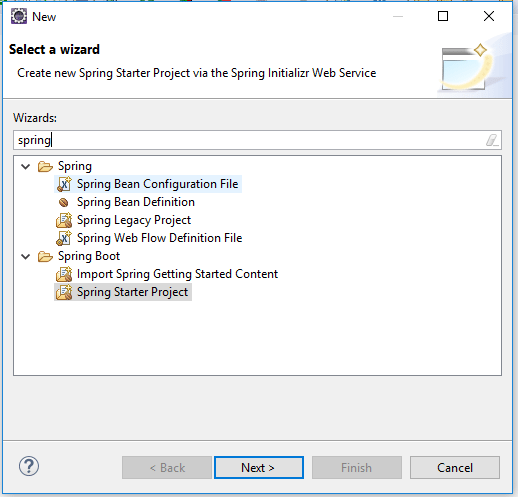
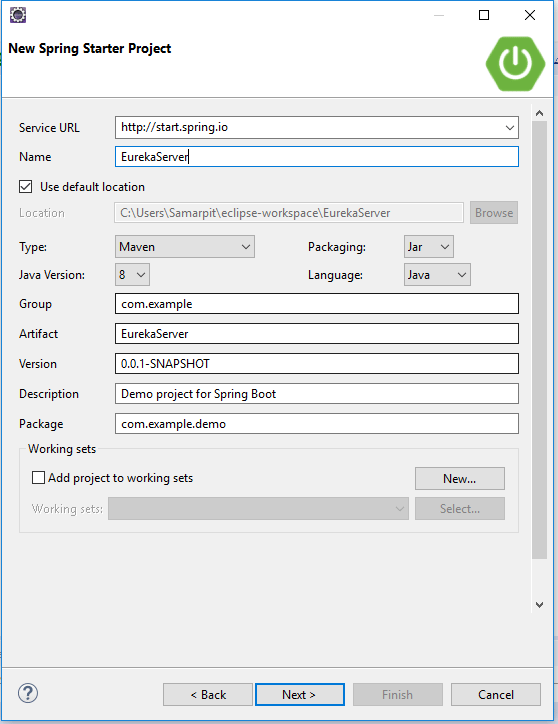
**Spring Boot Microservices: Creating a Eureka Service**

To begin with, create a EurekaServer Spring Starter Project in Eclipse IDE. Click on **Spring Starter Project**and click on **Next.**

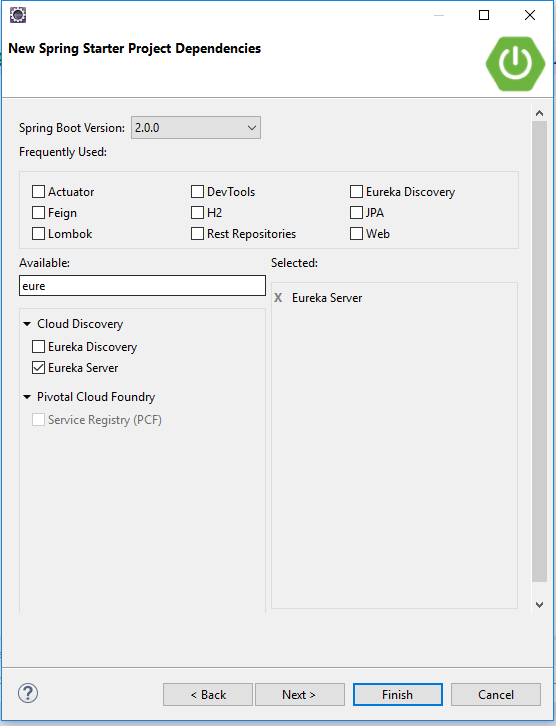


Name your Spring Starter Project as **EurekaServer**and other Information will be filled**automatically.**

**Note:-**Make sure your Internet is connected otherwise it will show an error.

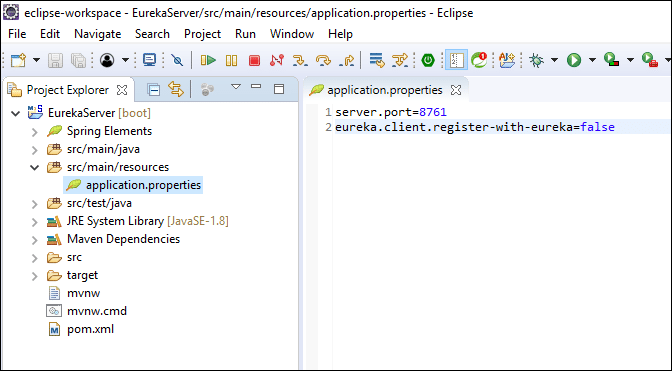


Now, select **Eureka Server**as a dependency and click on **Finish.**



Now, modify EurekaServer/src/main/resources/application.properties file to add a port number and disable registration.

server.port=8761  
eureka.client.register-with-eureka=false



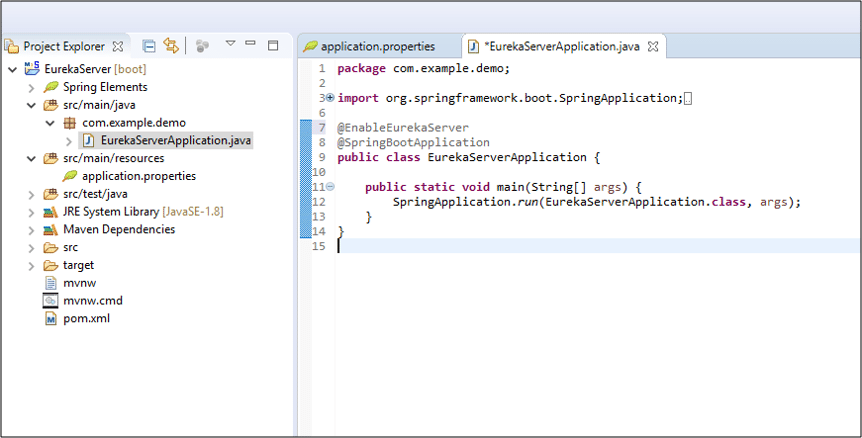
Open EurekaServer/src/main/java/com/example/EurekaServiceApplication.java and add @EnableEurekaServer above @SpringBootApplication.

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer

@SpringBootApplication

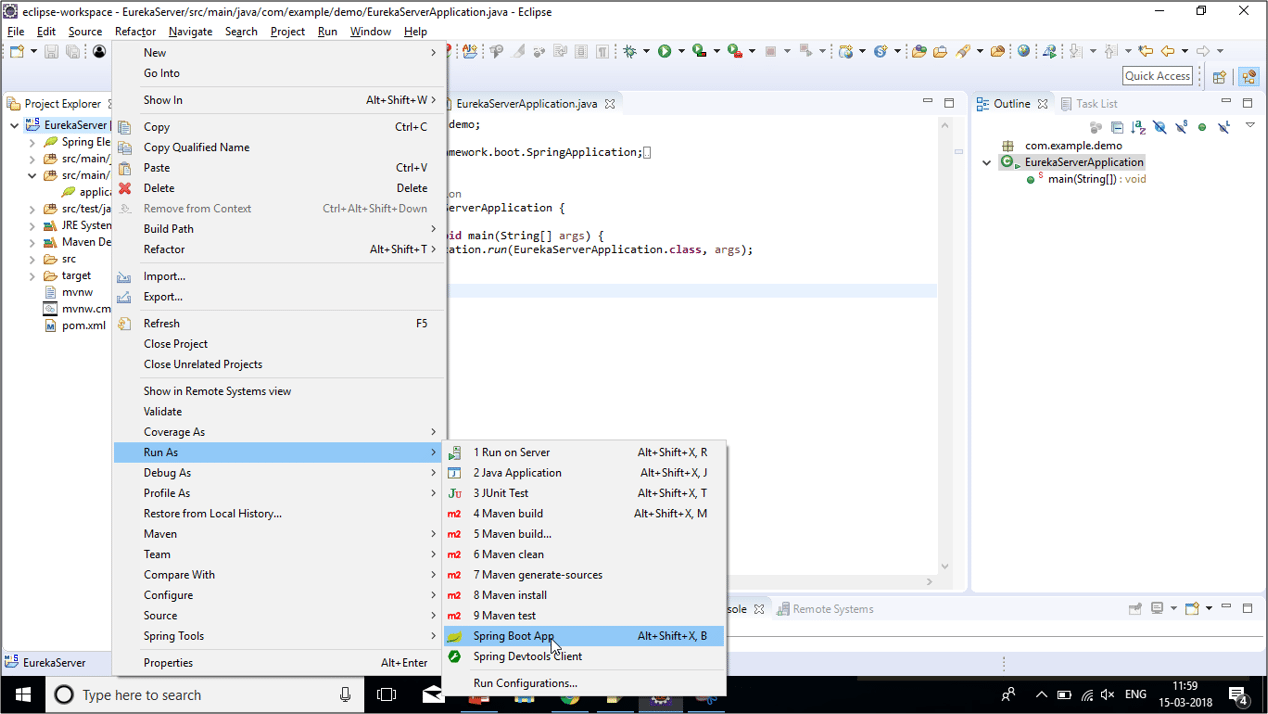
This annotation will configure a registry that will allow other applications to communicate.

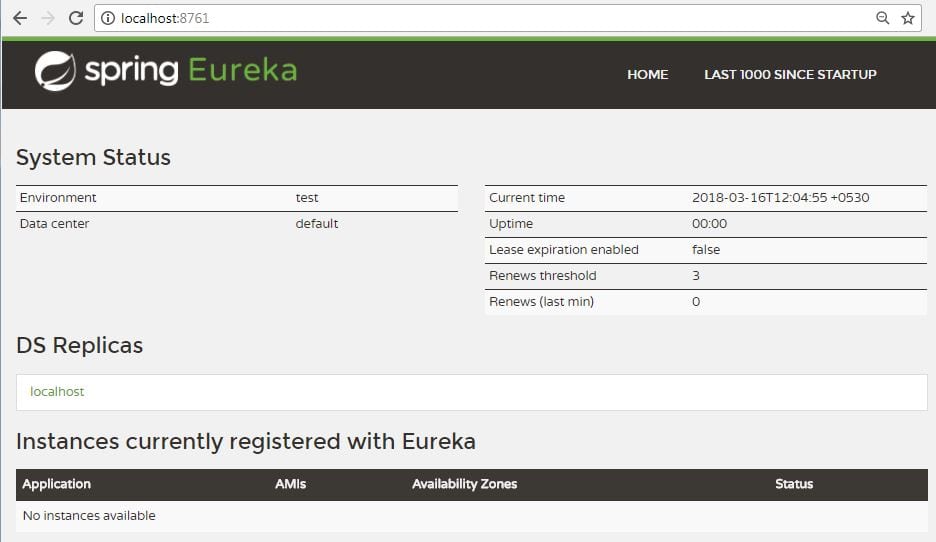
To start the Application: Right Click on the **Project** –> **Run As** –> Click on “**Spring Boot App**“

