Spring Boot & Microservices

Pre-requisites

1. Java
2. Spring Framework & Rest API’s
3. Git

Contents Overview

Spring Boot

* Autoconfigurations
* Webservices
* Adding Data JPA to interact with DB
* Configuring different servers like jetty, undertow
* Running the applications using jars

Spring Microservices

* Overview Microservices
* Monolithic vs Microservice architecture
* Design patterns used for microservices
* Spring projects for microservices - Spring Boot & Spring Cloud
* Service Discovery
* Discovery Client
* Load Balancer
* Circuit Breaker with Resilience4J
* Distributed Configuration
* API gateway
* Securing Microservices
* Deploying microservices using Docker & AWS

Spring Boot:

* It simplifies development by reducing all the generic configurations you do in the application like XML configurations, Server Configuration, Front Controller Configuration, Component Scanning, Bean Dependency configurations,
* It doesn’t need any XML
* It uses a plain text property file for any application related configurations which are much easier to maintain
* It provides starter projects to automatically setup the project/environment
* Spring Boot Starter Web: This starter helps spring boot to setup necessary features for developing web applications like Server, Front Controller, Component Scanning
* Spring Boot Starter Data JPA: This starter helps spring boot setup necessary features to interact with DB, like establishing connection, supplying the datasource dependencies to other beans which does CRUD operations
* Spring Boot Starter Actuator: This starter helps spring boot to provide endpoints to check the application status, metrics, health
* Spring Boot provides a Starter Parent project to avoid versioning conflicts between the spring libraries

Spring Provides Spring Initializr for developers to quickly create spring boot projects or they can use STS IDE or STS plugin in Eclipse to use spring initializr feature

<https://start.spring.io/>

The above website allows developers to quickly create a ready to run project

Some of the dependencies

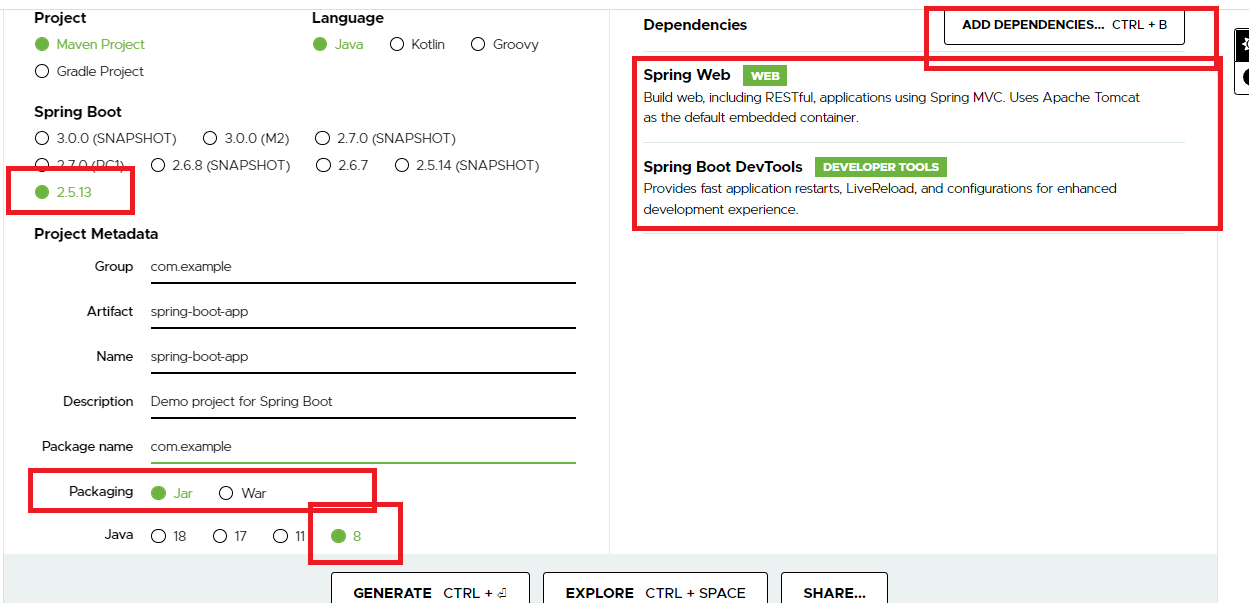
Web: This is used to develop web applications, it is a spring boot starter web library takes care of all the set up required for developing web applications

Devtools: This reloads your application to detect the changes while you are coding

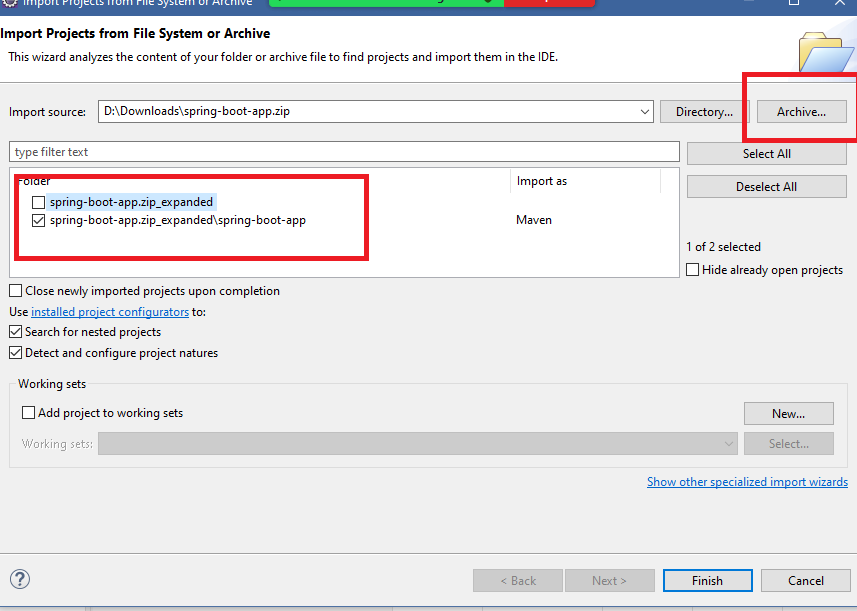
Embedded Servers: These are embedded in the project, by default spring boot provides Tomcat, but it supports two more servers as embedded server like Jetty, Undertow

Note: When you build your application for production based on the packaging like jar or war the build file is created, if its jar packaging then jar will be created, to run this you just need JDK, because you can run using java -jar command, in case its war then a war file will be built and it must deployed in the external server

Creating our first project

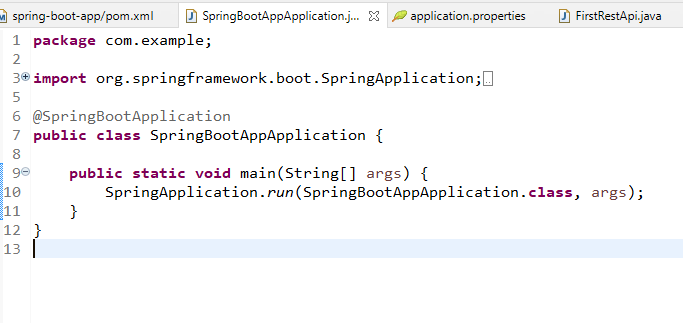


After downloading Open the zip file from eclipse



Spring Initializr provides an entrypoint to launch the spring boot application

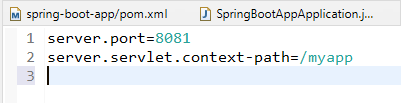
SpringBootAppApplication.java



@SpringBootApplication: It is a predefined annotations which does auto-configuration based on the libraries present in the classpath, it does component scan from the package it is present

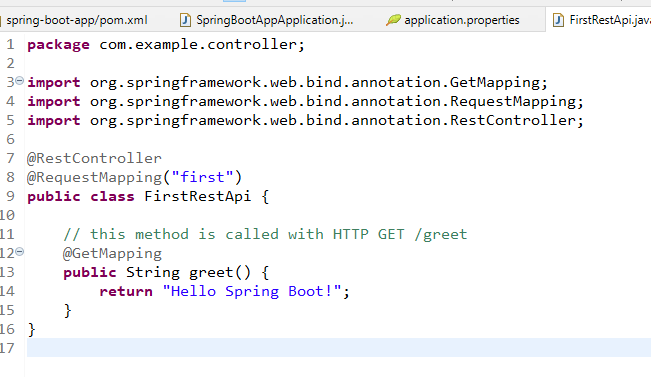
Note: Create all your Spring component classes or other stereo type classes like @RestController, @Service, @Configuration, @Component, @Aspect and so on inside the package or subpackage where @SpringBootApplication class exists.

application.properties

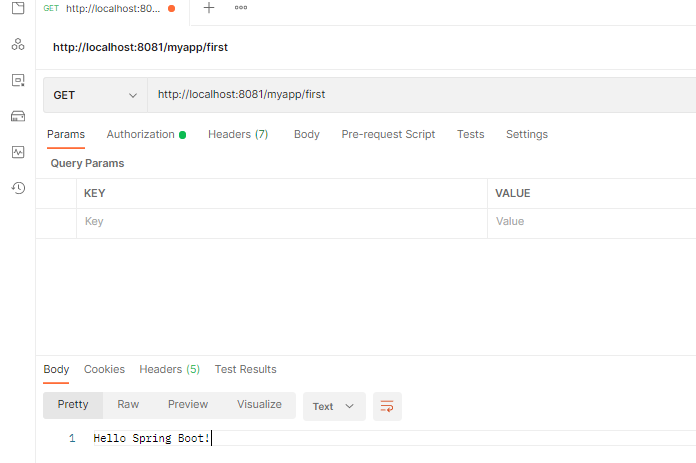


This property file is a default file the spring boot loads, it reads the properties and sets up the configuration accordingly

FirstRest.java



Output:

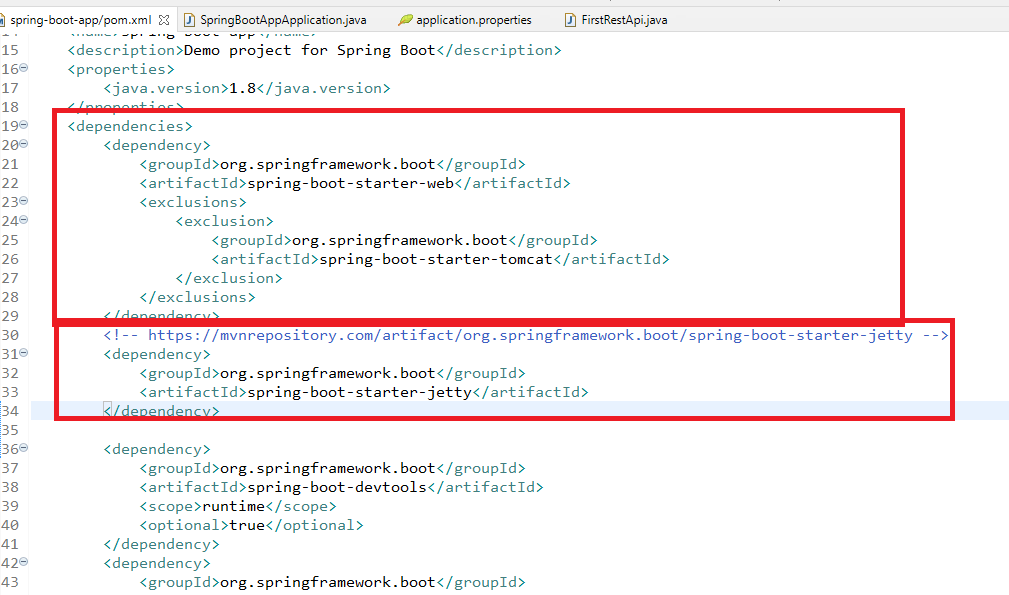


Here the default embedded server is Tomcat, but spring boot supports other embedded servers like Jetty, Undertow

Configuring Jetty Server

* You need to add spring boot starter jetty library in the pom.xml
* You need to exclude spring boot tomcat server in the pom.xml

pom.xml

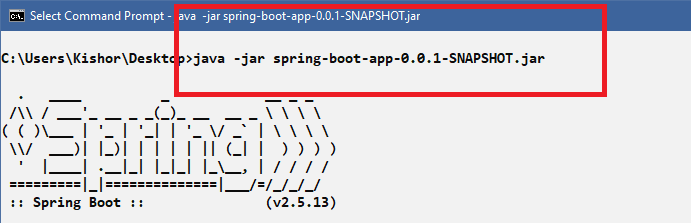


Note: Stop & Restart the application, you can see in the console which server started

Note: The spring boot applications are ready to run application even in production environment, for that you can build the application

Since we have Maven, we can use its goal: package to build the application, it is jar package, hence we get jar file.

* To run the application you can use jar -jar file-name.jar

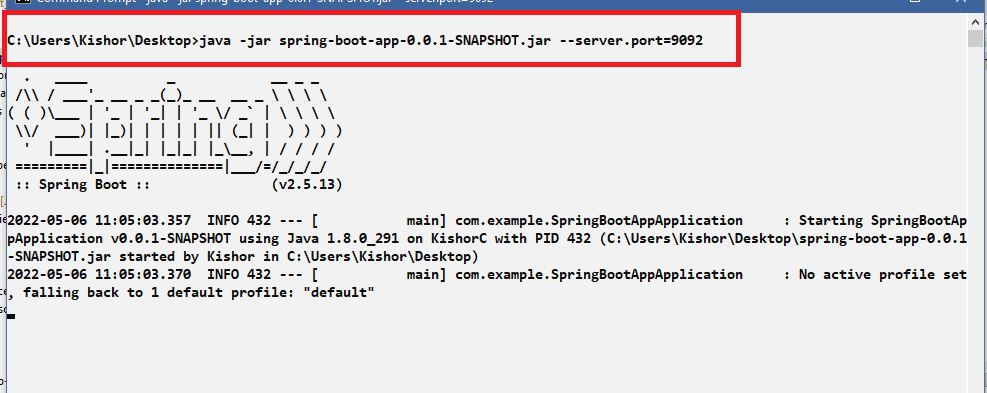


Note: You can assume this as a cloud machine and some unix based machine terminal, they must have java installed and the jar which is built, if they have the Java & Jar file then they can run it

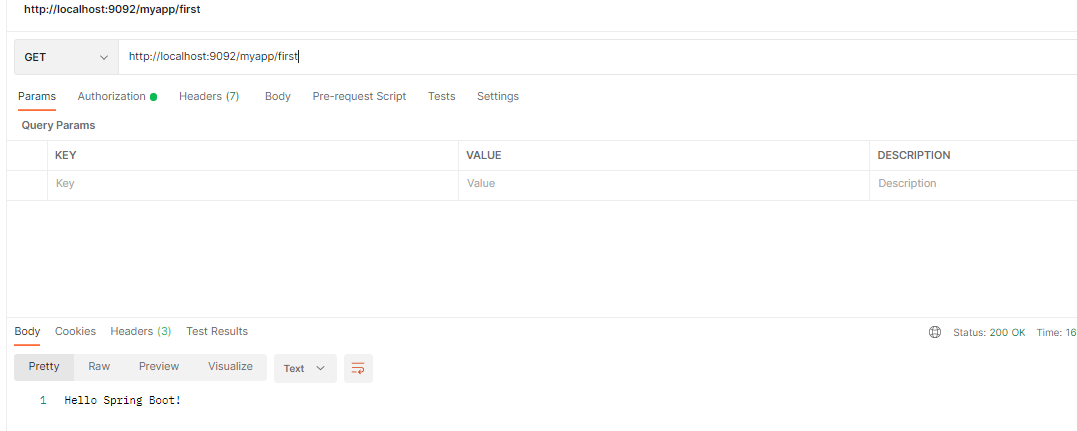
Overriding the properties

Since the application.properties is using server.port = 8081, you can override and give different port

i.e.java -jar file-name.jar --server.port=9092



Output:

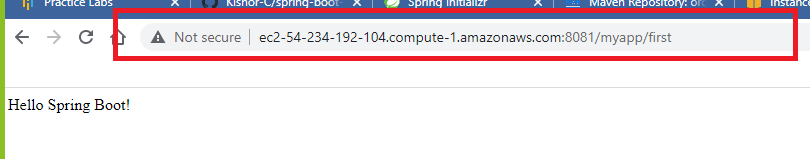


You can deploy this in AWS/Azure clouds, there you need a machine with Java installed

Things to do for launching our jar in the AWS machines

* You need to create an EC2 instance
* You need to connect to the EC2 instance either from GIT bash or putty
* You need to install git & java
* You need to get the build file i.e., jar from GIT in EC2 instance
* You need to run the jar in the EC2 instance
* Open TCP port that can accept incoming request for you app

Once you run your application in EC2 machine you must able to access the application through the port



Above application is accessed via AWS-EC2 machine URL, but to run that application we need follow all the necessary steps like

* Using Git
* Opening TCP port in EC2 machine
* Installing java in ec2 machine

We can interact with any databases through spring boot

* H2 (in memory database)
* MySQL / Derby / Oracle and so

You have a starter libraries to interact with the database which is spring boot starter jpa

Benefits you get from the spring boot starter data jpa

* It establishes connection with the database looking at properties in the application.properties
* It injects all the dependencies required to perform Database operations
* It provides proxy implementation for the database logics with the help of Repository interfaces
* It gives two repository interfaces like CrudRepository<T, ID> & JpaRepository<T, ID>
* You need to create an entity class that can be used with the above repositories and you need to create an interface that can extend the above repositories
* Spring Boot data jpa implements the interface based on the entities you have used in the repository
* You don’t have to write database logics at all
* You can inject the proxy implementation in the service layer using the interface you have created

Assume you have an entity mapped to employee table

@Entity  
class Employee { … } // mapped to employee table

You can create an interface to work with employee table as below:-

interface EmployeeDao extends JpaRepository<Employee, Integer> {   
}

(or)  
interface EmployeeDao extends CrudRepository<Employee, Integer> {   
  
}

(or)

interface EmployeeDao extends CrudRepository<Employee, Integer> {   
 @Query(“select e from Employee e where e.salary = ?1”)  
 public List<Employee> getEmployeesBasedOnSalary(double salary);  
}

Here the spring boot data jpa implements the method getEmployeesBasedOnSalary() with the query mentioned.

