|  |  |
| --- | --- |
| **Sr No** | **Topic** |
| **1** | Introduction to Spring Cloud |
| **2** | Monolithic V/S Micro services |
| **3** | Spring Cloud Service Discovery with Netflix Eureka |
| **4** | Spring Cloud Service Discovery with Netflix Eureka |
| **5** | Spring Cloud Hystrix Circuit Breaker Pattern |
| **6** | Spring Cloud ribbon with eureka – Client side load balancer |
| **7** | Spring Cloud Zipkin and Sleuth |
| **8** | Spring Cloud Service Monitoring – Hystrix, Eureka admin and Spring boot admin |
| **9** | Spring Cloud Config with File System Backend |
| **10** | Spring Cloud Config with Git Backend |
| **11** | How to Deploy Spring Boot Application to Cloud Foundry Platform |

**Spring Micro Services + Cloud**

Spring Cloud is a framework for building robust cloud applications. Spring Cloud provides a solution to the commonly encountered patterns when developing a distributed system

## Why is Spring Cloud used?

Spring Cloud framework provides tools for developers to build a robust cloud application quickly. We can also build the microservice-based applications, for example, **configuration management, service discovery, circuit breakers, intelligent routing, cluster state, micro-proxy, a control bus, one time tokens, etc**. Using Spring Cloud, a developer can quickly develop services and applications that implement the design patterns. These patterns work well in any distributed environment, including the **bear metal data centers, developer's laptop,** and managed platform such as **Cloud Foundry**.

**Bear metal data centers:** It is a physical server dedicated to a single-tenant (a person who occupies server on rent). These are not shared between the customers. The tenant can optimize the performance according to its needs for performance, security, and reliability. The bare metal data centers are also known as Single-tenant physical server or managed dedicated server. The operating system is installed directly on the bare metal server and delivers better performance.

## Spring Cloud Components

There are the following components:

* Configuration
* Service Discovery
* Circuit Breakers
* Routing and Messaging
* API Gateway
* Tracing
* CI Pipeline and Testing
* **What is a Monolith Application?**

Monolith applications are typically huge - more 100,000 line of code. In some instances even more than million lines of code

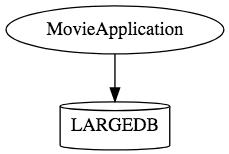
* Which is released (taken to production) once every few months
* Which has a wide range of features and functionality
* Which has a team of more than 50 working for it
* Where debugging problems is a big challenge
* Where bringing in new technology and new process is almost impossible

**Monoliths are characterized by**

* Large Application Size
* Long Release Cycles
* Large Teams

**Typical Challenges include**

* Scalability Challenges
* New Technology Adoption
* New Processes - Agile?
* Difficult to Automation Test
* Difficult to Adapt to Modern Development Practices
* Adapting to Device Explosion



* **Microservices**

**Microservice Architecture** is a Service Oriented Architecture. In the microservice architecture, there are a large number of **microservices**. By combining all the micro services, it constructs a big service. In the microservice architecture, all the services communicate with each other.

In the **Microservices** tutorial, we will understand how to implement Microservices using **Spring Cloud**. We will learn how to establish communication between microservices, **enable** **load balancing**, **scaling up and down of microservices**. We will also learn to **centralize the configuration of microservices**with **Spring Cloud Config Server**. We will implement **Eureka Naming Server** and **Distributed tracing** with **Spring Cloud Sleuth** and **Zipkin**. We will create fault tolerance microservices with **Zipkin**.

Our **microservices** tutorial discusses the basic functionalities of **Microservice Architecture**along with relevant examples for easy understanding.

* **What are Microservices**

**Definition**: According to **Sam Newman**, "Microservices are the small services that work together."

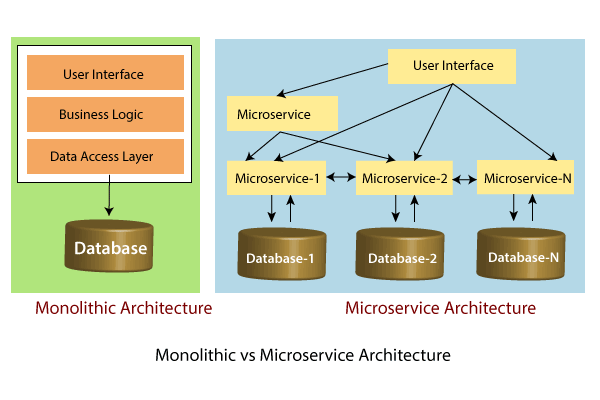
According to **James Lewis and Martin Fowler**, "The microservice architectural style is an approach to develop a single application as a suite of small services. Each microservice runs its process and communicates with lightweight mechanisms. These services are built around business capabilities and independently developed by fully automated deployment machinery."

There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.

* **Points to remember**
* These are the services which are exposed by REST.
* These are small well-chosen deployable units.
* The services must be cloud-enabled.

The microservice defines an approach to the architecture that divides an application into a pool of loosely coupled services that implements business requirements. It is next to **Service-Oriented Architecture (SOA)**. The most important feature of the microservice-based architecture is that it can perform **continuous delivery** of a large and complex application.

Microservice helps in breaking the application and build a logically independent smaller applications. For example, we can build a cloud application with the help of Amazon AWS with minimum efforts.



In the above figure, each microservice has its own business layer and database. If we change in one microservice, it does not affect the other services. These services communicate with each other by using lightweight protocols such as HTTP or REST or messaging protocols.

* **Principles of Microservices**

There are the following principles of Microservices:

* Single Responsibility principle
* Modeled around business domain
* Isolate Failure
* Infrastructure automation
* Deploy independently
* **Single Responsibility Principle**

The single responsibility principle states that a class or a module in a program should have only one responsibility. Any microservice cannot serve more than one responsibility, at a time.

* **Modeled around business domain**

Microservice never restricts itself from accepting appropriate technology stack or database. The stack or database is most suitable for solving the business purpose.

* **Isolated Failure**

The large application can remain mostly unaffected by the failure of a single module. It is possible that a service can fail at any time. So, it is important to detect failure quickly, if possible, automatically restore failure.

* **Infrastructure Automation**

The infrastructure automation is the process of scripting environments. With the help of scripting environment, we can apply the same configuration to a single node or thousands of nodes. It is also known as configuration management, scripted infrastructures, and system configuration management.

* **Deploy independently**

Microservices are platform agnostic. It means we can design and deploy them independently without affecting the other services.

# Spring Cloud Service Discovery with Netflix Eureka

In typical microservice architecture we have many small applications deployed separately and they often need to communicate with each other. Specifically, when we say client service, we mean a service that needs to make REST calls to some other end service.

The problem in this type of architecture is how the client service finds all of its end services. We could hardcode the hostname/port in some property file, but this isn't always practical or feasible in a cloud environment. There could be any number of microservices, and it's time and resource-consuming to hard-code when there's an uncertain amount of them, and when their locations may change.

To further add to the complexity, services could have multiple instances of themselves (based on the load). Which instance will actually serve the response could be challenging as we want to have equal load distribution.

**Netflix Eureka**

**Netflix Eureka** is a lookup server (also called a registry). All the microservices in the cluster register themselves to this server.

When making a REST call to another service, instead of providing a hostname and port, they just provide the service name.

The actual routing is done at runtime along with equally distributing the load among the end services. There are other service discovery clients like [Consul](https://www.consul.io/discovery.html), [Zookeeper](https://spring.io/projects/spring-cloud-zookeeper) etc

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.

Example :

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.

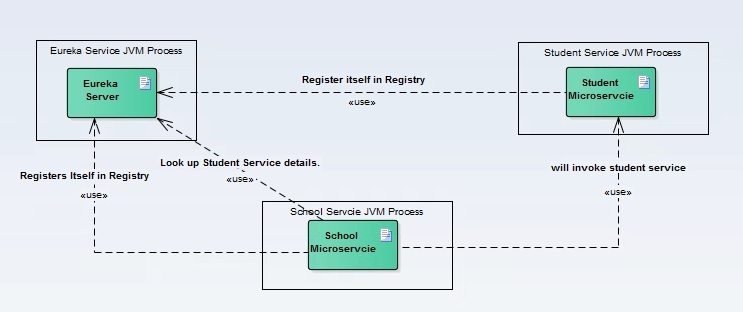
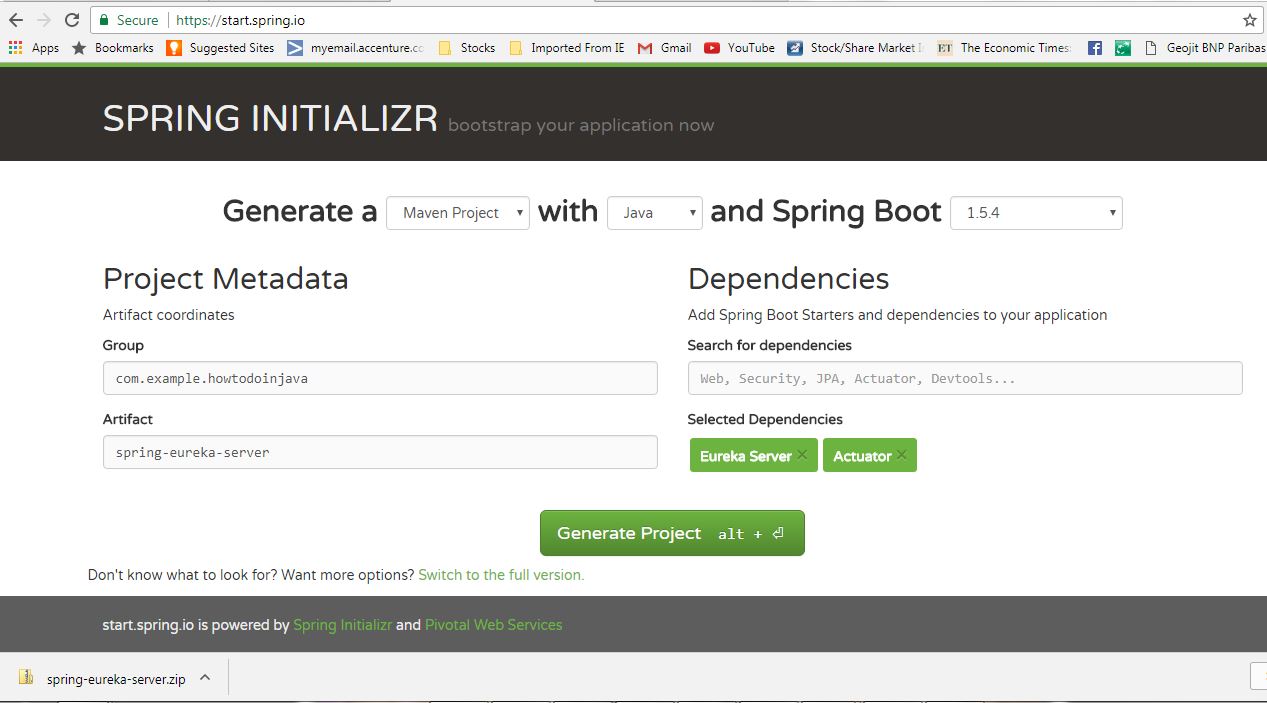
[](https://howtodoinjava.com/wp-content/uploads/2017/07/Discovery_interratction-Diagram.jpg)

Fig. Component Interaction with each other

* 1. **Eureka Service Registry 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://howtodoinjava.com/wp-content/uploads/2017/07/server_projec-generation.jpg)

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 –

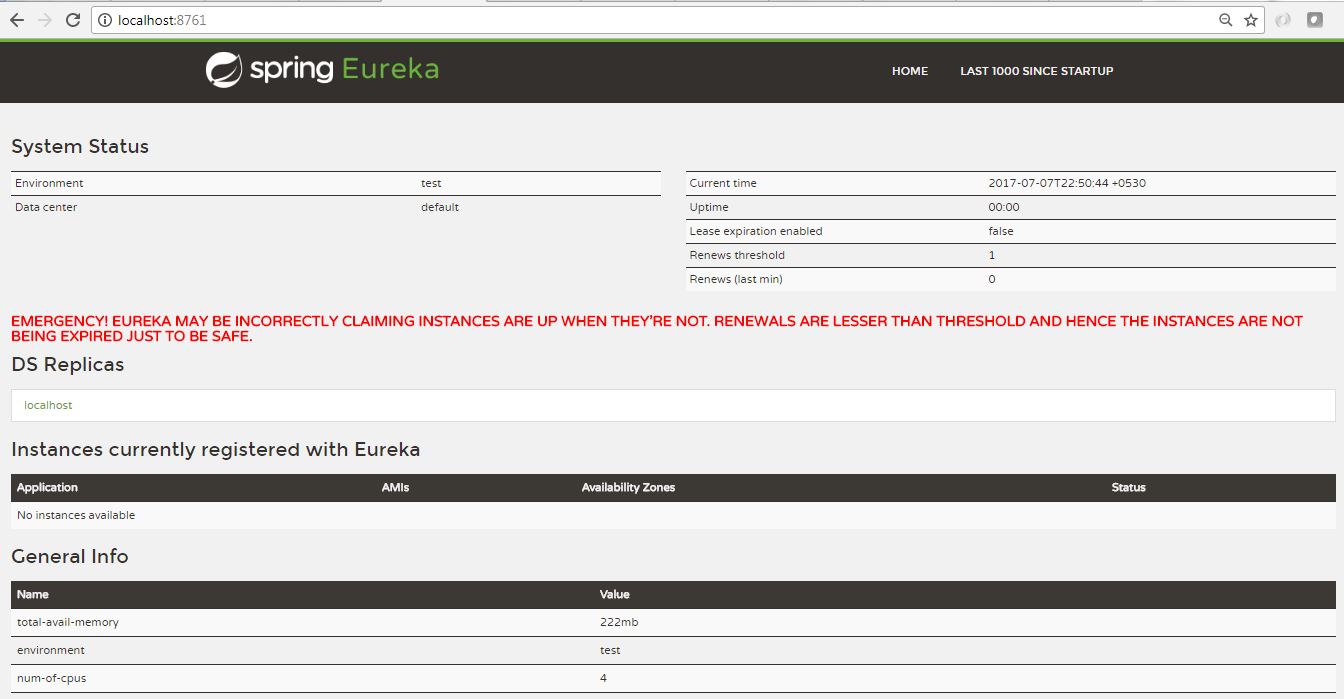
|  |
| --- |
| server:    port: ${PORT:8761} # Indicate the default PORT where this service will be started    eureka:    client:      registerWithEureka: false   #telling the server not to register himself in the service registry      fetchRegistry: false    server:      waitTimeInMsWhenSyncEmpty: 0    #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://howtodoinjava.com/wp-content/uploads/2017/07/eureka_console_without_anyClient.jpg) Fig. Eureka Console Without Any Client

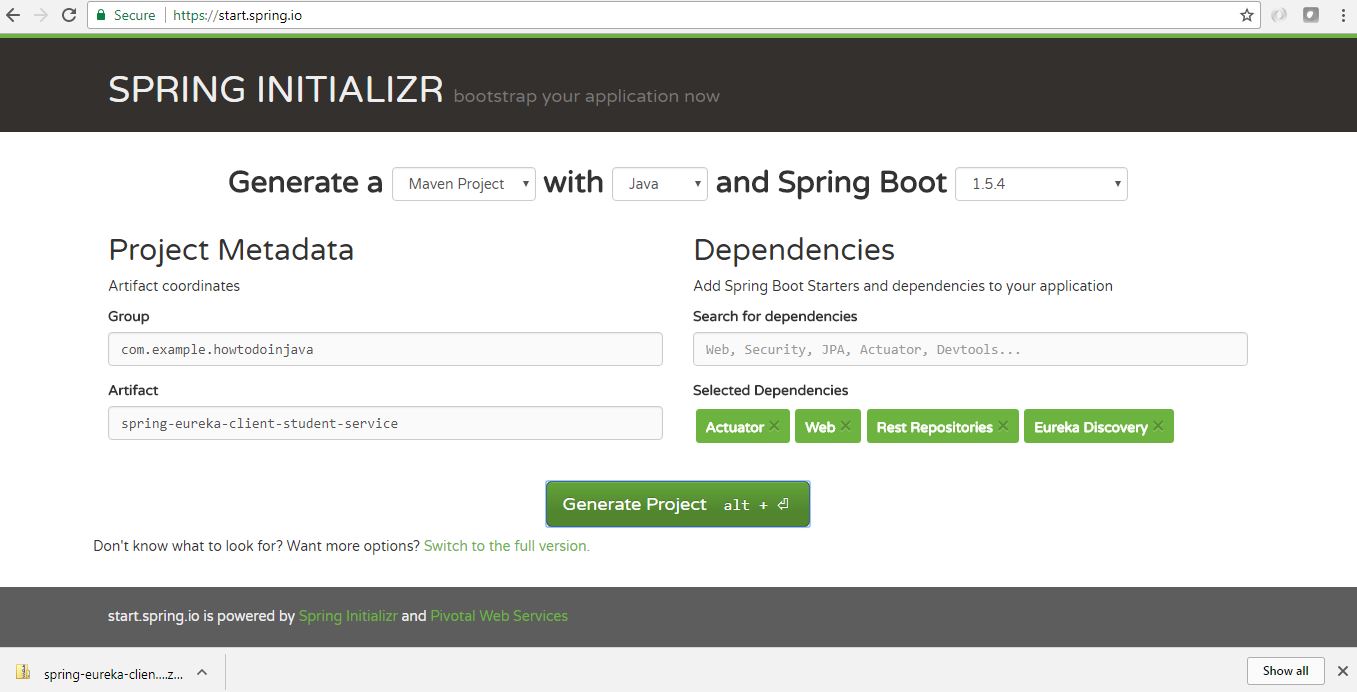
Please note that at this point no service is registered here which is expected and once we will spin up the client services, this server will automatically updated with the details of the client services.

* 1. **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://howtodoinjava.com/wp-content/uploads/2017/07/client_projec-generation_Student.jpg) Fig.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.