**[Microservices Architecture Training](https://www.edureka.co/microservices-architecture-training" \t "_blank)**

* *[Instructor-led Sessions](https://www.edureka.co/microservices-architecture-training" \t "_blank)*
* *[Real-life Case Studies](https://www.edureka.co/microservices-architecture-training" \t "_blank)*
* *[Assignments](https://www.edureka.co/microservices-architecture-training" \t "_blank)*
* *[Lifetime Access](https://www.edureka.co/microservices-architecture-training" \t "_blank)*

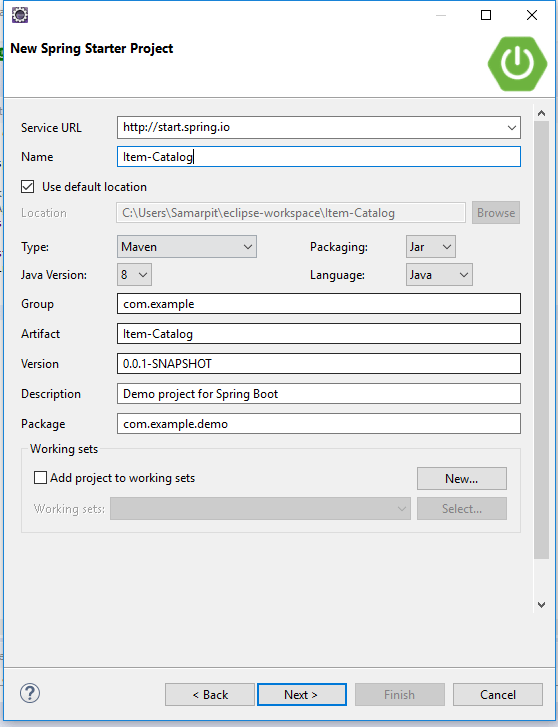
[Explore Curriculum](https://www.edureka.co/microservices-architecture-training" \t "_blank)

After it starts, you should be able to open http://localhost:8761 and see there are no services available.

Now open http://localhost:8761. Here Spring Eureka Server will open and will show no service will be running.

**Spring Boot Microservices: Creating an Item Catalog Service**

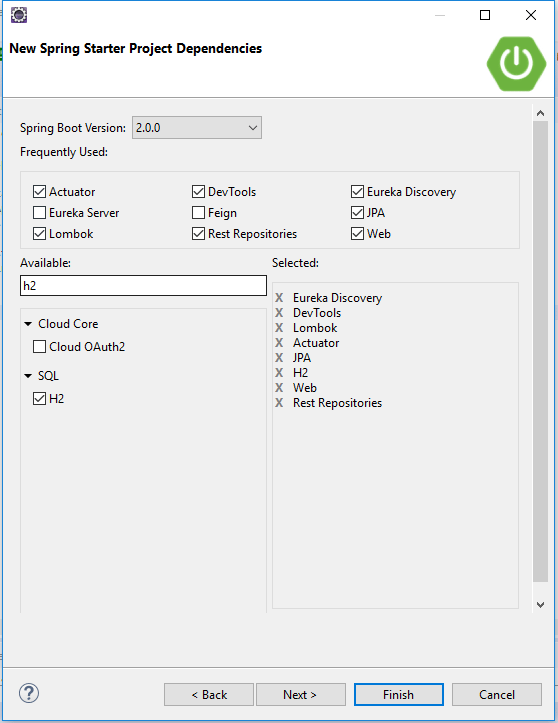
Again create a new project. Use Item-catalog-service for the artifact name and click on **Next**.



Add the following dependencies:

* **Actuator**: features to help you monitor and manage your application
* **Eureka** **Discovery**: for service registration
* **JPA**: to save/retrieve data
* **H2**: an in-memory database
* **Rest** **Repositories**: to expose JPA repositories as REST endpoints
* **Web**: Spring MVC and embedded Tomcat
* **DevTools**: to auto-reload the application when files change
* **Lombok**: to reduce boilerplate code

Click on **Finish.**



Now, create an Item entity, toItemCatalogServiceApplication.java  . The code below assumes you’re putting all classes in the same file.

@Data

@AllArgsConstructor

@NoArgsConstructor

@ToString

@Entity

class Item {

public Item(String name) {

this.name = name;

}

@Id

@GeneratedValue

private Long id;

private String name;

}

@RepositoryRestResource

interface ItemRepository extends JpaRepository<Item, Long> {}

@Component

class ItemInitializer implements CommandLineRunner {

private final ItemRepository ItemRepository;

ItemInitializer(ItemRepository itemRepository) {

this.itemRepository = itemRepository;

}

@Override

public void run(String... args) throws Exception {

Stream.of(""Lining", "PUMA", "Bad Boy", "Air Jordan", "Nike", "Adidas", "Reebok")

.forEach(item -> itemRepository.save(new Item(item)));

itemRepository.findAll().forEach(System.out::println);

}

}

If you’re using an editor that doesn’t auto-import classes, here’s the list of imports needed at the top of ItemCatalogServiceApplication.java.

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import lombok.ToString;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import org.springframework.stereotype.Component;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import java.util.stream.Stream;

Add an application name in item-catalog-service/src/main/resources/application.properties file to display in the Eureka service, and set the port to 8088.

server.port=8088

spring.application.name=item-catalog-service

**Now, Create the Cloud Properties file**

Click on**File –> New –> Other –> File**and add the below code in this file and save it.

eureka.instance.hostname=${vcap.application.uris[0]:localhost}

eureka.instance.nonSecurePort=80

eureka.instance.metadataMap.instanceId=${vcap.application.instance\_id:${spring.application.name}:${spring.application.instance\_id:${server.port}}}

eureka.instance.leaseRenewalIntervalInSeconds = 5

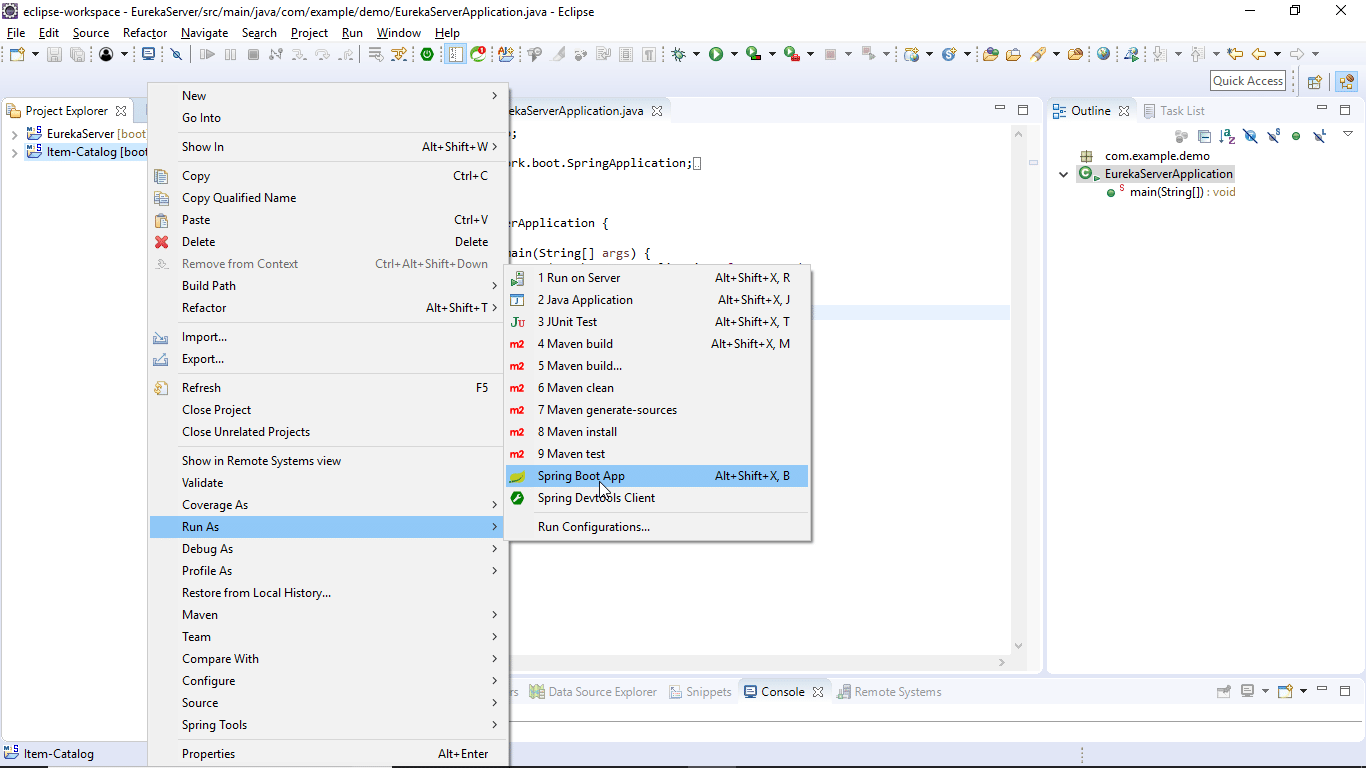
eureka.client.region = default

eureka.client.registryFetchIntervalSeconds = 5

eureka.client.serviceUrl.defaultZone=${vcap.services.pwa-eureka-service.credentials.uri}/eureka/

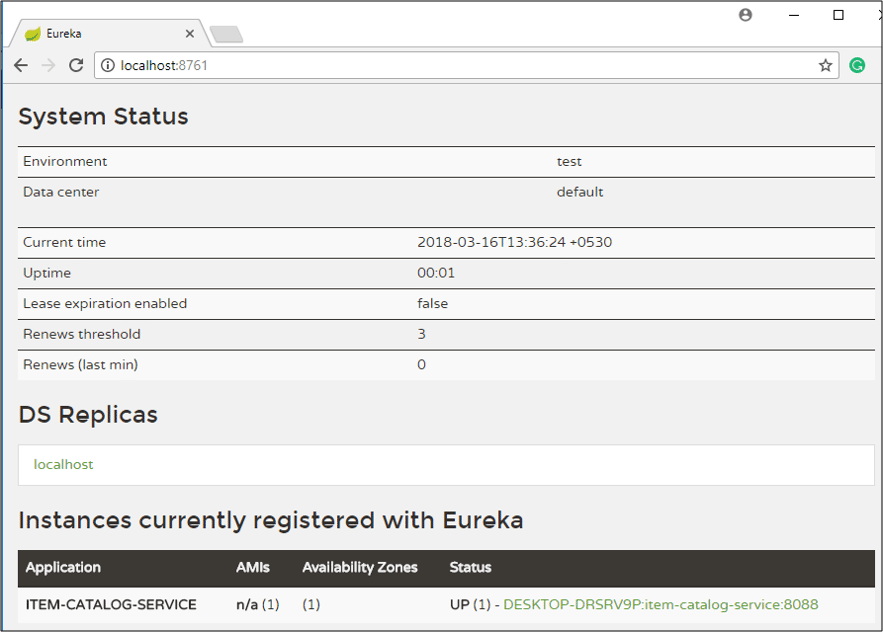
Now, to start the Application:

Right Click on Project –> **Run As** –> Click on “**Spring Boot App**“



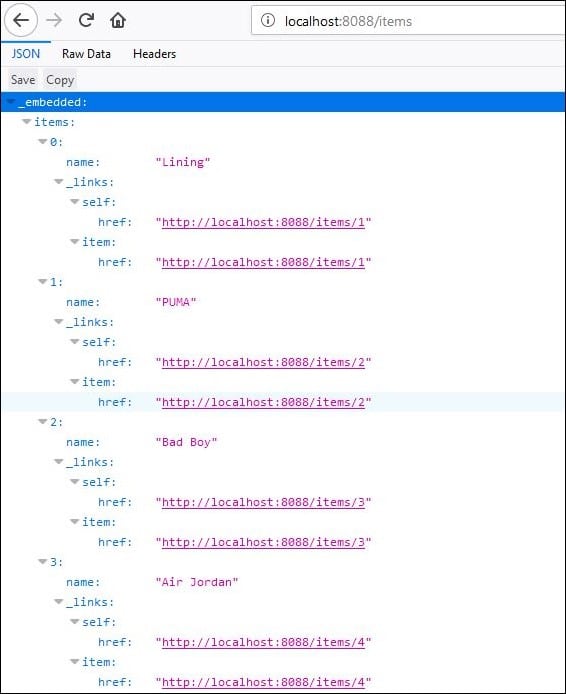
Note: In case of error try this step: Right Click on the **Project** –> **Run As** –> Click on “**Maven Build**“

Now open http://localhost:8761. Here you will see Item Catalog service will be running.



