let’s understand those properties.

* 1. server.port defines the port on which the embedded server will start.
  2. spring.cloud.config.server.git.uri will bind the git location to look for the configuration. Here we are using local git repo but can be switched to remote got location by just changing this location.
  3. management.security.enabled=false will disable the spring security on the management enpoints like /env, /refresh etc. This is for development settings, in production security should be enabled.

So, this step will point to a git location and server port.

This is very much we need to do in the config server side, not do a final mvn clean install command on this project so that everything gets compiled properly and packaged also in the target folder as well as in local maven repository. We will start the config server service once we have the client part ready and we will finally test the feature.

###### Verify Server Side Configuration

The command to run the service in embedded mode is java -jar target\spring-config-server-0.0.1-SNAPSHOT.jar however we will revisit this in the testing part.

To check if the Config-server can recognize the properties, first run the config server microservice from command prompt by using the given command from command prompt of the project code base location.

|  |
| --- |
| java -jar target\spring-config-server-0.0.1-SNAPSHOT.jar |

Now open browser and check below Urls, it will return the JSON output and in propertySources section we can see all the properties we have added in the properties. This ensures that config-server is running successfully, it has recognized the git location and it is serving configuration for different environments.

* 1. http://localhost:8888/config-server-client/development
  2. http://localhost:8888/config-server-client/production

Also, check if any runtime change in the property file is reflected by the server without restart – 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**.

To do the git check in, after doing the change and save the file by any text editor, run the command git add . and git commit -m "test"from **config-server-repo** directory in the desktop.

##### Config Server – Client Side Configuration

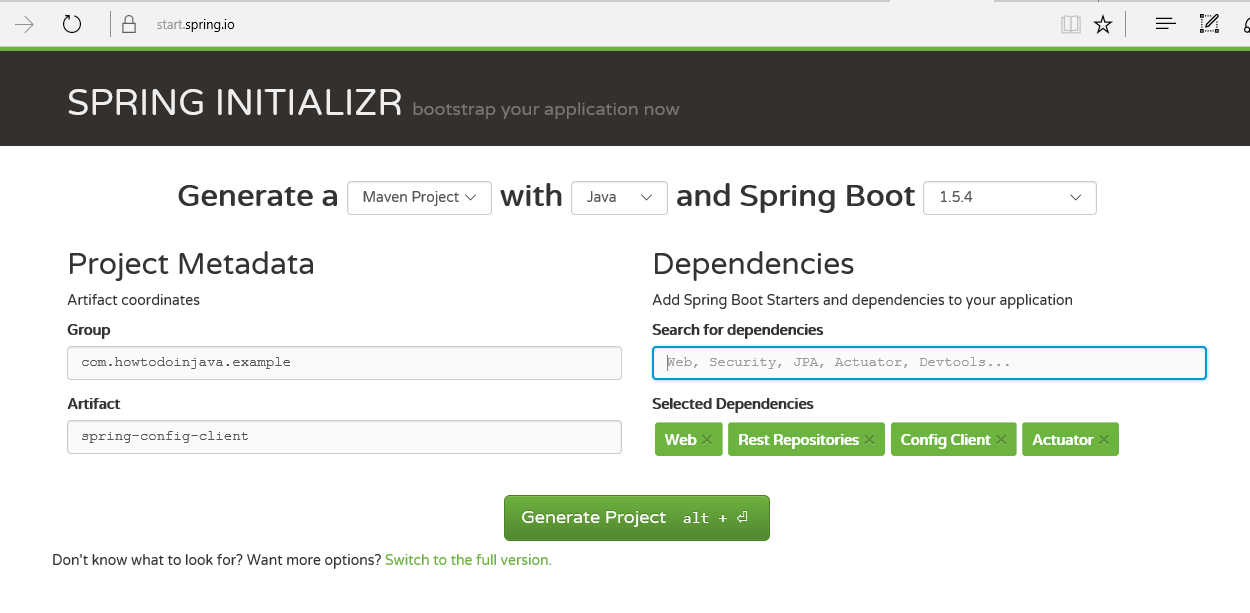
Now we will proceed to the client side implementation where we will use those properties from a separate microservice which is our final goal – to externalize the configuration to different service.

###### Create Maven Project

Go to <https://start.spring.io/> web portal and generate client project with the below selected artifacts:

* 1. Actuator
  2. Config Client
  3. Web
  4. Rest Repositories

The screen will look like below before generation, once we click on generate, we will get the .zip file download option. Like Spring-Config-Server, unzip the file in some directory and import in eclipse.

Generate Client Project with Listed Dependencies

###### Create REST Resource

Add one [RestController](https://howtodoinjava.com/spring/spring-boot/spring-boot-tutorial-with-hello-world-example/) to view the Server side property values in the response. To do that open the [@SpringBootApplication](https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html) class file that has been generated, and add the below small class in the end of that file. This is very simple and straight forward, we are just exposing one method at /message URL where we will just return the property value of msg that will be supplied by the config server microservice, which is configured to a local git repository (which will be migrated to a remote git repository in production!).

|  |
| --- |
| package com.howtodoinjava.example.springconfigclient;    import org.springframework.beans.factory.annotation.Value;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.context.config.annotation.RefreshScope;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;    @SpringBootApplication  public class SpringConfigClientApplication {        public static void main(String[] args) {          SpringApplication.run(SpringConfigClientApplication.class, args);      }  }    @RefreshScope  @RestController  class MessageRestController {        @Value("${msg:Hello world - Config Server is not working..pelase check}")      private String msg;        @RequestMapping("/msg")      String getMsg() {          return this.msg;      }  } |

###### Bind with the Config Server

Create one file called bootstrap.properties in the src\main\resources directory and add the below properties to connect with the config server along with some required configuration.

|  |
| --- |
| spring.application.name=config-server-client    #Active Profile - will relate to development properties file in the server.  #If this property is absent then default profile will be activated which is  #the property file without any environment name at the end.    spring.profiles.active=development    # this is the default    spring.cloud.config.uri=http://localhost:8888    management.security.enabled=false |

Let’s understand the properties now.

* 1. spring.application.name is just the application name of the microservice that would be deployed.
  2. spring.cloud.config.uri is the property to mention the config server url. Point to note that our config server is running on port 8888; verify it by opening the application.properties file of the spring config server code base and check the server.port=8888.
  3. management.security.enabled=false will disable the spring security on the management endpoints like /env, /refresh etc. This is for development settings, in production security should be enabled.

###### Verify Client Config

This is very much we need to do in the config client side, not do a final mvn clean install command on this project so that everything gets compiled properly and packaged also in the target folder as well as in local maven repository. We will start the config client service along with the server side and we will finally test the feature.

##### Demo

Lets test the config server application.

###### Build and Run Config Server Project

Open command prompt from **spring-config-server** folder and run mvn clean install command. Once build is completed run the application from that command prompt itself by java -jar command like java -jar target\spring-config-server-0.0.1-SNAPSHOT.jar.

This will start the config server service in 8888 port in localhost.

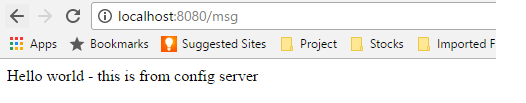
###### Build and Run Config Client Project

Similarly, Open command prompt from **spring-config-client** folder and run mvn clean install command. Once build is completed run the application from that command prompt itself by java -jar command like java -jar target\spring-config-client-0.0.1-SNAPSHOT.jar.

This will start the Config Client service in 8080 port of localhost.

###### Test REST Endpoint

Now in the browser open the /msg rest endpoint by browsing the url http://localhost:8080/msg. It should return Hello world - this is from config server which is mentioned in the config-server-client-development.properties file.

Test REST End Point

###### Test Property Change

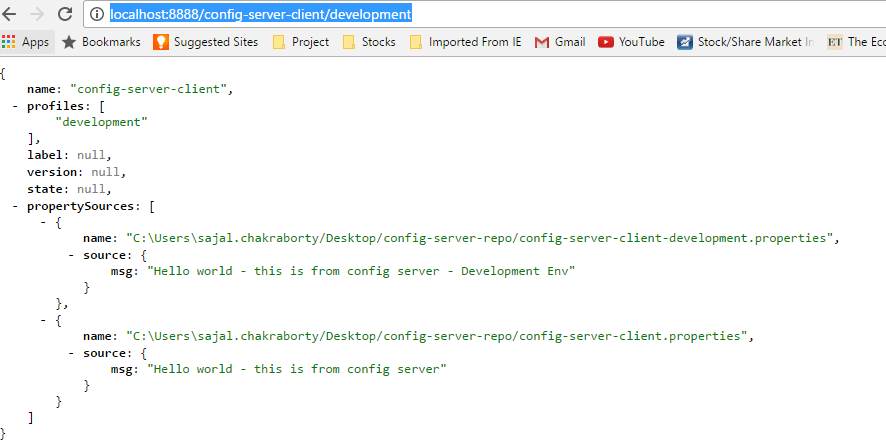
Now we will do a property change and test if this can be reflected in the config client service without restarting any of the Microservices.  
Do some change, in the value of the msg property in the config-server-client-development.properties and check-in in the local git, then hit the http://localhost:8080/msg again in the browser, You will the old value only.

To reflect the new value, we need to **refresh the configuration** by hitting http://localhost:8080/refresh endpoint using POST method from any of the REST client.

Once you have successfully refreshed the config client service, the new value should be reflected in the service response. This is because [@RefreshScope](https://cloud.spring.io/spring-cloud-static/docs/1.0.x/spring-cloud.html#_refresh_scope) annotation the Rest Controller that we have exposed.

##### Things to check if facing any error

* Property file name and the Client module service name spring.application.name=config-server-client should be exactly same, otherwise, properties will not be detected. Actually, Config Server exposes the properties in an end point of property file name, if you browse URL http://localhost:8888/config-server-client/development it will return all the dev environment values.

All Dev Properties View

* Make sure you have checked-in the properties files in the git repo by using git init/add/commit commands as described above.
* Make sure you have refreshed the client service environment by invoking POST method of http://localhost:8080/refresh by any REST client. Otherwise changed values will not be reflected in the client service.
* Make sure at the time of starting the config client service, config server service is running already, otherwise it might take some time to register, which might create confusion while testing.

#### **Eureka Server (Netflix) - Service Registration and Discovery**

Eureka Server or service will register every microservice and then the client microservice will look up the Eureka server to get a dependent microservice to get the job done without knowing the IP address of dependent microservice.

Eureka Server is owned by Netflix and in this, Spring Cloud offers a declarative way to register and invoke services by Java annotation.

It’s duty to give names to each microservice. Why?

* No need to hardcode the IP addresses of microservices.
* What if services use dynamic IP addresses; when autoscaling.

So, every service register itself with Eureka, and pings Eureka server to notify that it’s alive.

If Eureka server didn’t receive any notification from a service. This service is **unregistered from the Eureka server automatically.**

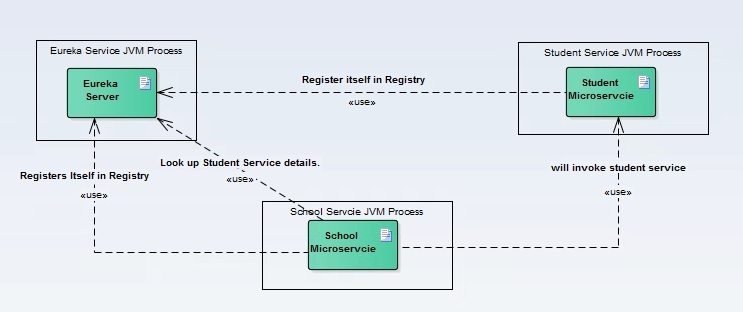
The steps are fairly simple. 1, 2, 3, … and we’re done!

##### 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.

Here is the interaction diagram between above listed three services.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/Discovery_interratction-Diagram.jpg)Component Interaction with each other

Tech Stack and Runtime

* Java 1.8
* Eclipse IDE
* Spring cloud
* Spring boot
* Spring Rest
* Maven

Table of Contents

[What is Netflix Eureka Server and Clients?](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#what-is-netflix-eureka)

[Eureka Service Registry Server](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#eureka-server)

[Eureka Client - Student Service](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#eureka-client-student-service)

[Eureka Client - School Service](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#eureka-client-school-service)

[Demo of Service Discovery and Calling](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#demo)

[Things to check if facing any error](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#how-to-debug)

[Summary](https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/#summary)

##### What is Netflix Eureka Server and Clients?

As we know these days, there is a lot of momentum around Microservices. 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. This task can be hugely complex – depending on number of services and their dynamic nature. If whole infrastructure is distributed and there is some replication as well, then maintaining this service addresses becomes harder.

To solve this, in the distributed computing are 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.

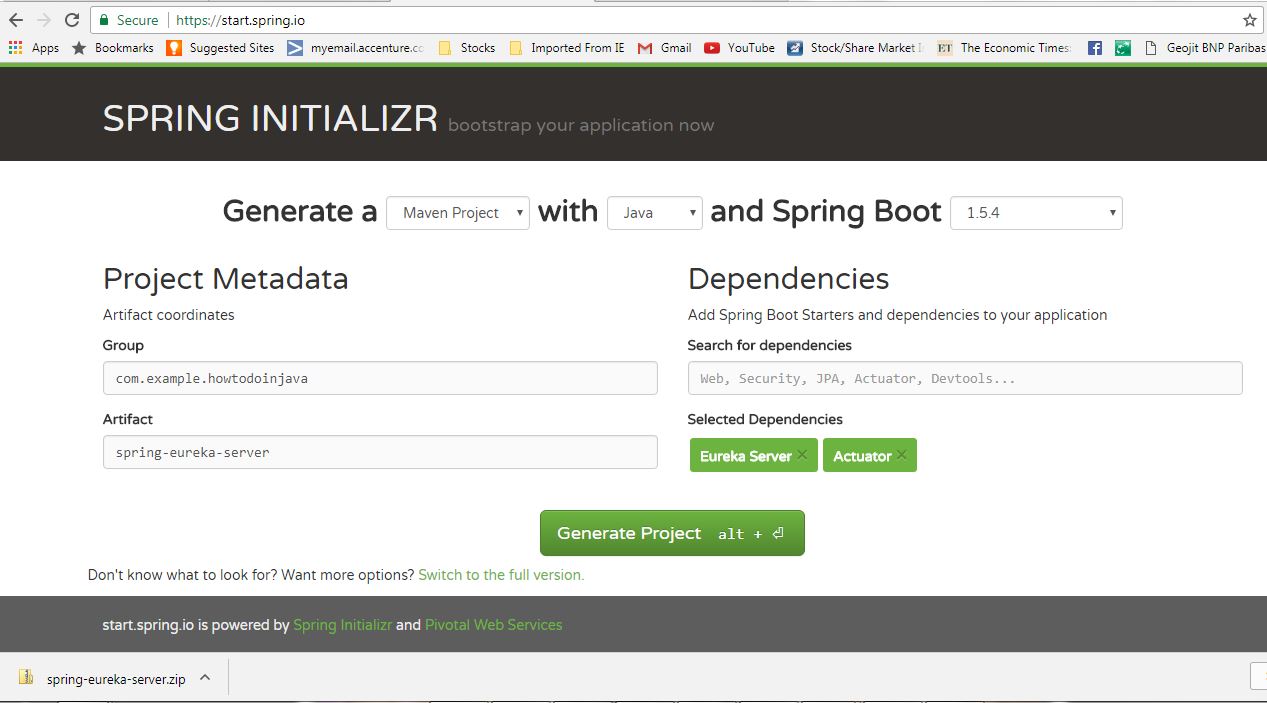
Think of it as a lookup service where microservices (clients) can register themselves and discover other registered microservices. When a client microservice registers with Eureka it provides metadata such as host, port, and health indicator thus allowing for other microservices to discover it. The discovery 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, which is a next to impossible task if the scale up/down is very frequent, on demand and we use virtual host to host the services specially in the cloud environment.

##### Eureka Service Registry Server

Follow these steps to create and run Eureka 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. Give other maven GAV coordinates and download the project.

[](https://cdn2.howtodoinjava.com/wp-content/uploads/2017/07/server_projec-generation.jpg)Eureka Server Service Project Generation

Unzip and import the project into Eclipse as existing maven project. In this step, all necessary dependencies will be downloaded from maven repository.

Now open SpringEurekaServerApplication class that spring already has generated in the downloaded project and add the [@EnableEurekaServer](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-eureka-server/src/main/java/org/springframework/cloud/netflix/eureka/server/EnableEurekaServer.java)annotation on the class.

|  |
| --- |
| package com.example.howtodoinjava.springeurekaserver;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;    @EnableEurekaServer  @SpringBootApplication  public class SpringEurekaServerApplication {        public static void main(String[] args) {          SpringApplication.run(SpringEurekaServerApplication.class, args);      }  } |

Build the project once again. With this annotation, this artifact will act like microservice registry and discovery server.

Server Configuration

Create one file called application.yml in the src\main\resources directory. Add these properties –

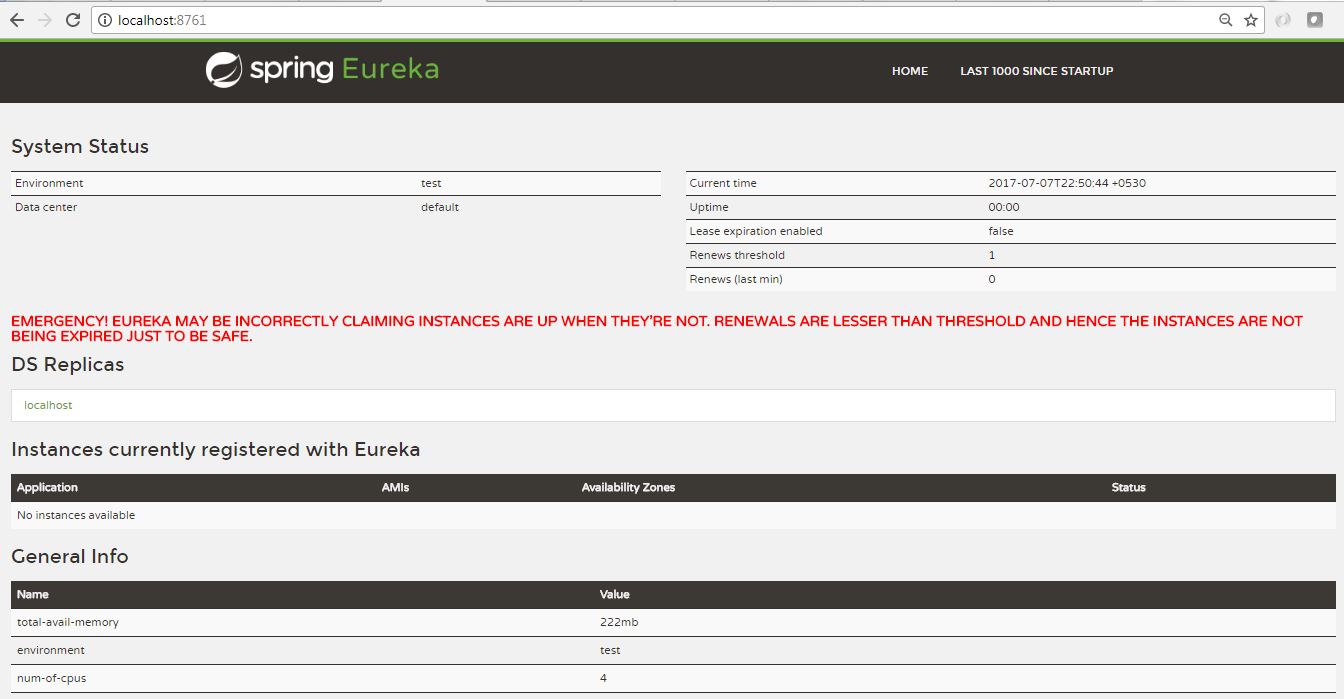
|  |
| --- |
| server:    port: ${PORT:8761} # Indicate the default PORT where this service will be started    eureka:    client:      registerWithEureka: false   #telling the server not to register himself in the service registry      fetchRegistry: false    server:      waitTimeInMsWhenSyncEmpty: 0    #wait time for subsequent sync |

Create another file called bootstrap.yml in the src\main\resources directory. Add these properties –

|  |
| --- |
| spring:    application:      name: eureka    cloud:      config:        uri: ${CONFIG\_SERVER\_URL:http://localhost:8888} |

Test Eureka Server

Start the application as spring boot application. Open browser and go to http://localhost:8761/, you should see the eureka server home page which looks like below.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/eureka_console_without_anyClient.jpg)Eureka Console Without Any Client

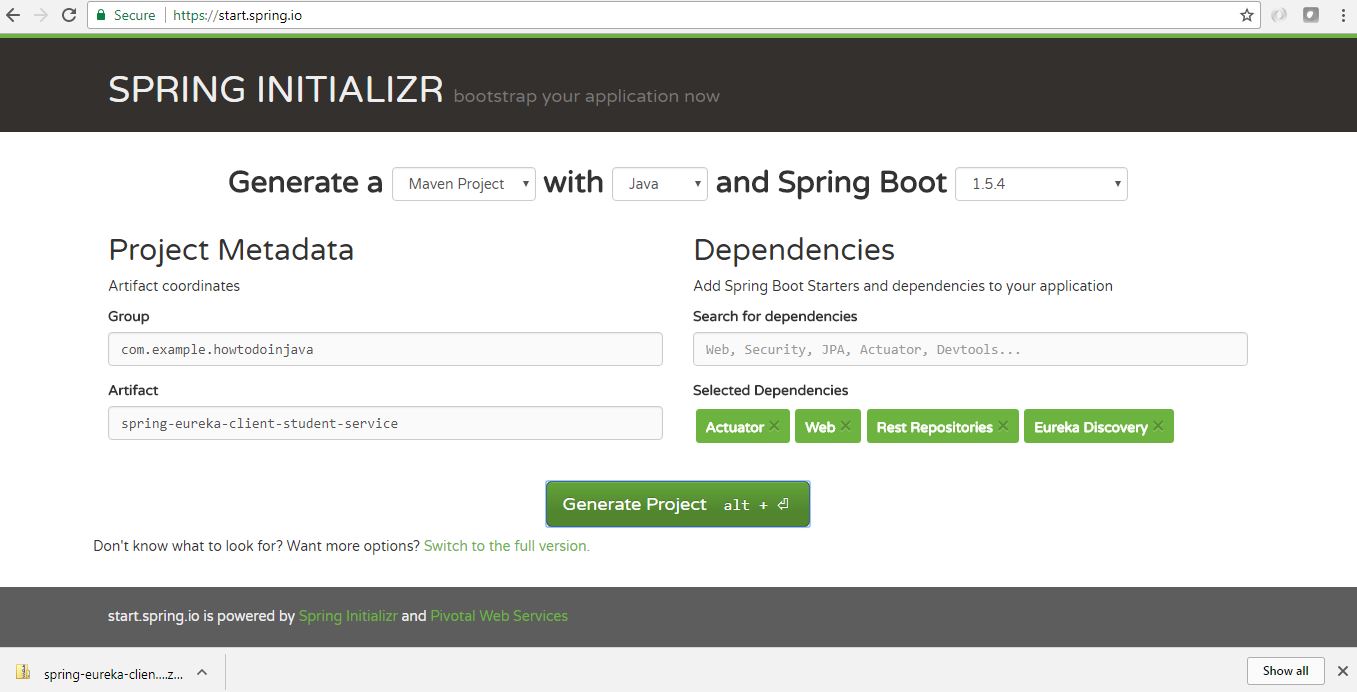
Please note that at this point no service is registered here which is expected and once we will spin up the client services, this server will automatically updated with the details of the client services.

##### 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. Give other maven GAV coordinates and download the project.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/client_projec-generation_Student.jpg)Client Project Generation –  
Student Microservice

Unzip and import the project into Eclipse as existing maven project.

Now add the [@EnableEurekaClient](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EnableEurekaClient.java) 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.

|  |
| --- |
| package com.example.howtodoinjava.springeurekaclientstudentservice;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;    @SpringBootApplication  @EnableEurekaClient  public class SpringEurekaClientStudentServiceApplication {        public static void main(String[] args) {          SpringApplication.run(SpringEurekaClientStudentServiceApplication.class, args);      }  } |

Client Configuration

Create one file called application.yml in the src\main\resources directory and add below lines.

|  |
| --- |
| server:    port: 8098    #default port where the service will be started    eureka:         #tells about the Eureka server details and its refresh time    instance:      leaseRenewalIntervalInSeconds: 1      leaseExpirationDurationInSeconds: 2    client:      serviceUrl:        defaultZone: http://127.0.0.1:8761/eureka/      healthcheck:        enabled: true      lease:        duration: 5    spring:    application:      name: student-service   #current service name to be used by the eureka server    management:    security:      enabled: false  #disable the spring security on the management endpoints like /env, /refresh etc.    logging:    level:      com.example.howtodoinjava: DEBUG |

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.