You can inject the proxy implementation using the EmployeeDao interface via @Autowired

@Service  
public ServiceImpl {   
 @Autowired  
 private EmployeeDao dao;  
}

From EmployeeDao you can access all the methods of JpaRepository/CrudRepository all the methods interact with the table the entity is mapped with.

CrudRepository<T, ID> has following methods

* save(T)
* deleteById(ID)
* findAll()
* findById(ID)

JpaRepository<T, ID> has following methods

* All the methods of CRUD
* sort()

All the above methods perform the operations depending on the entity class

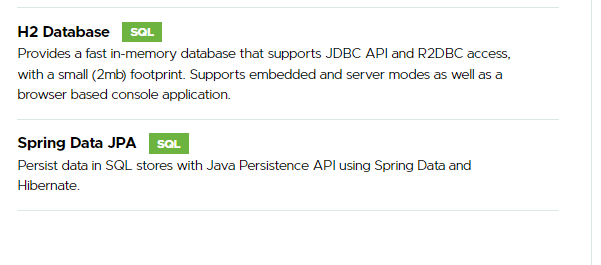
Now we can interact with any databases using the Repository interface

* You can use H2 database which is a in-memory database

JpaRepository & CrudRepository: They work with SQL Databases

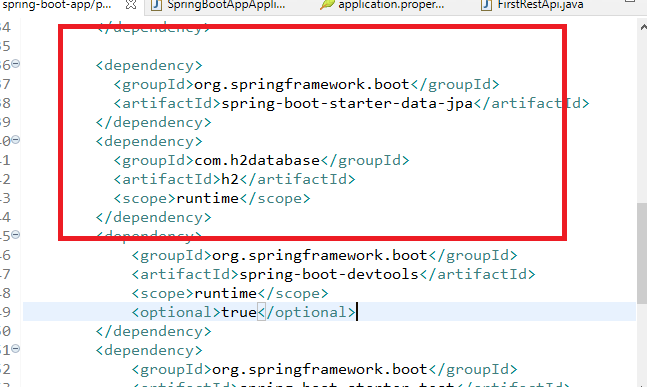
MongoRepository: This interface helps to work with MongoDB which is a NoSQL Database

Dependencies required to work with H2 database



You can copy these dependencies entry to your project

pom.xml

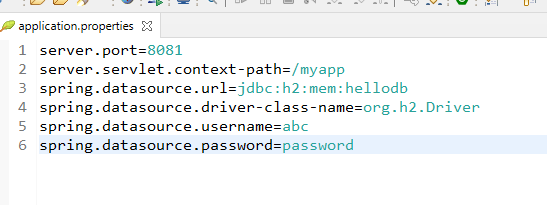


Note: You don’t have to create any table, you can use entity class

Things to create in the application

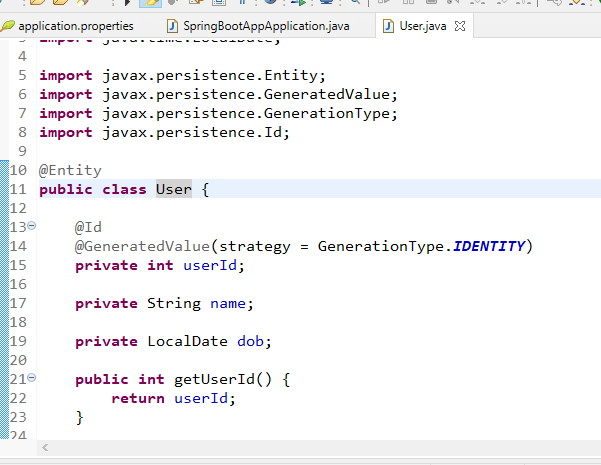
1. Entity class
2. Database configurations in application.properties
3. Interface that extends Repository
4. Injecting the Repositories in the Service
5. Injecting the Service in the controller
6. Create web services that does CRUD operations with appropriate HTTP methods

application.properities



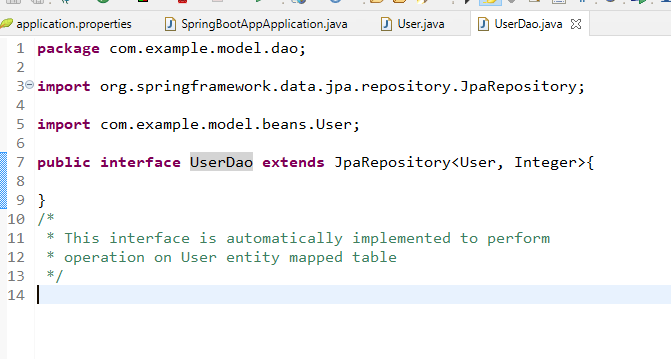
Entity class

User.java



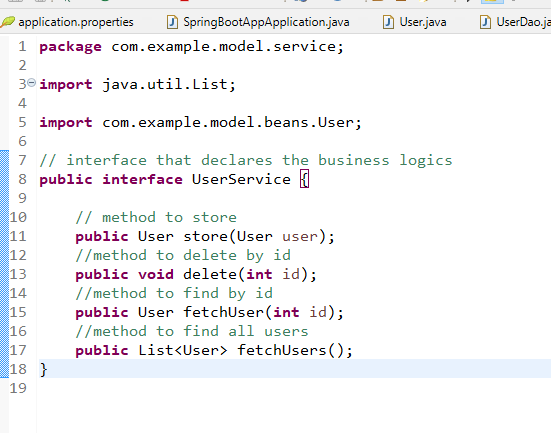
Since we are using H2 we don’t need any physical database

UserDao.java



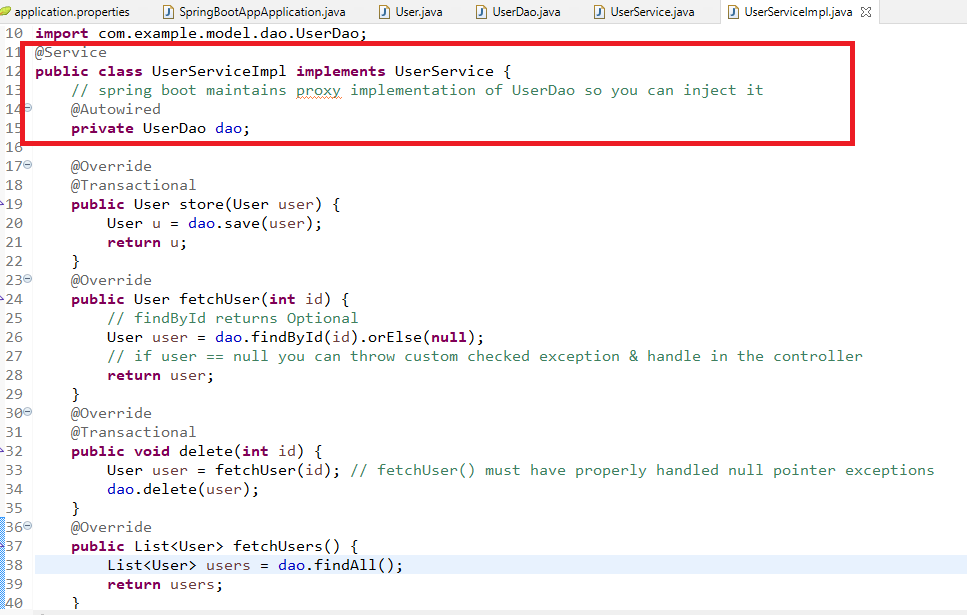
You can create a Service interface & Implement it

UserService.java

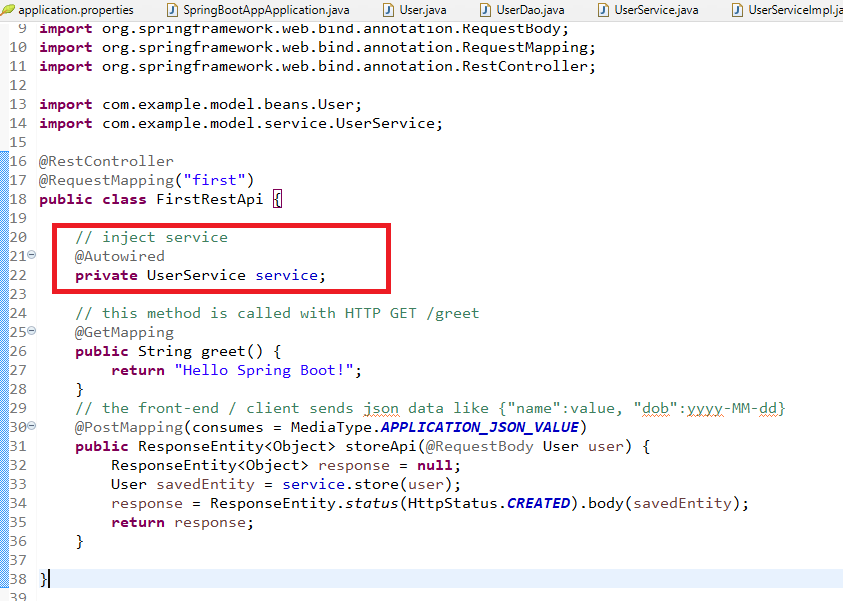


You need to implement this interface to call the repository methods & save() , delete() must use @Transactional as they modify the entity

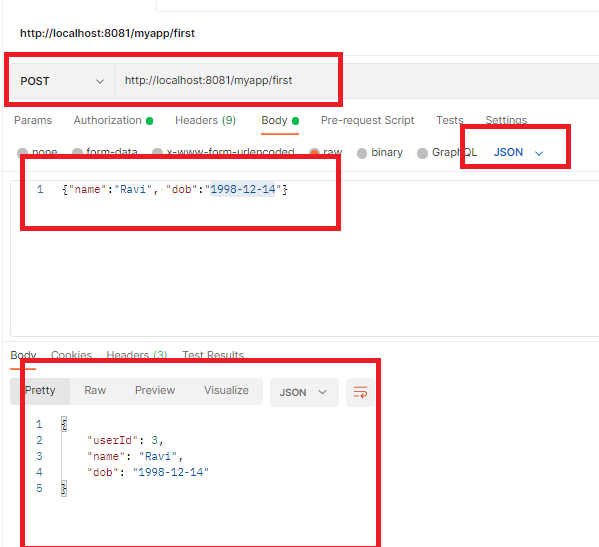
UserServiceImpl.java



Now you can create webservices to perform CRUD operations



This webservice has currently storing the user, you can pass user data from postman



Representing the data in XML format

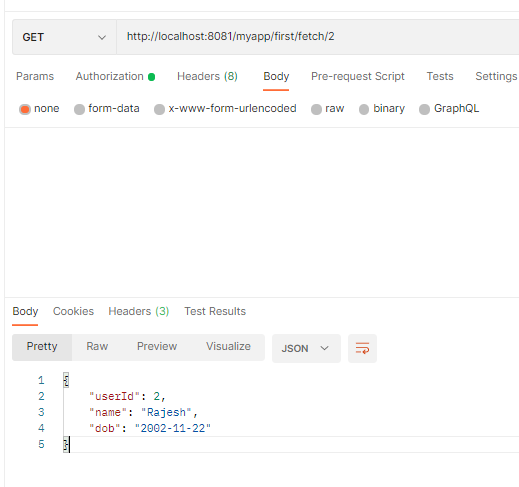
You need to use @XmlRootEelement on top of the bean, this specifies the spring boot that java bean can be converted to XML format and vice versa



Webservices for fetch by id and fetch all users



Output:



Http methods follows the standard for the particular operations

1. GET: It is for fetch operations
2. POST: It is for creating new resources
3. DELETE: It is for deleting the resources
4. PUT: It is for updating the existing resources

According to the Http methods spring boot has annotations

@GetMapping, @PostMapping, @DeleteMapping, @PutMapping

Use these annotations based on the operation you do

Activity:

1. Try all the above examples
2. Perform other operations on the User entity & use the appropriate mapping
   1. Deleting the user by id: Hint: *DELETE /100*
   2. Update the DOB using the id: Hint *PUT /100/2000-10-25*
3. For the same example create a Checked Exception which helps to avoid null pointer exception when the user id is not found in the database

Suppose user id 100 is not present then you must throw a checked exception ‘UserNotFoundException with an id 100’, this error message must appear in JSON format

ex: GET /fetch/100

Output: {“message”:”UserNotFoundException with an id 100”}

If GET /fetch/100 has user in the database then you can show the user details in JSON format

1. Create an entity that is part of the User, which will have One to Many association, the entity must be Friend, so you should able to add friends to the user using /POST, a user can have multiple friends,

Friend entity can have 2 properties id & name

ex: If user 100 is retrieved you can show all the friends of user 100 as below:

{  
 userId: 100, name : “Raj”, dob : 2000-10-22,   
 friendsList : [{id: 1, name:”Alex”}, {id: 2, name:”Bruce”}]  
}

ex: If user 100 needs to add a friend you can use

HTTP method & URL: /POST /addFriends  
JSON Data: {name: “Raj”}

Hint: Use @OneToMany annotation in the User entity to map list of Friend entities

Docker: It is a self-contained package to run the applications in any environment without downloading & installing all the setup in the machines

It helps to ship the application with a Docker Image which will have the information’s on how to run the application & software & libraries required to run the application.

Docker helps the machines not to have the environment set-up physically, because all the set up is part of the docker container, as a self contained package, if you stop the container then along with the application the software’s libraries all will removed

To Run the application’s in docker container, you must create a Docker Image which is created with docker commands with the help of Dockerfile

Dockerfile: It is a text file with all the instructions to launch the application

Docker Image: It is an executable file for docker which is created from Dockerfile

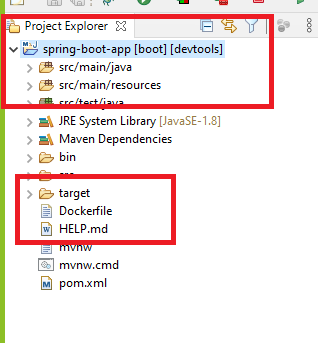
Docker: It is a software that can run the image which creates a container that will have entire application & its environment packed.

Docker Hub: It is like Git Hub, it is a repository of docker images where you can share them over the internet so that from any machine’s you can download those images

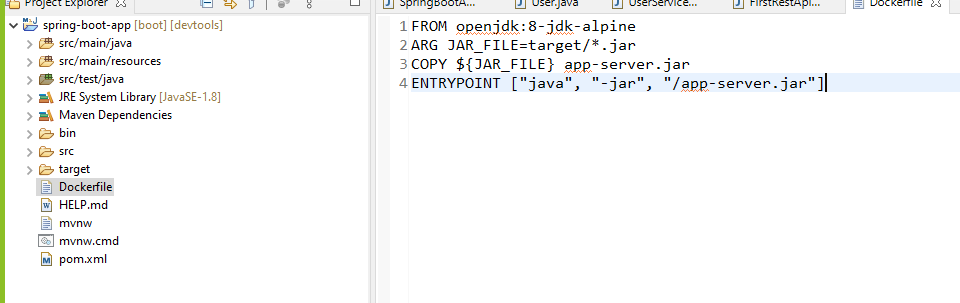
Creating Dockerfile

Note: Dockerfile must not have any extension, it is understood by docker to create docker images

Create Dockerfile in the project directory



We need to write the softwares required to run this application & the command to run this application in the Dockerfile



Since the Dockerfile uses target folder to give instruction to the Docker image we need to have a target folder before building the docker image

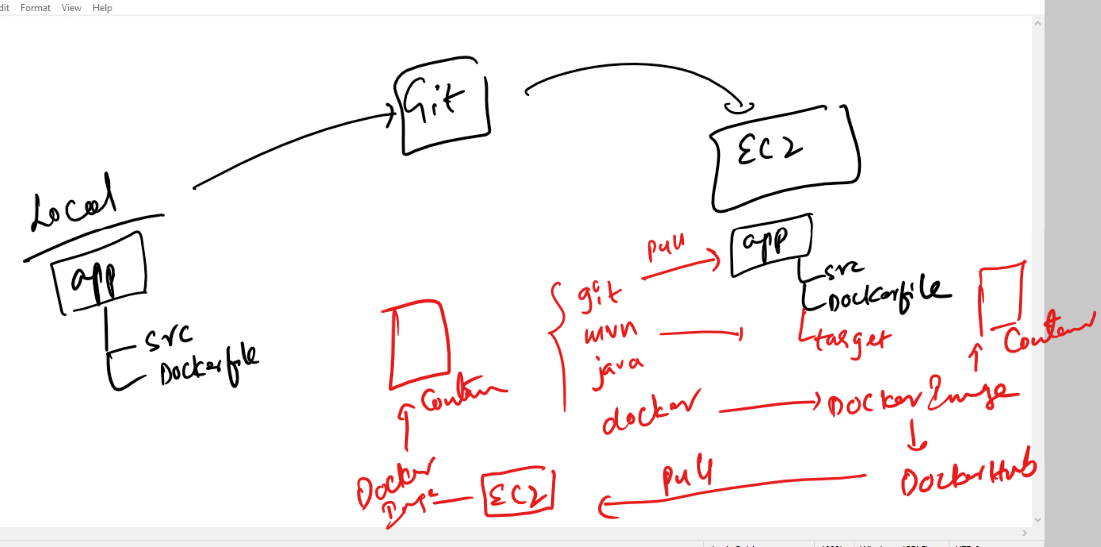
Note: Every spring initializr gives us .gitignore which doesn’t track target folder, so when you push this project to GIT and pull in any environment you need to rebuild again to get the target folder

To rebuild the application we need maven & java, if in case we are building the application to get the target folder in EC2 we need install Java & Maven both.

