Spring Boot & Spring Cloud Microservices

Pre-requisites

1. Java Technology: OOPS & Collection Framework
2. Spring Framework: Dependency Injection, Spring MVC & Annotations

Spring Boot:  
It simplifies developing the spring applications by taking care of all the generic configurations

i.e,

- You don’t have to write any configuration file like xml files

- No need to setup because the server is added into the application itself

- Set up required for applications are provided by spring boot starter projects

Starter projects in Spring boot

* Spring Boot Starter Web: It automatically configures setup for web development like Front Controller, Server Configuration (Embedded servers), Component Scanning
* Spring Boot Starter JPA: It automatically configures the dependencies required for Databases like Connections, Connection Factories, Templates (JdbcTemplate & HiberanteTemplate)

For Spring Boot we need following software’s to be installed

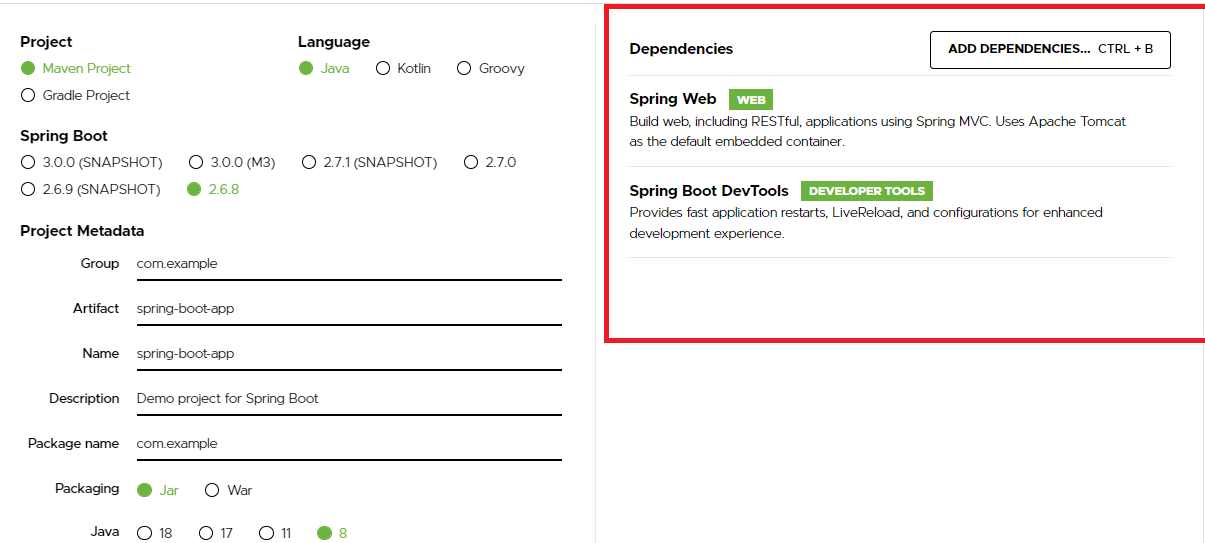
* Java
* Eclipse IDE / STS
* Postman App

Spring provides us a website to set up the spring boot project i.e., Spring Initializr

Creating our first web application in Spring Boot

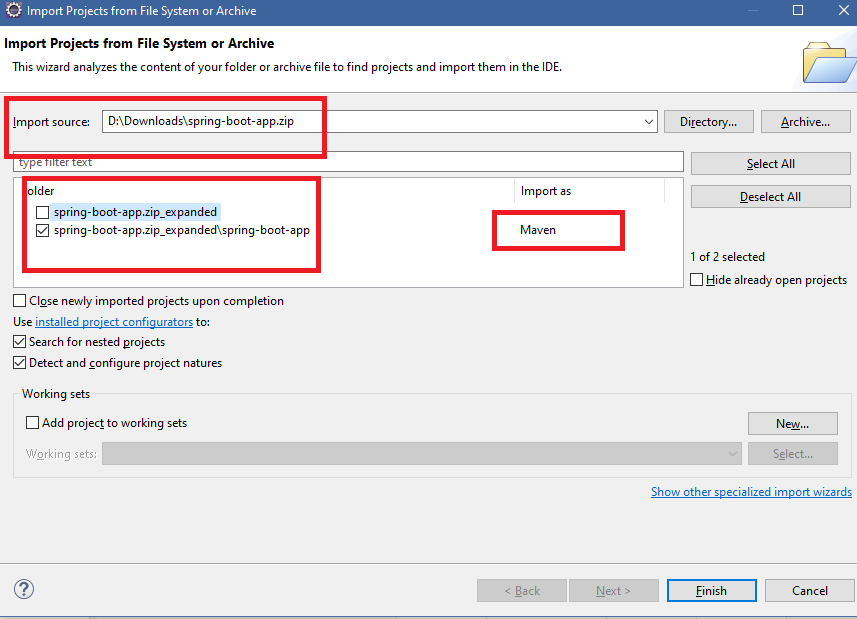
Dependencies required are:

* Web : Provides configurations for web applications like Front Controller, Server, Component Scanning – we get everything auto-configured for us
* Devtools (Optional) : auto-reload feature

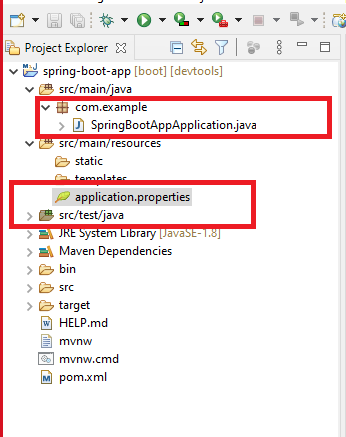


Click on Generate so that it downloads the zip file of the project, then you need to open this project from the Eclipse

Open project with Maven selected in the eclipse



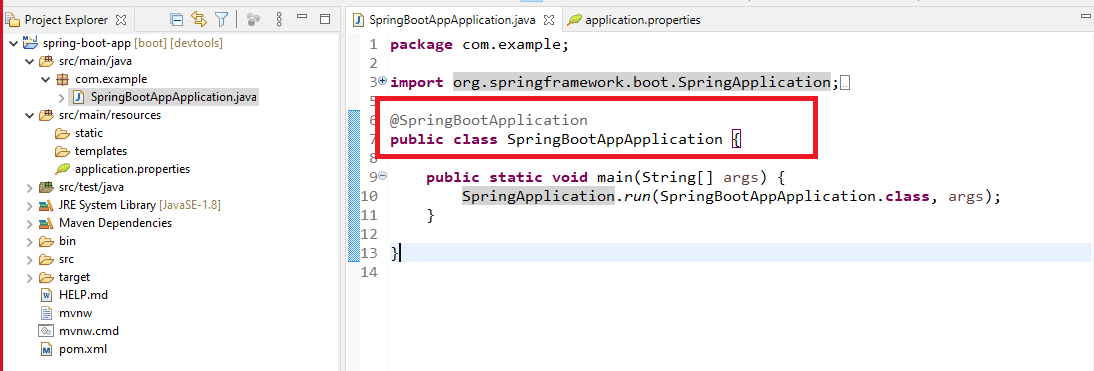
When we see the project structure we can observe two files



The Java file is the entry point file to launch your application, the application will be launched in the embedded server (Tomcat)

The application.properties is the configuration file for your application which keeps all the application related configurations like data-source informations, server port information’s, microservice related configurations

Note: Spring Boot uses application.properties as the default configuration file, but you can change this name also and load a different configuration file

The Java file looks like below  


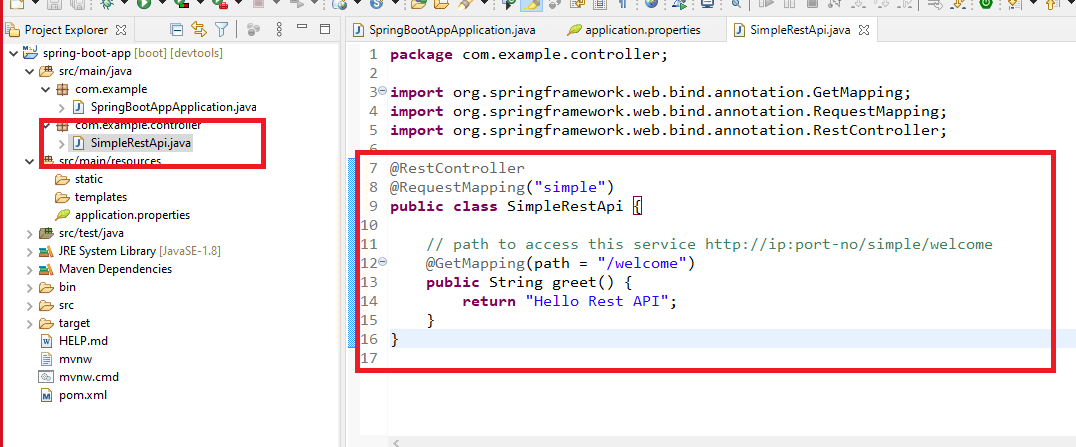
@SpringBootApplication:

* It takes care of doing all the auto-configurations for the application based on the library we add in our classpath
* The class having this needs to be loaded so that an application which is fully configured will be running for use.

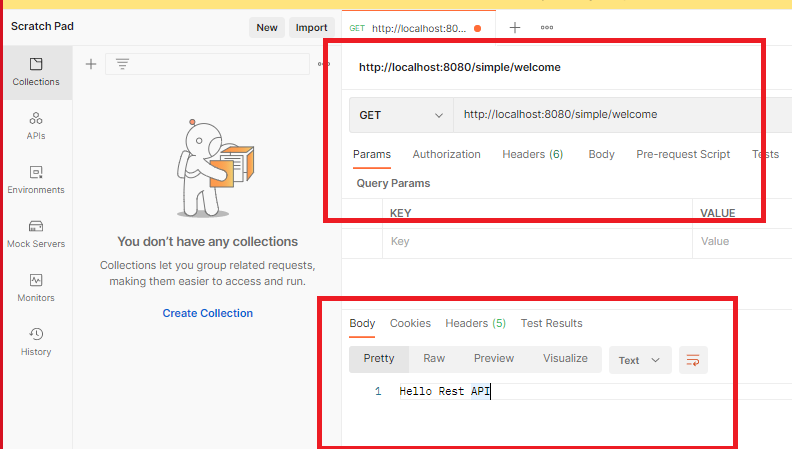
Since we have a web starter added we can create webservices in the application so that different applications can consume the data.

In Spring Framework we can create webservices using @RestController on top of the class

SimpleRest.java

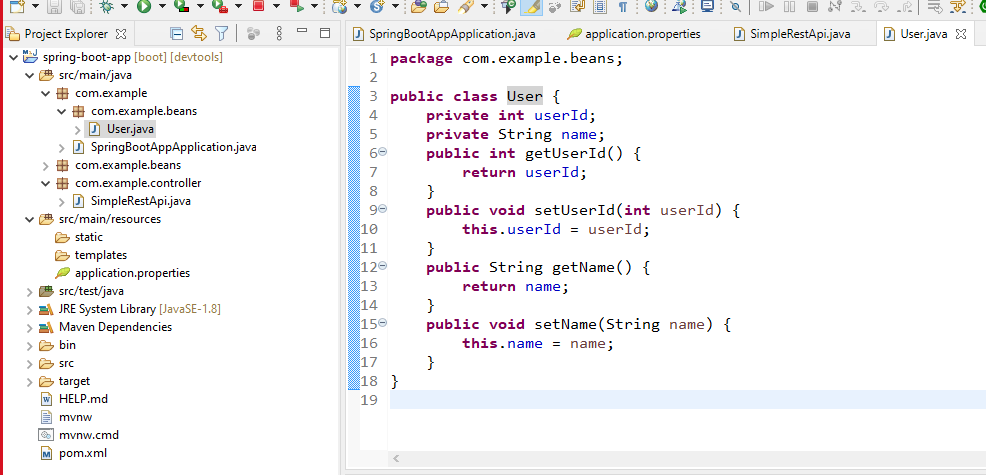


Now we are ready to run this application, since it is run in an embedded server the default port will be 8080 and the server is apache tomcat, but we can change all these configurations if required.

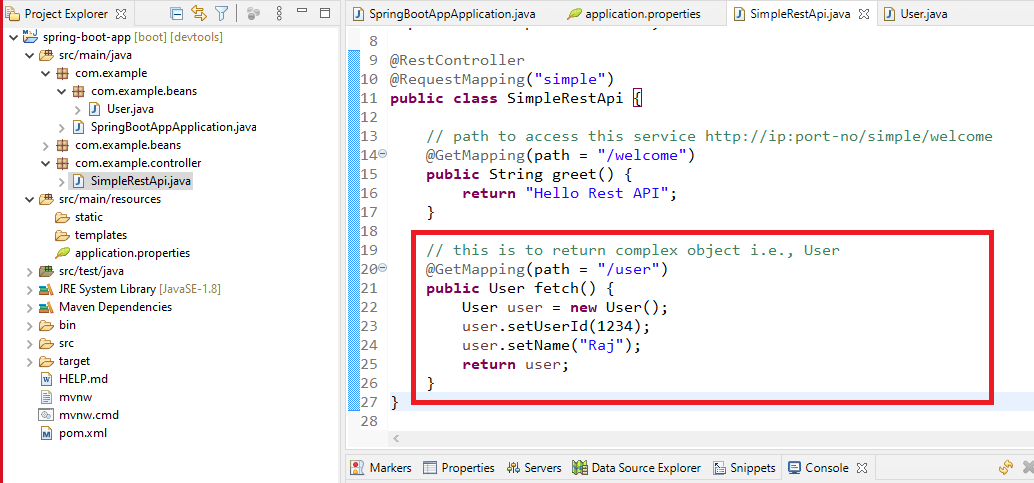


Here the response data is simple type hence it is coming in text format, however if the data is an object then it comes in JSON format.