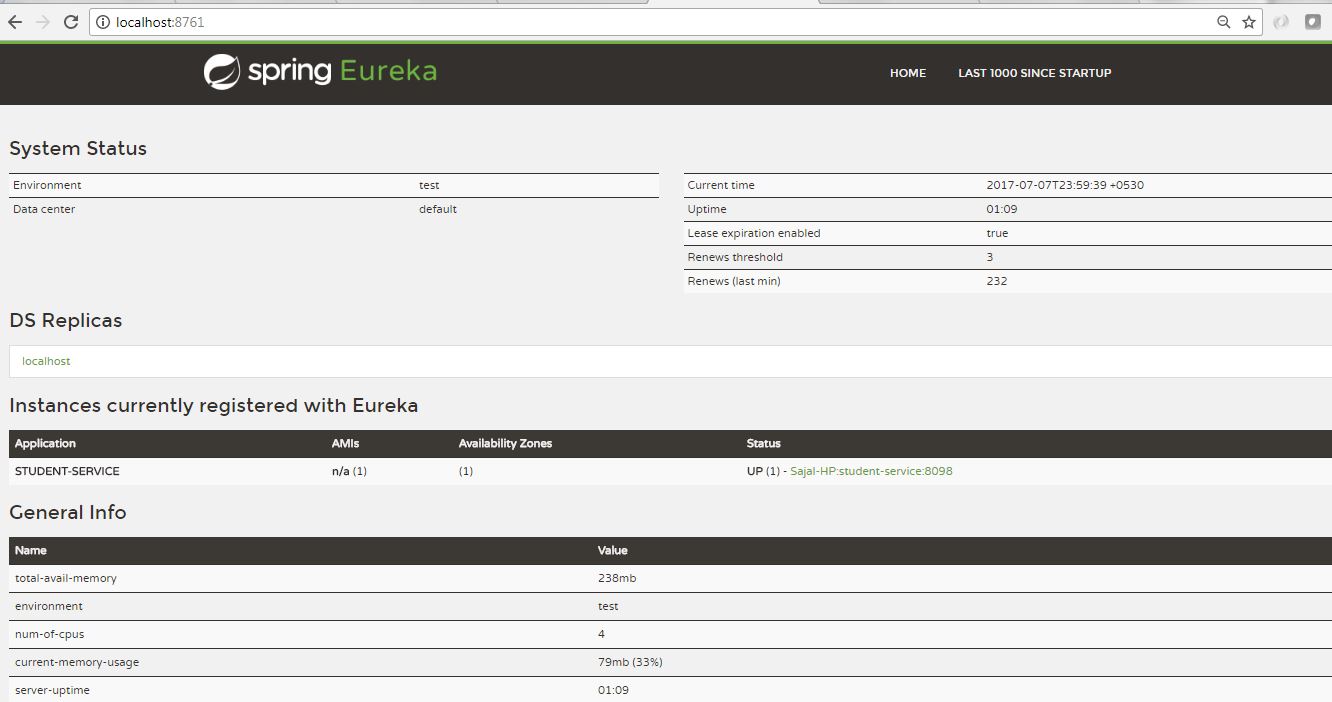
|  |
| --- |
| package com.example.howtodoinjava.springeurekaclientstudentservice.controller;    import java.util.ArrayList;  import java.util.HashMap;  import java.util.List;  import java.util.Map;    import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestMapping;  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://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.

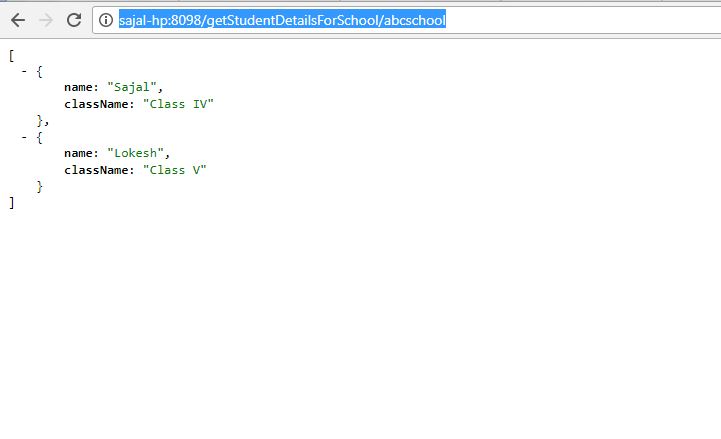
[](https://howtodoinjava.com/wp-content/uploads/2017/07/Student-Service-Responding.jpg)

Fig.Student Service response

* 1. **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.

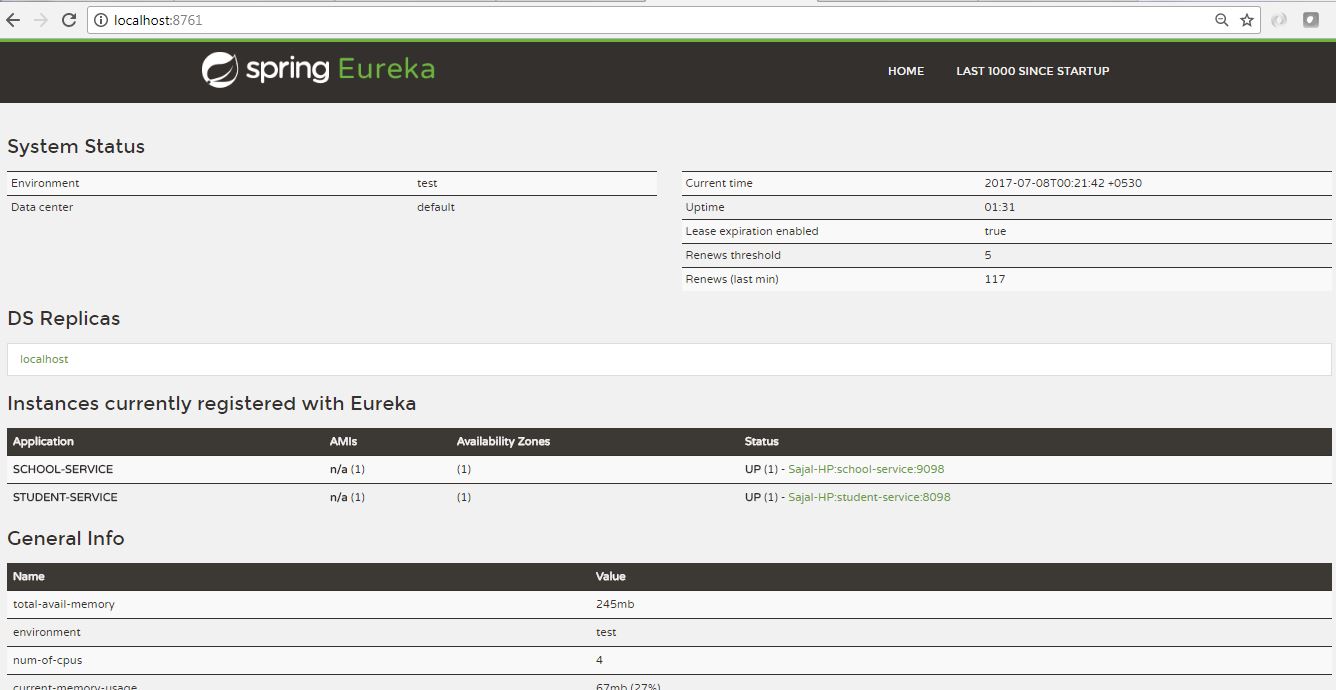
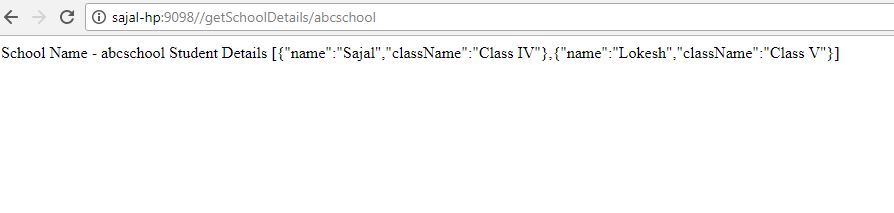
[](https://howtodoinjava.com/wp-content/uploads/2017/07/eureka_console_with_both_servcies.jpg)

Fig. 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://howtodoinjava.com/wp-content/uploads/2017/07/School-Service-Responding.jpg) Fig. School Service Response

# Spring Cloud Hystrix Circuit Breaker Pattern

 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.

## Why 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?

**Example :**

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

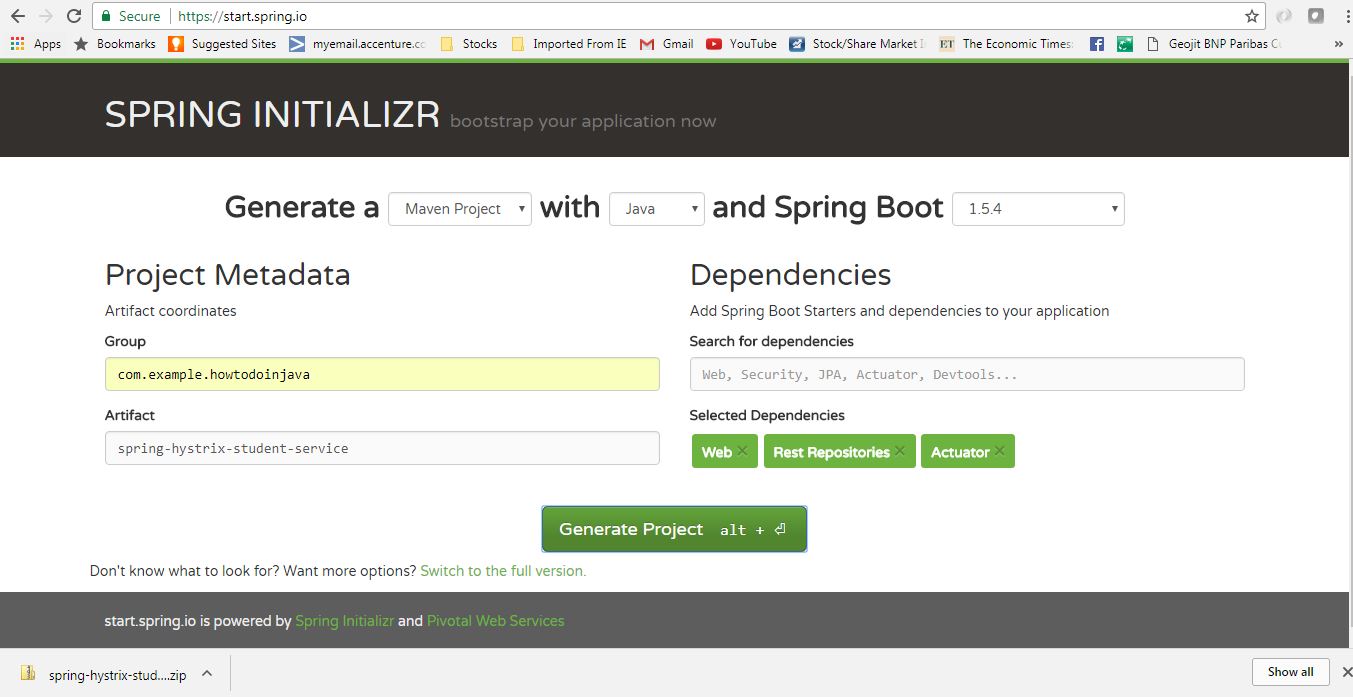
* **Student Microservice** – Which will give some basic functionality on Student entity. It will be a REST based service. We will call this service from School Service to understand Circuit Breaker. It will run on port 8098 in localhost.
* **School Microservice** – Again a simple REST based microservice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

## Create Student Service

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

#### Create spring boot project

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

[](https://howtodoinjava.com/wp-content/uploads/2017/07/studentservciegeneration.jpg) Fig. 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 –

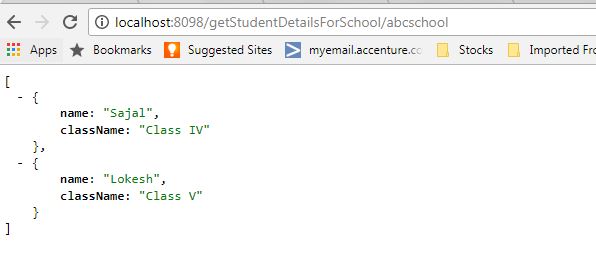
[](https://howtodoinjava.com/wp-content/uploads/2017/07/studentserviceresponse.jpg)

Fig.Student Service Response

## Create School Service – Hystrix Enabled

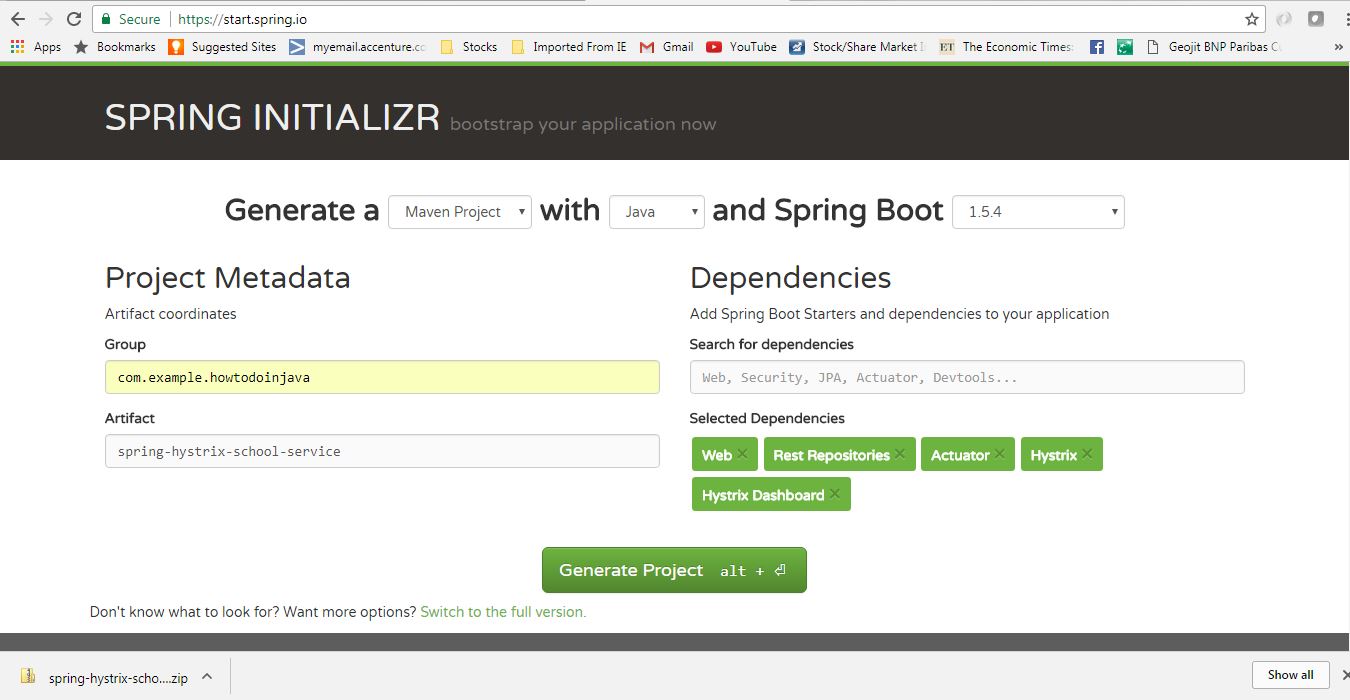
Similar to Student service, create another microservice for School. It will internally invoke already developed Student Service.

#### Generate spring boot project

Create a Spring boot project from [Spring Boot initializer portal](https://start.spring.io/) 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://howtodoinjava.com/wp-content/uploads/2017/07/schoolservicegeneration.jpg) Fig.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://howtodoinjava.com/wp-content/uploads/2017/07/schoolserviceresponse.jpg)

## Test Hystrix Circuit Breaker – Demo

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

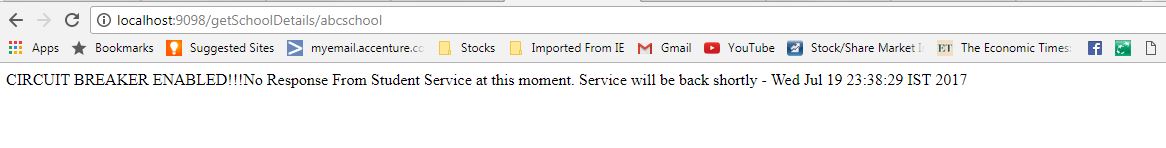
It should show the below output in browser –

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

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.