Note: If in case the target folder is also pushed to git, then you will get the build file in the target folder then in EC2 you don’t need to install Java & Maven,

Steps to run the application in Docker

1. Create a Dockerfile in the application
2. Write all the instructions in the Dockerfile how the application should be run
3. Since the Dockerfile uses the application build to run the application we need to create Docker image with the application build
4. Create Docker image through docker command
5. Run the Docker image
6. If you want to share the Docker image you can push to Docker Hub using docker command
7. If Docker image is available in Docker Hub you can pull it and run within the Docker container.



We will create 2 EC2 instances

* On First EC2 we will install Git, Maven, Java, Docker, after building the target folder, we will build docker image & push the image to docker hub
* On Second EC2 we will only install Docker & Run Docker image

To install GIT, Maven, Java & Docker follow the instruction given in AWS Docs

<https://docs.aws.amazon.com/neptune/latest/userguide/iam-auth-connect-prerq.html>

Above link gives you idea about how to install Java & Maven

To install git you need to use *sudo yum install git -y*

To install docker below link

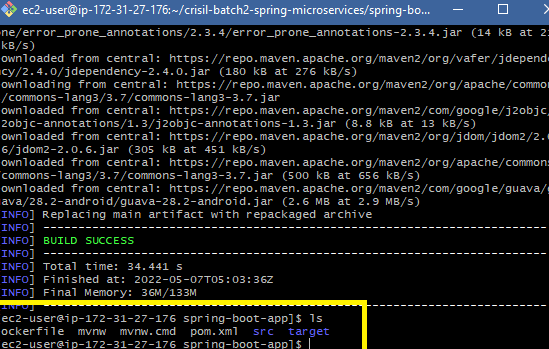
<https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-container-image.html>

Note: Use the Installing Docker in Amazon Linux 2

After installing all the softwares on the first EC2 we will build the application to get the target folder then we can install docker & build docker image

Steps to create target folder

* mvn package



Note: Since Dockerfile uses the target folder to run the application it is required to exist in the application

Next is to build Docker image, but for that you need Docker

Steps to install docker

sudo yum update -y

sudo amazon-linux-extras install docker

sudo service docker start

sudo systemctl enable docker

sudo usermod -a -G docker ec2-user

The above command lets us use all docker commands without sudo, you need to reconnect to EC2 to use that feature

docker info

Create an account in Docker Hub

<https://hub.docker.com/>

This maintains all the images you can push the docker images here and also pull them to any machines

Some of the commands of Docker

docker image: To show all the docker images in the machine

docker build -t image-name path: To create the docker image

docker push image-name: This is to push the image, but you must have logged in to docker hub through terminal

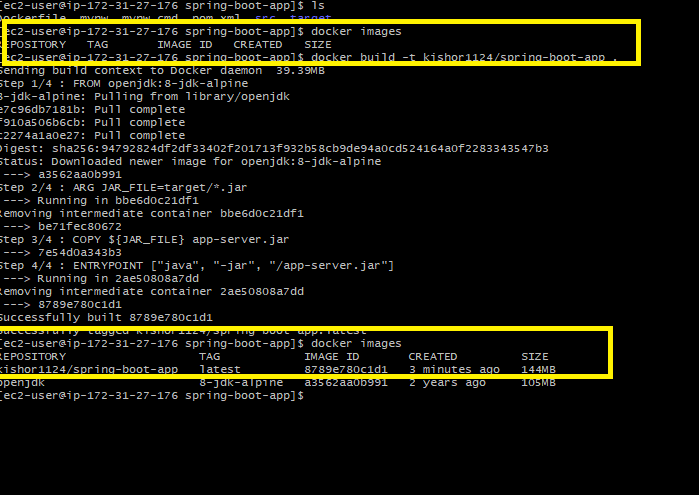
docker login: This asks username & password after this you can use docker push

docker pull image-name: This pulls the image from the logged in account, it needs to be done in the second EC2 machine

docker run: This will run the docker image

How to create Docker image

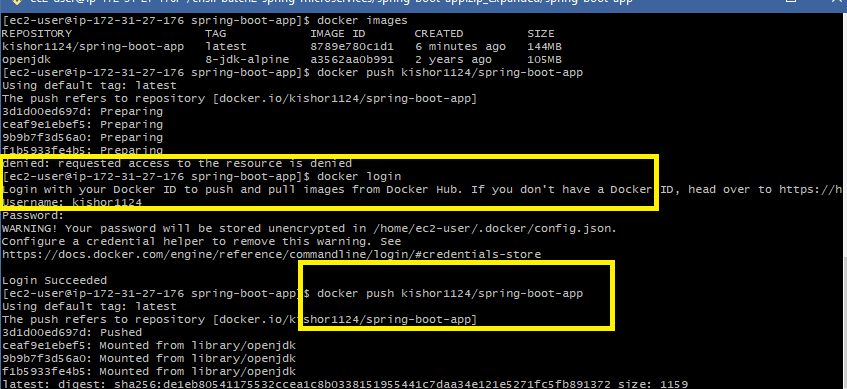
>> docker build -t image-name path



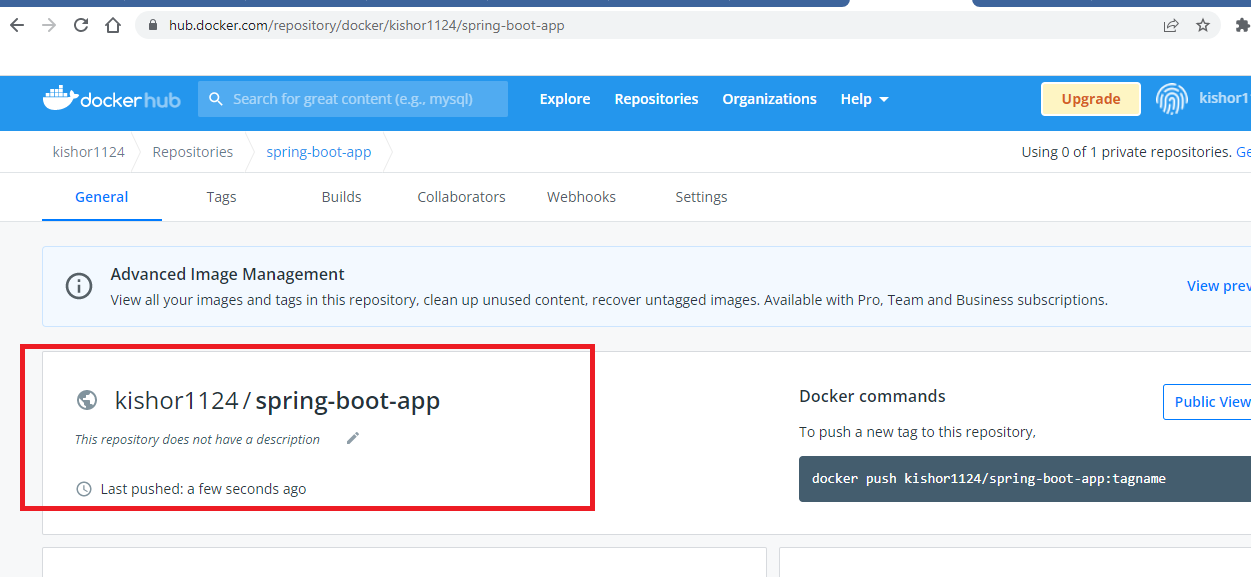
You can observe the docker build created the image the repository of the image is kishor1124/spring-boot-app

You can run this docker image here itself i.e., in first EC2 machine or you can push it to the docker hub and pull in another ec2 machine

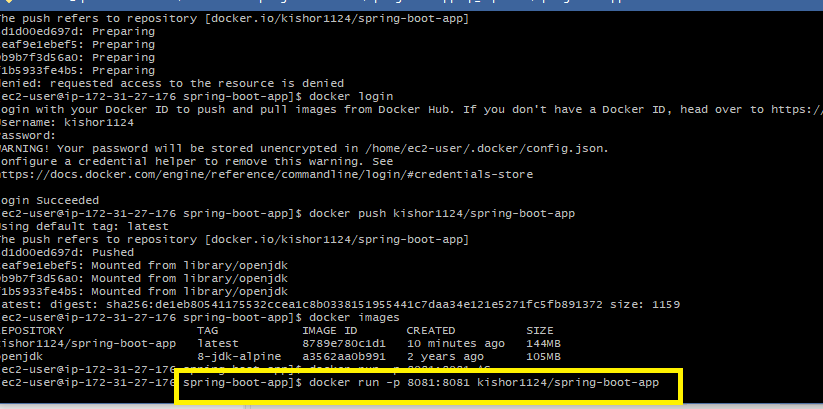
Note: To push or pull you must have logged in using docker login.



You can see the image in the Docker Hub



Note: You can run this image using docker run command

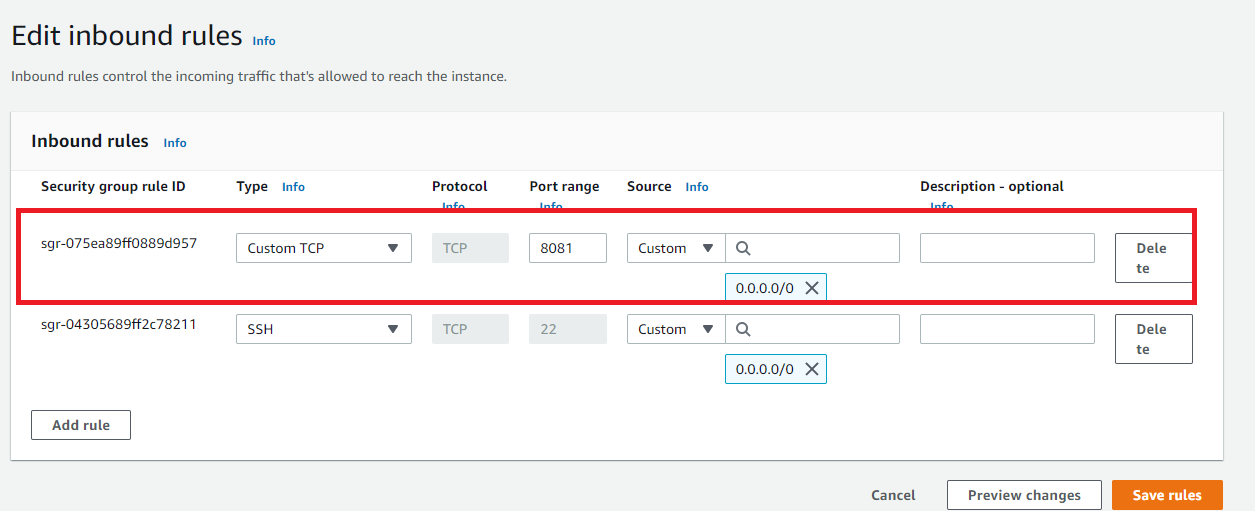


-p 8081:8081 means, the application runs in 8081 and 8081 is exposed port for the public, public must use 8081 to access the application

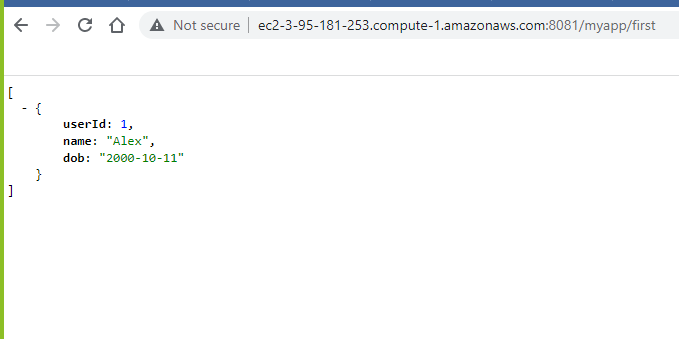
-p 8082:8081 means, you must use 8082 to access

Note: You must open the TCP port of the EC2 instance to accept the incoming request

EC2 >> Security Groups >> Inbound rules >> Add Rule >> Custom TCP >> port 8081 >>



Output:

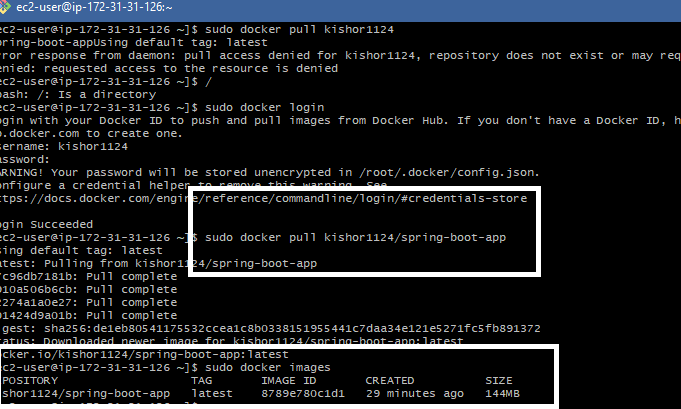


You can now download the docker image in any machine having Docker & can run it without installing any softwares

How to pull the docker image

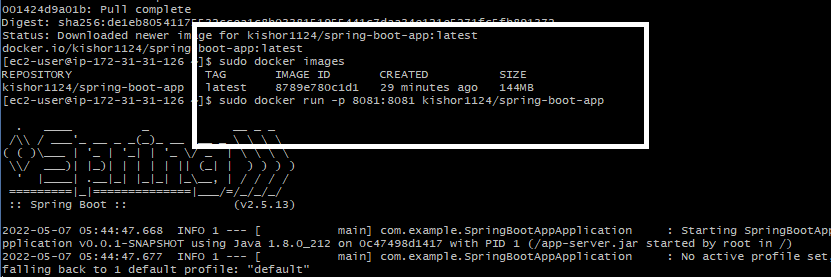
>> sudo docker pull image-id

Note: sudo is optional if in case you have done the sudo settings for docker

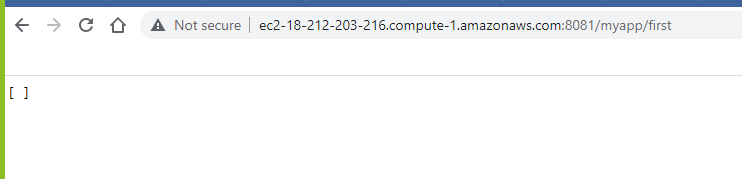


Now you can run this docker image

Note: We have not installed any software’s this is new EC2, it has only Docker



Output:

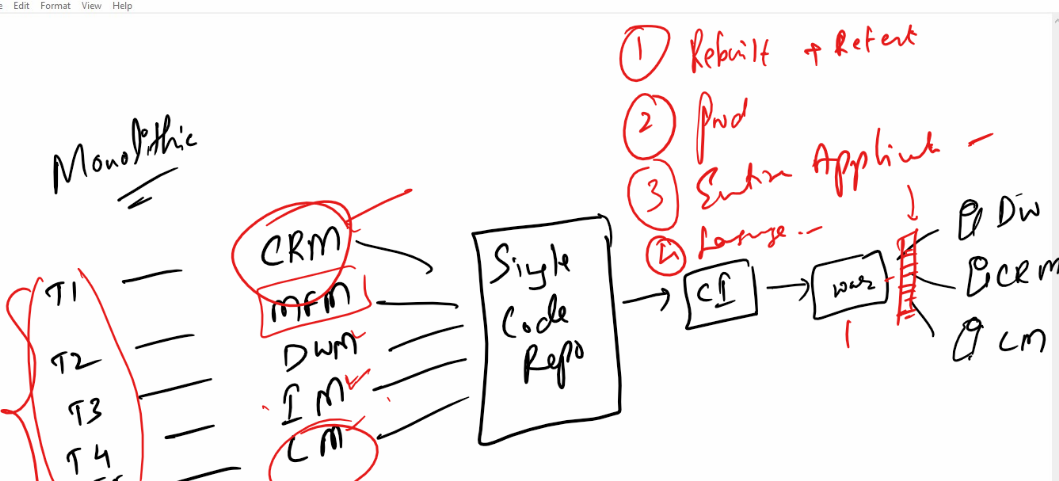


Microservices

These are loosely coupled services which can be developed, test & deploy independently from the services of same or different applications.