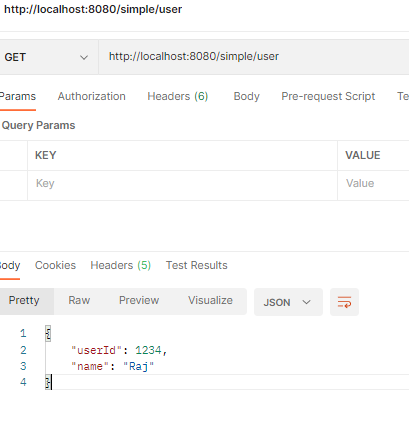
User.java



Now the Rest service can respond in json format if it returns user object



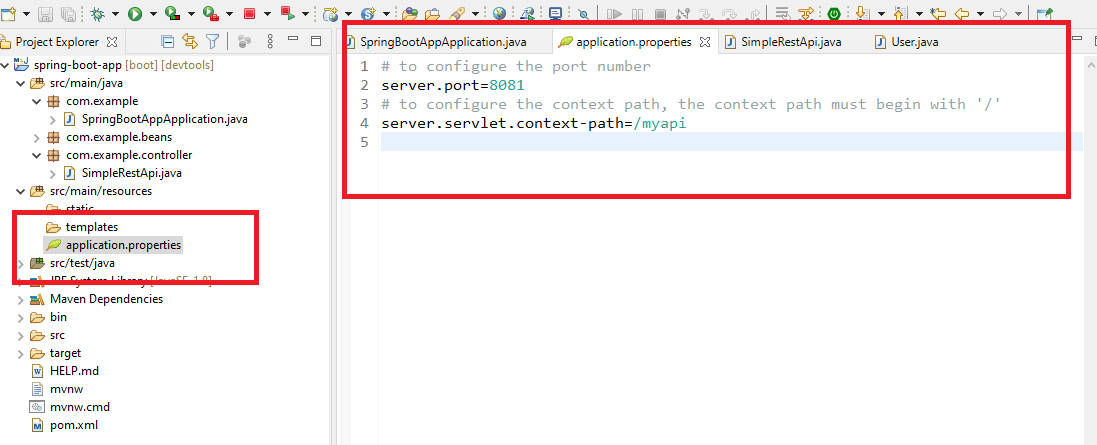
Output:



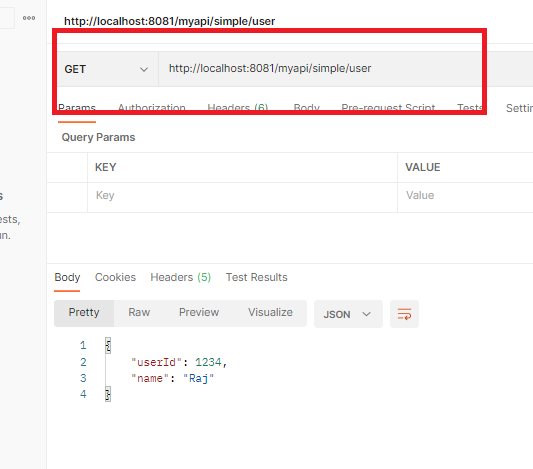
Configurations that needs to be done for the application

* How to add the context path to the application i.e., the root path to access the application
* How to configure the port number
* How to configure data-source information

All these configurations we need to do it in the application.properties

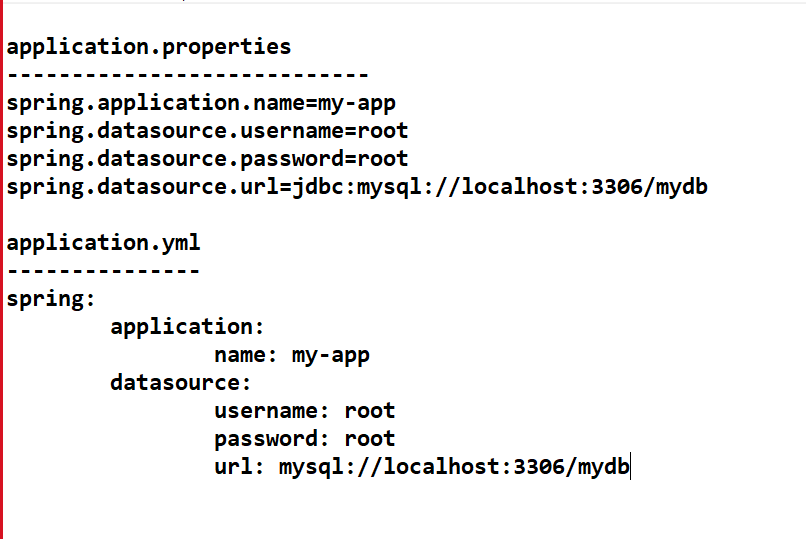


Now we must able to access the application using /myapi

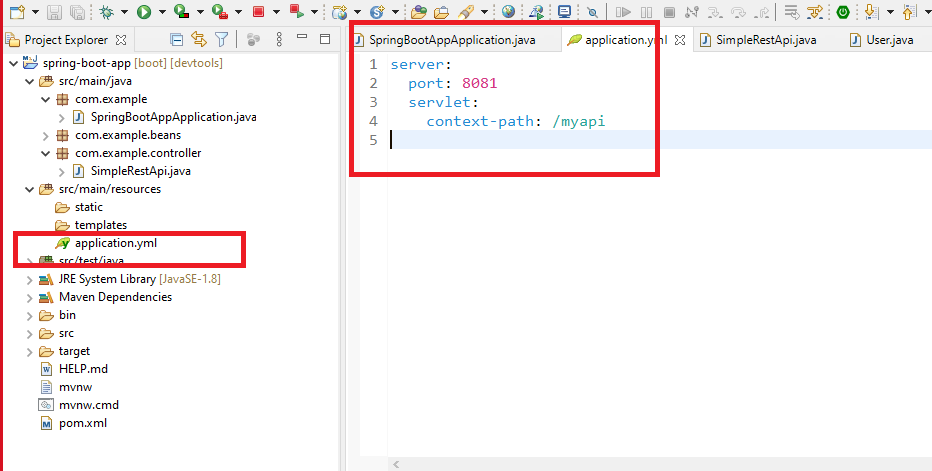


YAML Files:

These are alternate files for property files, they use indentation to mention the properties & sub-properties so that you can avoid writing property names repeatedly.



In Eclipse you will get plugin to convert properties to yaml files



Note: In Eclipse the plugin is taking care of providing the indentations to the existing properties, however if have yml already & want to add properties then we must take care of providing the indentation.

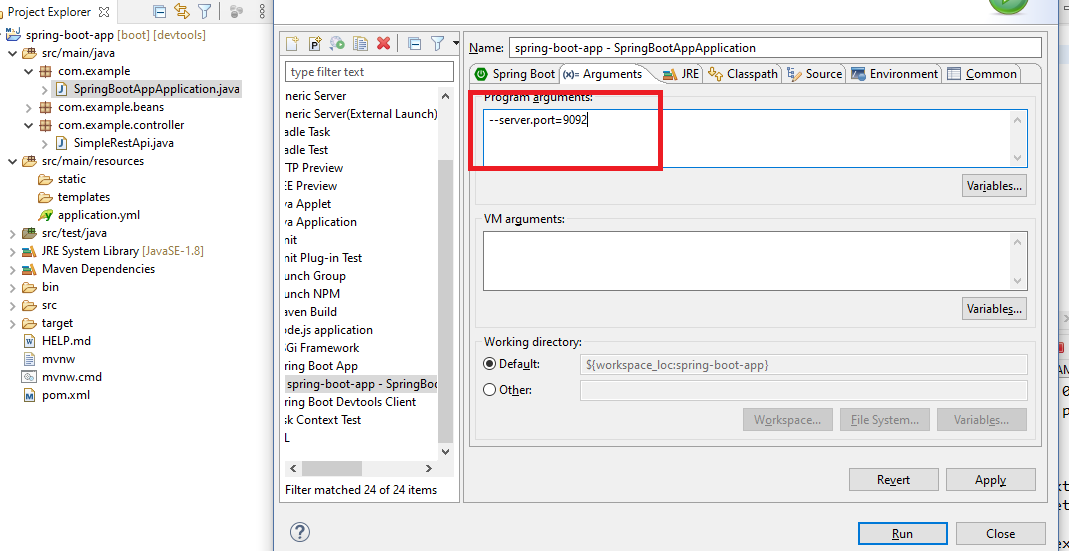
Overriding the properties

We can override the properties mention in the application.properties/yml while running the application.

* You can launch the application in eclipse and change the command line arguments
* If you have a deployable jar file then you can use command line arguments in the terminal

How to launch the application in eclipse by changing command line arguments

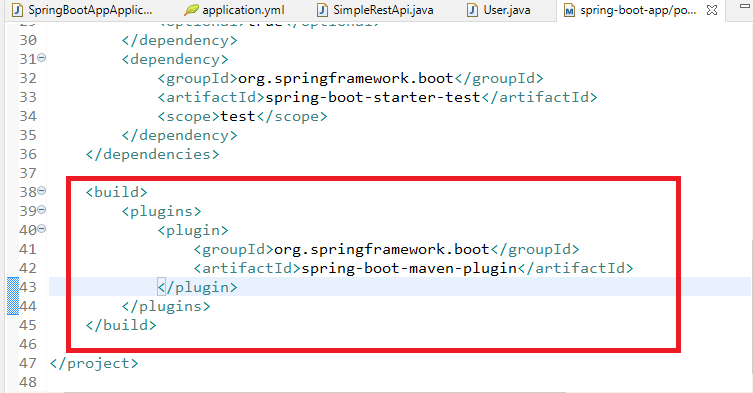
We need to use the properties with two hyphens followed by properties separated by dot



If in case you have a deployable jar then you can use the same arguments at the of time running

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

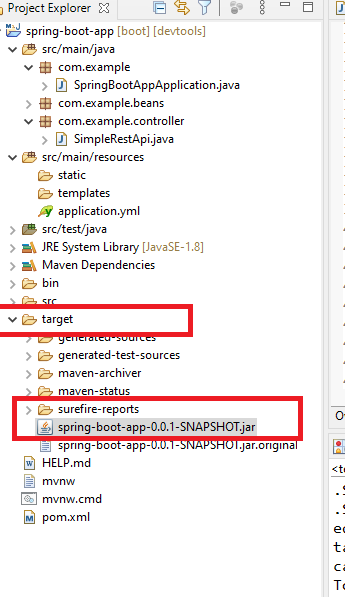
We can use maven built to build the deployable jar file, In Spring Boot its already configured to build a jar which will be an executable jar because of the entry in pom.xml



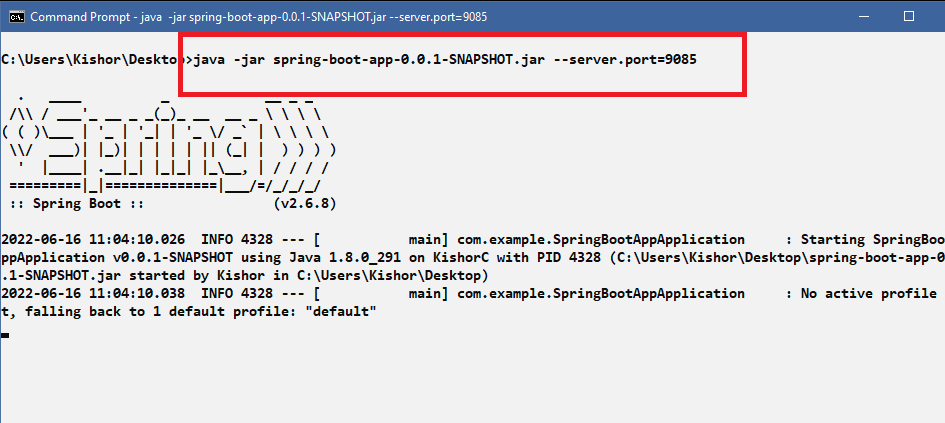
Note: This entry comes only we create project from spring initializr

How to build the jar from eclipse

Project -> Run As -> Maven Built -> Goals: package



Note: refresh the target folder to see the build file, we can run this jar using java command now



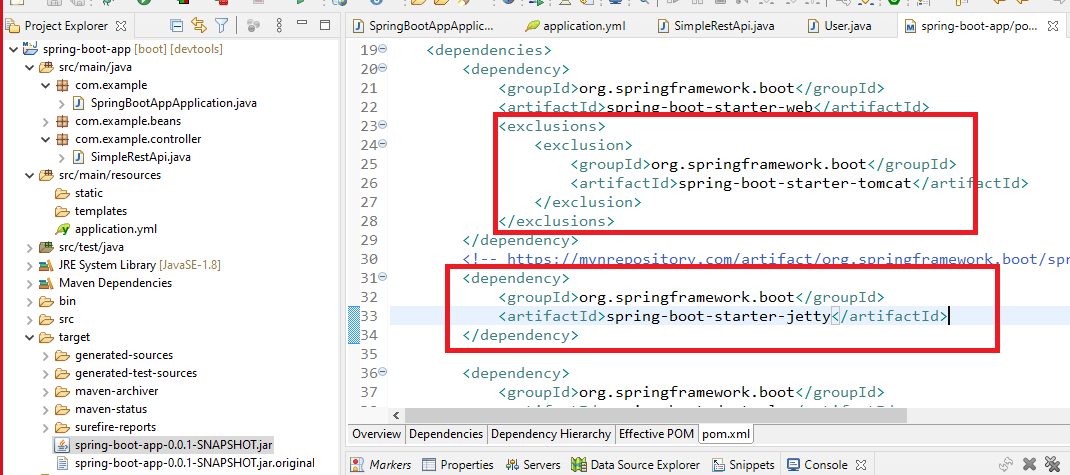
Spring boot provides following Embedded servers

1. Tomcat
2. Jboss – Undertow
3. Eclipse – Jetty

To configure different servers you need to exclude the default container tomcat and add different embedded server.