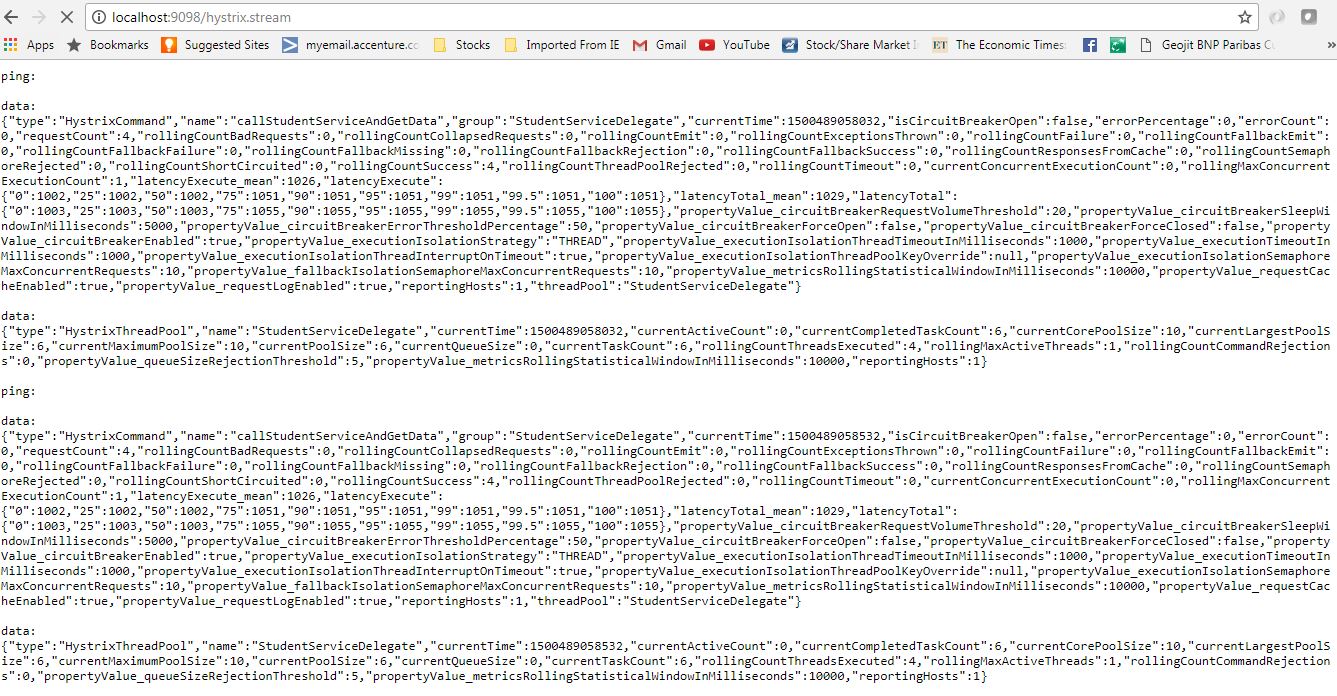
[](https://howtodoinjava.com/wp-content/uploads/2017/07/schoolserviceresponse_fallback.jpg)School Service Response Fallback path

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

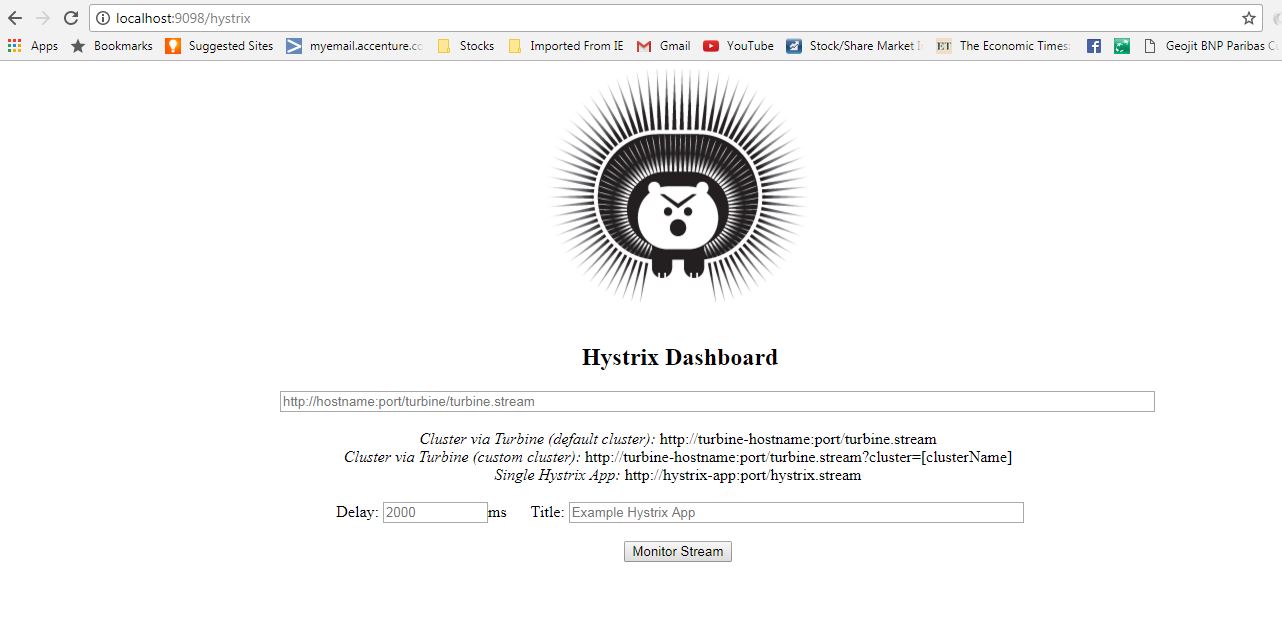
As we have added hystrix dashboard dependency, hystrix has provided one nice Dashboard and a Hystrix Stream in the bellow URLS:

* [**http://localhost:9098/hystrix.stream**](http://localhost:9098/hystrix.stream) – 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 –

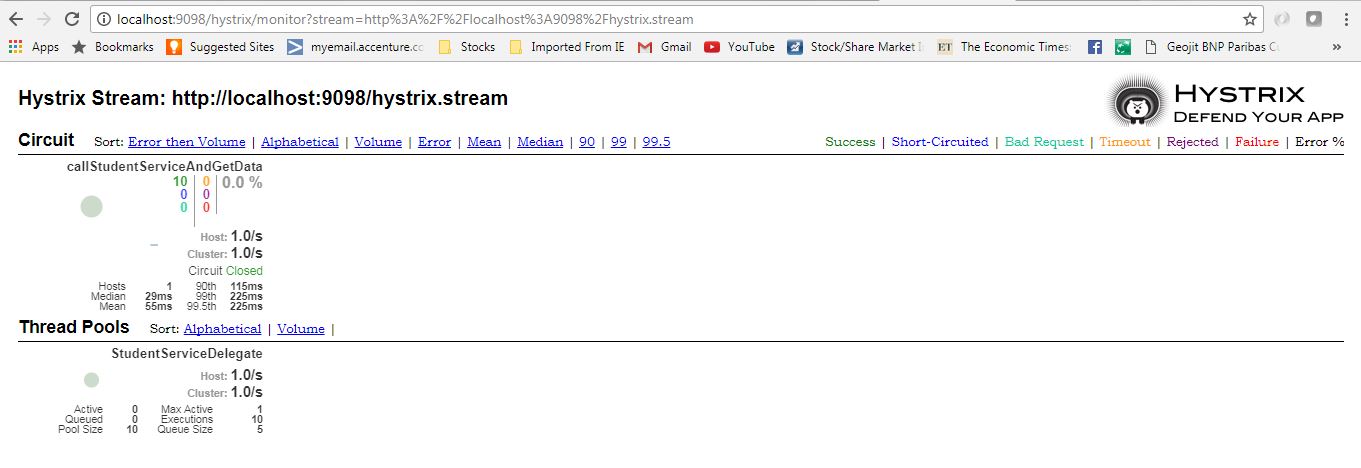
[](https://howtodoinjava.com/wp-content/uploads/2017/07/HystrixStream.jpg)

Hystrix Stream output

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

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

* Now add <http://localhost:9098/hystrix.stream> 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://howtodoinjava.com/wp-content/uploads/2017/07/HystrixDashboard.jpg)Hystrix visual Dashboard

# Spring Cloud ribbon with eureka – Client side load balancer

## 1. Traditional server side load balancing

Server side load balancing is involved in monolithic applications where we have limited number of application instances behind the load 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/#search%7Cga%7C1%7Cribbon). Here is an example to add dependency in Maven:

|  |
| --- |
|  |
| <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 one 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

###### **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 registry 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();      }  } |

**MyClientSideController.java:**

**package** com.example.ribbonclient;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.cloud.client.loadbalancer.LoadBalanced;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

**import** org.springframework.web.client.RestTemplate;

@RestController

**public** **class** MyClientSideController {

@LoadBalanced

@Bean

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

@Autowired

RestTemplate restTemplate;

@RequestMapping("/client/frontend")

**public** String hi() {

String randomString = **this**.restTemplate.getForObject("http://server/backend", String.**class**);

**return** "Server Response :: " + randomString;

}

}

## 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.

# Spring Cloud Zipkin and Sleuth

[Zipkin](http://zipkin.io/) is very efficient tool for **distributed tracing** in [micro services](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/) ecosystem. Distributed tracing, in general, is latency measurement of each component in a distributed transaction where multiple micro services are invoked to serve a single business usecase. 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.

## 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 analyzing 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/), [ElasticSearch](https://www.elastic.co/) 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). 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) 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/>.

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/) 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/) microproxy
* HTTP headers received at [Spring MVC](https://howtodoinjava.com/spring-mvc-tutorial/) controllers
* requests over messaging technologies like Apache Kafka or RabbitMQ etc.

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

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.

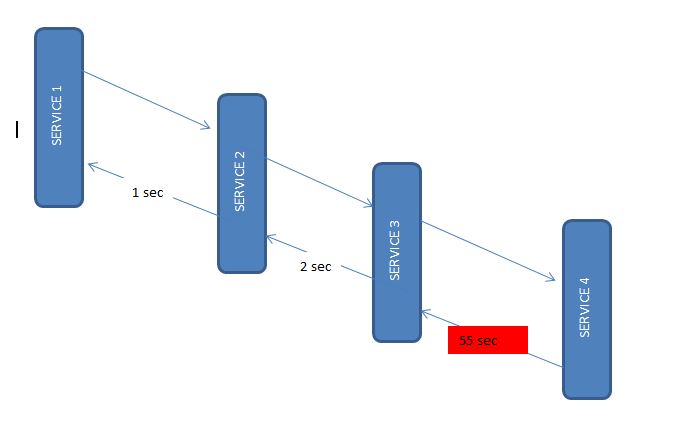
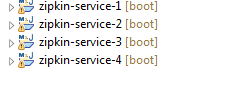


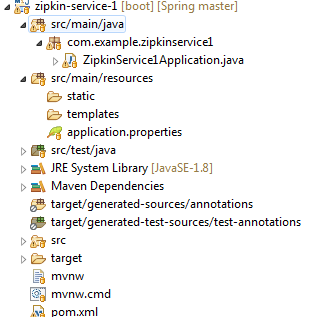
Fig. Microservices Interactions

#### Create Microservices

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.



#### Zipkin Service -1



***Pom.xml:***

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

***application.propperties:***

server.port = 8081

spring.application.name = zipkin-server1

***ZipkinService1Application.java:***

**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>() {

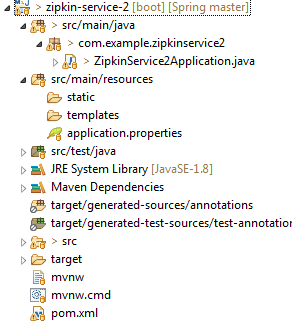
}).getBody();

**return** "Hi...";

}

}

#### Zipkin Service -2



***application.properties:***

server.port = 8082

spring.application.name = zipkin-server2

***ZipkinService2Application.java:***

**package** com.example.zipkinservice2;

**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** ZipkinService2Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ZipkinService2Application.**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="/zipkin2")

**public** String zipkinService1() {

***LOG***.info("Inside zipkinService 2..");

***LOG***.info("Now let's create some intentional delay...");

**try** {

Thread.*sleep*(20 \* 1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

***LOG***.info("returning afte delay..");

String response = (String) restTemplate.exchange("http://localhost:8083/zipkin3", HttpMethod.***GET***, **null**, **new** ParameterizedTypeReference<String>() {

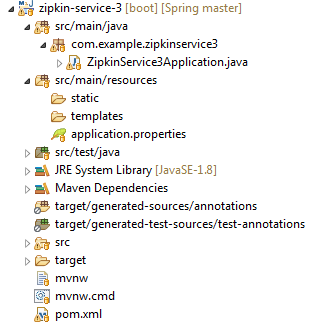
}).getBody();

**return** "Hi...";

}

}

#### Zipkin Service -3



***application.properties:***

server.port = 8083

spring.application.name = zipkin-server3

***ZipkinService3Application.java:***

**package** com.example.zipkinservice3;

**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** ZipkinService3Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ZipkinService3Application.**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="/zipkin3")

**public** String zipkinService1() {

***LOG***.info("Inside zipkinService 3..");

String response = (String) restTemplate.exchange("http://localhost:8084/zipkin4", HttpMethod.***GET***, **null**, **new** ParameterizedTypeReference<String>() {

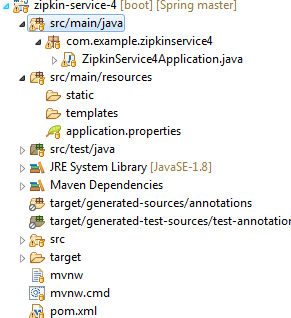
}).getBody();

**return** "Hi...";

}

}

#### Zipkin Service -4



**application.properties:**

server.port = 8084

spring.application.name = zipkin-server4

**ZipkinService4Application.java:**

**package** com.example.zipkinservice4;

**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.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RestController;

**import** org.springframework.web.client.RestTemplate;

@SpringBootApplication

**public** **class** ZipkinService4Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ZipkinService4Application.**class**, args);

}

}

@RestController

**class** ZipkinController {

@Autowired

RestTemplate restTemplate;

@Bean

**public** AlwaysSampler alwaysSampler() {

**return** **new** AlwaysSampler();

}

**private** **static** **final** Logger ***LOG*** = Logger.*getLogger*(ZipkinController.**class**.getName());

@Bean

**public** RestTemplate getRestTemplate() {

**return** **new** RestTemplate();

}

@GetMapping(value = "/zipkin4")