You will see the list of items from the catalog service.

Open http://localhost:8088/items

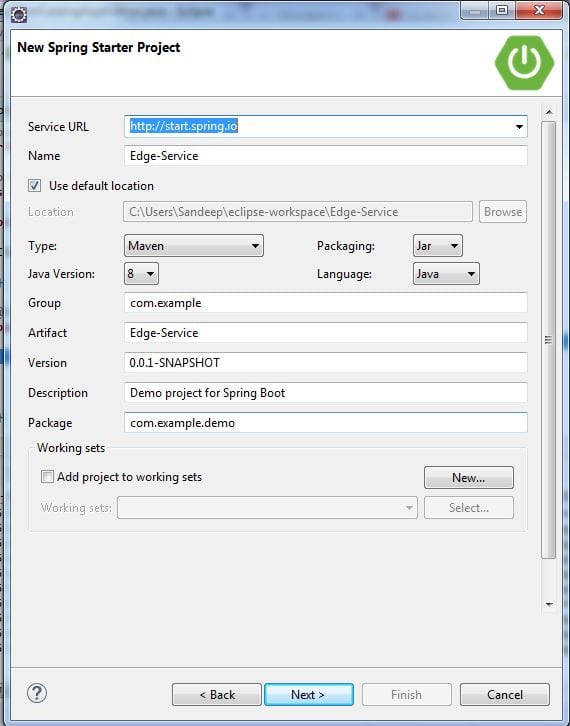


Now let us move forward and create the Edge Service.

**Spring Boot Microservices: Creating an Edge Service**

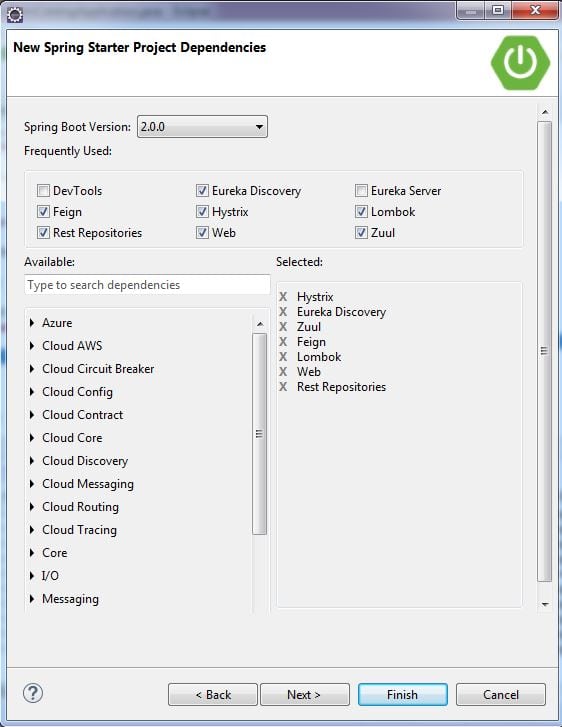
It is similar to the standalone Item service created in **Bootiful Development with Spring Boot and Angular**. However, it will have fallback capabilities which prevent the client from receiving an HTTP error when the service is not available.

Again create a new project. Use edge-service for the artifact name



* **Eureka Discovery**: for service registration
* **Feign**: a declarative web service client
* **Zuul**: provides intelligent routing
* **Rest Repositories**: to expose JPA repositories as REST endpoints
* **Web**: Spring MVC and embedded Tomcat
* **Hystrix**: a circuit breaker to stop cascading failure and enable resilience
* **Lombok**: to reduce boilerplate code

Click on **Finish.**



Since the item-catalog-service is running on port 8088, you’ll need to configure this application to run on a different port. Modify edge-service/src/main/resources/application.properties to set the port to 8089 and set an application name.

server.port=8089

spring.application.name=edge-service

**Now, Create the Cloud Properties file**

Click on**File –> New –> Other –> File**and add below code in this file and save it.

eureka.instance.hostname=${vcap.application.uris[0]:localhost}

eureka.instance.nonSecurePort=80

eureka.instance.metadataMap.instanceId=${vcap.application.instance\_id:${spring.application.name}:${spring.application.instance\_id:${server.port}}}

eureka.instance.leaseRenewalIntervalInSeconds = 5

eureka.client.region = default

eureka.client.registryFetchIntervalSeconds = 5

eureka.client.serviceUrl.defaultZone=${vcap.services.pwa-eureka-service.credentials.uri}/eureka/

To enable Feign, Hystrix, and registration with the Eureka server, add the appropriate annotations toEdgeServiceApplication.java:

package com.example.edgeservice;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

import lombok.Data;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.netflix.feign.EnableFeignClients;

import org.springframework.cloud.netflix.feign.FeignClient;

import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

import org.springframework.hateoas.Resources;

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

import java.util.ArrayList;

import java.util.Collection;

import java.util.stream.Collectors;

@EnableFeignClients

@EnableCircuitBreaker

@EnableDiscoveryClient

@EnableZuulProxy

@SpringBootApplication

public class EdgeServiceApplication {

public static void main(String[] args) {

SpringApplication.run(EdgeServiceApplication.class, args);

}

}

Create a Item DTO (Data Transfer Object) in this same file. Lombok’s [@Data](https://projectlombok.org/features/Data.html) will generate a toString() methods, getters, setters, and appropriate constructors.

@Data

class Item {

private String name;

}

Create a ItemClient interface that uses Feign to communicate to the Item-catalog-service.

public class EdgeServiceApplication {

public static void main(String[] args) {

SpringApplication.run(EdgeServiceApplication.class, args);

}

}

@Data

class Item {

private String name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@FeignClient("item-catalog-service")

interface ItemClient {

@GetMapping("/items")

Resources<Item> readItems();

}

Create a RestController below the ItemClient that will filter out less-than-top brands and shows a /top-brandsendpoint.

@RestController

class GoodItemApiAdapterRestController {

private final ItemClient itemClient;

public GoodItemApiAdapterRestController(ItemClient ItemClient) {

this.itemClient = itemClient;

}

@GetMapping("/top-brands")

public Collection<Item> goodItems() {

return itemClient.readItems()

.getContent()

.stream()

.filter(this::isGreat)

.collect(Collectors.toList());

}

private boolean isGreat(Item item) {

return !item.getName().equals("Nike") &&

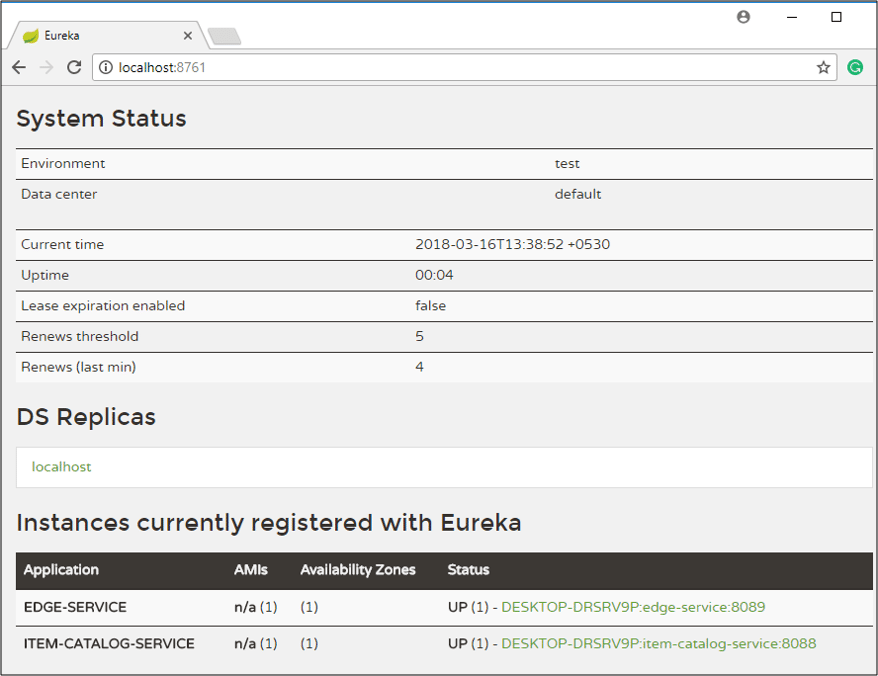
!item.getName().equals("Adidas") &&

!item.getName().equals("Reebok");

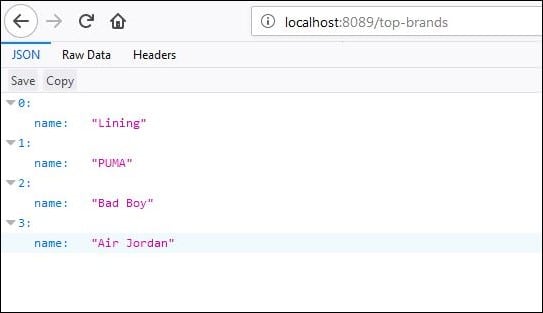
}

}

Start the edge-service application with Maven or your IDE and verify it registers successfully with the Eureka server.



Now invoke localhost:8089/top-brands, you will see the list of top brands from the catalog service.



**Note:**If you shut down the item-catalog-service application, you’ll get a 500 internal server error.

$ http localhost:8089/top-brands

HTTP/1.1 500

Connection: close

Content-Type: application/json;charset=UTF-8

Date: Fri, 16 Mar 2018 12:51:22 GMT

Transfer-Encoding: chunked

X-Application-Context: edge-service:8089

{

"error": "Internal Server Error",

"exception": "feign.RetryableException",

"message": "connect timed out executing GET http://item-catalog-service/items",

"path": "/top-brands",

"status": 500,

"timestamp": 1328088897672

}

To fix this, you can use Hystrix to create a fallback method and tell the goodItems() method to use it.

public Collection<Item> fallback() {

return new ArrayList<>();

}

@HystrixCommand(fallbackMethod = "fallback")

@GetMapping("/top-brands")

public Collection<Item> goodItems() {

return …

}

Restart the edge-service and you should see an empty list returned.

