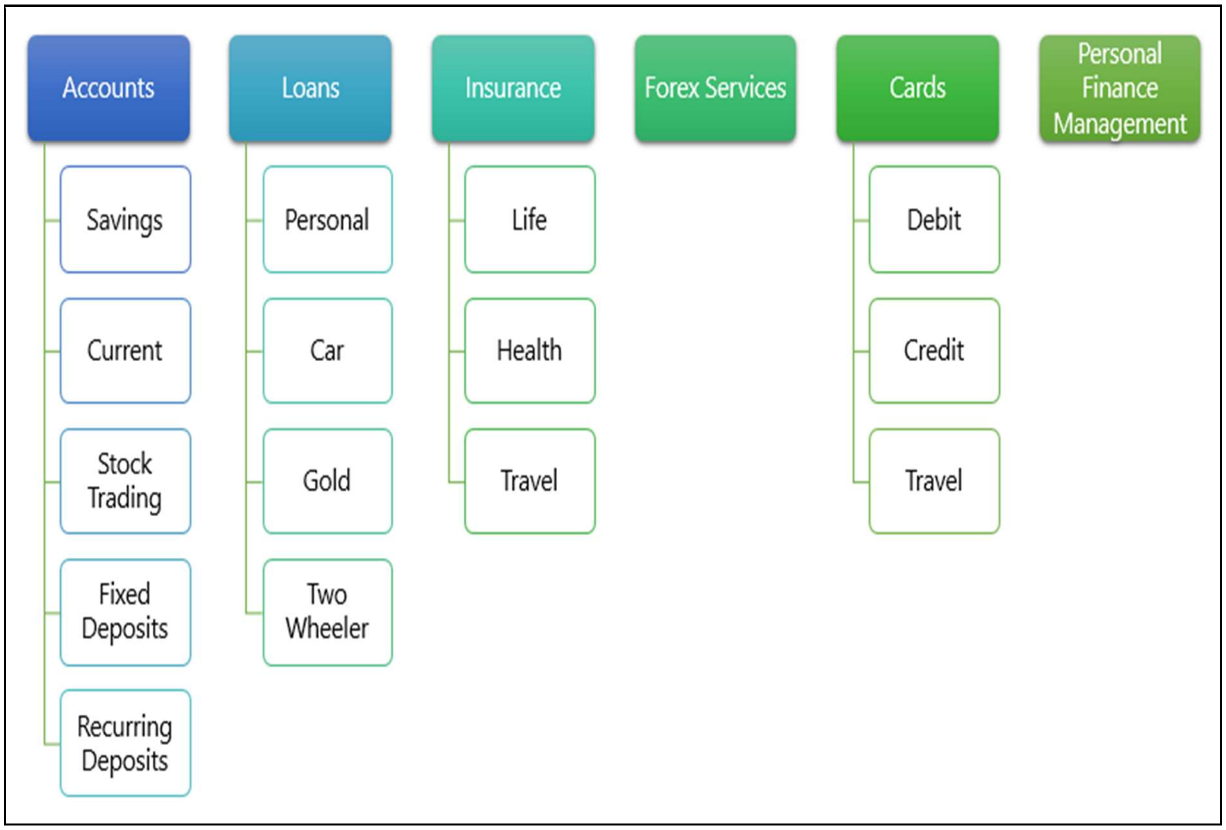
Theory:

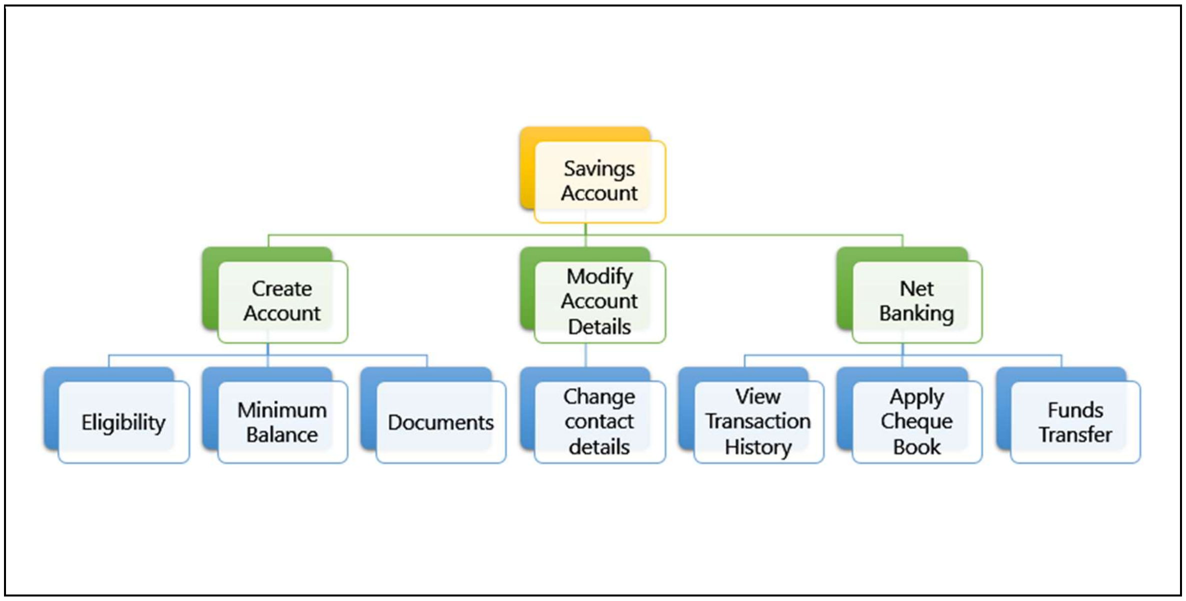
Enterprise Application

Have you ever thought of various products and services offered by a Bank. Have a look at the diagram below.



Still there are few aspects missing like Phone Banking, SMS Banking, Micro Finance, etc.

Let us take one service "Savings Account" from the above diagram. Few aspects associated with Savings Account is defined below and this is not an exhaustive list.



Systems and applications that manages the service levels of these kind of organizations are called **Enterprise Application**.

Activity

- Go to https://www.airtel.in/ and find out all products and services that are offered.
- SME to write the services offered by airtel on the board based on inputs from the learners.

Monolithic Service

Let us take the frequently used operation in a Bank, get account balance of an account. We already understand that this operation can be initiated from various roles and using various devices. Few example scenarios where this service is used:

- A customer viewing his account details in a mobile app
- A bank teller viewing account details of a customer
- A customer viewing account details over net banking application
- An IVR system reading out an account balance to a customer
- A customer service representative views the account details of a customer whom had called customer services
- A batch job that deducts EMI on your account should know the account balance before performing the transaction
- A customer viewing his account details in an ATM
- A SMS system that needs to send transaction details to customer's mobile should know the account balance, so that it can be included in the SMS text message.
- so on and so forth ......