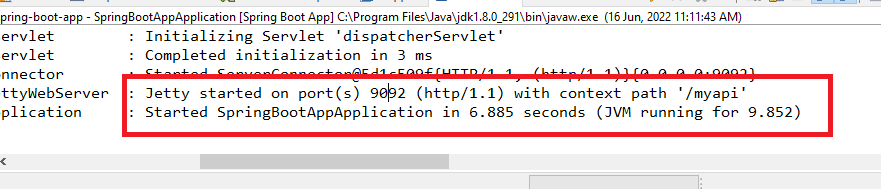
Steps

1. Exclude the embedded tomcat dependency
2. Add the other embed server you want to use



Output:

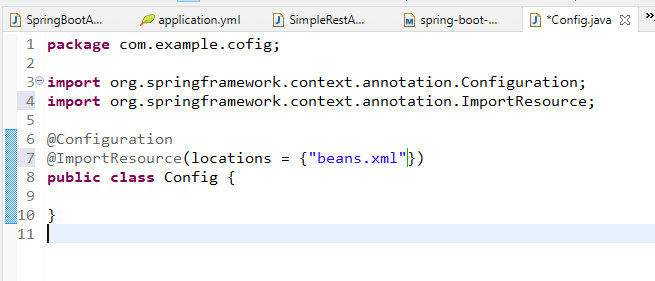
We must see in the console the jetty server has been used instead of tomcat



By default spring boot applications are packaged as jar, however if we want to deploy in our own external server then we need to package as war.

Can spring boot load XML files and use the configured dependencies in the application

Spring boot provides @ImportResource annotation which accepts the location of xml files so that the dependencies of the application can be used



Here if the beans.xml has <bean> configurations which registers in the spring container we can just load the xml files and use the registered beans.

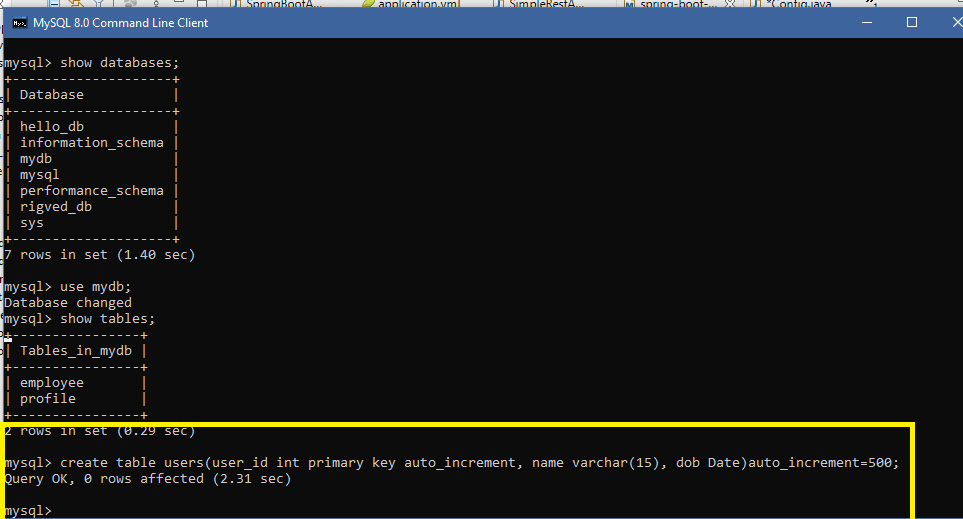
Spring Boot Best Practices:

1. Build System – Maven or Gradle
   1. Dependency Management: No need to provide version for any list of dependencies
   2. Upgrading from one spring boot version to another version would be easier
2. Spring Boot Starters: Avoids lot of copy paste codes you do for some common tasks like database connections, dispatcher servlet, dependency injection
3. Packaging the application as an executable jar in the production & it works in cloud environment also
4. Referring the official spring boot migration guide to know about how to migrate old spring boot versions to new spring boot which gives you step by step solution whenever you want to migrate
5. Using developer tools to get a development experience because it re-loads the application when you modify the code & this is automatically disabled in the production.

Performing CRUD operations using Database, we can use either in memory database(H2) or the physical database (MySQL, Derby, …).

We have a spring boot starter to configure the data-source which is Spring Boot Starter JPA.

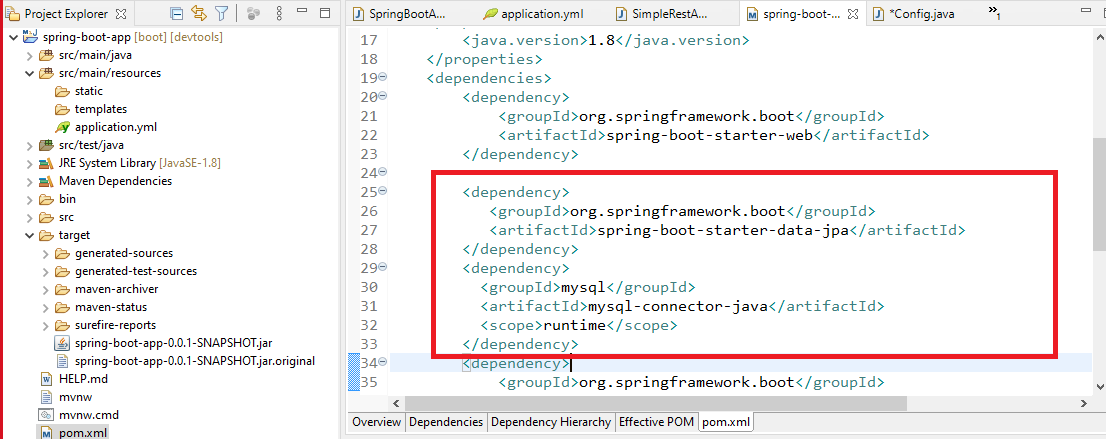
We can create a User table which will have user\_id, name & dob columns



Now we need an entity class that maps to this table and its column

We need to add a library to get all the relevant annotations for entity classes & also for database operations

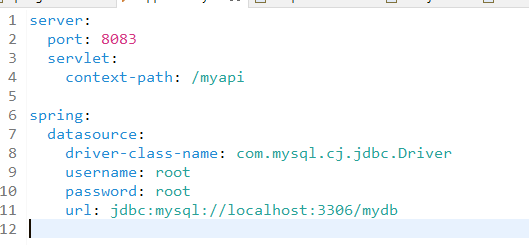
* Spring Boot Starter Data JPA



Note: If you don’t have database you can use H2 library that creates in memory database

Configuring the datasource information to interact with MySQL

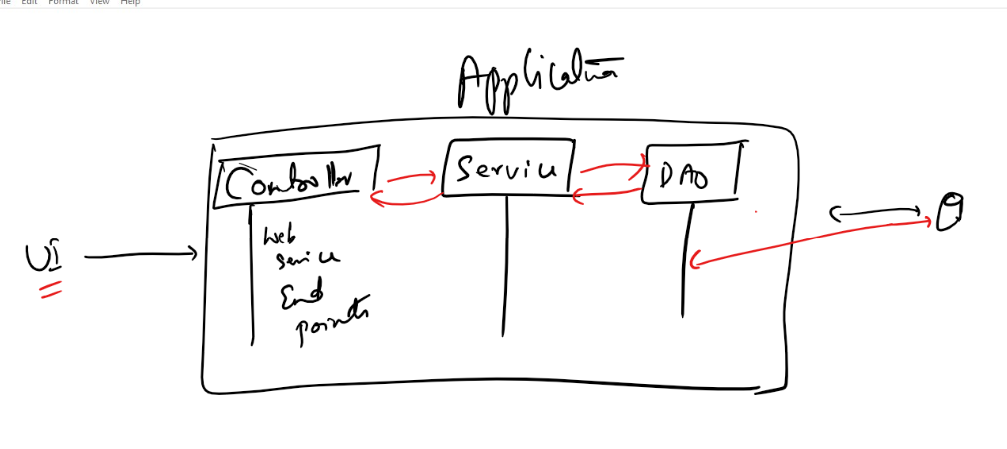
application.yml



If you use H2 then you can provide any values to the username, password & url

Note: When we have spring boot data jpa library already, then it always looks for data-source configurations in the property file, if we don’t have it then spring boot raises error.

Since we are interacting with the database, we need to separate the logics into multiple layers like Controller layer, Service layer & DAO layer

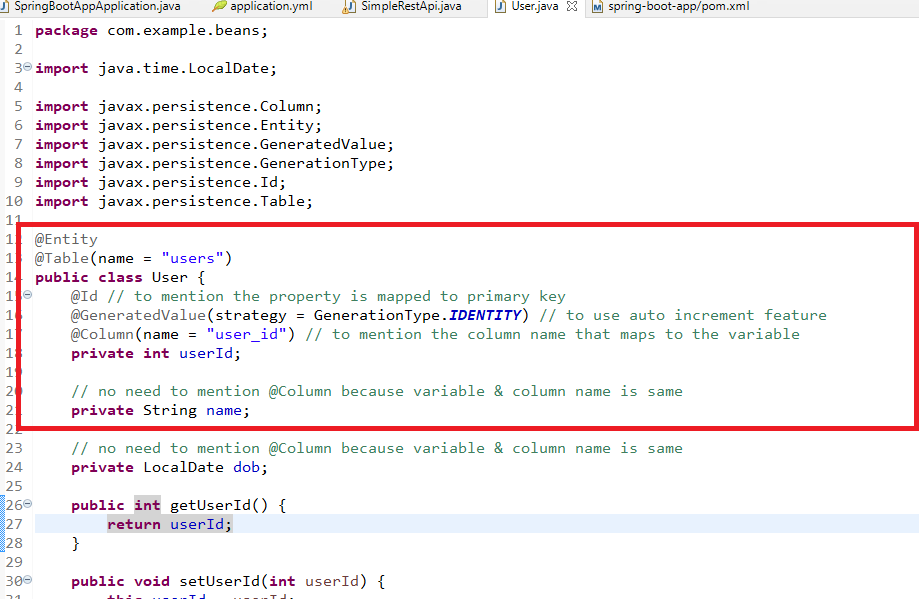
Controller will have web service end point url, It uses Service instance that will have business logics & Service uses DAO instance that will have methods to perform Database operations like Insert, Update, Delete, Retrieve.

Transactions are handled in the Service layer itself.

Since Controller talks to Service & Service talks to DAO we need to abstract the implementation of the Service & DAO and use only the interfaces in the client code.

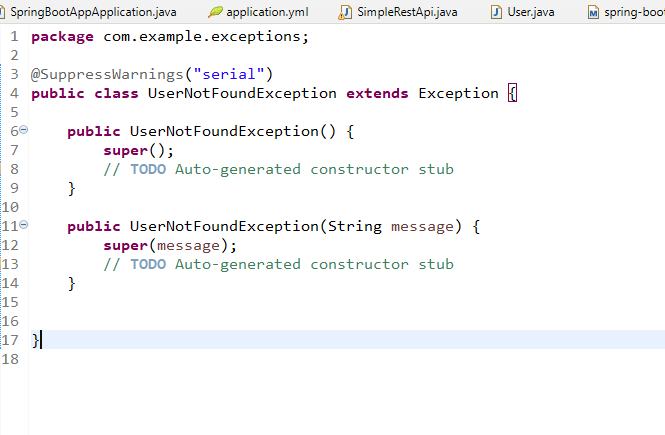
We can also create an Exception class to throw if any problem occurs so that user will know what happened.