$ http localhost:8089/top-brands

HTTP/1.1 200

Content-Type: application/json;charset=UTF-8

Date: Fri, 16 Mar 2018 12:59:02 GMT

Transfer-Encoding: chunked

X-Application-Context: edge-service:8089

[]

Start the item-catalog-service again and this list should eventually return the full list of top brands names.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Spring Cloud - Getting Started Example [Updated: Jul 6, 2019, Created: Sep 8, 2018] | Top of Form   |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | |  |   Bottom of Form  [Next Page](https://www.logicbig.com/tutorials/spring-framework/spring-cloud/rest-template-load-balancer.html) |
| Following is a quick-start example of Spring Cloud. We are going to develop very simple microservices using Spring Cloud, Spring Boot and Eureka Server.  In microservice architecture, an application is composed of loosely coupled small services as opposed to a single monolithic application.  In microservice architecture a registry service is used to register the microservices so that they can be discovered.  https://www.logicbig.com/tutorials/spring-framework/spring-cloud/hello-world/images/microservices.png Example In this example we are going to use Eureka Server as the service registry. Eureka is developed by Netflix; it is open source. Spring has integrated Eureka into dedicated Spring Cloud modules to make it easier to use it.  We are going to develop two microservices:  First is 'hello-service' which will just return a hello message.  Second service 'hello-web-client-service' will handle the request coming from a client. On receiving a request it will call 'hello-service' and will return a web page in response.   There will be three separate servers; one for Eureka and two of microservices. Also there will be three separate maven projects. Eureka ServerMaven dependenciespom.xml <project .....>  <modelVersion>4.0.0</modelVersion>  <groupId>com.logicbig.example</groupId>  <artifactId>hello-eureka-server</artifactId>  <version>1.0-SNAPSHOT</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.4.RELEASE</version>  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-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> Configurationsrc/main/resources/application.yml server:  port: 7777  eureka:  instance:  hostname: localhost  client:  registerWithEureka: false  fetchRegistry: false  In above configuration, the properties eureka.client.\* are related to the service clients who want to register with Eureka.   The property eureka.client.register-with-eureka=false specifies that this server should not be registered to the service client itself.   The property eureka.client.fetch-registry=false specifies that the server should not fetch the registered information to itself. Main class @SpringBootApplication  @EnableEurekaServer  public class HelloEurekaServerMain {  public static void main(String[] args) {  SpringApplication.run(HelloEurekaServerMain.class, args);  }  }  Run above main class from your IDE. That will start the Eureka Server.  Now we can access the Eureka server at http://localhost:7777 as shown:  https://www.logicbig.com/tutorials/spring-framework/spring-cloud/hello-world/c/images/outputKey1.png hello-serviceMaven dependenciespom.xml <project .....>  <modelVersion>4.0.0</modelVersion>  <groupId>com.logicbig.example</groupId>  <artifactId>hello-service</artifactId>  <version>1.0-SNAPSHOT</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.4.RELEASE</version>  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-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> Domain object public class HelloObject {  private String message;  public String getMessage() {  return message;  }  public void setMessage(String message) {  this.message = message;  }  } A Rest Controller @RestController  public class HelloController {  private AtomicLong counter = new AtomicLong();  @GetMapping("/hello")  public HelloObject getHelloWordObject() {  HelloObject hello = new HelloObject();  hello.setMessage("Hi there! you are number " + counter.incrementAndGet());  return hello;  }  } src/main/resources/application.properties eureka.client.serviceUrl.defaultZone=http://localhost:7777/eureka/ src/main/resources/bootstrap.properties spring.application.name=hello-service Boot main class @SpringBootApplication  @EnableDiscoveryClient  public class HelloServiceMain{  public static void main(String[] args) {  SpringApplication.run(HelloServiceMain.class, args);  }  }  Run above main class from your IDE.  On refreshing Eureka page you should see HELLO-SERVICE instance listed in the registry:  https://www.logicbig.com/tutorials/spring-framework/spring-cloud/hello-world/c/images/outputKey2.png hello-web-client-servicepom.xml <project .....>  <modelVersion>4.0.0</modelVersion>  <groupId>com.logicbig.example</groupId>  <artifactId>hello-web-client-service</artifactId>  <version>1.0-SNAPSHOT</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.4.RELEASE</version>  </parent>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-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> A Spring MVC Controller @Controller  public class HelloWebClientController {  @Autowired  private DiscoveryClient discoveryClient;  @GetMapping("/")  public String handleRequest(Model model) {  //accessing hello-service  List<ServiceInstance> instances = discoveryClient.getInstances("hello-service");  if (instances != null && instances.size() > 0) {//todo: replace with a load balancing mechanism  ServiceInstance serviceInstance = instances.get(0);  String url = serviceInstance.getUri().toString();  url = url + "/hello";  RestTemplate restTemplate = new RestTemplate();  HelloObject helloObject = restTemplate.getForObject(url,  HelloObject.class);  model.addAttribute("msg", helloObject.getMessage());  model.addAttribute("time", LocalDateTime.now());  }  return "hello-page";  }  } src/main/resources/templates/hello-page.html <!DOCTYPE html>  <html xmlns="http://www.w3.org/1999/xhtml"  xmlns:th="http://www.thymeleaf.org">  <body>  <h2>Hello Page</h2>  <div th:text="${msg}"/>  <div>Time: <span th:text="${time}"/></div>  </body>  </html> src/main/resources/application.properties server.port=9080  eureka.client.serviceUrl.defaultZone=http://localhost:7777/eureka/ src/main/resources/bootstrap.properties spring.application.name=hello-service Boot main class @SpringBootApplication  @EnableDiscoveryClient  public class HelloWebClientServiceMain {  public static void main(String[] args) {  SpringApplication.run(HelloWebClientServiceMain.class, args);  }  }  Run above class from your IDE.  Refresh Eureka web page again:  https://www.logicbig.com/tutorials/spring-framework/spring-cloud/hello-world/c/images/outputKey3.png Final output Now make request to 'hello-web-client-service' by entering localhost:9080 in web-browser:  https://www.logicbig.com/tutorials/spring-framework/spring-cloud/hello-world/c/images/outputKey5.png |  |

***What is Spring Cloud?***

* It is building blocks for Cloud and Microservices
* It provides microservices infrastructure like provide use services such as Service Discovery, a Configuration server and Monitoring.
* It provides several other open source projects like Netflix OSS.
* It provides PaaS like Cloud Foundry, AWS and Heroku.
* It uses Spring Boot style starters

There are many use-cases supported by Spring Cloud like Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Client-side Load Balancing etc. But in this post we concentrate on following microservices support

* Service Discovery (How do services find each other?)
* Client-side Load Balancing (How do we decide which service instance to use?)

**Service Discovery**  
**Problem without discovery**

* How do services find each other?
* What happens if we run multiple instances for a service

microservices without discovery server

**Resolution with service discovery**

microservices with discovery server

**Implementing Service Discovery**

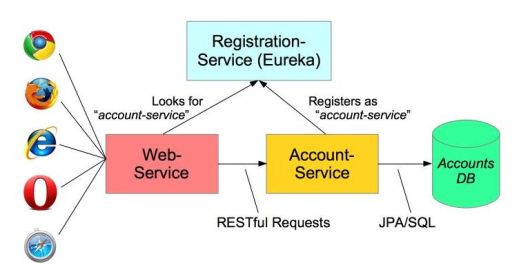
Spring Cloud support several ways to implement service discovery but for this, I am going to use Eureka created by Netflix. Spring Cloud provides several annotation to make it use easy and hiding lots of complexity.

**Client-side Load Balancing**

Each service typically deployed as multiple instances for fault tolerance and load sharing. But there is the problem how to decide which instance to use?

**Implementing Client-Side Load Balancing**

We will use Netflix Ribbon, it provides several algorithms for Client-Side Load Balancing. Spring provides smart **RestTemplate**for service discovery and load balancing by using ***@LoadBalanced***annotation with **RestTemplate**instance.



*@ImageSource-Spring.io*

**6. Developing Simple Microservices Example**

**To build a simple microservices system following steps required**

1. Creating Discovery Service (Creating Eureka Discovery Service)
2. Creating MicroService (the Producer)
   1. Register itself with Discovery Service with logical service.
3. Create Microservice Consumers find Service registered with Discovery Service
   1. Discovery client using a smart **RestTemplate**to find microservice.

**Maven Dependencies**

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.hsqldb</groupId>

<artifactId>hsqldb</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

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

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

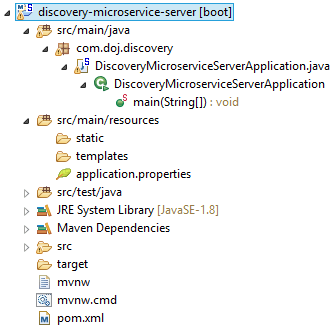
<scope>test</scope>

</dependency>

</dependencies>

**Step 1: Creating Discovery Service (Creating Eureka Discovery Service)**

* Eureka Server using Spring Cloud
* We need to implement our own registry service as below.



**application.yml**

# Configure this Discovery Server

eureka:

instance:

hostname: localhost

client: #Not a client

registerWithEureka: false

fetchRegistry: false

# HTTP (Tomcat) port

server:

port: 1111

**DiscoveryMicroserviceServerApplication.java**

package com.doj.discovery;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class DiscoveryMicroserviceServerApplication {

public static void main(String[] args) {

SpringApplication.run(DiscoveryMicroserviceServerApplication.class, args);

}

}

**pom.xml**

<!-- Eureka registration server -->

<dependency>

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

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

For Whole Source Code for the Discover Server Application, you could download from GitHub as below link.

[**discovery-microservice-server**](https://github.com/DOJ-SoftwareConsultant/discovery-microservice-server)

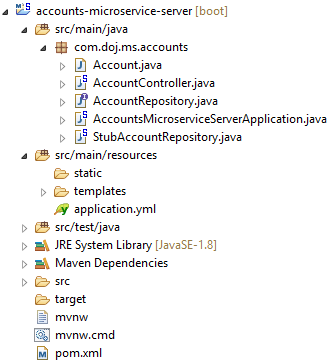
Run this Eureka Server application with right click and run as Spring Boot Application and open in browser **http://localhost:1111/**

discovery-service-home-page

**Step 2: Creating Account Producer MicroService**  
Microservice declares itself as an available service and register to Discovery Server created in **Step 1**.

* Using ***@EnableDiscoveryClient***
* Registers using its application name

Let’s see the service producer application structure as below.



