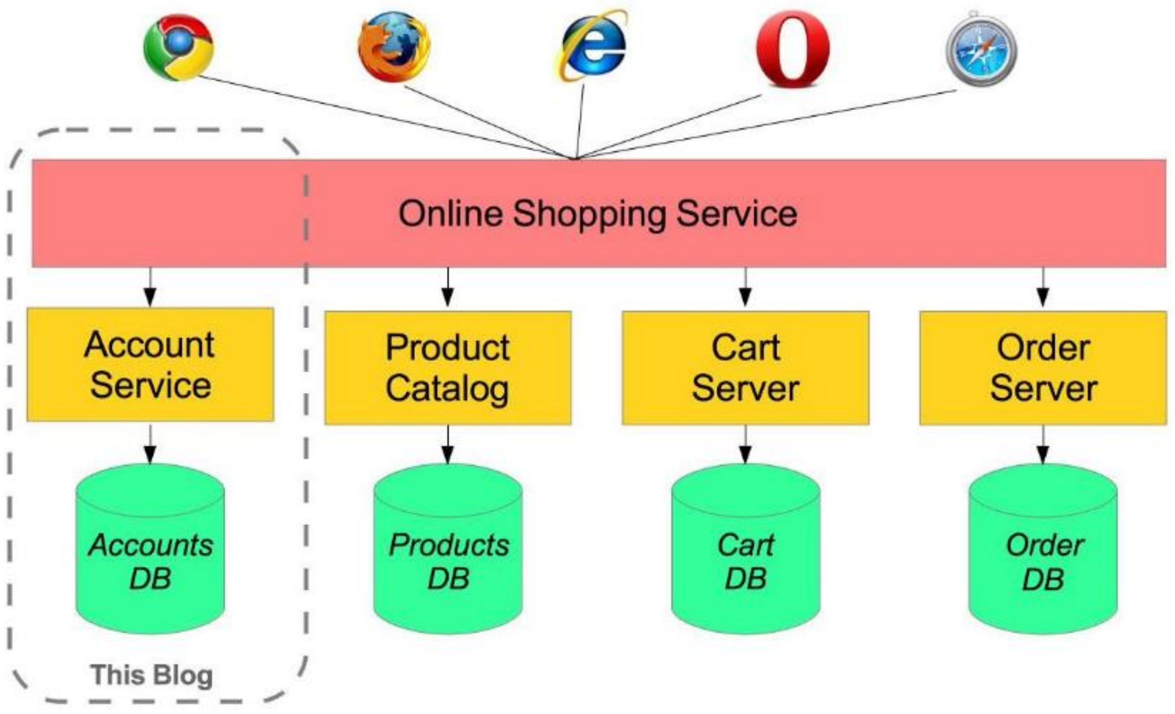
Microservices

It is an architectural style that structures an application as a collection of services:

- Each microservice can have its own database.
- Each microservice can be developed independently.
- Each microservice can be deployed independently.



Spring Boot

It is a very popular Java framework for building Restful web services and microservices. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

Steps to create Microservices using Spring-Boot:

1) Build the Registration service.
2) Build business logic services.
3) Deploy business logic services on registration services.

Maven Dependencies you may need:

```xml
<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-eureka-server</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
   </dependency>
   <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
   </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>
         <configuration>
            <excludes>
               <exclude>
                  <groupId>org.projectlombok</groupId>
                  <artifactId>lombok</artifactId>
               </exclude>
            </excludes>
         </configuration>
      </plugin>
   </plugins>
</build>
<repositories>
```

```
   <repository>
      <id>netflix-candidates</id>
      <name>Netflix Candidates</name>
      <url>https://artifactory-oss.prod.netflix.net/artifactory/maven-oss-
candidates</url>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
   </repository>
</repositories>
```

Or you can use https://start.spring.io/ and add dependencies you need automatically.

Dependency you may need:

1) Lombok
2) Eureka
3) spring-boot dev-tools

Steps:

1.Building Registration Service

When you have multiple processes working together they need to find each other. If you have ever used Java's RMI mechanism you may recall that it relied on a central registry so that RMI processes could find each other. Microservices has the same requirement.

The developers at Netflix had this problem when building their systems and created a registration server called Eureka ("I have found it" in Greek).