Similar to the one above, various operations of a bank can get initiated from various devices, roles, internal systems, etc.

A XYZ Software Services company develops an application for a Bank that implements all operations of the bank as RESTful Web Services. You name any service offered by the Bank, a service is available that can be consumed by the respective application. This project had been completed and had been recently launched live.

After few days of launch, on one fine day at 4PM ...

- A loan agent was not able to submit a loan application, he might miss his monthly target
- An insurance agent was not able to process closure of an insurance and hand over the sum assured payment cheque. The customer is waiting for more than one hour to receive this cheque.
- A customer is waiting in customer service queue for the 25 minutes to report a stolen credit card

The primary reason for the above situation is that the RESTful Web Service application recently launched has become very slow in responding. Due to festival season shopping there were huge volume of transactions for getting account balance, since there was a memory leak in the code and there is not memory left, because of which new request coming to the server were either rejected or timed out.

To overcome this situation the entire server had to be restarted. After restart, the situation becomes normal after 2 to 3 hours. The support team keeps their fingers crossed not sure when this issue crops us again.

Activity
SME to discuss with learners and come up with ideas to handle this situation

Monolithic Services vs. Microservices

Monolithic Services - A large number of critical enterprise applications hosted as a single web application is called as monolithic service.

The following are the drawbacks of this Monolithic Services:

- As everything is packaged in one EAR/WAR. If any one service has a performance or memory leak issue, it brings down all the services.
- There is no possiblity to try a different technology stack for building the services

Microservices - Instead of having a single monolithic service, the services can be split into multiple services. Taking the example from the bank, insurance related operations can be handled by a separate service. Each microservice will be running

Advantages of Microservices

- Decentralized
- Independent (Let us consider the bank application, the get balance service failure resulted in bringing down other services related to insurance and loan processing. This single point failure can be avoided when implementing as microservices)
- Doing one this well
- Agility in development of a service
- Scalable - add multiple instances of a service with new hardware included without affecting the existing production environment
- Easier to identify which service is in fault
- Makes it easier for a new developer to understand the functionality of a service, enables continuous delivery.

Challenges of Microservices

- Developing distributed systems can be complex
- Initial implementation of microservice is difficult

Creating Microservices for account and loan

In this hands on exercises, we will create two microservices for a bank. One microservice for handing accounts and one for handling loans.

Each microservice will be a specific independent Spring RESTful Webservice maven project having it's own pom.xml. The only difference is that, instead of having both account and loan as a single application, it is split into two different applications. These webservices will be a simple service without any backend connectivity.

Follow steps below to implement the two microservices:

Account Microservice

- Create folder with employee id in D: drive
- Create folder named 'microservices' in the new folder created in previous step. This folder will contain all the sample projects that we will create for learning microservices.
- Open https://start.spring.io/ in browser
- Enter form field values as specified below:
    - **Group:** com.cognizant
    - **Artifact:** account
- Select the following modules
    - Developer Tools > Spring Boot DevTools
    - Web > Spring Web
- Click generate and download the zip file
- Extract 'account' folder from the zip and place this folder in the 'microservices' folder created earlier
- Open command prompt in account folder and build using mvn clean package command
- Import this project in Eclipse and implement a controller method for getting account details based on account number. Refer specification below:
    - Method: GET
    - Endpoint: /accounts/{number}
    - Sample Response. Just a dummy response without any backend connectivity.

```
{ number: "00987987973432", type: "savings", balance: 234343 }
```

- Launch by running the application class and test the service in browser

Loan Microservice

- Follow similar steps specified for Account Microservice and implement a service API to get loan account details
    - Method: GET
    - Endpoint: /loans/{number}
    - Sample Response. Just a dummy response without any backend connectivity.

```
{ number: "H00987987972342", type: "car", loan: 400000, emi: 3258, tenure:
18 }
```

- Launching this application by having account service already running
- This launch will fail with error that the bind address is already in use
- The reason is that each one of the service is launched with default port number as 8080. Account service is already using this port and it is not available for loan service.
- Include "server.port" property with value 8081 and try launching the application
- Test the service with 8081 port

Now we have two microservices running on different ports.

NOTE: The console window of Eclipse will have both the service console running. To switch between different consoles use the monitor icon within the console view.

Create Eureka Discovery Server and register microservices

Eureka Discovery Server holds a registry of all the services that are available for immediate consumption. Anybody whom wants to consume a RESTful Web Service can come to the discovery server and find out what is available and ready for consumption. Eureka Discovery Server is part of spring cloud module.

Follow steps below to implement:

Create and Launch Eureka Discovery Server

- Using https://start.spring.io generate a project with following configuration:
  - Group: com.cognizant
  - Artifact: eureka-discovery-server
  - Module: Spring Cloud Discovery > Eureka Server
- Download the project, build it using maven in command line
- Import the project in Eclipse
- Include @EnableEurekaServer in class EurekaDiscoveryServerApplication
- Include the following configurations in application.properties:

```
server.port=8761


eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false


logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF
```

- The above configuration runs the discovery service in port 8761
- The eureka properties prohibits direct registration of services, instead discovery server will find available services and register them.
- Launch the service by running the application class
- The discovery service can be view by launching http://locahost:8761 in the browser.
- This will display the discover server details

- Look into the section "Instances currently registered with Eureka", which will have an empty list
- Follow steps below to add account and loan service to this discovery server.

Register Account REST API to eureka discovery

- Go to https://start.spring.io and provide the following configuration:
    - Group: com.cognizant
    - Artifact: account
    - Modules:
        - Spring Boot DevTools
        - Eureka Discovery Client
        - Spring Web
- Click "Explore", which will open pom.xml
- Use copy option in the opened window to copy the pom.xml and overwrite the pom.xml in account project
- Build the project using maven in console
- Include @EnableDiscoveryClient annotation to application class of account project
- Include application name for account application as specified below in application.properties. This is the name that will be displayed in the eureka discovery registry.

