

Microservices with Spring Boot

Building Microservices Application Using Spring Boot

1. Introduction to Microservices

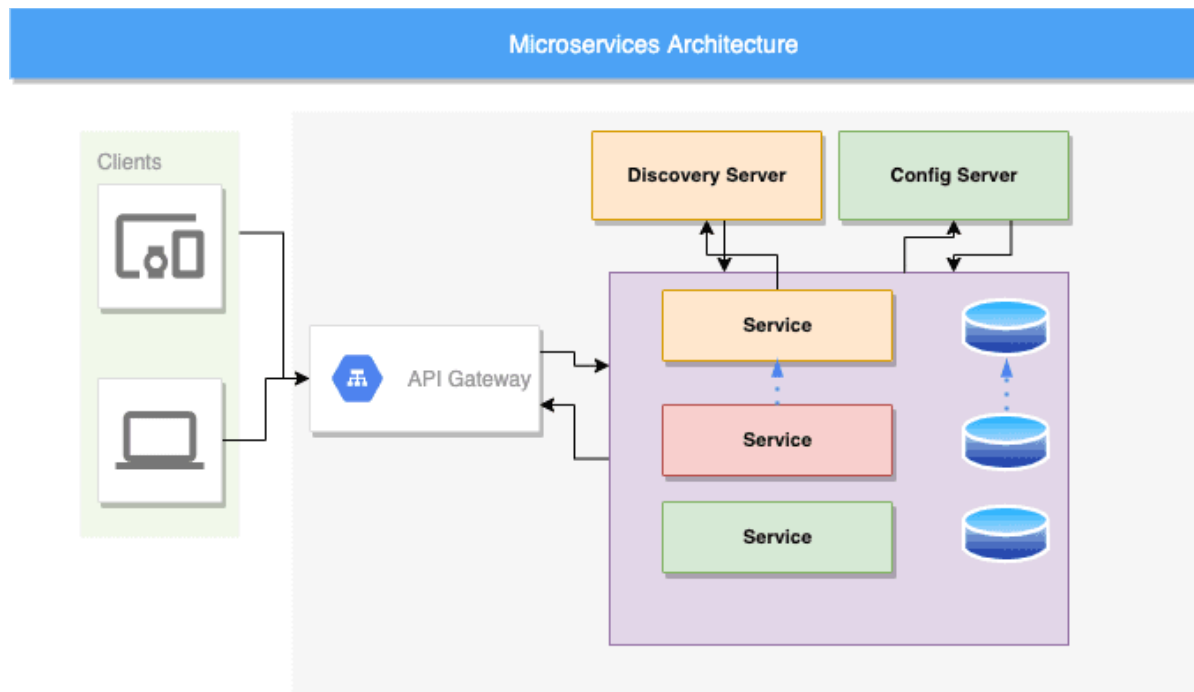
Microservices means many small services, building small, self-contained, ready to run applications. In monolithic architecture, a combination of many components in an application becomes a large application that has many disadvantages. For example, if a monolithic application down, the entire application will be down. Even it's difficult to maintain a large monolithic application.

Microservices breaks a large application to different smaller parts, so it is easy to identify where the problem occurs and also if a component goes down it will not affect the whole application environment. In this article, we will summarize the basics of building microservices with spring boot and spring cloud.

2. Microservices architecture

The concept of microservices is simple. It should break a large service with many small independent services. Let's discuss some important points of microservices based on below architecture:

- Each micro service has its own database.
- Client API do not have direct access to the services. It can only interact through API gateway.
- We will register each service with the discovery server. The discovery has information of all the microservices available in the system.
- Configuration server contains all the configurations for the microservices and we will use this server to get configuration information like hostname, url etc. for our microservices.