**public** String zipkinService1() {

***LOG***.info("Inside zipkinService 4..");

**return** "Hi...";

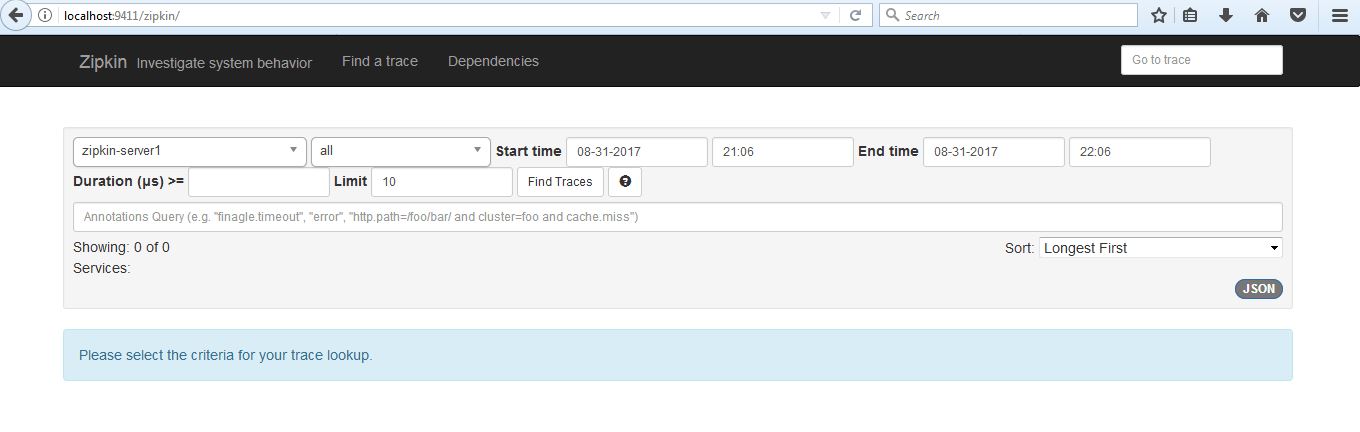
}

}

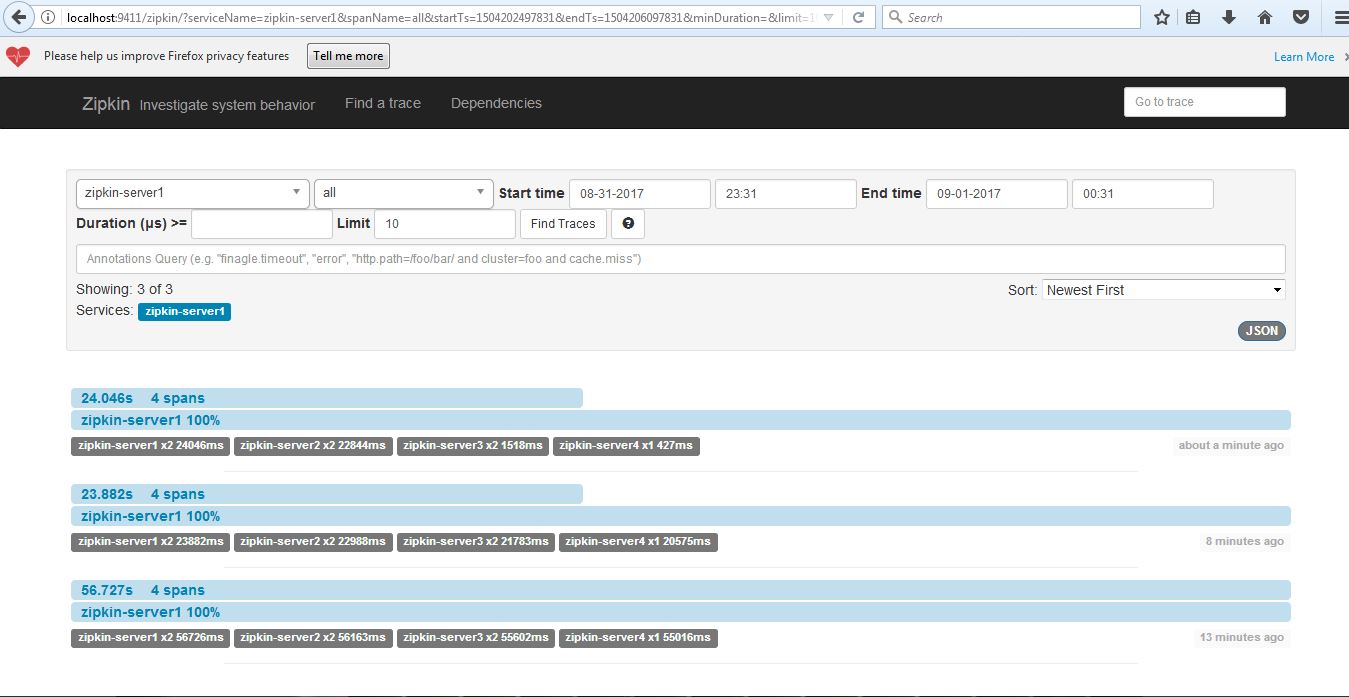
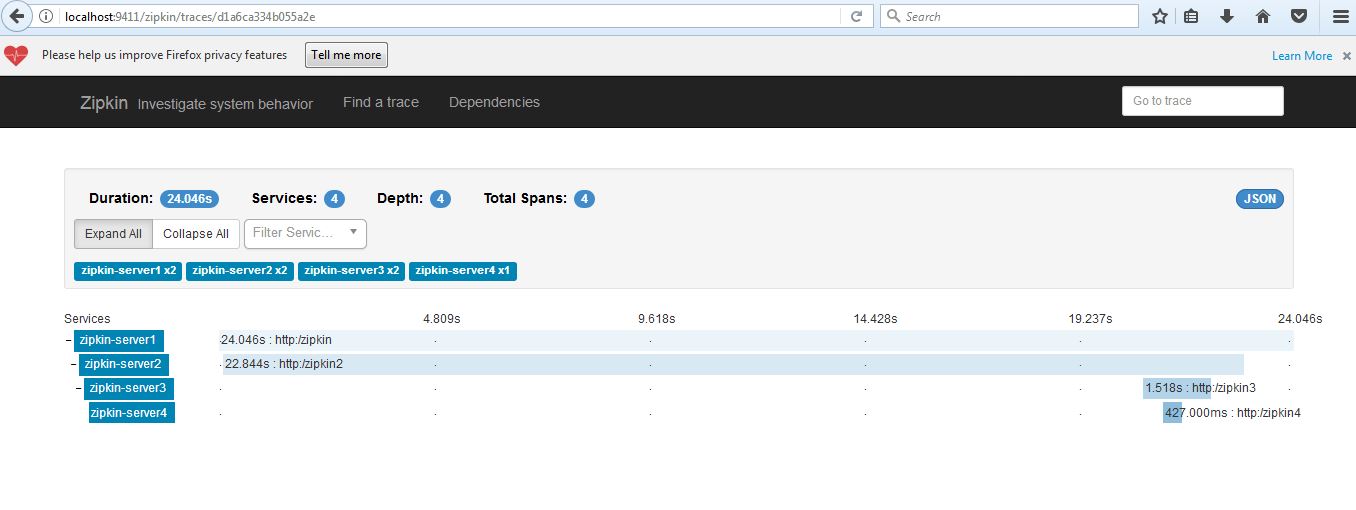
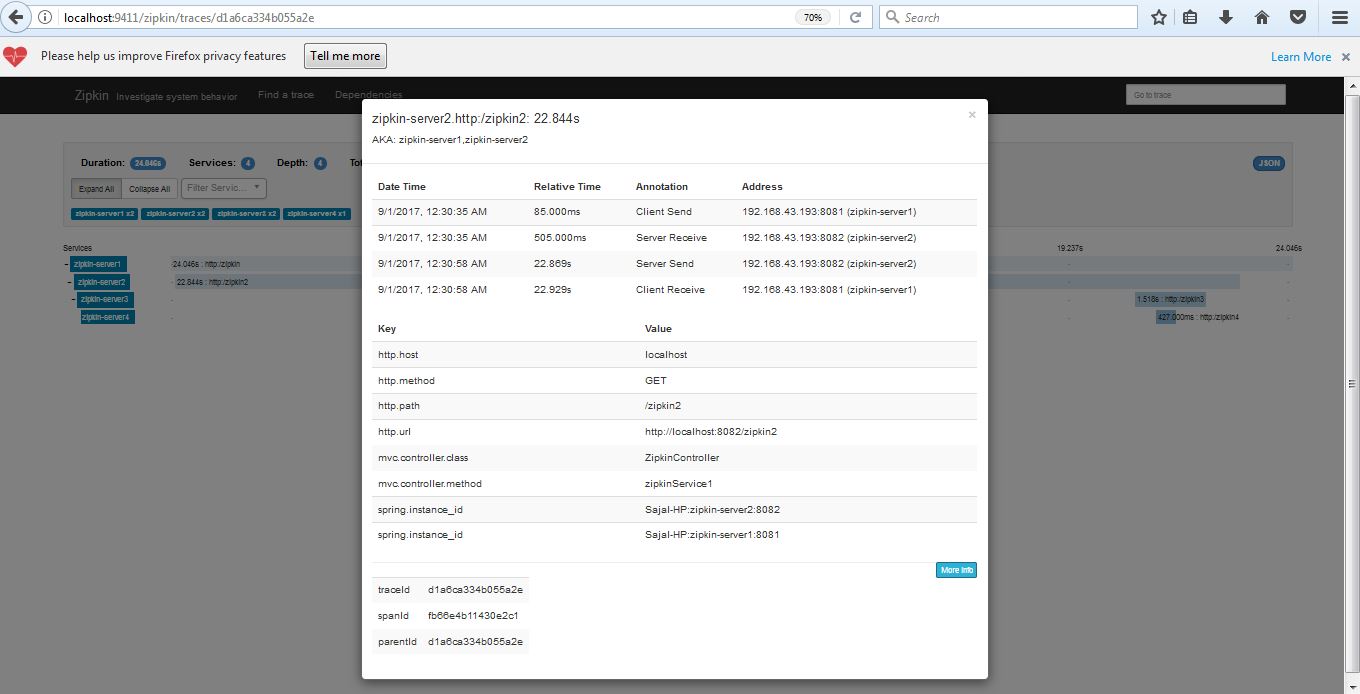
**Demo**

* Do a final maven build using command mvn clean install in microservices, start all the 4 applications along with the zipkin server.
* Now test the first service endpoint couple of time from browser – <http://localhost:8081/zipkin>. Please be aware that there is an intentional delay in one of the above 4 services. So there will be delay is final response which is expected, just don’t give up.

After API call succeed, we can see the latency statistics at zipkin UI <http://localhost:9411/zipkin/>. Choose the first service in the first drop-down, and once click on **Find Traces** button.

 **Fig. Zipkin Home screen**

You should see this type of UI where you can do performance analysis by looking at tracing data.

**Fig. Find Traces UI**[](https://howtodoinjava.com/wp-content/uploads/2017/08/7-3.jpg)**Fig.One particular transaction overview**[](https://howtodoinjava.com/wp-content/uploads/2017/08/8-3.jpg)**Fig.Details of a particular service call statistics**

# Spring Cloud Service Monitoring – Hystrix, Eureka admin and Spring boot admin

## 1. Overview

In this demo, we will create three applications.

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

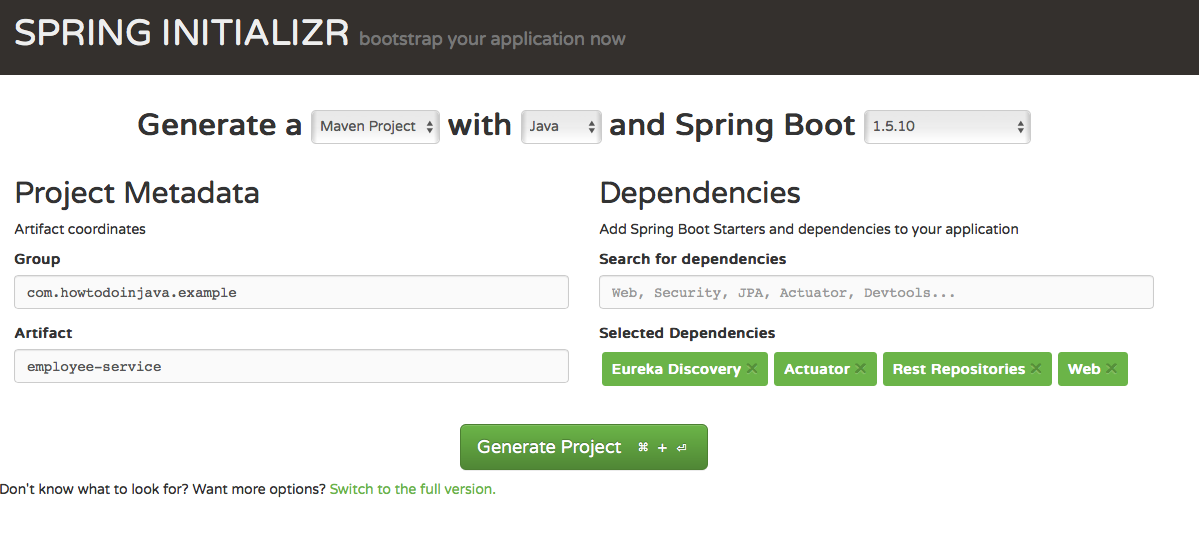
This demo has been created around **Netflix Eureka** to centrally manage and monitor registered applications. As you may already know that Netflix Eureka server is for building **service registry server** and associated Eureka clients, which will register themselves to lookup other services and communicate through REST apis.

## 2. Technology stack

* Java 1.8
* Spring tool suite
* Spring cloud
* Spring boot
* Spring Rest
* Maven

## 3. Employee Service

* Create a Spring boot project from [Spring boot initializer](https://start.spring.io/)/[Spring tool suite](https://spring.io/tools/sts) with dependencies **Eureka Discovery**, **Actuator**, **Web**, **Rest repositories**.



* The main application class EmployeeServiceApplication to start the Spring boot application.

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

**@EnableEurekaClient** – This annotation register this service as an Eureka client in [Eureka Server application](https://howtodoinjava.com/spring-cloud/microservices-monitoring/#eureka-dashboard) created below.

Create a Rest controller class [EmployeeServiceController] to expose Employee data.

|  |
| --- |
| **EmployeeServiceController.java** |
| package com.howtodoinjava.example.employee.controller;  import java.util.HashMap;  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.howtodoinjava.example.employee.beans.Employee;   @RestController  public class EmployeeServiceController {       private static final Map<Integer, Employee> employeeData = new HashMap<Integer,Employee() {           private static final long serialVersionUID = -3970206781360313502L;          {              put(111,new Employee(111,"Employee1"));              put(222,new Employee(222,"Employee2"));          }      };        @RequestMapping(value = "/findEmployeeDetails/{employeeId}", method = RequestMethod.GET)      public Employee getEmployeeDetails(@PathVariable int employeeId) {          System.out.println("Getting Employee details for " + employeeId);            Employee employee = employeeData.get(employeeId);          if (employee == null) {                employee = new Employee(0, "N/A");          }          return employee;      }  } |

* Associated Employee Bean class is below.

|  |
| --- |
| **Employee.java** |
| package com.howtodoinjava.example.employee.beans;   public class Employee {      private String name;      private int id;        @Override      public String toString() {          return "Employee [name=" + name + ", id=" + id + "]";      }  } |

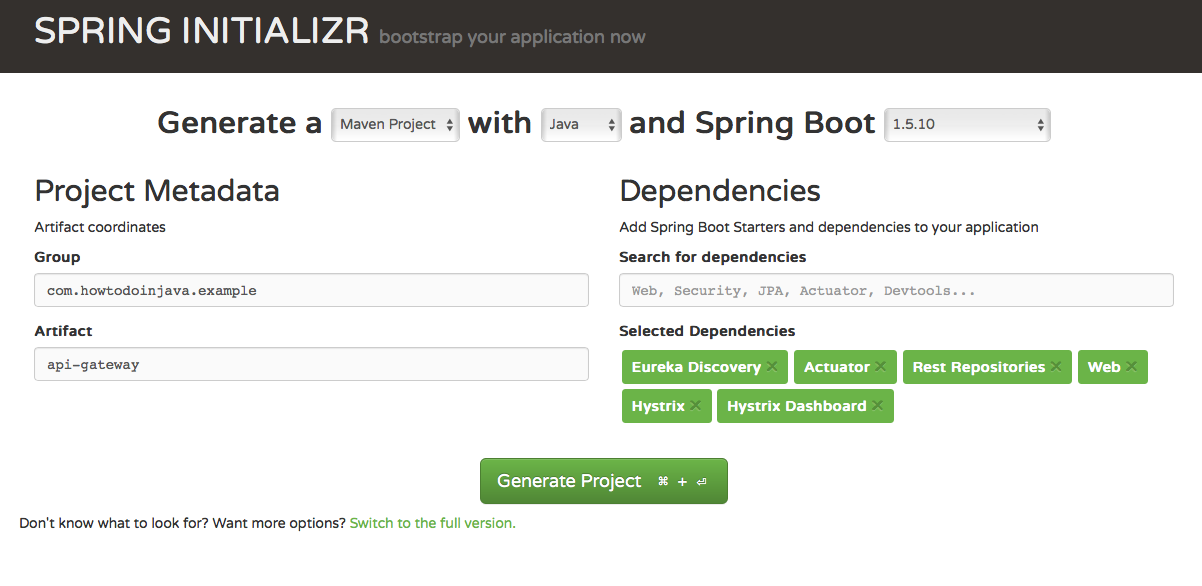
* Create application.yml in src/main/resources directory.

|  |
| --- |
| **application.yml** |
| server:    port: 8011    eureka:    instance:      leaseRenewalIntervalInSeconds: 5      leaseExpirationDurationInSeconds: 2    client:      serviceUrl:        defaultZone: http://localhost:8761/eureka/      healthcheck:        enabled: true      lease:        duration: 5    spring:    application:      name: employee-service    management:    security:      enabled: false    logging:    level:      com.self.sprintboot.learning.employee: DEBUG |

* Start this application reachable http://localhost:8011/findEmployeeDetails/111  
  https://howtodoinjava.com/wp-content/uploads/2018/03/Screen-Shot-2018-03-10-at-5.26.43-PM.png

## 4. API-Gateway with Hystrix

* Create a Spring boot project from [Spring boot initializer](https://start.spring.io/)/[Spring tool suite](https://spring.io/tools/sts) with dependencies Eureka Discovery, Actuator, Web, Hystrix, Hystrix Dashboard, Rest repositories.



* The main application class ApiGatewayApplication to start Spring boot application.

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

**@EnableHystrixDashBoard** – To give dashboard view of Hystrix stream.  
**@EnableCircuitBreaker** – To enable Circuit breaker implementation.

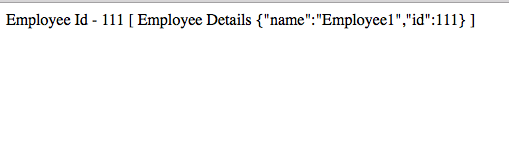
Create a REST controller class [ EmployeeController ] to expose Employee data.

|  |
| --- |
| **EmployeeController.java** |
| package com.howtodoinjava.example.apigateway.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;   import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;    @RestController  public class EmployeeController {      @Autowired      RestTemplate restTemplate;        @RequestMapping(value = "/employeeDetails/{employeeid}", method = RequestMethod.GET)      @HystrixCommand(fallbackMethod = "fallbackMethod")      public String getEmployees(@PathVariable int employeeid)      {          System.out.println("Getting Employee details for " + employeeid);           String response = restTemplate.exchange("<http://employee-service/findEmployeeDetails/>{employeeid}",                                  HttpMethod.GET, null, new ParameterizedTypeReference<String>() {}, employeeid).  getBody();            System.out.println("Response Body " + response);           return "Employee Id -  " + employeeid + " [ Employee Details " + response+" ]";      }        public String  fallbackMethod(int employeeid){            return "Fallback response:: No employee details available temporarily";      }        @Bean      @LoadBalanced      public RestTemplate restTemplate() {          return new RestTemplate();      }  } |
|  |

* Create application.yml in src/main/resources directory.

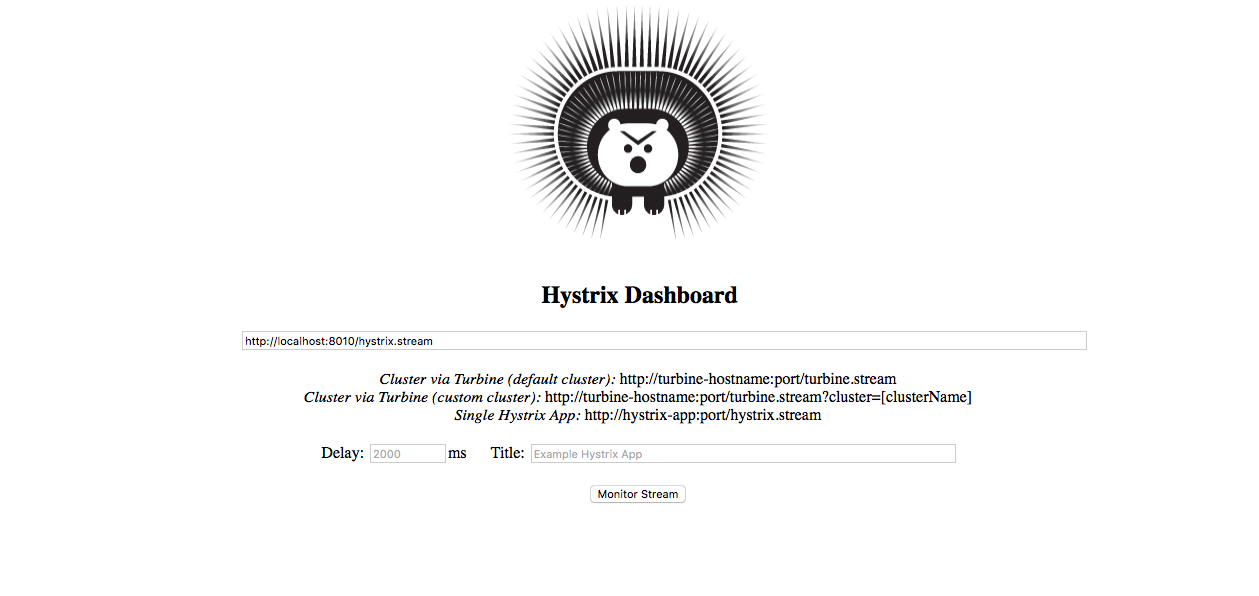
|  |
| --- |
| **application.yml** |
| server:    port: 8010    #port number    eureka:    instance:      leaseRenewalIntervalInSeconds: 5      leaseExpirationDurationInSeconds: 2    client:      serviceUrl:        defaultZone: http://localhost:8761/eureka/      healthcheck:        enabled: true      lease:        duration: 5    spring:    application:      name: api-gateway   management:    security:      enabled: false    logging:    level:      com.self.sprintboot.learning.apigateway: DEBUG |

* Start the application reachable at http://localhost:8010/employeeDetails/111.



