Course Content

* Spring Boot
* Spring REST Webservices
* Spring Cloud and Microservices

Software’s required

* Java 8 / JDK 8
* Eclipse IDE / STS
* H2 or MySQL
* Open Internet

Spring Framework: It is a Java Framework used to develop different types of applications like web, mobile, enterprise, cloud and so on, it simplifies the development process by providing many design patterns required to build an application

* Dependency Injection
* Singleton & Factory
* MVC
* Proxy
* Prototype

Note: Spring Framework uses XML configuration file to configure the dependencies, in this XML file you will mention all the classes that Spring should instantiate in the container.

Spring Boot

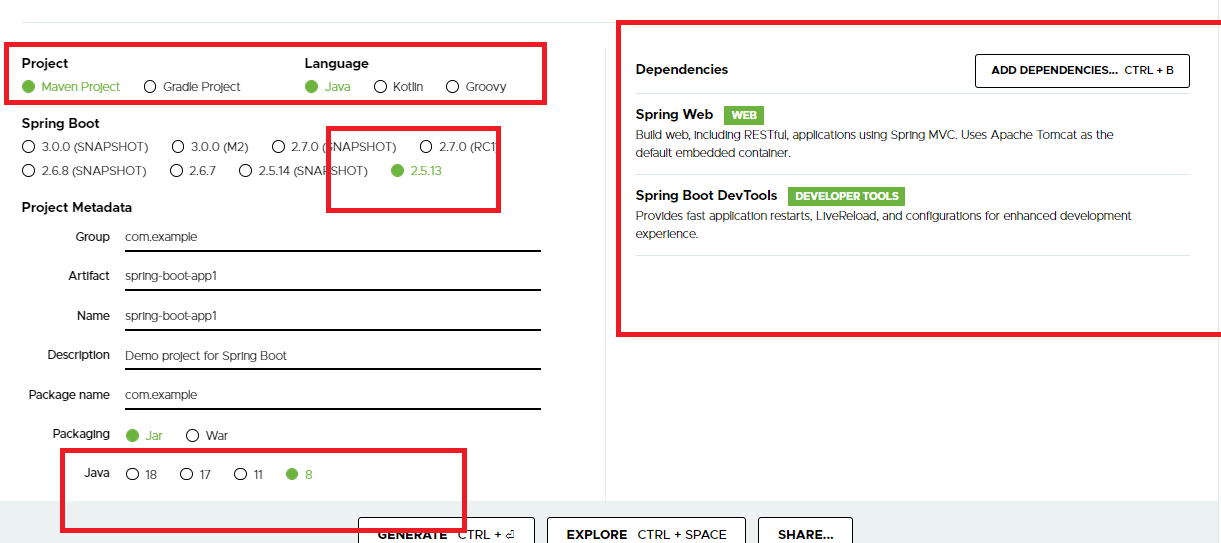
It simplifies developing spring applications in a simple way by taking care of all the generic configurations i.e., spring boot automatically does the configurations based on the spring boot starter libraries you add in pom.xml

* You don’t need any xml file because there’s a spring boot starter library that takes care of creating objects in the container
* You don’t need to set up server because there’s a spring boot starter web library that adds server to your application
* You don’t need to supply the dependencies for datasources because there’s a spring boot starter data library it takes care of autoconfiguring the datasources
* You can migrate to another version without making more changes because spring boot provides a parent library and all other spring boot libraries follow this parent library, so if you change parent library then all the other spring boot libraries follow that change automatically.

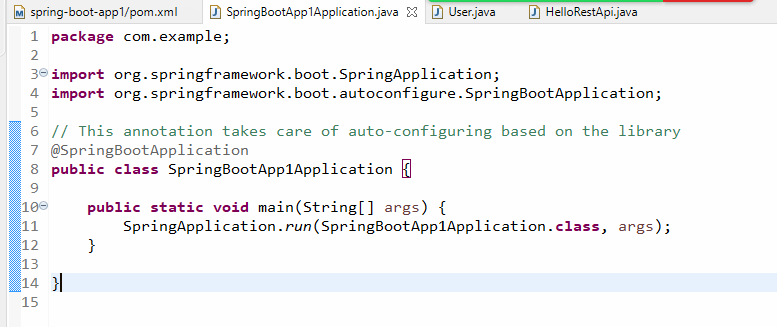
Spring vs Spring Boot: Suppose you want to develop any web application’s or webservices you will configure following things in both

|  |  |  |
| --- | --- | --- |
| Features | Spring Old approach | Spring Boot |
| pom.xml | spring mvc, faster xml | spring boot starter web |
| DispatcherServlet | need to configure in one xml file | Automatically done by the web library |
| Bean object creations | need to configure in one xml | Automatically one by the spring boot |
| Server | need to configure | Automatically provided in the web library |
| Migration | need to change the version of each libraries | need to change only the parent version and automatically other libraries follow |