They made their discovery server open-source and Spring has incorporated into Spring Cloud, making it even easier to run up a Eureka server. Here is the *complete* discovery-server application:

```
@SpringBootApplication //related to spring-boot project
@EnableEurekaServer // Annotation for eureka server
public class ServiceRegistrationServer {
  public static void main(String[] args) {
    // Tell Boot to look for registration-server.yml
    SpringApplication.run(ServiceRegistrationServer.class, args);
  }

}
```

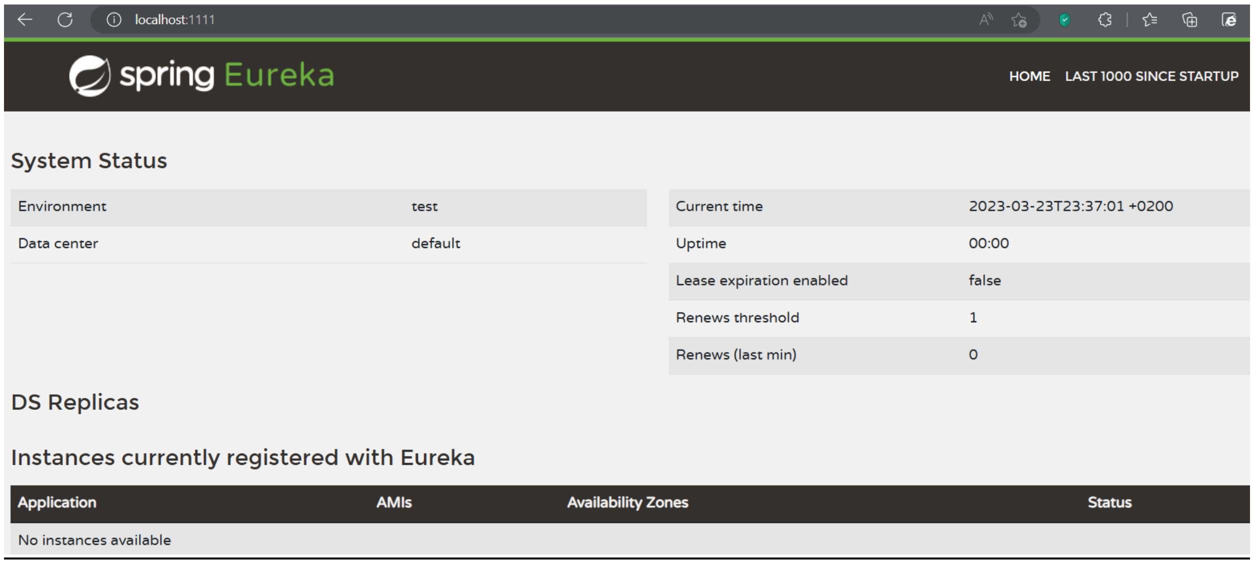
Registration-server.yml:

```
eureka:
  instance:
    hostname: localhost
  client:
    fetchRegistry: 'false'
    registerWithEureka: 'false'
server:
  port: '1111'
```

if it is a properties file:

```
eureka.instance.hostname=localhost
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
server.port=1111
```

When you run registration service and go to localhost:1111, you can see eureka page without instance available which means there is no service registered on the eureka service so we will build the business logic services.



System Status

Environment	test
Data center	default

Current time	2023-03-23T23:37:01 +0200
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

2.Building business logic service

Maven Dependencies you may need:

The same maven dependencies from the previous project

You will have two classes:

1) Service (responsible for run service)
2) Service controller

Service class:

```
@SpringBootApplication
@EnableDiscoveryClient

public class service {

    /**
     * Run the application using Spring Boot and an embedded servlet engine.
     *
     * @param args Program arguments - ignored.
     */
    public static final String REGISTRATION_SERVER_HOSTNAME =
"registration.server.hostname";
    public static void main(String[] args) {
        // Default to registration server on localhost

        if (System.getProperty(REGISTRATION_SERVER_HOSTNAME) == null)
            System.setProperty(REGISTRATION_SERVER_HOSTNAME, "localhost");

        // Tell server to look for service.properties or service.yml
        System.setProperty("spring.config.name", "calculator");

        SpringApplication.run(service.class, args);
    }
}
```

Service.properties:

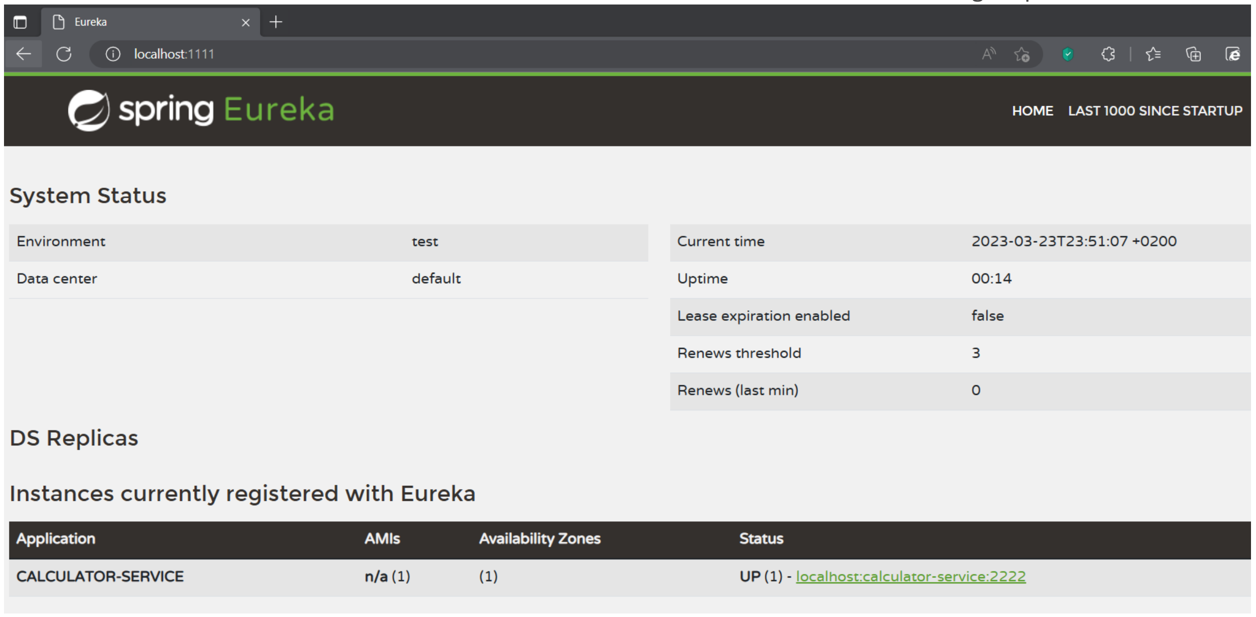
```
spring.application.name=calculator-service
spring.freemarker.enabled=false
spring.thymeleaf.cache=false
spring.thymeleaf.prefix=classpath:/accounts-server/templates/
error.path=/error
server.port=2222
eureka.client.serviceUrl.defaultZone=http://${registration.server.hostname}:1111/eurek
a/
eureka.instance.leaseRenewalIntervalInSeconds=10
management.endpoints.web.exposure.include=*
```

Service Controller:

```
@RestController //controller
@RequestMapping("/test")// Prefix need to be in link to run these methods
public class service_controller {
{
@GetMapping("/addition")// http://localhost:9090/prints
@ResponseBody
public int add(@RequestParam int  num1,@RequestParam int num2)
{
    return num1+num2;
}
}
```

<u>3. Deploy business logic services on registration services:</u>

When we run this service and refresh localhost:1111 we will see that service is running on port 2222



System Status

Environment	test
Data center	default

Current time	2023-03-23T23:51:07 +0200
Uptime	00:14
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

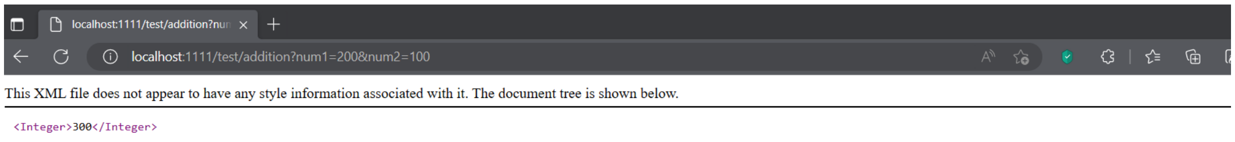
DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CALCULATOR-SERVICE	n/a (1)	(1)	UP (1) - localhost:calculator-service:2222