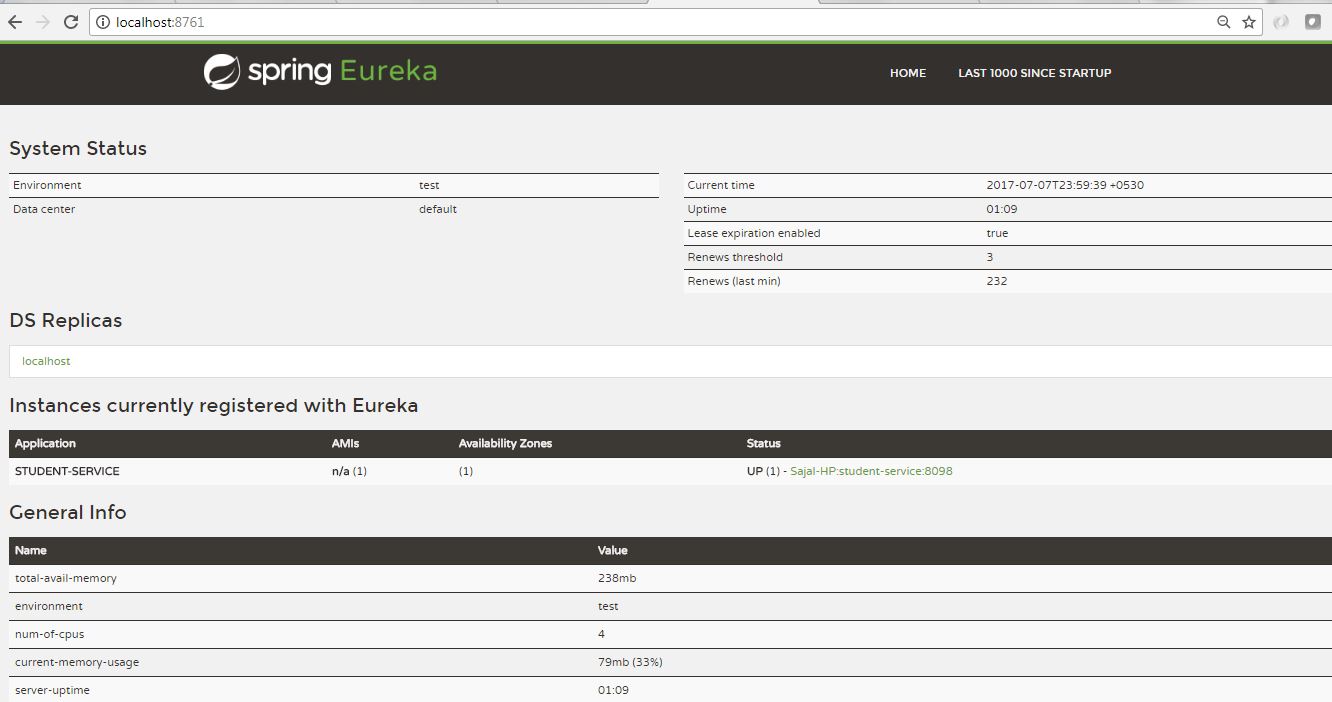
|  |
| --- |
| package com.example.howtodoinjava.springeurekaclientstudentservice.controller;    import java.util.ArrayList;  import java.util.HashMap;  import java.util.List;  import java.util.Map;    import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;    import com.example.howtodoinjava.springeurekaclientstudentservice.domain.Student;    @RestController  public class StudentServiceController {        private static Map<String, List<Student>> schooDB = new HashMap<String, List<Student>>();        static {          schooDB = new HashMap<String, List<Student>>();            List<Student> lst = new ArrayList<Student>();          Student std = new Student("Sajal", "Class IV");          lst.add(std);          std = new Student("Lokesh", "Class V");          lst.add(std);            schooDB.put("abcschool", lst);            lst = new ArrayList<Student>();          std = new Student("Kajal", "Class III");          lst.add(std);          std = new Student("Sukesh", "Class VI");          lst.add(std);            schooDB.put("xyzschool", lst);        }        @RequestMapping(value = "/getStudentDetailsForSchool/{schoolname}", method = RequestMethod.GET)      public List<Student> getStudents(@PathVariable String schoolname) {          System.out.println("Getting Student details for " + schoolname);            List<Student> studentList = schooDB.get(schoolname);          if (studentList == null) {              studentList = new ArrayList<Student>();              Student std = new Student("Not Found", "N/A");              studentList.add(std);          }          return studentList;      }  } |

Student class is a simple POJO.

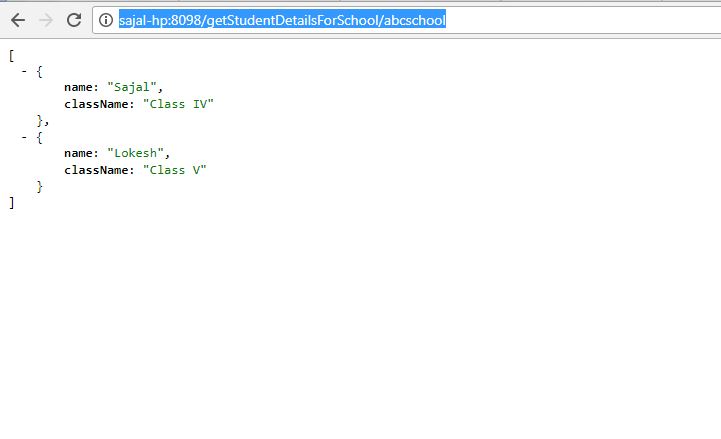
|  |
| --- |
| public class Student  {      private String name;      private String className;        public Student(String name, String className) {          super();          this.name = name;          this.className = className;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        public String getClassName() {          return className;      }        public void setClassName(String className) {          this.className = className;      }  } |

Test Eureka Client

Start this project as spring boot application. Now verify that this service has been registered in Eureka server automatically. Go to Eureka service console and refresh the page. Now if everything goes well, we will see one entry for **student-service** in the eureka service console. This indicates that both Eureka server and client are aware each other.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/eureka_console_with_Student_1service_registered.jpg)Eureka console with Student service registered

We will now verify that the /getStudentDetailsForSchool/{schoolname} endpoint is up and running. Go to browser and go to http://localhost:8098/getStudentDetailsForSchool/abcschool, it will give the Student details for a particular school abcschool.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/Student-Service-Responding.jpg)Student Service response

##### 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.

Follow exact steps for creating student service, to create and run Eureka client running school service as well.

Create Eureka Client Project

Create a Spring boot project from initializer portal with four dependencies i.e. Actuator, Web, Rest Repositories, Eureka Discovery. Give other maven GAV coordinates and download the project.

Unzip and import the project into Eclipse as existing maven project.

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.

|  |
| --- |
| package com.example.howtodoinjava.springeurekaclientschoolservice;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.EnableEurekaClient;    @SpringBootApplication  @EnableEurekaClient  public class SpringEurekaClientSchoolServiceApplication {        public static void main(String[] args) {          SpringApplication.run(SpringEurekaClientSchoolServiceApplication.class, args);      }  } |

Client Configuration

Create one file called application.yml in the src\main\resources directory and add below lines. These configurations are very similar to student service except port number and service name.

|  |
| --- |
| server:    port: 9098    #port number    eureka:    instance:      leaseRenewalIntervalInSeconds: 1      leaseExpirationDurationInSeconds: 2    client:      serviceUrl:        defaultZone: http://127.0.0.1:8761/eureka/      healthcheck:        enabled: true      lease:        duration: 5    spring:    application:      name: school-service    #service name    logging:    level:      com.example.howtodoinjava: DEBUG |

Add REST API which consume student service’s REST API

Now add one RestController and expose one rest endpoint for getting school details. This endpoint will use the service discovery style URL using the application name, instead full URL with host:port.

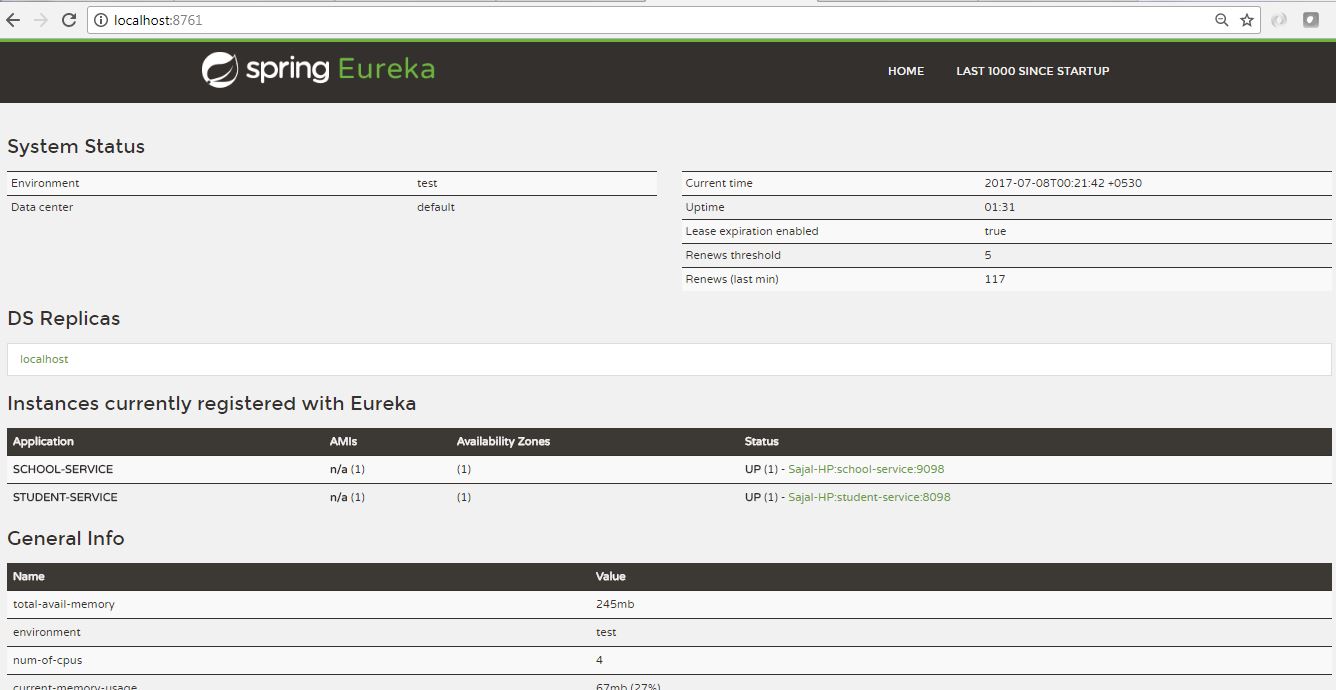
|  |
| --- |
| package com.example.howtodoinjava.springeurekaclientschoolservice.controller;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.cloud.client.loadbalancer.LoadBalanced;  import org.springframework.context.annotation.Bean;  import org.springframework.core.ParameterizedTypeReference;  import org.springframework.http.HttpMethod;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;    @RestController  public class SchoolServiceController {      @Autowired      RestTemplate restTemplate;        @RequestMapping(value = "/getSchoolDetails/{schoolname}", method = RequestMethod.GET)      public String getStudents(@PathVariable String schoolname)      {          System.out.println("Getting School details for " + schoolname);            String response = restTemplate.exchange("<http://student-service/getStudentDetailsForSchool/>{schoolname}",                                  HttpMethod.GET, null, new ParameterizedTypeReference<String>() {}, schoolname).getBody();            System.out.println("Response Received as " + response);            return "School Name -  " + schoolname + " \n Student Details " + response;      }        @Bean      @LoadBalanced      public RestTemplate restTemplate() {          return new RestTemplate();      }  } |

This way we can get rid of specific service configuration and we can give the service look up responsibility to eureka server and rest template provided here. We can also apply load balancing (see @LoadBalanced annotation) here if the multiple instances are running for the same service.

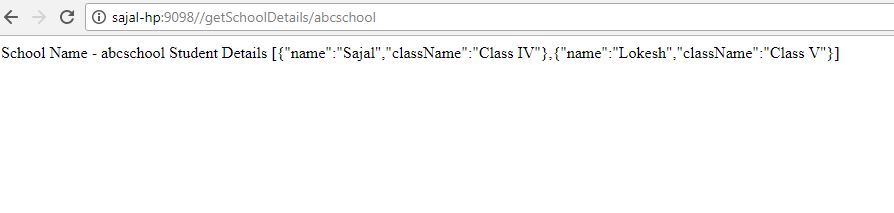
The URL we have used is http://student-service/getStudentDetailsForSchool/{schoolname}. Clearly we are using only service name student-service in the place of host:port. This will be handled internally by spring framework, eureka server and rest template together.

##### Demo of Service Discovery and Calling

Now start the school service as well. All three services are started. Check the eureka server console. Bothe student and school services must be registered there.

[](https://cdn2.howtodoinjava.com/wp-content/uploads/2017/07/eureka_console_with_both_servcies.jpg)Eureka console with both services registered

Go to browser and go to http://localhost:9098//getSchoolDetails/abcschool, it will give the school details for a particular school abcschool details. We have **invoked student service internally**. The response will look like in the browser:

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/School-Service-Responding.jpg)School Service Response

Things to check if facing any error

1. Annotations @EnableEurekaServer and @EnableEurekaClient are the heart of the application ecosystem. Without those two things will not work at all.
2. Make sure at the time of starting the config client service, eureka server service is running already, otherwise it might take some time to register, which might create confusion while testing.

##### Summary

We saw how easily one can deploy service registry and discovery server as well as clients efficiently. Spring framework is maintaining lots of things internally. Here we are just using couple of annotations and very minimal configuration to achieve the whole things quickly.

That’s all about creating spring could eureka server and service registration for microservices. Please add comments if you have any difficulty executing this article. We will be happy to look into the problem.

#### **Zuul (Netflix) - Gateway to client applications**

A gateway is a single-entry point into the system, used to handle requests by routing them to the corresponding microservice. It can also be used for authentication, monitoring, and more.

***Why it is required*:** And as we spin more instances of services, each with a different port numbers, So, now the question is: *How can we call the services from the browser and distribute the requests among their instances running at different ports?*

Well, a common solution is to use a Gateway.

**What’s Zuul?**

It’s a proxy, gateway, an intermediate layer between the users and your services.

Eureka server solved the problem of giving names to services instead of hardcoding their IP addresses.

But, still, we may have more than one service (instances) running on different ports. So, Zuul …

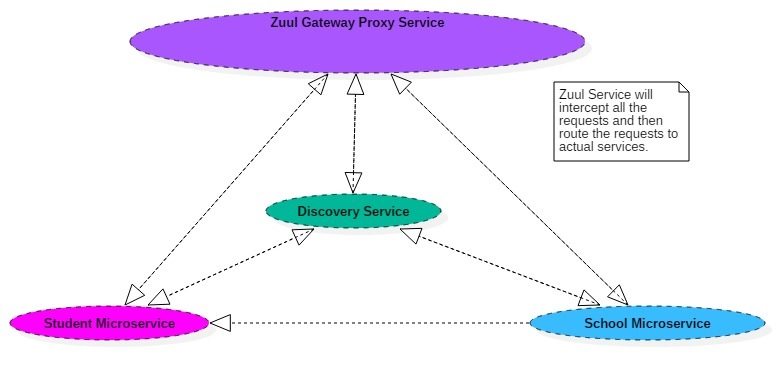
* Maps between a prefix path, say/gallery/\*\* and a service gallery-service. It uses Eureka server to route the requested service.
* It loads balances (using Ribbon) between instances of a service running on different ports.
* What else? We can filter requests, add authentication, etc.
* **It’s worth mentioning that Zuul acts as a Eureka client. So, we give it a name, port, and link to Eureka server (same as we did with image service).**

NOTE: When calling any microservice from the browser, we can’t call it by its name as we do when we call one microservice from another microservice — This is used internally between services.

**Where Zuul fits in microservices ecosystem?**

A common problem, when building microservices, is to provide a unique gateway to the client applications of your system. And to solve this problem, Netflix (a major adopter of microservices) created and open-sourced its **Zuul proxy server** and later Spring under Pivotal has adapted this in its **spring cloud stack** and enabled us to use zuul easily and effectively with just few simple steps.

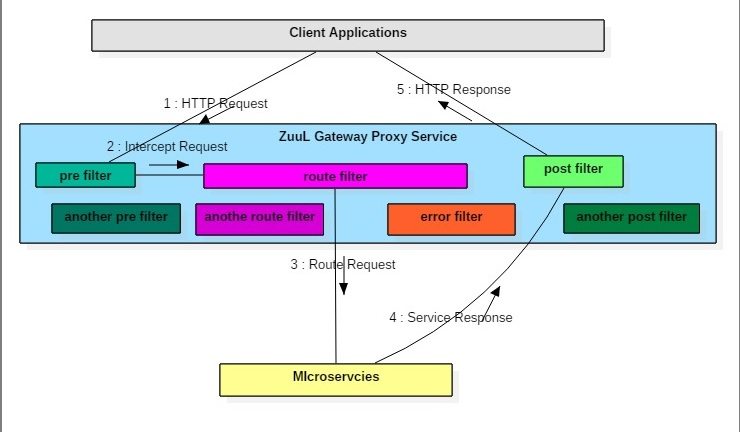
Zuul is an edge service that proxies requests to multiple backing services. It provides a unified “front door” to your ecosystem, which allows any browser, mobile app or other user interface to consume services from multiple hosts. You can integrate Zuul with other Netflix stack components like Hystrix for fault tolerance and Eureka for service discovery or use it to manage routing rules, filters and load balancing across your system. Most importantly all of those components are well adapted by spring framework through spring boot/cloud approach.

A microservice ecosystem with Zuul gateway at the front

###### 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.

Request Processing flow inside Zuul with different filters

###### **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
  + We can do **microservices insights and monitoring** of all the traffic that are going in to the ecosystem by enabling some logging to get meaningful data and statistics at the edge in order to give us an accurate view of production.
  + **Dynamic Routing** can route requests to different backend clusters as needed.
  + We can do **runtime stress testing** by gradually increasing the traffic to a new cluster in order to gauge performance in many scenarios e.g. cluster has new H/W and network setup or that has new version of production code deployed.
  + We can do **dynamic load shedding** i.e. allocating capacity for each type of request and dropping requests that go over the limit.
  + We can apply **static response handling** i.e. building some responses directly at the edge instead of forwarding them to an internal cluster for processing.

###### Tech Stack and Runtime

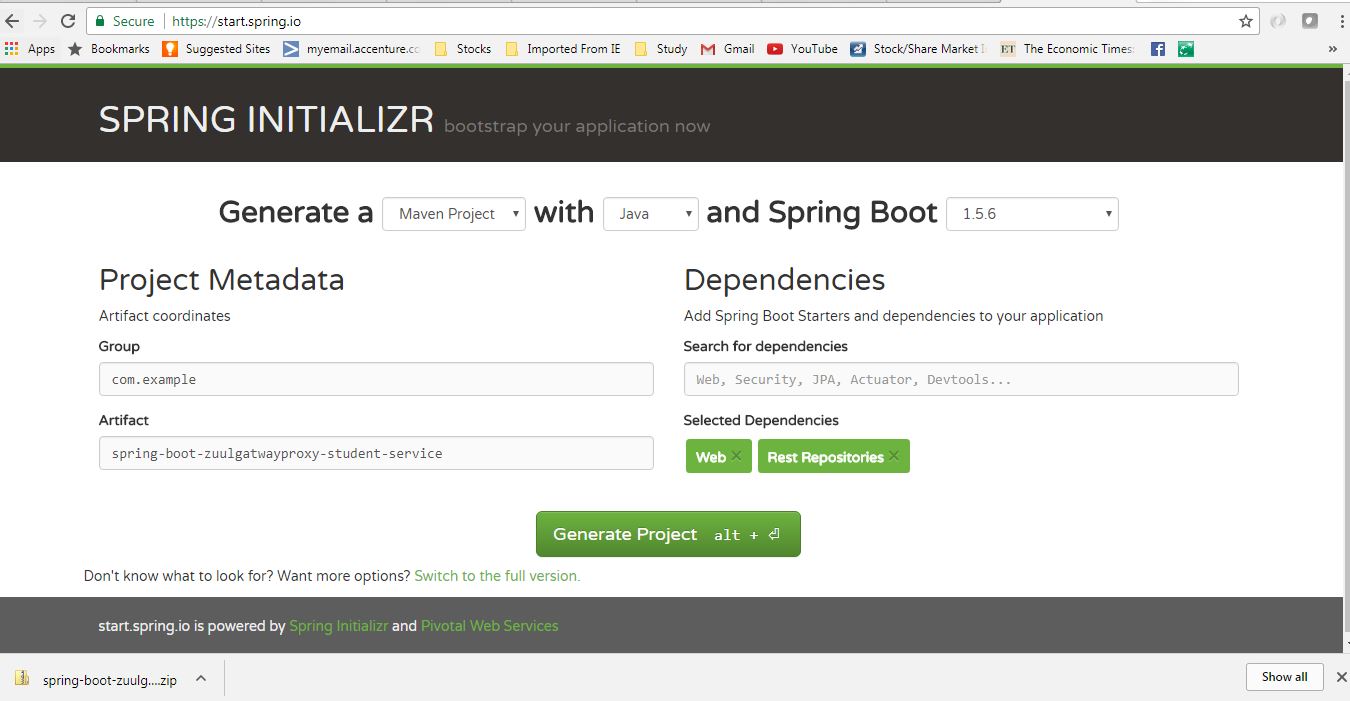
* Java 1.8 and Eclipse IDE as development environment
* Spring cloud Zuul as Gateway proxy provider
* Spring boot as application framework
* Spring Rest for exposing microservice as REST
* Maven as build tool

###### **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/" \t "_blank) with dependencies i.e.Web and Rest Repositories. Give other maven GAV coordinates and download the project.

Student service maven project generation

Unzip and import the project into Eclipse as existing maven project. In this step do a fresh maven build using command mvn clean install so that all maven dependencies gets download properly.

###### 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.

After all changes, the class will look like below.

|  |
| --- |
| package com.example.springboostudentservice;    import java.util.Date;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;    @RestController  @SpringBootApplication  public class SpringBootStudentServiceApplication  {      @RequestMapping(value = "/echoStudentName/{name}")      public String echoStudentName(@PathVariable(name = "name") String name)      {          return "hello  <strong style=\"color: red;\">" + name + " </strong> Responsed on : " + new Date();      }        @RequestMapping(value = "/getStudentDetails/{name}")      public Student getStudentDetails(@PathVariable(name = "name") String name)      {          return new Student(name, "Pune", "MCA");      }        public static void main(String[] args)      {          SpringApplication.run(SpringBootStudentServiceApplication.class, args);      }  }    class Student  {      String name;      String address;      String cls;        public Student(String name, String address, String cls) {          super();          this.name = name;          this.address = address;          this.cls = cls;      }        public String getName() {          return name;      }        public String getAddress() {          return address;      }        public String getCls() {          return cls;      }  } |

###### 4.3. Application Configurations

Now open application.properties file and add those entries.

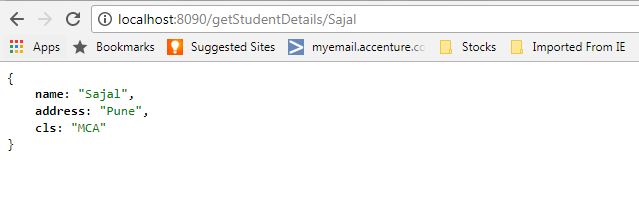
|  |
| --- |
| spring.application.name=student  server.port=8090 |

Here we are giving one name to this service by property spring.application.name=student and also we are defining the default port by server.port=8090. We need to override the default port, as we will have multiple instances of different microservices will run in localhost.

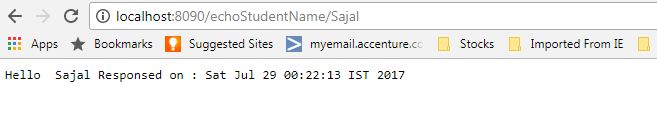
###### 4.4. Verify Student Service

Finally do a maven build using command mvn clean install and Start this project as spring boot application by running command java -jar target\spring-boot-zuulgatwayproxy-student-service-0.0.1-SNAPSHOT.jar. Now once the server started, go to browser and test whether the endpoints are working.

[http://localhost:8090/echoStudentName/Sajal](http://localhost:8090/echoStudentName/Sajal" \t "_blank)

Browser Output

[http://localhost:8090/getStudentDetails/Sajal](http://localhost:8090/getStudentDetails/Sajal" \t "_blank)

Browser Output

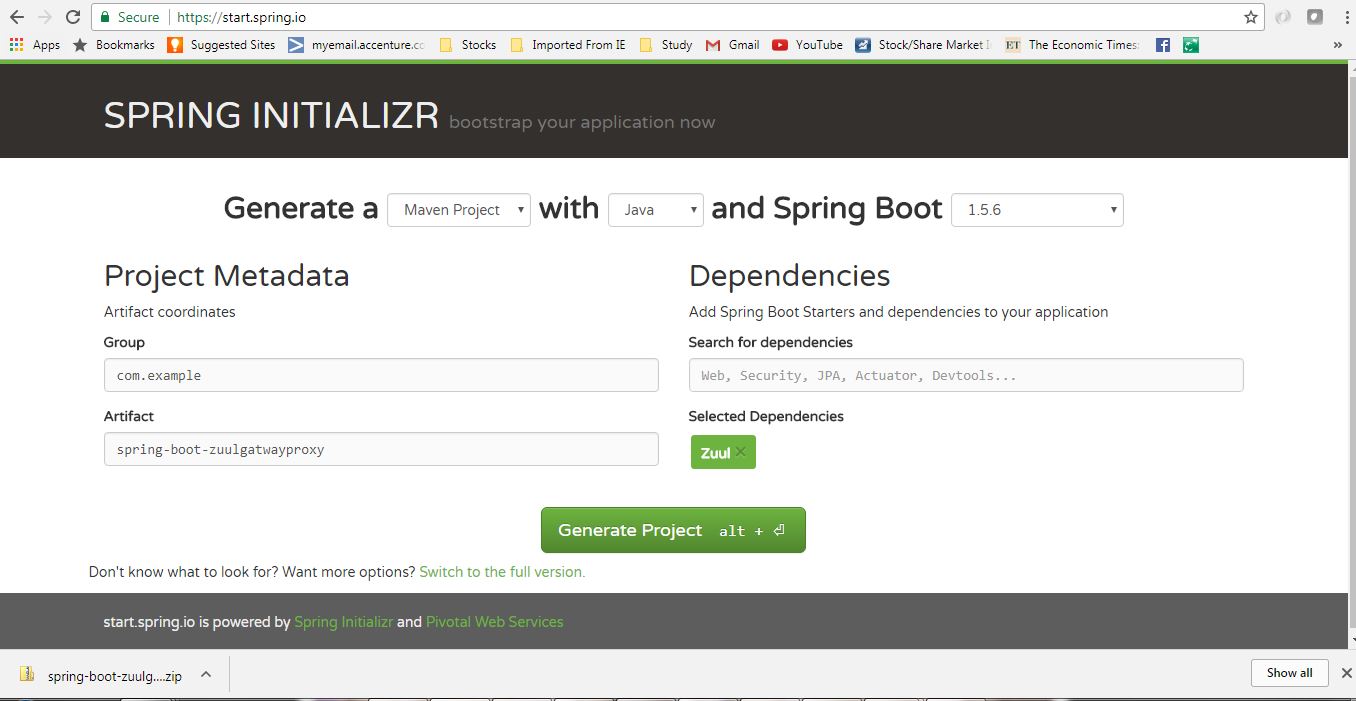
Now we will create the actual proxy service using Zuul.