**application.yml**

### Spring properties

# Service registers under this name

spring:

application:

name: accounts-microservice

# Discovery Server Access

eureka:

client:

serviceUrl:

defaultZone: http://localhost:1111/eureka/

# HTTP Server (Tomcat) Port

server:

port: 2222

# Disable Spring Boot's "Whitelabel" default error page, so we can use our own

error:

whitelabel:

enabled: false

**AccountsMicroserviceServerApplication.java**

package com.doj.ms.accounts;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class AccountsMicroserviceServerApplication {

public static void main(String[] args) {

SpringApplication.run(AccountsMicroserviceServerApplication.class, args);

}

}

**pom.xml**

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

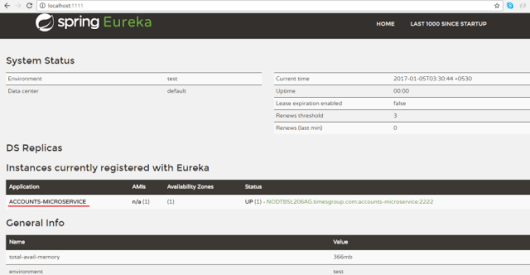
</dependency>

</dependencies>

Other required source files related to this application you could download from GitHub link as given below

[**accounts-microservice-server**](https://github.com/DOJ-SoftwareConsultant/accounts-microservice-server)

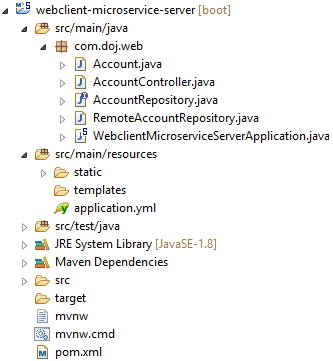
Now run this account service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server**at **http://localhost:1111/** in previous **Step 1**. Now one Service registered to the Eureka registered instances with Service Name “**ACCOUNT-MICROSERVICE**” as below



**Step 3: Consumer Service**

* Create Consumers to find the Producer Service registered with Discovery Service at Step 1.
* ***@EnableDiscoveryClient*** annotation also allows us to query Discovery server to find microservices.

Let’s see the consumer application structure as below.



**application.yml**

# Service registers under this name

# Control the InternalResourceViewResolver:

spring:

application:

name: accounts-web

mvc:

view:

prefix: /WEB-INF/views/

suffix: .jsp

# Discovery Server Access

eureka:

client:

serviceUrl:

defaultZone: http://localhost:1111/eureka/

# Disable Spring Boot's "Whitelabel" default error page, so we can use our own

error:

whitelabel:

enabled: false

**WebclientMicroserviceServerApplication.java**

package com.doj.web;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

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

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;

@SpringBootApplication

@EnableDiscoveryClient

public class WebclientMicroserviceServerApplication {

public static final String ACCOUNTS\_SERVICE\_URL = "http://ACCOUNTS-MICROSERVICE";

public static void main(String[] args) {

SpringApplication.run(WebclientMicroserviceServerApplication.class, args);

}

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

@Bean

public AccountRepository accountRepository(){

return new RemoteAccountRepository(ACCOUNTS\_SERVICE\_URL);

}

}

**pom.xml**

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-ribbon</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-actuator</artifactId>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<!-- These dependencies enable JSP usage -->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

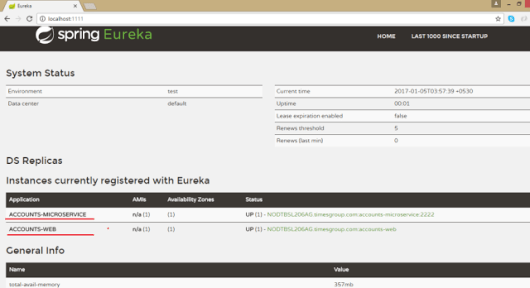
</dependency>

</dependencies>

Other required source files related to this application you could download from GitHub link as given below

[**web client-microservice-server**](https://github.com/DOJ-SoftwareConsultant/webclient-microservice-server)

Now run this consumer service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server**at **http://localhost:1111/** in previous **Step 1**. Now one more Service registered to the Eureka registered instances with Service Name “**ACCOUNTS-WEB**” as below



Lets our consumer consume the service of producer registered at discovery server.

package com.doj.web;

import java.util.Arrays;

import java.util.List;

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

import org.springframework.web.client.RestTemplate;

/\*\*

\* @author Dinesh.Rajput

\*

\*/

public class RemoteAccountRepository implements AccountRepository {

@Autowired

protected RestTemplate restTemplate;

protected String serviceUrl;

public RemoteAccountRepository(String serviceUrl) {

this.serviceUrl = serviceUrl.startsWith("http") ? serviceUrl

: "http://" + serviceUrl;

}

@Override

public List<Account> getAllAccounts() {

Account[] accounts = restTemplate.getForObject(serviceUrl+"/accounts", Account[].class);

return Arrays.asList(accounts);

}

@Override

public Account getAccount(String number) {

return restTemplate.getForObject(serviceUrl + "/accounts/{id}",

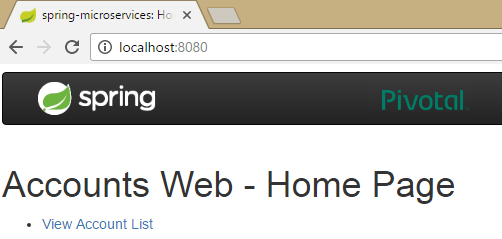
Account.class, number);

}

}

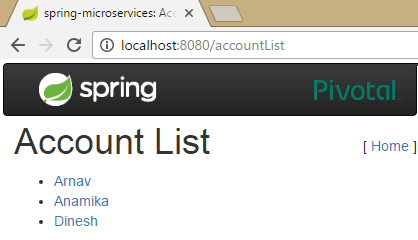
Let’s open web application which is a consumer of the account microservice registered at Eureka Discovery Server.

**http://localhost:8080/** as below



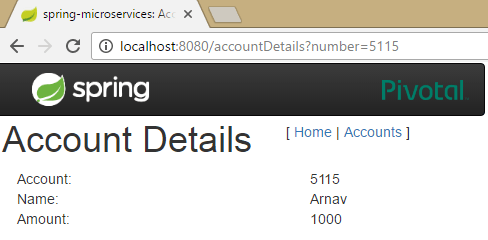
Now click on **View Account List** then fetch all accounts from account microservice.

**http://localhost:8080/accountList**



Now click on any account from the list of accounts to fetch the details of the account for account number from account microservice.

**http://localhost:8080/accountDetails?number=5115**



**Load Balanced *RestTemplate***  
**Create using** ***@LoadBalanced***– Spring enhances it to service lookup & load balancing

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

**Must inject using the same qualifier-**

* If there are multiple ***RestTemplate***you get the right one.
* It can be used to access multiple microservices

@Autowired

@LoadBalanced

protected RestTemplate restTemplate;

**Load Balancing with Ribbon**  
Our smart RestTemplate automatically integrates two Netflix utilities

* ***Eureka***Service Discovery
* ***Ribbon***Client Side Load Balancer

***Eureka***returns the URL of all available instances  
***Ribbon***determine the best available service too use

Just inject the load balanced ***RestTemplate***automatic lookup by ***logical service-name***

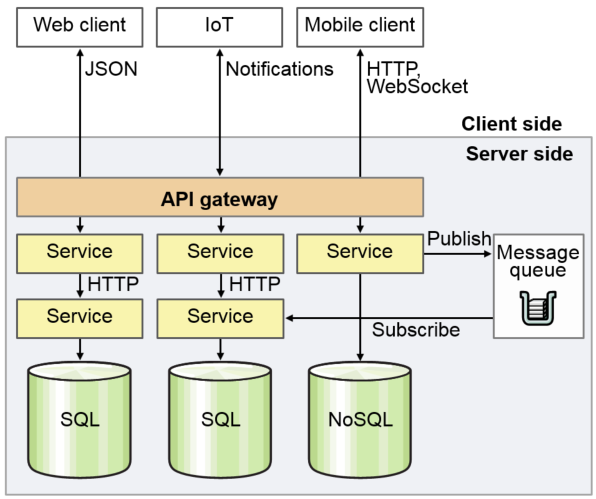
**7. Summary**  
After completion of this article you should have learned:

* What is the MicroServices Architecture
* Advantages and Challenges of MicroServices
* And some information about Spring Cloud such as Eureka Discover Server by Netflix and Ribbon.

Microservice, a component in this architecture:

* Each is a miniature application
* Each is focused on one task, a business capability (The Single Responsibility principle: ***Each microservice implements only one business responsibility from the bounded domain context. From a software point of view, the system needs to be decomposed into multiple components where each component becomes a microservice. Microservices have to be lightweight, in order to facilitate smaller memory footprints and faster startup times.***)
* Each can be deployed and updated independently
* They are loosely coupled
* Each has a well-defined interface: REST APIs

https://miro.medium.com/max/60/1*A52E6nH0o0JR91OvBm_fYw.png?q=20



**FIGURE 4: Microservices Architecture.**

# Zuul API Gateway Tutorial

Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, and more.

 It is mainly responsible to route the service to the appropriate microservice.

In this tutorial, you will learn how to create and run your own Zuul API Gateway and how to register with it more than one Spring Boot Microservice.

To be able to follow this tutorial you need to know how to create simple RESTful Web Services with Spring Boot. If you need to review that, please have a look at this tutorial that teaches [how to create a very simple RESTful Web Services with Spring Boot](http://appsdeveloperblog.com/eureka-discovery-server-tutorial/).

Here is what we will do in this tutorial:

1. Start up Eureka Discovery Server,
2. Start up a simple Users Microservice. This can be any other Microservice you have,
3. Create, configure and start up a new Zuul API Gateway,
4. Learn how to send HTTP Requests to a Microservice via Zuul API Gateway.

**For a step by step series of**[**video lessons**](https://www.udemy.com/spring-boot-microservices-and-spring-cloud/?couponCode=MICROSERVICES)**, please check this page:**[**Spring Boot Microservices and Spring Cloud**](https://www.udemy.com/spring-boot-microservices-and-spring-cloud/?couponCode=MICROSERVICES)**.**

## Start-Up Eureka Discovery Server

Since this tutorial is about Zuul API Gateway I will not go in details on how to create and configure your Eureka Discovery Server and will simply show you my source code. But if you are interested to learn in more details on how to create a Eureka Discovery Service, please follow this short tutorial I have created earlier: [Create and run a very simple Eureka Discovery Server](http://appsdeveloperblog.com/eureka-discovery-server-tutorial/).

Below are the details of my Eureka Discovery Server which I have created for this tutorial.