First Spring Boot Project

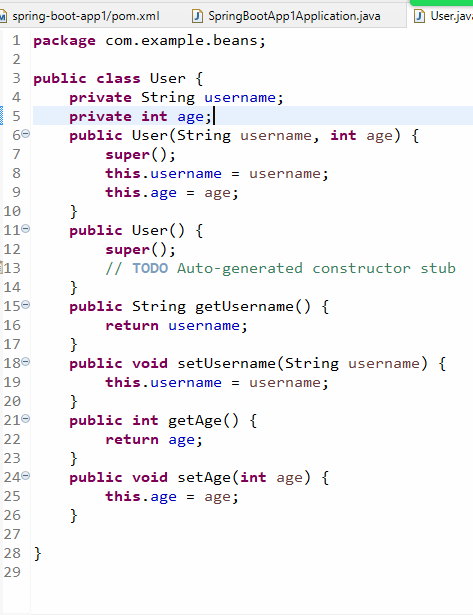


SpringBootApp1Application.java

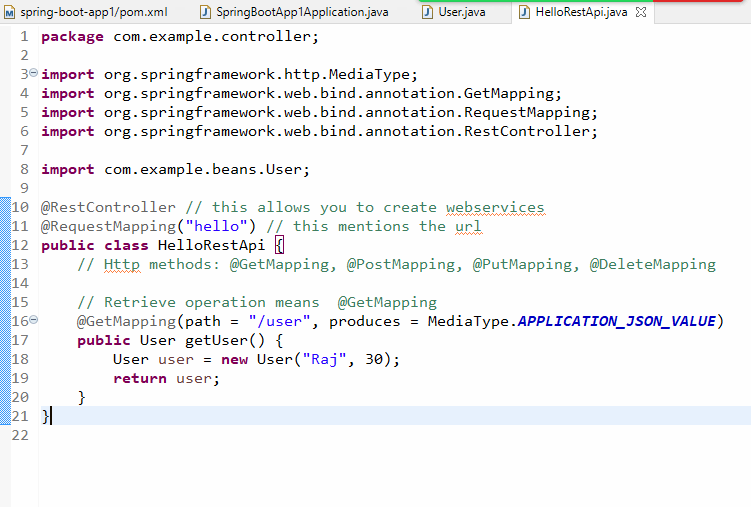


@SpringBootAppication: It auto-configures the server, scans all the classes having @Repository, @Service, @RestController, @Controller, @Configuration, @Aspect, and so on and registers in a spring container without having an xml file, it configures the application based on the libraries in the class path

User.java



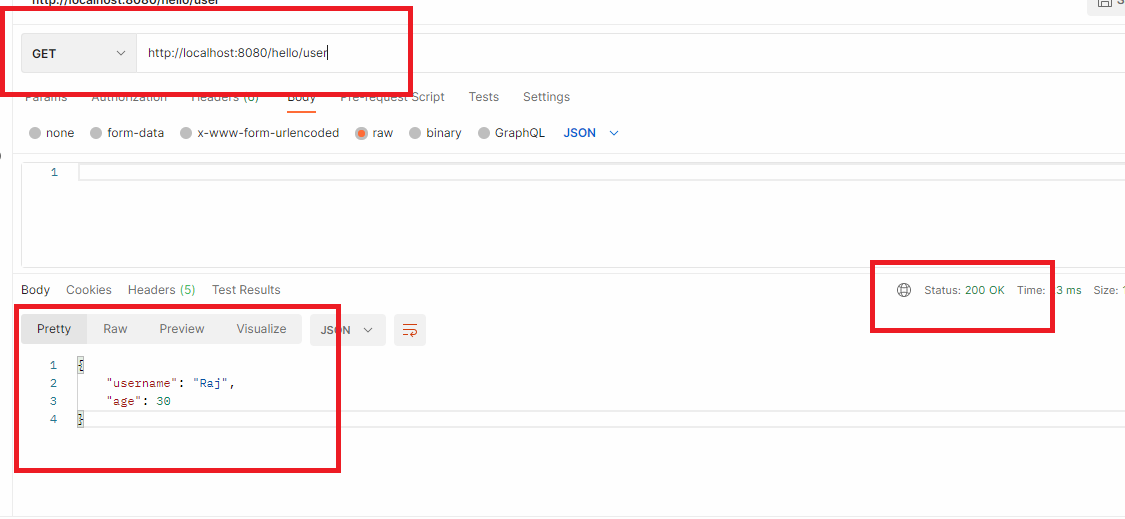
HelloRestApi.java



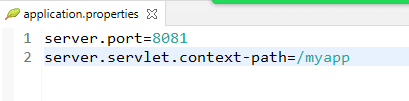
@RestController: It is an annotation used to create webservices

@GetMapping: Represents the operation is fetching

Postman output:

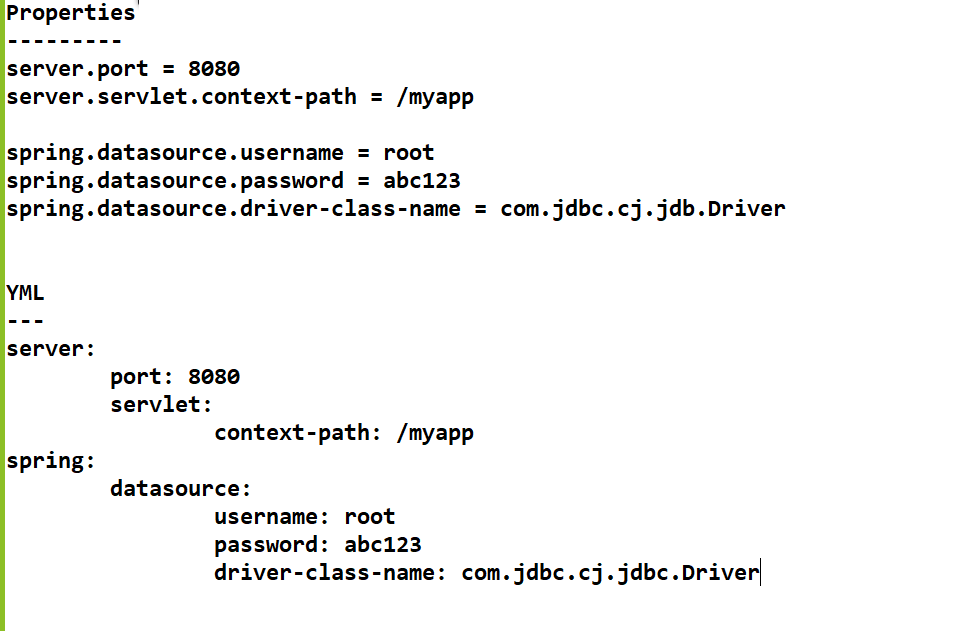


application.properties



Here the context path will be /myapp, so that your webservice needs to be accessed by <http://localhost:8081/myapp/hello/user>