###### 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/" \t "_blank) with Zuul dependency. Give other maven GAV coordinates and download the project.

Zuul proxy service maven project generation

Unzip and import the project into Eclipse as existing maven project. In this step do a fresh maven build using command mvn clean install so that all maven dependencies gets download properly.

###### 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.

|  |
| --- |
| package com.example.springbootzuulgatwayproxy;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.zuul.EnableZuulProxy;  import org.springframework.context.annotation.Bean;  import com.example.springbootzuulgatwayproxy.filters.ErrorFilter;  import com.example.springbootzuulgatwayproxy.filters.PostFilter;  import com.example.springbootzuulgatwayproxy.filters.PreFilter;  import com.example.springbootzuulgatwayproxy.filters.RouteFilter;    @SpringBootApplication  @EnableZuulProxy  public class SpringBootZuulgatwayproxyApplication {        public static void main(String[] args) {          SpringApplication.run(SpringBootZuulgatwayproxyApplication.class, args);      }        @Bean      public PreFilter preFilter() {          return new PreFilter();      }      @Bean      public PostFilter postFilter() {          return new PostFilter();      }      @Bean      public ErrorFilter errorFilter() {          return new ErrorFilter();      }      @Bean      public RouteFilter routeFilter() {          return new RouteFilter();      }  } |

###### 5.3. Zuul routes configuration

Open application.properties and add below entries-

|  |
| --- |
| #Zuul routes. Here for /student path, we are routing to localhost:8090 with extra path after that.  zuul.routes.student.url=http://localhost:8090    #Ribbon is auto integrated with Zuul and for this exercise we are not using that.  ribbon.eureka.enabled=false    #Will start the gateway server @8080  server.port=8080 |

Here zuul.routes.student.url will route all traffic to request for /student to the actual student service server. This way we can add route to other services as well.  
ribbon.eureka.enabled is auto integrated with Zuul.  
server.port – is needed to override the default port, as we will have multiple instances of different microservices will run in localhost.

###### 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 filer 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.

|  |
| --- |
| package com.example.springbootzuulgatwayproxy.filters;    import javax.servlet.http.HttpServletRequest;  import com.netflix.zuul.ZuulFilter;  import com.netflix.zuul.context.RequestContext;    public class PreFilter extends ZuulFilter {      @Override    public String filterType() {      return "pre";    }      @Override    public int filterOrder() {      return 1;    }      @Override    public boolean shouldFilter() {      return true;    }      @Override    public Object run() {      RequestContext ctx = RequestContext.getCurrentContext();      HttpServletRequest request = ctx.getRequest();        System.out.println("Request Method : " + request.getMethod() + " Request URL : " + request.getRequestURL().toString());      return null;    }  } |

**post filter**

|  |
| --- |
| package com.example.springbootzuulgatwayproxy.filters;    import com.netflix.zuul.ZuulFilter;    public class PostFilter extends ZuulFilter {      @Override    public String filterType() {      return "post";    }      @Override    public int filterOrder() {      return 1;    }      @Override    public boolean shouldFilter() {      return true;    }      @Override    public Object run() {     System.out.println("Inside Response Filter");        return null;    }  } |

**route filter**

|  |
| --- |
| package com.example.springbootzuulgatwayproxy.filters;    import com.netflix.zuul.ZuulFilter;    public class RouteFilter extends ZuulFilter {      @Override    public String filterType() {      return "route";    }      @Override    public int filterOrder() {      return 1;    }      @Override    public boolean shouldFilter() {      return true;    }      @Override    public Object run() {     System.out.println("Inside Route Filter");      return null;    }  } |

**Error filter**

|  |
| --- |
| package com.example.springbootzuulgatwayproxy.filters;    import com.netflix.zuul.ZuulFilter;    public class ErrorFilter extends ZuulFilter {      @Override    public String filterType() {      return "error";    }      @Override    public int filterOrder() {      return 1;    }      @Override    public boolean shouldFilter() {      return true;    }      @Override    public Object run() {     System.out.println("Inside Route Filter");        return null;    }  } |

###### 5.5. Register zuul filters

Create the bean definition of these filters to be auto-registered and enabled.

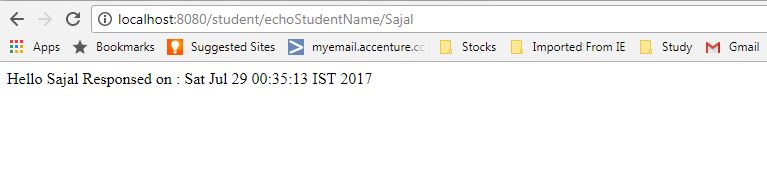
|  |
| --- |
| @Bean  public PreFilter preFilter() {      return new PreFilter();  }  @Bean  public PostFilter postFilter() {      return new PostFilter();  }  @Bean  public ErrorFilter errorFilter() {      return new ErrorFilter();  }  @Bean  public RouteFilter routeFilter() {      return new RouteFilter();  } |

###### **6. Netflix zuul example demo**

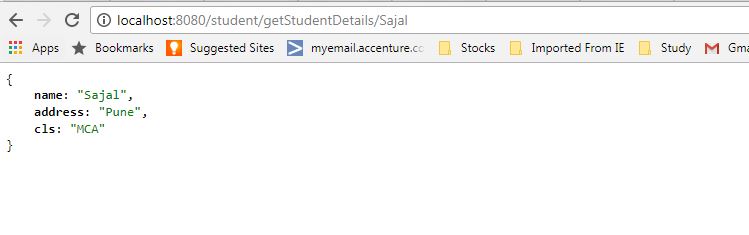
So we have enabled Zuul, added required configurations and developed Filters. So new we can do the basic testing to understand the whole thing.

Do a maven build using command mvn clean install and Start this project as spring boot application by running command java -jar target\spring-boot-zuulgatwayproxy-0.0.1-SNAPSHOT.jar. Now once the server started, go to browser and test whether the endpoints are working by accessing the student service bu it’s name i.e. /student.

<http://localhost:8080/student/getStudentDetails/Sajal>

Proxy service output

<http://localhost:8080/student/echoStudentName/Sajal>

Proxy service output

#### **FeignClient and RestTemplate**

The calls from one microservice (Rest Client) to the other microservices can be done using:

* **RestTemplate**. An object that’s capable of sending requests to REST API services.
* **FeignClient** (acts like a proxy), and provides another approach to RestTemplate.

Both, load balance requests across the services.

[Feign](https://github.com/Netflix/feign) is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.











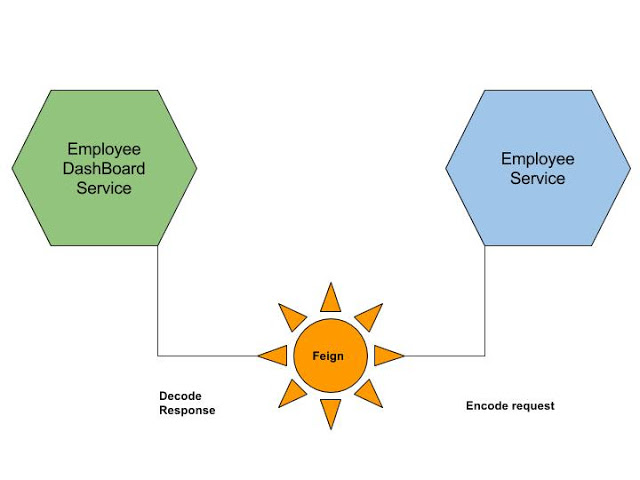
##### What Is a Feign Client?

Netflix provides Feign as an abstraction over REST-based calls, by which microservices can communicate with each other, but developers don't have to bother about REST internal details.

##### Why We Use Feign Client

For example, EmployeeDashBoard service has to communicate with EmployeeService, we programmatically construct the URL of the dependent microservice, then called the service using RestTemplate, so we need to be aware of the RestTemplate API to communicate with other microservices, which is certainly not part of our business logic.

The question is, why should a developer have to know the details of a REST API? Microservice developers only concentrate on business logic, so Spring addresses this issues and comes with Feign Client, which works on the declarative principle. We have to create an interface/contract, then Spring creates the original implementation on the fly, so a REST-based service call is abstracted from developers. Not only that — if you want to customize the call, like encoding your request or decoding the response in a Custom Object, you can do it with Feign in a declarative way. Feign, as a client, is an important tool for microservice developers to communicate with other microservices via Rest API.

[](https://1.bp.blogspot.com/-cLPct-Cfx3w/WYNgnPfYYJI/AAAAAAAAFqE/3mqTXBrAaOEA9JmkYx9uRgph0prtprSDgCLcBGAs/s1600/Microservices%2BCommunication_%2BFeign%2Bas%2BRest%2BClient.jpg)

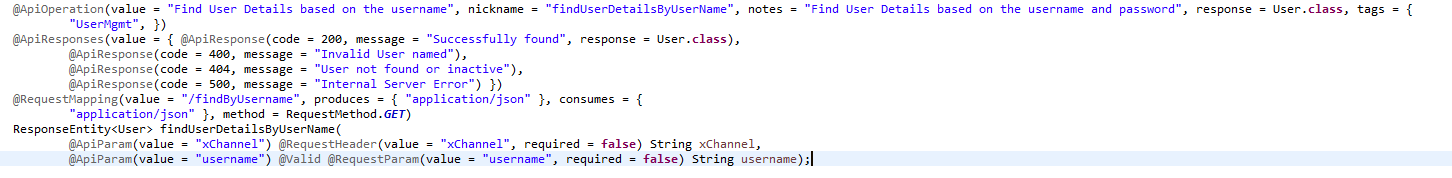
**Coding Time**

Here, we will alter our **Auth Service** to make it Feign-enabled.

**Step 1:** We will add the feign dependency into **Auth-Service**.



**Step 2:** Now, we have to create an interface where we declare the services we want to call. Please note that Service Request mapping is same as the**user-mgmt-service** Rest URL.

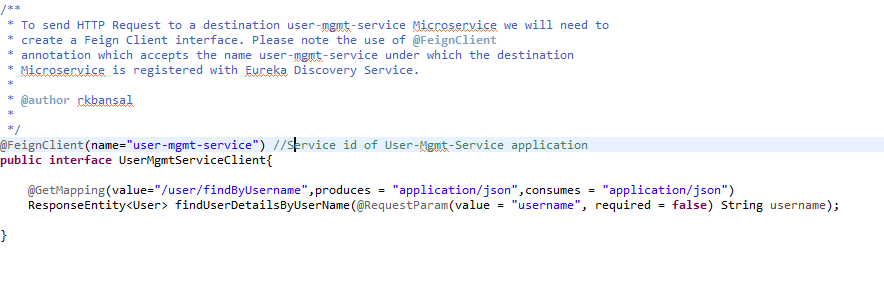


Feign will call this URL when we call the **auth-service** service.

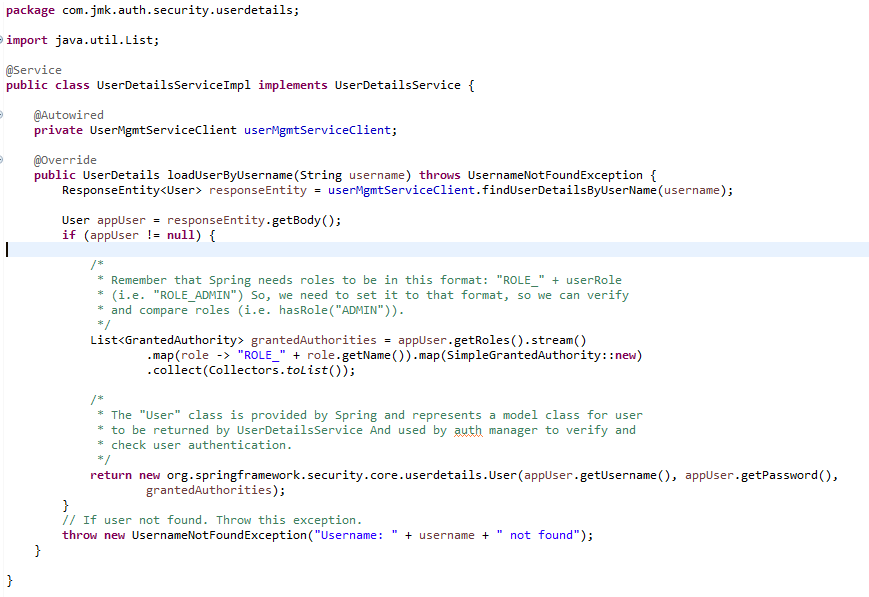
Feign dynamically generates the implementation of the interface we created, so Feign has to know which service to call beforehand. That's why we need to give a name for the interface, which is the {Service-Id} of UserMgmtService. Now, Feign contacts the Eureka server with this Service Id, resolves the actual IP/hostname of the UserMgmtService, and calls the URL provided in Request Mapping.

To send HTTP Request to a destination employee service Microservice we will need to create a Feign Client interface. Please note the use of @FeignClient annotation which accepts the name user-mgmt-service under which the destination Microservice is registered with Eureka Discovery Service.

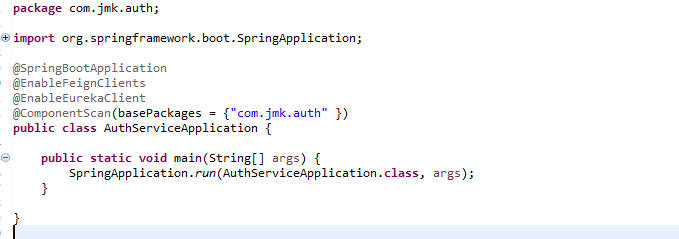
**N.B.:** When using @PathVariable for Feign Client, always use value property or it will give you the errorjava.lang.IllegalStateException: PathVariable annotation was empty on param 0



**Step 3:** Now we will create a UserDetailsServiceImpl where we autowire our Interface so Spring can Inject actual implementation during runtime. Then, we call that implementation to call the EmployeeService REST API.



**Step 4:** At last, we need to tell our project that we will use Feign client, so scan its annotation. For this, we need to add the @EnableFeignClients annotation on top of AuthServiceApplication**.**



Now we are all set to use Feign client.

**Testing Time**

Start the following microservices in order:

1. UserMgmtService
2. EurekaServer
3. AuthService

Now the URL  [http://localhost:8081/dashboard/feign/peers](http://localhost:8081/dashboard/feign/peers" \t "_blank) in the browser.

#### **Load Balancer (Using Ribbon Approach)**

*What if more than one instance of a service running on different ports. So, we need to balance the requests among all the instances of a service.*

*When using ‘Ribbon’ approach (default), requests will be distributed equally among them.*

In this Spring cloud tutorial, learn to use **client side load balancing using Netflix Ribbon** in spring boot/cloud projects. Learn to build microservice based applications which use **ribbon** as client side load balancer and **eureka** as registry service. Learn how we can dynamically add new instances of microservices under the load balancer.

Table of Contents

1. [Traditional server side load balancing](https://howtodoinjava.com/spring-cloud/spring-boot-ribbon-eureka/" \l "server-side-lb)

2. [Client side load balancing](https://howtodoinjava.com/spring-cloud/spring-boot-ribbon-eureka/" \l "client-side-lb)

3. [Netflix ribbon - Client side load balancer](https://howtodoinjava.com/spring-cloud/spring-boot-ribbon-eureka/" \l "ribbon)

4. [Netflix ribbon example](https://howtodoinjava.com/spring-cloud/spring-boot-ribbon-eureka/" \l "demo)

5. [Test the application](https://howtodoinjava.com/spring-cloud/spring-boot-ribbon-eureka/" \l "testing)

6. [Summary](https://howtodoinjava.com/spring-cloud/spring-boot-ribbon-eureka/" \l "Summary)

##### 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 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.

Load balancer has a public IP and DNS. The client makes a request using that public IP/DNS. Load balancer decides to which internal application server request will be forwarded to. It mainly use round robin or sticky session algorithm. We call it server side load balancing.

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.

Another problem is to have a fail-over policy to provide the client a seamless experience. Finally we need a separate server to host the load balancer instance which has the impact on cost and maintenance.

##### 2. 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.

Now let’s visualize the big picture. In microservice architecture, we will have to develop many microservices and each microservice may have multiple instances in the ecosystem. To overcome this complexity we have already one popular solution to use **service discovery pattern**. In spring boot applications, we have couple of options in the service discovery space such as eureka, consoul, zookeeper etc.

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

To get ribbon binaries, go to [maven central](https://search.maven.org/" \l "search%7Cga%7C1%7Cribbon). Here is an example to add dependency in Maven:

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

##### 4. Netflix ribbon example

4.1. Technology stack

* Java, Eclipse, Maven as Development Environment
* Spring-boot and Cloud as application framework
* Eureka as Service registry Server
* Ribbon as Client Side Load balancer

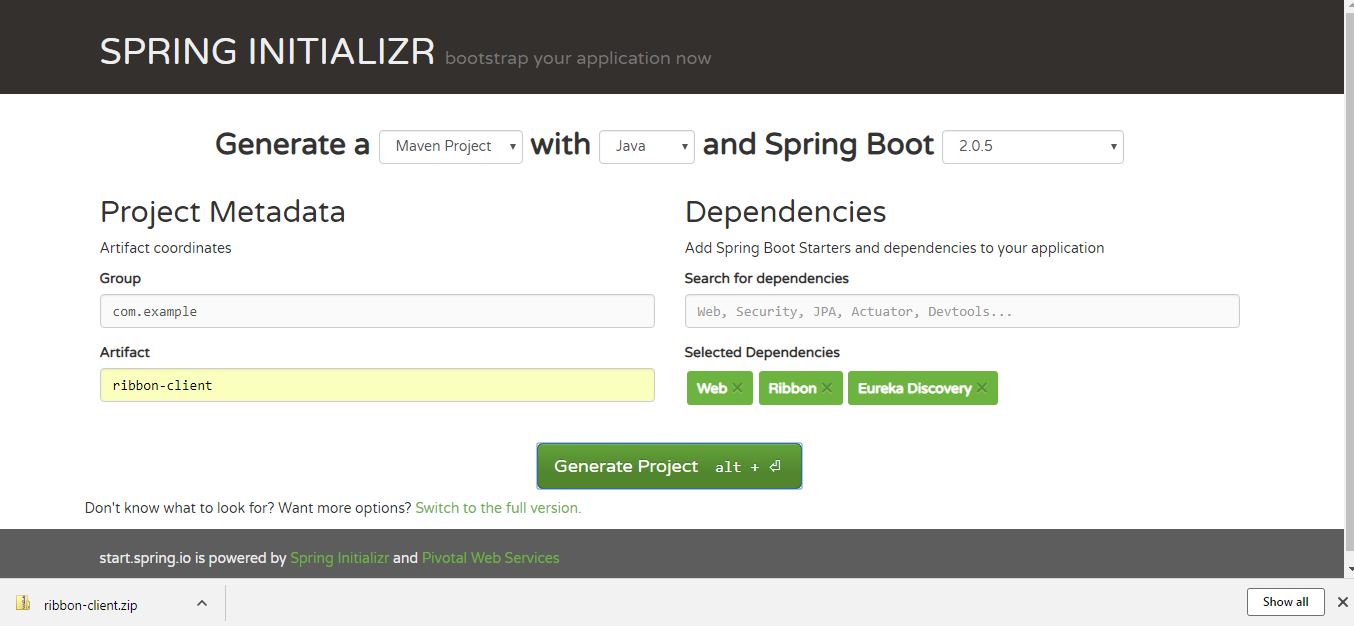
We will create the following components and see how the whole eco system coordinates in distributed environment.

* 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

We will create a simple microservice using Spring boot and will expose oe simple REST endpoint. Create one simple spring boot project named ribbon-server with **spring-boot-web** and **service discovery client** dependency for hosting this in web server and expose one Rest Controller to test.

To do this we need to go to https://start.spring.io/ and give the maven coordinates and select dependencies. Download the zip file containing the skeleton project. Then we need to import that in eclipse once unzipped in suitable folder.

Project generated from Spring Initializer

4.2.1. Create rest endpoint

Write one Rest Controller and expose one Rest Endpoint as below.

|  |
| --- |
| MyRestController.java |
| package com.example.ribbonserver;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.core.env.Environment;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;    @RestController  public class MyRestController {        @Autowired      Environment environment;        @GetMapping("/")      public String health() {          return "I am Ok";      }        @GetMapping("/backend")      public String backend() {          System.out.println("Inside MyRestController::backend...");            String serverPort = environment.getProperty("local.server.port");            System.out.println("Port : " + serverPort);            return "Hello form Backend!!! " + " Host : localhost " + " :: Port : " + serverPort;      }  } |

4.2.2 Enable discovery client

Register this service to eureka to do that we need to add **@EnableDiscoveryClient** in the application class. Also we need to add below entries in the application propererty file.

|  |
| --- |
| RibbonServerApplication.java |
| package com.example.ribbonserver;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.client.discovery.EnableDiscoveryClient;    @SpringBootApplication  @EnableDiscoveryClient  public class RibbonServerApplication {        public static void main(String[] args) {          SpringApplication.run(RibbonServerApplication.class, args);      }  } |
| application.properties |
| spring.application.name=server  server.port = 9090    eureka.client.serviceUrl.defaultZone= http://${registry.host:localhost}:${registry.port:8761}/eureka/  eureka.client.healthcheck.enabled= true  eureka.instance.leaseRenewalIntervalInSeconds= 1  eureka.instance.leaseExpirationDurationInSeconds= 2 |

4.3. Eureka service regstry server

Create the service discovery server. This is also very easy. Just we need to create a spring boot project as above with Eureka Server as dependency and do the below configurations.

4.3.1. Eureka server configuration

Once the spring boot service is ready and imported in eclipse, add **@EnableEurekaServer** annotation in the spring boot application class and also add the below configuration in application properties file.

|  |
| --- |
| RibbonEurekaServerApplication.java |
| package com.example.ribboneurekaserver;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;    @SpringBootApplication  @EnableEurekaServer  public class RibbonEurekaServerApplication {        public static void main(String[] args) {          SpringApplication.run(RibbonEurekaServerApplication.class, args);      }  } |
| application.properties |
| spring.application.name= ${springboot.app.name:eureka-serviceregistry}  server.port = ${server-port:8761}  eureka.instance.hostname= ${springboot.app.name:eureka-serviceregistry}  eureka.client.registerWithEureka= false  eureka.client.fetchRegistry= false  eureka.client.serviceUrl.defaultZone: http://${registry.host:localhost}:${server.port}/eureka/ |

4.4. Create another microservice

Follow previous section to create another service named ribbon-client with added depedency spring-cloud-starter-netflix-ribbon. Once downloaded, import the project in eclipse and do the following configurations.

4.4.1. Ribbon configuration

In the application class, add two annotations **@RibbonClient** and **@EnableDiscoveryClient** to enable ribbon and Eureka client for service registry.

|  |
| --- |
| RibbonClientApplication.java |
| package com.example.ribbonclient;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  import org.springframework.cloud.netflix.ribbon.RibbonClient;    @EnableDiscoveryClient  @SpringBootApplication  @RibbonClient(name = "server", configuration = RibbonConfiguration.class)  public class RibbonClientApplication {        public static void main(String[] args) {          SpringApplication.run(RibbonClientApplication.class, args);      }  } |

In the application.properties, we need to do the below configurations. Here **server.ribbon.listOfServers** is disabled, we can enable this to manually add server to this load balancer. We will check this in the testing section. Other properties are self explanatory.

|  |
| --- |
| application.properties |
| spring.application.name=client  server.port=8888    eureka.client.serviceUrl.defaultZone= http://${registry.host:localhost}:${registry.port:8761}/eureka/  eureka.client.healthcheck.enabled= true  eureka.instance.leaseRenewalIntervalInSeconds= 1  eureka.instance.leaseExpirationDurationInSeconds= 2    server.ribbon.eureka.enabled=true  #server.ribbon.listOfServers=localhost:9090,localhost:9091,localhost:9092  server.ribbon.ServerListRefreshInterval=1000  #logging.level.root=TRACE |

Now we need to create one more configuration class for ribbon to mention the **load balancing algorithm and health check**. We will now use the default once provided by Ribbon, but in this class we can very well override those and add ours custom logic.

|  |
| --- |
| RibbonConfiguration.java |
| package com.example.ribbonclient;  import com.netflix.client.config.IClientConfig;  import com.netflix.loadbalancer.AvailabilityFilteringRule;  import com.netflix.loadbalancer.IPing;  import com.netflix.loadbalancer.IRule;  import com.netflix.loadbalancer.PingUrl;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;    public class RibbonConfiguration {        @Autowired      IClientConfig config;        @Bean      public IPing ribbonPing(IClientConfig config) {          return new PingUrl();      }        @Bean      public IRule ribbonRule(IClientConfig config) {          return new AvailabilityFilteringRule();      }  } |

##### 5. Test the application

5.1. Start components

Do the final build use command mvn clean install and check if the build is successful. If there is any error you need to fix those to proceed. Once we have successful build for all the maven projects, we will start the services one by one.