**Application Java Class**

package com.appsdeveloperblog.photoapp.discovery.PhotoAppDiscoveryService;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class PhotoAppDiscoveryServiceApplication {

public static void main(String[] args) {

SpringApplication.run(PhotoAppDiscoveryServiceApplication.class, args);

}

}

**Eureka Server application.properties File**

**server.port=8010**

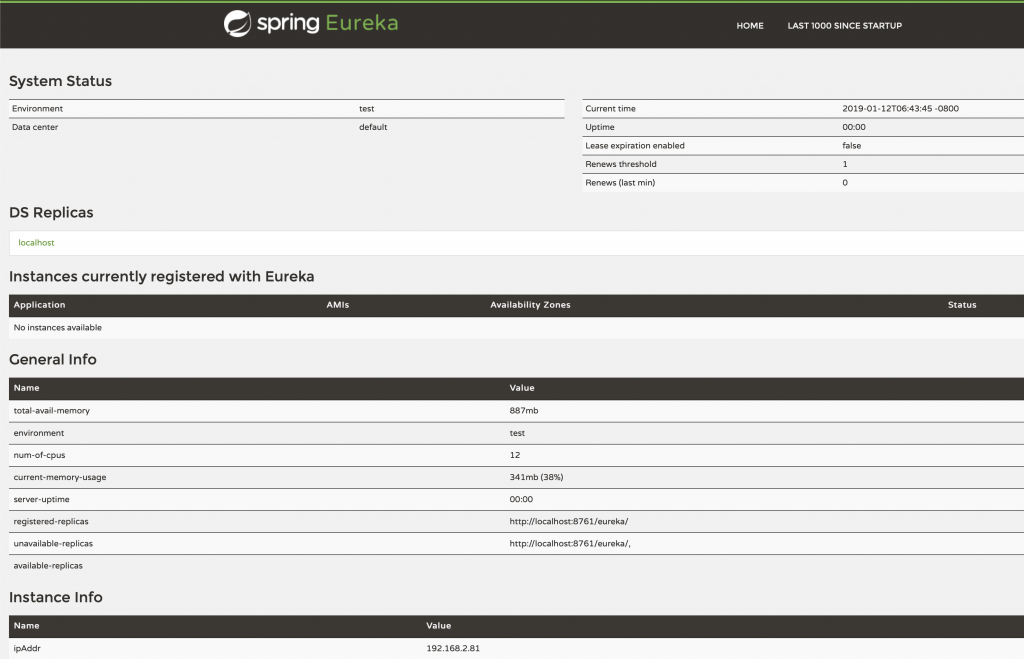
**spring.application.name=PhotoAppApi-eureka-server**

**eureka.client.registerWithEureka=false**

**eureka.client.fetchRegistry=false**

Start up Eureka Discovery Service and open its URL in the browser window. In the application.properties file above I have configured port 8010 to be used for my Eureka server. So the URL I need to open is:

http://localhost:8010



## Start Up Spring Boot Microservice

The Netflix Zuul API Gateway we are going to create in this tutorial will route HTTP Requests sent to a Microservice registered with Eureka. You might already have one or more Spring Boot Microservices created but if you do not have, please follow this tutorial on [how to make your Microservice registered with Eureka Discovery Server](http://appsdeveloperblog.com/register-web-service-with-eureka-server/).

Below are the details of the Spring Boot Microservice I have created for this tutorial.

**Application Java class**

package com.appsdeveloperblog.photoapp.api.users;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class PhotoAppApiApplication {

public static void main(String[] args) {

SpringApplication.run(PhotoAppApiApplication.class, args);

}

}

**Users Microservice application.properties File**

spring.devtools.restart.enabled = false

eureka.client.serviceUrl.defaultZone = http://localhost:8010/eureka

server.port=0

spring.application.name=users-ws

**Users Microservice Rest Controller Class**

package com.appsdeveloperblog.photoapp.api.users.io.controllers;

import org.springframework.web.bind.annotation.GetMapping;

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

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

@RestController

@RequestMapping("/users")

public class UsersController {

@GetMapping("/status/check")

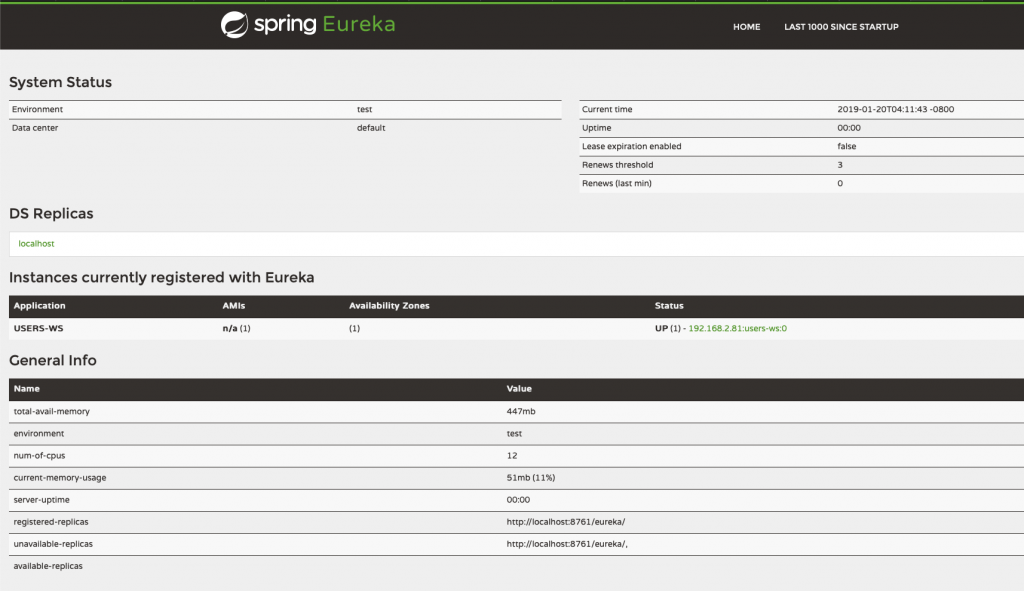
public String status() {

return "Working";

}

}

Start up your Spring Boot Microservice and if you have configured it to register with Eureka Discovery Server as I have done it in the example of application.properties file above, then your Microservice should register with Eureka successfully. Refresh Eureka page and it should now have your Spring Boot Microservice listed.



## Starting up a Second Microservice

Let’s assume I have two Microservices I want to register with my Eureka Discovery service. All I need to do is to create a new Microservice and follow the steps above to update its application.properties file to point to correct Eureka URL and to have a unique application name. Here is an example of application.properties file of my second Microservice which is called albums-ws.

**Albums Web Service application.properties file**

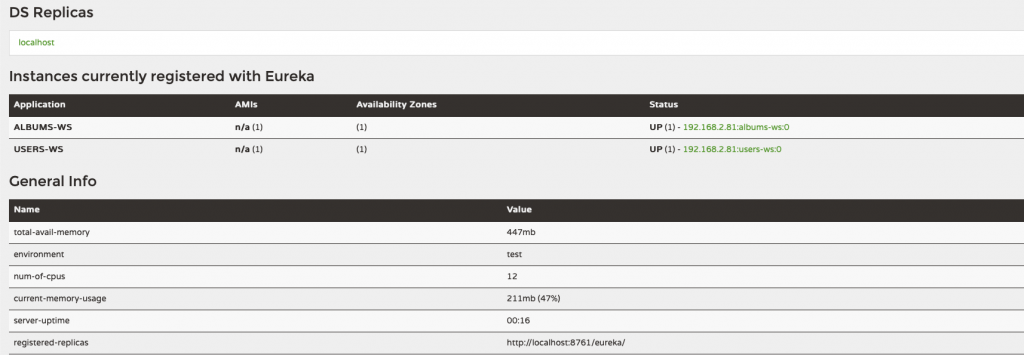
**spring.devtools.restart.enabled = true**

**eureka.client.serviceUrl.defaultZone = http://localhost:8010/eureka**

**server.port=0**

**spring.application.name=albums-ws**

As you can see the application.properties file of my second Microservice looks very much the same except it has a different application name: albums-ws. After starting up a second Microservice, if I refresh the Eureka Registry page I should see two Microservices registered.



## Zuul API Gateway

It is time to create a Zuul API Gateway. To create a new API Gateway with Netflix Zuul we will first need to create a very simple Spring Boot Web Service. I almost always create Spring Boot Web Services using the [Spring Initializr](https://start.spring.io/) project page. Go ahead and create a new Spring Boot Web Service project or [follow this step by step video tutorial](http://appsdeveloperblog.com/create-web-service-project-with-spring-boot/) to create a new Spring Boot Web Service.

## Zuul API Gateway Dependencies

Once you have your very simple Spring Boot Web project created, open its POM.xml file and add the following details:

**<dependenyManagement>**

<dependencies>

<dependency>

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

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

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**Gateway Starter and Eureka Client dependencies**

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

**The <repositories> element**

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://[repo.spring.io/milestone](http://repo.spring.io/milestone)</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

**The <properties> element**

<properties>

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

<spring-cloud.version>Greenwich.RC2</spring-cloud.version>

</properties>

**The complete 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>

<parent>

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

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

<version>2.1.2.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.appsdeveloperblog.photoapp.api.gateway</groupId>

<artifactId>ZuulApiGateway</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>ZuulApiGateway</name>

<description>Demo project for Spring Boot</description>

<properties>

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

<spring-cloud.version>Greenwich.RC2</spring-cloud.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://[repo.spring.io/milestone](http://repo.spring.io/milestone)</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

## Zuul API Gateway Application.properties File

server.port=8011

spring.application.name=zuul

eureka.client.serviceUrl.defaultZone = http://localhost:8010/eureka

eureka.client.fetchRegistry=true

eureka.client.registerWithEureka=true

## Zuul API Gateway Main Application Java File

Next, we need to update the main Application Java file of our Zuul Spring Boot app. Locate in your Zuul project the Java which contains public static void main(String[] args) and add above the class the following two annotations:

* **@EnableEurekaClient** – Zuul API Gateway also needs to register with Eureka Discovery Service
* **@EnableZuulProxy**– this annotation will make our Spring Boot App behave as a Zuul API Gateway.

package com.appsdeveloperblog.photoapp.api.gateway;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

import org.springframework.context.annotation.Bean;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@SpringBootApplication

@EnableEurekaClient

@EnableZuulProxy

public class ApiGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(ApiGatewayApplication.class, args);

}

}

And this is it. Although this is a very basic setup of Zuul API Gateway it should work well and you should be able to communicate with your RESTful Web Services via the Gateway.  You can also start up more instance of same RESTful Web Services and Zuul will also load balance HTTP Requests between them. More on how to start up more of the same Microservices and see how load balancer works I will share with you in my following tutorial.

## Sending HTTP Request

Now when you have your RESTful Web Services registered with Eureka Discovery Server and you have your Zuul API Gateway started, you should be able to send HTTP requests to any of the two web services registered and receive expected HTTP response.

**The URL to Users Microservice**

Let’s have a look at how the request URL to our Microservice should look like. Below are the URLs to the two Microservices we have started up in this tutorial and have made them available via the Zuul API Gateway. So the HTTP requests you send to your RESTful Web Services should be now sent via the gateway.

* [**http://localhost:8011/users-ws/users/status/check**](http://localhost:8011/users-ws/users/status/check)
* [**http://localhost:8011/albums-ws/users/status/check**](http://localhost:8011/albums-ws/users/status/check)

where:

* [**http://localhost**](http://localhost/)– is the protocol and the domain name of your Zuul API Gateway,
* **8011** – the port number you have specified in the application.properties file of your Zuul API Gateway,
* **/users-ws**and /**albums-ws**– are the application names you have assigned to your Microservices in the application.properties file. Have a look at the property called “spring.application.name” in the above-mentioned examples of application.properties file of each of the RESTful Web Services,
* **/users**and the **/albums**  – is the Request Mapping you have in you @RestController class. If you look at the example of Rest Controller call of Users Microservice above, it has a @RequestMapping(“/users”) annotation,
* **/status/check** – is the method level Request Mapping in the Rest Controller class. For example, @GetMapping(“/status/check”).