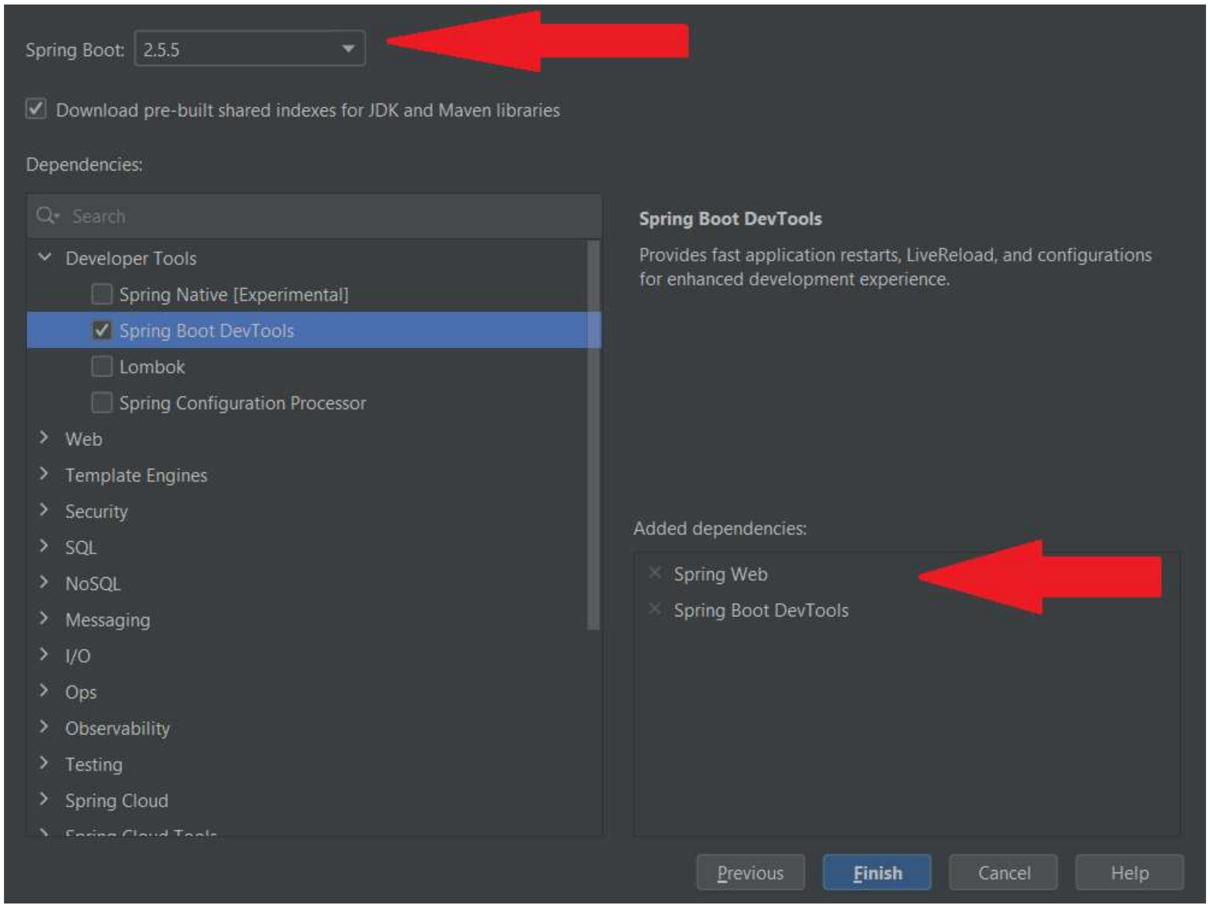
```
spring.application.name=account-service
```

- Stop all services (account, loan, eureka-discovery-server) using the console window of Eclipse. Use the monitor icon in console view to switch between applications and use the Terminate button to stop the server.
- First start eureka-discovery-server and wait till the application starts completely. Then open http://locahost:8761 in browser. The service list should be empty.
- Then start account application and wait till the application starts.
- Refresh the eureka-discovery-server web page in browser, the account-service will be listed in the registry
- Perform similar steps for loan application and have it registered with eureka-discovery-server.

Create a Spring Cloud API Gateway and call one microservice thru the API gateway. Configure a global filter to log each request targeting the microservice using Spring Cloud API Gateway.

Steps.

1. Create Simple Microservice **greet-service** that returns “Hello World” using Spring Initializer.
2. Select the latest version of Spring Boot and the dependencies as shown in the image below.



Spring Boot: 2.5.5

☑ Download pre-built shared indexes for JDK and Maven libraries

Dependencies:

- Developer Tools
  - ☐ Spring Native [Experimental]
  - ☑ Spring Boot DevTools
  - ☐ Lombok
  - ☐ Spring Configuration Processor
- Web
- Template Engines
- Security
- SQL
- NoSQL
- Messaging
- I/O
- Ops
- Observability
- Testing
- Spring Cloud

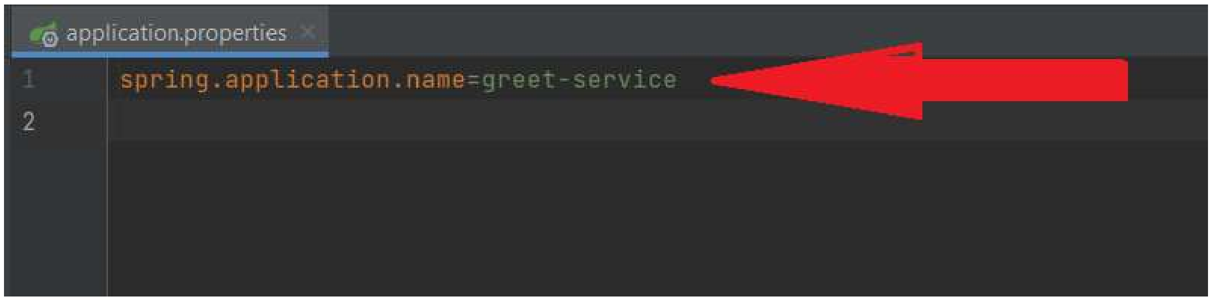
Spring Boot DevTools

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Added dependencies:

- Spring Web
- Spring Boot DevTools

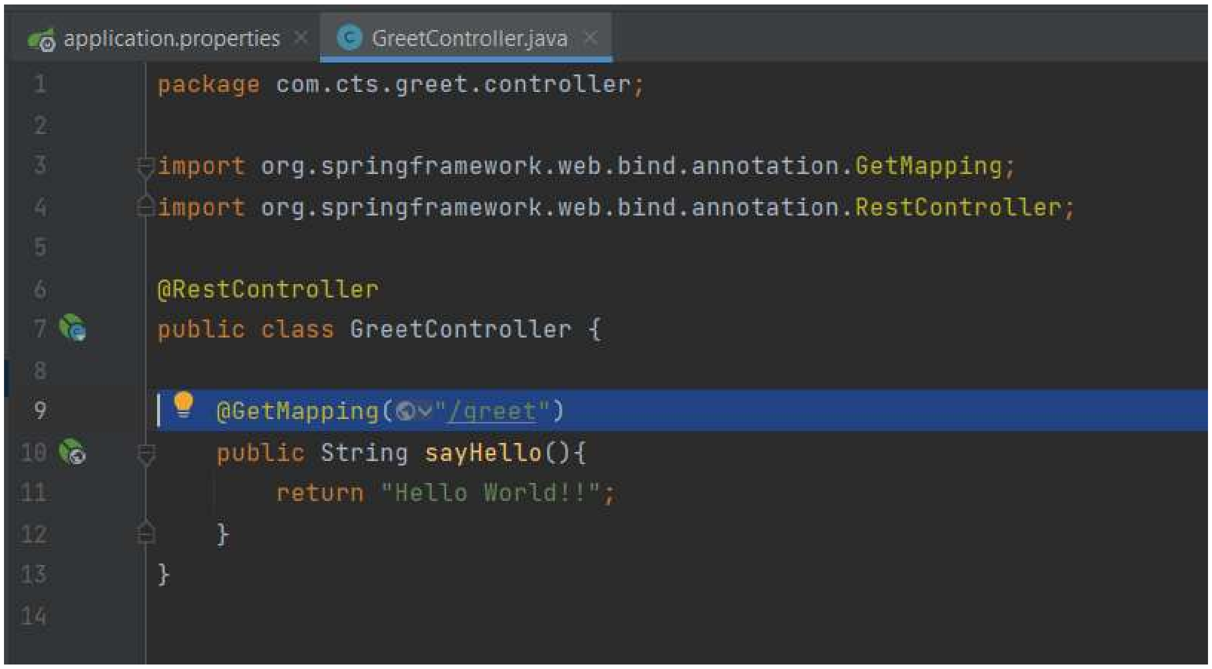
3. Configure the application name in “application.properties” as shown below