User.java



Since it is interacting with the database and always we can’t expect to get a successful response so we can also create an Exception class that will be raised if any data we are searching is not found

UserNotFoundException.java



To interact with the database Spring Data Jpa gives you interfaces which are called as Repository interfaces which will have CRUD operations.

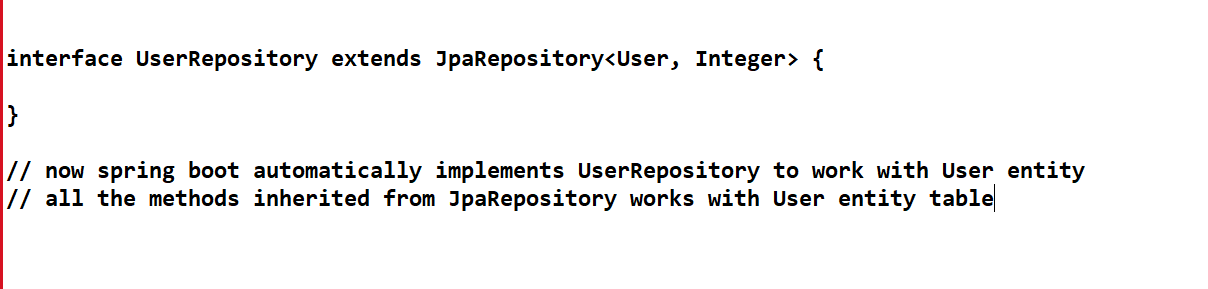
1. CrudRepository<T, ID>: Has methods to perform CRUD operations
2. JpaRepository<T, ID>: Extends CrudRepository and also provides methods for pagination & sorting

These two interfaces can dynamically perform the operation on the entity without any implementation, Spring Boot can implement these interfaces once it knows these repository is mapped to which entity class.

These repository interfaces like CrudRepository or JpaRepository has below methods.

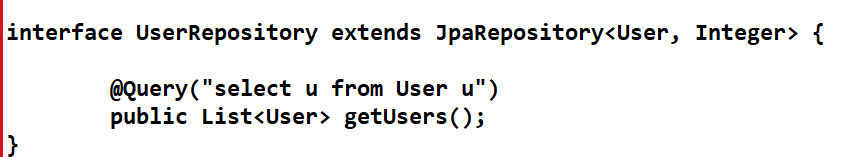
1. save(entity)
2. delete(entity)
3. findAll()
4. findOne()

All these methods are automatically implemented by Spring Boot

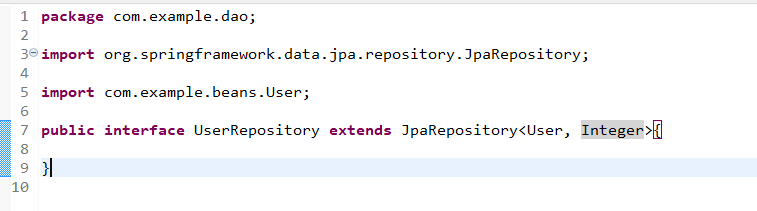


Now we need to just supply the implementation of UserRepository to the Service layer using @Autowired

Note: You can also create custom methods inside this interface which will also be automatically implemented.



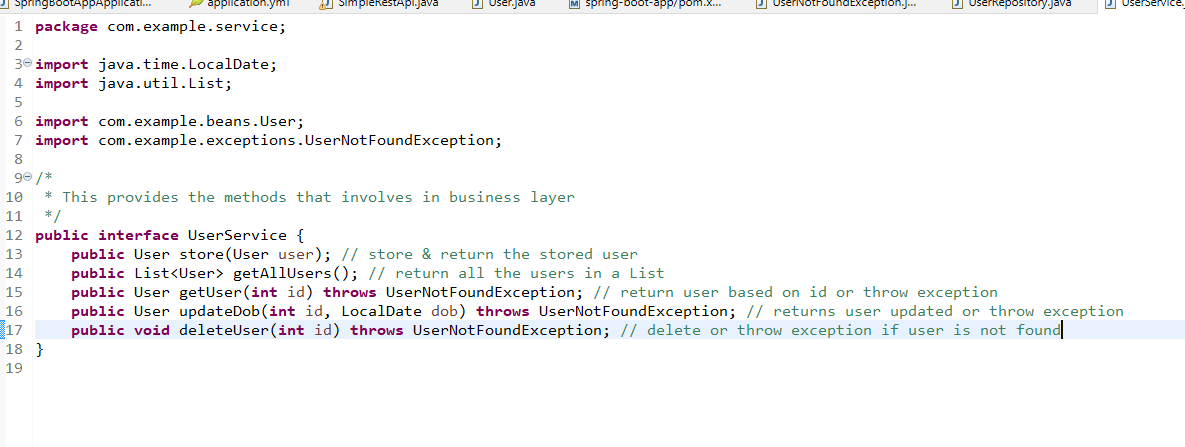
UserRepository.java



This UserRepository is automatically implemented and registered in the spring container, so we need to only supply the UserRepository in the Service layer.

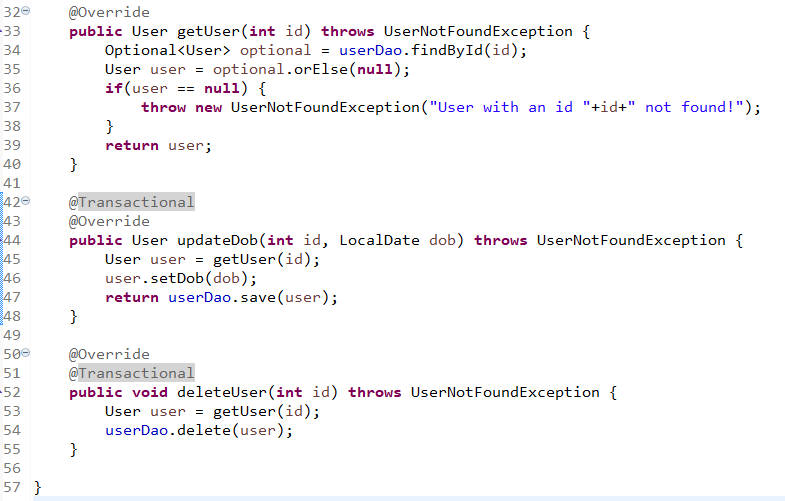
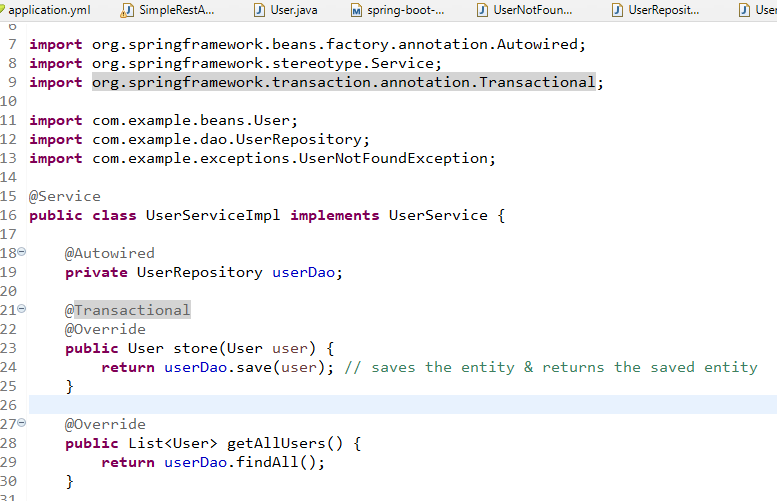
Firstly we need an interface in the Service layer that is used in the controller

UserService.java



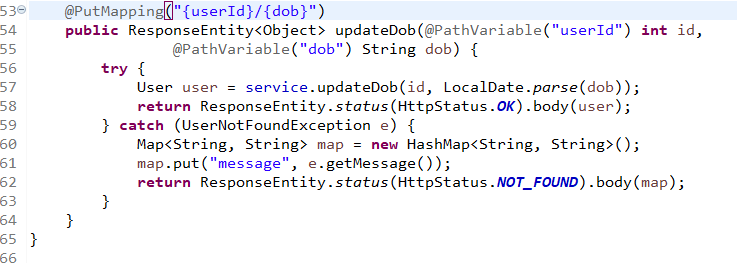
These methods we need to implement by providing an implement class that uses UserRepository

UserServiceImpl.java

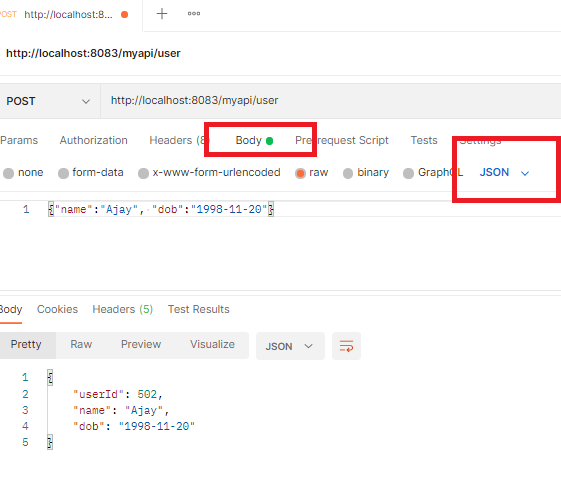


These methods are called from the controller

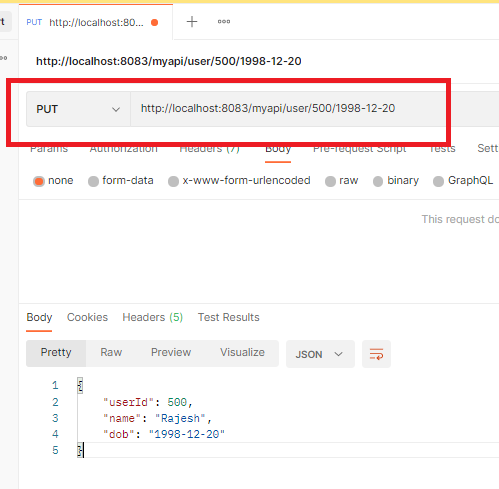
SimpleRestApi.java



Post method testing



PUT Method testing



In the Entity you can have an association mapping like @OneToOne, @OneToMany, where a single user may have one or more mapping of a particular object

ex:

User has an address

(or)

Order has one or more LineItem

(or)

Customer has one or more Order

Activity:

Create a layered architecture (Controller, Service & Dao i.e., Repository interfaces) which can perform customer to order items, the application must have a Customer who can place multiple orders and each order can have multiple line items

Customer will have: id, name, List<Order> (one to many)

Order will have: id, List<LineItems> (one to many)

LineItems will have: id, itemName, quantity, price

You must able to place one or more orders from the postman & also must able to view orders from the customers.

You will have 2 webservices

1. Place orders
2. View orders

Requirement

* Git on your local machine
* Git hub account from your personal mail id

Spring Cloud & Microservices

Pre-requisites

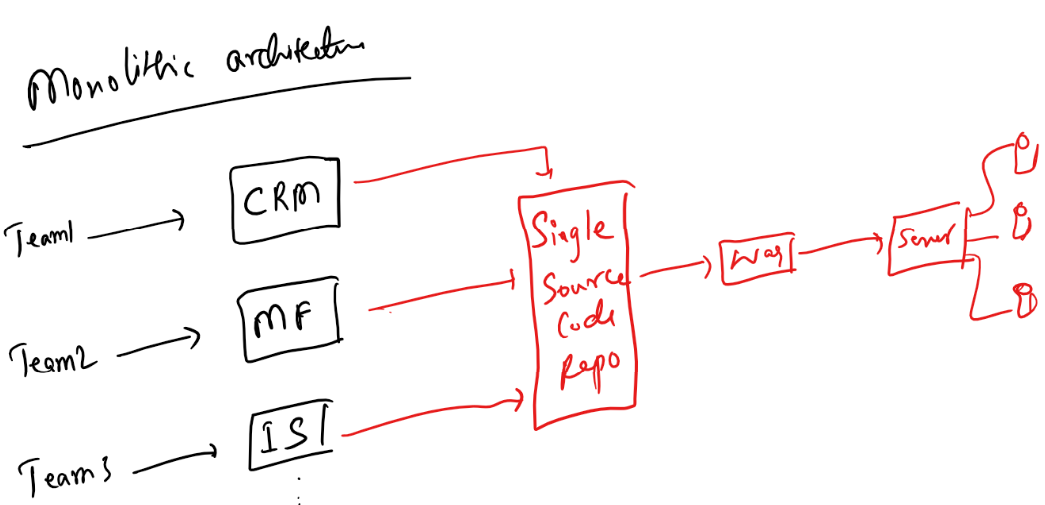
1. Spring REST
2. Spring Boot

Microservices

These are loosely coupled services which are independent from other services of same or different applications

Most of the application follow monolithic architecture

Monolithic Architecture



Consider a finance domain having CRM (Customer Relationship Management), MF (Mutual Fund), IS (Insurance Services) and other services which are tightly coupled because they all are packaged into a single artifact at the end