Eureka first, then the back-end micro service and finally the frontend micro service.

To start each microservice, we will use 'java -jar -Dserver.port=XXXX target/YYYYY.jar' command.

5.2. Deploy multiple instances of backend microservice

To do that we need to use different port for this, to start service in a specific port we need to pass the port in this way.  
java -jar -Dserver.port=XXXX target/YYYYY.jar. We will create 3 instances of this service in ports 9090, 9091 and 9092 ports.

5.3. Verify eureka server

Now go to http://localhost:8761/ in browser and check that eureka server is running with all microservices are registed with desired number of instances.

5.4. Check if client side load balancing is working

In the frontend microservice, we are calling the backend microservice using **RestTemplate**. Rest tempate is enabled as client side load balancer using **@LoadBalanced** annotation.

Now go to browser and open the client microservice rest endpoint http://localhost:8888/client/frontend and see that response is coming from any one of the backend instance.

To understand this backend server is returning it’s running port and we are displaying that in client microservice response as well. Try refreshing this url couple of times and notice that the port of backend server keeps changing, that means client side load balancing is working. Now try to add more instance of backend server and check that is also registered in eureka server and eventually considered in ribbon, as once that will be registered in eureka and ribbon automatically ribbon will send request to the new instances as well.

5.5. Test with hard code backends without service discovery

Go the frontend microservice application.properties file and enable this.

|  |
| --- |
| application.properties |
| server.ribbon.listOfServers=localhost:9090,localhost:9091,localhost:9092  server.ribbon.eureka.enabled=false |

Now test the client url. You will get response from the registered instances only. Now if you start new instance of backend microservice in different port, Ribbon will not send request to the new instance until we register that manually in the ribbon.

If you have difficulty in testing this, I will suggest too remove all the eureka related configurations from all the applications and also stop the eureka server. Hope you will not face any difficulty in testing this as well.























































#### **Hystrix Circuit Breaker Pattern**

**Hystrix** is the implementation of [Circuit Breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html), which gives a control over latency and failure between distributed services.

If you have, let’s say 3 services, A calls → B, and B calls → C service. What if, a failure happened at B? The error will cascade down to service C, right?.

A -> B (failure) -> C

Another example, lets say a service A calls a remote service R, and for some reason the remote service is down. How can we handle such a situation?

What we would like to do is stop failures from cascading down, and provide a way to self-heal, which improves the system’s overall resiliency.

**Hystrix** is the implementation of [Circuit Breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html), which gives a control over latency and failure between distributed services.

The main idea is to stop cascading failures by failing fast and recover as soon as possible — Important aspects of fault-tolerant systems that self-heal.

So, we’ll add Hystrix to gallery service, and we’ll simulate a failure at image service. In the pom.xml file

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>

In the spring boot main class

// ...  
**@EnableCircuitBreaker // Enable circuit breakers**public class SpringEurekaGalleryApp {  
 public static void main(String[] args) {  
 SpringApplication.run(SpringEurekaGalleryApp.class, args);  
 }  
}

Spring looks for any method annotated with the @HystrixCommandannotation, and wraps that method so that Hystrix can monitor it.

Hystrix watches for failures in that method, and if failures reached a threshold (limit), Hystrix opens the circuit so that subsequent calls will automatically fail. Therefore, and while the circuit is open, Hystrix redirects calls to the fallback method.

So, In the controller class, update getGallery(), add annotation and fallback method.

|  |
| --- |
|  |
|  |  |
|  | import java.util.List; |
|  |  |
|  | import org.springframework.beans.factory.annotation.Autowired; |
|  | import org.springframework.core.env.Environment; |
|  | import org.springframework.web.bind.annotation.PathVariable; |
|  | import org.springframework.web.bind.annotation.RequestMapping; |
|  | import org.springframework.web.bind.annotation.RestController; |
|  | import org.springframework.web.client.RestTemplate; |
|  |  |
|  | import com.eureka.gallery.entities.Gallery; |
|  | import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand; |
|  |  |
|  | @RestController |
|  | @RequestMapping("/") |
|  | public class HomeController { |
|  | @Autowired |
|  | private RestTemplate restTemplate; |
|  |  |
|  | @HystrixCommand(fallbackMethod = "fallback") |
|  | @RequestMapping("/{id}") |
|  | public Gallery getGallery(@PathVariable final int id) { |
|  | // create gallery object |
|  | Gallery gallery = new Gallery(); |
|  | gallery.setId(id); |
|  |  |
|  | // get list of available images |
|  | @SuppressWarnings("unchecked") // we'll throw an exception from image service to simulate a failure |
|  | List<Object> images = restTemplate.getForObject("http://image-service/images/", List.class); |
|  | gallery.setImages(images); |
|  |  |
|  | return gallery; |
|  | } |
|  |  |
|  | // a fallback method to be called if failure happened |
|  | public Gallery fallback(int galleryId, Throwable hystrixCommand) { |
|  | return new Gallery(galleryId); |
|  | } |
|  | } |

In the controller class of image service, throw an exception in getImages()

throw new Exception("Images can't be fetched");

Now, go to the browser, and hit localhost:8762/gallery/1. You should get an empty gallery object (no images).

{  
 "id": 1,  
 "images": null  
}

[Hystrix](https://github.com/Netflix/Hystrix" \t "_blank) 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. This is related to distributed computing style of Eco system using lots of underlying Microservices. This is where circuit breaker pattern helps and Hystrix is an tool to build this circuit breaker.

##### Hystrix Example for real impatient

Hystrix configuration is done in four major steps.

1. Add Hystrix starter and dashboard dependencies.

|  |
| --- |
| <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>  <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>  </dependency> |

1. Add @EnableCircuitBreaker annotation
2. Add @EnableHystrixDashboard annotation
3. Add annotation @HystrixCommand(fallbackMethod = "myFallbackMethod")

Table of Contents

[What is Circuit Breaker Pattern?](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "what-is-circuit-breaker)

[Hystrix Circuit Breaker Example](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "hystrix-example)

[Create Student Microservice](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "student-microservice)

[Create School Microservice - Hystrix Enabled](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "school-microservice)

[Test Hystrix Circuit Breaker](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "demo)

[Hystrix Dashboard](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "hystrix-dashboard)

[Summary](https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/" \l "summary)

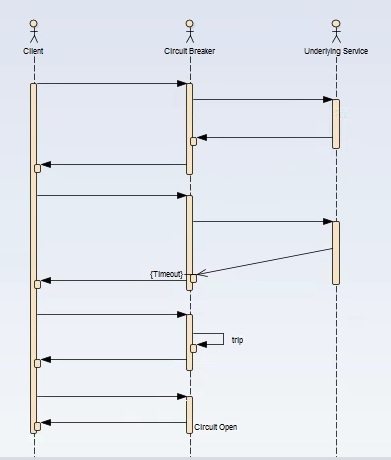
##### Whay is Circuit Breaker Pattern?

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.

So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls goto the circuit breaker fall back and finally restores automatically once the service came back !! That’s cool right?

[](https://cdn2.howtodoinjava.com/wp-content/uploads/2017/07/CB_Sequence.jpg)Circuit Breaker Sequence of Invocation

##### 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.

Tech Stack and Demo Runtime

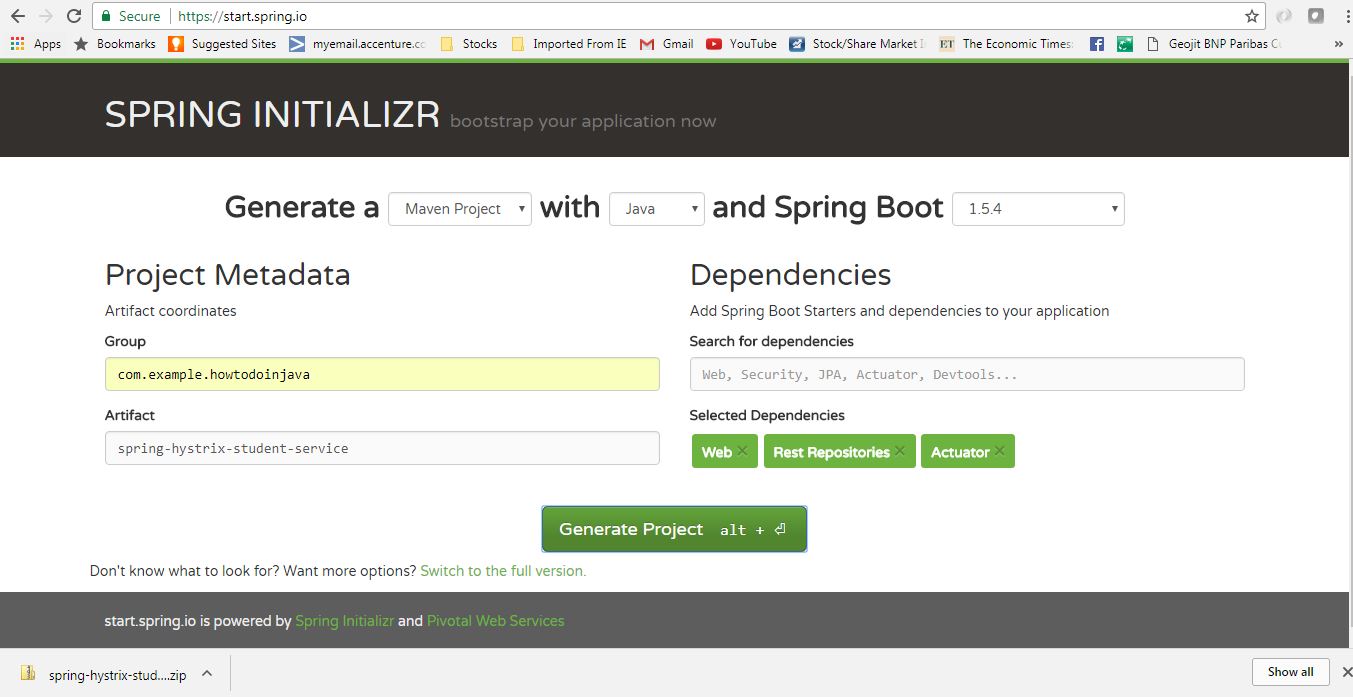
* Java 1.8
* Eclipse as IDE
* Maven as build tool
* Spring cloud Hystrix as circuit breaker framework
* Spring boot
* Spring Rest

##### 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/" \t "_blank) with three dependencies i.e. Web, Rest Repositories and Actuator. Give other maven GAV coordinates and download the project.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/studentservciegeneration.jpg)Student Service Generation

Unzip and import the project into Eclipse as existing maven project. In this step, all necessary dependencies will be downloaded from maven repository.

Server Port Settings

Open application.properties and add port information.

|  |
| --- |
| server.port = 8098 |

This will enable this application run on default port 8098. We can easily override this by supplying -Dserver.port = XXXX argument at the time of starting the server.

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.

**StudentServiceController.java**

|  |
| --- |
| package com.example.howtodoinjava.springhystrixstudentservice.controller;    import java.util.ArrayList;  import java.util.HashMap;  import java.util.List;  import java.util.Map;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.example.howtodoinjava.springhystrixstudentservice.domain.Student;    @RestController  public class StudentServiceController {        private static Map<String, List<Student>> schooDB = new HashMap<String, List<Student>>();        static {          schooDB = new HashMap<String, List<Student>>();            List<Student> lst = new ArrayList<Student>();          Student std = new Student("Sajal", "Class IV");          lst.add(std);          std = new Student("Lokesh", "Class V");          lst.add(std);            schooDB.put("abcschool", lst);            lst = new ArrayList<Student>();          std = new Student("Kajal", "Class III");          lst.add(std);          std = new Student("Sukesh", "Class VI");          lst.add(std);            schooDB.put("xyzschool", lst);        }        @RequestMapping(value = "/getStudentDetailsForSchool/{schoolname}", method = RequestMethod.GET)      public List<Student> getStudents(@PathVariable String schoolname) {          System.out.println("Getting Student details for " + schoolname);            List<Student> studentList = schooDB.get(schoolname);          if (studentList == null) {              studentList = new ArrayList<Student>();              Student std = new Student("Not Found", "N/A");              studentList.add(std);          }          return studentList;      }  } |

**Student.java**

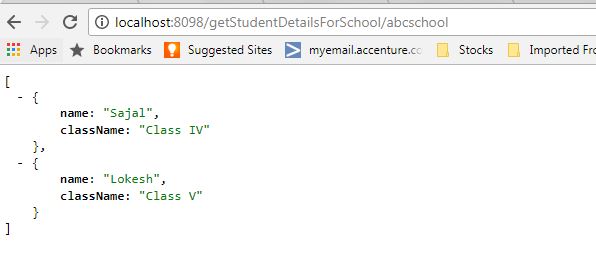
|  |
| --- |
| package com.example.howtodoinjava.springhystrixstudentservice.domain;    public class Student {        private String name;      private String className;        public Student(String name, String className) {          super();          this.name = name;          this.className = className;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        public String getClassName() {          return className;      }        public void setClassName(String className) {          this.className = className;      }  } |

Build and Test Student Service

Now do a final build using mvn clean install and run the server using command java -jar target\spring-hystrix-student-service-0.0.1-SNAPSHOT.jar. This will start the student service in default port 8098.

Open browser and type http://localhost:8098/getStudentDetailsForSchool/abcschool.

It should show the below output in browser –

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/studentserviceresponse.jpg)Student Service Response

##### Create School Service – Hystrix Enabled

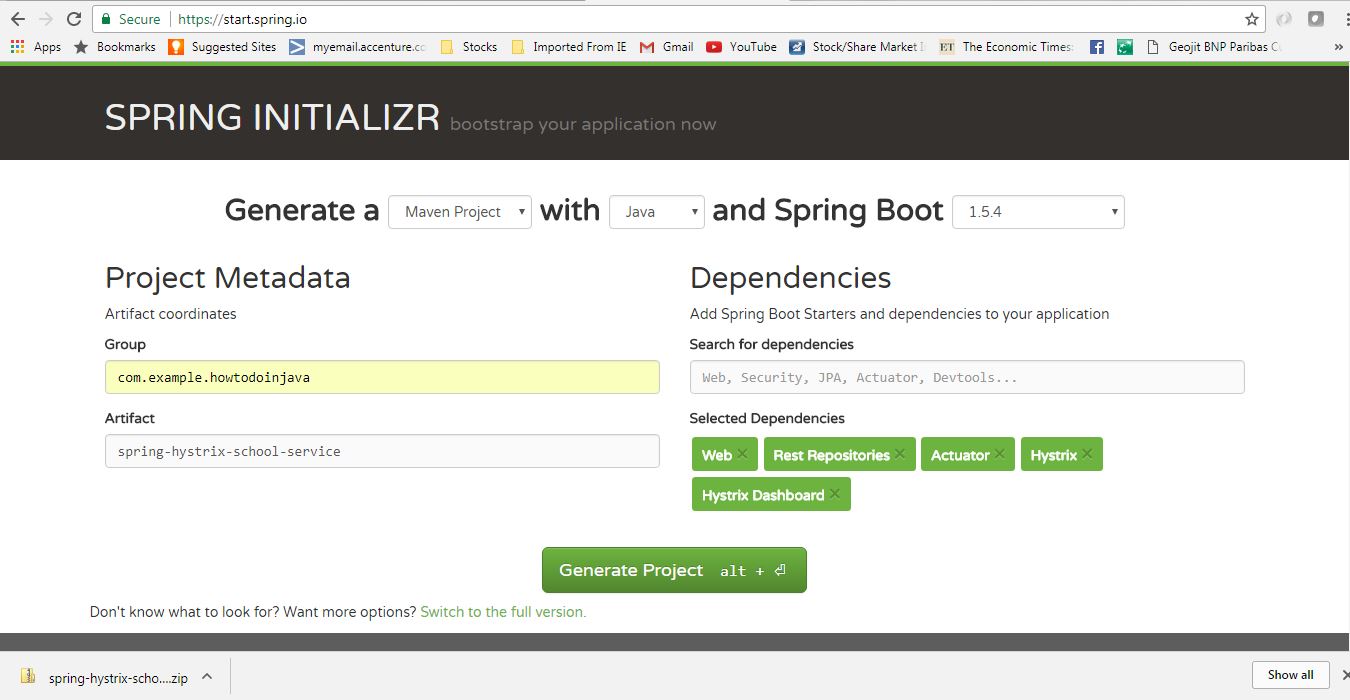
Similar to Student service, create another microservice for School. It will internally invoke already developed Student Service.

Generate spring boot project

Create a Spring boot project from [Spring Boot initializer portal](https://start.spring.io/" \t "_blank) 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

Give other maven GAV coordinates and download the project.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/schoolservicegeneration.jpg)School Service Project

Unzip and import the project into Eclipse as existing maven project. In this step, all necessary dependencies will be downloaded from maven repository.

Server Port Settings

Open application.properties and add port information.

|  |
| --- |
| server.port = 9098 |

This will enable this application run on default port 9098. We can easily override this by supplying -Dserver.port = XXXX argument at the time of starting the server.

Enable Hystrix Settings

Open SpringHystrixSchoolServiceApplication i.e the generated class with @SpringBootApplication and add @EnableHystrixDashboard and @EnableCircuitBreaker annotations.

This will **enable Hystrix circuit breaker** in the application and also will add one useful dashboard running on localhost provided by Hystrix.

|  |
| --- |
| package com.example.howtodoinjava.springhystrixschoolservice;    import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;  import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;    @SpringBootApplication  @EnableHystrixDashboard  @EnableCircuitBreaker  public class SpringHystrixSchoolServiceApplication {        public static void main(String[] args) {          SpringApplication.run(SpringHystrixSchoolServiceApplication.class, args);      }  } |

Add REST controller

Add SchoolServiceController Rest Controller where we will expose /getSchoolDetails/{schoolname} endpoint which will simply return school details along with its student details. For Student Details it will call the already developed Student service endpoint. We will create a Delegate layer StudentServiceDelegate.java to call the Student Service. This simple Code will look like

**SchoolServiceController.java**

|  |
| --- |
| package com.example.howtodoinjava.springhystrixschoolservice.controller;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.example.howtodoinjava.springhystrixschoolservice.delegate.StudentServiceDelegate;    @RestController  public class SchoolServiceController {        @Autowired      StudentServiceDelegate studentServiceDelegate;        @RequestMapping(value = "/getSchoolDetails/{schoolname}", method = RequestMethod.GET)      public String getStudents(@PathVariable String schoolname) {          System.out.println("Going to call student service to get data!");          return studentServiceDelegate.callStudentServiceAndGetData(schoolname);      }  } |

**StudentServiceDelegate**

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

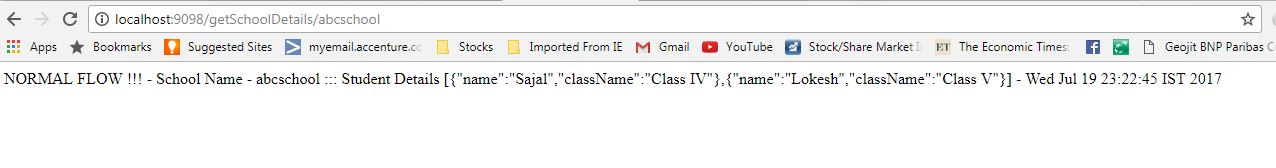
* Invoke Student Service through spring framework provided RestTemplate
* Add Hystrix Command to enable fallback method – @HystrixCommand(fallbackMethod = "callStudentServiceAndGetData\_Fallback") – this means that we will have to add another method callStudentServiceAndGetData\_Fallback with same signature,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

|  |
| --- |
| package com.example.howtodoinjava.springhystrixschoolservice.delegate;    import java.util.Date;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.context.annotation.Bean;  import org.springframework.core.ParameterizedTypeReference;  import org.springframework.http.HttpMethod;  import org.springframework.stereotype.Service;  import org.springframework.web.client.RestTemplate;  import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;    @Service  public class StudentServiceDelegate {        @Autowired      RestTemplate restTemplate;        @HystrixCommand(fallbackMethod = "callStudentServiceAndGetData\_Fallback")      public String callStudentServiceAndGetData(String schoolname) {            System.out.println("Getting School details for " + schoolname);            String response = restTemplate                  .exchange("<http://localhost:8098/getStudentDetailsForSchool/>{schoolname}"                  , HttpMethod.GET                  , null                  , new ParameterizedTypeReference<String>() {              }, schoolname).getBody();            System.out.println("Response Received as " + response + " -  " + new Date());            return "NORMAL FLOW !!! - School Name -  " + schoolname + " :::  " +                      " Student Details " + response + " -  " + new Date();      }        @SuppressWarnings("unused")      private String callStudentServiceAndGetData\_Fallback(String schoolname) {            System.out.println("Student Service is down!!! fallback route enabled...");            return "CIRCUIT BREAKER ENABLED!!! No Response From Student Service at this moment. " +                      " Service will be back shortly - " + new Date();      }        @Bean      public RestTemplate restTemplate() {          return new RestTemplate();      }  } |

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**.

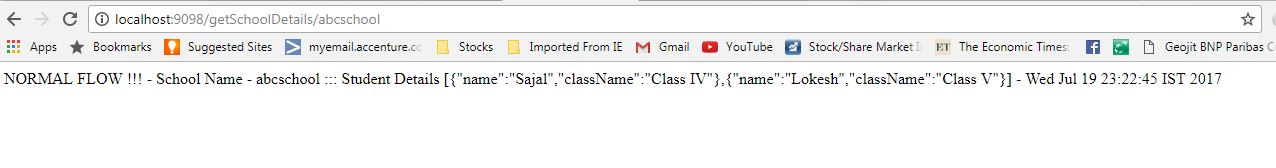
Start the student service as described above and then test school service by opening browser and type http://localhost:9098/getSchoolDetails/abcschool. It should show the below output in browser :

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/schoolserviceresponse.jpg)School Service Response

##### Test Hystrix Circuit Breaker – Demo

Opening browser and type http://localhost:9098/getSchoolDetails/abcschool.

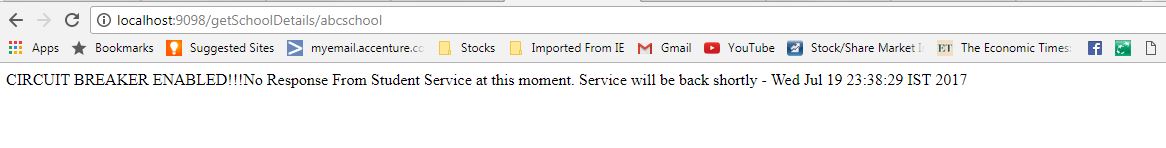
It should show the below output in browser –

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/schoolserviceresponse.jpg)School Service Response

Now we already know that School service is calling student service internally, and it is getting student details from that service. So if both the services are running, school service is displaying the data returned by student service as we have seen in the school service browser output above. This is **CIRCUIT CLOSED State**.

Now let us stop the student service by just pressing CTRL + C in the student service server console (stop the server) and test the school service again from browser. This time it will return the fall back method response. Here Hystrix comes into picture, it monitors Student service in frequent interval and as it is down, Hystrix component has opened the Circuit and fallback path enabled.

Here is the fall back output in the browser.

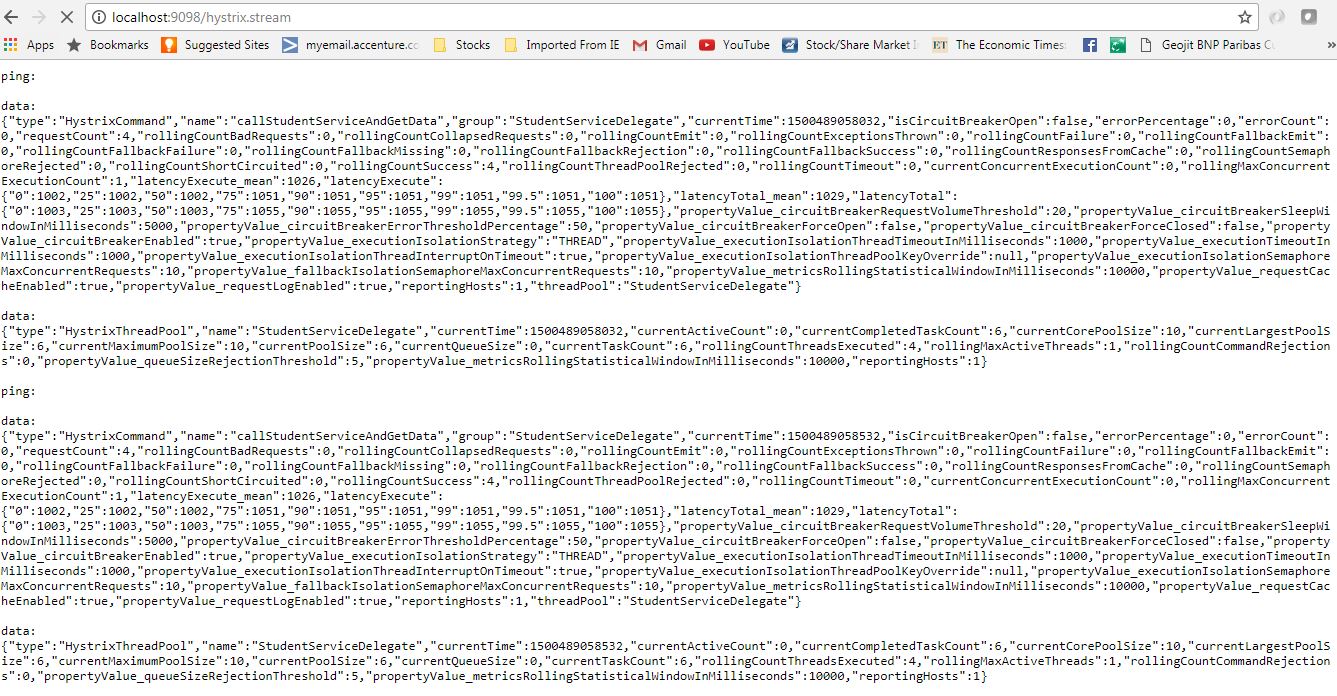
[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/schoolserviceresponse_fallback.jpg)School Service Response Fallback path

Again start the Student service, wait for few moments and go back to school service and it will again start responding in normal flow.