application.properties

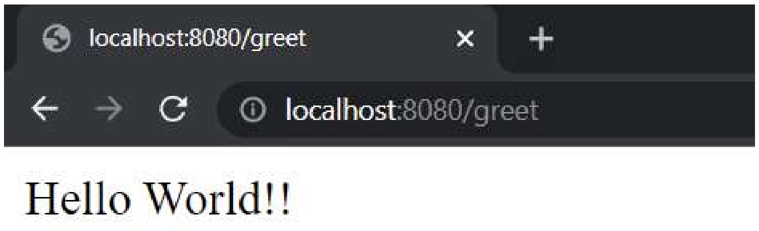
```
spring.application.name=greet-service
```

4. Create a controller as shown below.



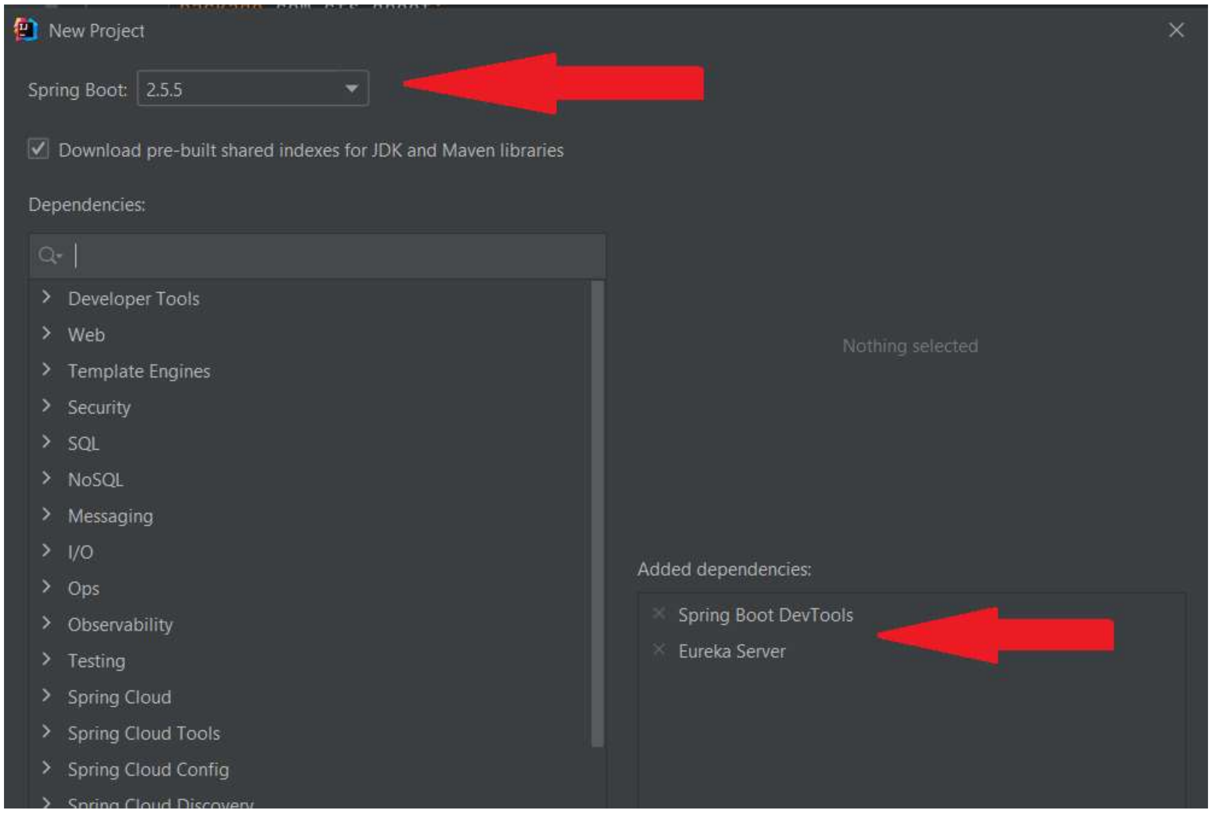
```java
package com.cts.greet.controller;

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

@RestController
public class GreetController {

    @GetMapping("/greet")
    public String sayHello(){
        return "Hello World!!";
    }
}
```

5. Run the microservice and make sure that it is working fine as shown below



```
Hello World!!
```

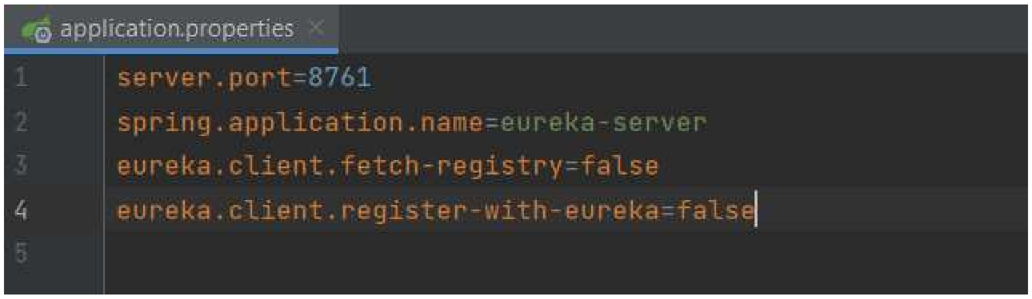
6. Create a second microservice that is acting as the Discovery Server using Spring Cloud Eureka Server.
7. Select the latest version of Spring Boot and the required dependencies as shown below