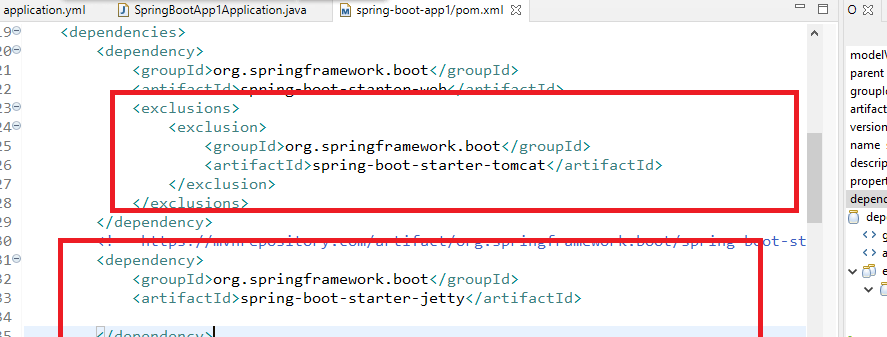
application.yml



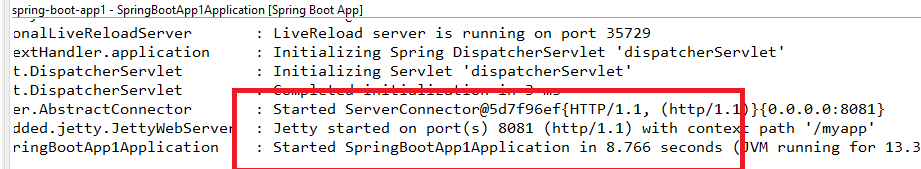
The spring boot application can be run on any JRE machine even in cloud machines also, you need to create an executable jar file for it

You can add other open source servers like jetty or undertow in spring boot apart from tomcat.

Since by default tomcat is used, you can exclude tomcat and add either jetty or undertow.

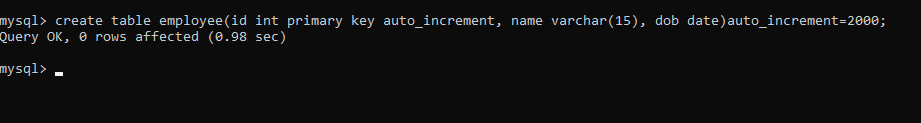


You can see in the console log the jetty server status



How to interact with the database

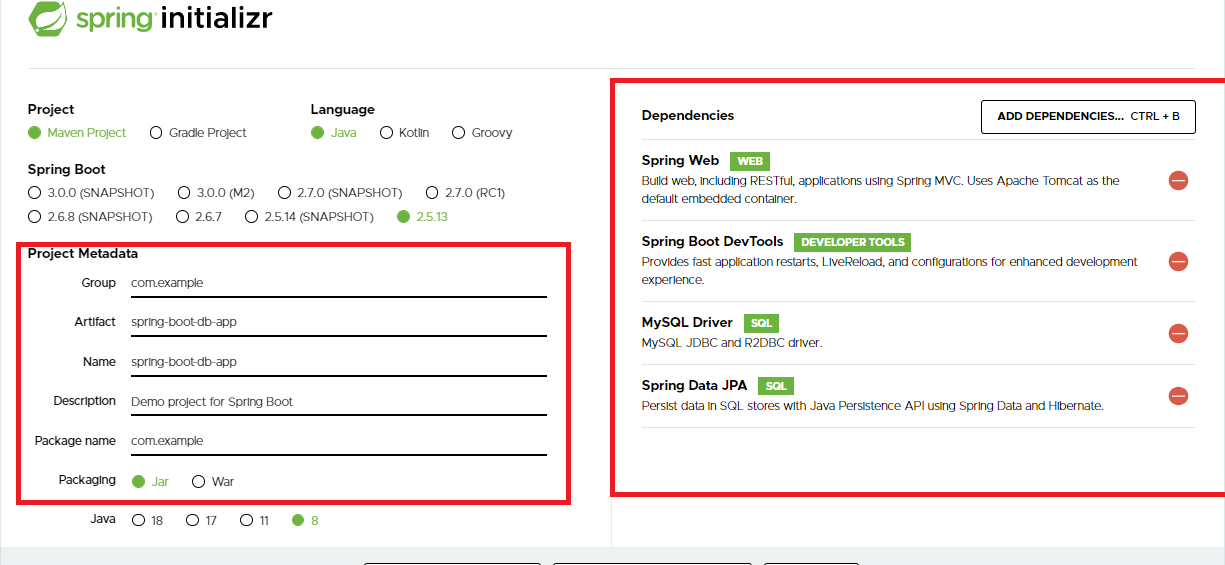
* You must add the database dependency like H2, MySQL, Derby so on
* You must create the class that maps to the database i.e., entity class
* You must configure the datasource in the application.yml
* You must use spring data jpa library to easily perform CRUD operation without any implementation



Now the application must have 3 layers

1. Controller: To have webservices takes care of converting JSON to Java and vice versa
2. Business layer/ service layer: To have business logics
3. DAO layer: This will have database logics

Note: In Spring Boot you don’t have to write any implementation to the database logics, because you get a proxy object to do the job for you, this proxy object needs which entity class you are mapping to which table, then it provides all the necessary methods to perform CRUD operations.



Spring Data Jpa: It takes care of implementing the database logics according to the configuration done in the repository interfaces, it takes care of establishing connections with the database as per the application.yml configuration

JpaRepository: It is an interface with all the CRUD operations like save(), delete(), findById() and so on, all these methods are implemented by proxy object at runtime based on the entity class mentioned

Entity Class: it is a class that maps to the table, suppose Employee is a class then it can be mapped to employee table in Mysql or anyother database

interface EmployeeDao extends JpaRepository<Employee> { } : This is an interface that will get all the methods of JpaRepository and all the methods will interact with Employee table because Employee class is an entity class maps to Employee table

Custom methods: These methods you can create in your dao interface which can perform some other operations on the table with the JPQL query

interface EmployeeDao extends JpaRepository<Employee> {   
 @Query(“select e from Employee e where e.name = ?1”)  
 public List<Employee> findEmployeesBasedOnName(String name);  
}

If you call findEmployeesBasedOnName() by passing some name then it gets all the employees having that name in List<Employee>

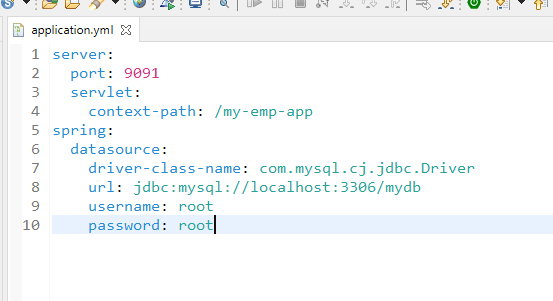
Whenever there are multiple layers in the code like DAO, Service, Controller then we need to know that service code calls dao code and controller code calls service code, means service depends on dao and controller depends on service.