Microservices came somewhere around 2014 to address the problems faced by the application that follows traditional monolithic architecture

Monolithic Architecture

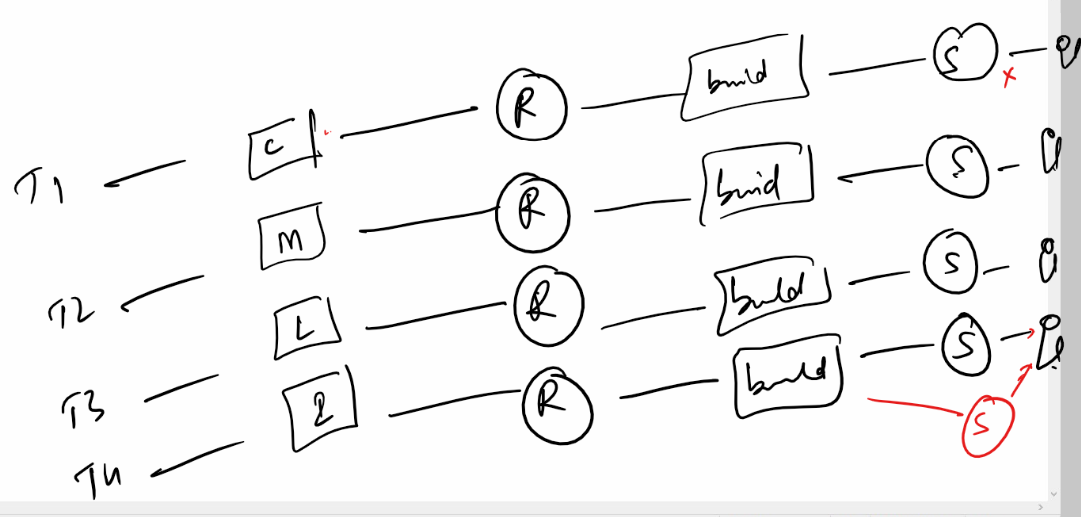


In Monolithic architecture multiple teams working in different modules of same application need to synchronize their code and build as a single artifact that can be deployed in the server

Challenges you face in monolithic style

1. When any change is done in any one module, then entire application should rebuilt & also need to test other modules as well, because they all are built as a single executable artifact
2. If any service gives exception in the production and if the entire application goes down then other services also go down though they had no problems
3. If demand increases for any service then you had to scale entire application this could utilize lot of backend resources
4. All the services must be implemented in the same language, it means you must have resources for the entire application

Microservice Architecture



1. All the services are maintained in separate code base & they can be built, test and deploy independently
2. If any service is modified then you need to test only the service that is modified, no need to test other services
3. If demand increases for any services, then you can scale only the service which are in demand
4. If any service goes down, it doesn’t affect other services, as they are deployed independently
5. You can use implement services in multiple languages as they all are rest based services & they can easily communicate with related services without any problem.

Whenever the application follows Microservice architecture it must have all the design patterns of microservices to get full benefits of the architecture

The design patterns of the microservices are:

1. Service Discovery
2. Discovery Client
3. Client Side Load Balancer
4. Circuit Breaker
5. Externalized Configuration
6. Distributed Log Tracing
7. Security

Spring Microservices

Spring Microservices uses all the design patterns the microservice needs and makes us to create microservice applications easily with simple annotations, it uses 2 projects for this

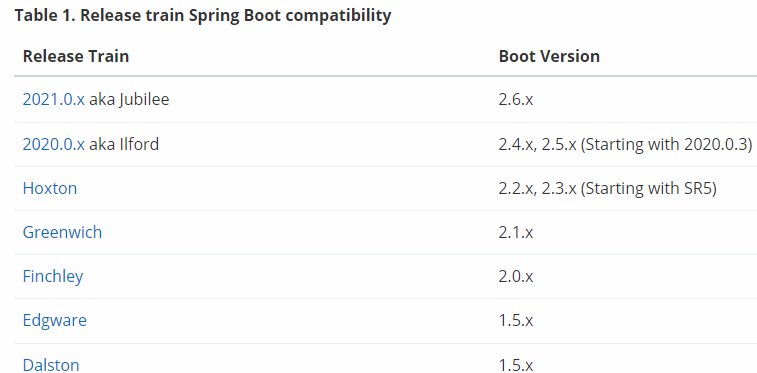
1. Spring Boot: It is used to quickly create spring applications with all the auto-configuration feature
2. Spring Cloud: It is a project that provides all the necessary tools & design patterns required to create the microservice applications with simple annotations.

Ex: If you want to Create Service Discovery then you can use an annotation called @EnableEurekaServer

This does all the service discovery job

* Keeping all the microservices in its registry with logical name & physical address
* Registering & Deregistering the services
* Giving information to the microservices

Spring Boot & Spring Cloud both are different modules hence their versions should be compatible, hence spring provides a release train using which you must understand which spring boot is compatible with which spring cloud version.



Since Hoxton is the stable release of spring cloud we can use Hoxton & Spring Boot 2.2.x or 2.3.x

Spring Cloud provides all the design patterns and helps to develop microservices with simple annotations,

What spring cloud provides for microservices

1. Service Discovery: Eureka Server : @EnableEurekaServer
2. Discovery Client: Eureka Client: @EnableEurekaClient
3. Load Balancing: Ribbon load balancer: @LoadBalanced
4. Circuit Breaker: Resilience4j: @CircuitBreaker
5. Externalized Configuration: @EnableConfigServer & Config Client
6. Security: OAuth2: @EnableAuthorizationServer, @EnableResourceServer
7. Distributed Log Tracing: Sleuth & Zipkin

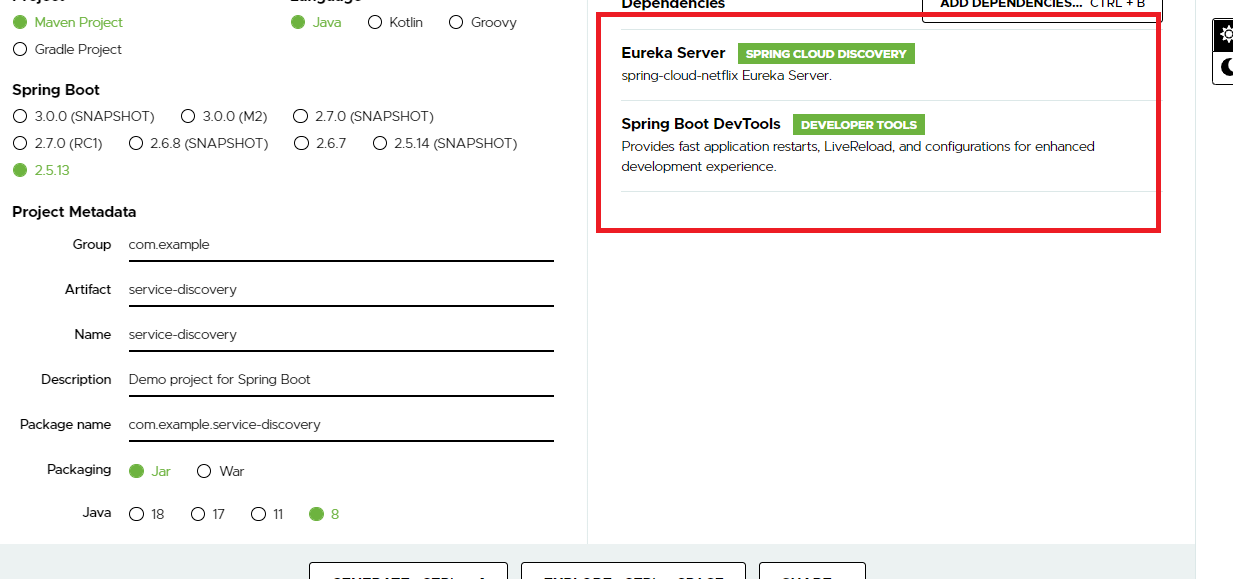
Note: Since in Spring Initializr we don’t have a provision to provide lower version of spring boot & spring cloud we need to manually change in the pom.xml

<version>2.3.6.RELEASE</version>

<spring-cloud.version>Hoxton.SR9</spring-cloud.version>

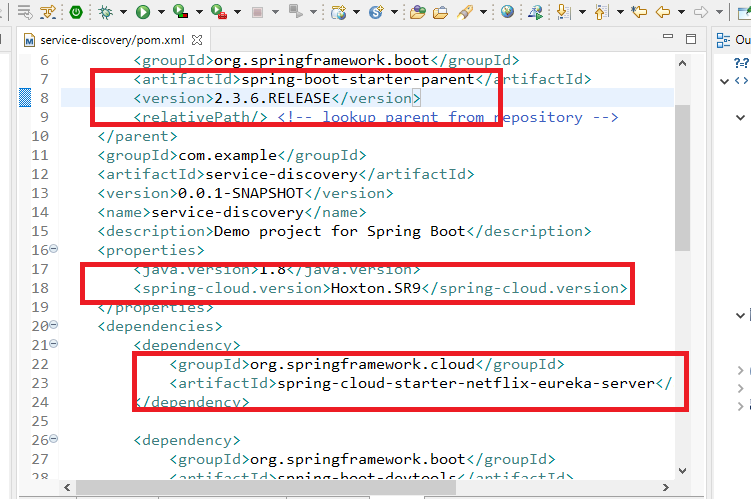
We need to use the above version to make everything work fine

Firstly all the microservices registers in Service Discovery, hence you need to create a service discovery in a separate project



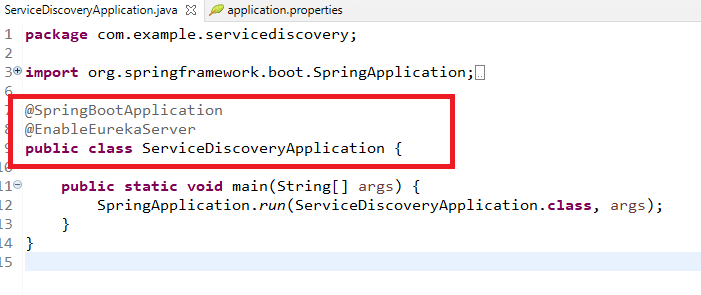
Note: Change the spring boot version & configure spring cloud version according to the compatible version

pom.xml



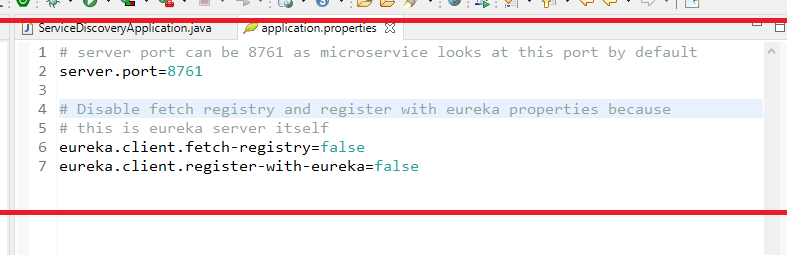
You can use @EnableEurekaServer to create service discovery, you can add this in the main class

Note: By Default all the Microservices register in the Service Discovery by searching service discovery in 8761 port number, hence you can run the service discovery in 8761 port.



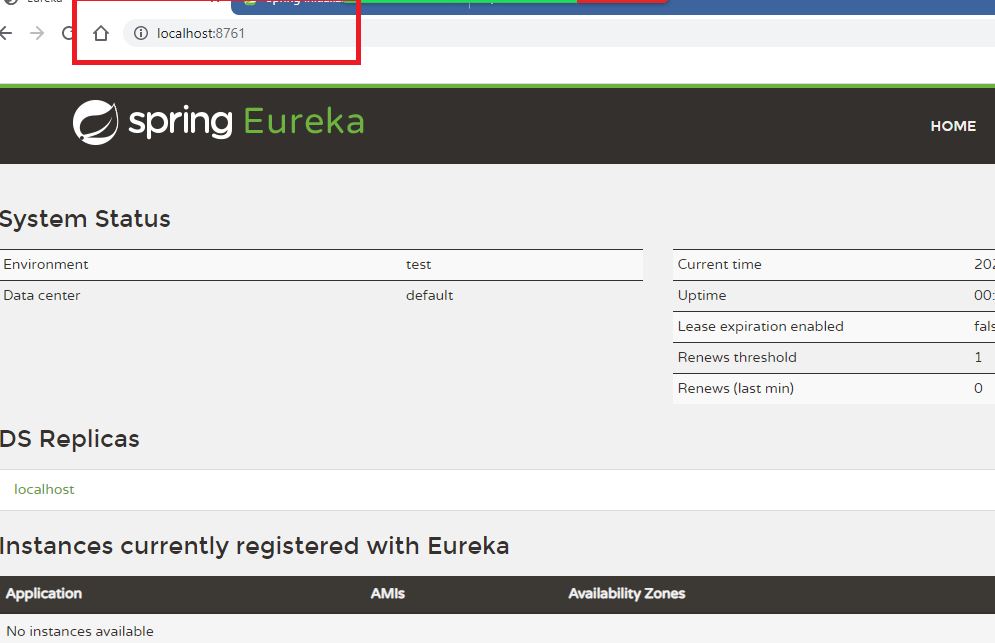
Since Eureka Server adds Eureka Client library, the spring boot automatically configures this application as Client i.e., microservices, which will try to register in the service discovery, hence we need to disable this application to register in service discovery

application.properties



Now you can run your application, this will provide a Dashboard where you can see all the registered clients, since its running in localhost you can access Eureka Dashboard in

<http://localhost:8761/>

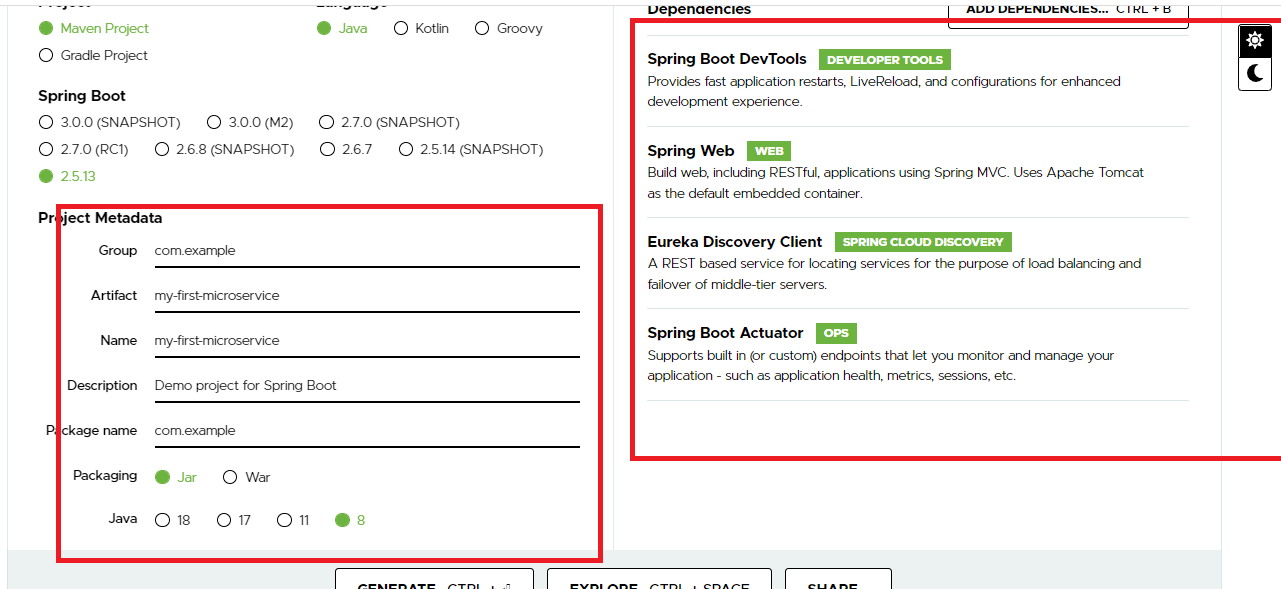


Since no microservices registered in this Eureka Server, we don’t see any instances.