Spring Boot: 2.5.5

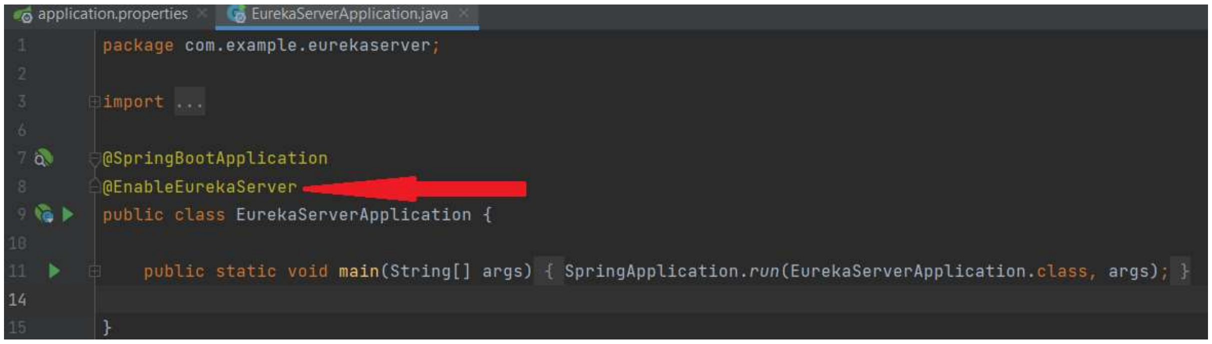
Added dependencies:
Spring Boot DevTools
Eureka Server

8. Include the following configurations in the “application.properties” of the Eureka server



```
server.port=8761
spring.application.name=eureka-server
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

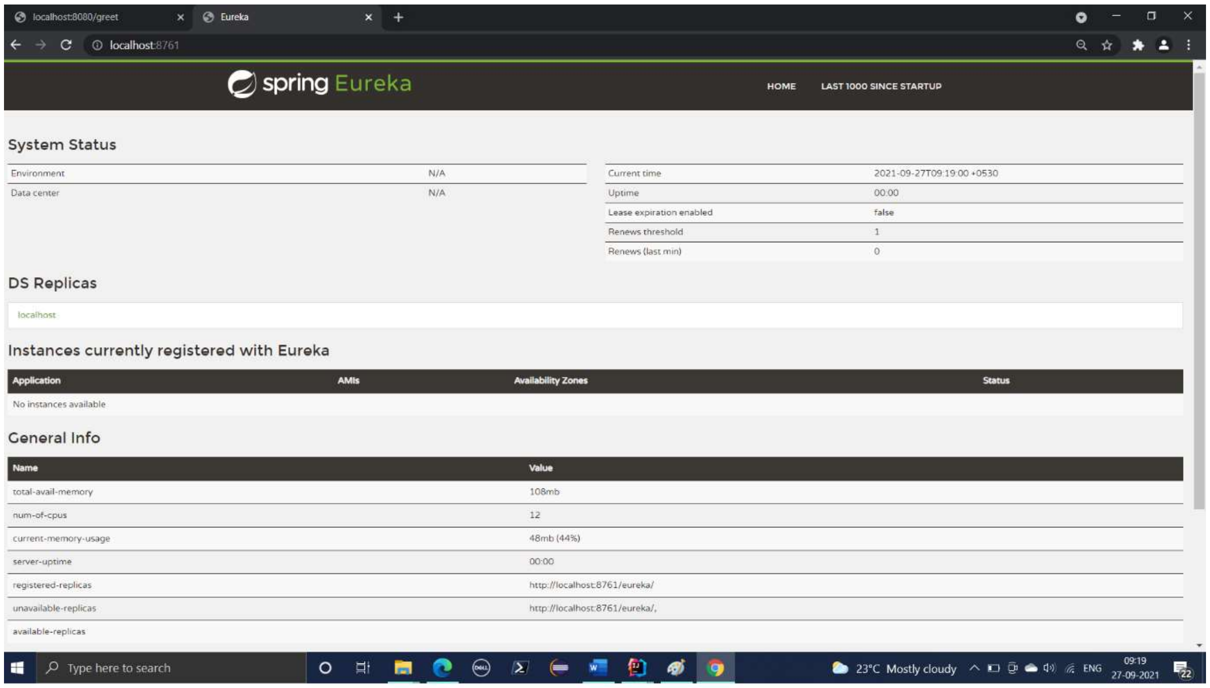
9. Annotate the main class in Eureka Server with @EnableEurekaServer as shown below.



```
package com.example.eurekaserver;

import ...

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

    public static void main(String[] args) { SpringApplication.run(EurekaServerApplication.class, args); }

}
```

10. Run the Eureka Server and ensure that the server is functioning properly by entering the below URL at any of the browser.

http://localhost:8761



spring Eureka

HOME LAST 1000 SINCE STARTUP

System Status

Environment	N/A	Current time	2021-09-27T09:19:00 +0530
Data center	N/A	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

localhost

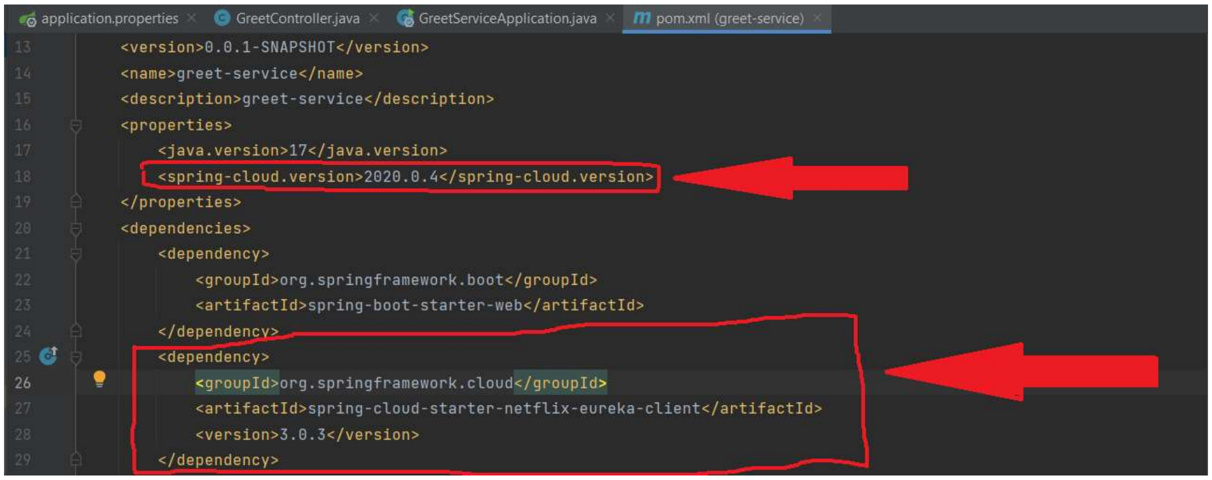
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

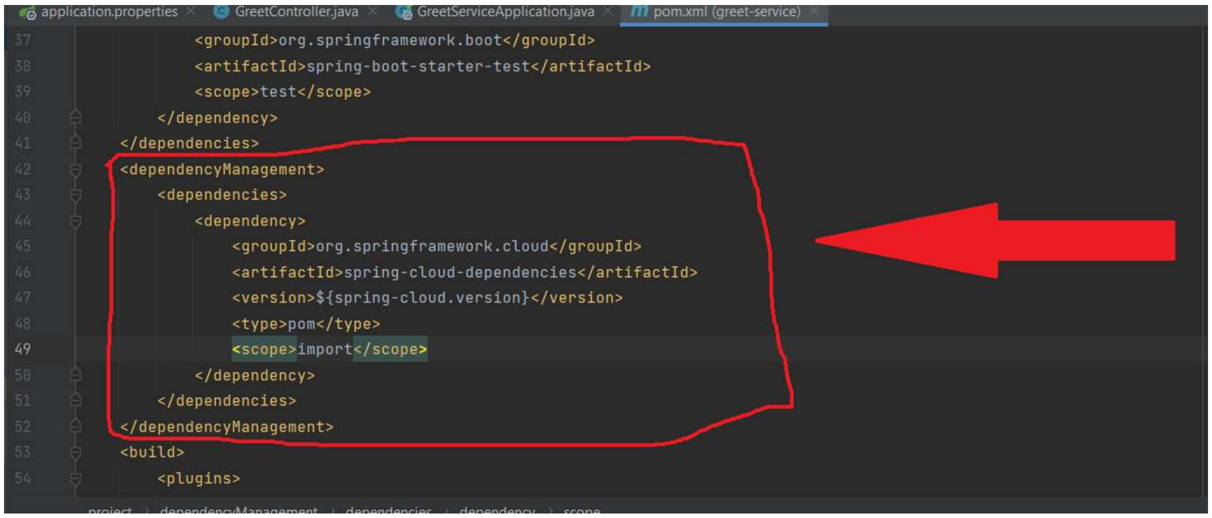
Name	Value
total-avail-memory	108mb
num-of-cpus	12
current-memory-usage	48mb (44%)
server-uptime	00:00
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

11. Open the pom.xml of greet-service microservice and add the eureka client dependencies as shown in the image below.
12. Copy the spring cloud version from pom.xml of eureka-server and paste it at the appropriate location in the pom.xml of greet-service as shown below.