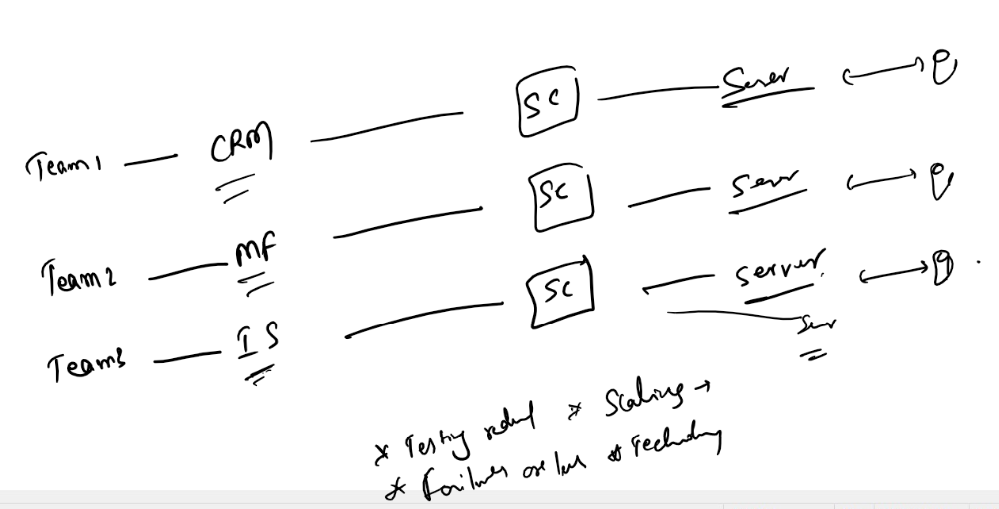
In a monolithic architecture there would be separate team to handle the responsibility of separate modules and it will be pushed to the single source code repository and everyone work will be collaborated & will be built as a single artifact & deployed in the production, the entire application will have knowledge all the datasources.

Challenges in Monolithic Architecture

1. If any module needs to be updated we need to ensure other modules don’t break because all goes into single build, so we need to test other modules that are not modified as well.
2. Failure in any service or the system may lead to entire service not available as all the services are packed as a single build & deployed in the same server
3. If the demand increases to one of the module then we may not able scale only that particular module, we may need to scale the entire application
4. We need to implement all the modules with the same technologies, we can’t use other technologies or resources to implement all the modules

To provide solution for these challenged Microservice architecture is introduced

Microservice architecture



The microservices allow you to independently create services that can be developed & deployed separately, Benefits

1. Testing will be reduced as all the services are pushed to separate source code repository & built as separate artifacts & deployed independently, we don’t need to test other services
2. Failures will be less, because all the services are independently running in separate production environment, if any one services goes down other services are not affected & they will be available
3. Scaling can be done only to those services which are in demand so that the entire application need not to be scaled.
4. We can free to implement single application in many technologies so that we can utilize the resources in the organization.

Microservice makes the applications loosely coupled & also the services present in the application.

Challenges in Microservices

1. Costlier
2. Important to identify the right approach is implemented
3. Performance overhead as services will be running on cloud
4. Adapting to changes
5. Operational complexity is higher

Microservices follow certain design principles to make the services loosely coupled & independent as much as possible

1. Service Discovery
2. Discovery Client
3. Client Load Balancer
4. Fault tolerance – Circuit Breaker
5. Distributed Configuration
6. API Gateway
7. Distributed Log Tracing
8. Security

Spring Microservices

It is a module that helps us to create microservices, it uses 2 project of spring

1. Spring Boot: To quickly create production grade spring applications
2. Spring Cloud: Gives us tools & libraries to build distributed applications

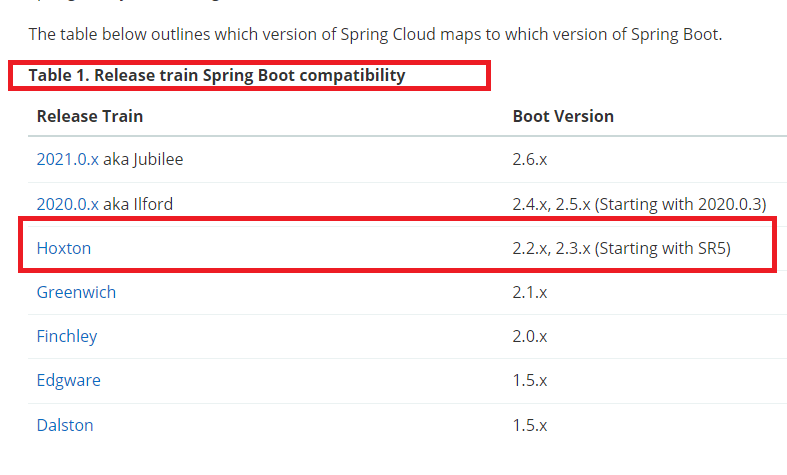
Spring Cloud: It provides us a simplified approach to develop microservices with all the microservice design patterns with simple annotations

It provides following features for microservices

1. Service Discovery – Eureka Server
2. Discovery Client – Eureka Client
3. Client side load balancer – Ribbon
4. Circuit Breaker – Resilience4j
5. Distributed Configuration – Cloud Config Server & Config Client
6. API gateway – Zuul
7. Distributed log tracing – Sleuth & Zikin
8. Security – Oauth2 & JWT

Note: Since spring microservice use spring boot & spring cloud these modules must be compatible as per the release train given by spring

The release train tells us which spring boot version is mapped with spring cloud version, it is better to take the stable releases here.



Note: In Spring Initializr we may not see Spring Boot 2.2.x or 2.3.x, we may need to change it manually to make it compatible with Spring Cloud stable release either Hoxton or Greenwich or Finchley

Creating Service Discovery

We can use Spring Cloud library Eureka Server to create the service discovery which provides us an annotation @EnableEurekaServer to create the service discovery

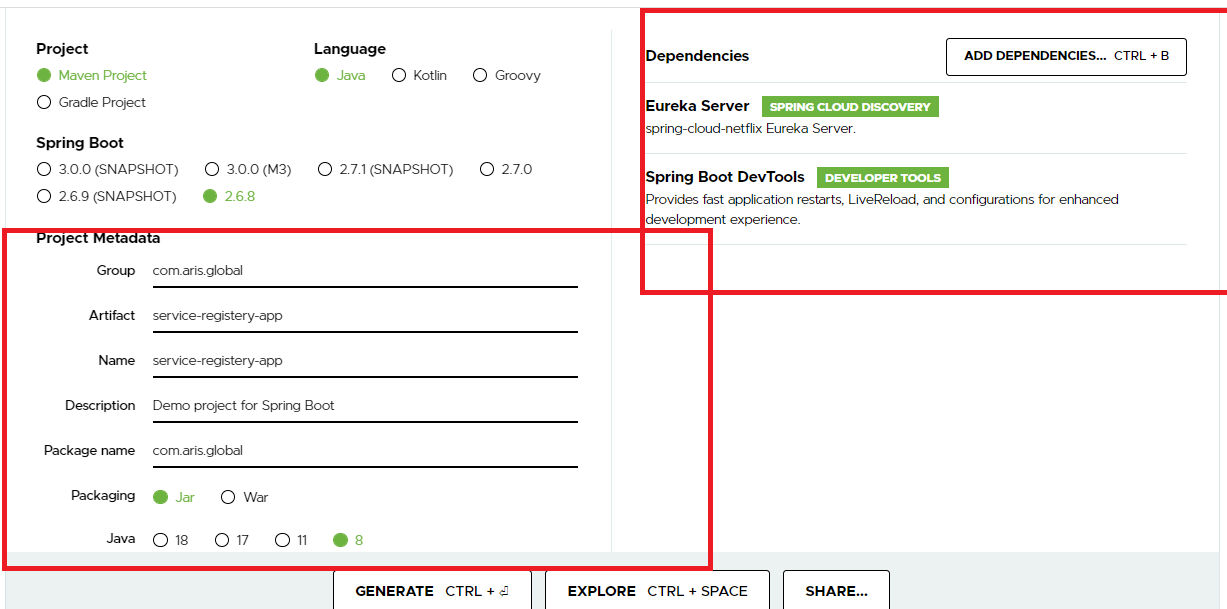
Role of Service Discovery

* It helps to register the microservices
* It helps clients to locate the microservices with application name (logical name) instead of the physical address (clients don’t use physical address of the microservice)
* It will automatically remote the instance of microservices from its registry if the microservice doesn’t ping its status to the service discovery

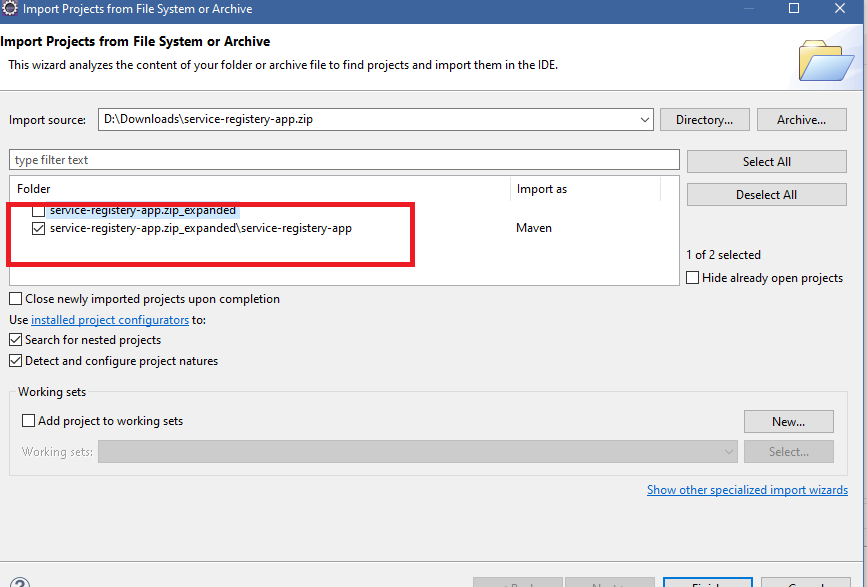
Note: Service Discovery acts like a registry we may not create any microservice in it

Dependencies for creating Service Discovery

1. Eureka Server
2. Dev tools (Optional)



Open the project from the eclipse

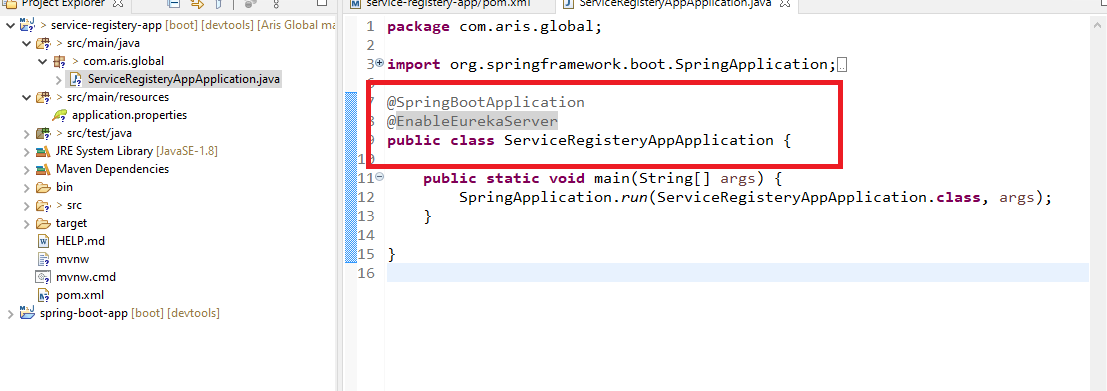


Note: Change pom.xml to make spring cloud & spring boot versions compatible



Ensure you will save & wait for all the dependencies to download & once download completes you need to update the project

We need to create this application as a service registry application hence we must use @EnableEurekaServer in our entry point class



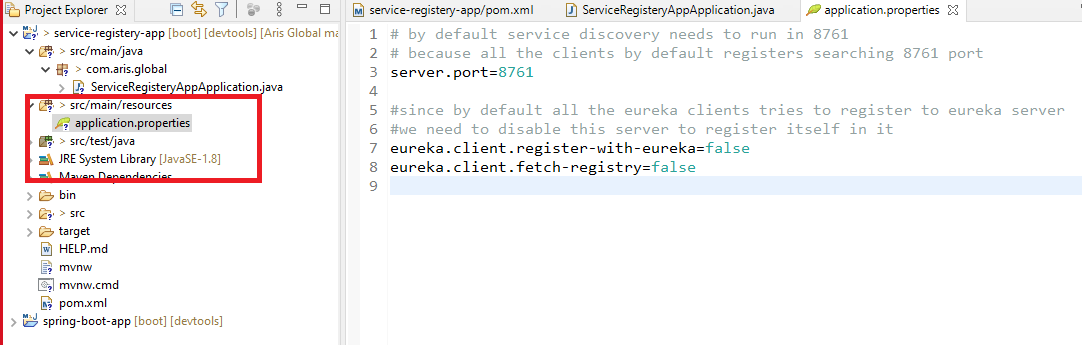
Though we get @EnableEurekaServer & @EnableEurekaClient we need to add @EnableEurekaServer

Note: Eureka Server library downloads server & client library both, because of this spring boot autoconfigures this application as server & client both

How to disable the client feature

We need add a few properties in the application.properties to disable the client feature

Note: By Default all the microservices registers to service discovery automatically by searching service discovery in the port 8761, hence we run service discovery in 8761



Now the Service Discovery is ready to launch, we will get a dashboard to see all the registered services i.e., Eureka Dashboard.