We can see if we go to http://localhost:1111/test/addition?num1=200&num2=100

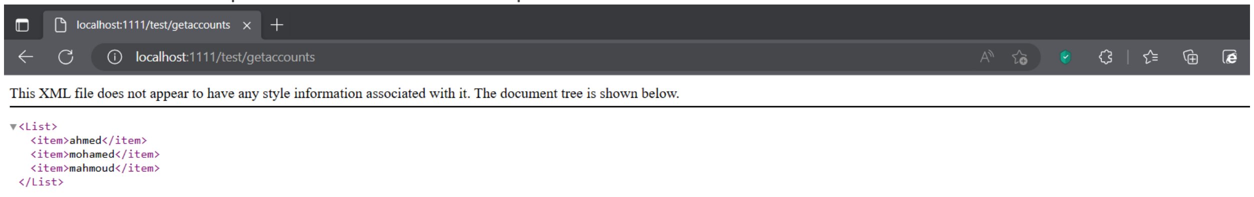
We can see the result:


localhost:1111/test/addition?nu
localhost:1111/test/addition?num1=200&num2=100

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<Integer>300</Integer>
```

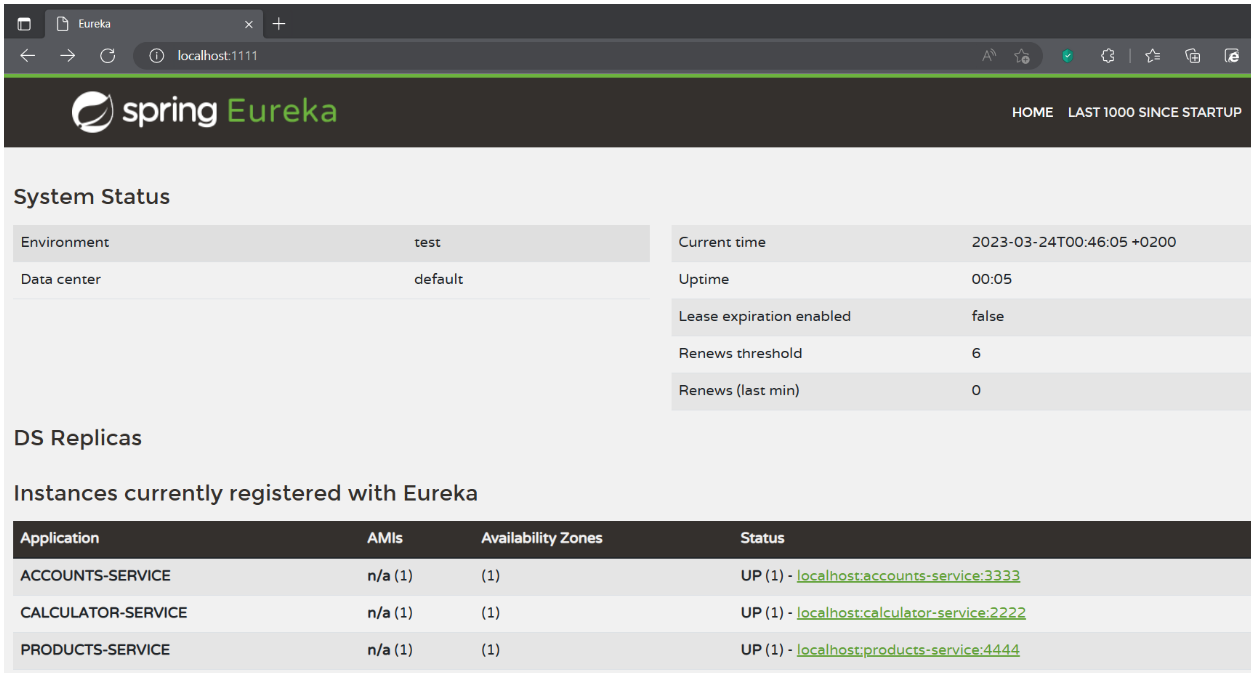
After we add other services(account service, product service)
These service are responsible for accounts and products


localhost:1111/test/getaccounts

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<List>
   <item>ahmed</item>
   <item>mohamed</item>
   <item>mahmoud</item>
 </List>
```

We can see here that three services on different port numbers, but we can access them using registration service on localhost:1111



System Status

Environment	test
Data center	default

Current time	2023-03-24T00:46:05 +0200
Uptime	00:05
Lease expiration enabled	false
Renews threshold	6
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS-SERVICE	n/a (1)	(1)	UP (1) - localhost:accounts-service:3333
CALCULATOR-SERVICE	n/a (1)	(1)	UP (1) - localhost:calculator-service:2222
PRODUCTS-SERVICE	n/a (1)	(1)	UP (1) - localhost:products-service:4444

Link for the complete microservices application: https://github.com/paulc4/microservices-demo.