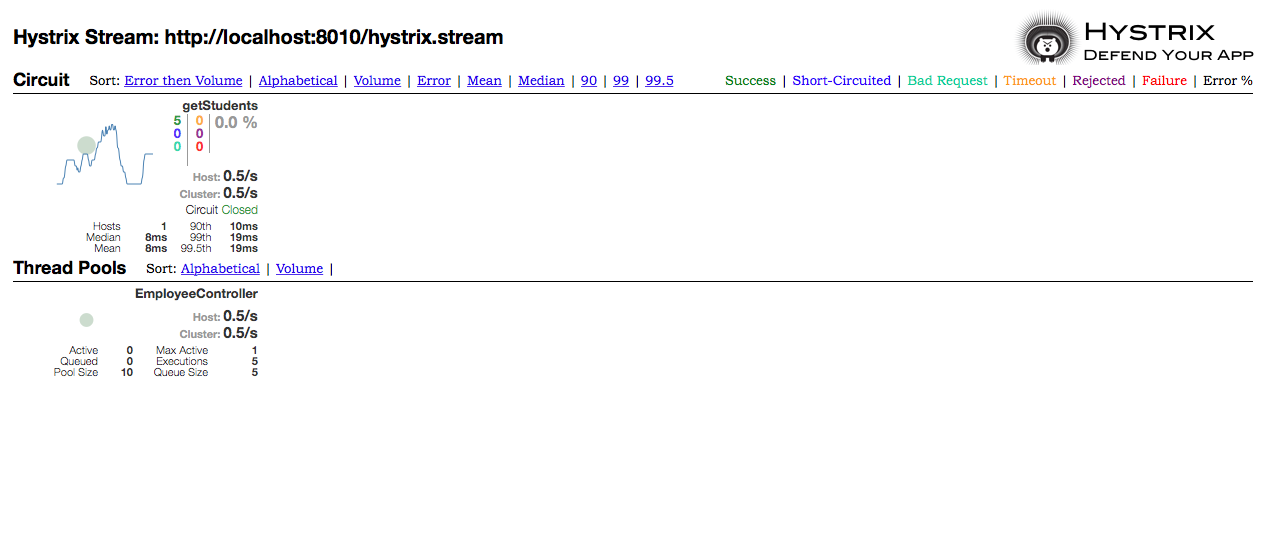
## 5. Hystrix dashboard view

* To **monitor via Hystrix dashboard**, open Hystrix dashboard at http://localhost:8010/hystrix.



This is the home page where event stream URL needs to be put for monitoring.

* Now view **hystrix stream** in dashboard – http://localhost:8010/hystrix.stream

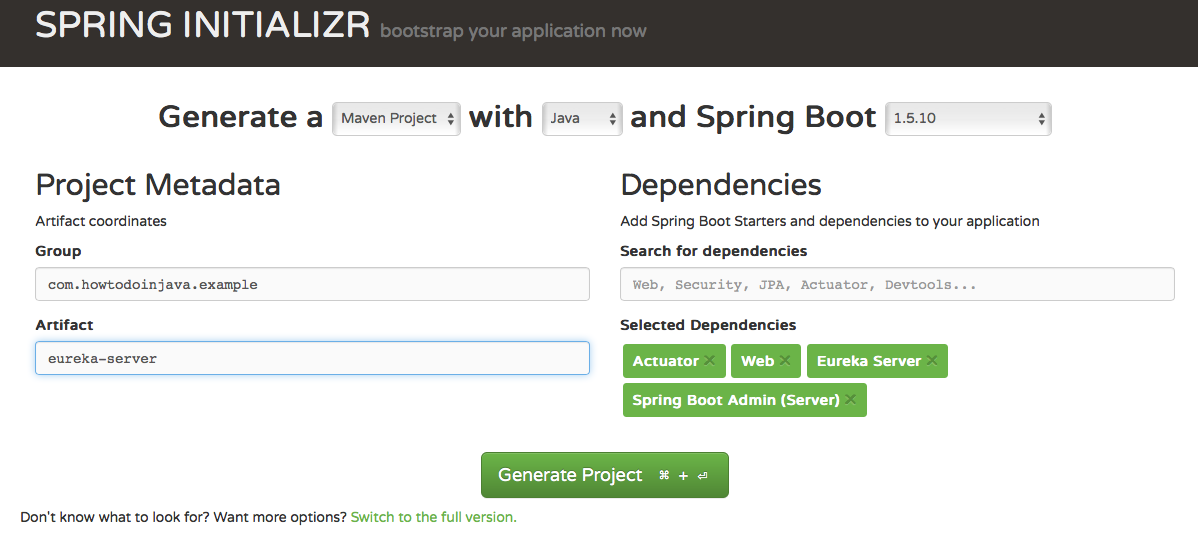


This provides realtime information of all Hystrix commands and thread pools.

## 6. Eureka admin dashboard view

Now let’s learn to use the Eureka admin dashboard view.

* Create a Spring boot project from [Spring boot initializer](https://start.spring.io/)/[Spring tool suite](https://spring.io/tools/sts) with these dependencies **Eureka Server**, **Actuator**, **Web**, **Spring Boot Admin Server**.



* The main application class EurekaServerApplication to start spring boot application.

|  |
| --- |
| **EurekaServerApplication.java** |
| package com.howtodoinjava.example.eureka;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  import de.codecentric.boot.admin.config.EnableAdminServer;  @SpringBootApplication  @EnableEurekaServer  @EnableAdminServer   public class EurekaServerApplication {       public static void main(String[] args) {          SpringApplication.run(EurekaServerApplication.class, args);      }  } |

**@EnableEurekaServer**– This annotation will make this application to act as Microservice registry and discovery server.  
**@EnableAdminServer**– This annotation provides Spring Boot Admin configuration.

Create application.yml and bootstrap.yml in src/main/resources directory.

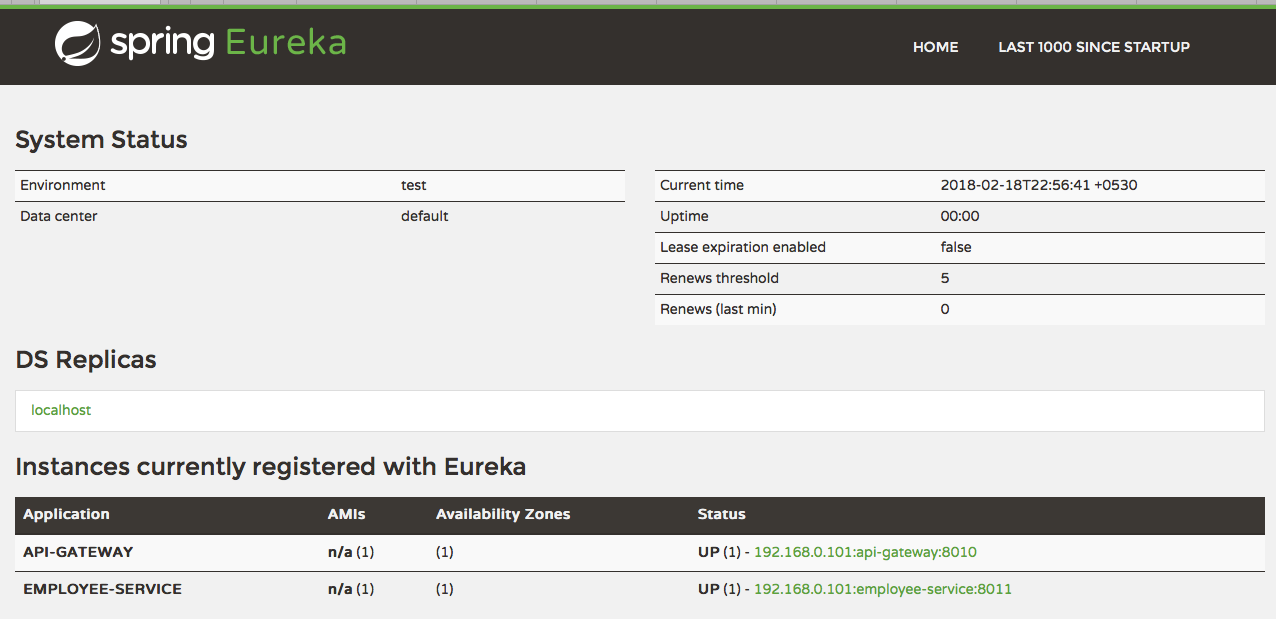
Add application.yml with given config. Please note that for Spring boot admin server a different context path /admin has been provided for not conflicting with /eureka.

|  |
| --- |
| **application.yml** |
| server:    port: ${PORT:8761}    eureka:    client:      registryFetchIntervalSeconds: 5      registerWithEureka: false      serviceUrl:        defaultZone: ${DISCOVERY\_URL:http://localhost:8761}/eureka/    instance:      leaseRenewalIntervalInSeconds: 10    management:    security:      enabled: false  spring:    boot:      admin:        context-path: /admin  #A different context path for Spring boot admin server has been provided avoiding conflict with eureka |

* Create bootstrap.yml and give this configuration.

|  |
| --- |
| bootstrap.yml |
| spring:    application:      name: Eureka-Server    cloud:      config:        uri: ${CONFIG\_SERVER\_URL:http://localhost:8888} |

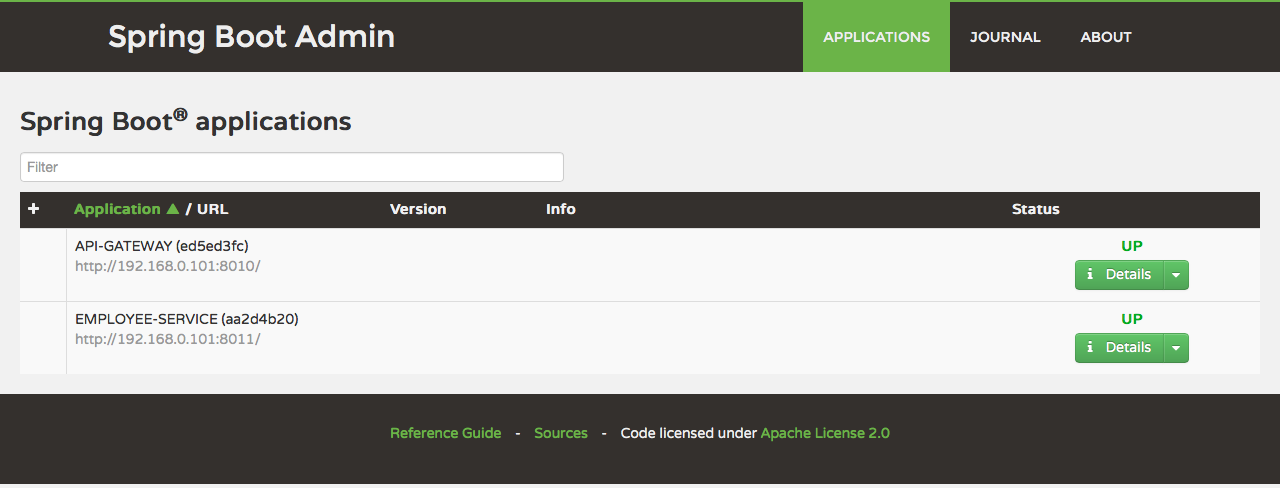
* Start the application. But before that make sure the rest of the client applications mentioned above are started before so as to see all registered applications. This application is reachable at http://localhost:8761.



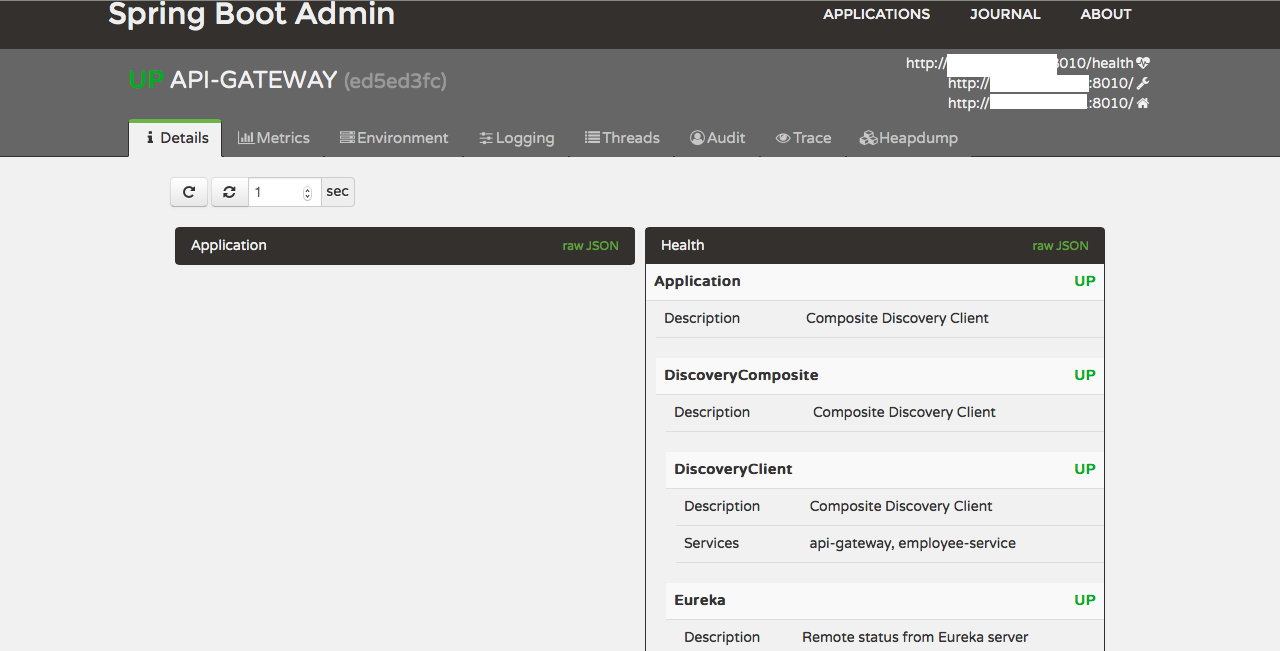
## 

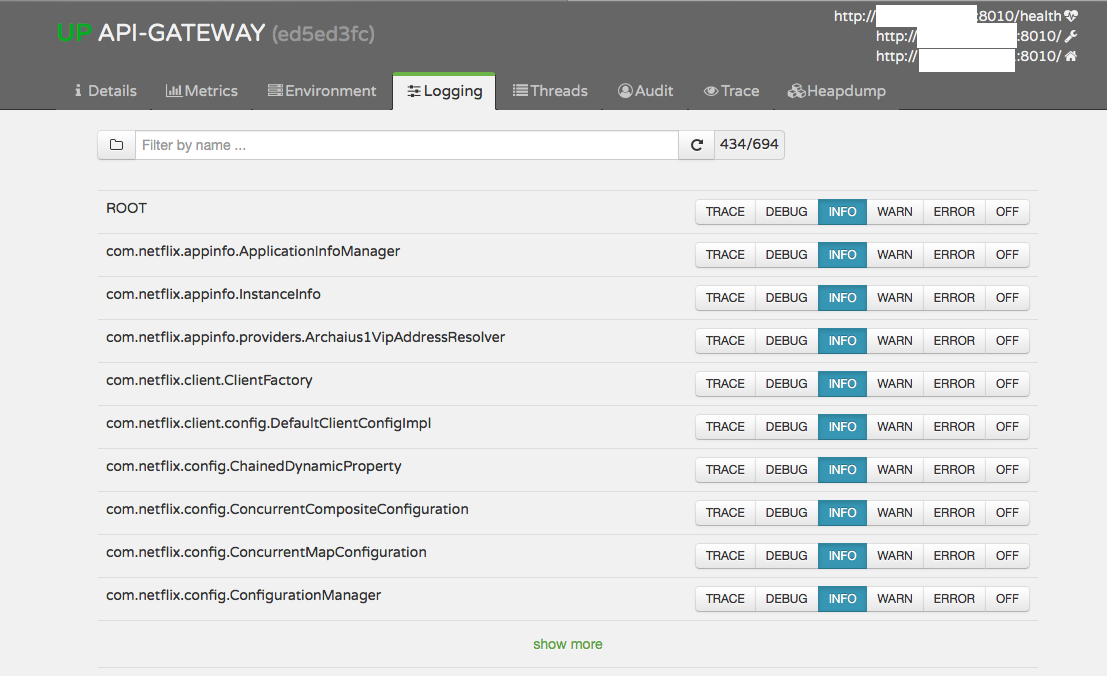
## 7 Spring boot admin dashboard view

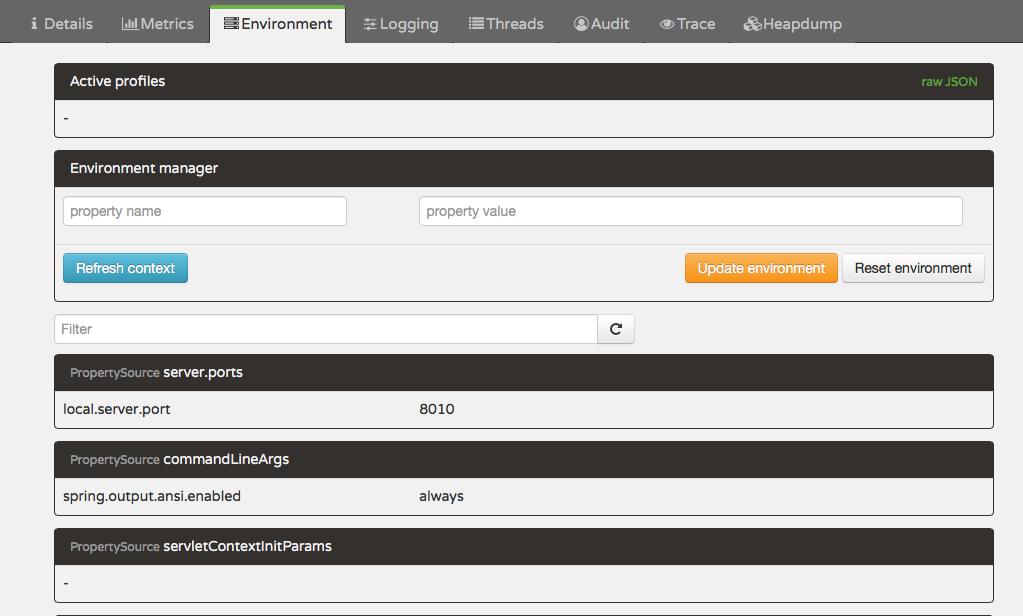
* To monitor via Spring Boot Admin server, invoke this URL running at different context path- http://localhost:8761/admin.