We need to create a Microservice in a separate project

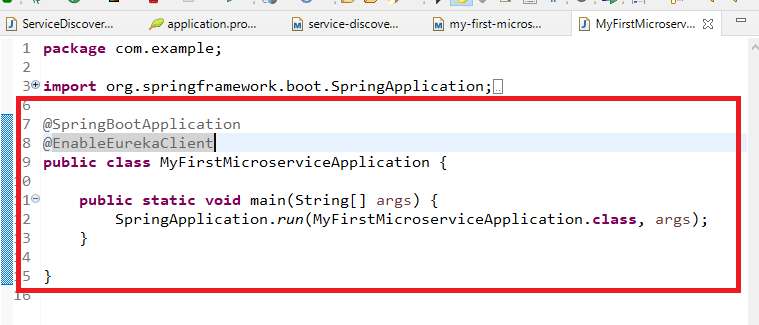
Dependencies required

1. Web
2. Eureka Client
3. Dev tools (Optional)
4. Actuator (Optional): This gives users endpoint url’s which is used to check application status, metrics, routes, circuit breaker status, and many more.



This needs to have compatible spring cloud & spring boot version as well, hence change the POM accordingly.

Use @EnableEurekaClient to create your application as a microservice

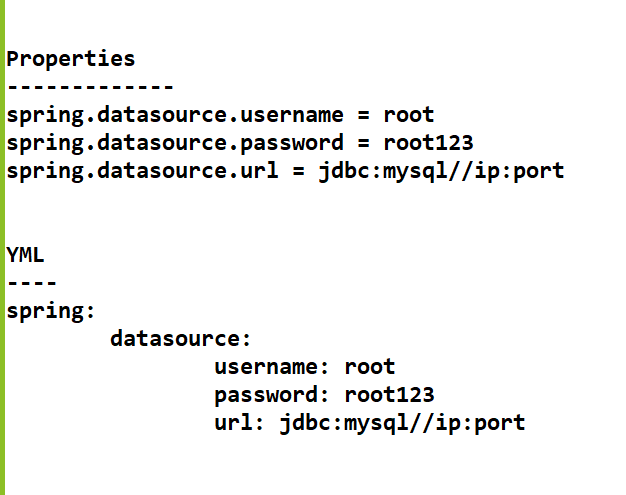


@EnableEurekaClient:

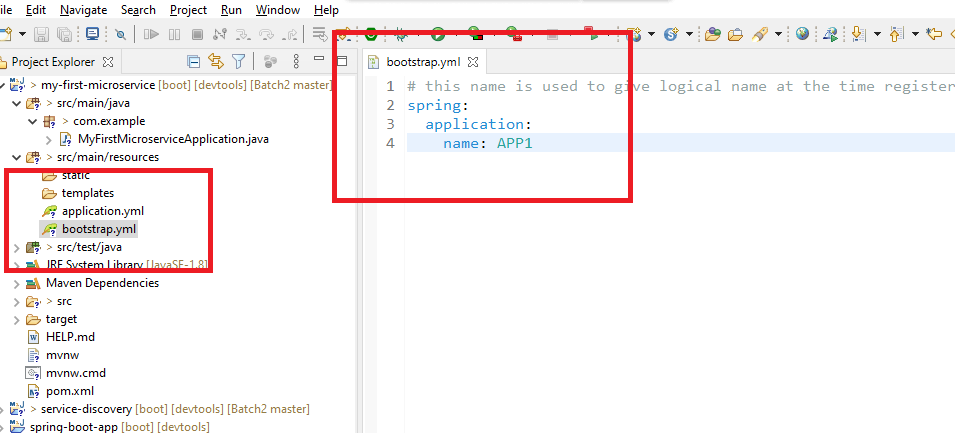
* Registers this application as a microservice in Eureka Server
* Searches eureka server in 8761 by default
* Pings Service Discovery every 30s to its update health status
* Fetch Service Discovery
* Microservice registers with the logical name configured in the property file

Note: spring boot loads bootstrap.properties file before application.properties, we can write the logical name in either bootstrap or application properties, but there are some properties which you may need before application property loads.

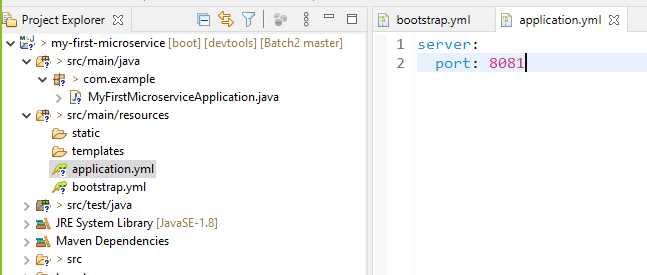
Note: You can use YML files instead of properties file, yml avoid writing repeated properties through indentation



bootstrap.yml

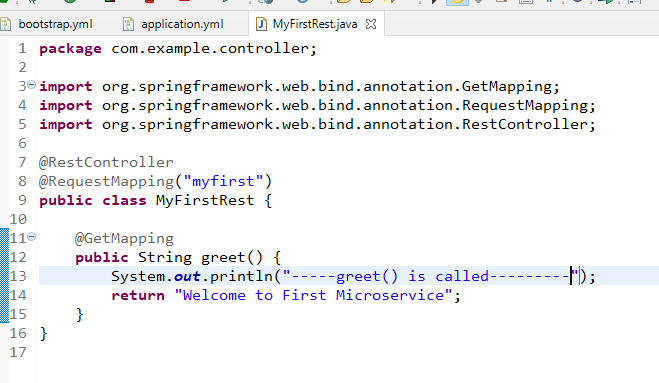


application.yml



This port is useful when you want to independently test the microservice using 8081, however when microservice communicates with this microservices then it doesn’t use this port, instead it uses APP1 (logical name), since all the microservices registers in the service discovery it will know APP1 is the microservice it needs to communicate.

MyFirstRest.java

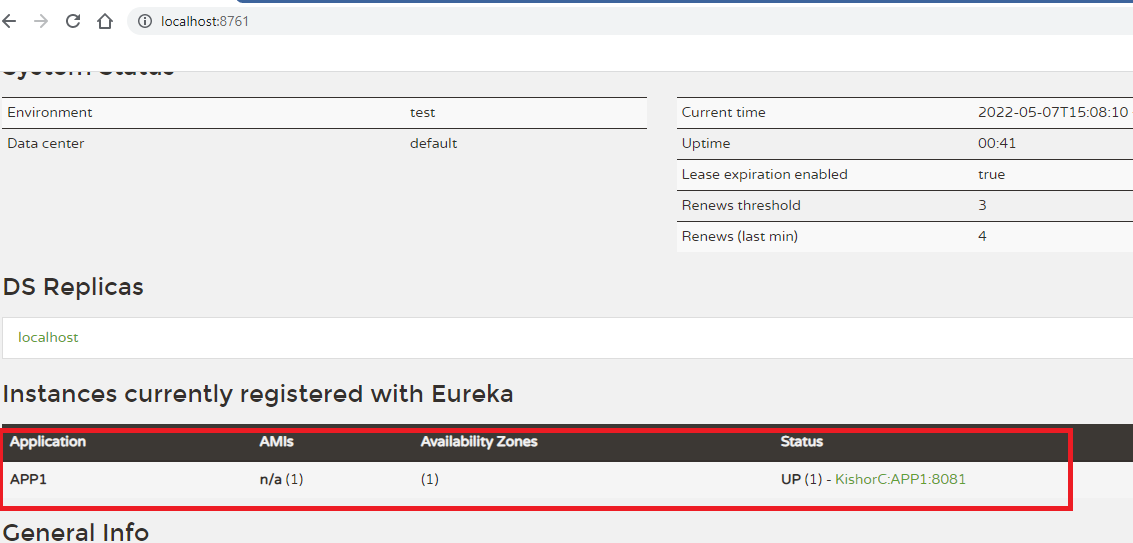


Though it looks like a webservice, it is a microservice that registers in service discovery with a name that is used by other microservices to communicate

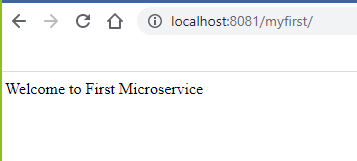
Once you run the above microservice it registers with Eureka Server & also its name will be shown in the eureka dashboard

Note: You can run the same microservice in different port which will create another instance of same microservice & in eureka dashboard you can see 2 instances of same microservice, their application will be same

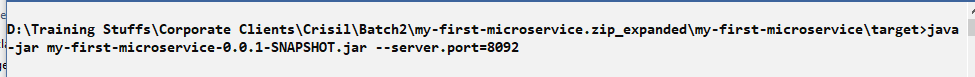
Note: You can use java -jar command to run multiple instances of same microservices & change the port using *--server.port*



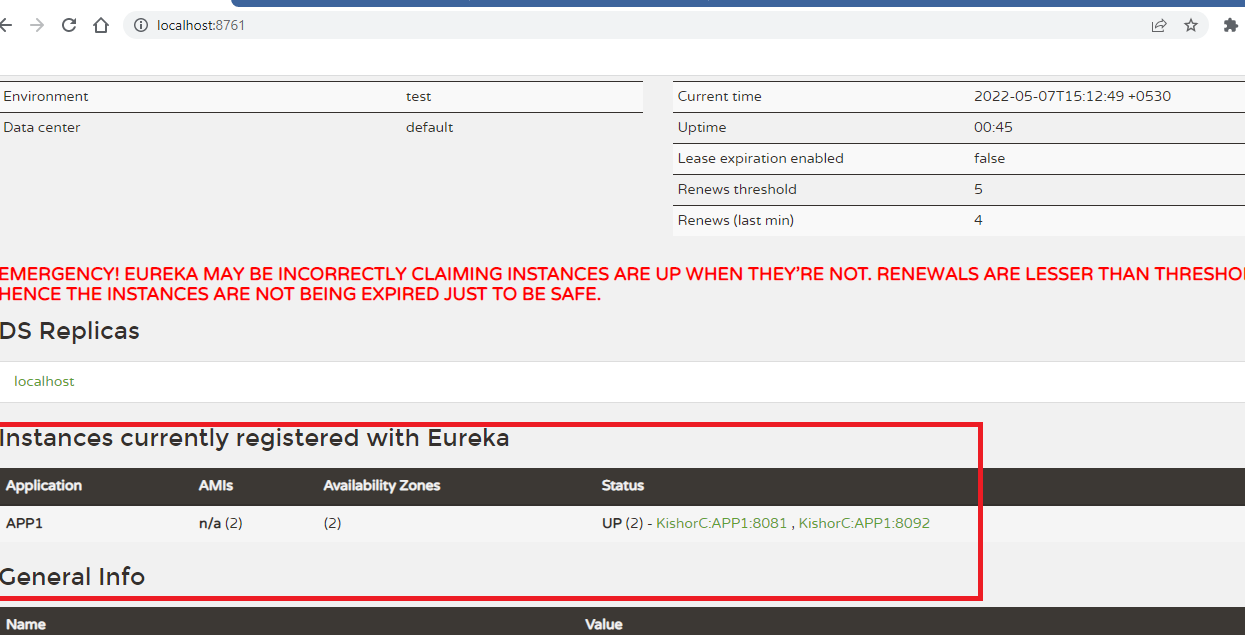
But if you want to test this independently you can use 8081 port, but if you are communicating with a microservice then you can use <http://APP1/>



You can run the same jar in different port to see another instance registered with the same application name but different IP



Now this application also registers in eureka with the same application name, hence you see 2 instances in eureka dashboard



Note: When another microservice is registered, then it can use APP1 to call this microservice, but the request will be distributed by the client side load balancer

Microservice Communication

When a microservice communicates with another microservice it has to happen via REST calls, You can use following methods to call the microservices from another microservice

1. RestTemplate
2. FeignClient

RestTemplate: It is an instance that provides methods to perform HTTP calls & it takes an extra parameter that helps the response data to convert to specific type

You must use @LoadBalanced RestTemplate so that load balancer helps to determine which instance of the microservice to access, RestTemplate uses another argument that represents the response data type.

ex:

RestTemplateObject.getForObject(URL, classname)

RestTempalteObject.postForObject(URL, data, classname)

..

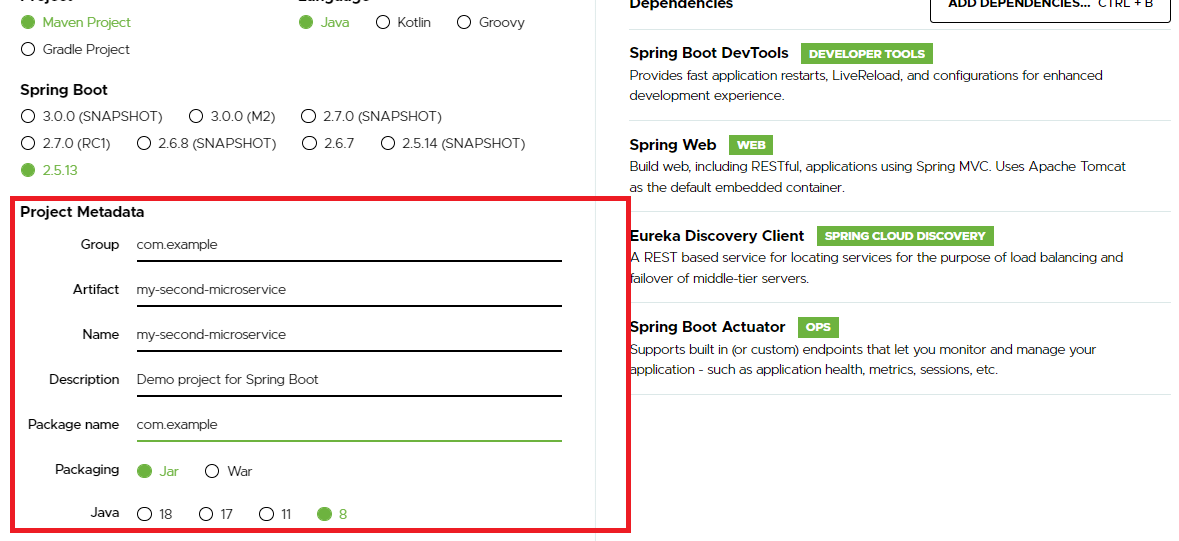
Feign Client: It uses interfaces to access the microservice, in the interface methods you mention the end point of Microservice, response type & HttpMethod to access end point.

You don’t need to implement this because spring boot takes care of implementing it, you have to call these methods to invoke microservice

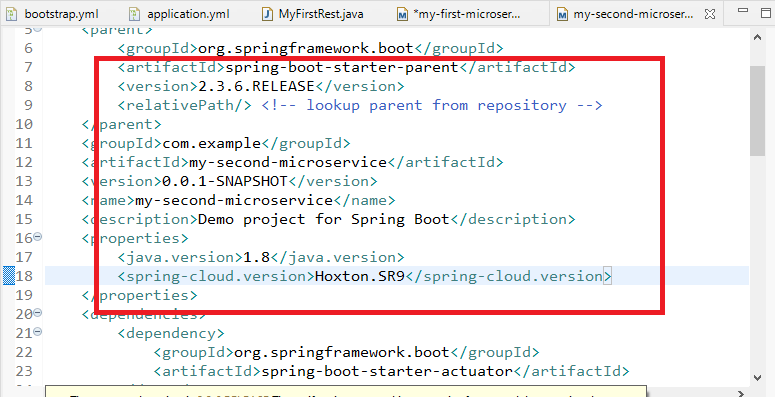
Advantage:

1. It is reusable
2. It reduces mentioning the microservice url & type of the response every time
3. It internally uses load balancer, you don’t have to use @LoadBalanced

We need to create another project and create microservice to call another microservice

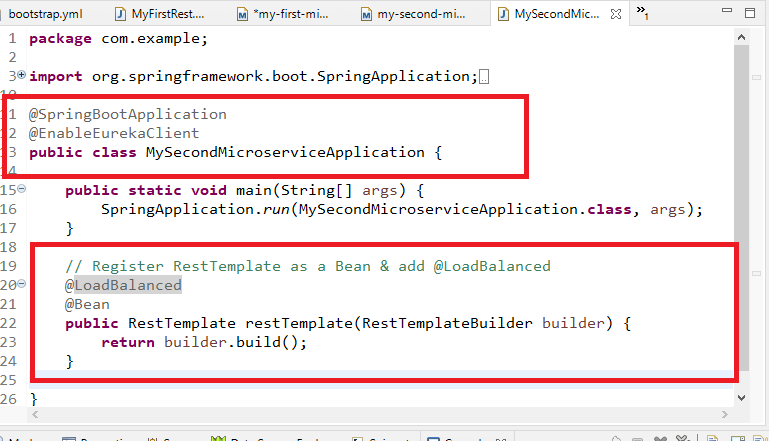


Note: Modify pom.xml



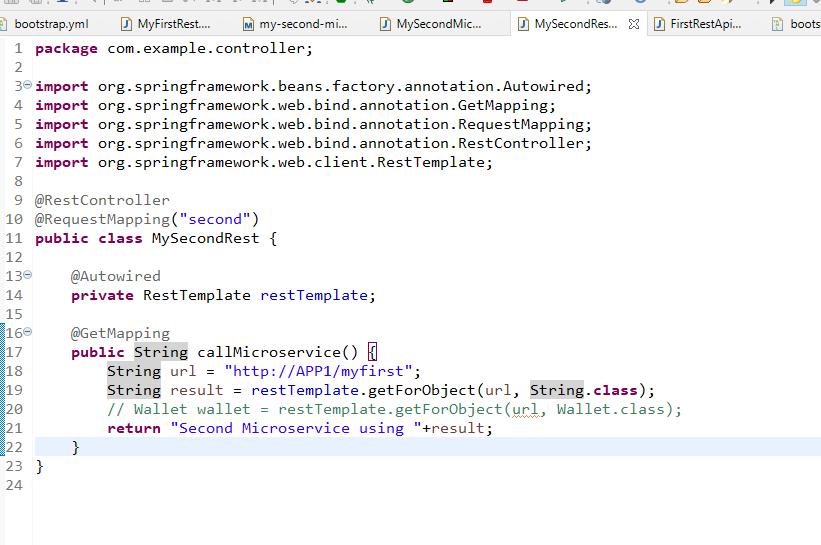
Since we are using RestTemplate to access the microservice we need register this object in a spring container & also use @LoadBalanced on top of it, it can be autowired in any place.