Note: Spring Boot automatically takes care of injecting the dependencies like DAO will be injected to Service & Service will be injected to Controller.

Things we need to create to interact with Employee table from Spring Boot

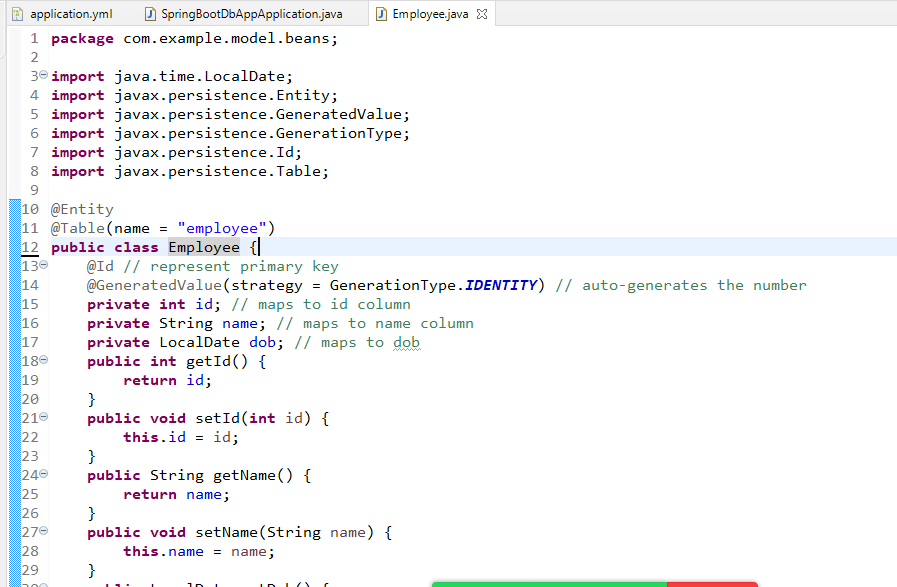
1. Employee class: Entity class maps to Employee table
2. EmployeeDao: Interface that extends JpaRepository<Employee>
3. EmployeeService interface: All the methods to perform business logics
4. EmployeeServiceImpl: All the methods of service must be implemented like transactions, other business tasks
5. EmployeeRest: Webservices
6. EmployeeNotFoundException: Exception that needs to be generated if employee id is wrong

application.yml

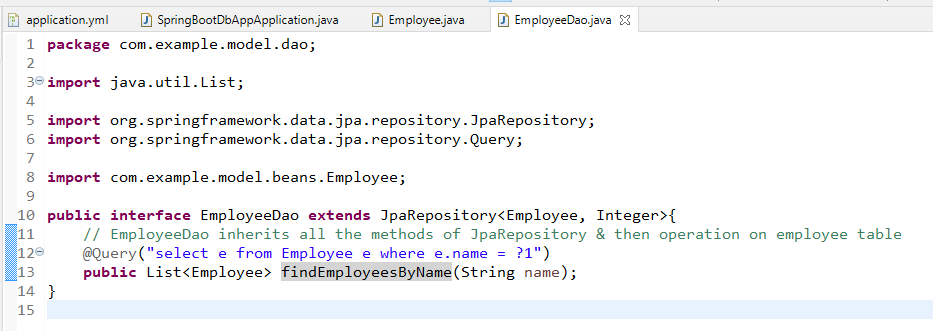


The above configuration helps spring boot to automatically connect to the database & create all the beans required to interact with the database.