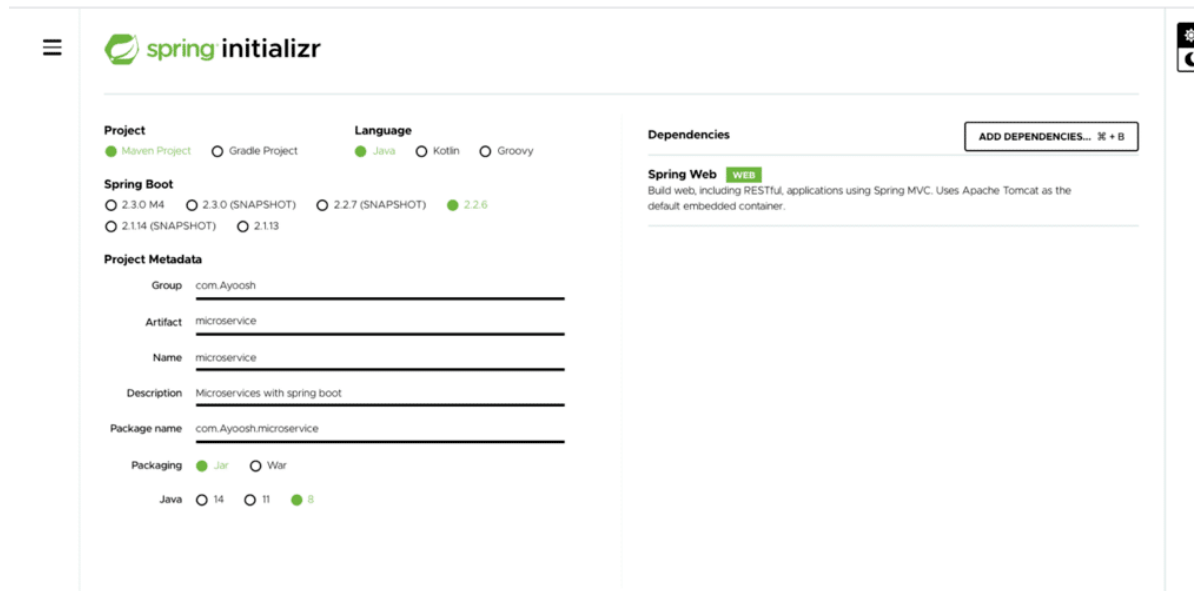


3. Application Setup and Overview.

We will use [Spring Boot](#) to build our microservices example. Spring boot projects can easily be configured using the spring initializer or using [IDE](#) if you like. Will configure the discovery service and config server and one core service in this article. Let's build our application.

3.1. Root project setup.

We will create a root project to do our code and then add other modules like discovery server, config server to our core module. Let's create a spring boot project with spring web dependency.



The image shows the Spring Initializr web application interface. At the top left is the Spring Initializr logo. Below it, the 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.2.6' selected. The 'Project Metadata' section contains fields for Group (com.Ayoosh), Artifact (microservice), Name (microservice), Description (Microservices with spring boot), and Package name (com.Ayoosh.microservice). The 'Packaging' section has 'Jar' selected. The 'Dependencies' section has 'Spring Web' selected. A button 'ADD DEPENDENCIES...' is visible in the top right of the Dependencies section.

Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 2.3.0 M4 ☐ 2.3.0 (SNAPSHOT) ☐ 2.2.7 (SNAPSHOT) ☒ 2.2.6 ☐ 2.1.14 (SNAPSHOT) ☐ 2.1.13

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging

☒ Jar ☐ War

Java

☐ 14 ☐ 11 ☒ 8

Dependencies

Spring Web ☒ WEB





Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

ADD DEPENDENCIES... 35 + 8

After the core module creation, let's create discovery and config server module using the [Spring Initializr](#).

3.2. Discovery Server Setup

For Discovery Server we should configure like this



Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 2.3.0 M4 ☐ 2.3.0 (SNAPSHOT) ☐ 2.2.7 (SNAPSHOT) ☒ 2.2.6 ☐ 2.1.14 (SNAPSHOT) ☐ 2.1.13

Project Metadata

Group

com.ayooosh

Artifact

discoveryserver

Name

discoveryserver

Description

Discovery Server

Package name

com.ayooosh.discoveryserver



Packaging


☒ Jar ☐ War

Java


☐ 14 ☐ 11 ☒ 8

Dependencies


ADD DEPENDENCIES...  

Spring Boot DevTools 

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

Spring Web 

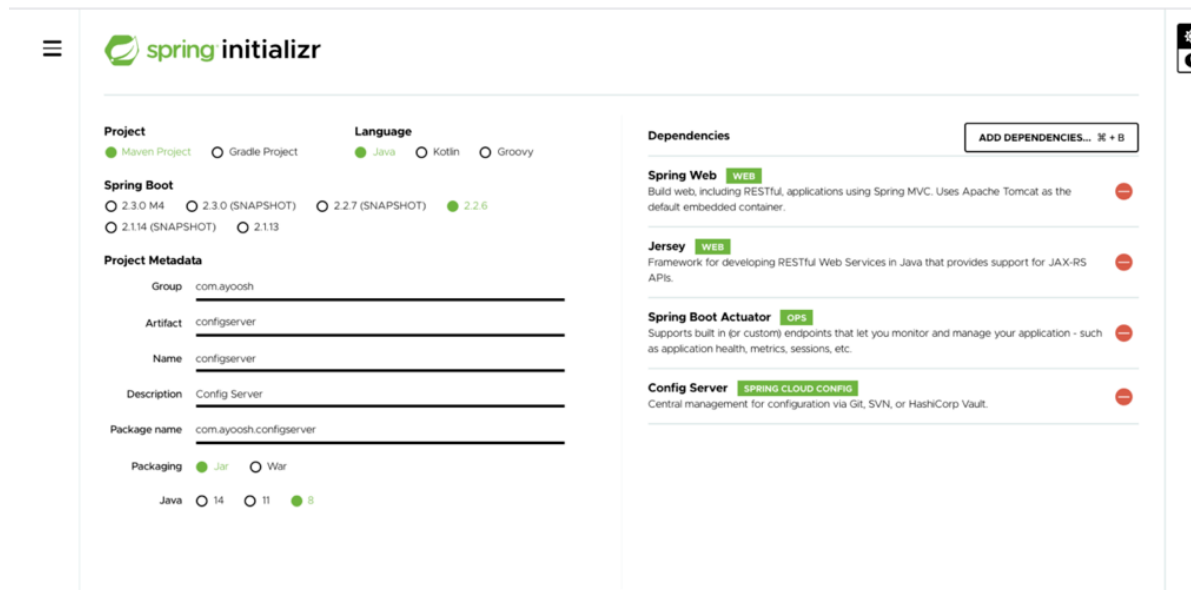
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Eureka Server 

spring-cloud-netflix Eureka Server.

3.3. Config Server configuration

To create a config server, let's create the module with dependencies listed below image.