* Add @EnableEurekaClient as it needs to be registered
* Register RestTemplate as a bean & add @LoadBalanced so that through logical name you can access microservices not by using physical address



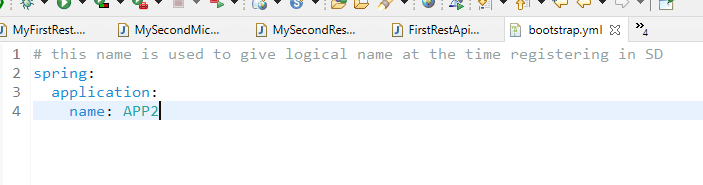
Now you can call the first microservice from second microservice via Rest Calls, for that we can create a Rest Controller.

MySecondRest.java

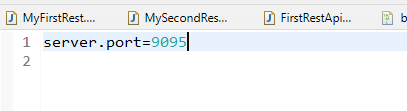


If your call gets a json like {“walletNo”:12345, “amount”:2000}, then you must have a model to represent walletNo & amount i.e., Wallet class

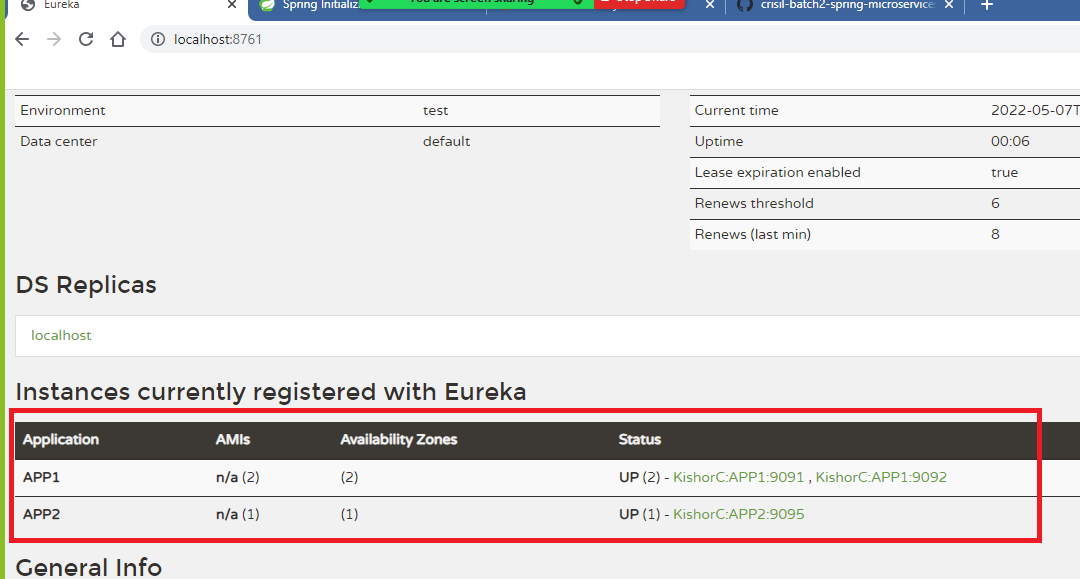
Configure bootstrap.yml to mention the logical name for the second microservice



application.properties

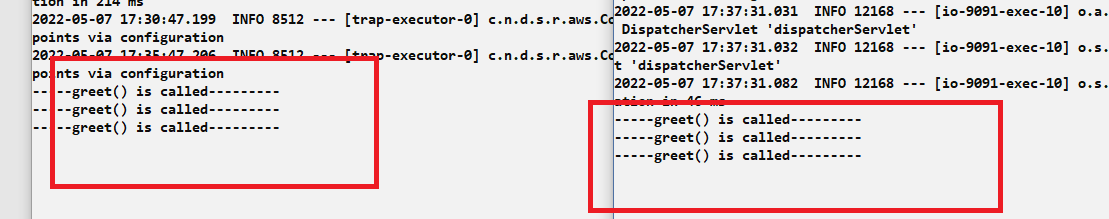


You will see 2 instances of APP1 and 1 instance of APP2



We can test second microservice with localhost:9095, but we must see both 9091 & 9092 can respond alternatively as LoadBalancer in second microservice distributes the load of the request in a round robin fashion.

If you send request to 9095/second multiple times you must see greet() printing in different terminal alternatively.



Drawbacks of using RestTemplate

1. It is an older way of using rest client
2. It is designed to access webservices with physical address
3. You need to manually configure the LoadBalancer to use the microservice application name (Logical Name) so that the call you make to the microservice should be mapped to which physical address is determined by the LoadBalancer
4. It uses its method like getForObject(), postForObject(), putForObject() and so on to access with HTTP methods & they need to mention class names in their arguments to specify the response data needs to be converted what type

ex: getForObject(url, String.class): The response you get is converted to String

ex: getForObject(url, Wallet.class): The response you get is converted to Wallet object, however the response JSON must have the properties that matches to wallet

ex: postForObject(url, inputData, Account.class): The inputData is converted to JSON and response JSON data is converted to Account object

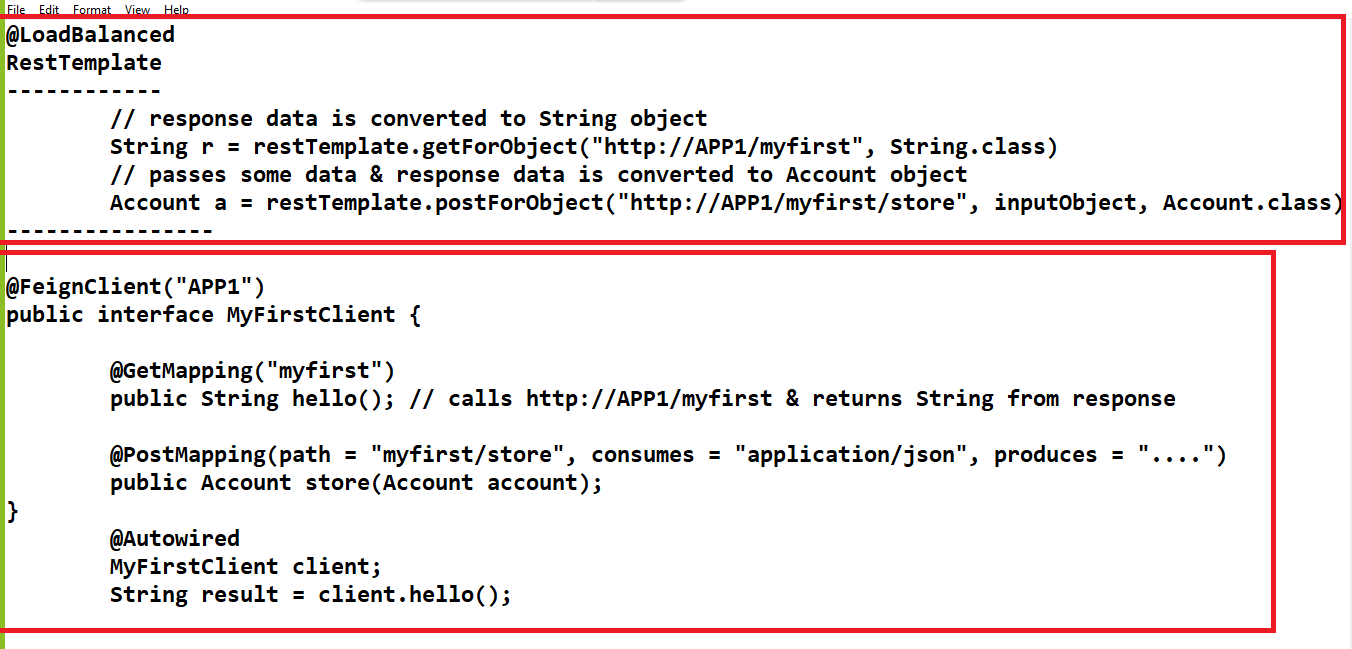
1. The REST calls are not reusable, If you need to call same service then you need to repeat the REST calls you write via RestTemplate

ex: if you want to call postForObject(url, inputData, Account.class) in three different places, then you need to repeat those calls with same set of arguments

1. Since RestTemplate REST calls are redundant in nature it will be a bit difficult to maintain when there is some change

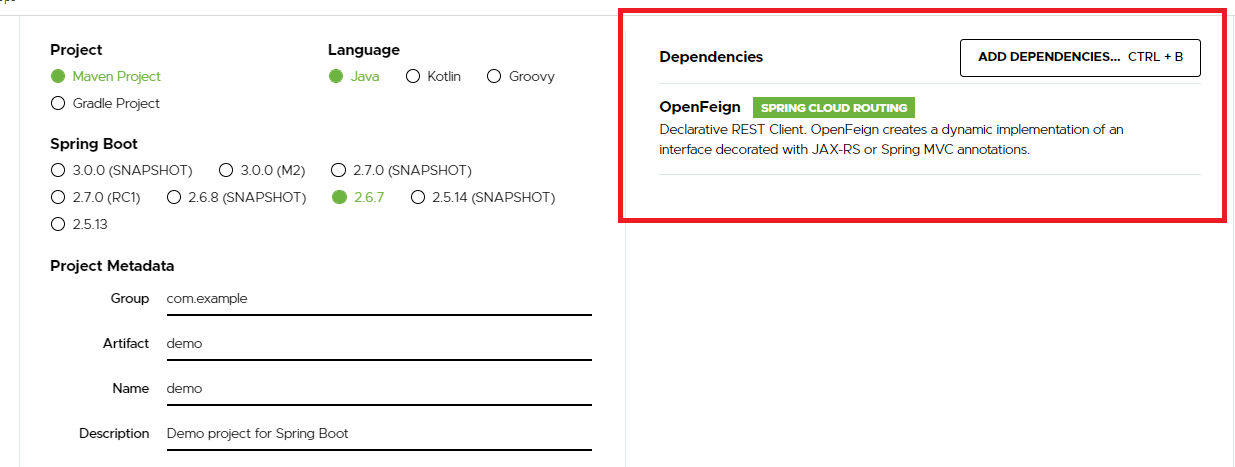
Feign Client

* It is Rest Client used to call the Microservice in a declarative way, means completely abstracts the way microservice is called, i.e., abstracts URL, response type, request content type, and so on
* Feign Clients are created with Interfaces, you can use that interface reference to call the microservices
* Feign Client are interfaces that declare the methods, which will have the call to the microservice and these methods also specifies the data-structure of request & response

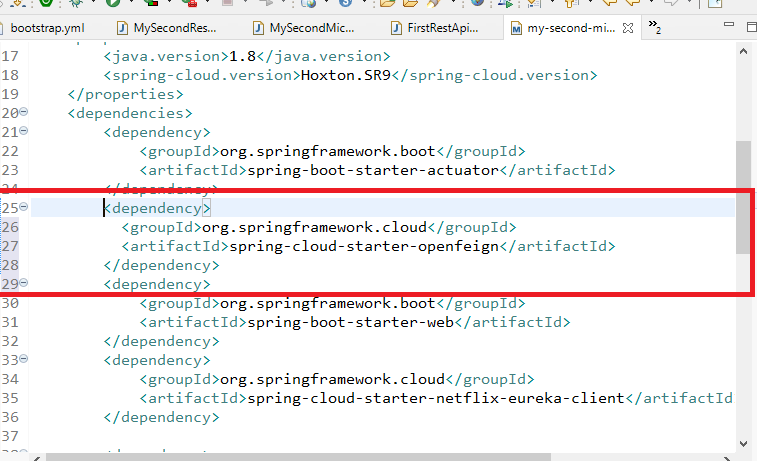


* When you use @FeignClient, it automatically uses the LoadBalancer which determines the physical address through the logical name
* Spring Boot implements the interface to do the Rest calls when you call the @FeignClient interface methods
* For Spring Boot to implement the @FeignClient it needs to use Open Feign library and @EnableFeignClients annotation in the spring boot application, this annotation scans for @FeignClient to make spring boot to implement it

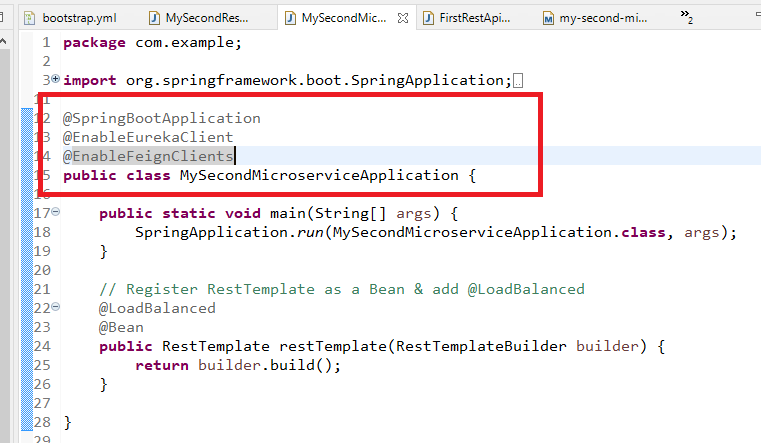
Add the Open Feign library in the existing project



mysecond-microservice/pom.xml

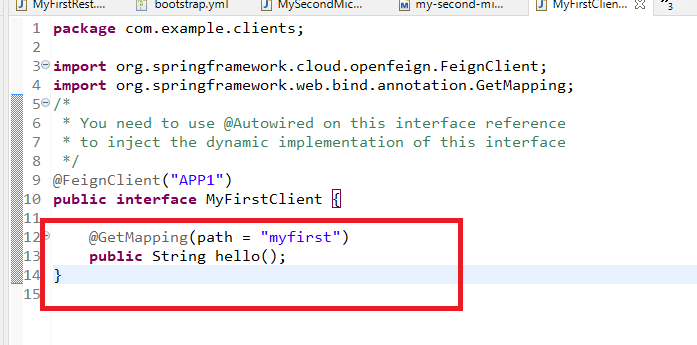


Add @EnableFeignClients at the top - level to scan the @FeignClient interfaces



Note: Since RestTemplate is not used we can comment out it else don’t need to use it

MyFirstClient.java



Note: Now the hello() method makes a REST call using HTTP GET the end point address will be <http://APP1/myfirst> which returns String object

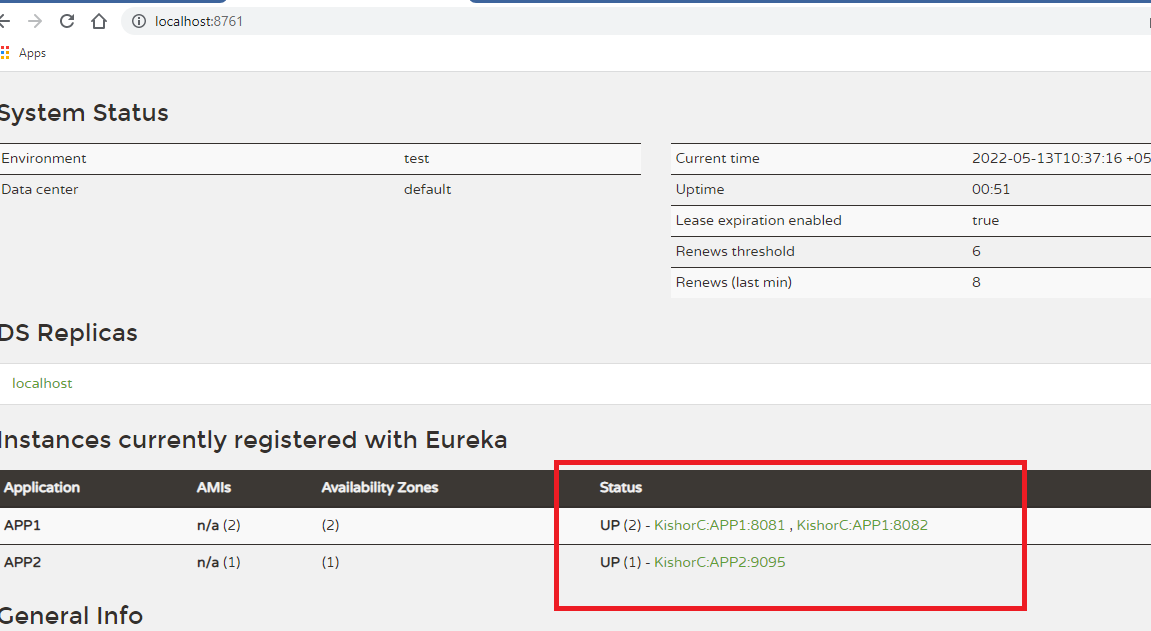
Note: You need to use this interface reference and @Autowired annotation to inject the dynamic implementation of this interface