Docker

It provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows you to run many containers simultaneously on a given host.

Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host.

You can easily share containers while you work and be sure that everyone you share with gets the same container that works in the same way.

What can I use Docker for?

Fast, consistent delivery of your applications

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services.

Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Your developers write code locally and share their work with their colleagues using Docker containers.

They use Docker to push their applications into a test environment and execute automated and manual tests.

When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

Responsive deployment and scaling

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.

Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

Running more workloads on the same hardware

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your server capacity to achieve your business goals.

Docker architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers.

The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out.
The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Desktop

Docker Desktop is application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices.
It includes the Docker (daemon, client, Compose ,Content Trust) and Kubernetes, and Credential Helper.

Docker registries

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Images

An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization.
For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.
You might create your own images or you might only use those created by others and published in a registry.

To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it.

Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

Running Microservices Using Docker

We are going to run the Accounts Microservice application available at https://github.com/paulc4/microservices-demo using three docker containers, one for the Eureka registration server and one for each microservice.

Build the Image

1. Fetch the code using git clone https://github.com/paulc4/microservices-demo
2. Build using either mvn package or gradle assemble as you prefer.
   This will create the jar: target/microservices-demo-1.2.0.RELEASE.jar
3. Here is the Dockerfile:

   ```
   FROM openjdk:8-jre
   ADD target/microservices-demo-1.2.0.RELEASE.jar app.jar
   EXPOSE 1111
   EXPOSE 2222
   EXPOSE 3333
   ```

   What this does:

   - Use the OpenJDK 8 docker image (freely available at Docker hub) as a starting point. This image defines a minimal Linux system with OpenJDK 8 preinstalled.
   - Copy the demo jar into the container and rename it to app.jar to save typing later. By default, app.jar will be copied into the root of the container file system.
   - Expose ports 1111, 2222 and 3333.

4. To build the container (**note** the . at the end, indicating to use the current directory as its working directory):
   - ➔ docker build -t paulc4/microservice .
5. Check it worked. You should see paulc4/microservice listed.
   - ➔ docker images

Running the Application

We will run the container three times, each time running the Java application in a different mode.

1. They need to talk to each other, so let's give them a network:

   ➔ docker network create accounts-net

2. Now run the first container. This runs up the Eureka registration server, which will allow the other microservices to find each other:

   ➔ docker run --name reggo --hostname reggo --network accounts-net -p 1111:1111 paulc4/microservice java -jar app.jar reg

   The -d (detach) flag is missing so all output will stream to the console so we can see what is happening.

   As soon as the application starts up, it displays its IP address. Remember this for later.

3. In your browser, go to http://localhost:1111 and you should see the Eureka dashboard. There are no instances registered.

4. *In a new CMD/Terminal window*, run a second container for the accounts microservice. This holds a database if 21 available accounts (stored using the H2 in-memory RDBMS database)

   ➔ docker run --name accounts --hostname accounts --network accounts-net -p 2222:2222 paulc4/microservice java -jar app.jar accounts --registration.server.hostname=<reg server ip addr>

   Replace <eg server ip addr> with the IP address you determined earlier.

5. Return to the Eureka Dashboard in your browser and refresh the screen. You should see that ACCOUNTS-SERVICE is now registered.

6. *In a new CMD/Terminal window*, run a third container for the accounts web-service. This is a web-application for viewing account information by requesting account data from the accounts microservice.

   ➔ docker run --name web --hostname web --network accounts-net -p 3333:3333 paulc4/microservice java -jar app.jar web --registration.server.hostname=<eg server ip addr>

   Replace <eg server ip addr> with the IP address you determined earlier.

7. Return to the Eureka Dashboard in your browser and refresh the screen. You should see that ACCOUNTS-SERVICE and WEB-SERVICE are now registered.

8. In a second browser tab, go to http://localhost:3333. This is the web interface you just deployed and you should be able to view, list and search for account information.

References:

1. https://spring.io/blog/2015/07/14/microservices-with-spring
2. https://github.com/paulc4/microservices-demo.
3. https://docs.docker.com/get-started/overview/
4. https://github.com/paulc4/microservices-demo/blob/master/use-docker.md