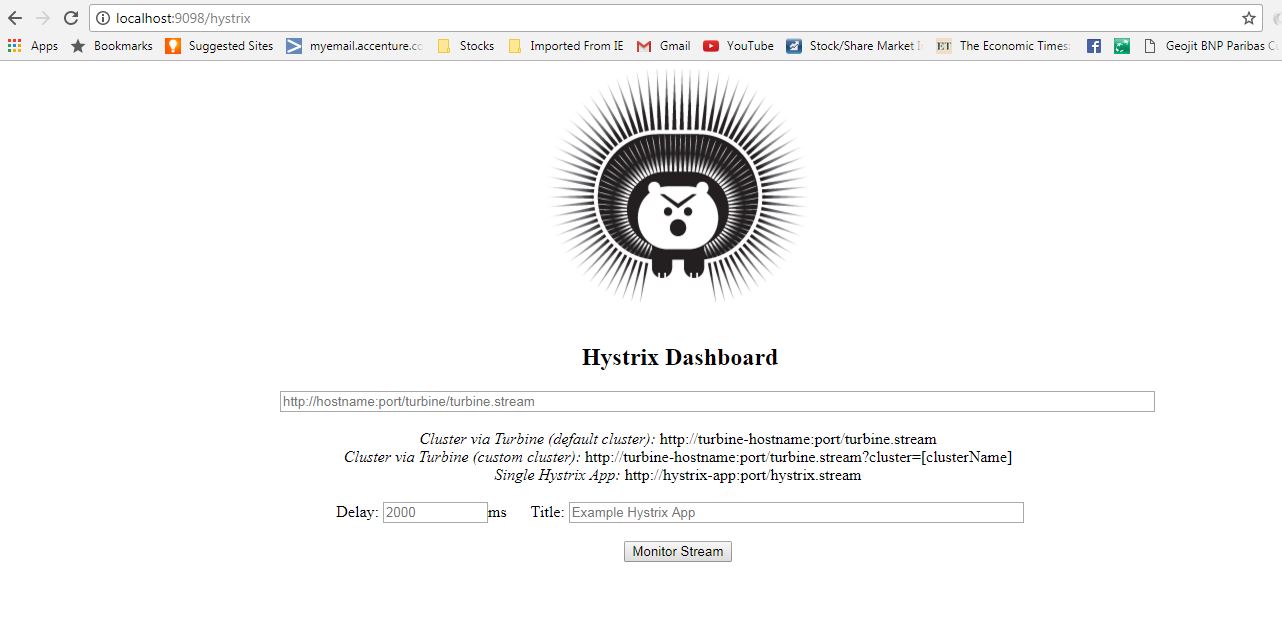
##### Hystrix Dashboard

As we have added hystrix dashboard dependency, hystrix has provided one nice Dashboard and a Hystrix Stream in the bellow URLS:

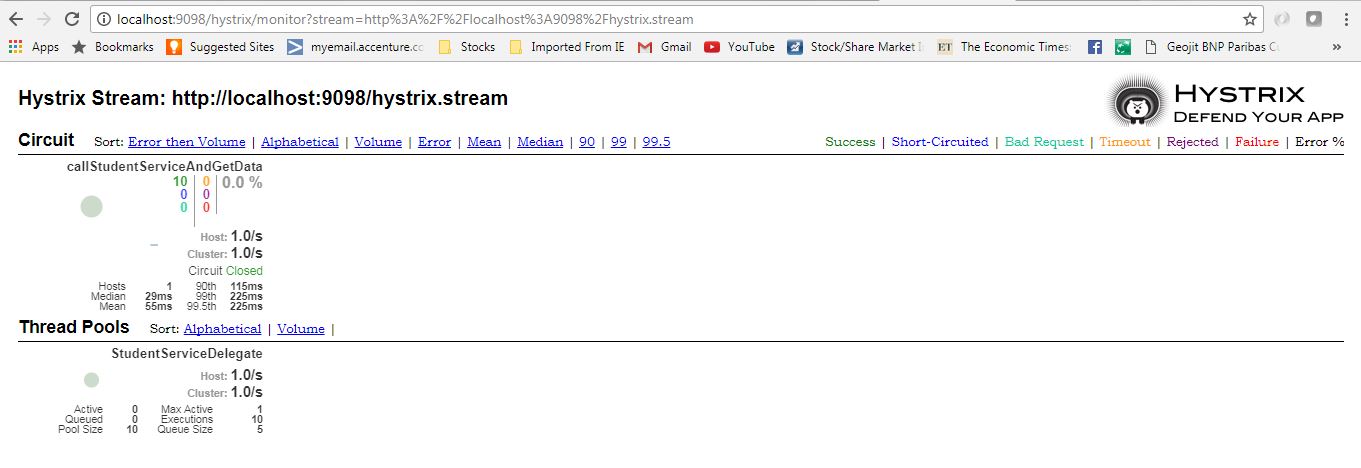
* **[http://localhost:9098/hystrix.stream](http://localhost:9098/hystrix.stream" \t "_blank)** – It’s a continuous stream that Hystrix generates. It is just a health check result along with all the service calls that are being monitored by Hystrix. Sample output will look like in browser –

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/HystrixStream.jpg)Hystrix Stream output

* **[http://localhost:9098/hystrix](http://localhost:9098/hystrix" \t "_blank)** – This is visual dashboard initial state.

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/Hystrix_initial.jpg)Hystrix Initial Dashboard

* Now add [http://localhost:9098/hystrix.stream](http://localhost:9098/hystrix.stream" \t "_blank) in dashboard to get a meaningful dynamic visual representation of the circuit being monitored by the Hystrix component. Visual Dashboard after providing the Stream input in the home page –

[](https://cdn1.howtodoinjava.com/wp-content/uploads/2017/07/HystrixDashboard.jpg)Hystrix visual Dashboard

##### Summary

That’s all about **creating spring could Hystrix Circuit Breaker**, we have tested both **circuit open path** and **circuit closed path**. Do the setup on your own and play with different combination service state to be more clear of whole concept.

#### **Zipkin and Sleuth -- Distributed log tracing**

[Zipkin](http://zipkin.io/" \t "_blank) is very efficient tool for **distributed tracing** in [microservices](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/" \t "_blank) ecosystem. Distributed tracing, in general, is latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business use case. Let’s say from our application, we have to call 4 different services/components for a transaction. Here with distributed tracing enabled, we can measure which component took how much time.

This is useful during debugging when lots of underlying systems are involved and the application becomes slow in any particular situation. In such case, we first need to identify see which underlying service is actually slow. Once the slow service is identified, we can work to fix that issue. Distributed tracing helps in identifying that slow component among in the ecosystem.

Table of Contents

[Zipkin](https://howtodoinjava.com/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/" \l "zipkin)

[Sleuth](https://howtodoinjava.com/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/" \l "sleuth)

[Zipkin and Sleuth Integration Example](https://howtodoinjava.com/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/" \l "integration)

[Demo](https://howtodoinjava.com/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/" \l "demo)

[Summary](https://howtodoinjava.com/spring-cloud/spring-cloud-zipkin-sleuth-tutorial/" \l "summary)

###### **Zipkin**

Zipkin was originally developed at Twitter, based on a concept of a Google paper that described Google’s internally-built distributed app debugger – [dapper](http://research.google.com/pubs/pub36356.html). It manages both the collection and lookup of this data. To use Zipkin, applications are instrumented to report timing data to it.

If you are troubleshooting latency problems or errors in ecosystem, you can filter or sort all traces based on the application, length of trace, annotation, or timestamp. By analysing these traces, you can decide which components are not performing as per expectations, and you can fix them.

Internally it has 4 modules –

1. **Collector** – Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.
2. **Storage** – This module store and index the lookup data in backend. [Cassandra](https://cassandra.apache.org/" \t "_blank), [Elasticsearch](https://www.elastic.co/" \t "_blank) and [MySQL](https://howtodoinjava.com/mysql/how-to-installuninstallexecute-mysql-as-windows-service/) are supported.
3. **Search** – This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.
4. **Web UI** – A very nice UI interface for viewing traces.

How to install Zipkin

Detailed installation steps can be found for different operating system including [Docker](https://howtodoinjava.com/cloud/docker-hello-world-example/) image at [quickstart page](http://zipkin.io/pages/quickstart.html" \t "_blank). For windows installation, just download the latest Zipkin server from [maven repository](https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec" \t "_blank) and run the [executable jar](https://howtodoinjava.com/maven/maven-shade-plugin-create-uberfat-jar-example/) file using below command.

|  |
| --- |
| java -jar zipkin-server-1.30.3-exec.jar |

Once Zipkin is started, we can see the Web UI at [http://localhost:9411/zipkin/](http://localhost:9411/zipkin/" \t "_blank).

Above command will start the Zipkin server with default configuration. For advanced configuration, we can configure many other things like storage, collector listeners etc.

To **install Zipkin in spring boot application**, we need to add Zipkin starter dependency in spring boot project.

|  |
| --- |
| <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-zipkin</artifactId>  </dependency> |

###### **Sleuth**

[Sleuth](https://cloud.spring.io/spring-cloud-sleuth/" \t "_blank) is a tool from Spring cloud family. It is used to generate the *trace id*, *span id* and add these information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and [ELK](https://howtodoinjava.com/microservices/elk-stack-tutorial-example/) etc. to store, index and process log files. As it is from spring cloud family, once added to the CLASSPATH, it automatically integrated to the common communication channels like –

* requests made with the [RestTemplate](https://howtodoinjava.com/spring/spring-restful/spring-restful-client-resttemplate-example/) etc.
* requests that pass through a [Netflix Zuul](https://howtodoinjava.com/spring/spring-cloud/spring-cloud-api-gateway-zuul/) micro proxy
* HTTP headers received at [Spring MVC](https://howtodoinjava.com/spring-mvc-tutorial/) controllers
* requests over messaging technologies like Apache Kafka or RabbitMQ etc.

If you have, let’s say 3 services, A, B and C. We made three different requests.

One request went from A → B, another from A →B →C, and last one went from B →C.

A -> B  
A -> B -> C  
B -> C

As the number of microservices grow, tracing requests that propagate from one microservice to another and figure out how a requests travels through the application can be quite daunting.

**Sleuth** makes it possible to trace the requests by adding unique ids to logs.

A trace id (1st) is used for tracking across the microservices; represents the whole journey of a request across all the microservices, while span id (2nd) is used for tracking within the individual microservice.

Using Sleuth is very easy. We just need to add it’s started pom in the spring boot project. It will add the Sleuth to project and so in its runtime.

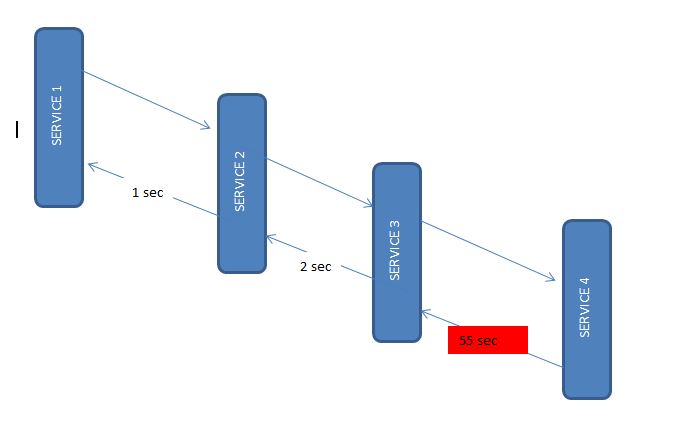
|  |
| --- |
| <dependency>      <groupId>org.springframework.cloud</groupId>      <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency> |

So far we have integrated Zipkin and Sleuth to microservices and ran Zipkin server. Let’s see how to utilize this setup.

###### **Zipkin and Sleuth Integration Example**

For this demo, lets create 4 spring boot based microservices. They all will have both Zipkin and Sleuth starter dependencies. In each microservice, we will expose one endpoint and from the first service we will call second service, and from second service we will invoke third and so on using rest Template.

And as we have already mentioned, Sleuth works automatically with rest template so it would send this instrumented service call information to attached Zipkin server. Zipkin will then start the book keeping of latency calculation along with few other statistics like service call details.

Microservices Interactions

###### Create Microservice

All the four services will have the same configuration, only difference is the service invocation details where the endpoint changes. Let’s [create Spring boot applications](https://howtodoinjava.com/spring/spring-boot/spring-boot-tutorial-with-hello-world-example/) with *Web*, *Rest Repository*, *Zipkin* and *Sleuth* dependencies.

I have packaged those services inside a parent project so that those four services can be build together to save time. You can proceed with individual set up if you wish to. Also I have added useful windows scripts so start/stop all the services with a single command.

This is one sample rest controller which exposes one endpoint and also invokes one downstream service using rest template.

|  |
| --- |
| package com.example.zipkinservice1;    import org.apache.log4j.Logger;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.sleuth.sampler.AlwaysSampler;  import org.springframework.context.annotation.Bean;  import org.springframework.core.ParameterizedTypeReference;  import org.springframework.http.HttpMethod;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;    @SpringBootApplication  public class ZipkinService1Application {        public static void main(String[] args) {          SpringApplication.run(ZipkinService1Application.class, args);      }  }    @RestController  class ZipkinController{        @Autowired      RestTemplate restTemplate;        @Bean      public RestTemplate getRestTemplate() {          return new RestTemplate();      }        @Bean      public AlwaysSampler alwaysSampler() {          return new AlwaysSampler();      }        private static final Logger LOG = Logger.getLogger(ZipkinController.class.getName());        @GetMapping(value="/zipkin")      public String zipkinService1()      {          LOG.info("Inside zipkinService 1..");             String response = (String) restTemplate.exchange("<http://localhost:8082/zipkin2>",                          HttpMethod.GET, null, new ParameterizedTypeReference<String>() {}).getBody();          return "Hi...";      }  } |

###### App Configurations

As all services will run in a single machine, so we need to run them in different ports. Also to identify in Zipkin, we need to give proper names. So configure application name and port information in application.properties file under resources folder.

|  |
| --- |
| server.port = 8081  spring.application.name = zipkin-server1 |

Similarly for other 3 services, we will use ports **8082**, **8083**, **8084** and name will also be like **zipkin-server2**, **zipkin-server3** and **zipkin-server4**.

Also we have intentionally introduced a delay in the last service so that we can view that in Zipkin.

#### **ELK (Elasticsearch, Logstash, Kibana) - [Managing, Searching, and Visualizing Logs](https://medium.com/oneclicklabs-io/streaming-spring-boot-application-logs-to-elk-stack-part-1-a68bd7cccaeb)**

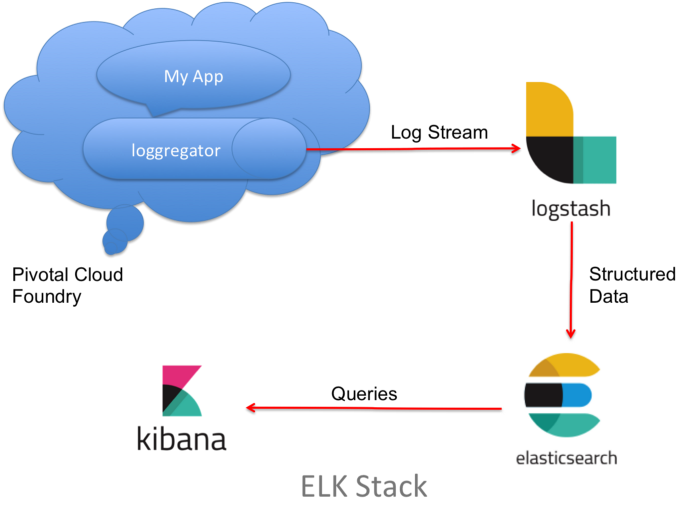


Logstash is a tool for managing logs. It supports virtually any type of log, including system logs, error logs, and custom application logs. It can receive logs from numerous sources, including syslog, messaging (for example, rabbitmq), and jmx, and it can output data in a variety of ways, including email, websockets, and to Elasticsearch.

Elasticsearch is a full-text, real-time search and analytics engine that stores the log data indexed by Logstash. It is built on the Apache Lucene search engine library and exposes data through REST and Java APIs. Elasticsearch is scalable and is built to be used by distributed systems.

Kibana is a web-based graphical interface for searching, analyzing, and visualizing log data stored in the Elasticsearch indices. It utilizes the REST interface of Elasticsearch to retrieve the data, and not only enables users to create customized dashboard views of their data, but also allows them to query and filter the data in an ad hoc manner.

The following image illustrates how the ELK Stack components are used to collect log data



##### Install & Configure Logstash

###### Install

* Download Logstash zip from <https://www.elastic.co/downloads/logstash>
* Extract it (unzip it)

wget <https://artifacts.elastic.co/downloads/logstash/logstash-5.1.1.zip>unzip logstash-5.1.1.zip

###### Configuration

Typical Logstash config file consists of three main sections: input, filter and output. Each section contains plugins that do relevant part of the processing.

Create a log.conf file in the root directory of the Logstash installation and copy the following code into it

input {  
 tcp {  
 port => 9600  
 type => syslog  
 }  
 udp {  
 port => 9600  
 type => syslog  
 }}filter {  
grok {  
 match => [ "message", "%{GREEDYDATA}" ]  
 }   
}output {  
 elasticsearch {  
 hosts => ["localhost:9200"]  
 index => "logstash-%{+YYYY.MM.dd}"  
 workers => 1  
 }  
}

**Input section**

Input section defines from where Logstash will read input data — in our case it will be a file hence we will use a file plugin with multiline codec, which basically means that our input file may have multiple lines per log entry.

input {  
 tcp {  
 port => 9600  
 type => syslog  
 }  
 udp {  
 port => 9600  
 type => syslog  
 }}

**Filter section**

Filter section contains plugins that perform intermediary processing on an a log event. In our case, event will either be a single log line or multiline log event grouped according to the rules described above. In the filter section we will do several things:

* Tag a log event if it contains a stacktrace. This will be useful when searching for exceptions later on.
* Parse out (or grok, in logstash terminology) timestamp, log level, pid, thread, class name (logger actually) and log message.
* Specified timestamp field and format — Kibana will use that later for time based searches.

Filter section for Spring Boot’s log format that aforementioned things looks like this:

filter {  
grok {  
 match => [ "message", "%{GREEDYDATA}" ]  
 }   
}

**Output section**

Output section contains output plugins that send event data to a particular destination. Outputs are the final stage in the event pipeline. We will be sending our log events to stdout (console output, for debugging) and to Elasticsearch.

Compared to filter section, output section is rather straightforward:

output {  
 elasticsearch {  
 hosts => ["localhost:9200"]  
 index => "logstash-%{+YYYY.MM.dd}"  
 workers => 2  
 }  
}

Finally, the three parts — input, filter and output — need to be copy pasted together and saved into logstash.conf config file. Once the config file is in place and Elasticsearch is running, we can run Logstash:

bin/logstash -f logstash.conf

If everything went well, Logstash is now shipping log events to Elasticsearch.

##### Install Elasticsearch

* Download elasticsearch zip file from <https://www.elastic.co/downloads/elasticsearch>
* Extract it to a directory (unzip it)
* Run it (bin/elasticsearch or bin/elasticsearch.bat on Windows)
* Check that it runs using curl -XGET [http://localhost:9200](http://localhost:9200/)

Here’s how to do it (steps are written for OS X but should be similar on other systems):

wget <https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-5.1.1.zip>

unzip elasticsearch-5.1.1.zip  
**cd** elasticsearch-5.1.1bin/elasticsearch

Elasticsearch should be running now. You can verify it’s running using:

[http://localhost:9200](http://localhost:9200/)

If all is well, you should get the following result:



##### Install Kibana

* Download Kibana archive from <https://www.elastic.co/downloads/kibana>
* Please note that you need to download appropriate distribution for your OS, URL given in examples below is for OS X
* Extract the archive
* Run it (bin/kibana)
* Check that it runs by pointing the browser to the Kibana’s WebUI

Here’s how to do it:

wget https://artifacts.elastic.co/downloads/kibana/kibana-5.1.1-darwin-x86\_64.tar.gztar xvzf kibana-5.1.1-darwin-x86\_64.tar.gz**cd** kibana-5.1.1-darwin-x86\_64bin/kibana

Kibana should be running

[http://localhost:5601](http://localhost:5601/)

First, you need to point Kibana to Elasticsearch index(s) of your choice. Logstash creates indices with the name pattern of logstash-YYYY.MM.DD. In Kibana Settings → Indices configure the indices:

* Index contains time-based events (select this option)
* Use event times to create index names (select this option)
* Index pattern interval: Daily
* Index name or pattern: [logstash-]YYYY.MM.DD
* Click on “Create Index”



##### Create log-drain service in PCF

Create a user-provided log draining service and bind the service to an application. The configuration above tells logstash to listen on port 9600, so the user-provided service creation and binding might look something like this:

$ cf cups logstash-drain -l syslog://**[**logstashserver**]**:9600$ cf bind-service **[**app-name**]** logstash-drain$ cf restart **[**app-name**]**

where [logstashserver] is the name or IP address of the server where logstash is running and [app-name] is the name of an application running on Cloud Foundry.

# **Spring Rest Boot Application**

1. **eureka-server:** This application is developed to register all the microservices. So that every microservice can communicate with each other without knowing their IP address of the microservice.
2. **gateway-service:** A gateway is a single-entry point into the system, used to handle requests by routing them to the corresponding microservice.
3. **member-mgmt-service:**
4. **donation-mgmt-service:**
5. **gateway-service:**

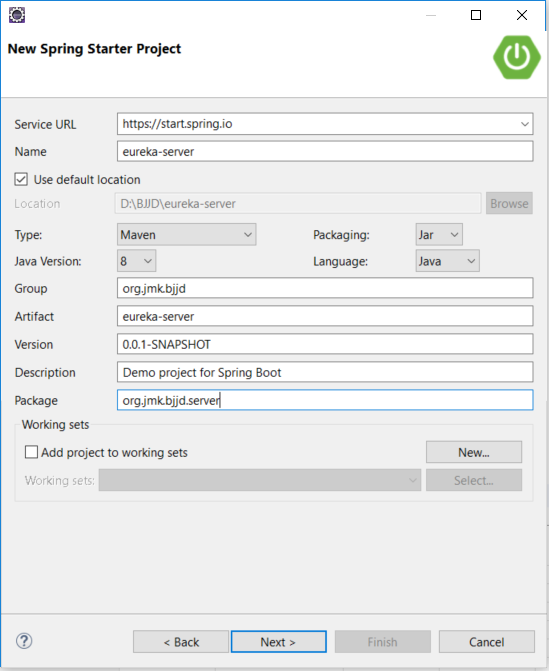
## **Eureka-server**

This application is developed to register all the microservices. So that every microservice can communicate with each other without knowing their IP address of the microservice.