MySecondRest.java



Now you can run first microservices atleast 2 instances to see the load distribution & a second microservice and call the second microservice end point second/v2

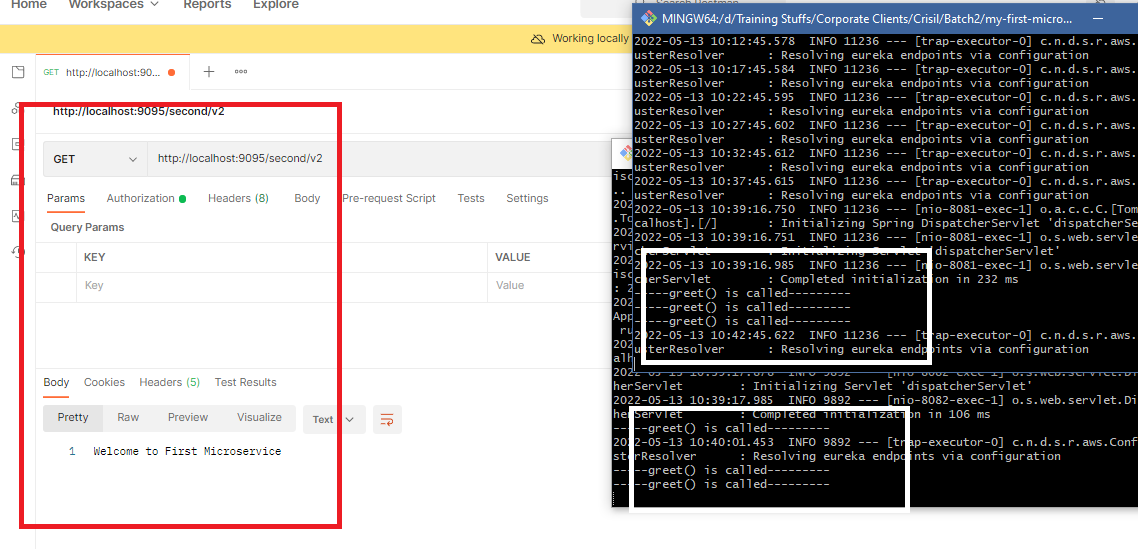
The eureka dashboard shows the number of instances of first application



When you send request to second microservice the request is sent to first microservice

Note: You must call the second microservice using physical address when you are testing through postman

Output:

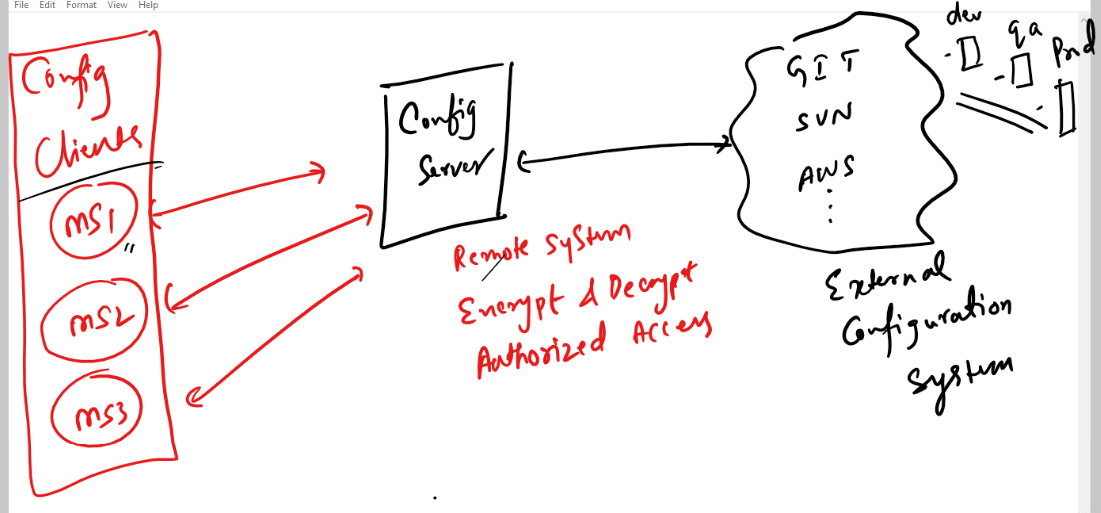


External Configuration / Centralized Configuration

* It is a centralized location where you can maintain all the application configuration files i.e., properties/yaml files.
* It can be made private so that only authorized programs can access it with right credentials
* It can keep the sensitive data in encrypted format so that nobody could able to understand except the program that encrypted can decrypt it

Spring Cloud two programs to use the external configuration system

1. Configuration Server
2. Configuration Client



Configuration Server:

* It is a program that connects to the External configuration system like GIT, AWS S3, Local Repository of the organization and so on
* It will have full information’s about the cloud system it needs to connect like URL of the external config system, authentication details to connect to private repository, encrypt & decrypt logic
* It accepts the request from the config client to provide the configuration the client needs
* It completely abstracts the external configuration locations from the config client, so that if the location of the external configuration changes then configuration server needs to be changed but not the configuration client

Configuration Client:

* It is a program that connects to configuration server
* It can be a microservice also
* It can specify at runtime what configuration file it needs based on the environment it runs like dev, qa, prod
* it needs property or yaml files but it doesn’t need to maintain the configuration files in the project if its common to multiple clients, it can be loaded at runtime when the application starts

Note: Configuration Server is a program that will connect to cloud systems, all the microservice related programs like Service Discovery, Microservices can act as a clients to this configuration server

Note: Configuration server will not have any business logics related microservices

We need to create 2 programs minimum

1. Configuration Server that connects to cloud for external configuration files
2. Configuration Client that connects to Configuration server

Before creating these programs we need to keep the configuration files in the cloud i.e., GIT/AWS S3

We can create 4 configuration files with the name hello

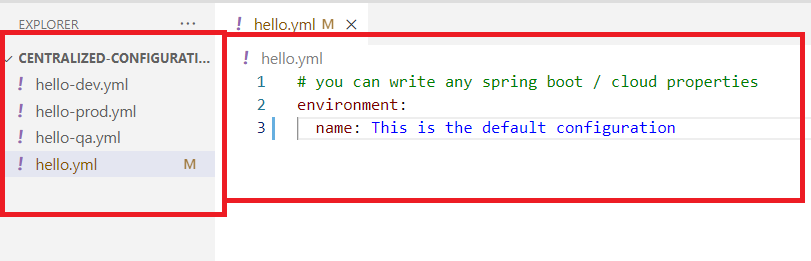
1. hello.yml
2. hello-dev.yml
3. hello-qa.yml
4. hello-prod.yml

In all these configuration files we can store some spring cloud/boot/ custom properties that can be used by config clients / microservice

hello.yml is the default file loaded if config clients don’t specify which file they need

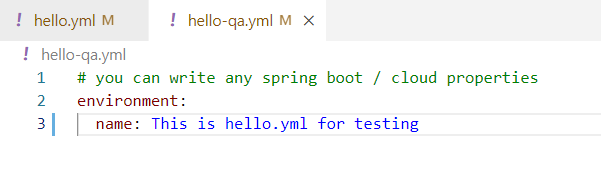
hello-dev.yml is the file loaded if config client specifies ‘dev’

similarly other files are loaded based on the profiles like dev, qa, prod

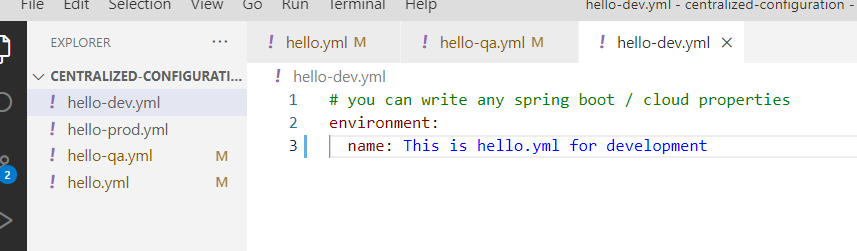


Note: environment.name is the custom property but we can other properties also like dasource properties, eureka server / client properties, actuator endpoints, circuit breaker properties

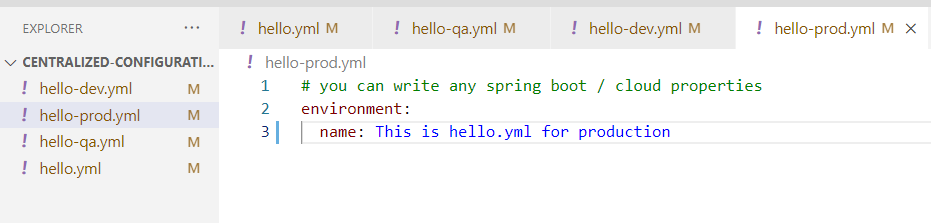
hello-qa.yml



hello-dev.yml



hello-prod.yml



Though we have 4 properties Microservice can specify which property file it needs using spring.profiles.active & spring.application.name

ex:

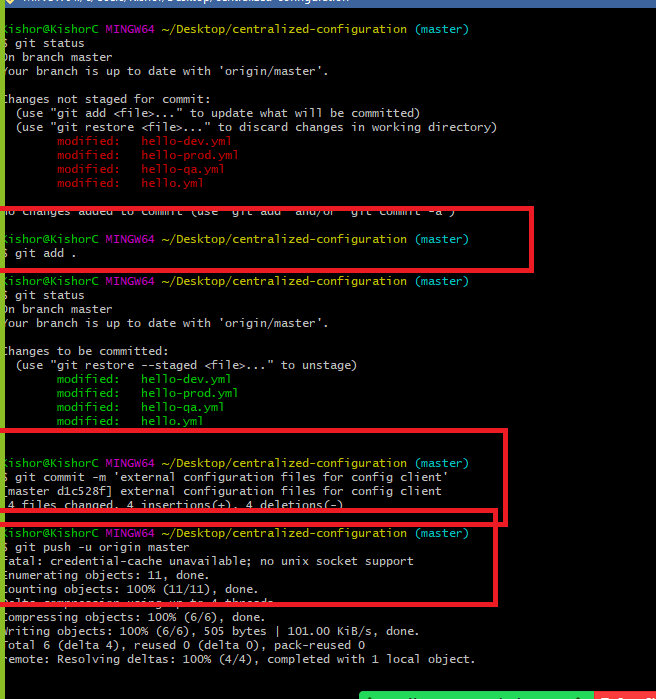
# spring.application.name = hello

# spring.profiles.active = dev

If the above properties are used in the config client then it sends request to configuration server that it needs hello-dev.yml

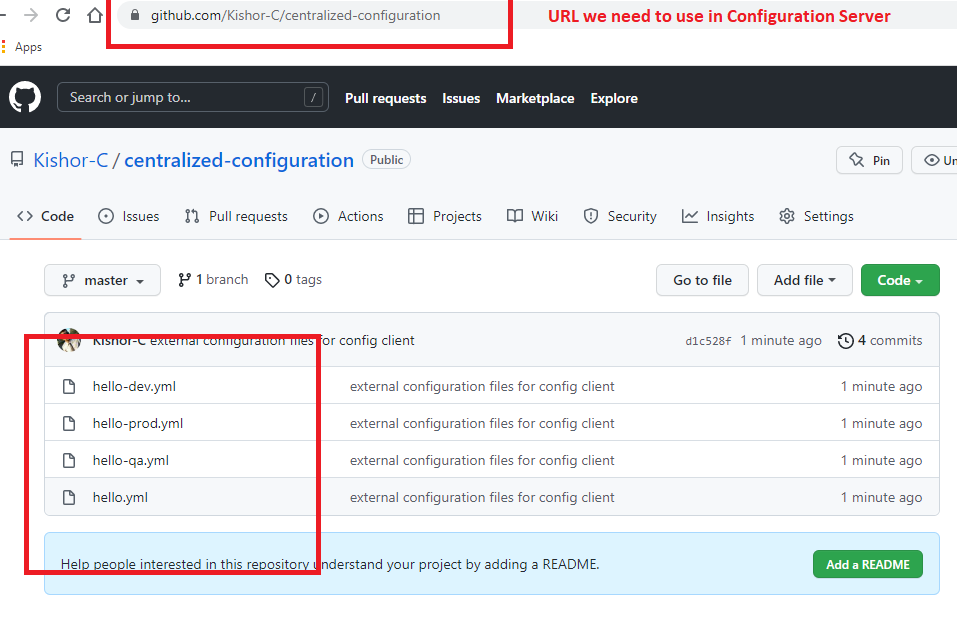
If the profiles.active is ignored then the configuration server gives hello.yml

Now you can push the above 4 configuration files to the cloud i.e., Git repository and its url you can use in Configuration server

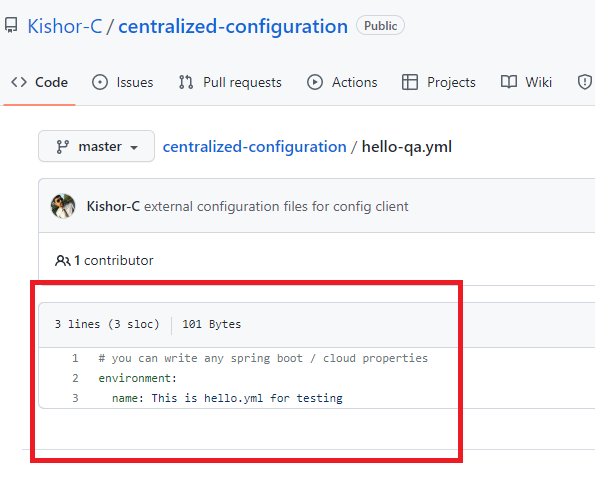


Now you can see the configurations in the GIT, the URL of the GIT repository is

<https://github.com/Kishor-C/centralized-configuration.git>



If you see any yml file you can see the text data as it is, it is not encrypted



Note: Configuration server will take care of encrypting & decrypting sensitive informations, it needs to use a key to encrypt & decrypt & provides an end point /encrypt & /decrypt which can be used to encrypt & decrypt

Now the configuration files is ready in the cloud, we need to create 2 programs

1. Configuration Server:
   * Dependencies required: Config Server
2. Configuration Client
   * Dependencies required: Config Client, Web

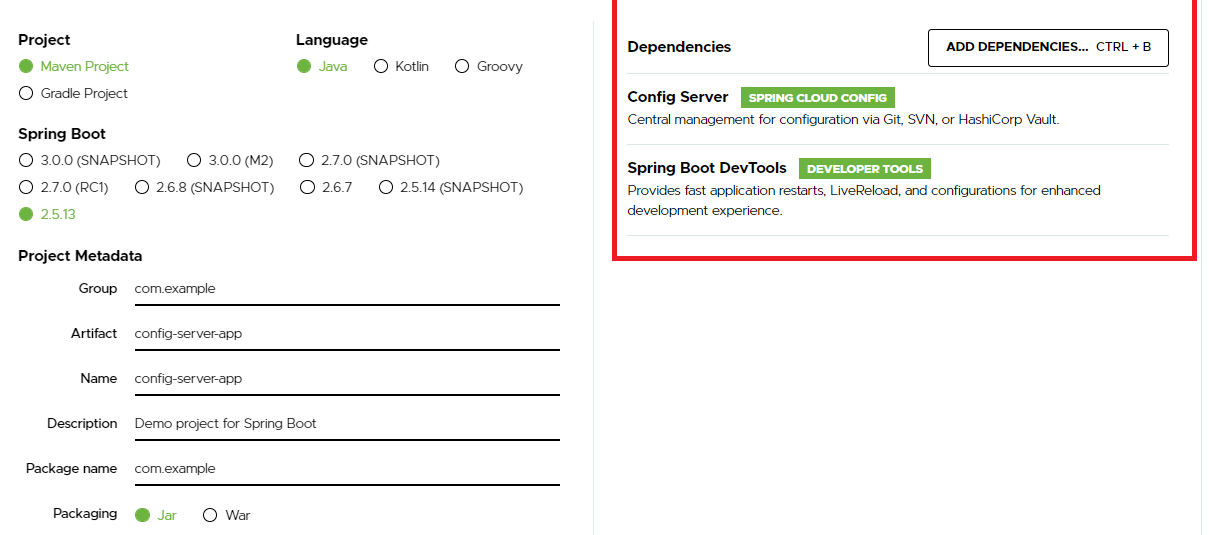
Configuration Server:

* It uses @EnableConfigServer to act as a configuration server
* It will have a configuration about the cloud system it needs to connect

Configuration Client:

* It uses the url of configuration server
* it uses application.name to specify it needs from the configuration server
* it uses profiles.active to specify it needs configuration file of which domain
* Optionally it can be registered as a microservice, have webservice end points and so on