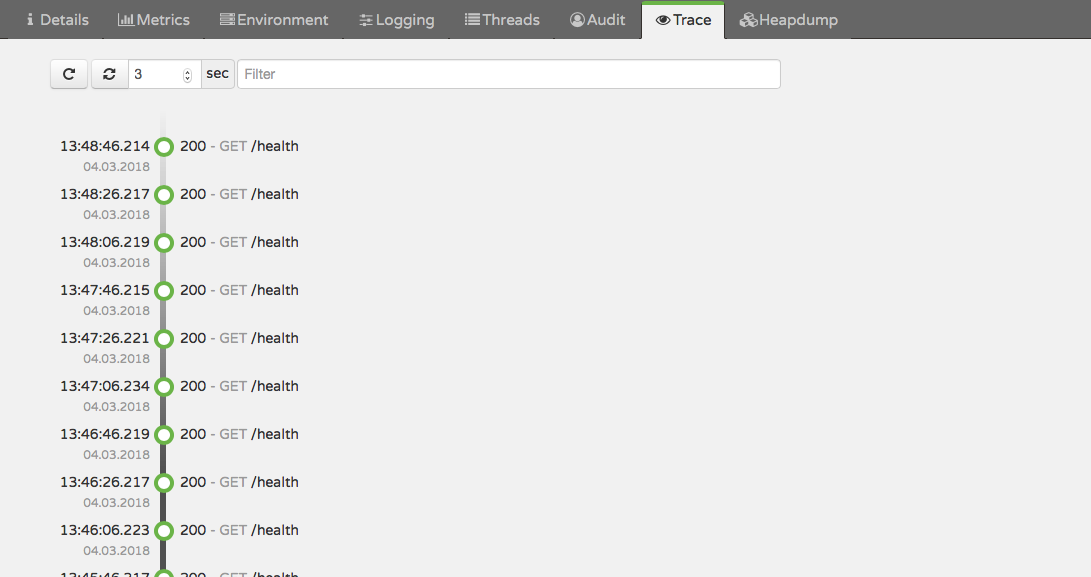
* This admin interface provides application overview, desktop notifications, application health check, log file browsing, JMX Beans, thread heap dump etc. To see individual application health and monitor its metrics, click on the detail button. It will take you to the admin dashboard of individual application.



* Using dashboard to manage log levels.  
  
* Using dashboard to manage runtime environment properties.



* Also you can use it to view HTTP traces.



* **Spring Cloud Config with File System Backend**

It is more like externalizing properties/resource file out of project codebase to an external service altogether so that any changes to any given property does not necessitate the re-deployment of service which is using that property. All such property changes will be reflected without redeploying the microservice.

**Why to Use Spring Cloud Config Server**

The idea of config server has come from the [12-factor app](https://12factor.net/config) manifesto related to the best practices guidelines of developing modern cloud-native applications. It suggests **to externalize properties or resource files out of server** where the values of those resources vary during runtime – usually different configurations that will differ in each environment.

As an example, let’s say one service is dependent on another service (invoked for specific business scenarios) and if that dependent service URL got changed to something else. Then usually we need to build and deploy our service with the updated URL. Now, if we go by the 12-factor app approach and if we read those config properties from external service, then we just need to update URL in the config server and refresh that client service configuration to use the updated URL.

* **Cloud Config Server**

The config properties are hosted on Spring Cloud Config Server.

To embed the config server we need to use @EnableConfigServer on our spring boot main class.

By default Spring Cloud Config uses a Git backend. A **backend** is where config properties are stored.

In this tutorial we will use File System Backend which is easier for getting started quickly and for testing.

To use File System Backend we need to specify followings in application.properties on the Cloud Config Server side:

* spring.profiles.active=native
* spring.cloud.config.server.native.searchLocations=<configFileLocations>

Where configFileLocations can be of classpath (using classpath:/ prefix) or locations on file system (using file:// prefix). If we don't specify this property then Spring Application default classpath are searched.

* **Cloud Config Clients**

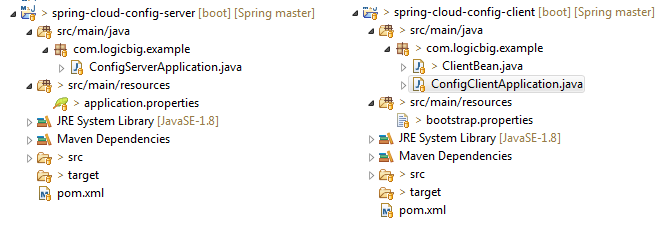
The configurations hosted by a Cloud Config Server can be accessed by a Config Client.  
A Config Client can be a microservice, registry server or any other component in microservice architecture.  
A Config Client can even be a standalone Spring Boot application.

The cloud externalized properties approach fits well with Spring [Environment](https://www.logicbig.com/tutorials/spring-framework/spring-core/spring-env-properties.html) abstractions. The access of different environment properties on the client side is based on Spring Boot [Loading Profile Specific properties](https://www.logicbig.com/tutorials/spring-framework/spring-boot/profile-specific-properties.html).

We need to specify following properties on the client side (bootstrap.properties):

* spring.cloud.config.uri where config server runs
* spring.profiles.active=myProfileName
* spring.application.name as comma-separated-list which are same as the names of property files used on the backend

***Example:***

******

***Cloud Config Server :***

***Pom.xml:***

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.logicbig.example</groupId>

<artifactId>spring-cloud-config-server</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

</parent>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

***application.properties:***

server.port=7777

spring.profiles.active=native

spring.cloud.config.server.native.searchLocations=file:///D:/app-config

At location D:/app-config we have following files:

D:\app-config>dir /B  
test-dev.properties  
test-prod.properties  
test2-dev.properties  
test2-prod.properties

#### test-dev.properties:

test.greeting=Hi developer!

#### test2-dev.properties:

test.msg=How is your coding going?

#### test-prod.properties

test.greeting=Hi there!

#### test2-prod.properties

test.msg=How are you doing?

### Boot main class

package com.logicbig.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer

@SpringBootApplication

public class ConfigServerApplication {

public static void main(String[] args) {

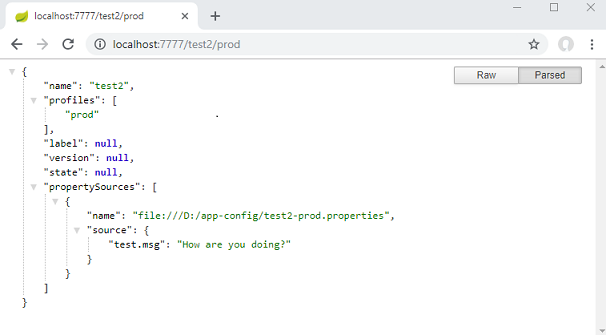
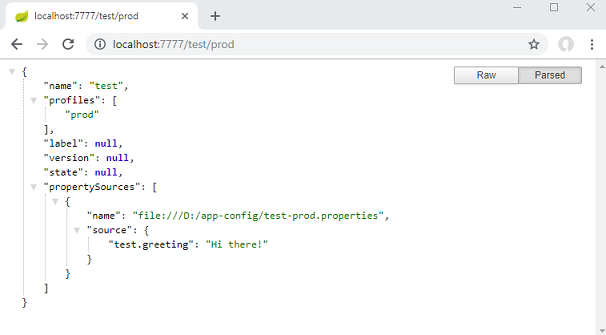
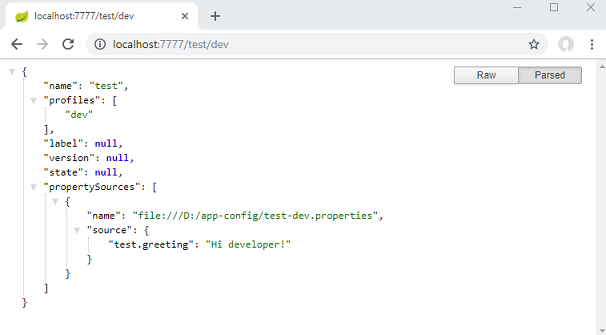
SpringApplication.run(ConfigServerApplication.class, args);

}

}

Run above class from your IDE.

Now we can access the backend via http://localhost:7777/{application}/{profile}



We can also access properties using following URLs:

/{application}-{profile}.yml

/{application}-{profile}.properties

***Cloud Config Client***

We are going to create a simple standalone Boot application as a Config Client.

#### pom.xml

<project .....>  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.logicbig.example</groupId>  
 <artifactId>spring-cloud-config-client</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.5.RELEASE</version>  
 </parent>  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-client</artifactId>  
 </dependency>  
 </dependencies>  
 <dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Finchley.SR1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
</project>

#### src/main/resources/bootstrap.properties

spring.cloud.config.uri=http://localhost:7777

spring.application.name=test,test2

spring.profiles.active=dev

### Example client bean

package com.logicbig.example;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component

public class ClientBean {

@Value("${test.greeting}")

private String msg1;

@Value("${test.msg}")

private String msg2;

@PostConstruct

public void postConstruct() {

System.out.println(msg1);

System.out.println(msg2);

}

}

### Main class

package com.logicbig.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ConfigClientApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApplication.class, args);

}

}

Run above class from your IDE.

### Output

Hi developer!

How is your coding going?

On the client side, if we change the active profile property to spring.profiles.active=prod (in bootstrap.properties) and run the client again:

Hi there!

How are you doing?

* **Spring Cloud Config with Git Backend**

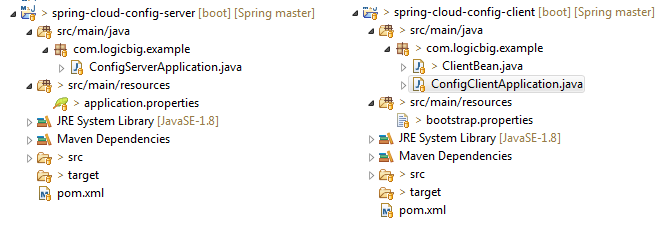
By default Spring Cloud Config uses a Git backend, that means on the config server side, we don't have to activate any profile.  
  
To specify git repository location we need to use following property in application.properties:

* spring.cloud.config.server.git.uri=<gitLocation>

where gitLocation can be local (e.g. file:///D:/example-app-config) or remote (e.g. http://example-git-config-repo).

At the specified location we need to add property files and commit them to Git before the clients can access them

***Example:***

******

***Cloud Config Server :***

***Pom.xml:***

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.logicbig.example</groupId>

<artifactId>spring-cloud-config-server</artifactId>

<version>1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

</parent>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Finchley.SR1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

***application.properties:***

server.port=7777

spring.cloud.config.server.git.uri=https://github.com/incredibledevelopers/config-server-repo

At location D:/app-config we have following files:

D:\app-config>dir /B  
test-dev.properties  
test-prod.properties  
test2-dev.properties  
test2-prod.properties

#### test-dev.properties:

test.greeting=Hi developer!

#### test2-dev.properties:

test.msg=How is your coding going?

#### test-prod.properties

test.greeting=Hi there!

#### test2-prod.properties

test.msg=How are you doing?

Commit above files:

Joe@jpc MINGW64 /d/example-app-config (master)

$ ls -la

total 48

drwxr-xr-x 1 Joe 197610 0 Jan 29 00:18 ./

drwxr-xr-x 1 Joe 197610 0 Jan 29 00:09 ../

drwxr-xr-x 1 Joe 197610 0 Jan 29 00:17 .git/

-rw-r--r-- 1 Joe 197610 34 Oct 16 16:26 test2-dev.properties

-rw-r--r-- 1 Joe 197610 27 Oct 15 15:41 test2-prod.properties

-rw-r--r-- 1 Joe 197610 27 Oct 16 15:07 test-dev.properties

-rw-r--r-- 1 Joe 197610 23 Oct 15 15:41 test-prod.properties

Joe@jpc MINGW64 /d/example-app-config (master)

$ git add -A

Joe@jpc MINGW64 /d/example-app-config (master)

$ git commit -m "adding props"

[master (root-commit) 8755c6e] adding props

4 files changed, 4 insertions(+)

create mode 100644 test-dev.properties

create mode 100644 test-prod.properties

create mode 100644 test2-dev.properties

create mode 100644 test2-prod.properties

### Boot main class

package com.logicbig.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer

@SpringBootApplication

public class ConfigServerApplication {

public static void main(String[] args) {

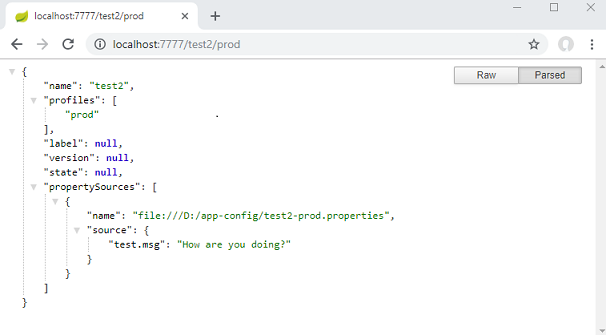
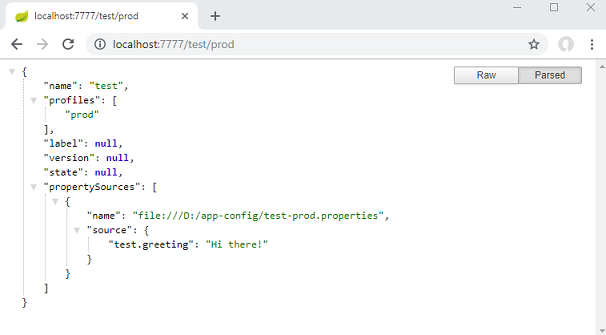
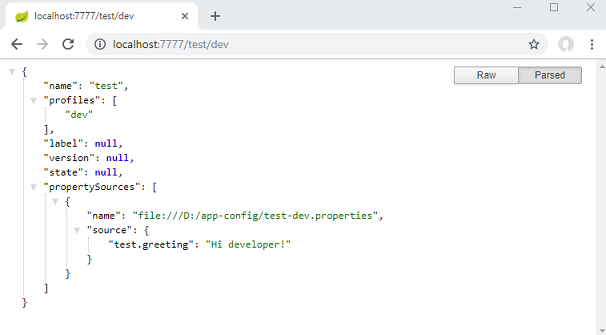
SpringApplication.run(ConfigServerApplication.class, args);

}

}

Run above class from your IDE.

Now we can access the backend via http://localhost:7777/{application}/{profile}



We can also access properties using following URLs:

/{application}-{profile}.yml

/{application}-{profile}.properties

***Cloud Config Client***

We are going to create a simple standalone Boot application as a Config Client.

#### pom.xml

<project .....>  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>com.logicbig.example</groupId>  
 <artifactId>spring-cloud-config-client</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.5.RELEASE</version>  
 </parent>  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-client</artifactId>  
 </dependency>  
 </dependencies>  
 <dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Finchley.SR1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
</project>

#### src/main/resources/bootstrap.properties

spring.cloud.config.uri=http://localhost:7777

spring.application.name=test,test2

spring.profiles.active=dev

### Example client bean

package com.logicbig.example;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;

@Component

public class ClientBean {

@Value("${test.greeting}")

private String msg1;

@Value("${test.msg}")

private String msg2;

@PostConstruct

public void postConstruct() {

System.out.println(msg1);

System.out.println(msg2);

}

}

### Main class

package com.logicbig.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ConfigClientApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigClientApplication.class, args);

}

}

Run above class from your IDE.

### Output

Hi developer!

How is your coding going?

On the client side, if we change the active profile property to spring.profiles.active=prod (in bootstrap.properties) and run the client again:

Hi there!

How are you doing?

* **How to Deploy Spring Boot Application to Cloud Foundry Platform**

Now a days [**Cloud Computing**](https://howtodoinjava.com/cloud/java-cloud-development-introduction-and-tools/) and [**Microservice**](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/) have become very popular concept and almost all the organizations are investing and adapting it very fast. Currently there are only few popular cloud providers in the market and **Cloud Foundry** is one of them. It is a [*PaaS service*](https://en.wikipedia.org/wiki/Platform_as_a_service) where we can easily deploy and manage our applications and the Cloud Foundry will take care of the rest of the cloud based offerings like scalability, high availability etc.

Today we will learn to deploy spring boot application in cloud foundry starting from setting up cloud foundry in local workstation. There are many cloud foundry distributions currently available and in this article, we will mainly concentrate on the **Pivotal Cloud Foundry** platform called as [Pivotal Web Services](https://run.pivotal.io/).

## What is Cloud Foundry

Cloud Foundry is an open-source platform as a service (PaaS) that provides you with a choice of clouds, developer frameworks, and application services. It is open source and it is governed by the Cloud Foundry Foundation. The original Cloud Foundry was developed by VMware and currently it is managed by Pivotal, a joint venture company by GE, EMC and VMware.

Now since Cloud Foundry is open source product many popular organizations currently provides this platform separately and below are the list of current certified providers.