Employee.java



EmployeeDao.java



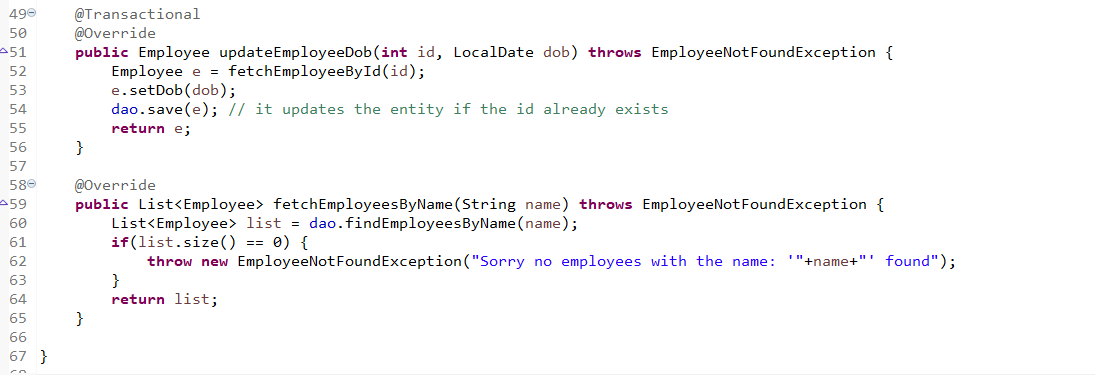
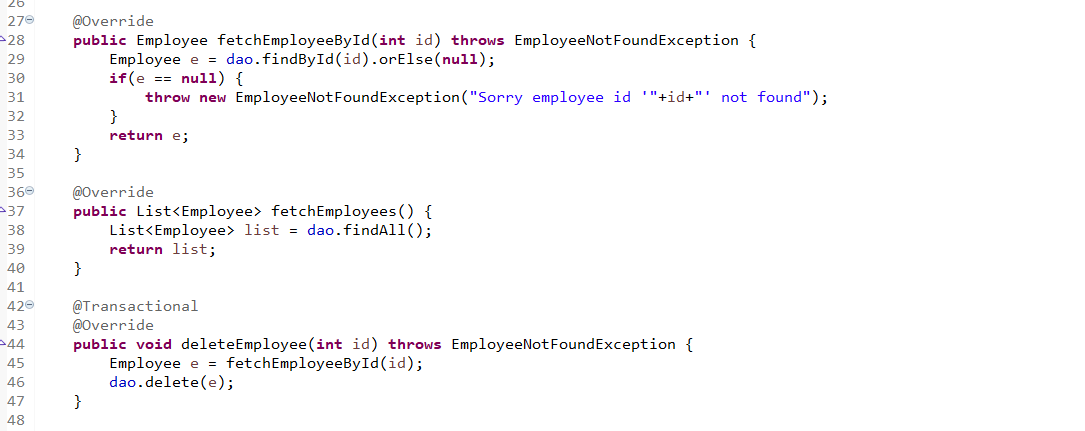
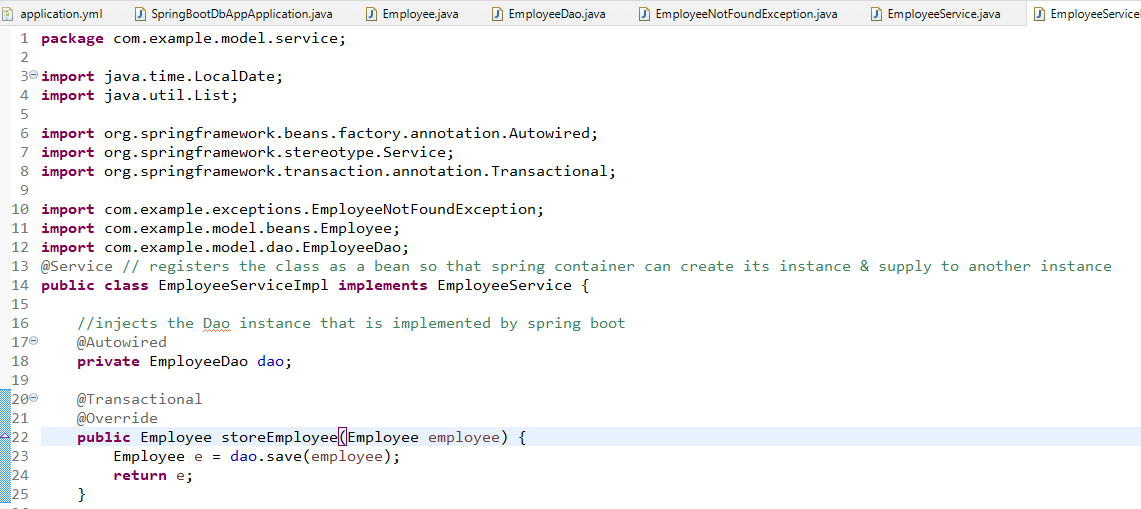
EmployeeNotFoundException.java



EmployeeService.java



EmployeeServiceImpl.java



@Transactional: It is used when you do any modification

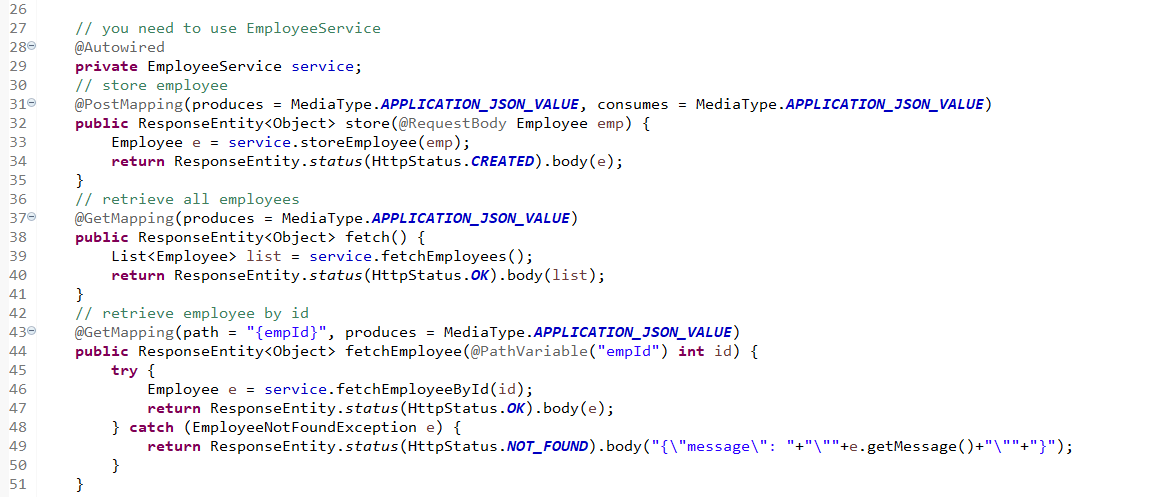
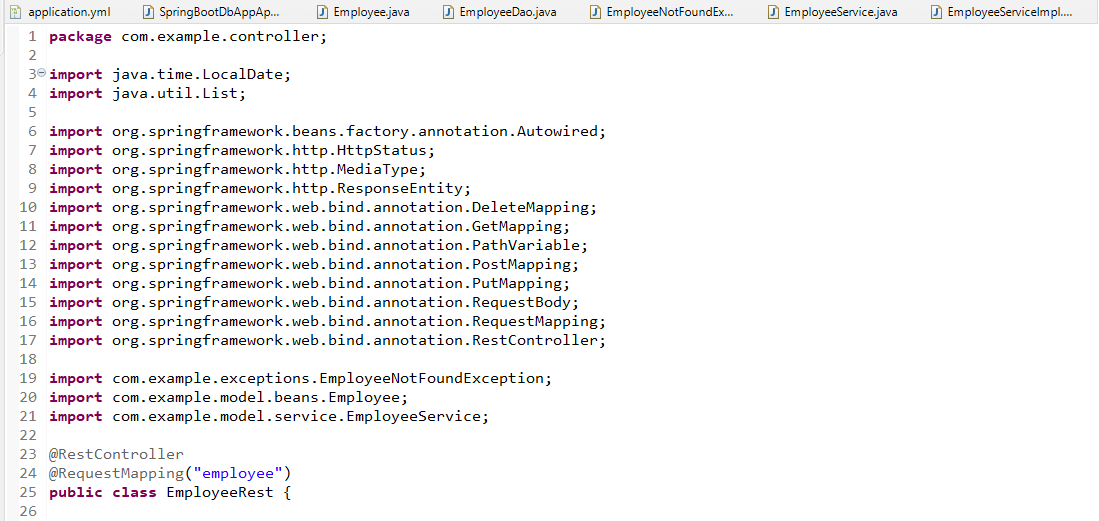
You can now implement Rest Apis to perform CRUD operations, here you need to use POST for new data , PUT for updating existing data, GET for fetching data, DELETE to delete the existing data