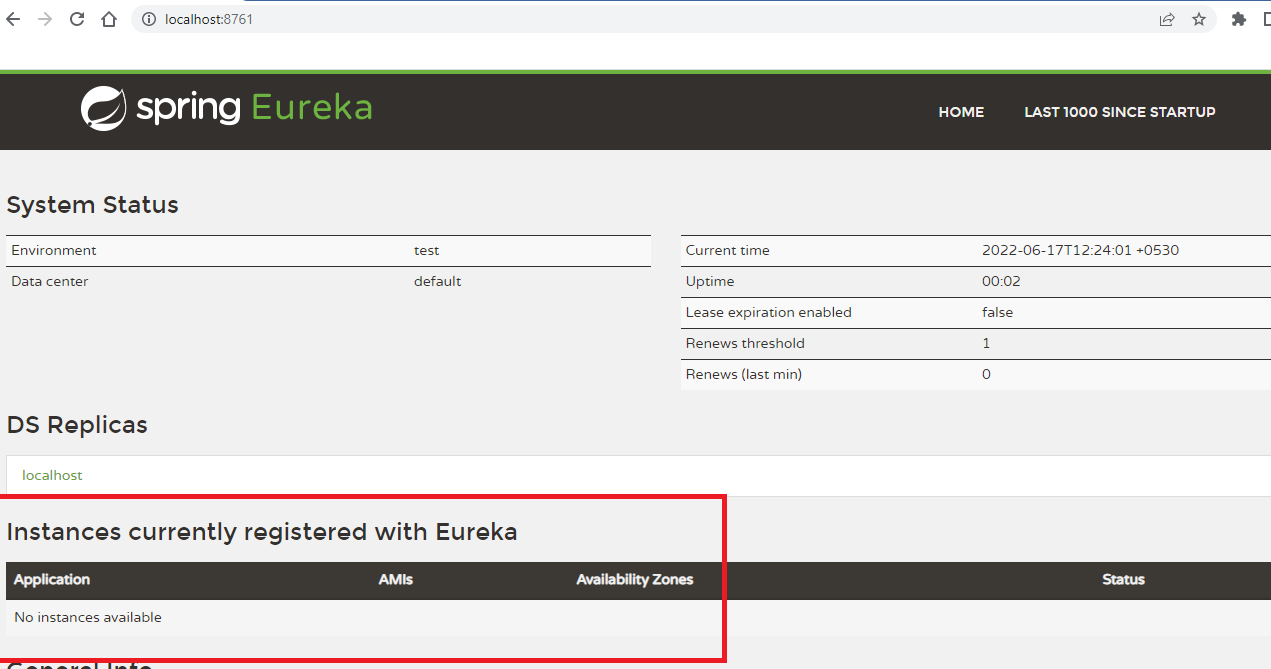
Build the application into an executable jar & run it in the terminal

we can use java -jar filename.jar



We can access eureka server in 8761

<http://localhost:8761>



Creating Microservices

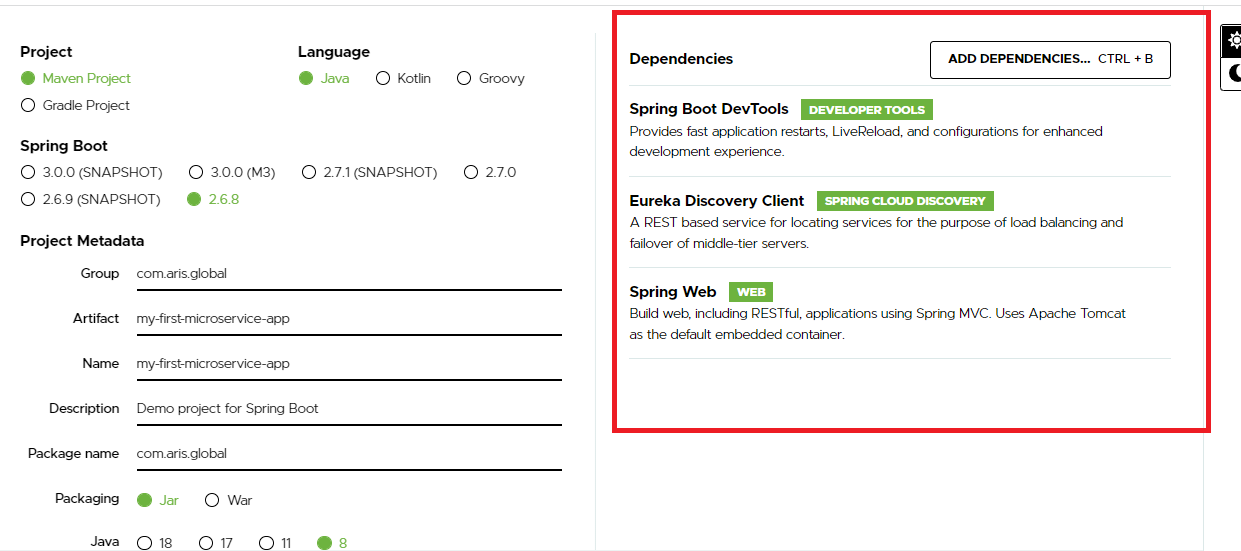
We can create clients which are microservices that get registered in the service discovery

Dependencies required

Eureka Client

Web

Dev tools



Note: You need to change spring boot & cloud versions to be compatible in pom.xml as mentioned per the release train



Update the project after you enter these in the pom.xml

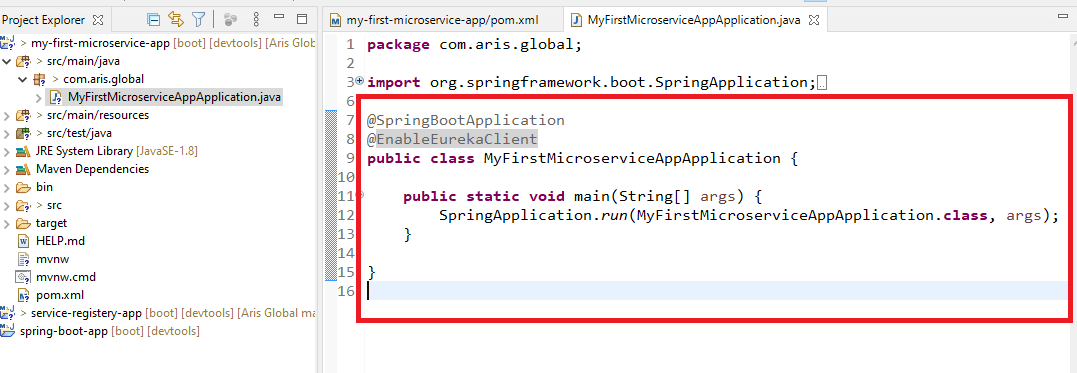
Features of microservices

1. They will all the applicated related layers like Controller, Service, DAO layer
2. They will have a logical name i.e., an application name that is registered in the Service Discovery
3. It uses all the design patterns necessary for microservices
4. They can also act like client to consume other microservices, but they must not use physical address of the microservice, they must use logical name of the microservice registered in the service discovery.

To create our application as a microservice

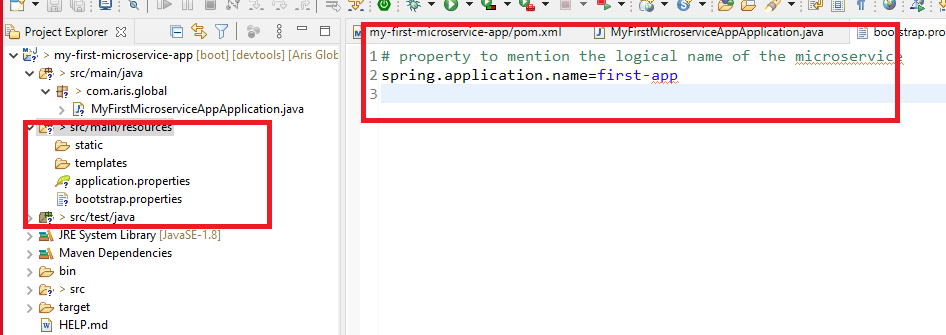
@EnableEurekaClient is the annotation used to make our application as a microservice, this annotation takes care of all the things a microservice should do like

* Registering to the service discovery
* Sending heart beans to help service discovery to identify its status

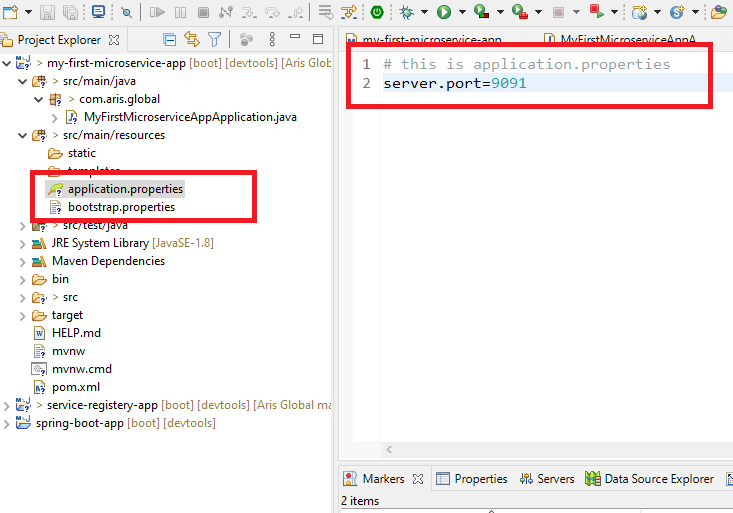


We can give an application name to the microservice so that name is used by all the client microservices

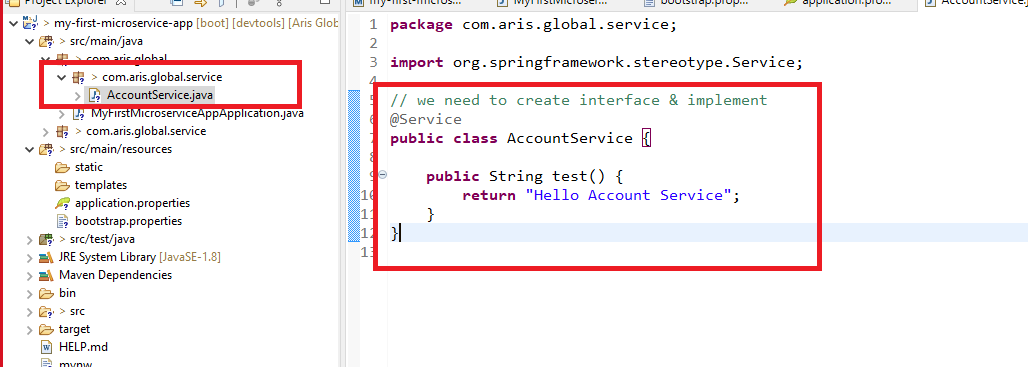
bootstrap.properties: This is the property which is loaded before any other property files, its loaded even before application.properties, and it is loaded at the time starting the application