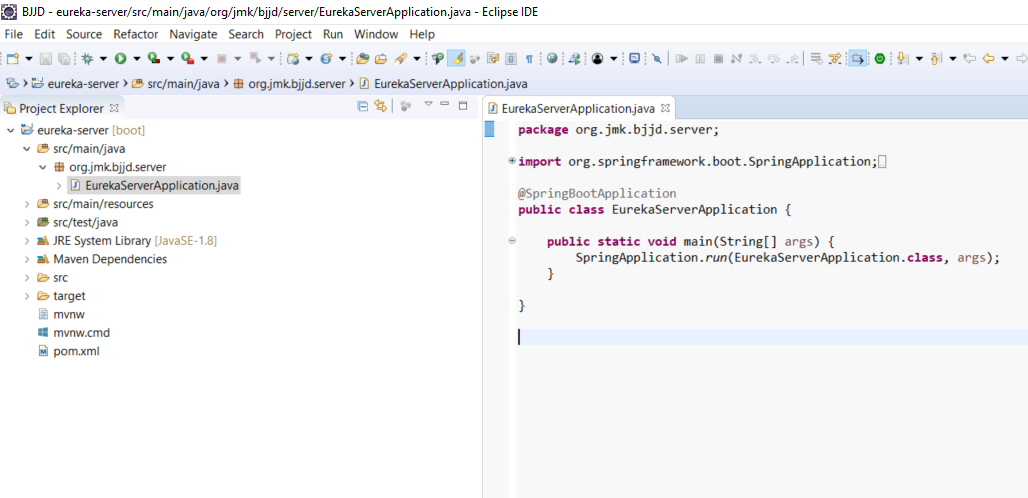
* Create Spring Boot Project



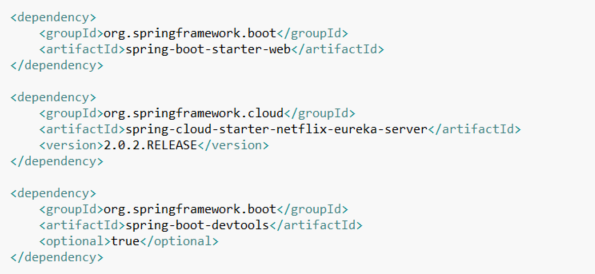
* Click on next



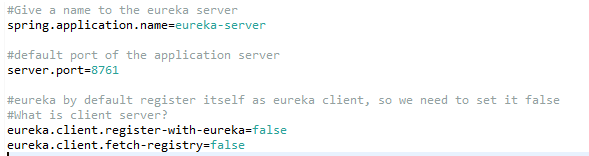
* Click on **Next** and **Finish** and created project will look as per the below screenshot:



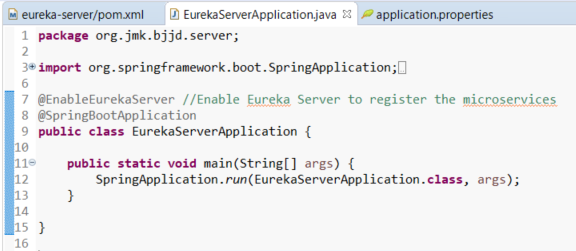
* Now need to update the **pom.xml** with the following dependencies:
  1. Web
  2. Eureka Server
  3. DevTools (optional)



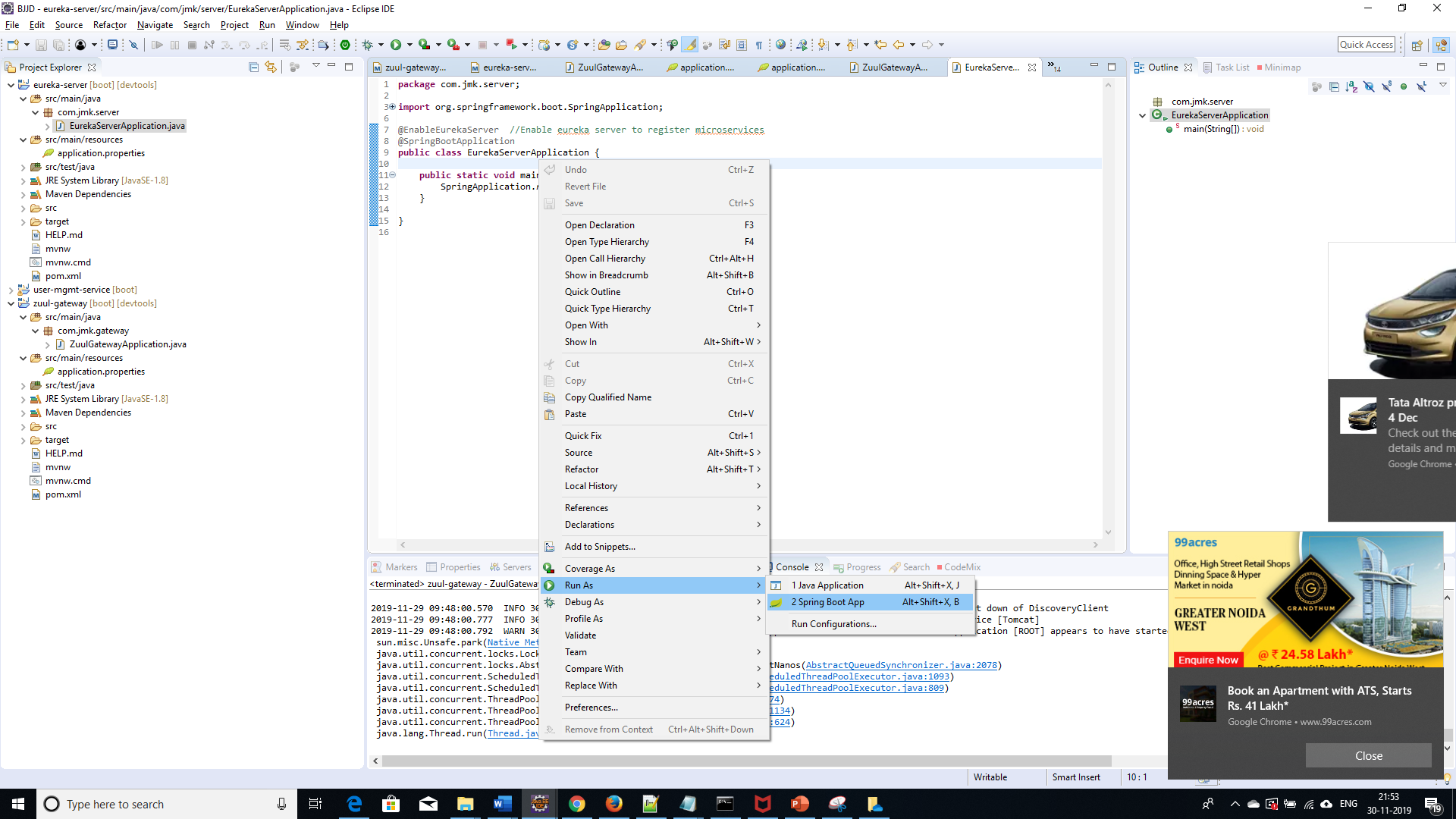
* Make changes in the **application.properties**.



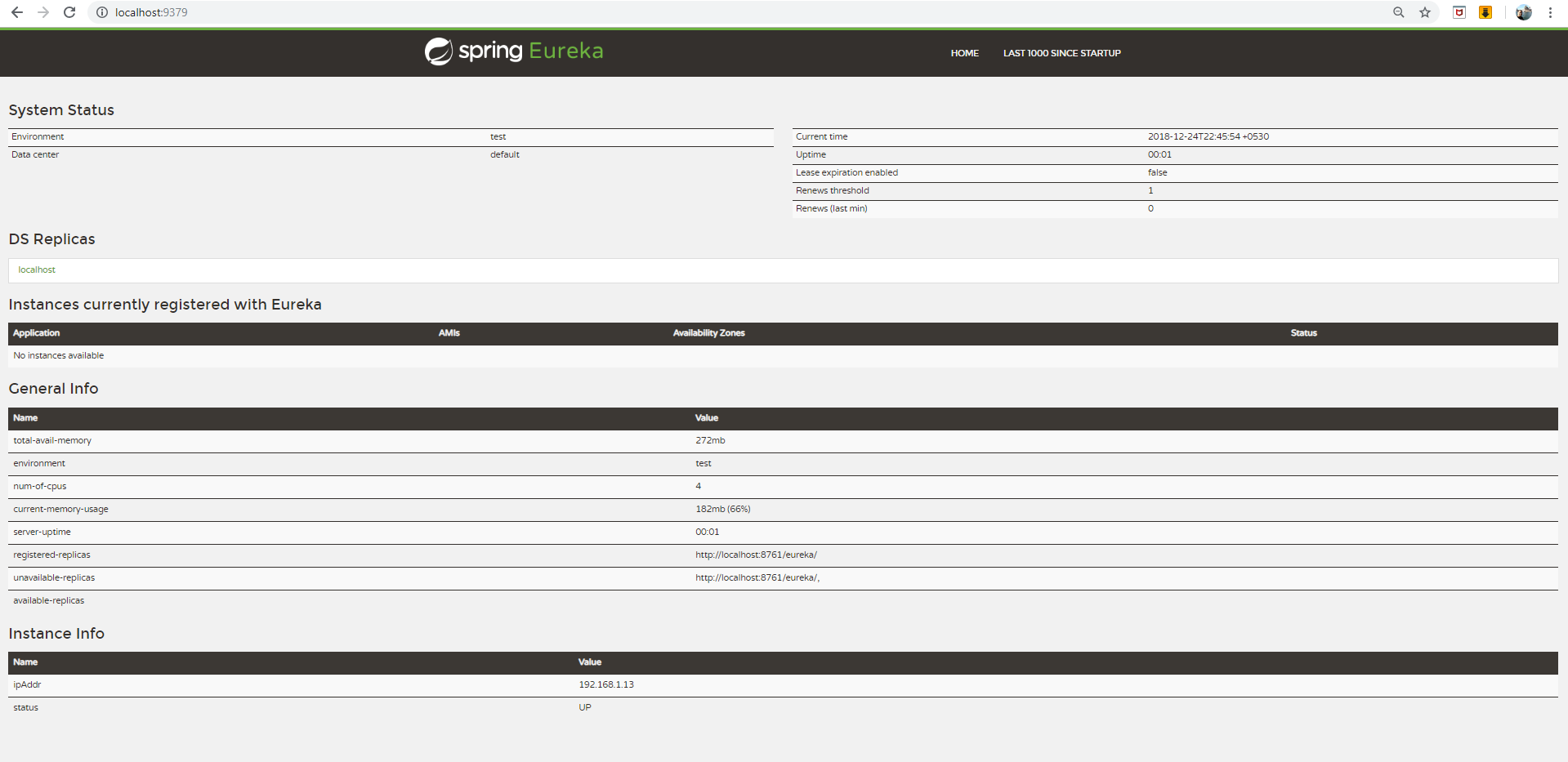
* Made changes in the **EurekaServerApplication** Class



* Run the application as **Spring Boot App**: EurekaServerApplication



* Open the browser and enter the url: <http://localhost:8761/>



## **Gateway-service(Zuul)**

A gateway is a single-entry point into the system, used to handle requests by routing them to the corresponding microservice.

It’s a proxy, gateway, an intermediate layer between the users and your services.

Eureka server solved the problem of giving names to services instead of hardcoding their IP addresses.

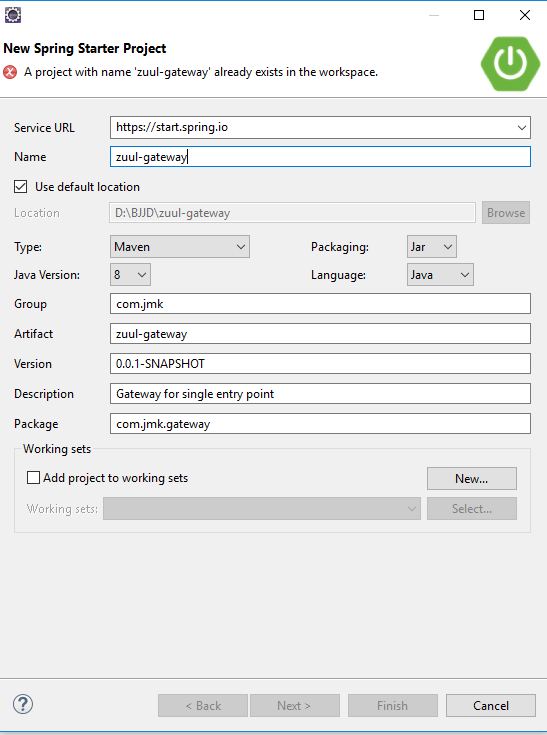
But, still, we may have more than one service (instances) running on different ports. So, Zuul …

1. Maps between a prefix path, say/gallery/\*\* and a service gallery-service. It uses Eureka server to route the requested service.
2. It loads balances (using Ribbon) between instances of a service running on different ports.
3. What else? We can filter requests, add authentication, etc.
4. *It’s worth mentioning that Zuul acts as a Eureka client. So, we give it a name, port, and link to Eureka server (same as we did with image service).*
5. Steps:

* Create Spring Boot Project



* Click on next and fill the required details shown in the below screen:

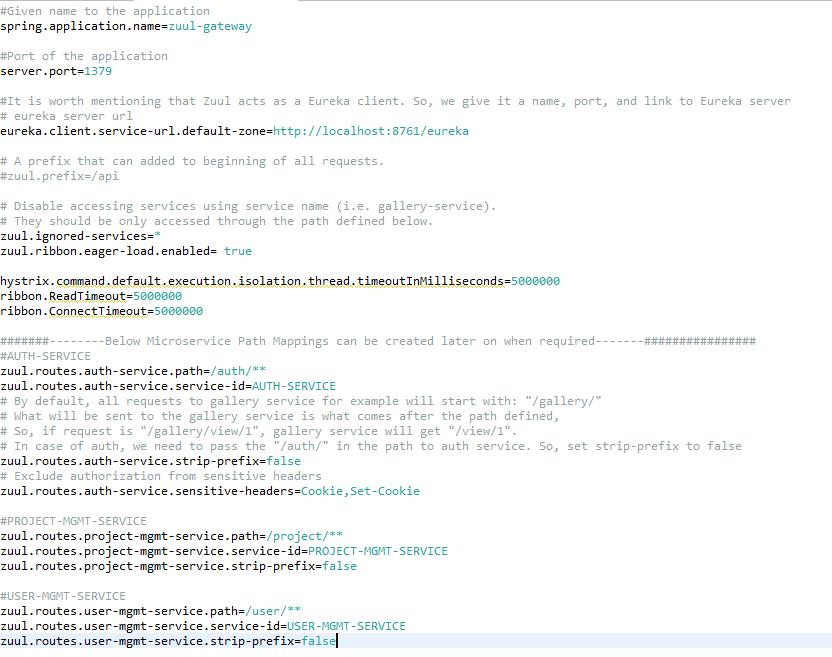


* In the pom.xml add dependencies:
  1. Web
  2. Eureka Client
  3. Zuul.

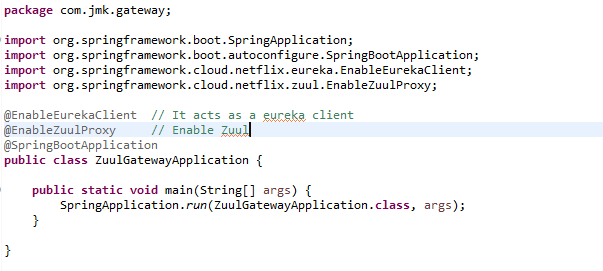
It’s worth mentioning that Zuul acts as a Eureka client. So, we give it a name, port, and link to Eureka server (same as we did with image service).



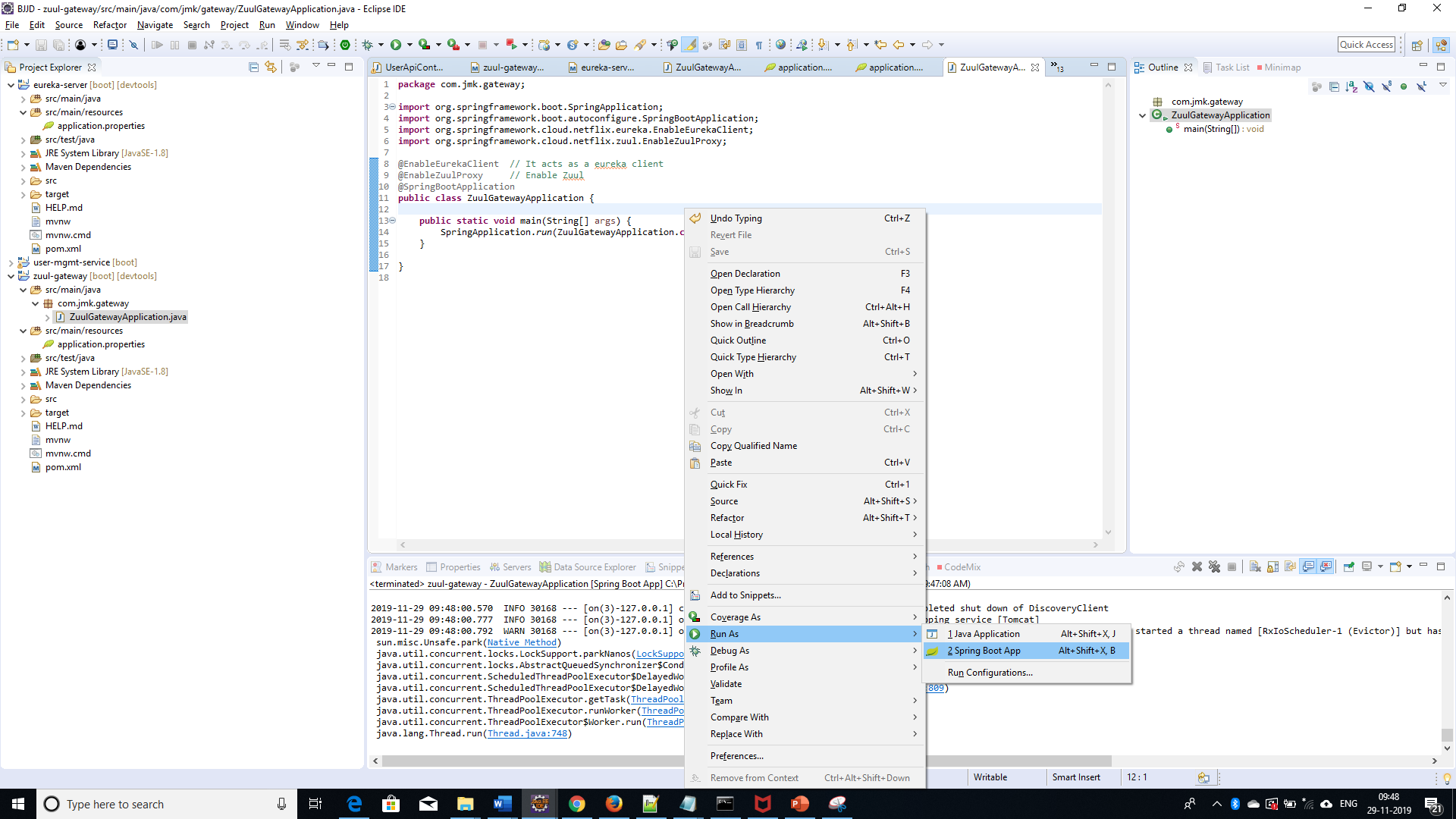
* Make changes in the **application.properties**.



* Finally, make the changes in the ZuulGatewayApplication to enable Zuul and Eureka Client.



* Run the application as **Spring Boot App** in the following order
  1. EurekaServerApplication
  2. ZuulGatewayApplication



* Open the browser of Eureka Server at localhost:8761, you should see the running zuul-gateway microservices and enter the url: <http://localhost:8761/>



## **User-Mgmt Service**

Create the user-mgmt-service application as microservice with the following functionalities:

1. Create the user:

While creating user, people-mgmt-service will call dependent microservice: **user-mgmt-service** to create the user credentials/information for every member so that member would be able to login into the BJJD application with his details)

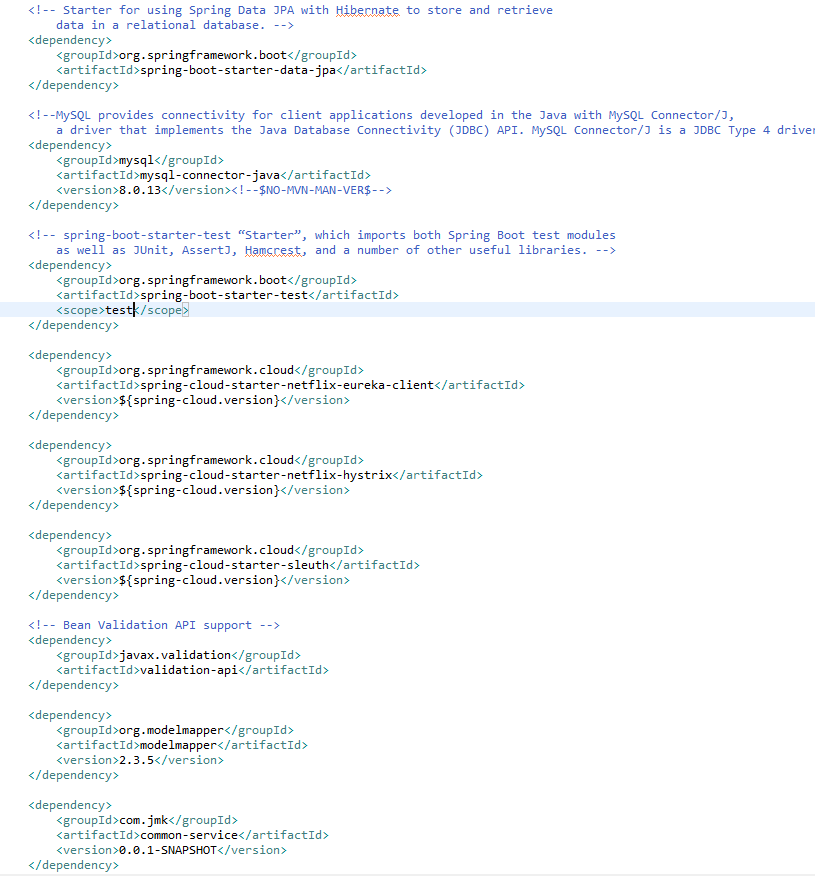
1. Update the user
2. Get the list of all the user
3. Delete the user
4. Get the user details
5. Follow the document to implement user-mgmt-service: **DemoProjectWork-member-mgmt-service.doc** along with the following changes:

|  |  |
| --- | --- |
| Database/Schema Name | users\_schema |
| User name | Users |
| Password | Users |

1. Use the following document related to the swagger, database scripts, ER diagram of Users:



1. Add the following dependencies



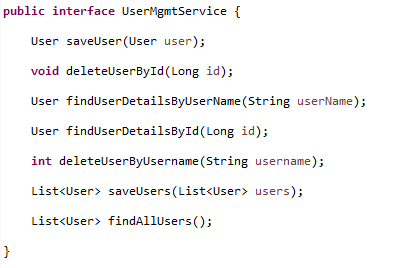
1. Rename the package of io.swagger to com.jmk.user
2. Rename and refactor the Swagger2SpringBoot.java to UserMgmtWebApplication.java with the following details:
   1. Enable Eureka Client so that it can register with Eureka Server
   2. Enable Circuit breaker
   3. Enable JpaRepositories



1. Update the application.properties to register with Eureka client



1. Service should be exposed as following:



1. Made changes for swagger documentation
   1. in the SwaggerDocumentationConfig, @EnableSwagger2 - Annotation to Enable Swagger Documentation on the API

@EnableSwagger2

@Configuration

**public** **class** SwaggerDocumentationConfig {

ApiInfo apiInfo() {

**return** **new** ApiInfoBuilder()

.title("User Managerment Service API")

.description("User Managerment Service API for BJJD")

.license("")

.licenseUrl("http://unlicense.org")

.termsOfServiceUrl("")

.version("18.10.0")

.contact(**new** Contact("","", ""))

.build();

}

@Bean

**public** Docket customImplementation(){

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.select()

.apis(RequestHandlerSelectors.*basePackage*("com.jmk.user.api")).paths(PathSelectors.*regex*("/user.\*"))

.build()

.directModelSubstitute(org.threeten.bp.LocalDate.**class**, java.sql.Date.**class**)

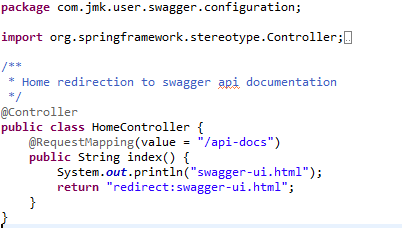
.directModelSubstitute(org.threeten.bp.OffsetDateTime.**class**, java.util.Date.**class**)

.apiInfo(apiInfo());

}

}

* 1. Made changes in the Swagger’s HomeController



1. Run the application as **Spring Boot App** in the following order
   1. EurekaServerApplication
   2. UserMgmtRestApplication

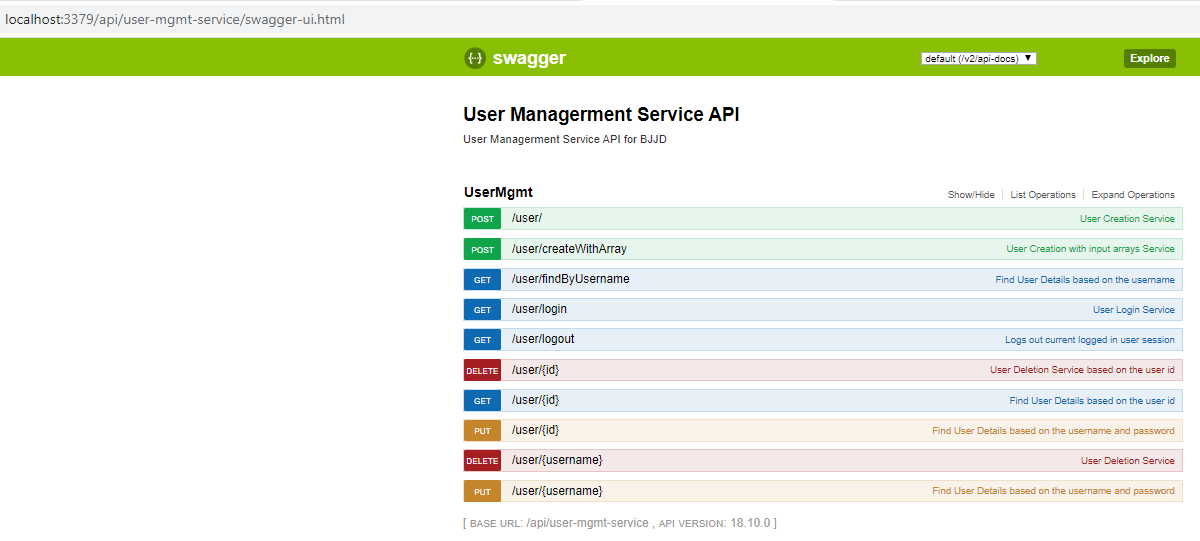
Check Eureka Server is running and UserMgmtServiceApplication is registered with Eureka Server: <http://localhost:8761/>



1. After running the application, should be visible following functions for the following url: <http://localhost:3379/api/user-mgmt-service/swagger-ui.html>

Or

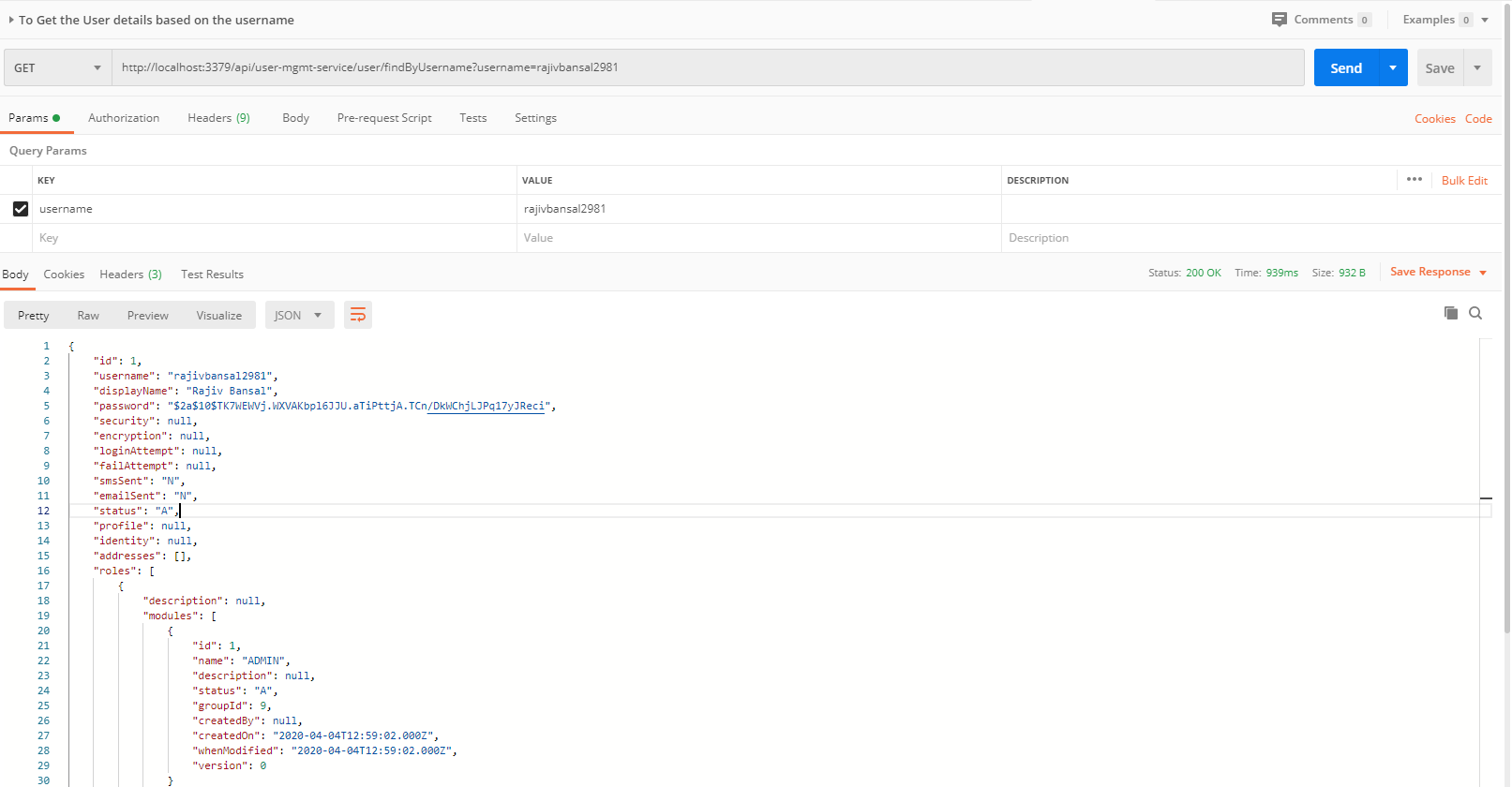
<http://localhost:3379/api/user-mgmt-service/api-docs>



1. Test the User Api using JUnit



1. Test using Postman



## **entication**

Authentication Service is used for validating user credentials, and issuing tokens.