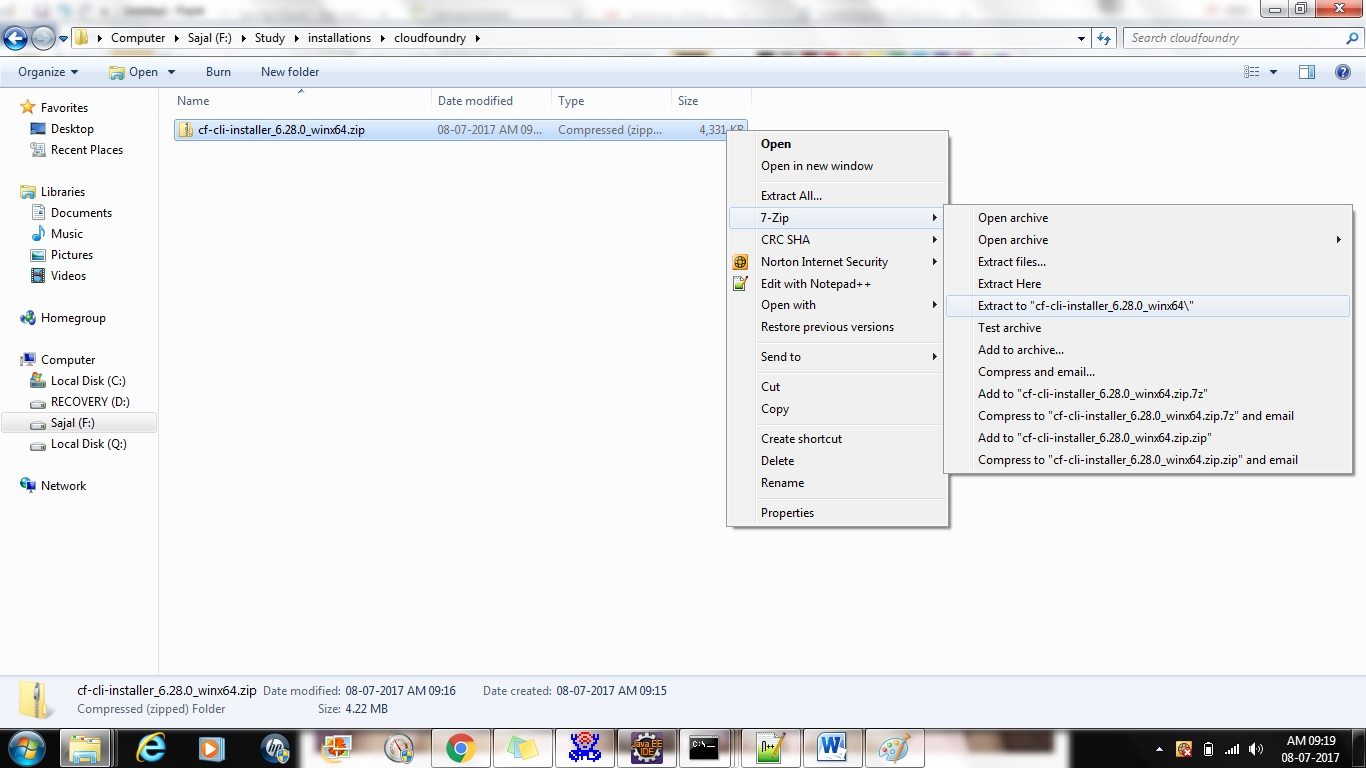
* Pivotal Cloud Foundry
* IBM Bluemix
* HPE Helion Stackato 4.0
* Atos Canopy
* CenturyLink App Fog
* GE Predix
* Huawei FusionStage
* SAP Cloud Platform
* Swisscom Application Cloud

## Cloud Foundry Installation for Windows

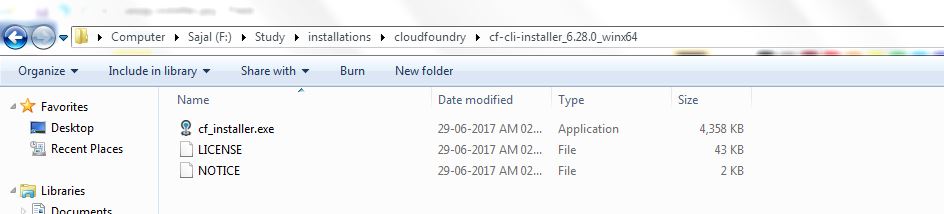
Here are the installation steps for Windows, for other operating system, cloud foundry has a very good documentation which we can easily follow.

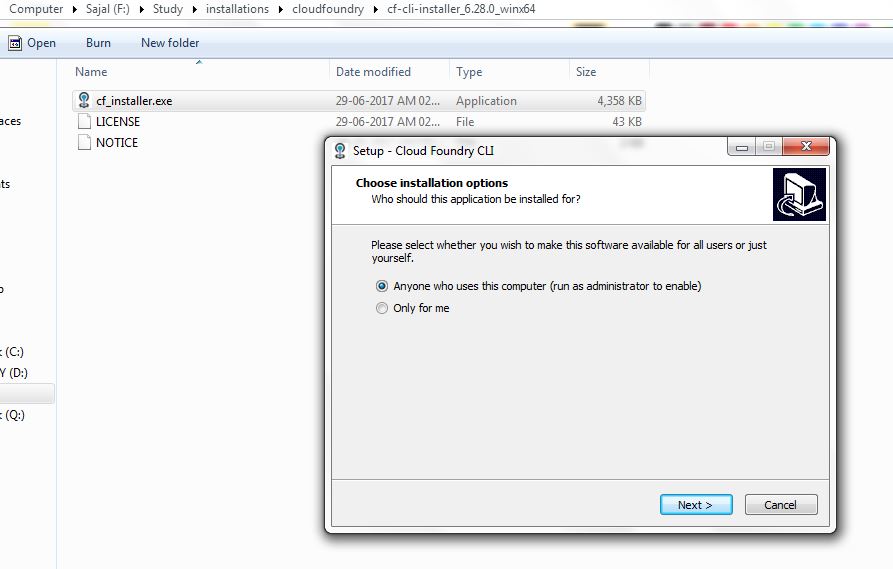
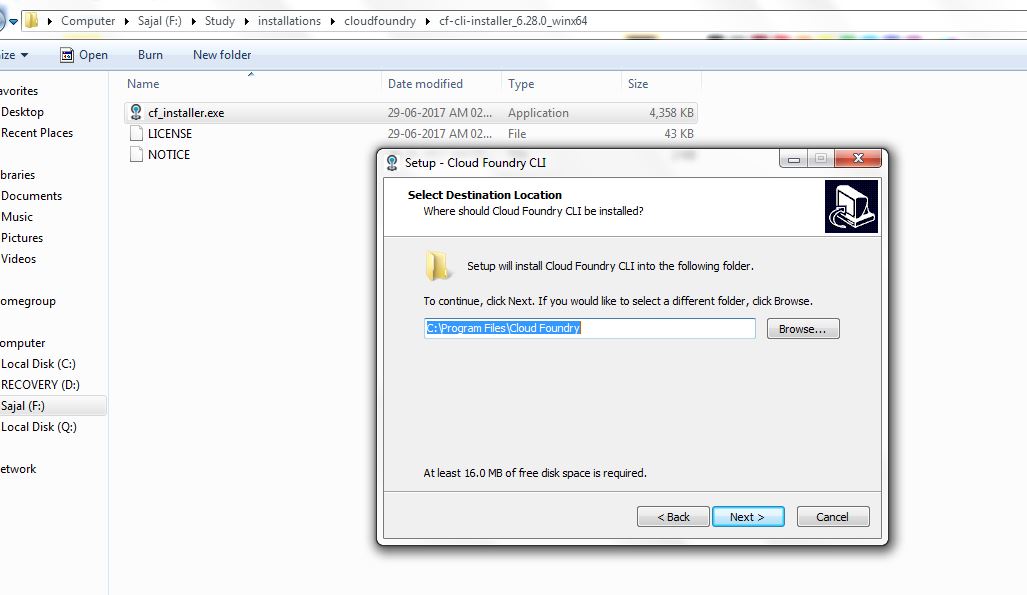
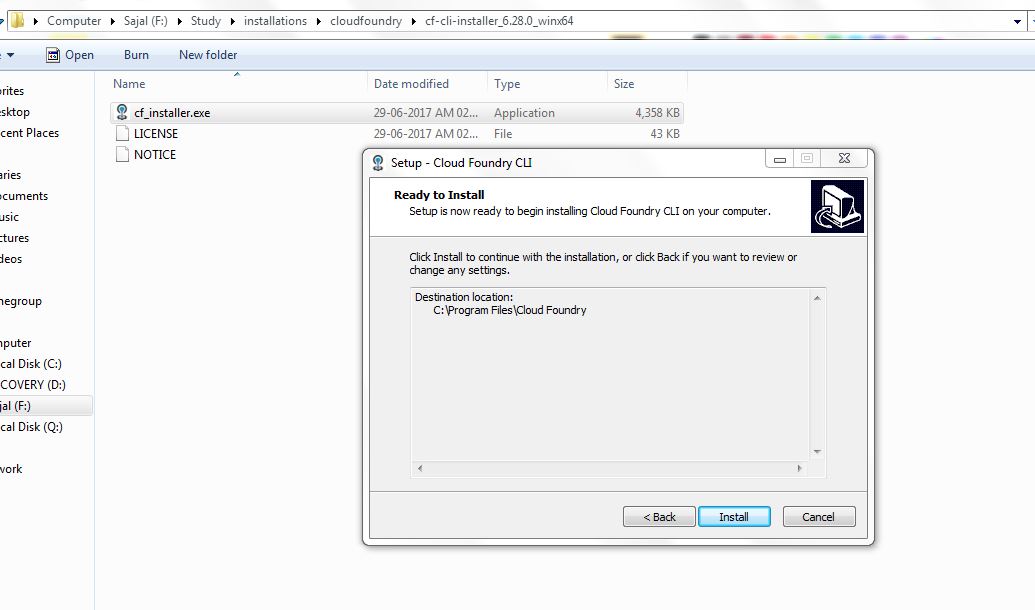
The cloud foundry works very well from command prompt and cloud foundry has provided one command line tool called cf which does almost all the activities for us. So to make this tool (cf command) available in local workstation, first we need to install and configure the Cloud Foundry Command line (CLI) interface.

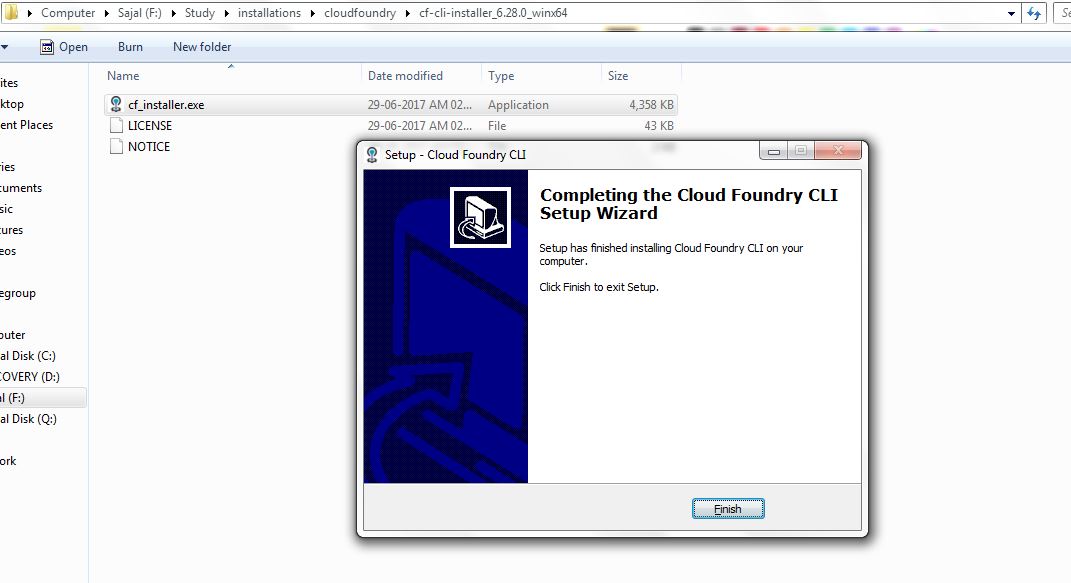
1. Download the [CF Windows installer](https://cli.run.pivotal.io/stable?release=windows64&source=github). It will prompt for the download. Save the zip file distribution.
2. Unpack the zip file to a suitable place in your workstation.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/unzip-installer.jpg) Fig.Unzip Installer

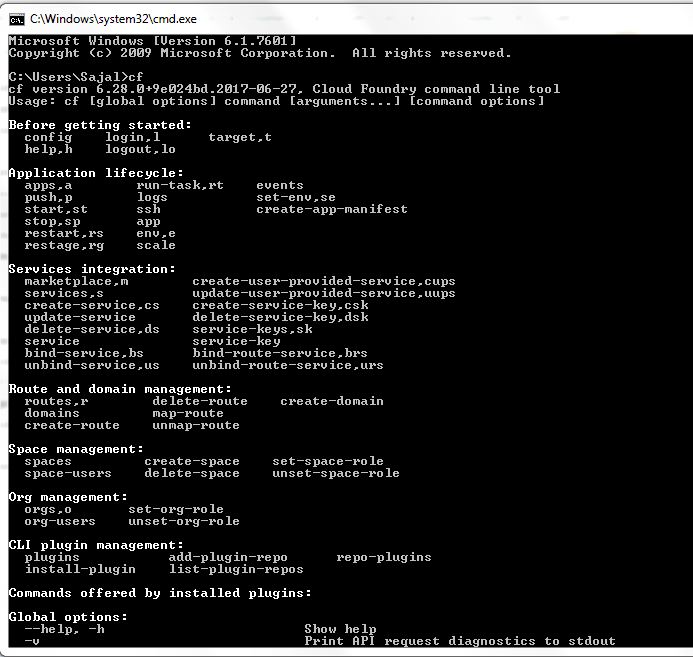
1. After successfully **unzip** operation, double cick on the cf CLI executable.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/installer-location.jpg)CF Installer Location

1. When prompted, click **Install**, then Close. Here are the sample steps for the same. This is very straight froward, you can select the default values.[](https://howtodoinjava.com/wp-content/uploads/2017/07/s1.jpg)Step 1[](https://howtodoinjava.com/wp-content/uploads/2017/07/s2.jpg)Step 2[](https://howtodoinjava.com/wp-content/uploads/2017/07/s3.jpg)Step 3

[](https://howtodoinjava.com/wp-content/uploads/2017/07/s4.jpg)Step 4

1. Verify the installation by opening a terminal window and type cf. If your installation was successful, the cf CLI help listing appears. This indicates that you are ready to go with any cloud foundry platform from your local workstation.

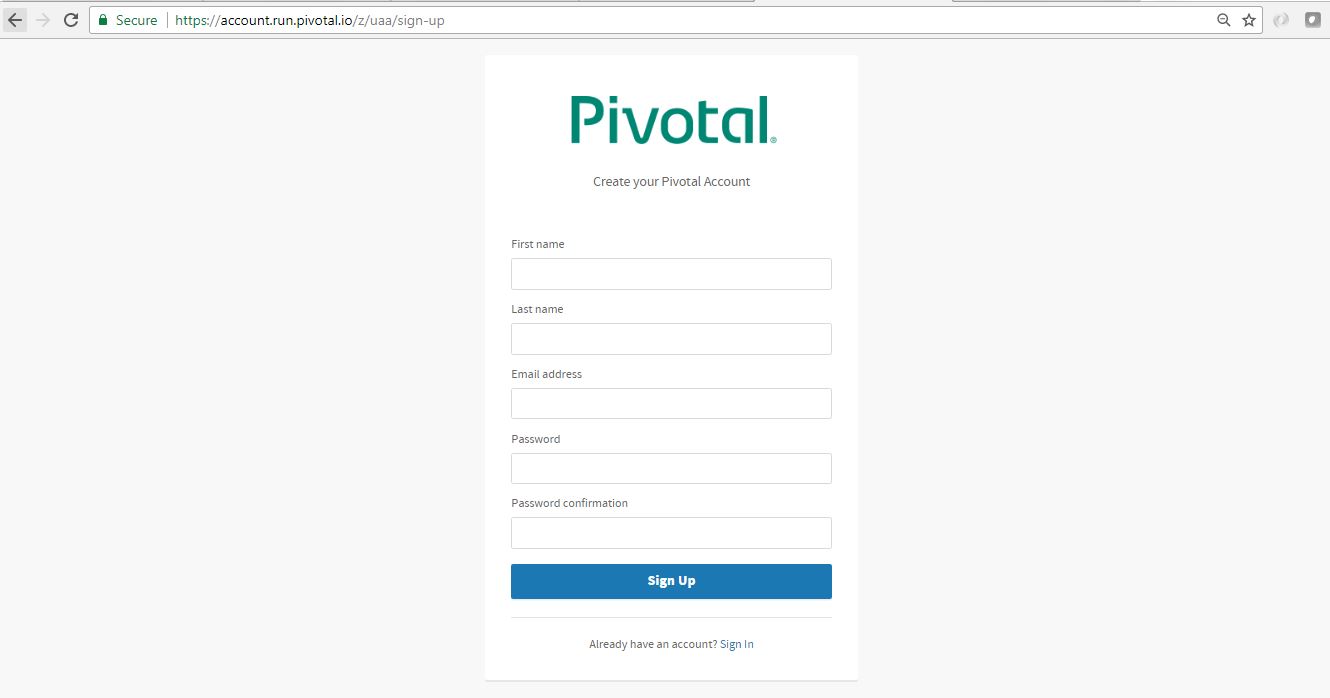
[](https://howtodoinjava.com/wp-content/uploads/2017/07/cf_install_verify.jpg)

Cloud Foundry Installation Verification

We will now proceed with Pivotal Web service account sign up and development of a sample application and push to cloud foundry.

## Setup PWS Console

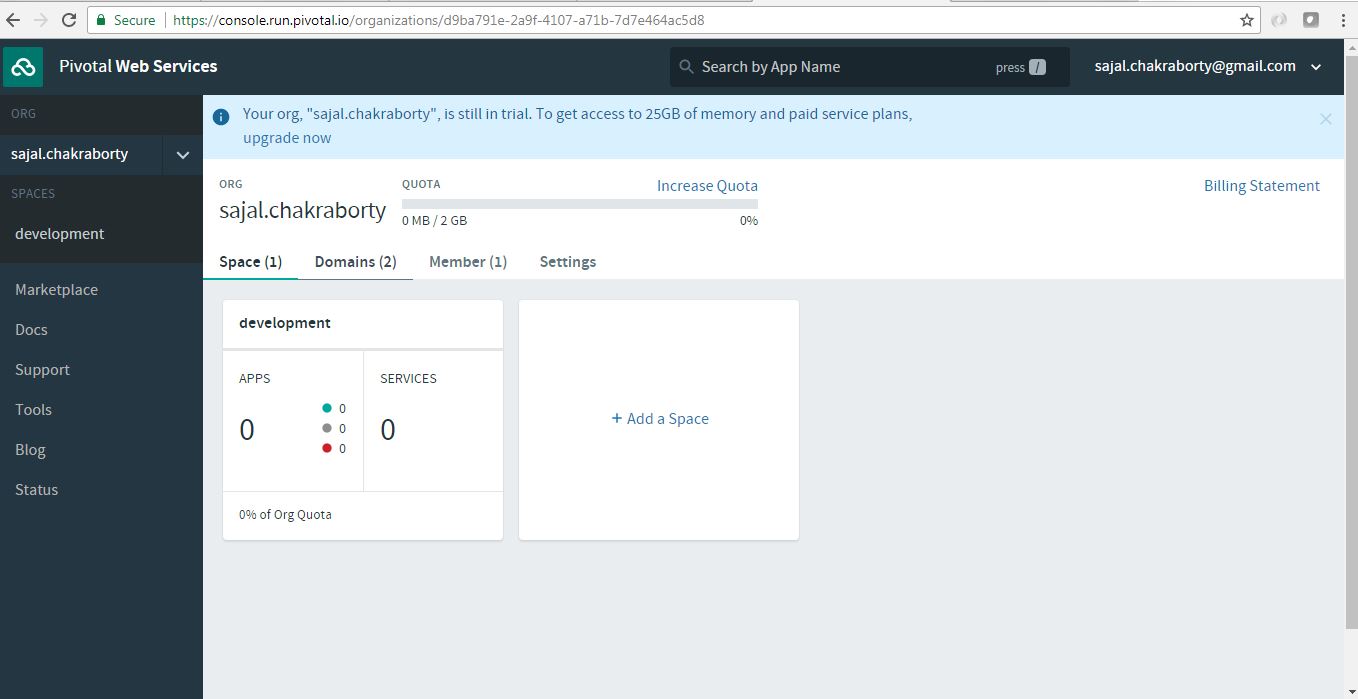
Now we need to create one account in pivotal in order to deploy our application in Pivotal Cloud Foundry Platform. We need to [register](https://account.run.pivotal.io/z/uaa/sign-up) in the below page to start with the sign up process. It is free and it will just ask some very common things like email address, name etc.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/pivotal_console_signup.jpg)**Fig.Pivotal web service console signup**

Once sign up is completed, we can log into the console through the [log in screen](https://login.run.pivotal.io/login) of the pivotal web service console.