```xml
    <version>0.0.1-SNAPSHOT</version>
    <name>greet-service</name>
    <description>greet-service</description>
    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2020.0.4</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>
            <version>3.0.3</version>
        </dependency>
```

13. Copy the “dependency-management” session in the pom.xml of the eureka-server and paste it immediately after the dependencies session of the pom.xml of the greet-service as shown below.

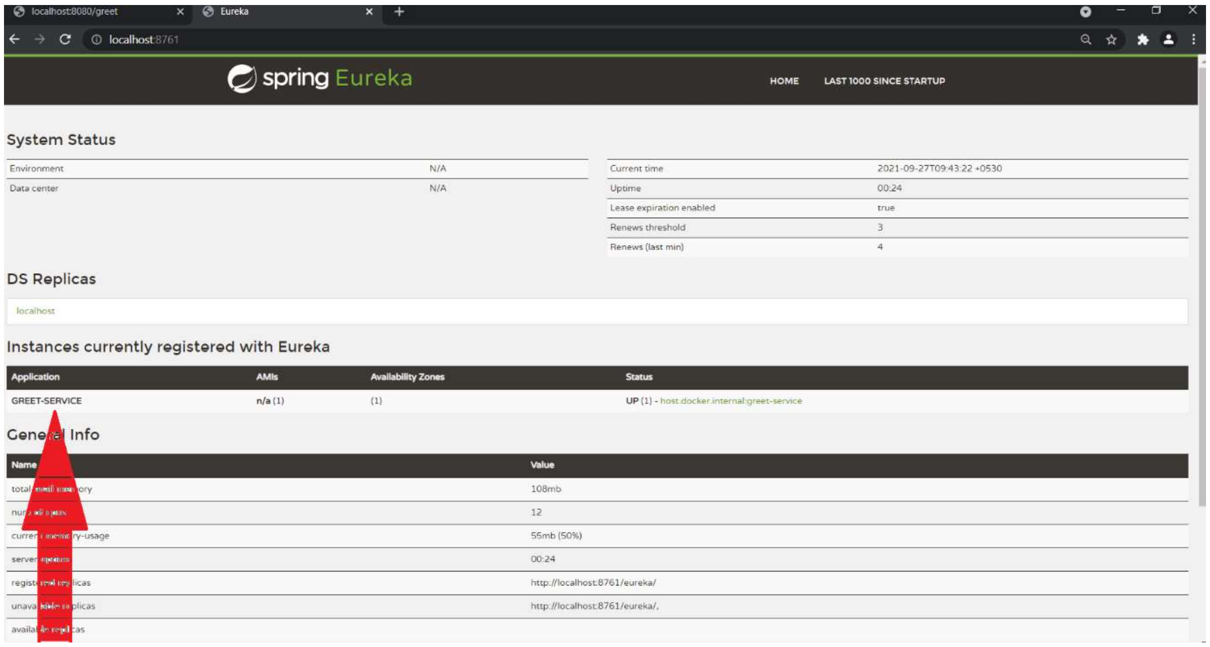


application.properties
GreetController.java
GreetServiceApplication.java
pom.xml (greet-service)