The image shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', version '2.2.6' is selected. The 'Project Metadata' section contains the following fields: Group (com.ayooosh), Artifact (configserver), Name (configserver), Description (Config Server), and Package name (com.ayooosh.configserver). Under 'Packaging', 'Jar' is selected. Under 'Java', version '8' is selected. On the right, the 'Dependencies' section shows a list of selected dependencies: Spring Web, Jersey, Spring Boot Actuator, and Config Server. Each dependency has a red minus icon to its right. At the top right, there is a button that says 'ADD DEPENDENCIES... 38 + 8'.

4. Core Service Configuration

For core service, we also need database (including spring data JPA and MySQL dependency for our example). Please remember to setup MySQL on your machine before moving to the next step.If you like, you can also use the in memory database for development. Database name should be **profile_management** and other configuration like username password are located in configuration server.

Once the database setup is done, let's create employee_profile table by running the following SQL script.

```
Create Table employee_profile(id int(11) NOT NULL AUTO_INCREMENT,name varchar(255),address
```

```
varchar(255),PRIMARY KEY (`id`));
```

spring initializr

☰

⚙️

🌙

Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 2.3.0 RC1 ☐ 2.3.0 (SNAPSHOT) ☐ 2.2.7 (SNAPSHOT) ☒ 2.2.6

☐ 2.1.14 (SNAPSHOT) ☐ 2.1.13

Project Metadata

Group

com.aycoosh

Artifact

microservices

Name

microservices

Description

Demo project for Spring Boot

Package name

com.aycoosh.microservices

Packaging

☒ Jar ☐ War

Java

☐ 14 ☐ 11 ☒ 8

Dependencies

ADD DEPENDENCIES... 0 + 0

Spring Data JPA SOL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Eureka Discovery Client SPRING CLOUD DISCOVERY

a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

Config Client SPRING CLOUD CONFIG

Client that connects to a Spring Cloud Config Server to fetch the application's configuration.

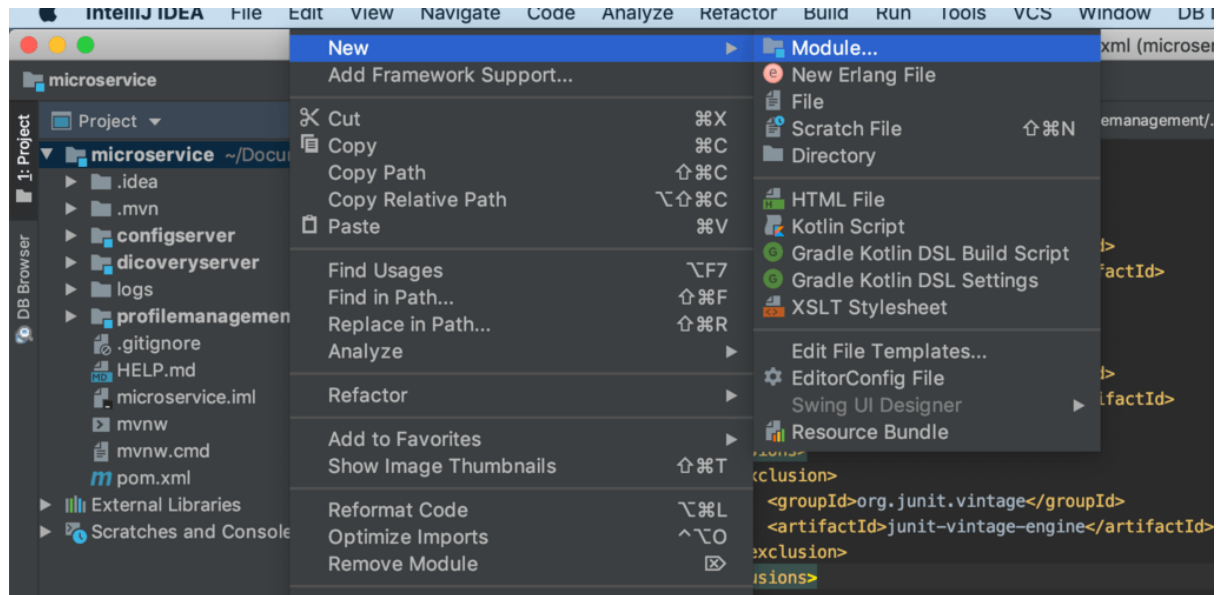
Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