# Tools

To provide the solution we would use Spring Cloud which is primarily based on netflix stack. Let us see the component from spring cloud and their usability

**Eureka**— Provides a common platform for all the (micro)services to be discovered.

**Config**— it is a nice to have tool to bring all the properties from different services together and be a single point of contact

**Feign**— provides a way to wrap rest call from one service to another service in a nice spring-ish manner.

**Ribbon**— Behind the scene load balancer from one service to another service.

**Hystrix**— Provides the ability to apply fallback and provide the metrics.

**Zuul**— Zuul is used for routing the request to different services, additionally allows us to add filters.

**Turbine**— Has the ability to get the metrics from different services and provide a stream combining them.

Spring Boot Starters:

Starter is like a small spring project for each module such as web MVC, JDBC,ORM and so on.For your spring application ,you just add the starters of the respective module in the classpath and Spring Boot will ensure that the necessary libraries are added to the build by using Maven.

Spring Boot Starters reduces a build’s dependencies and Spring Boot auto configuration reduces the Spring configuration.

If you want to exclude auto-configuration for the some of the modules ,then you use the exclude property of @SpringBootApplication.Lets look at the following

@SpringBootApplication(exclude={DataSourceAutoConfiguration.class,HibernateJpaAutoConfiguration.class})

@RestController annotation: It indicates that this is the controller class and its result writes in to the response body and does not want to render view.

@GetMapping annotation:It is a shortform of @RequestMapping(method=RequestMethod.GET)

@SpringBootApplication : This annotation tells Spring Boot,when launched,to scan recursively for Spring components inside this package and register them.It also tells Spring Boot to enable auto-configuration.

SpringApplication.run():The SpringApplication class is responsible for creating the ApplicationContext from the classpath, scan the configuration classes and launch the application.

## Steps Executed under this method –

So in short when the main method runs following **steps** occur:

1. **Application Context** is started.
2. Using application context **autodiscovery** occurs: @ComponentScan
3. All default configurations are set up ie based on dependencies mentioned spring boot automatically sets up defaults. It makes use of intelligence that if we have included spring-web starter then dispatcher servlet is auto-configured. (**@EnableAutoConfiguration**)
4. An embedded servlet container is started. ( No need to set up a separate web server ) . Note embedded servlet container is launched only if the web is mentioned in a dependency

When SpringApplication.run() command is invoked, the Application Context is created by calling the method below:

public ConfigurableApplicationContext run(String... args) {

// Create, load, refresh, and run the ApplicationContext

context = createApplicationContext();

return context ; // handle to the context object for the developer

}

**Question:** What exactly is the type of this context?

**Answer:**

The createApplicationContext method checks if it is a web or standalone application based on the type it creates for the context. I was creating a REST-based controller for which a context of type **AnnotationConfigEmbeddedWebApplicationContext** was initialized. In the case of a standalone application, **AnnotationConfigApplicationContext** will be initialized.

**Question:**How are the beans created once the context is initialized?

**Answer:**

When the constructor of the context is invoked, it will register the annotated class beans with the context. That's why no XML configurations are required. All your @Repository, @Component, @Service, and Controller beans will be registered and the context is returned. The following lines of code are executed for context initialization and bean creation for a web application.

public AnnotationConfigEmbeddedWebApplicationContext(Class<?>... annotatedClasses) {

this();

register(annotatedClasses);

refresh(); // Refreshing org.springframework.boot.context.embedded. This log appears in the console

}

**Question:** Which servlet acts as a front controller?

**Answer:**

No prizes for guessing that: DispatcherServlet.

The AnnotationConfigEmbeddedWebApplicationContext class extends the EmbeddedWebApplicationContext, which registers the dispatcher servlet.

public static final String DISPATCHER\_SERVLET\_NAME = ServletContextInitializerBeans.DISPATCHER\_SERVLET\_NAME;

**Question:** What about the embedded Tomcat?

**Answer:**

Normally, starting an embedded Tomcat is as easy as instantiating the Tomcat class.

Include the following dependencies in Maven         P

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-core</artifactId>

<version>${tomcat.version}</version>

</dependency>

And write a class to bootstrap Tomcat:

Tomcat tomcat = new Tomcat();

tomcat.setPort(8080);

// Create context object and set it

tomcat.addContext ("/mycontext);

tomcat.start();

tomcat.getServer().await();

So with regards to Spring Boot. the EmbeddedWebApplicationContext creates an instance of org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainer and adds the context.

TomcatEmbeddedServletContainer class has Tomcat as an instance variable.

Check the selfInitialize() method and prepareEmbeddedWebApplicationContext of the EmbeddedWebApplicationContext class:

prepareEmbeddedWebApplicationContext() {

servletContext.log("Initializing Spring embedded WebApplicationContext"); // these logs are printed in your STS console.

logger.info("Root WebApplicationContext: initialization completed in " );  // these logs are printed in your STS console.

}

# How to rename application.properties file in Spring Boot application?

By default, [**Spring Boot**](https://projects.spring.io/spring-boot/) will look for your externalized properties in a file named application.properties located in resources folders :

Once we create a Spring Boot project,

* Let’s delete the *application.properties* file from the resources folder of our project.
* Then, let’s create a *customapp.properties*file in the resources folder of our project. We’ll provide all our application configurations in this properties file.

*spring.config.name* property holds the name of our properties file in the Spring Boot application.

## System Property:

@SpringBootApplication

public class MasteringSpringBootApplication {

public static void main(String[] args) {

**System.setProperty("spring.config.name", "myapp");**

SpringApplication.run(MasteringSpringBootApplication.class, args);

}

}

## SpringBootApplication scanBasePackages

By default SpringApplication scans the configuration class package and all it’s sub-pacakges. So if our SpringBootRestApplication class is in com.journaldev.spring.main package, then it won’t scan com.journaldev.spring.controller package. We can fix this situation using SpringBootApplication scanBasePackages property.

@SpringBootApplication(scanBasePackages="com.journaldev.spring")

public class SpringBootRestApplication {

}

## Spring Boot Auto-Configured Beans

Since Spring Boot provides auto-configuration, there are a lot of beans getting configured by it. We can get a list of these beans using below code snippet.

ApplicationContext ctx = SpringApplication.run(SpringBootRestApplication.class, args);

String[] beans = ctx.getBeanDefinitionNames();

for(String s : beans) System.out.println(s);

Below is the list of beans configured by our spring boot application.

## Spring Boot Devtools Tutorial

## Enabling Dev Tools Module

To enable dev tools in spring boot application is very easy. Just add the spring-boot-devtoolsdependency in your build file.

**Maven**

|  |
| --- |
| pom.xml |
| <dependencies>      <dependency>          <groupId>org.springframework.boot</groupId>          <artifactId>spring-boot-devtools</artifactId>          <optional>true</optional>      </dependency>  </dependencies> |

## Automatic UI refresh

The spring-boot-devtools module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. Precondition is that your browser should have supported extention for it.

By default, live reload is enabled. If you wish to disable this feature for some reason, then set spring.devtools.livereload.enabled property to false.

|  |
| --- |
| application.properties |
| spring.devtools.livereload.enabled  = false #Set false to disable live reload |

## Excluding Resources from auto-reload

By default, Auto-reload works on these paths:

1. /META-INF/maven
2. /META-INF/resources
3. /resources
4. /static
5. /public
6. /templates

If you want to disable auto-reload in browser for files in few of these paths, then use spring.devtools.restart.exclude property. e.g.

|  |
| --- |
| spring.devtools.restart.exclude=static/\*\*,public/\*\* |

## Watching/Excluding Additional Paths

There may be few files not in classpath, but you still may want to watch those addtional files/paths to reload the application. To do so, use the spring.devtools.restart.additional-paths property.

|  |
| --- |
| spring.devtools.restart.additional-paths=script/\*\* |

Similarily, If you want to keep those defaults and **add additional exclusions**, use the spring.devtools.restart.additional-exclude property instead.

|  |
| --- |
| spring.devtools.restart.additional-exclude=styles/\*\* |

## Automatic server restart

Auto-restart means reloading the java classes and consiguration at server side. After the server side changes are re-deployed dynamically, server restart happen and load the modified code and configutation.

## Enable/disable logging of auto-configuration changes

By default, each time your application restarts, a report showing the condition evaluation delta is logged. The report shows the changes to your application’s auto-configuration as you make changes such as adding or removing beans and setting configuration properties.

To disable the logging of the report, set the following property:

|  |
| --- |
| spring.devtools.restart.log-condition-evaluation-delta = false |

## Disabling Restart

To disable the restart of server on non-static code changes, use the property spring.devtools.restart.enabled.

|  |
| --- |
| spring.devtools.restart.enabled = false |

## **Logging in** spring **boot**

**Logging in spring boot** is very flexible and easy to configure. Spring boot active enabled logging is determined by [spring-boot-starter-logging](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-logging/pom.xml) artifact

If we do not provide any logging specific configuration, we will still see logs printed in “console”. These are because of **default logging support** provided in spring boot which uses **Logback**.

Spring boot’s internal logging is written with Apache Commons Logging so it is one and only mandatory dependency. Till, boot 1.x – we had to import it manually. Since boot 2.x, it is downloaded transitively. To be more precise, spring-boot-starter-web depends on spring-boot-starter-logging, which pulls in spring-jcl for us.

## Add log statements

To add log statements in application code, use org.slf4j.Logger and org.slf4j.LoggerFactory from SLF4J. It provides lots of useful methods for logging anf also decouple the logging implementation from application.

|  |
| --- |
| Application.java |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;    @SpringBootApplication  public class Application  {      private static final Logger LOGGER=LoggerFactory.getLogger(Application.class);        public static void main(String[] args) {          SpringApplication.run(Application.class, args);            LOGGER.info("Simple log statement with inputs {}, {} and {}", 1,2,3);      }  } |
| Console |
| 2019-07-28 12:16:57.129  INFO 3416 --- [main]  com.howtodoinjava.demo.Application: Simple log statement with inputs 1, 2 and 3 |

## 1.2. Logging Level

Logback supports ERROR, WARN, INFO, DEBUG, or TRACE as logging level. By default, logging level is set to **INFO**. It means that code>DEBUG and TRACE messages are not visible.

To enable debug or trace logging, we can set the logging level in application.properties file. Also, we can pass the –debug or –trace arguments on the command line while starting the application.

|  |
| --- |
| Configuration |
| # In properties file  debug=true    # In Console  $ java -jar target/my-app-0.0.1-SNAPSHOT.jar --trace |

We can apply logging levels to specific packages as well. It can be done either in console or application.properties file.

|  |
| --- |
| Configuration |
| # In Console  -Dlogging.level.org.springframework=ERROR  -Dlogging.level.com.howtodoinjava=TRACE    # In properties file  logging.level.org.springframework=ERROR  logging.level.com.howtodoinjava=TRACE |

If the log level for a package is defined multiple times with different log levels, the lowest level will be used. TRACE is lowest and ERROR is highest.

## 1.3. Log format

The default log statement formatting is mentioned in [defaults.xml](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot/src/main/resources/org/springframework/boot/logging/logback/defaults.xml) file.

|  |
| --- |
| defaults.xml |
| <conversionRule conversionWord="clr"  converterClass="org.springframework.boot.logging.logback.ColorConverter" />    <conversionRule conversionWord="wex"  converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />    <conversionRule conversionWord="wEx"  converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter" />    <property name="CONSOLE\_LOG\_PATTERN" value="${CONSOLE\_LOG\_PATTERN:-%clr(%d{${LOG\_DATEFORMAT\_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}})  {faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39})  {cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}"/>    <property name="FILE\_LOG\_PATTERN" value="${FILE\_LOG\_PATTERN:-%d{${LOG\_DATEFORMAT\_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}}  ${LOG\_LEVEL\_PATTERN:-%5p} ${PID:- } --- [%t] %-40.40logger{39} : %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}"/> |

It outputs following information.