```
            <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>
```

project › dependencyManagement › dependencies › dependency › scope

14. Restart greet-service and refresh the http://localhost:8761 to see whether the name of the greet-service figuring in the eureka-server console as shown below.



System Status

Environment	N/A
Data center	N/A

Current time	2021-09-27T09:43:22 +0530
Uptime	00:24
Lease expiration enabled	true
Renews threshold	3
Renews (last min)	4

DS Replicas

localhost

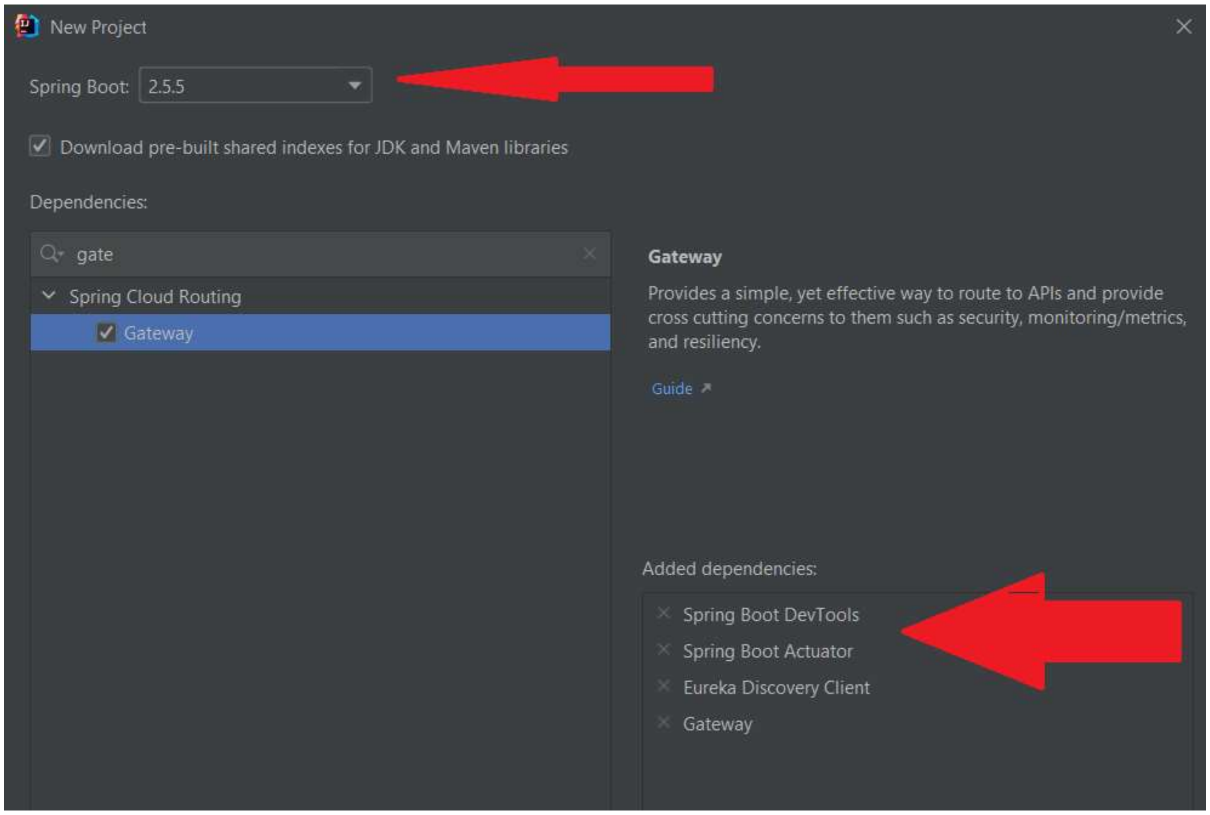
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
GREET-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:greet-service

General Info

Name	Value
total-avail-memory	108mb
num-of-cpus	12
current-memory-usage	55mb (50%)
server-uptime	00:24
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

15. Create another microservice “api-gateway” using spring initializer.
16. Select the latest version of Spring Boot and add the required dependencies as shown below.



Spring Boot: 2.5.5

[✓] Download pre-built shared indexes for JDK and Maven libraries

Dependencies:

gate

Spring Cloud Routing

[✓] Gateway

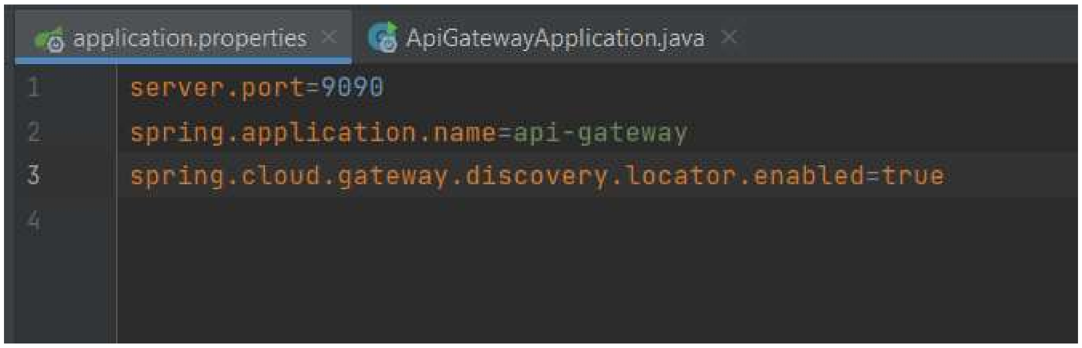
Gateway

Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency.

Added dependencies:

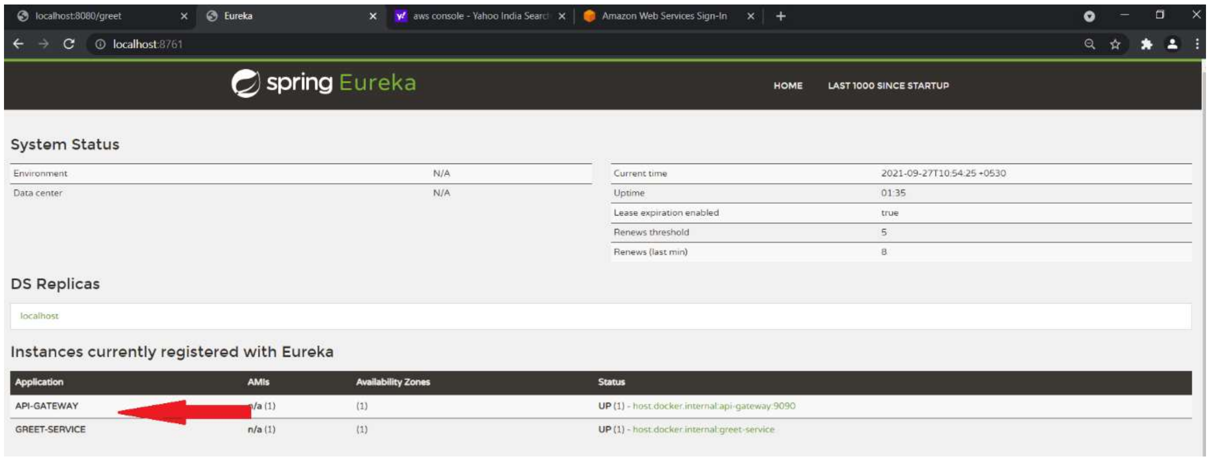
- Spring Boot DevTools
- Spring Boot Actuator
- Eureka Discovery Client
- Gateway

17. Make the necessary configurations in the “application.properties” file as shown below.