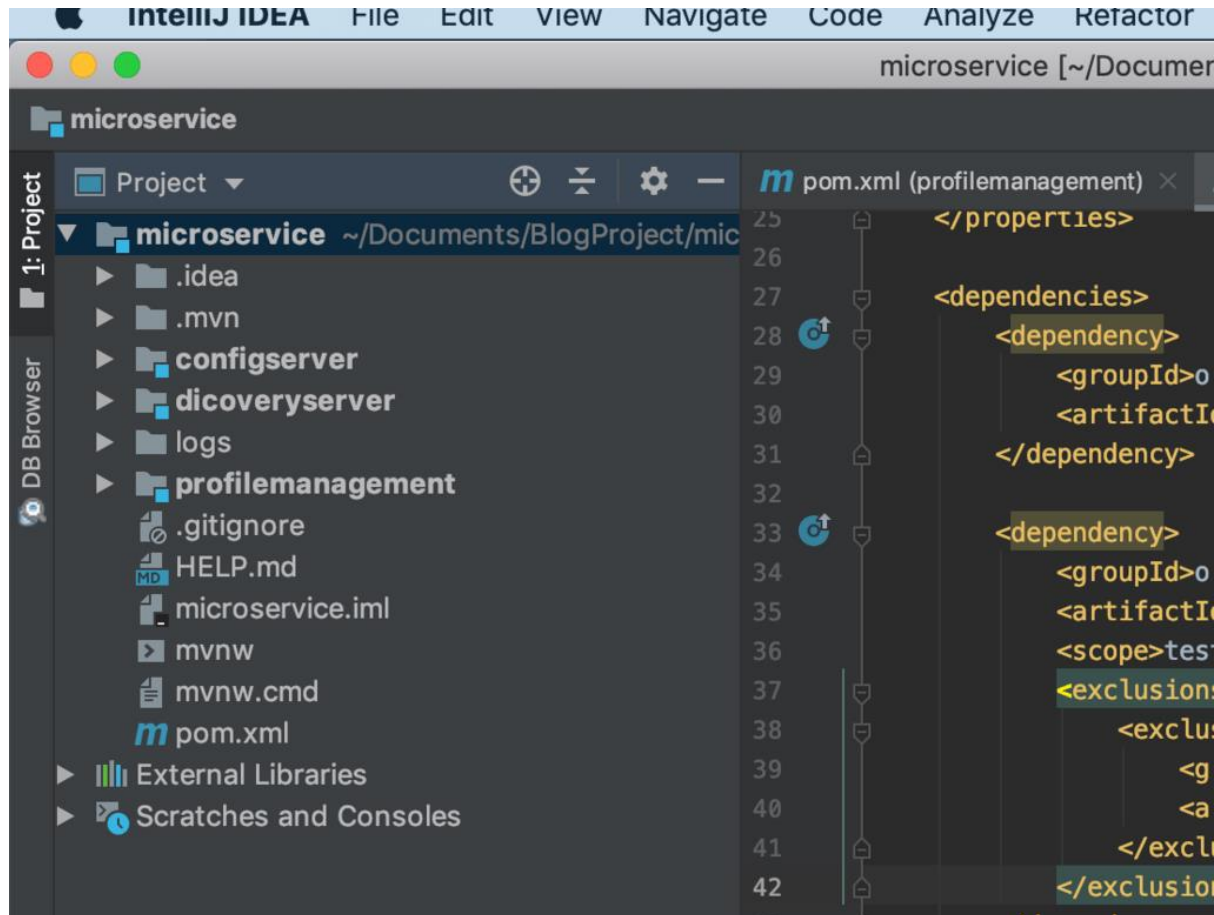
MySQL Driver SOL

MySQL JDBC and R2DBC driver.

After creation all projects, open the IntelliJ Idea or other IDE. Open our root project and then need to create a new module for all services. For IntelliJ Idea, it will be like this



This is how the final project layout might look like:



Our root project `pom.xml` will be like this

```
<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>
```

```
  <packaging>pom</packaging>
```

```
  <modules>
```

```
    <module>dicoveryservice</module>
```

```
    <module>configserver</module>
```

```
    <module>profilemanagement</module>
```

```
  </modules>
```

```
  <parent>
```

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

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

```
<version>2.2.6.RELEASE</version>
```

```
<relativePath />
```

```
<!-- lookup parent from repository -->
```

```
</parent>
```

```
<groupId>com.javadevjournal</groupId>
```

```
<artifactId>microservice</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
```

```
<name>microservice</name>
```

```
<description>Microservices with spring boot</description>
```

```
<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-test</artifactId>
```

```
<scope>test</scope>
```

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

```
</plugin>
```

```
</plugins>
```

```
</build>
```

```
</project>
```

5. Service discovery with Eureka servers.

The Discovery server used to register all services after they go to live. Then it will be easy to find out the service from the registry. When a service registers with eureka it will provide the metadata like host, port and health indicator parameters to the client to connect.

We already created a eureka service from the spring-boot initializer and included our root project.

```
<?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">
```

```
<parent>
```

```
<artifactId>microservice</artifactId>
```

```
<groupId>com.javadevjournal</groupId>
```

```
<version>0.0.1-SNAPSHOT</version>
```

```
</parent>
```

```
<modelVersion>4.0.0</modelVersion>
```

```
<artifactId>dicoveryservice</artifactId>
```

```
<properties>
```

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

```
<spring-cloud.version>Hoxton.SR3</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-server</artifactId>
```

```
</dependency>
```

`<dependency>`

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

`<artifactId>spring-boot-devtools</artifactId>`

`<scope>runtime</scope>`

`<optional>true</optional>`

`</dependency>`

`<dependency>`

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

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

`<scope>test</scope>`

`<exclusions>`

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</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>
```

```
</project>
```

We need to add annotation **@EnableEurekaServer** to the application main class. This annotation will enable the services registration with the eureka server.