After providing logon credentials successfully we will get into the **cloud foundry console** where we can see all the deployed applications can monitor the applications and do many more activities. Here we need to [add org and space](https://docs.cloudfoundry.org/concepts/roles.html) etc. which is very straight forward and self-describing. Below is one sample console screen after login.

Currently no applications are deployed as we have not yet pushed any application.

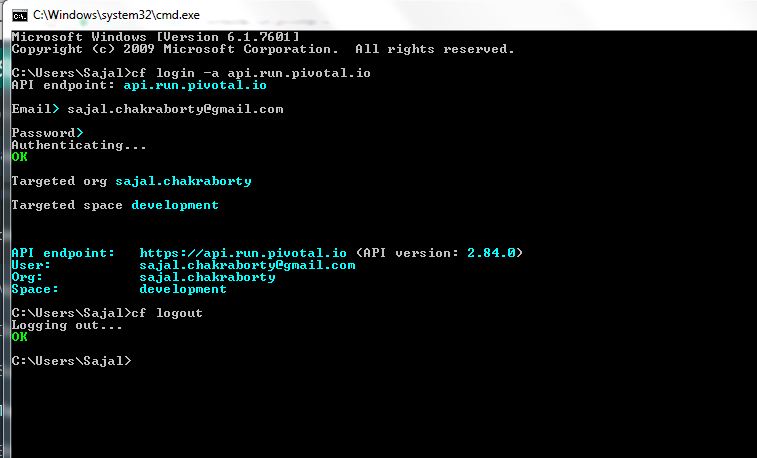
[](https://howtodoinjava.com/wp-content/uploads/2017/07/pcf_console_without_any_apps.jpg)**Fig.Console of PWS after login**

#### Login and logout from PWS Console using CLI

1. **Login to PWS** – We will use cf login -a api.run.pivotal.io command to login to pivotal web service console from CLI tool that we have installed in our local workstation. It will logon the CLI tool to PWS platform so that we can deploy and manage our applications from our workstation. After giving command, it will ask for registered email and password and once provided successfully, it will logon to the platform.
2. **Logout from PWS Console** – We will use command cf logout to logout from the platform, once we have all the work done for that session.

|  |
| --- |
| //To login    >> cf login -a api.run.pivotal.io    //To logout    >> cf logout |

Here is the login and logout looks like from command prompt.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/Login-logout-with-CF.jpg)

**Fig.Login logout with CF**

## Create Spring Boot Application

We will now create one Spring boot application and will deploy to PWS console and access from Cloud Foundry itself. We will create an application which will expose one simple REST endpoint, which we will test from our workstation once deployed in **Pivotal Web Service Platform**.

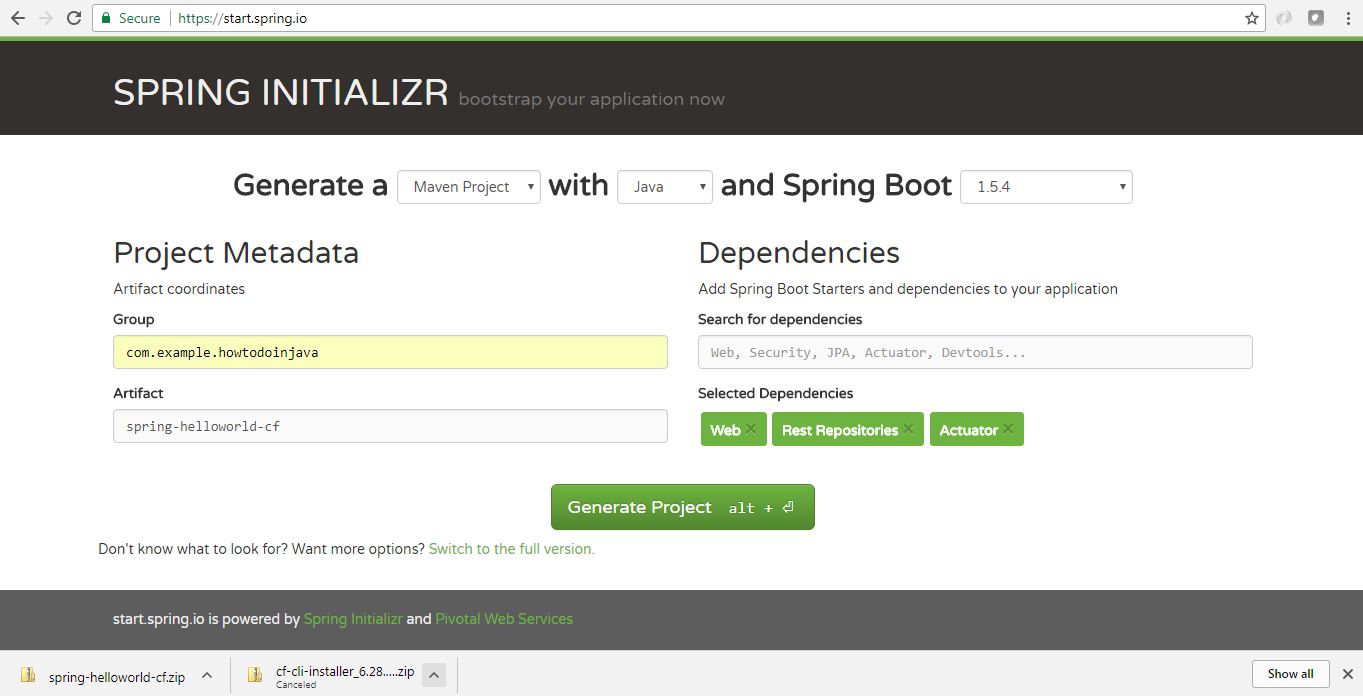
#### Technology Stack

We will use below technology stack for the spring boot application development and testing.

* Spring Boot
* Spring REST
* Maven
* Eclipse
* Cloud Foundry CLI
* Web Browser

#### Generate Spring boot application

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.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/Spring-Boot-Project-Generation.jpg)**Fig.Spring Boot Project Generation**

Import the project to eclipse as existing maven project. Let maven download the dependencies and setup class path entries for you.

#### Add REST Controller and Endpoint

We need to add simple REST endpoint to test the deployment from cloud foundry. Open boot application class that has already provided by the automated project generation SpringHelloworldCfApplication.java and add the below lines to add one simple endpoint which will just echo something based on the input.

Final class will look like below.

|  |
| --- |
| package com.example.howtodoinjava.springhelloworldcf;   import java.util.Date;   import org.springframework.beans.factory.annotation.Value;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestParam;  import org.springframework.web.bind.annotation.RestController;   @SpringBootApplication  public class SpringHelloworldCfApplication {       public static void main(String[] args) {          SpringApplication.run(SpringHelloworldCfApplication.class, args);      }  }    @RestController  class MessageRestController {        @RequestMapping("/hello")      String getMessage(@RequestParam(value = "name") String name) {          String rsp = "Hi " + name + " : responded on - " + new Date();          System.out.println(rsp);          return rsp;      }  } |

#### Project Configuration

Add Context path and required properties in bootstrap.properties file in src\main\resources directory and add two properties there.

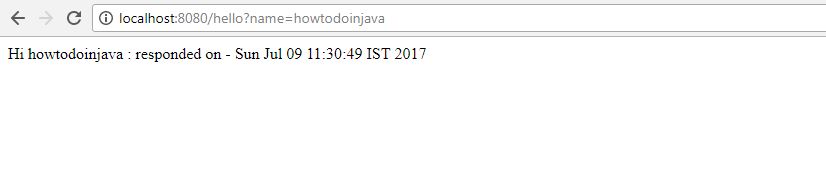
|  |
| --- |
| server.contextPath = /hello  management.security.enabled = false |

This will set one context path /hello for the application and management.security.enabled=false will disable security for management endpoints of spring boot like /env, /refresh etc.

#### Test locally

Finally build and test the application in Local in an embedded tomcat container. To do this, start the application as spring boot application.

Go to browser and type http://localhost:8080/hello?name=howtodoinjava. It should echo the name along with some greeting message and response process time.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/LocalTestinBrowser.jpg)

**Fig.REST API Output**

Now we will push [deploy] the application in the pivotal cloud foundry where we have registered already.

## Deploy Spring Boot Application in Cloud Foundry Platform

As we have Cloud Foundry CLI already configured, we will use CLI cf push command to deploy the application in cloud foundry console.

#### Login to PWS Console

To do that open command prompt and go to maven application’s home directory and use cf login -a api.run.pivotal.io command to login to pivotal web service console.

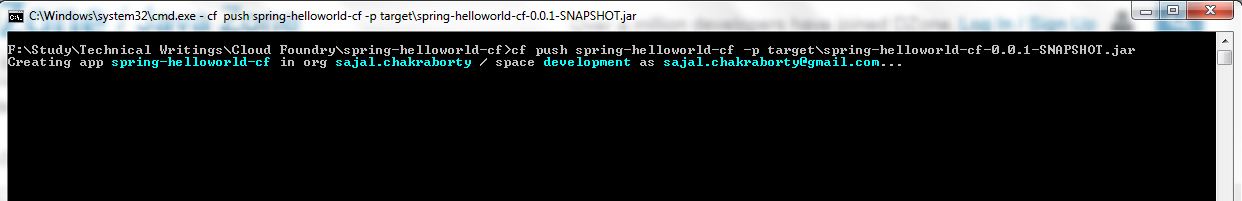
It will ask for the registered credentials and finally log on to the console.

#### Push Application to Console

Now we need to push the application with the command cf push.

|  |
| --- |
| cf push spring-helloworld-cf -p target\spring-helloworld-cf-0.0.1-SNAPSHOT.jar |

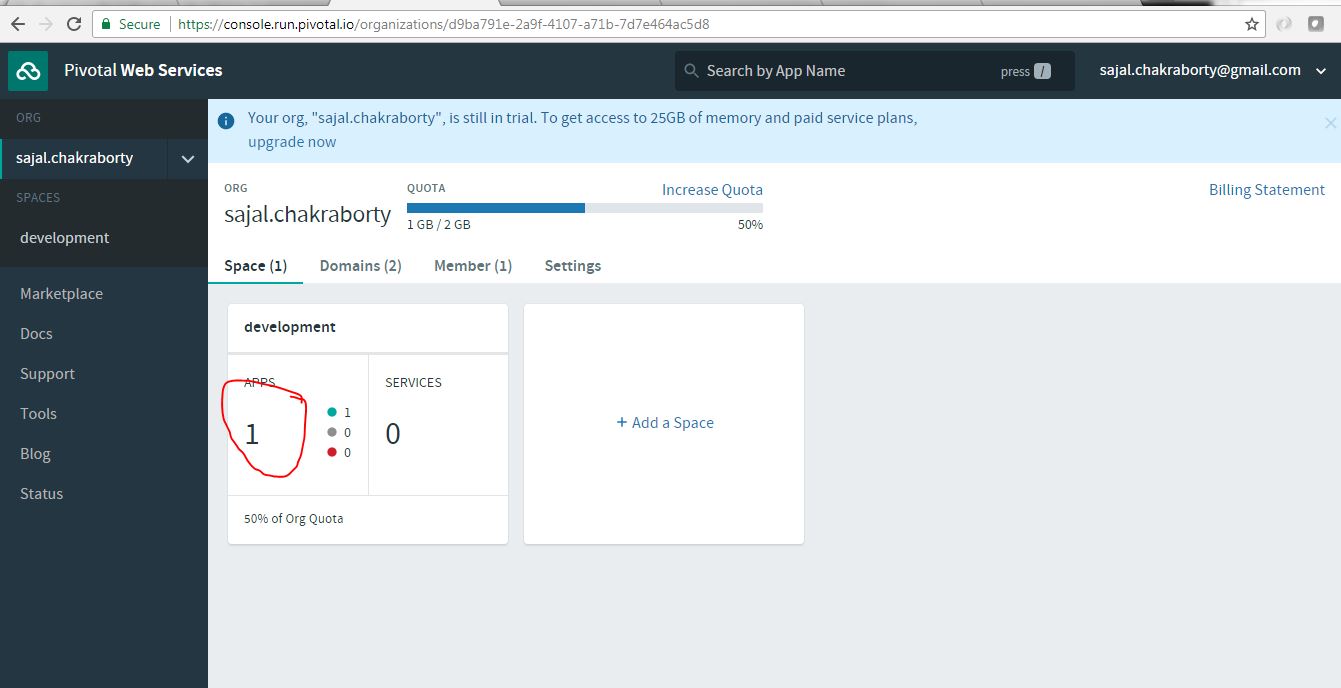
This will deploy the application to the already logged in PWS console from the previous step.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/Deploy-cf-push.jpg)**Fig.Deployment by cf push**

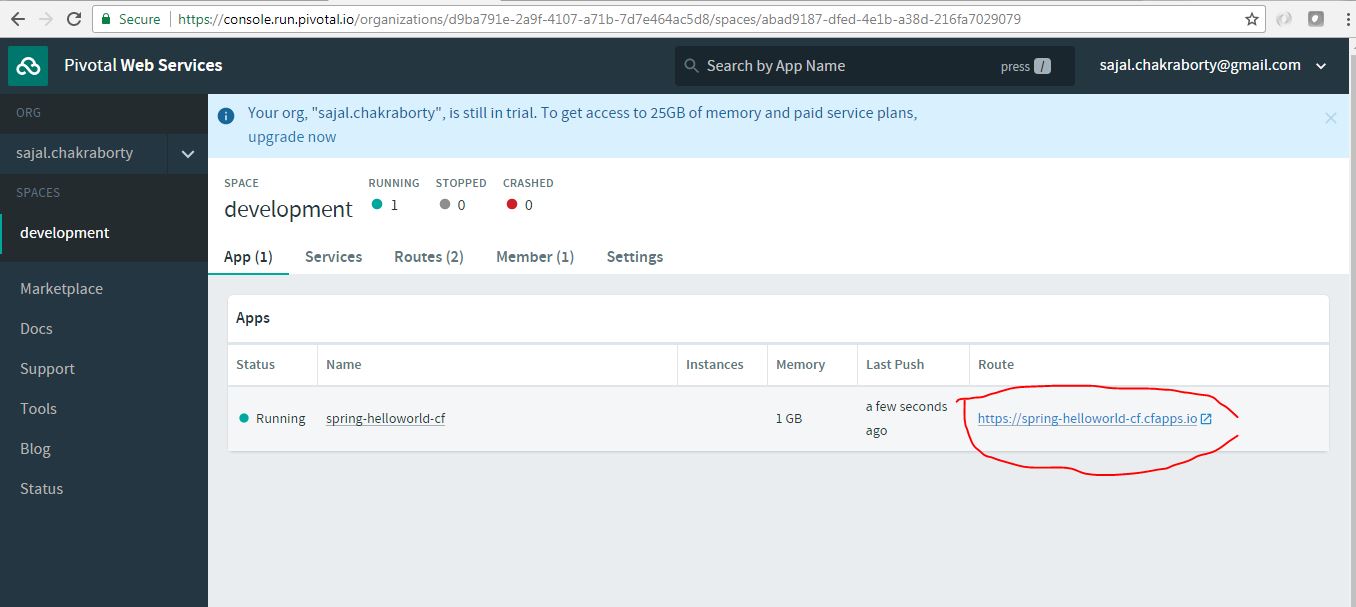
Read the full console log for push command in [attached log file](https://howtodoinjava.com/wp-content/uploads/2017/07/Cf_push_Command_log.txt).

#### Verify Application Deployment

Verify into PWS console to check that the newly deployed application is showing up. If everything went fine in the previous steps then, screen will look like this.

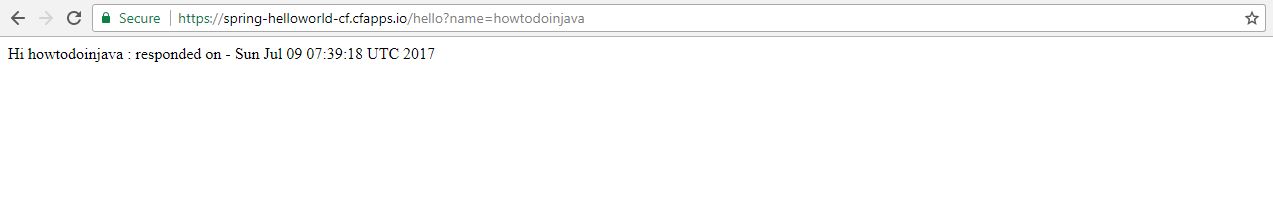
[](https://howtodoinjava.com/wp-content/uploads/2017/07/PWS_CONSOLE_APP_DEPLOYED.jpg)Application deployed in PWS – count increased.

Now click on the Apps section highlighted in the previous step to go to the application details screen. Below view will be shown and it will show the url where application has been deployed as highlighted. Note this url to test it from browser. In this case it will be somethig like https://spring-helloworld-cf.cfapps.io/. This URL will change based on the application name we choose.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/PWS_CONSOLE_APP_DEPLOYED_app_details-1.jpg)Application URL published in PWS console.

#### Test REST Endpoint

Now to the browser and access the application with the url host published in the cf console. For this application url is https://spring-helloworld-cf.cfapps.io/hello?name=howtodoinjava.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/Application-accessed-from-cf.jpg)Application accessed from cf directly

Congratulations!! You have successfully deployed your first spring boot application into Pivotal Cloud Foundry Platform.