```
server.port=9090
spring.application.name=api-gateway
spring.cloud.gateway.discovery.locator.enabled=true
```

18. Run the api-gateway service and check if it is getting registered with the eureka-server.



System Status

Environment	N/A
Data center	N/A

Current time	2021-09-27T10:54:25 +0530
Uptime	01:35
Lease expiration enabled	true
Renews threshold	5
Renews (last min)	8

DS Replicas

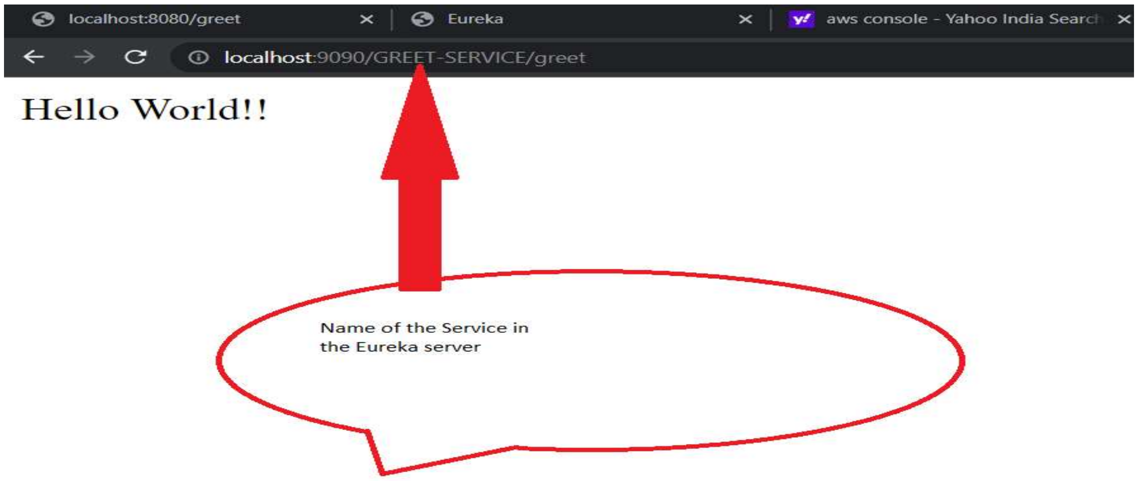
localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - host.docker.internal:api-gateway:9090
GREET-SERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:greet-service

19. Try to access the following URL from any of the browser and see if you are able to access the greet-service thru the api-gateway.

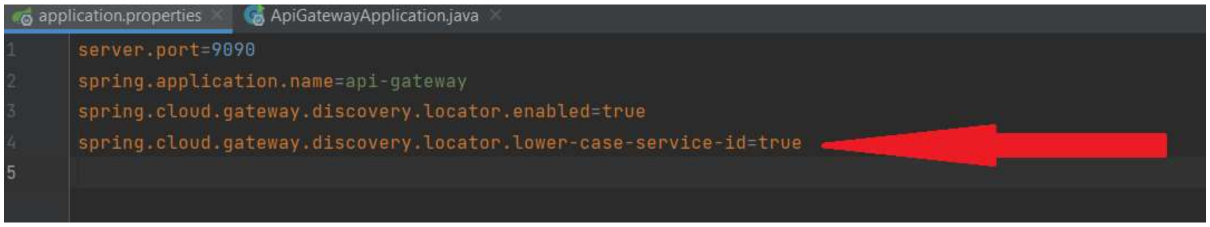
    http://localhost:9090/GREET-SERVICE//greet



localhost:9090/GREET-SERVICE/greet
Name of the Service in the Eureka server

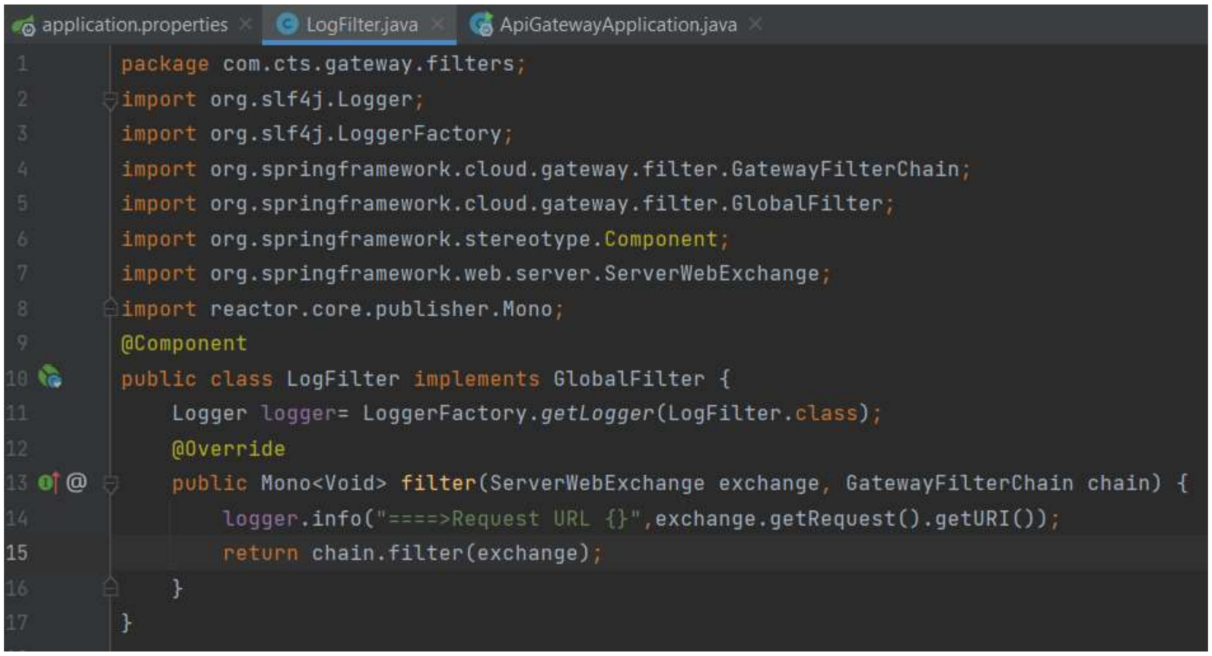
```
Hello World!!
```

20. Include the following configuration in the " application.properties" of the api-gateway to specify the service name in lower case.



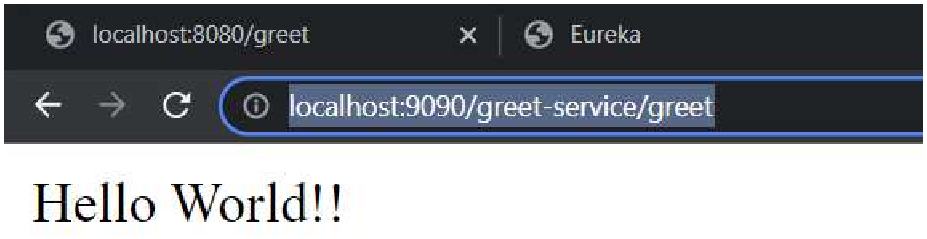
```
server.port=9090
spring.application.name=api-gateway
spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true
```

21. Implement a global filter which logs all incoming requests.
    a. Create a LogFilter class as shown below.



```java
package com.cts.gateway.filters;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;
@Component
public class LogFilter implements GlobalFilter {
    Logger logger= LoggerFactory.getLogger(LogFilter.class);
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        logger.info("====>Request URL {}",exchange.getRequest().getURI());
        return chain.filter(exchange);
    }
}
```

22. Try to access the url http://localhost:9090/greet-service/greet. You will get the output as shown below.



Hello World!!

23. Check the console of the api-gateway service and you should be getting the log shown below.