The authentication flow is simple as:

* 1. The user sends a request to get a token passing his credentials.
  2. The server validates the credentials and sends back a token.
  3. With every request, the user has to provide the token, and server will validate that token.
* Validating the token in Authentication service for every request: Gateway Service has to call the auth service to validate user credentials and the issued token before allowing the requests to go to any service.
* Authentication Service Tools
  1. Spring Cloud Security
  2. OAuth2
  3. JWT
* Authentication service can call any user-mgmt-service to validate the user credentials with the DB or any other source.

We’ll introduce another service called ‘auth service’ for validating user credentials, and issuing tokens.

What about validating the token? Well, it can be implemented in the auth service itself, and the gateway has to call the auth service to validate the token before allowing the requests to go to any service.

Instead, we can validate the tokens at the gateway level, and let the auth service validate user credentials, and issue tokens. And that’s what we’re going to do here.

In both ways, we are blocking the requests unless it’s authenticated (except the requests for generating tokens).

**JSON Based Token (JWT)**

* A token is an encoded string, generated by our application (after being authenticated) and sent by the user along each request to allow access to the resources exposed by our application.
* JSON Based Token (JWT) is a JSON-based open standard for creating access tokens. It consists of three parts; header, payload, and signature.
* The header contains the hashing algorithm

{type: “JWT”, hash: “HS256”}

* The payload contains attributes (username, email, etc) and their values.

{username: "Omar", email: "omar@example.com", admin: true }

* The signature is hashing of: Header + “.” + Payload + Secret key

**Steps:**

**To implement the Authentication Service there are two major steps :**

1. Need to change in the gateway-service(gateway-zuul) previously created project.
2. Create Authentication Service

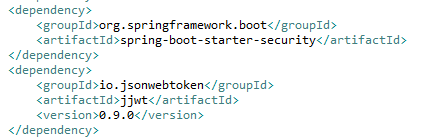
**I. Need to change in the gateway-service(gateway-zuul) previously created project**

In the gateway, we need to do two things:

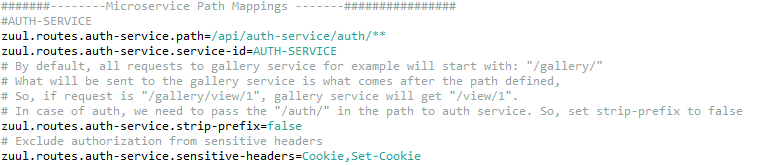
* 1. validate tokens with every request, and
  2. prevent all unauthenticated requests to our services. Fair enough?

**🡪Steps:**

* In the pom.xml add spring security and JWT dependencies.



* Made changes in the application.properties



* Three Java Classes need to be created
  1. SecurityTokenConfig
  2. **JwtTokenAuthenticationFilter**
  3. **JwtConfig**
  4. JwtAuthenticationEntryPoint – to handle an authorized attempts
* To define our security configurations, create a class, and annotated with @EnableWebSecurity, and extends WebSecurityConfigurerAdapter class to override and provide our own custom security configurations.



* Create the filter **JwtTokenAuthenticationFilter**

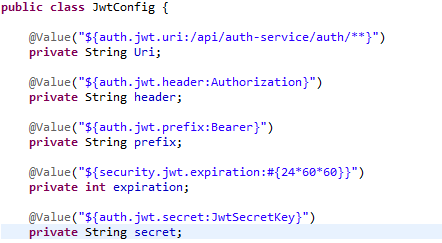
Spring has filters that will get executed within the life cycle of the request (filter chain). To enable and use these filters, we need to extend the class of any of these filters.

By default, spring will try to figure out when the filter should be executed. Otherwise, we can also define when should be executed (after or before another filter).

In this step, we implement our filter that validates the tokens. We’re using OncePerRequestFilter. It guarantee a single execution per request (since you can have a filter on the filter chain more than once).

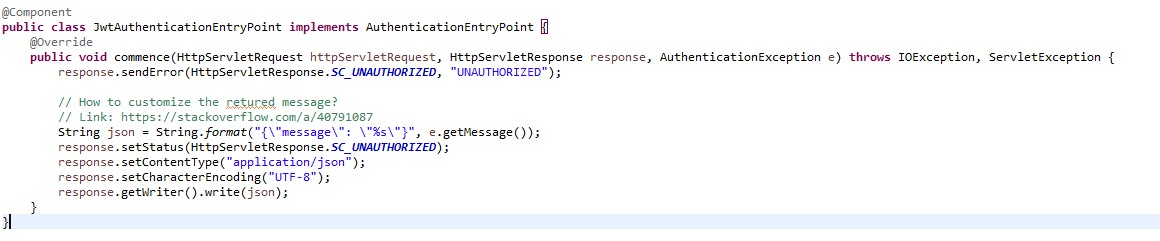


* Create **JwtConfig** Class first which contains configuration variables with getters and setters.



* Create **JwtAuthenticationEntryPoint** to handle the authorized attempt

This class is used to return a 401 unauthorized error to clients that try to access a protected resource without proper authentication. It implements Spring Security's AuthenticationEntryPoint interface.



1. **Create Authentication Service**

In the auth service, we need to

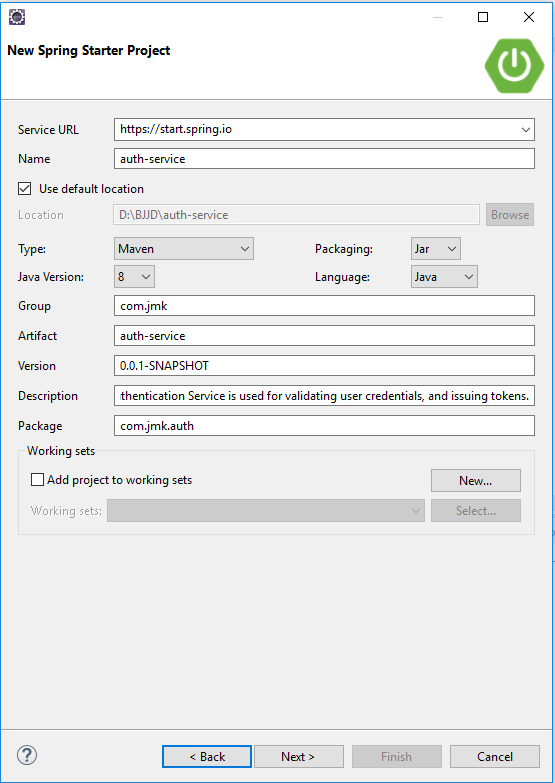
* + 1. Validate the user credentials, and if valid: by calling UserMgmtService using Fiegn Client
    2. generate a token, otherwise, throw an exception.

**🡪Steps:**

* Create Spring Boot Project



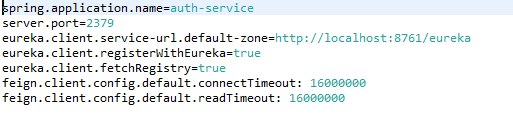
* Click on next and fill the required details shown in the below screen:



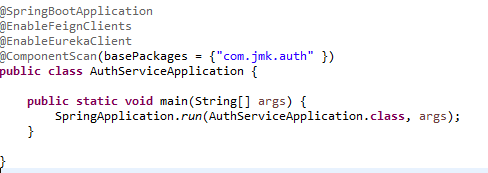
* In the pom.xml add dependencies:
  1. Web
  2. Eureka Client
  3. Spring Security
  4. JWT
  5. Spring Cloud Open Feign Client
  6. Common-service(user created service for reusing the enum, model and utility classes)



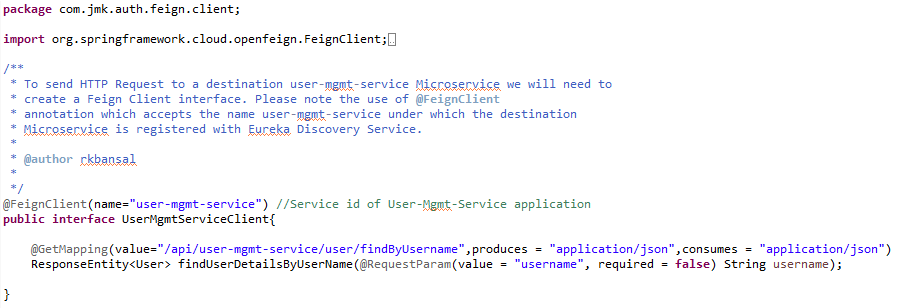
* Rename the application.properties file to application.yml and add the following configuration:



* Update the main **AuthServiceApplication** application with the following details:
  1. Enable Eureka Client so that it can register with Eureka Server
  2. Enable FeignClient to interact with user-mgmt-service
  3. Enable JpaRepositories



* Following Classes need to be created
  1. UserMgmtServiceClient – Feign Client for user-mgmt-service
  2. UserDetailsServiceImpl – It will call microservice : user-mgmt-service to validate the credentials using feign client created above.
  3. SecurityCredentialsConfig
  4. JwtUsernameAndPasswordAuthenticationFilter
  5. JwtConfig (same as already created in the above section)
* UserMgmtServiceClient- Feign Client for user-mgmt-service.
* Feign dynamically generates the implementation of the interface we created, so Feign has to know which service to call beforehand. That's why we need to give a name for the interface, which is the {Service-Id} of UserMgmtService. Now, Feign contacts the Eureka server with this Service Id, resolves the actual IP/hostname of the UserMgmtService, and calls the URL provided in Request Mapping.
* To send HTTP Request to a destination **user-mgmt-service** Microservice we will need to create a Feign Client interface. Please note the use of @FeignClient annotation which accepts the name user-mgmt-service under which the destination Microservice is registered with Eureka Discovery Service.



* UserDetailsServiceImpl –

It will implement UserDetailsService(Spring Security API) interface and will call microservice: user-mgmt-service to validate the credentials using feign client created above.

This class acts like a provider for the user; meaning it loads the user from the database (or any data source). It doesn’t do authentication. It just loads the user given his username.



* SecurityCredentialsConfig :

As we did in the Gateway for security configurations, create a class: SecurityCredentialsConfig, annotated with @EnableWebSecurity, and extends WebSecurityConfigurerAdapter



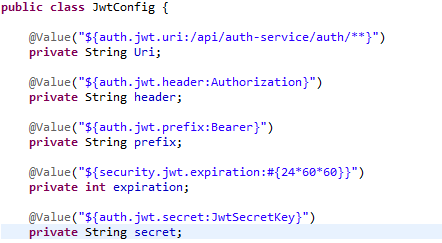
* JwtUsernameAndPasswordAuthenticationFilter

We’re using JwtUsernameAndPasswordAuthenticationFilter. It’s used to validate user credentials, and generate tokens. The username and password must be sent in a POST request.

NOTE: we can complete the following class with getter and setter of UserCredentials not showing in the below pic.



* Create **JwtConfig** Class first which contains configuration variables with getters and setters.

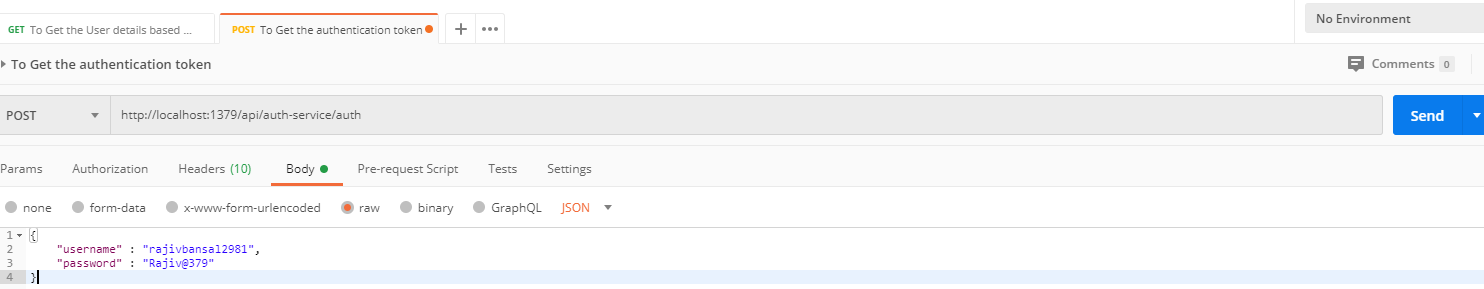


* Run the application as **Spring Boot App** in the following order
  1. EurekaServerApplication
  2. UserMgmtRestApplication
  3. AuthServiceApplication
  4. ZuulGatewayApplication
* Open the browser of Eureka Server at localhost:8761, you should see the all the microservices running and enter the url: <http://localhost:8761/>

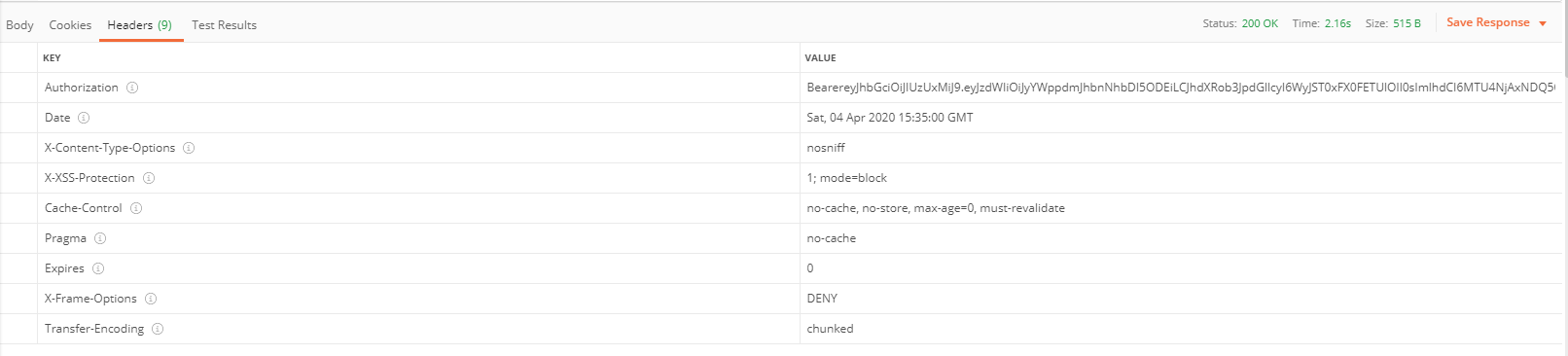


* To use any microservices, we have to go via zuul-gateway port: 1379.
  1. To get the token, user will pass his credentials and call auth service via zuul-gateway service.
  2. The Auth service validates the credentials and sends back a token.
  3. With every request, the user has to provide the token, and server will validate that token.

**Request**



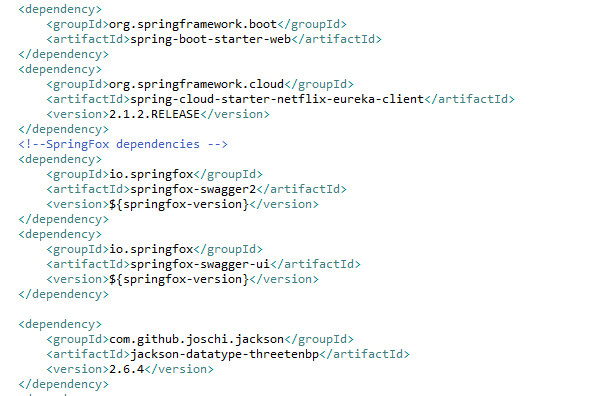
**Response –** see the response header containing the token in the Authorization attribute.



## **Common-Service**

When we have common configuration variables, classes, enum classes, or logic, used by multiple microservices, like the one we had JwtConfig, Status Enum, User Model related classes used in the Authentication Services and User Management Service.

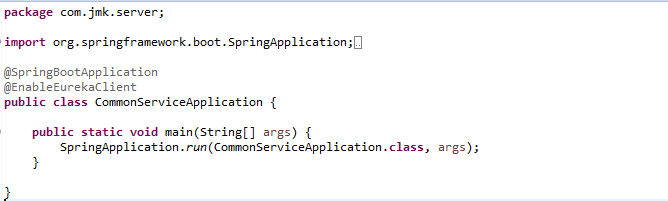
* To do so, just create a new project (service), call it ‘common-service’, and follow the same steps as we did with the image service. So, In pom.xml file



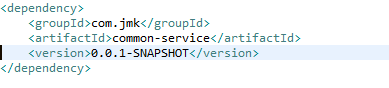
* In Application.properties



* Changes in the main application



* Put all the
  1. common configuration variables,
  2. classes, enum classes, or logic, used by multiple microservices,
  3. like the one we had JwtConfig, Status Enum, User Model related classes used in the Authentication Services and User Management Service
* Now, to be able to call JwtConfig class, User Model related classes from other services, like auth and gateway, we just need to add the common service in pom.xml as dependency.
* Add the dependency of common-service in auth-service pom.



* Now we can use the common configurations and classes.

## **Project-Mgmt Service**

Create the project-mgmt-service application as microservice with the following functionalities:

* Create the project:
* Update the project based on the project id
* Get the list of all the project based on the status
* Get the project based on the project id
* Delete the project based on the project id

1. Follow the document to implement project-mgmt-service: **DemoProjectWork-member-mgmt-service.doc** along with the following changes:

Commands:

create user 'project'@'%' identified by 'project';

create database project\_schema;

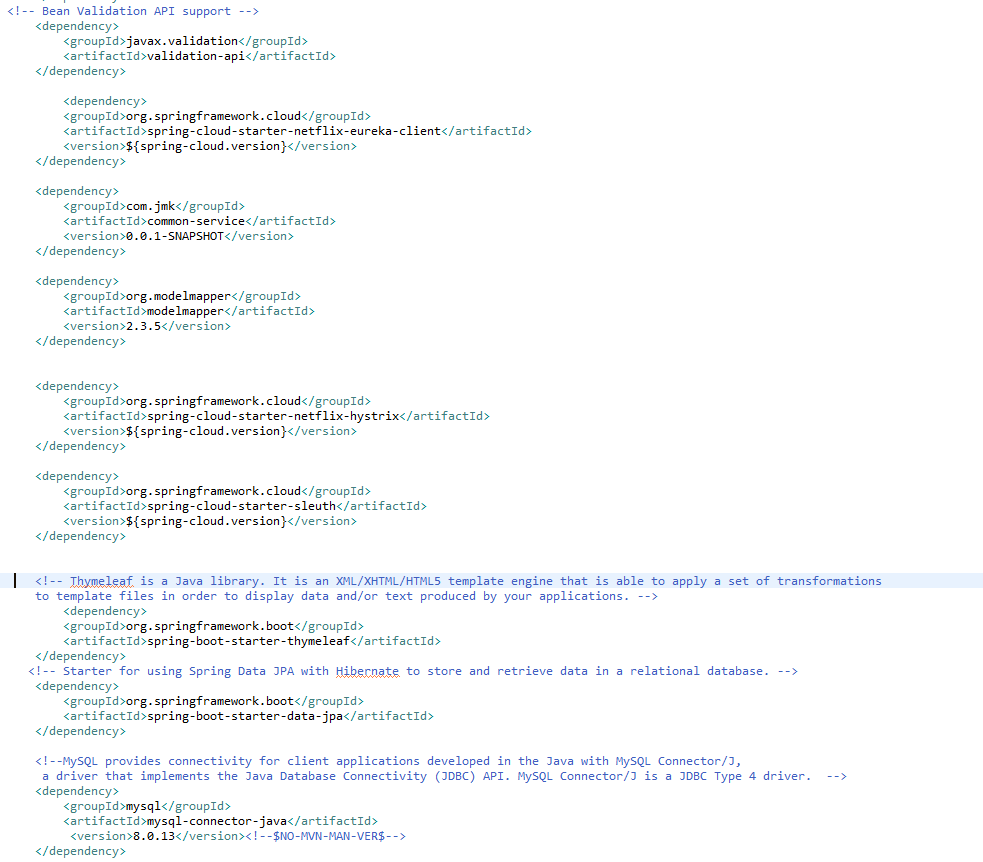
grant all on project\_schema.\* to project@'%';

|  |  |
| --- | --- |
| Database/Schema Name | project\_schema |
| User name | Project |
| Password | Project |

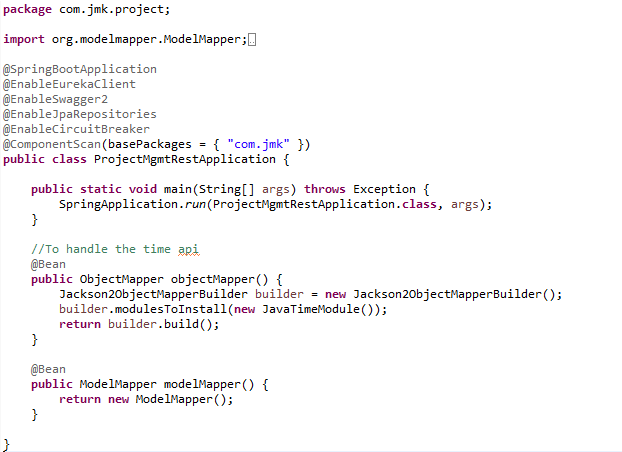
1. Use the following document related to the swagger, database scripts, ER diagram of Users:



1. Add the following dependencies



1. Rename the package of io.swagger to com.jmk.project
2. Rename and refactor the Swagger2SpringBoot.java to ProjectMgmtWebApplication.java with the following details:
   1. Enable Eureka Client so that it can be register with Eureka Server
   2. Enable Circuit breaker
   3. Enable JpaRepositories
   4. Enable swagger2 so that we can view the document



1. Create the **schema name**: project\_schema and **user**: project and **password**:project

Create the database named: root and user: bjjd using mysql command line or UI

User Id: root

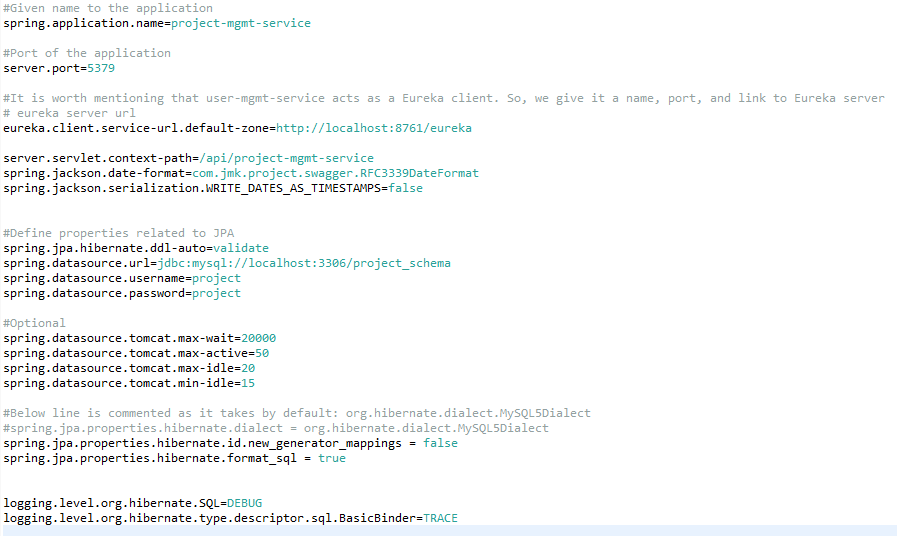
Password: rajiv999

create user 'project'@'%' identified by 'project';