```
package com.javadevjournel.discoveryserver;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

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

```
@SpringBootApplication
```

```
@EnableEurekaServer
```

```
public class DiscoveryserverApplication {
```

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

```
        SpringApplication.run(DiscoveryserverApplication.class, args);
```

```
    }
```

```
}
```

Now we need to define some properties to **application.properties** file.

```
spring.application.name=discoveryserver
```

```
server.port=8081
```

```
eureka.client.register-with-eureka=false
```

```
eureka.client.fetch-registry=false
```

6. Configuration Server

The purpose of the configuration server is to use a centralized configuration directory for all servers. Config server can fetch configuration data from external centralized location. As GitHub provides file storage facility to us, we will use GitHub for our configuration file location. After including the configuration project to our root project, we will see the `pom.xml` file like this.

```
<?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">
```

```
<parent>
```

```
<artifactId>microservice</artifactId>
```

```
<groupId>com.javadevjournals</groupId>
```

```
<version>0.0.1-SNAPSHOT</version>
```

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>configserver</artifactId>

<properties>

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

<spring-cloud.version>Hoxton.SR3</spring-cloud.version>

</properties>

<dependencies>

<dependency>

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

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

```
</dependency>
```

```
<dependency>
```

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

```
<artifactId>spring-boot-starter-jersey</artifactId>
```

```
</dependency>
```

```
<dependency>
```

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

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

```
</dependency>
```

```
<dependency>
```

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

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

```
</dependency>
```

```
<dependency>
```

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

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

```
<scope>test</scope>
```

```
<exclusions>
```

```
<exclusion>
```

```
<groupId>org.junit.vintage</groupId>
```

```
<artifactId>junit-vintage-engine</artifactId>
```

```
</exclusion>
```


`</exclusions>`

`</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>`

```
</project>
```

And the Main Application class will be like this

```
package com.javadevjournal.configserver;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

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

```
@SpringBootApplication
```

```
@EnableConfigServer
```

```
public class ConfigserverApplication {
```

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

```
        SpringApplication.run(ConfigserverApplication.class, args);
```

```
}
```

```
}
```

After adding the annotation to the main class, we will look at the properties file for configuration properties.

```
server.port=8082
```

```
spring.application.name=configserver
```

```
spring.cloud.config.server.git.uri = https://github.com/flopocoder82/microservices
```

```
spring.cloud.config.server.git.username=XXXXXX
```

```
spring.cloud.config.server.git.password=XXXXXXXX
```

```
spring.cloud.config.server.git.clone-on-start=true
```

7. MicroServices (Profile Management)

Now need to configure core service. Core service will contain business module. Pom.xml file should be like below code snapshot.

```
<?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">
```

```
<parent>
```

```
<artifactId>microservice</artifactId>
```

```
<groupId>com.javadevjournal</groupId>
```

```
<version>0.0.1-SNAPSHOT</version>
```

```
</parent>
```

```
<modelVersion>4.0.0</modelVersion>
```

```
<artifactId>profilemanagement</artifactId>
```

```
<properties>
```

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

```
<spring-cloud.version>Hoxton.SR3</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-config</artifactId>
```

```
</dependency>
```

```
<dependency>
```

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

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

```
</dependency>
```

```
<dependency>
```

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

```
<artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
</dependency>
```

```
<dependency>
```

```
<groupId>mysql</groupId>
```

```
<artifactId>mysql-connector-java</artifactId>
```

```
<scope>runtime</scope>
```

```
</dependency>
```

```
<dependency>
```

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

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

```
<scope>test</scope>
```

```
<exclusions>
```

```
<exclusion>
```

```
<groupId>org.junit.vintage</groupId>
```

```
<artifactId>junit-vintage-engine</artifactId>
```


`</exclusion>`

`</exclusions>`

`</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>
```

```
</project>
```

We will simply see an example of an employee profile. For this reason, we have to create a service, controller domain class. At first, have to configure our main application class as below. We will enable eureka and config the client by annotation.

```
package com.javadevjournal.profilemanagement;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

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

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

```
@SpringBootApplication
```

```
@EnableEurekaClient
```

```
@EnableDiscoveryClient
```

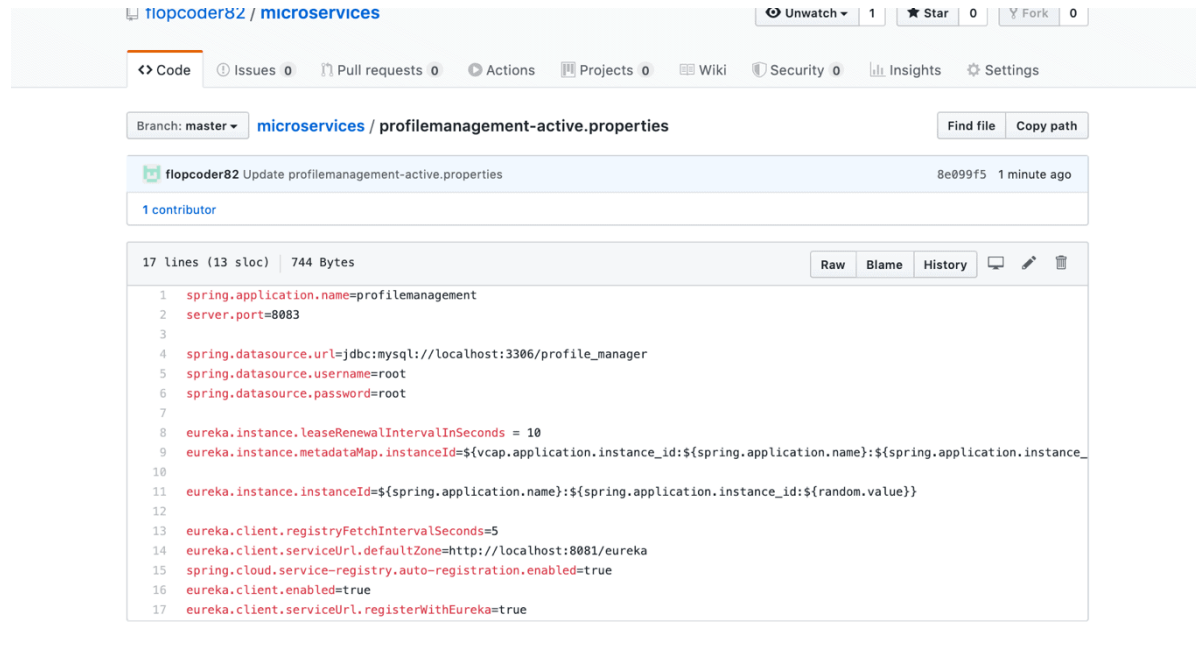
```
public class ProfilemanagementApplication {
```

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