You can use ResponseEntity<T> as a return type in webservice, so that you can customize the response to have different status code & the contents

ResponseEntity.status(404).body(object);   
ResponseEntity.status(401).body(object);  
ResponseEntity.status(201).body(object);

Here the object in the body is sent to the consumer and consumer can use status code to decide his operations.

EmployeeRest.java



You can use MongoRepository to interact with MongoDB database, mongodb stores data in NoSQL format i.e., javascript object it stores

Activity:

1. Try the above activity on employee table
2. Create an address table & use a foreign key in the address table that will have employee id, in the address table you can have state, city & pincode
   1. When you store employee object you also need to update address table
   2. When you retrieve object you must able to retrieve address of that employee as well.

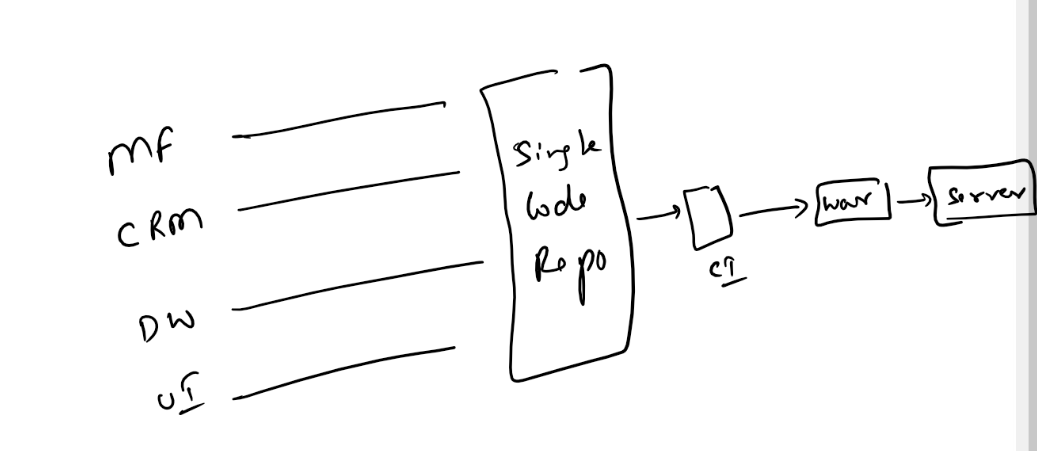
Hint: Use @OneToOne annotation, create an address reference in the Employee class

Spring Microservices

* Spring Boot
* Spring REST

Microserivces: are loosely coupled services that are independent form other services of either same or different applications, it came in 2014 to solve some of the issues / problems faced in Monolithic architecture

Monolithic Architecture:

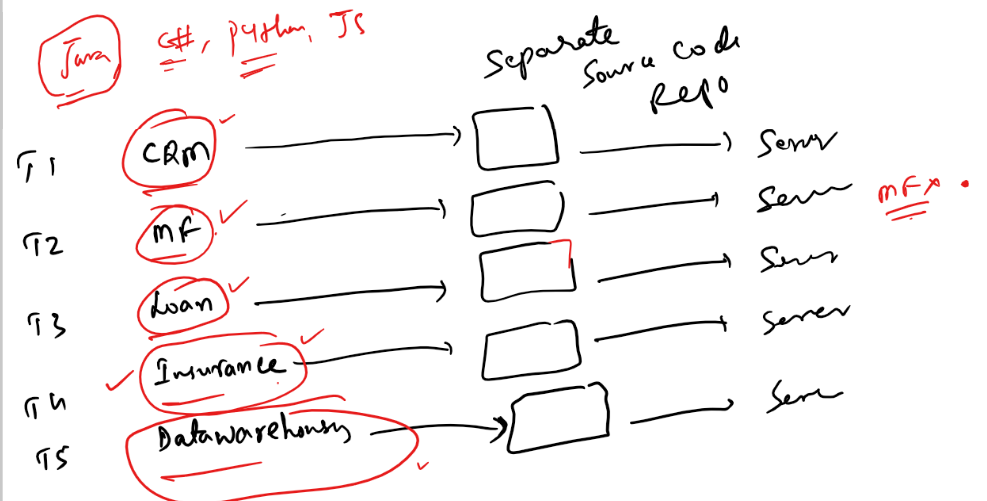


Before Microservices, most of the web applications followed monolithic architecture, here an application will have multiple modules and each team handles separate modules & they push to the single code repository later it will integrated to create a single build file which will be tested & deployed.

Problems with monolithic architecture

* Changes made in any service forces all the services to re-test & re-build
* If any one of the service goes down then there could be possibility that entire application might go down, because all the services are tightly coupled in a single artifact
* If any one of the service takes more traffic you can’t scale just a single service, you need to scale entire application, you need more server resources
* All the services need to be created with the same technology

Microservice Architecture



Microservices are reusable services that can be developed, tested & deployed independently.