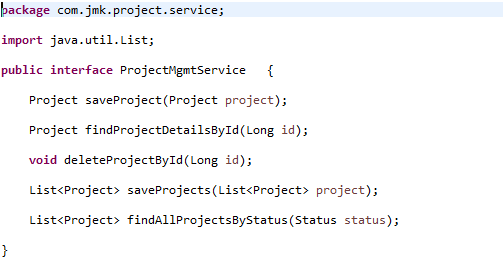
create database project\_schema;

grant all on project\_schema.\* to project@'%';

1. Update the application.properties to register with Eureka client



1. Service should be exposed as following:



1. Made changes in the SwaggerDocumentationConfig

@EnableSwagger2 - Annotation to Enable Swagger Documentation on the API

@EnableSwagger2

@Configuration

**public** **class** SwaggerDocumentationConfig {

ApiInfo apiInfo() {

**return** **new** ApiInfoBuilder()

.title("Project Management Service API")

.description("No description provided (generated by Swagger Codegen https://github.com/swagger-api/swagger-codegen)")

.license("")

.licenseUrl("http://unlicense.org")

.termsOfServiceUrl("")

.version("18.10.0")

.contact(**new** Contact("","", ""))

.build();

}

@Bean

**public** Docket customImplementation(){

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.select()

.apis(RequestHandlerSelectors.*basePackage*("com.jmk.project.api")).paths(PathSelectors.*regex*("/project.\*"))

.build()

.directModelSubstitute(org.threeten.bp.LocalDate.**class**, java.sql.Date.**class**)

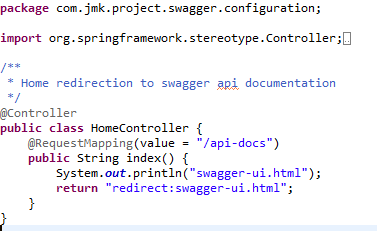
.directModelSubstitute(org.threeten.bp.OffsetDateTime.**class**, java.util.Date.**class**)

.apiInfo(apiInfo());

}

}

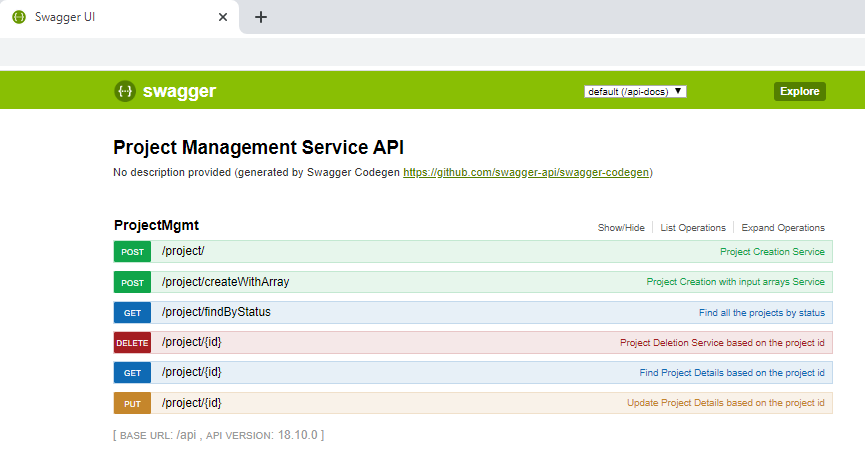
1. Made changes in the Swagger’s HomeController



1. After running the application, should be visible following functions for the following url: <http://localhost:5379/api/project-mgmt-service/swagger-ui.html>

Or

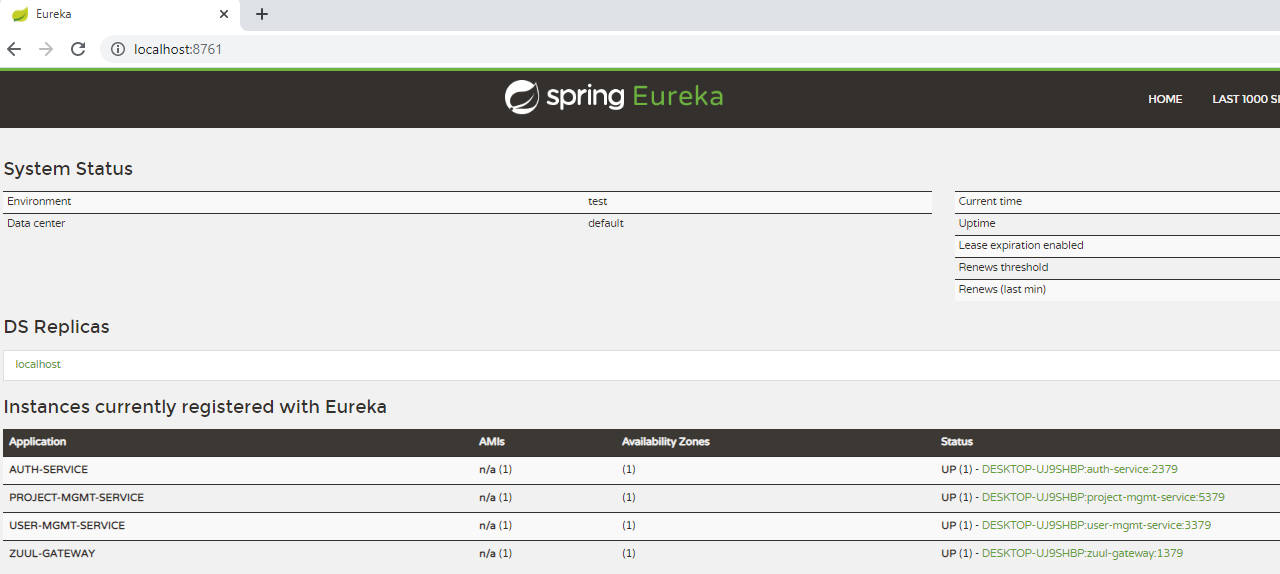
<http://localhost:5379/api/project-mgmt-service/api-docs>



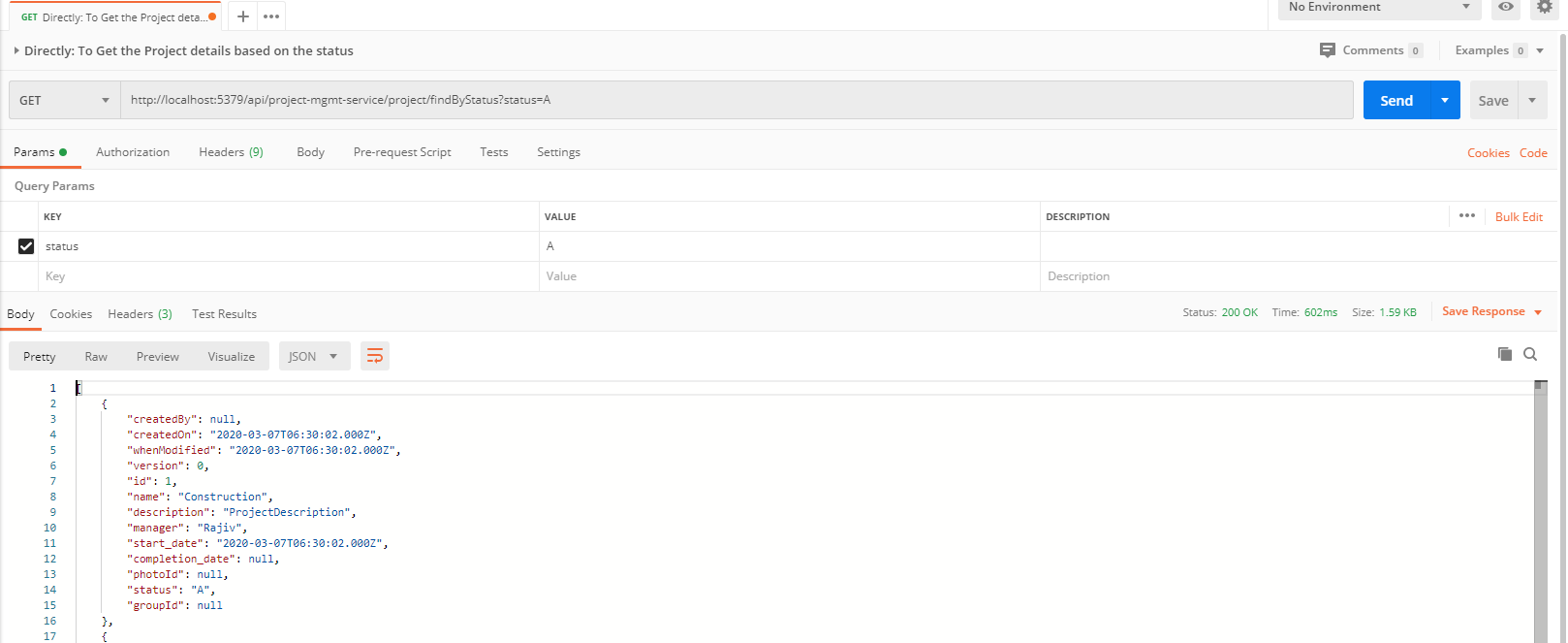
1. Test the Project Api using JUnit



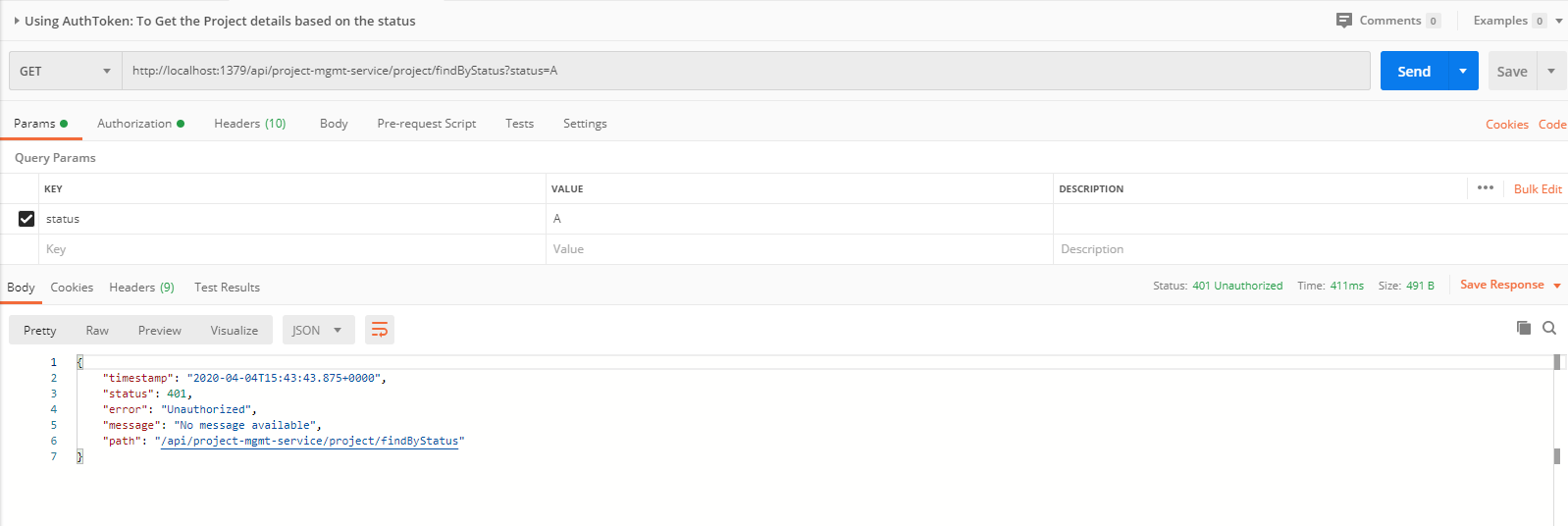
* Run the application as **Spring Boot App** in the following order
  1. EurekaServerApplication
  2. UserMgmtRestApplication
  3. AuthServiceApplication
  4. ZuulGatewayApplication
  5. ProjectMgmtRestApplication
* Open the browser of Eureka Server at localhost:8761, you should see the all the microservices running and enter the url: <http://localhost:8761/>



* Testing
  1. Without authentication means directly hitting the project-mgmt-project running on 5379
  2. With authentication means every request to project-mgmt-service microservice will be hit via gateway running 1379
     + without authentication means directly hitting the project-mgmt-project running on 5379

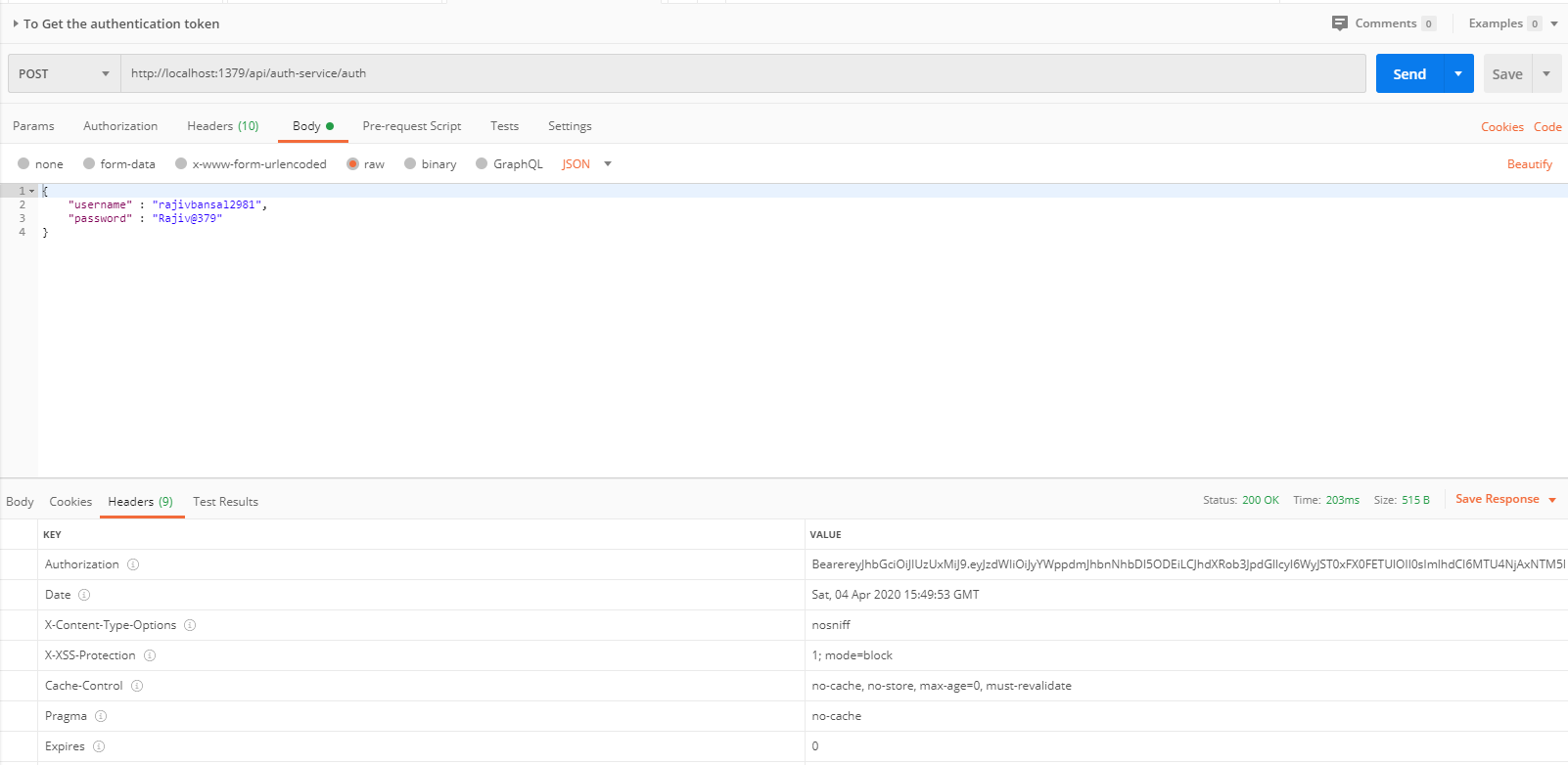


* + - with authentication means every request to project-mgmt-service microservice will be hit via gateway running on 1379
      * getting project details with status “A” but without token means no authorization code(Bearer token) in the header will give the error.

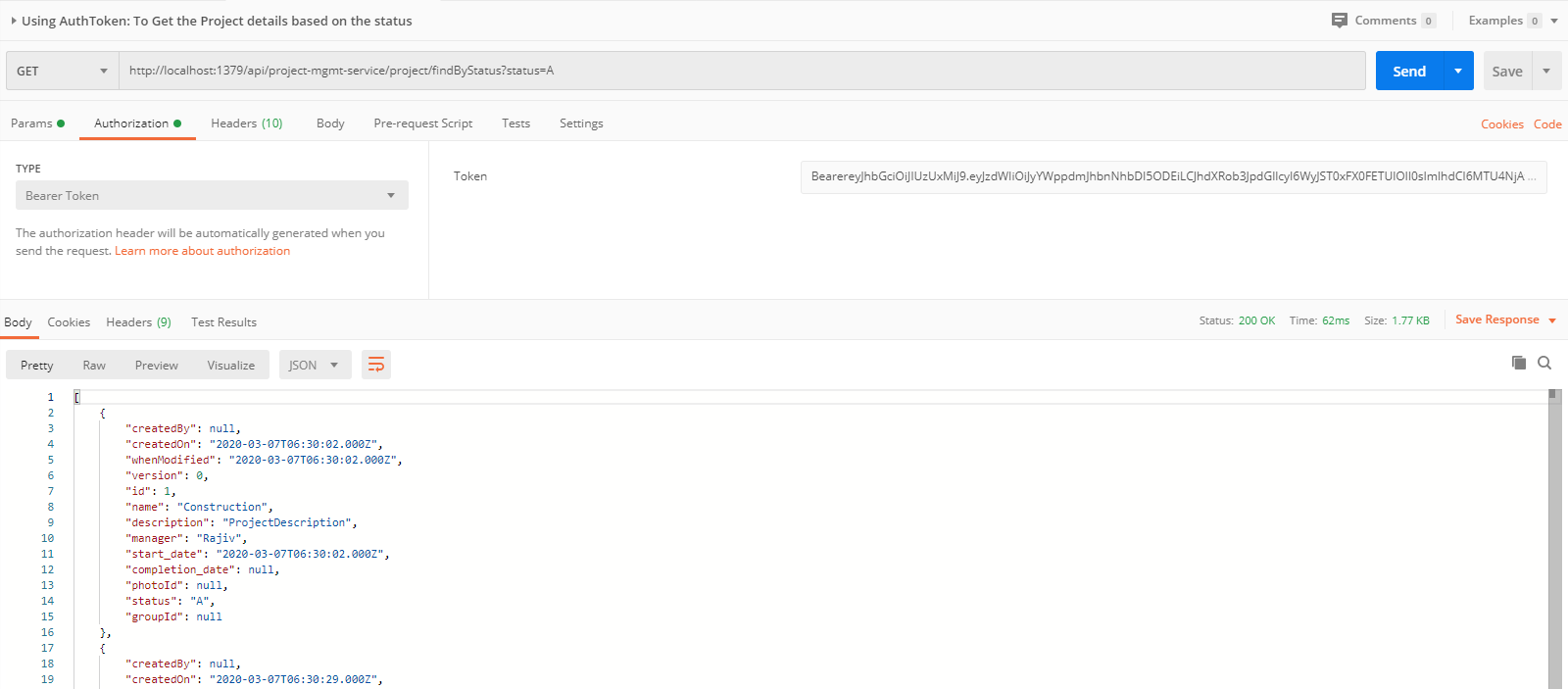


* getting project details with status “A” but with token means authorization code(Bearer token) in the header will be provided using authentication service

1. **To get authentication code in response header using authentication service**



1. **Now hit the project-mgmt-service to** get project details with status “A” by providing the token means authorization code(Bearer token) in the header received in above step



## **People-Mgmt-Service**

Create the member-mgmt-service application as microservice with the following functionalities:

1. Create the member:

While creating member, it will call dependent microservice: **user-mgmt-service** to create the user credentials/information for every member so that member would be able to login into the BJJD application with his details)

1. Update the member
2. Get the list of all the members
3. Delete the member
4. Get the member details
5. Check the member exists:

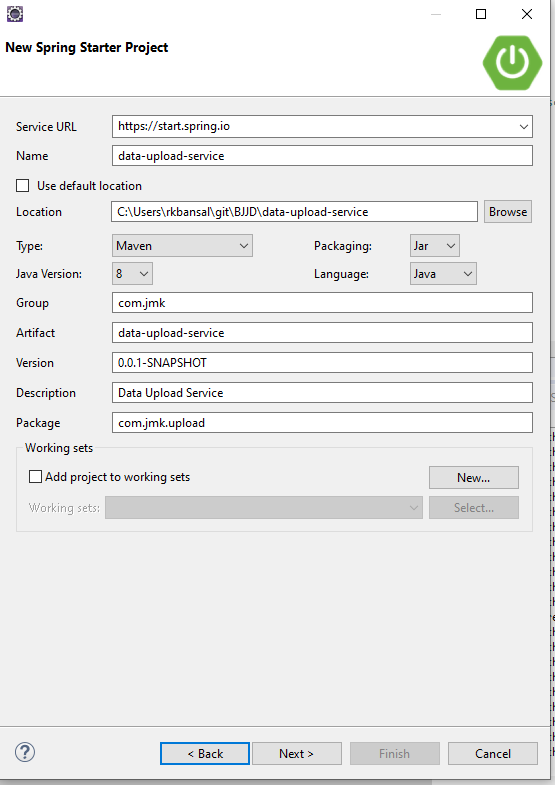
this method would be invoked from **expense-mgmt-service** and **donation-mgmt-service** to check the member exists or not.

After that made the following changes:

## **Data-Upload-Service**

This service will upload the data in bulk for Member, Devotee etc. in different format and call the appropriate microservice to save the data.

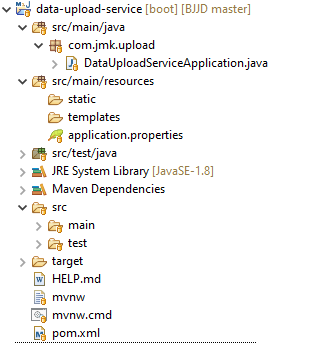
* Create the Project using spring Starter Project.



* Click on Next and select the Spring Web Module.

## 

* Click on Finish and import the project.



* Add the following dependencies in the pom.xml



* Configuring Server and File Storage Properties

**First thing First!**

* + Let’s configure our Spring Boot application to enable Multipart file uploads, and define the maximum file size that can be uploaded. We’ll also configure the directory into which all the uploaded files will be stored.
  + Add the eureka client properties so that it can register with Eureka Server
  + Enable the feign client properties so that we can communicate with different microservices.

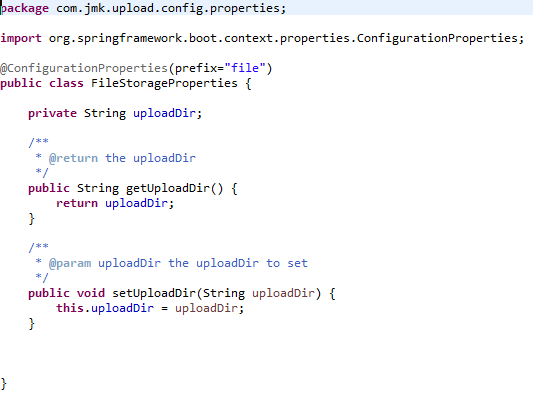
Open src/main/resources/application.properties file, and add the following properties to it.



* **Automatically binding properties to a POJO class**

Spring Boot has an awesome feature called [@ConfigurationProperties](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html" \l "boot-features-external-config-typesafe-configuration-properties) using which you can automatically bind the properties defined in the application.properties file to a POJO class.

Let’s define a POJO class called FileStorageProperties to bind all the file storage properties.

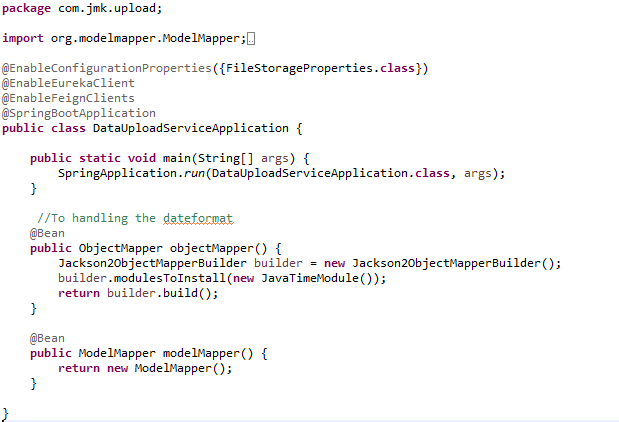


The @ConfigurationProperties(prefix = "file") annotation does its job on application startup and binds all the properties with prefix file to the corresponding fields of the POJO class.

If you define additional file properties in future, you may simply add a corresponding field in the above class, and spring boot will automatically bind the field with the property value.

* **Update the main DataUploadServiceApplication application with the following details:**

1. Enable Eureka Client so that it can register with Eureka Server
2. Enable FeignClient to interact with people-mgmt-service, account-mgmt-service and project-mgmt-service
3. Enable JpaRepositories to store the data upload history.
4. To enable the ConfigurationProperties feature, you need to add [@EnableConfigurationProperties](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/EnableConfigurationProperties.html) annotation to any configuration class.



* Write FileController to upload the single file or multiple files.