```
        SpringApplication.run(ProfilemanagementApplication.class, args);
```

```
    }<code>
```

We will create a `bootstrap.properties` file to our resource folder of the application. Because we will use the config server to get the properties file. When we use this properties file, during application startup it will first fetch the properties from **GitHub** through the config server. Yes, we will also create a git repository to the GitHub named by **microservice**. Then we should add a properties file like named by `profilemanagement-active.properties`. Github repository will look like the below image. We can check this from [here](#) also.



```
1 spring.application.name=profilemanagement
2 server.port=8083
3
4 spring.datasource.url=jdbc:mysql://localhost:3306/profile_manager
5 spring.datasource.username=root
6 spring.datasource.password=root
7
8 eureka.instance.leaseRenewalIntervalInSeconds = 10
9 eureka.instance.metadataMap.instanceId=${vcap.application.instance_id:${spring.application.name}:${spring.application.instance_id:${random.value}}}
10
11 eureka.instance.instanceId=${spring.application.name}:${spring.application.instance_id:${random.value}}
12
13 eureka.client.registryFetchIntervalSeconds=5
14 eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka
15 spring.cloud.service-registry.auto-registration.enabled=true
16 eureka.client.enabled=true
17 eureka.client.serviceUrl.registerWithEureka=true
```

Our **bootstrap.properties** file will be like this

```
spring.cloud.config.uri=http://localhost:8082
```

```
spring.cloud.config.name=profilemanagement
```

```
spring.cloud.config.profile=active
```

and `application.properties` file should be like this `spring.application.name=profilemanagement`

Then we will create following classes to our project.

```
package com.javadevjournal.profilemanagement.domain;
```

```
import javax.persistence.*;
```

```
@Entity
```

```
@Table(name = "employee_profile")
```

```
public class EmployeeProfile {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "id")
```

```
    private int id;
```

```
    @Column(name = "name")
```

```
private String name;
```

```
@Column(name = "address")
```

```
private String address;
```

```
public int getId() {
```

```
    return id;
```

```
}
```

```
public void setId(int id) {
```

```
    this.id = id;
```

```
}
```

```
public String getName() {
```

```
return name;
```

```
}
```

```
public void setName(String name) {
```

```
this.name = name;
```

```
}
```

```
public String getAddress() {
```

```
return address;
```

```
}
```

```
public void setAddress(String address) {
```

```
this.address = address;
```



```
}
```

```
}
```

7.1. Repository

```
package com.javadevjournal.profilemanagement.dao;
```

```
import com.ayoosh.profilemanagement.domain.EmployeeProfile;
```

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

```
import org.springframework.stereotype.Repository;
```

```
@Repository
```

```
public interface ProfileRepository extends JpaRepository < EmployeeProfile, Integer > {
```

```
}
```

Services

```
package com.javadevjournal.profilemanagement.service;
```

```
import com.ayoosh.profilemanagement.domain.EmployeeProfile;
```

```
import java.util.List;
```

```
public interface EmployeeProfileService {
```

```
    void addEmployeeProfile(EmployeeProfile profile);
```

```
    List < EmployeeProfile > getEmployeeProfiles();
```

```
}
```

7.2. Service Implementation

```
package com.javadevjournal.profilemanagement.service;
```

```
import com.ayoosh.profilemanagement.dao.ProfileRepository;
```

```
import com.ayoosh.profilemanagement.domain.EmployeeProfile;
```

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

```
import org.springframework.stereotype.Service;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
@Service
```

```
public class EmployeeProfileServiceImpl implements EmployeeProfileService {
```

```
@Autowired
```

```
ProfileRepository repository;
```

```
List < EmployeeProfile > employeeProfileList = new ArrayList < > ();
```

```
@Override
```

```
public void addEmployeeProfile(EmployeeProfile profile) {
```

```
    repository.save(profile);
```

```
}
```

```
@Override
```

```
public List < EmployeeProfile > getEmployeeProfiles() {
```

```
    return repository.findAll();
```

```
}
```

```
}
```

7.3. Controller

```
package com.ayoosh.profilemanagement.controller;
```

```
import com.ayoosh.profilemanagement.domain.EmployeeProfile;
```

```
import com.ayoosh.profilemanagement.service.EmployeeProfileService;
```

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

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

```
import java.util.List;
```

```
@RestController
```

```
@RequestMapping(value = "/")
```

```
public class EmployeeController {
```

```
    @Autowired
```

```
    EmployeeProfileService employeeProfileService;
```

```
@PostMapping
```

```
public void saveEmployeeProfile(@RequestBody EmployeeProfile employeeProfile) {
```