application.properties

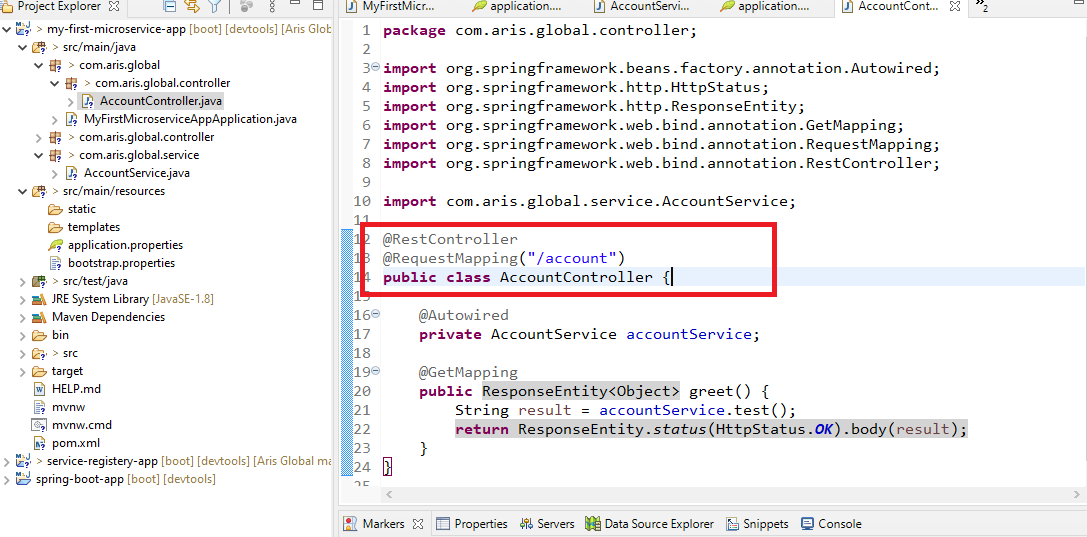


Creating a Service layer



Now we can create a RestController class that calls the service layer methods

AccountRestApi.java

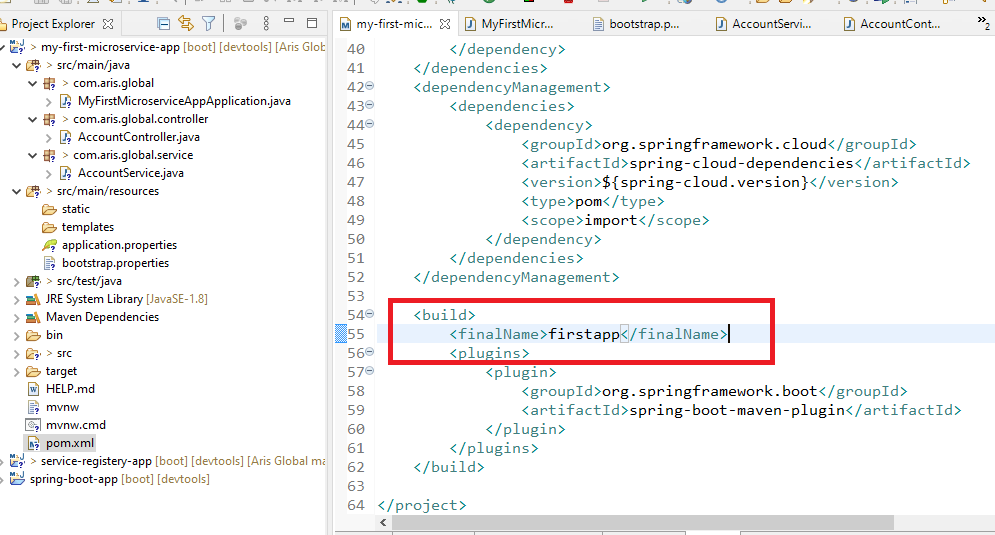


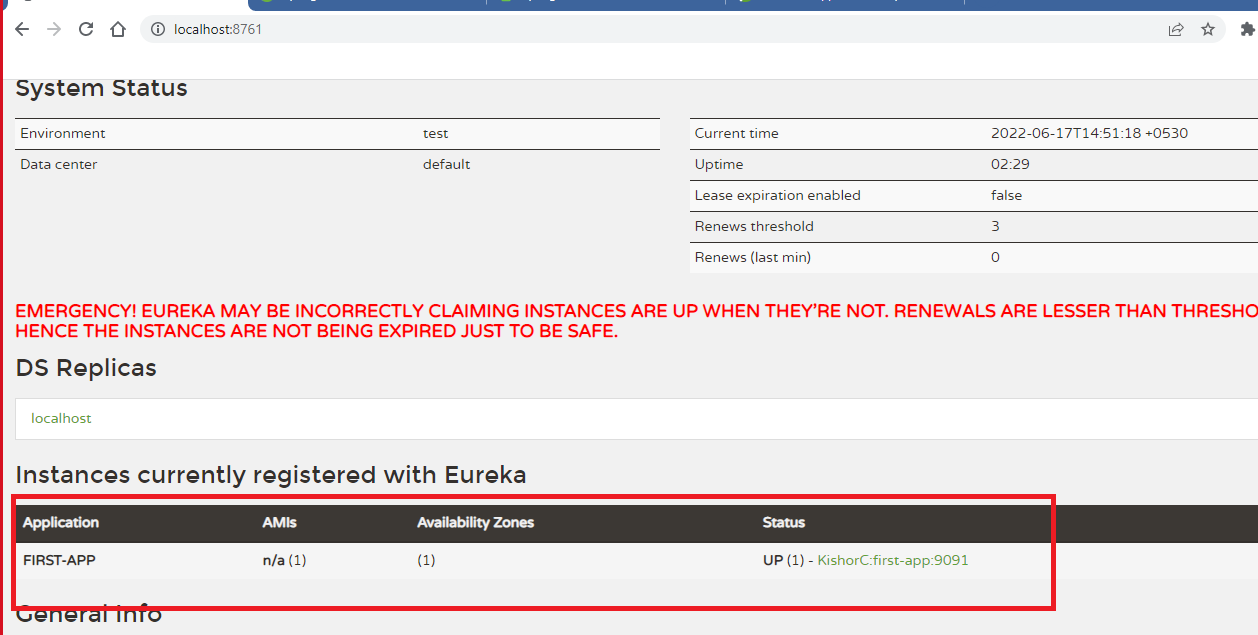
Here we have a REST end point /account & this application runs in port=9091 and this has a logical name first-app, because we have @EnableEurekaClient this will be registered as a microservice in the Service Discovery, @EnableEurekaClient will automatically look in 8761 port.

Build the application & run the jar

If we want to create a different build name we can mention this in pom.xml

pom.xml

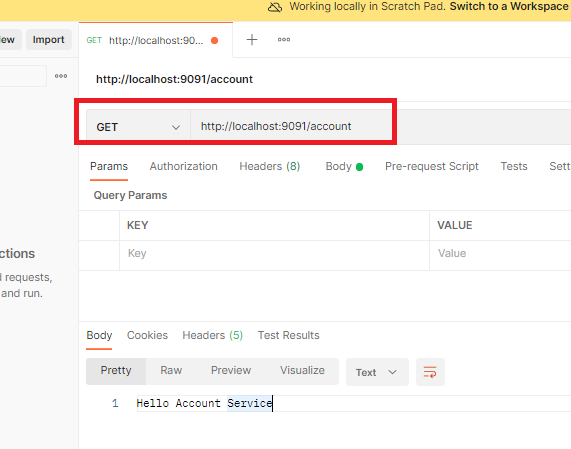




Here the FIRST-APP is the name of the microservice that must be used by other client microservices, it means the clients don’t use physical address

Note: While we need to test we need to physical address of this microservice

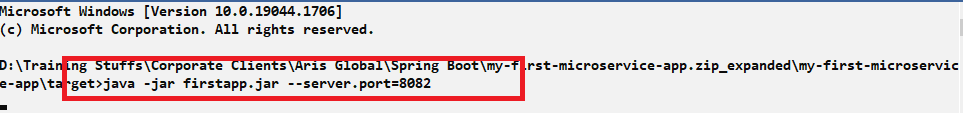
In postman you can’t use logical name because it is not registered application in the service discovery.



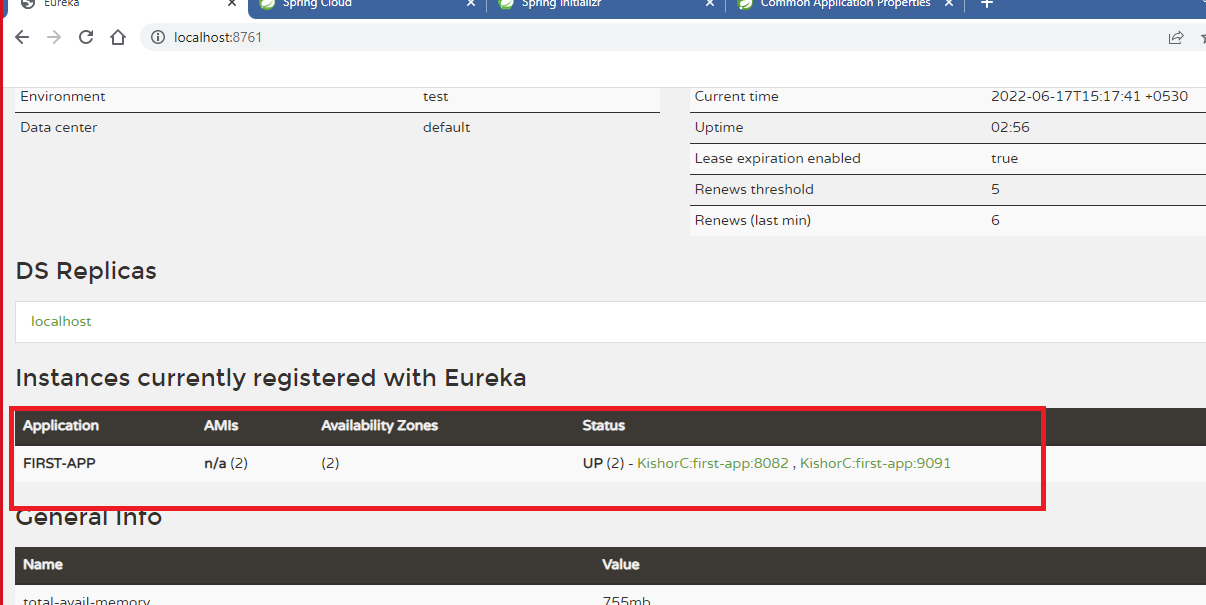
Creating another instance of the microservice

You can use same java -jar command to run another instance of the microservice

java -jar firstapp.jar *--*server.port=8082

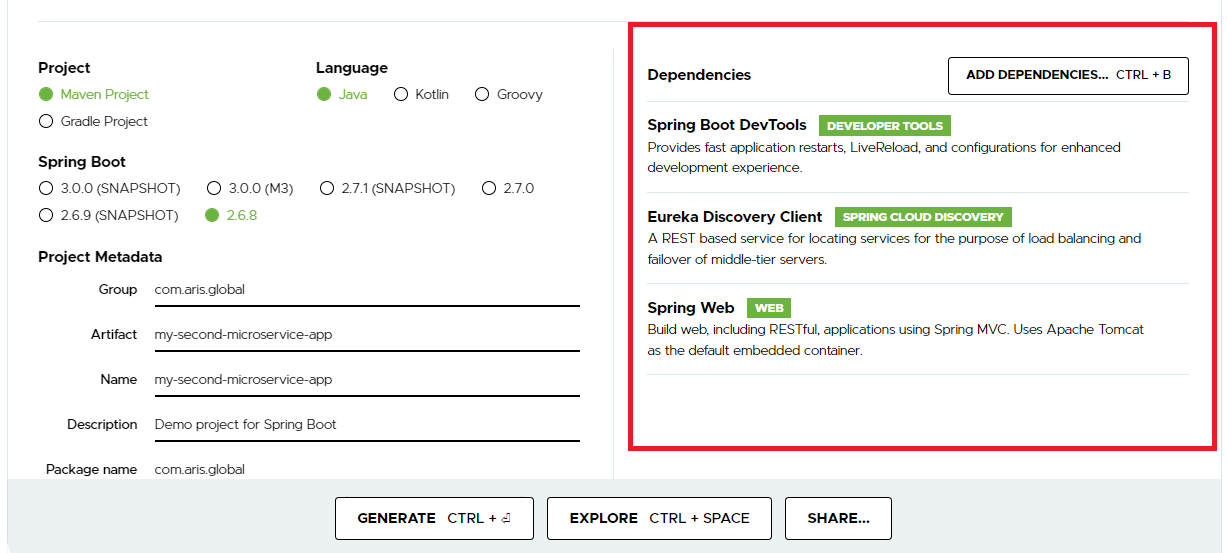


We must able to see the same microservice with 2 instances running in the eureka dashboard

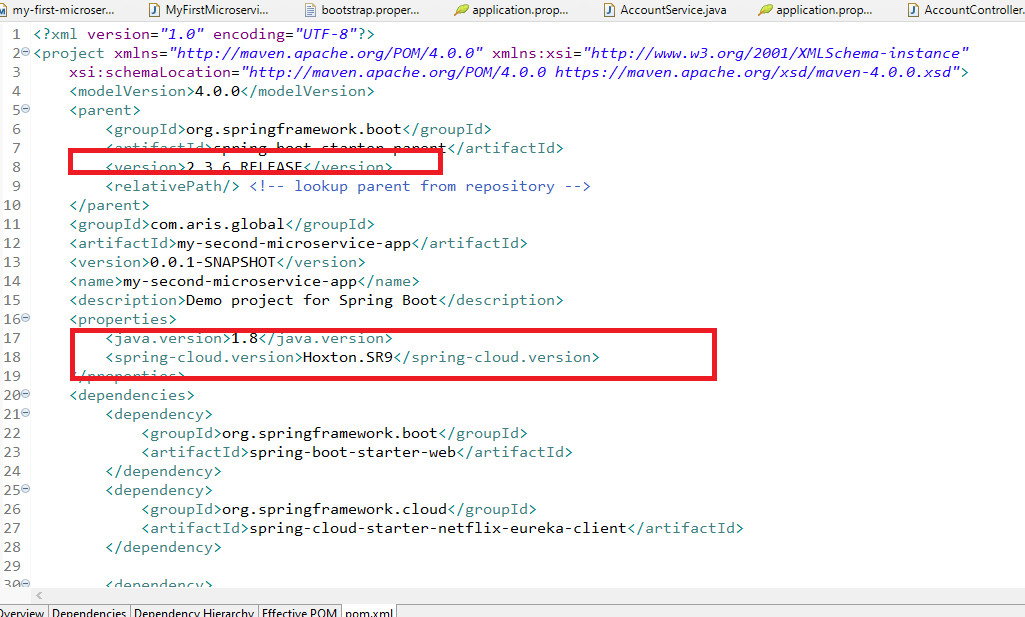


The above microservice has same name FIRST-APP that runs 2 instances one in 8082 & another in 9091, when client microservice wants to communicate they don’t use remote microservice physical address instead they use logical name i.e., FIRST-APP with Client Side load balancer, here the load balancer takes care of distributing the requests to the microservice one by one.