Each team can work on services independently and can push the code in a separate repository, so that each code can be built separately and deployed separately so that each build are independent

Advantages:

* Loosely coupled
* If any changes is done in any of the service then only that service needs to be tested
* If any problem occurs in any of the service, then it doesn’t affect other services as they are independent from every service
* You can scale a particular service that takes more traffic
* You can use multiple technologies to create microservices of same applications without depending on one technology
* Each microservice can communicate with other microservice with REST calls

Microservice patterns

* Service Discovery
* Discovery Client
* Circuit Breaker
* Load Balancing
* Externalized Configuration
* Distributed Log tracing
* Security

Service Discovery: It is a registry that registers all the microservices, it keeps a list of microservices and their physical location, every microservice has to send heartbeat to the service discovery to confirm its health status, this happens every 30s,

Note: Service Discovery will give 3 chances for microservice every 30s before delisting the microservice from its list

Discovery Client: These are the microservices which will have an ability to register to the service discovery and send hearth beats every 30s to the service discovery, Discovery Client will have a logical name that is used by other microservices, they register logical name along with physical address, Even if there are multiple instances of Microservices their logical name should be same, because client uses logical name

Circuit Breaker: When microservices are communicating with other microservices, there could be chance of cascade of failures, where if one microservice is down other microservice will wait for the response & at one point of time it will also go down, in that case the entire microservices may go down, hence you need a circuit breaker so that when the microservice is down the circuit breaker will check the failure rates & stops sending request to microservices which are down, so that it can avoid cascade of failures i.e., other microservices wouldn’t go down.

Load Balancing: It distributes the load across the microservices, it will be at the client side it will know the location of all the instances of microservices, if any instance is decreased or increased then client side load balancer will automatically refresh in 30s because discovery client sends heart beat for every 30s along with that it fetches service discovery

Externalized Configuration:

It is a common configuration that can be used by multiple microservices, it can be secured by making it private so that microservices can access it only with proper authentication, you can also encrypt the external configuration so that microservices can decrypt.

Distributed Log Tracing: It takes care of aggregating all the logs generated by microservices and show how the request is processed sequentially

Security: Only authenticated & authorized users & application must able to access microservices

Languages that can implement microservices

1. Java
2. Javascript
3. C#
4. Python

and so on.

Note: No matter you use which language to create microservices they all should implement with the above design patterns

In Java you can create microservices using Spring

Spring provides 2 projects from its module to create microservices

1. Spring Boot
2. Spring Cloud

Spring Boot: It quickly creates a product grade spring application by providing automated features

Spring Cloud: It gives all the necessary tools required to create microservices with all the common design patterns like Service Discovery, Discovery Client, Circuit Breaker, Load Balancing, Externalized Configuration, Security etc.

You can just use annotations provided by spring cloud to quickly set up the microservices in less time, these annotations will take care of doing their respective roles

ex:

@EnableEurekaServer: Creates service discovery & all the roles of service discovery

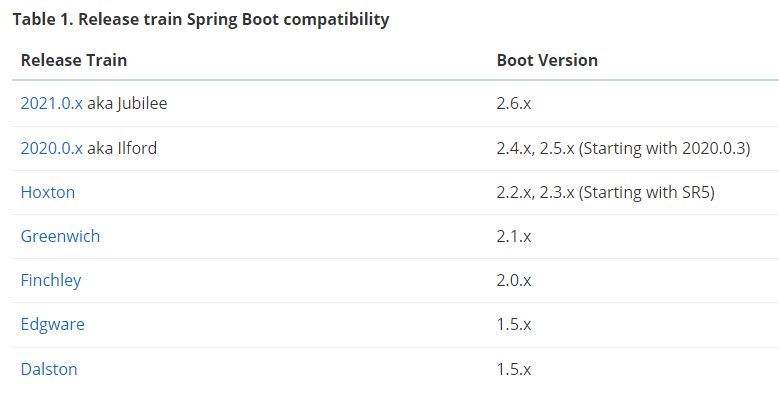
@EnableEurekaClient: Creates microservices & registers in the service discovery & other roles of discovery client - fetching from service discovery, sending heart beats every 30s

@CircuitBreaker: Creates a circuit breaker & takes care of performing its role based on the configuration

@LoadBalanced: Creates a client side load balancer

Note: Spring Cloud & Spring Boot versions must always be compatible with the release train given by spring

<https://spring.io/projects/spring-cloud>



Here we need to understand that the Hoxton is stable release and the corresponding spring boot version must be either 2.2.x or 2.3.x, but in spring initializr, you may not see spring version 2.2 or 2.3 hence you must manually change the pom.xml to 2.2 or 2.3 and spring cloud version to Hoxton.

Spring Cloud:

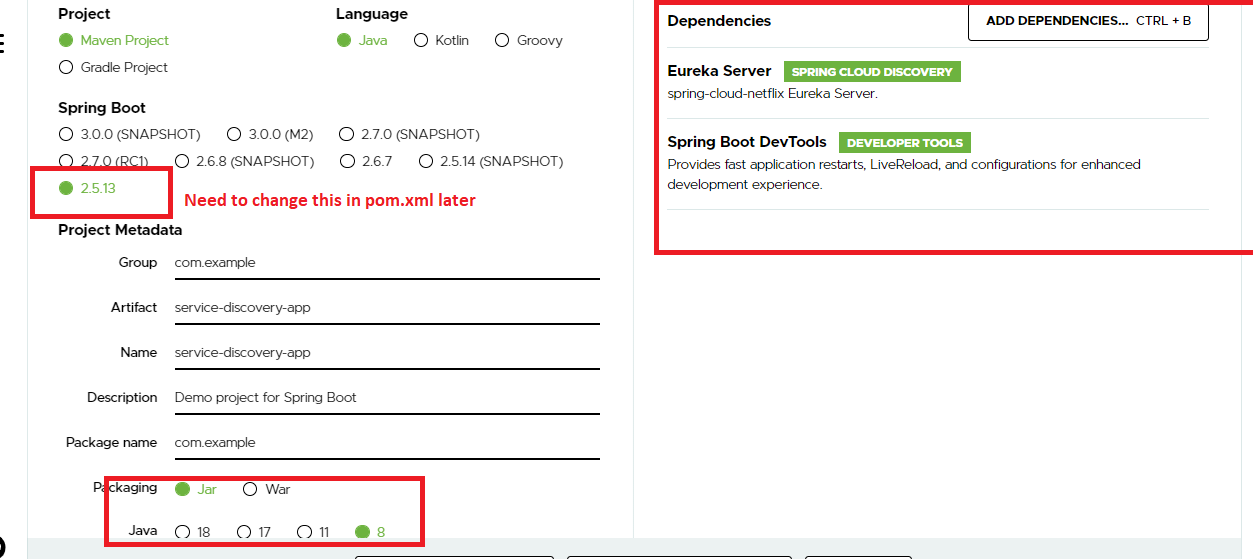
It is a separate project by spring community to develop microservices by taking advantage of spring boot & features of microservices, it uses many spring boot features.