```
    employeeProfileService.addEmployeeProfile(employeeProfile);
```

```
}
```

```
@GetMapping
```

```
public List < EmployeeProfile > getAllEmployee() {
```

```
    return employeeProfileService.getEmployeeProfiles();
```

```
}
```

```
}
```

8. Running the example and display the results.

Our configuration to build microservices with Spring Boot is complete. We are ready to run our application . Till now we have configured the following things.

1. Eureka server
2. Configuration server
3. Core service

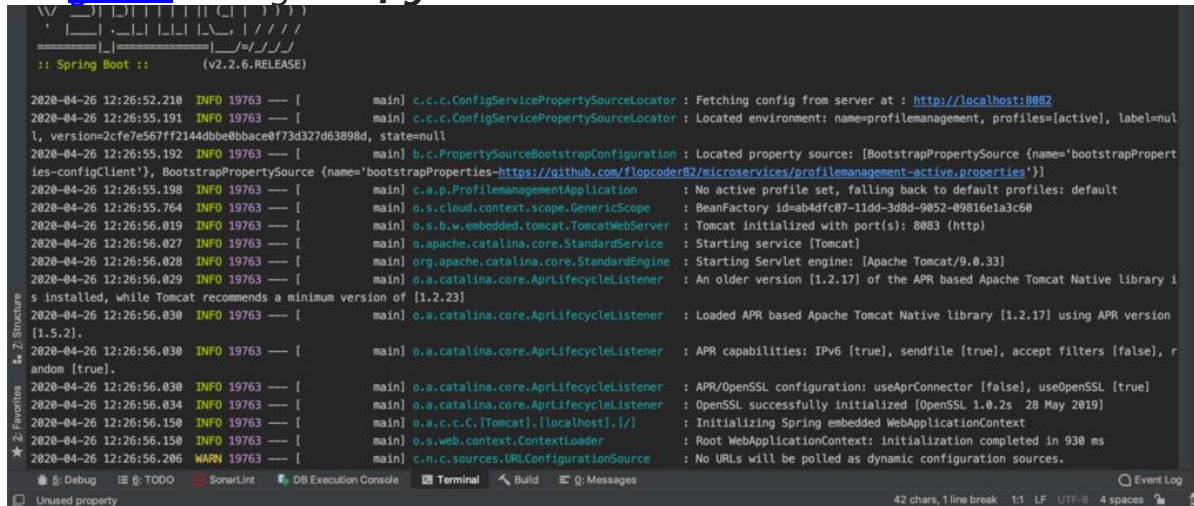
Now we will go to the **eureka-server** directory from the terminal and will run **mvn spring-boot:run**. As we configured the eureka server to port 8081, it will listen to 8081 port. Then we will go to the config server and run **mvn spring-boot:run** command. As we configured the config server at port 8082, it will listen to port 8082.

Now we will run our core service **profilemanagement**. Go to the directory of **profilemanagement** project and execute **mvn spring-boot:run** command. It will listen to port 8083. Now all server is running. So, we can test now. At first, we will check the eureka server at <http://localhost:8081/>. It will be like the below image. Here we can see that **PROFILEMANAGEMENT** project is registered here.

The screenshot displays the Spring Eureka web interface. At the top, there's a navigation bar with the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

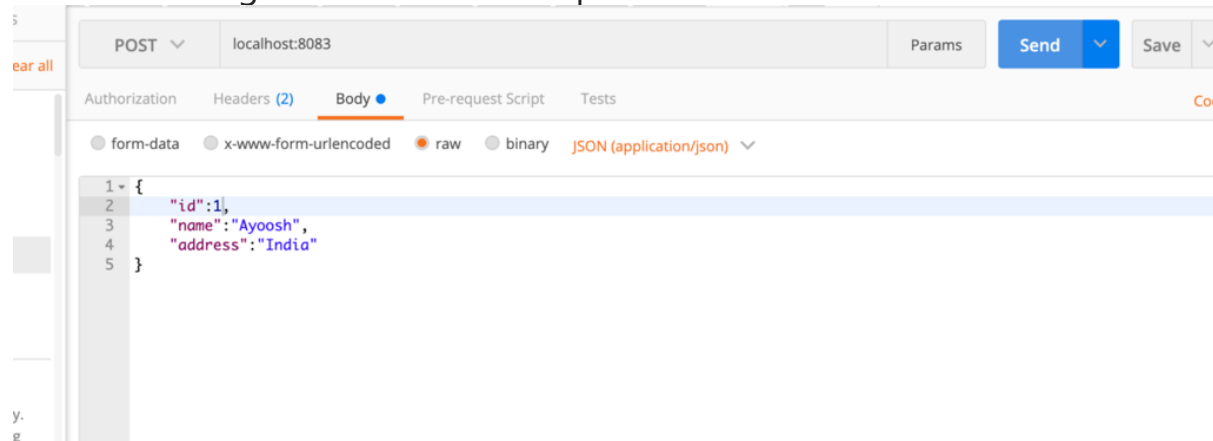
- System Status:** A table showing environment details (test, default) and system metrics (Current time: 2020-04-26T12:47:05 +0600, Uptime: 00:29, Lease expiration enabled: true, Renewal threshold: 3, Renewal (last min): 6).
- DS Replicas:** A section showing the local host 'localhost' as a data source replica.
- Instances currently registered with Eureka:** A table listing the 'PROFILEMANAGEMENT' application, which is currently up and running. It shows 1 instance in the 'n/a (1)' availability zone.
- General Info:** A table providing system metrics such as total available memory (495mb), environment (test), number of CPUs (4), current memory usage (394mb, 79%), server uptime (00:29), and registered/unavailable/available replicas.
- Instance Info:** A section for detailed information about the current instance.

And we will check the log of the core server. We will see that it will take the properties file from the [github](#) through **configserver**.

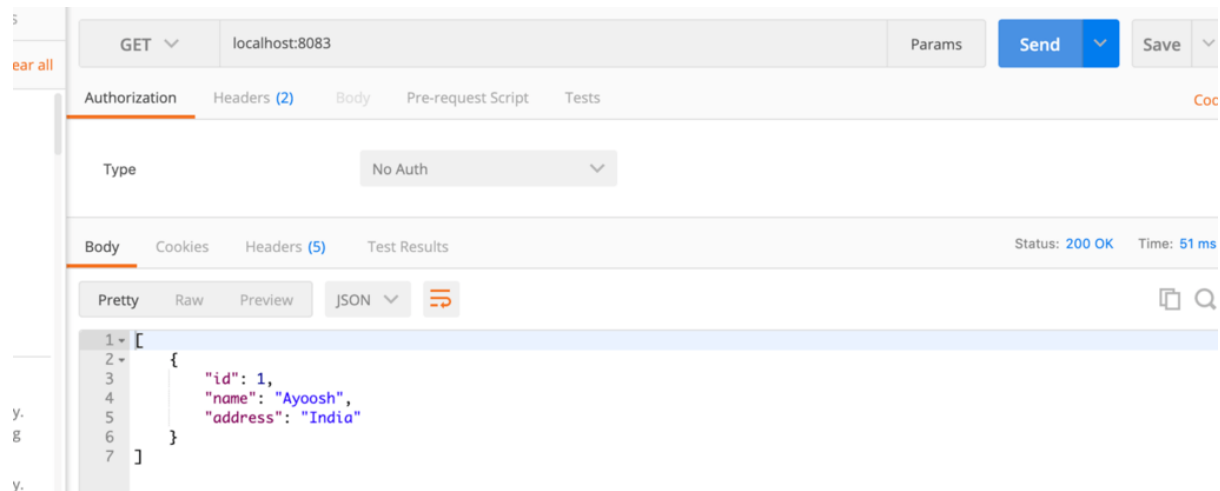