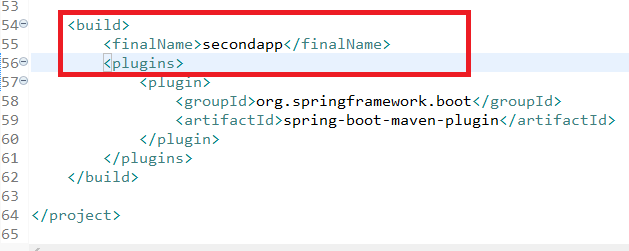
Create another project with the same dependencies



Note: Change the spring.cloud & spring boot versions in pom.xml to make it compatible with Hoxton.SR9 & also you can change the name of the jar file you want in <build>



Adding <final-name>



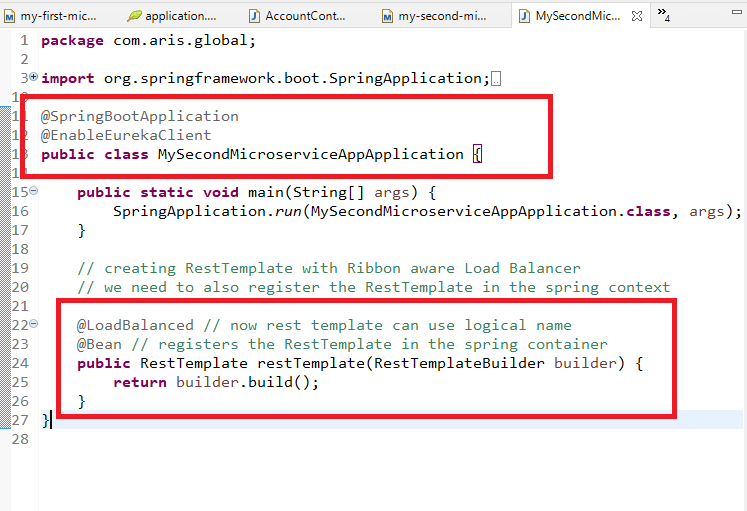
How to communicate from one microservice to another microservice

@LoadBalanced:

We need to use Client Side Load Balancer, spring cloud provides us a Ribbon aware load balancer with an annotation @LoadBalanced which you need to mention on the instance RestTemplate.

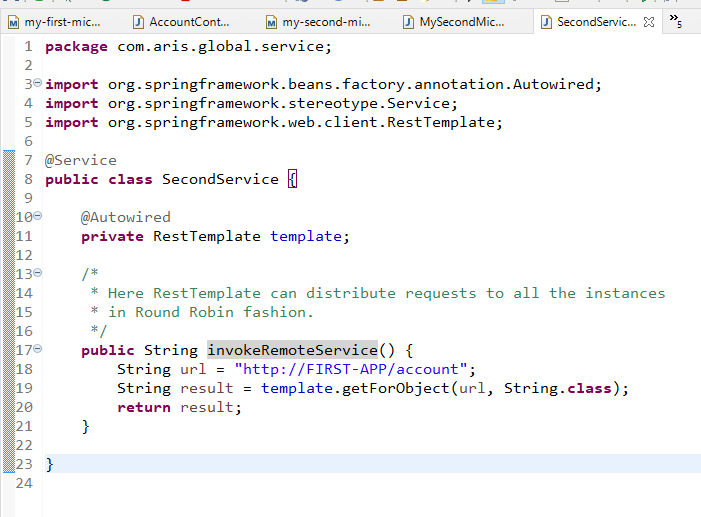
RestTemplate: It is an instance used to call online services with the URL & HTTP methods.

We can create RestTemplate and register as a bean and also Enable the application to register in the Service Discovery.



We can use this RestTemplate in the service layer to call the first microservice

SecondService.java

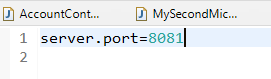


SecondController.java

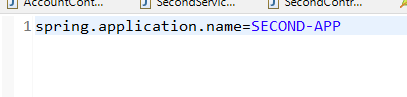


Update the property files to configure the port & the logical name

application.properties

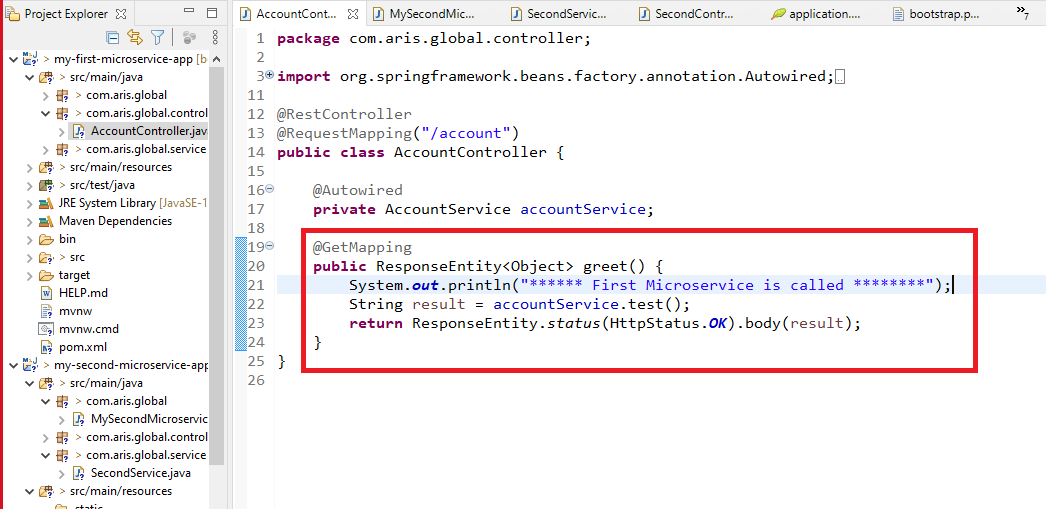


bootstrap.properties



Just to confirm that multiple instances of First Microservice gets the request from the client we will put a simple print statement in the rest webservices

my-first-microservie-app/AccountController.java

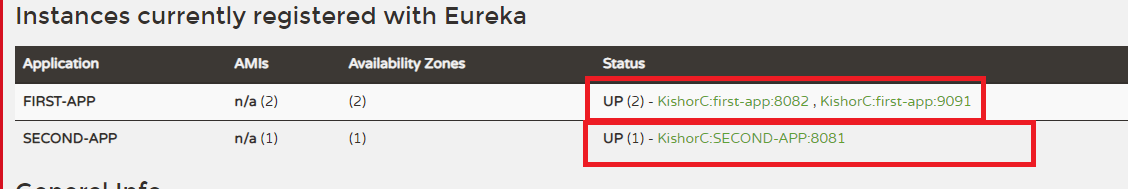


We will rebuild both the microservice & run the first microservice with 2 instances & second with one instance

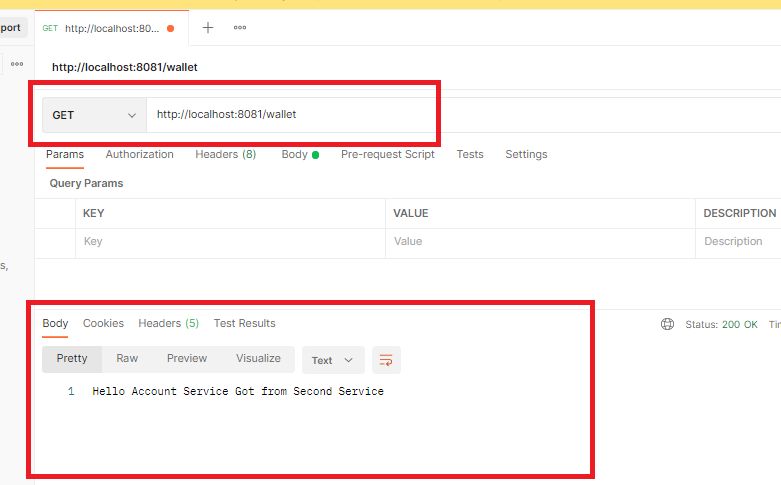
Totally we will be having 4 programs in 4 terminals

1. First Microservice 2 instances
2. Second Microservice 1 instance
3. Service Discovery

In Eureka Dashboard you will see 2 instances of first microservice & 1 instance of second microservice

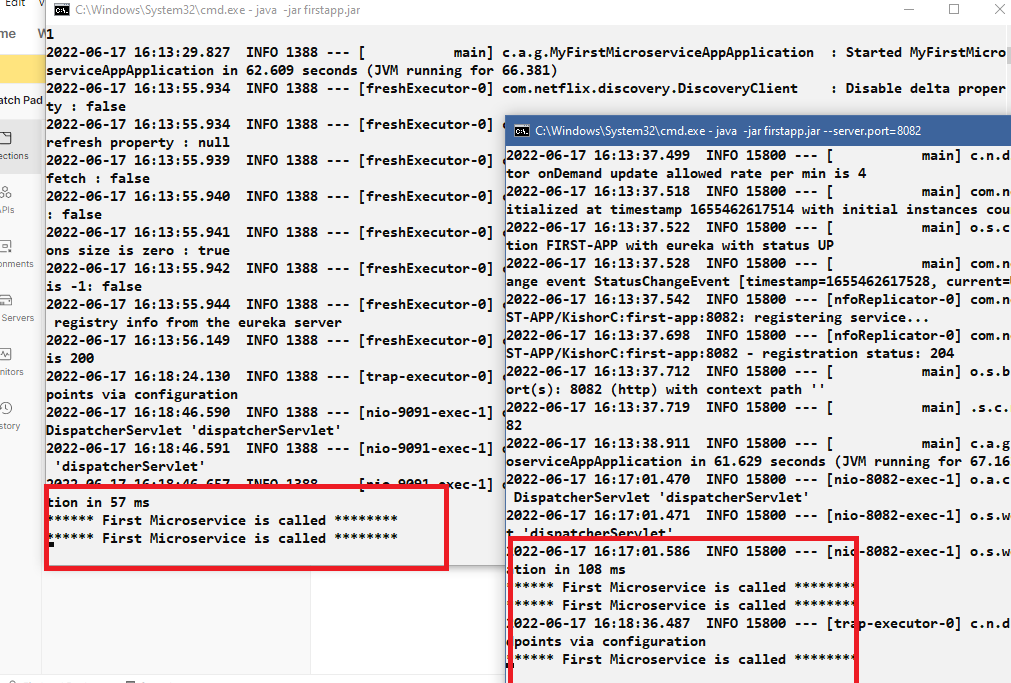


Now we can test second microservice from postman and see that terminals of first microservices logging the print statement

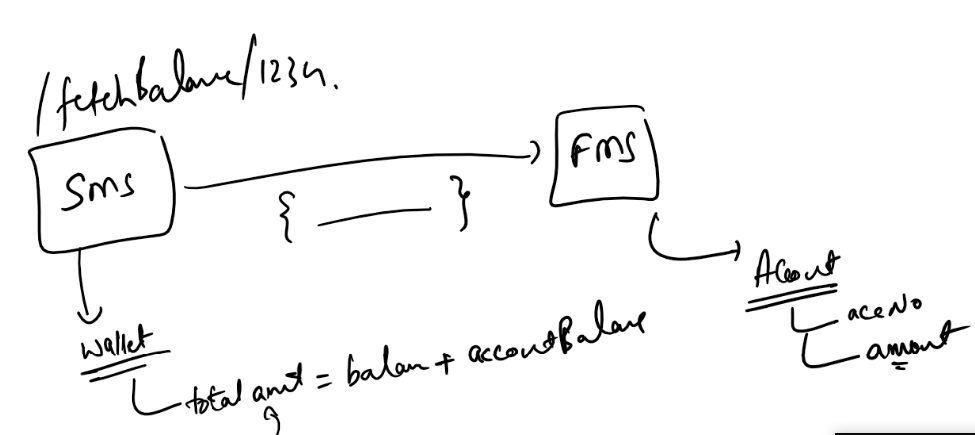


But we must see the terminal which will have prints that specifies who is handling the request.

Output:



Activity:



Create an Account at the First Microservice that will return some amount to the Second Microservice & the second microservice can use that amount and update the Wallet present in the Second Microservice and you must able to tell user how much amount you can utilize from the wallet.

Suppose /fetchBalance/1234 is the request then you must able to see the balance present in account + balance present in wallet