* **Date and Time**: Millisecond precision and easily sortable.
* **Log Level**: ERROR, WARN, INFO, DEBUG, or TRACE.
* Process ID.
* A --- separator to distinguish the start of actual log messages.
* **Thread name**: Enclosed in square brackets (may be truncated for console output).
* **Logger name**: This is usually the source class name (often abbreviated).
* The log message.

To **customize the log format**, use logging.pattern.console and logging.pattern.file properties.

|  |
| --- |
| application.properties |
| # Logging pattern for the console  logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n    # Logging pattern for file  logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg% |

## 1.4. Logging to file

By default spring boot logs to console only. If we want to enable file logging, we can easily do it using simple property logging.file or logging.path.

When using logging.path, it will create a file named spring.log in mentioned package.

|  |
| --- |
| application.properties |
| # Output to a temp\_folder/file  logging.file=c:/temp/application.log    #logging.path=/my-folder/    # Logging pattern for file  logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg% |

## 2. Logback Logging

The default logging is good enough for most usecases. But sometimes in enterprise applications, we need more fine control over logging with other complex requirements. In that case, having a dedicated logging configuration is suitable.

Spring boot by default uses logback, so to customize it’s behavior, all we need to add only **logback.xml in classpath** and define customization over the file.

|  |
| --- |
| logback.xml |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration>        <property name="LOG\_LOCATION" value="c:/temp" />        <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">          <encoder>              <pattern>%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n</pattern>          </encoder>      </appender>        <appender name="FILE" class="ch.qos.logback.core.FileAppender">          <File>{LOG\_LOCATION}/mylog.log</File>          <encoder>               <pattern>%d{yyyy-MM-dd HH:mm:ss} - %logger{36} - %msg%n</pattern>          </encoder>          <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">              <fileNamePattern>${LOG\_LOCATION}/archived/mylog-%d{yyyy-MM-dd}.%i.log              </fileNamePattern>              <timeBasedFileNamingAndTriggeringPolicy                  class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">                  <maxFileSize>10MB</maxFileSize>              </timeBasedFileNamingAndTriggeringPolicy>          </rollingPolicy>      </appender>        <root level="INFO">          <appender-ref ref="CONSOLE"/>          <appender-ref ref="FILE"/>      </root>        <!-- Application logs at trace level -->      <logger name="com.howtodoinjava" level="trace" additivity="false">          <appender-ref ref="RollingFile" />          <appender-ref ref="Console" />      </logger>    </configuration> |

## 3. Log4j2 Logging

## Step 1: Exclude logback and include log4j2

As mentioned earlier, spring boot uses logback as default. So if we have to use any other logging framework e.g. log4j2, we must **exclude logback** from classpath of the application. Also, add **spring-boot-starter-log4j2** to classpath.

|  |
| --- |
| pom.xml |
| <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-web</artifactId>      <exclusions>          <exclusion>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-logging</artifactId>          </exclusion>      </exclusions>  </dependency>    <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-log4j2</artifactId>  </dependency> |

## Step 2: Add log4j2 configuration file

Now, add log4j2 specific configuration file in classpath (typically in **resources** folder). It can be named as any of the following:

* log4j2-spring.xml
* log4j2.xml

If we have logging configuration in any other file (e.g. log4j2.properties, applogs.xml etc), we can use logging.file property to specify it’s path application.properties file.

|  |
| --- |
| log4j2.xml |
| <?xml version="1.0" encoding="UTF-8"?>  <Configuration status="WARN" monitorInterval="30">      <Properties>          <Property name="LOG\_PATTERN">%d{yyyy-MM-dd'T'HH:mm:ss.SSSZ} %p %m%n</Property>          <Property name="APP\_LOG\_ROOT">c:/temp</Property>      </Properties>      <Appenders>          <Console name="console" target="SYSTEM\_OUT">              <PatternLayout pattern="${LOG\_PATTERN}" />          </Console>            <RollingFile name="file"              fileName="${APP\_LOG\_ROOT}/SpringBoot2App/application.log"              filePattern="${APP\_LOG\_ROOT}/SpringBoot2App/application-%d{yyyy-MM-dd}-%i.log">              <PatternLayout pattern="${LOG\_PATTERN}" />              <Policies>                  <SizeBasedTriggeringPolicy size="19500KB" />              </Policies>              <DefaultRolloverStrategy max="1" />          </RollingFile>        </Appenders>      <Loggers>          <Root level="info">              <AppenderRef ref="console" />              <AppenderRef ref="file" />          </Root>      </Loggers>  </Configuration> |

## Step 3: With or without Slf4j

By default, if you are using SLF4J logger classes i.e. org.slf4j.Logger and org.slf4j.LoggerFactory, nothing needs to be changed in application code and all log statement will continue printing in target appenders.

If you are targeting to use log4j2 specific classes only, use org.apache.logging.log4j.Logger and org.apache.logging.log4j.LogManager.

I will recommend to use SLF4J logger classes.

|  |
| --- |
| SLF4J logger classes |
| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;    @SpringBootApplication  public class Application  {      private static final Logger LOGGER = LoggerFactory.getLogger(Application.class);        public static void main(String[] args) {          SpringApplication.run(Application.class, args);            LOGGER.info("Simple log statement with inputs {}, {} and {}", 1,2,3);      }  } |
| LOG4J2 logger classes |
| import org.apache.logging.log4j.LogManager;  import org.apache.logging.log4j.Logger;    @SpringBootApplication  public class Application  {      private static Logger LOGGER = LogManager.getLogger(Application.class);        public static void main(String[] args) {          SpringApplication.run(Application.class, args);            LOGGER.info("Simple log statement with inputs 1, 2 and 3");      }  } |

## Spring Boot with H2 Database

Learn to configure H2 database with [Spring boot](https://howtodoinjava.com/spring-boot-tutorials/) to create and use an in-memory database in runtime, generally for [unit testing](https://howtodoinjava.com/junit-4/) or POC purposes. Remember an in-memory database is created/initialized when an application starts up; and destroyed when the application shuts down.

## What is H2 Database?

H2 is one of the popular in-memory databases written in Java. It can be embedded in Java applications or run in the client-server mode.

## Maven Dependency

To use H2 in Spring boot application, all we need to do is adding H2 runtime jar into dependencies. The best way to add is through maven.

|  |
| --- |
| pom.xml |
| <dependency>      <groupId>com.h2database</groupId>      <artifactId>h2</artifactId>      <scope>runtime</scope>  </dependency> |

## H2 Configuration Options

#### 3.1. **Simple configuration**

Spring provides very easy configuration options to connect to any database using simple properties. Below are the configuration properties, we shall have in application.properties file.

|  |
| --- |
| application.properties |
| spring.datasource.url=jdbc:h2:mem:testdb  spring.datasource.driverClassName=org.h2.Driver  spring.datasource.username=sa  spring.datasource.password=  spring.jpa.database-platform=org.hibernate.dialect.H2Dialect |

Please note by default, Spring Boot configures the in-memory database connection with the username 'sa' and an empty password ' '. If you wish to change these values, override them in above properties options.

## H2 Console

#### 5.1. **Enable H2 console**

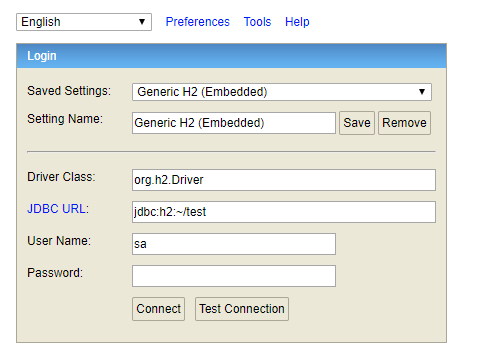
By default, the console view of H2 database is disabled. We must enable it to view and access it in browser. Note that we can customize the URL of H2 console which, by default, is '/h2'.

|  |
| --- |
| application.properties |
| # Enabling H2 Console  spring.h2.console.enabled=true    # Custom H2 Console URL  spring.h2.console.path=/h2 |

#### 5.2. **Accessing H2 console**

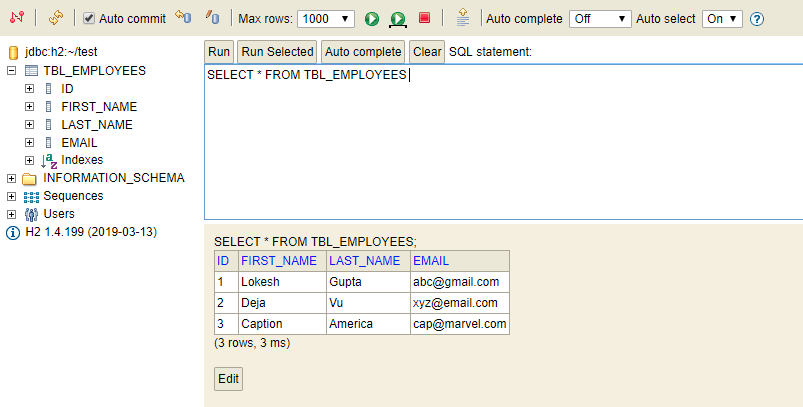
Start the spring boot application and access the console in browser with URL : http://localhost:8080/h2.

We can see the console like this.



**H2 Database Console Login Window**

Now enter the configured username and password. We can verify the table structure and default data inserted through SQL files.

**H2 Console View**

#### 5.3. **Other configuration options**

Spring boot provides two more properties to further customize the behavior of H2 console. i.e. we can enable/disable the database trace logs and we can enable/disable the remote access of H2 console.

By default both properties are false.

|  |
| --- |
| application.properties |
| # Whether to enable trace output.  spring.h2.console.settings.trace=false    # Whether to enable remote access.  spring.h2.console.settings.web-allow-others=false |

#### **Feign Client in Microservices**

**Introduction:**

In microservice architecture, all services are running independently with respect to other microservices.

All services are registered with Eureka server and Gateway is responsible to redirect the all incoming requests.

In microservices sometimes there is a requirement to get data from one microservice to complete an Incoming request on other microservice.

So we can simply use rest Api call for performing such tasks or we can use feignClient provided by Netflix.

Here is the example of same

1. This is the main class where we are required to use @EnableEurekaClient annotation

<https://www.oodlestechnologies.com/blogs/Feign-Client-in-Microservices/>

2. This is the interface where Feign client integrate the Rest API call

(i) @FeignClient("api-accounts")

Here api-accounts is application name which rest API is called by using the feign client in this application

(ii) UrlMapping.GET\_MARKET\_STATUS is the URL of api-accounts application

Here is the code of UrlMapping

public static final String GET\_MARKET\_STATUS = "/api-accounts/v1/market/status";

Use Netflix Feign client to call a rest API of a microservice in another microservice