```
2020-04-26 12:26:52.218 INFO 19763 --- [main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8082
2020-04-26 12:26:55.191 INFO 19763 --- [main] c.c.c.ConfigServicePropertySourceLocator : Located environment: name=profilemanagement, profiles=[active], label=null, version=2cfe7e567ff2144dbbe8bbace0f73d327d63898d, state=null
2020-04-26 12:26:55.192 INFO 19763 --- [main] b.c.PropertySourceBootstrapConfiguration : Located property source: [BootstrapPropertySource {name='bootstrapProperties-configClient'}, BootstrapPropertySource {name='bootstrapProperties-https://github.com/flopcoder82/microservices/profilemanagement-active.properties'}]
2020-04-26 12:26:55.198 INFO 19763 --- [main] c.a.p.ProfilemanagementApplication : No active profile set, falling back to default profiles: default
2020-04-26 12:26:55.764 INFO 19763 --- [main] o.s.c.cloud.context.scope.GenericScope : BeanFactory id=ab4dfc07-11dd-3d8d-9052-09816e1a3c60
2020-04-26 12:26:56.019 INFO 19763 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8083 (http)
2020-04-26 12:26:56.027 INFO 19763 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-04-26 12:26:56.028 INFO 19763 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.33]
2020-04-26 12:26:56.029 INFO 19763 --- [main] o.a.catalina.core.AprLifecycleListener : An older version [1.2.17] of the APR based Apache Tomcat Native library is installed, while Tomcat recommends a minimum version of [1.2.23]
2020-04-26 12:26:56.030 INFO 19763 --- [main] o.a.catalina.core.AprLifecycleListener : Loaded APR based Apache Tomcat Native library [1.2.17] using APR version [1.5.2].
2020-04-26 12:26:56.030 INFO 19763 --- [main] o.a.catalina.core.AprLifecycleListener : APR capabilities: IPv6 [true], sendfile [true], accept filters [false], random [true].
2020-04-26 12:26:56.034 INFO 19763 --- [main] o.a.catalina.core.AprLifecycleListener : APR/OpenSSL configuration: useAprConnector [false], useOpenSSL [true]
2020-04-26 12:26:56.034 INFO 19763 --- [main] o.a.catalina.core.AprLifecycleListener : OpenSSL successfully initialized [OpenSSL 1.0.2s 28 May 2019]
2020-04-26 12:26:56.150 INFO 19763 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-04-26 12:26:56.150 INFO 19763 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 930 ms
2020-04-26 12:26:56.206 WARN 19763 --- [main] c.n.c.sources.URLConfigurationSource : No URLs will be polled as dynamic configuration sources.
```

Now we will check **profilemanagement** service through postman. First, we will create a profile through postman like the below image. This is a **POST** request.



And then, we will fetch the profiles through **GET** request



We can see that its working fine. With POST request its saving data to a list and with GET request its fetching data from the server.