Configuration Server



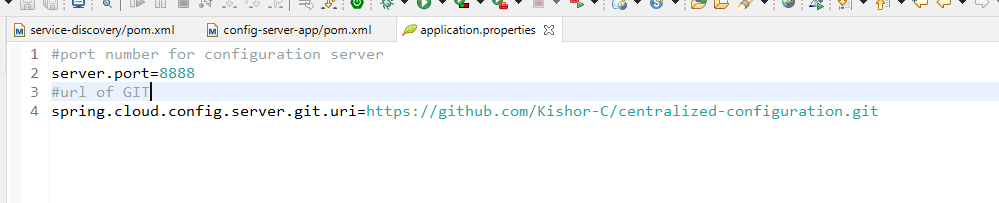
Note: Change pom.xml to use Hoxton.SR9 for spring-cloud version & change spring boot version compatible to spring cloud version



Now you need to connect to the GIT as we have configuration files in the GIT, the URL for the configuration files is

<https://github.com/Kishor-C/centralized-configuration.git>

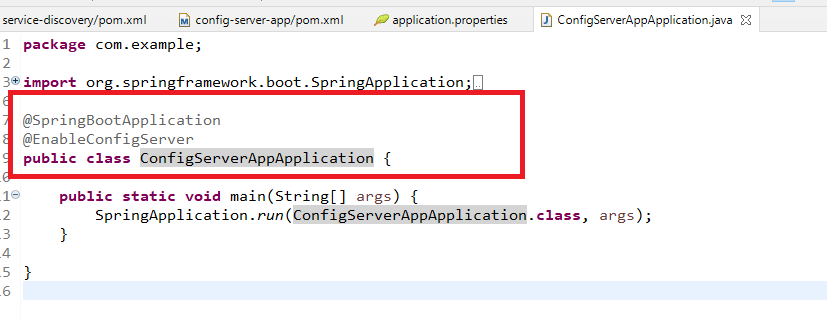
application.properties



Note: you can use yaml files also

Now the program must use @EnableConfigServer to act like configuration server

ConfigServerAppApplication.java



@EnableConfigServer:

* Takes care of connecting to the GIT according to the URL mentioned in the property file
* Takes care of using a key for encryption & decryption
* Takes care of authenticating to the private repositories using username & password

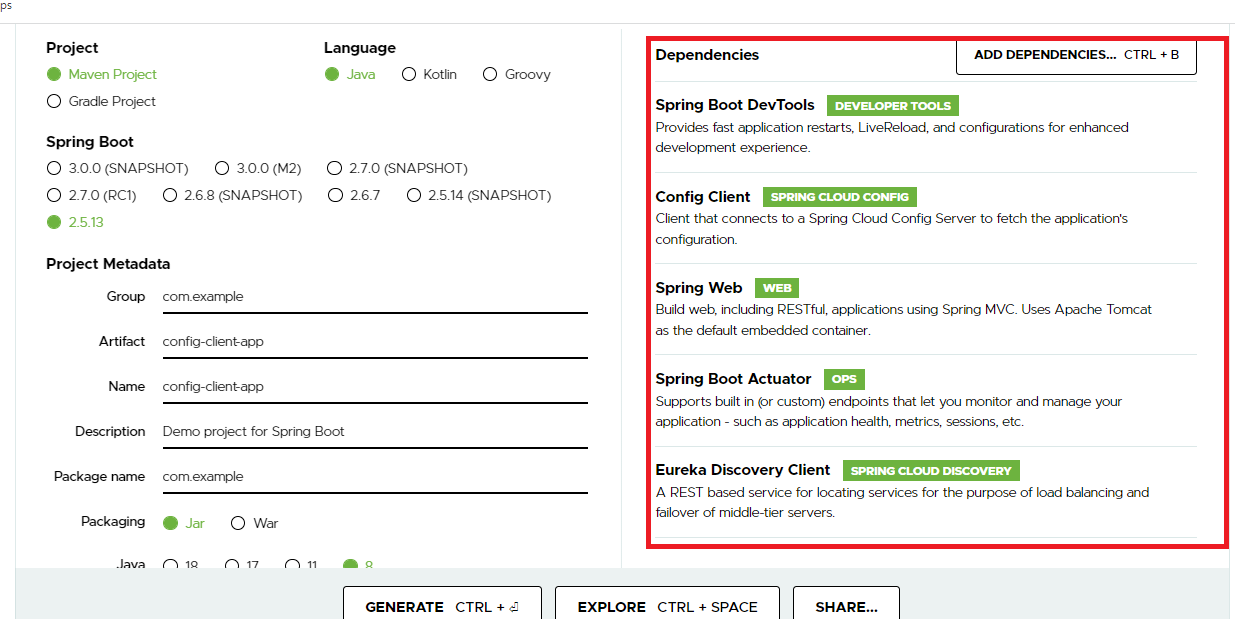
Now you can run the configuration server program

You can use config client program to connect to the configuration server, since configuration server runs in 8888 port we need to use this in the configuration client.

Configuration Client:

It is a service / microservice that will connect to the configuration server to get the configuration properties it needs, it needs following dependencies.

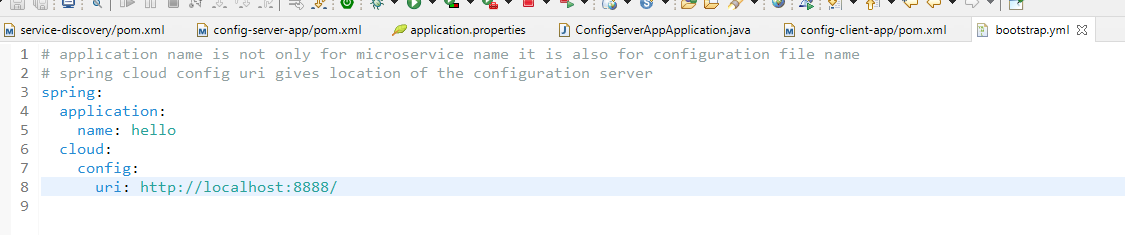
1. Config Client
2. Web
3. Eureka Client
4. Dev tools
5. Actuator



Note: Change the pom.xml as per the stable release of spring boot & cloud

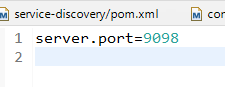
Use bootstrap.yml to connect to configuration server & mention the application.name & profiles.active to fetch the configuration files

bootstrap.yml



Configuration client will connect to configuration server & expects it to fetch hello.yml file as there’s not active.profiles, if it exists then it can fetch hello-dev.yml or hello-prod.yml and etc.

application.properties

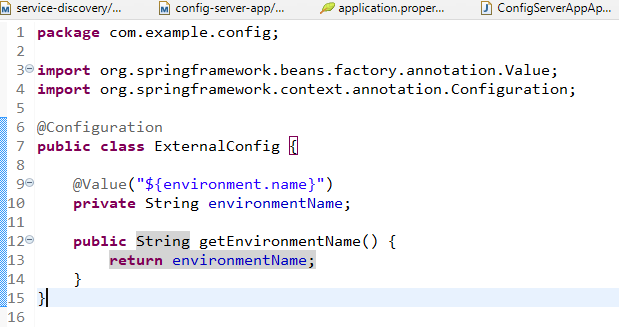


We have 4 configuration files

1. hello.yml
2. hello-dev.yml
3. hello-qa.yml
4. hello-prod.yml

All the configuration has environment.name which you can use in this application at runtime, for that you need to create a @Configuration class that can use the value of environment.name

ExternalConfig.java



This class uses the value environment.name present in the cloud, it could be from any file like hello.yml, hello-dev.yml, hello-qa.yml, hello-prod.yml, but by default it uses hello.yml

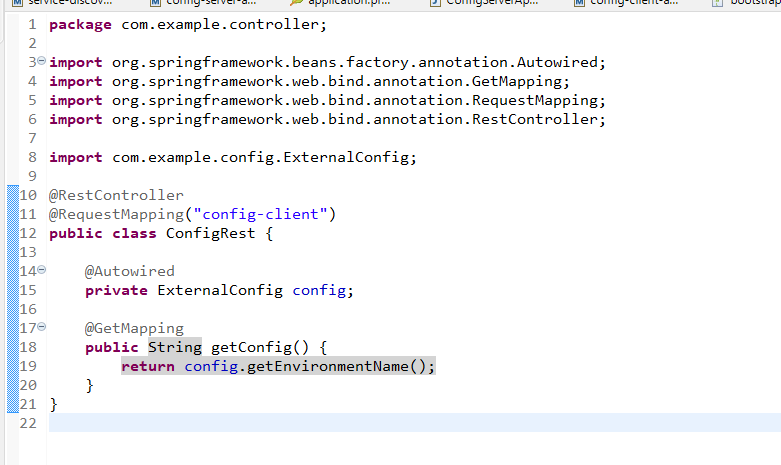
Note: If you want to use the property from other files then you must give the profiles.active, it can be provided at runtime during the application launch

i.e., java -jar file\_name.jar --spring.profiles.active=dev

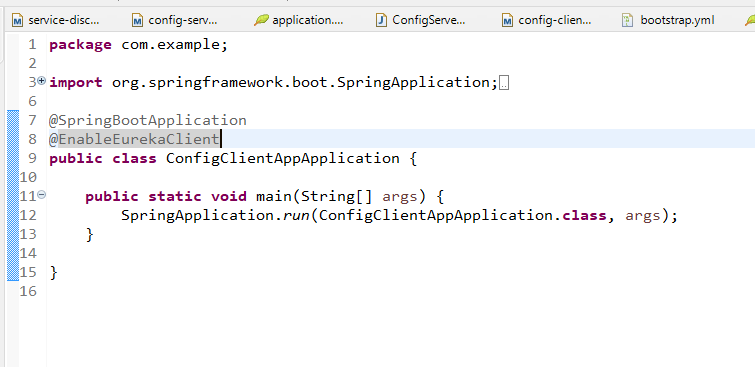
So the application would fetch the hello-dev.yml, since hello is already part of the application.name it is understood

We can use the RestAPI to show what value is fetched

ConfigRest.java



Use @EnableEurekaClient as we have this library in our classpath



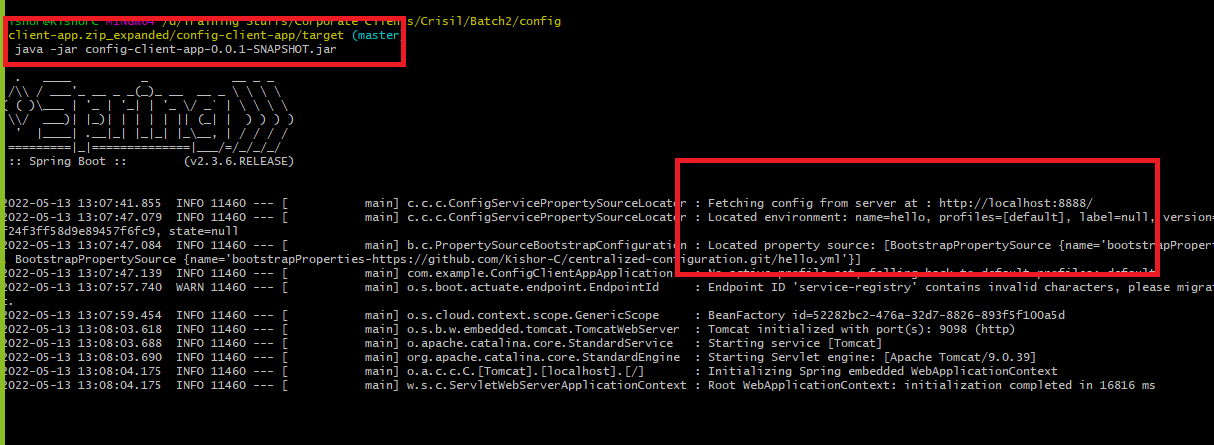
Now if you use the endpoint config-client then you must able to see the properties fetched from the cloud

Note: You can also see which property file is loaded by the client when it is run

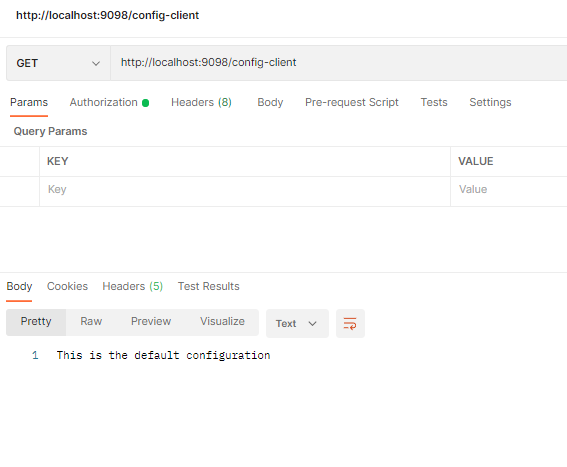
We need to run 3 programs here

1. service discovery
2. config server demo
3. config client demo

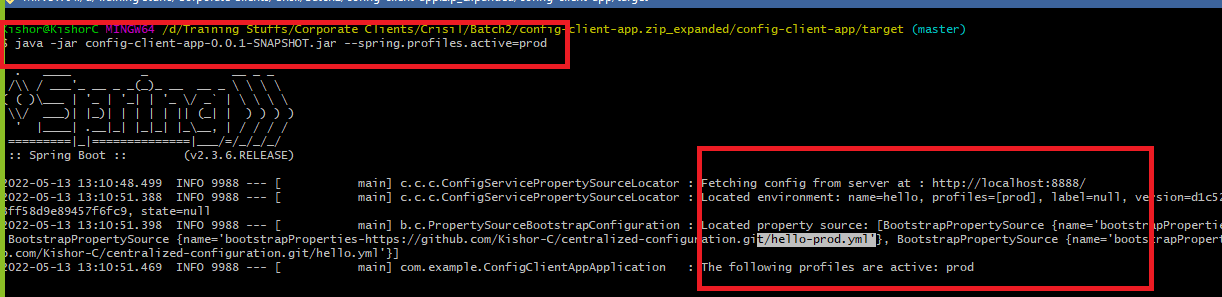
While running config client we can run with spring.profiles.active property to load different configuration files



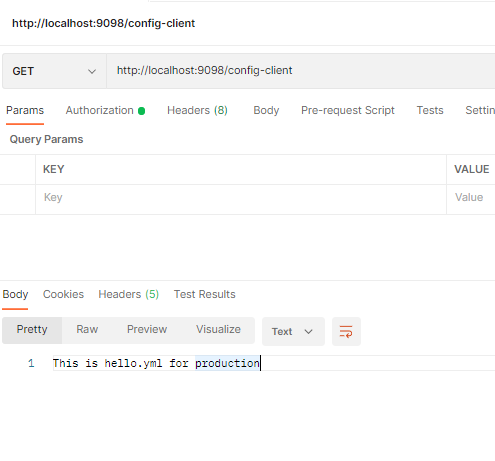
It fetched hello.yml, so it must show the environment.name of hello.yml



You can fetch other configurations through profiles.active



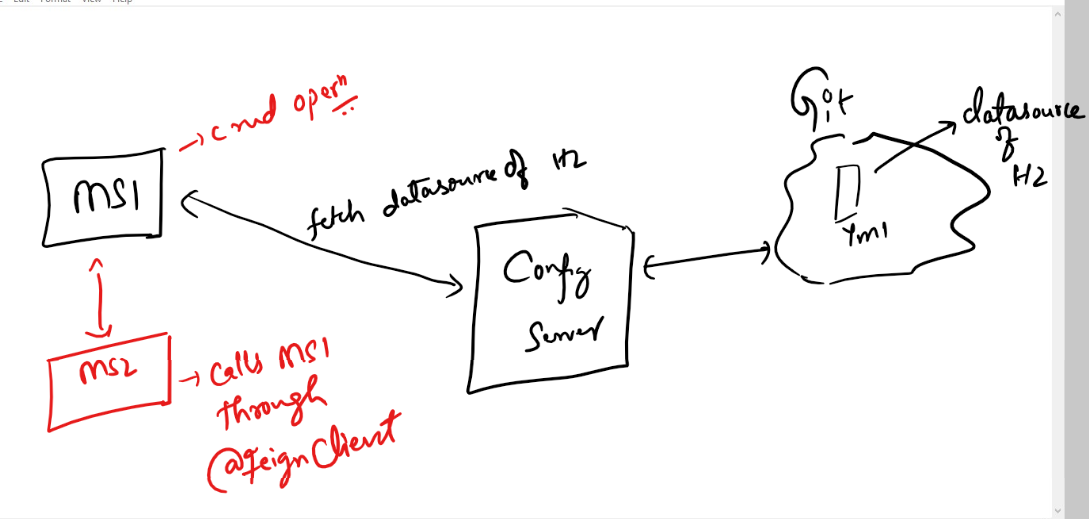
Output:



Activity:

1. Try out Feign Client example
2. Try out the Centralized Cloud Configuration example
3. With the Centralized configuration, store the datasource information of H2 Database in the GIT, and microservice must able to fetch this datasource information and perform some CRUD operations
   * add all the necessary libraries in the microservice to interact with the database
4. Create another microservice and through Feign Client call the CRUD operations implemented in the another microservice i.e., config client
   * Use at-least 2 or 3 HTTP methods like POST, PUT, GET

3rd and 4th activity explained in diagram



Duration: 1 hr 30 mins