* Starter projects
* Compatible version of spring boot with spring cloud
* Creating an executable jar which can be deployed in any cloud environment
* Property configuration files yml/properties
* Passing the command lines to override configurations
* JpaRepository
* Working on GIT
* Rest based services

Steps to create microservices

1. Create Service Discovery - Separate project
2. Create & Register Microservices (Discovery Client) - Separate project

Create Service Discovery



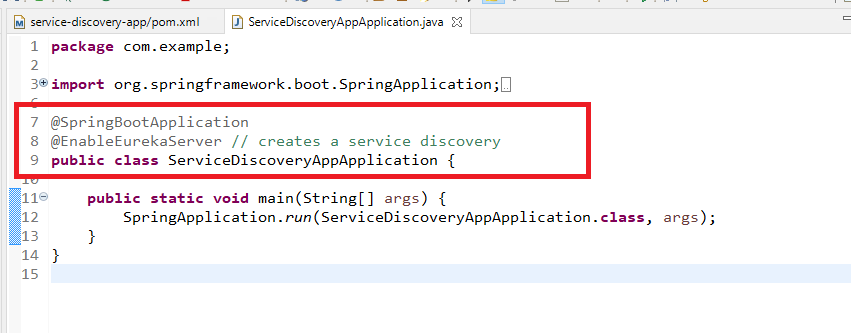
Note: Change the pom.xml so that you will use spring boot 2.3.x and spring cloud Hoxton.SR release



Note: Once it downloads the library, Update the project

The Spring Cloud gives us many annotations that helps to implement lot of design patterns, but now we need to use @EnableEurekaServer

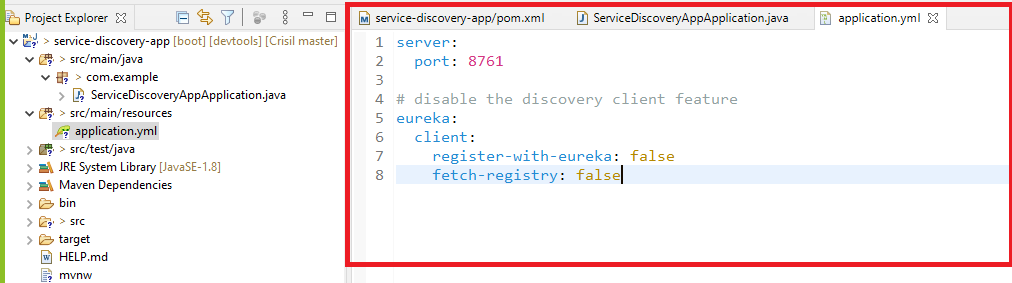
ServiceDiscoveryAppApplication.java



Since the spring boot has embedded server that runs in 8080, you need to change to 8761, because every microservice looks for service discovery in 8761 port

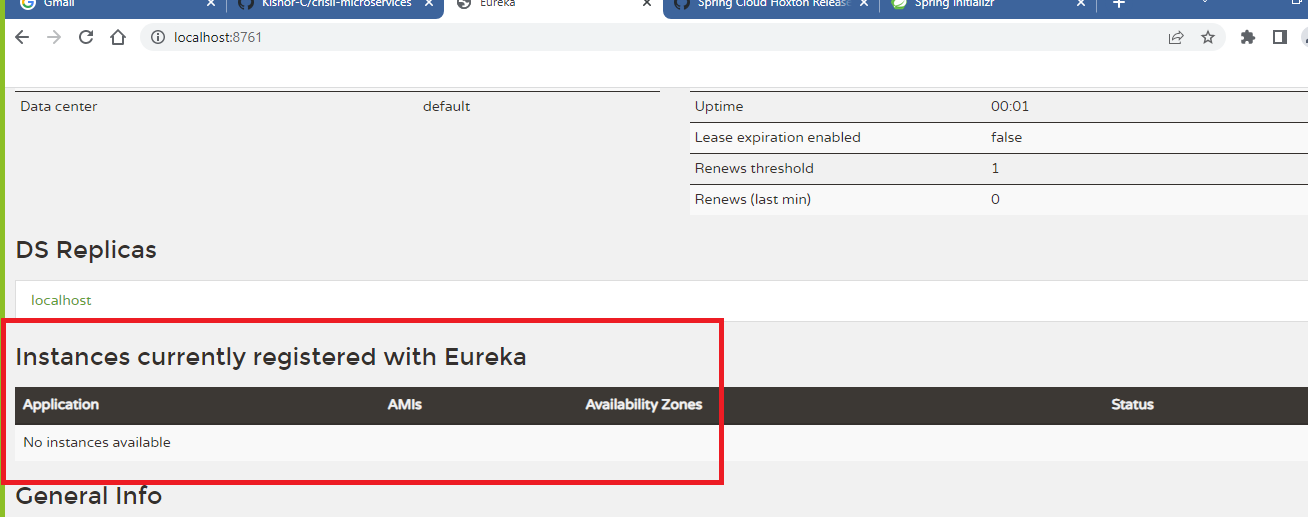
Note: When you add Eureka Server library it also downloads Eureka Client library, it means it will try to register service discovery as a microservice, so you need to disable that feature in application.yml like register with eureka & fetch register to be made false.

application.yml



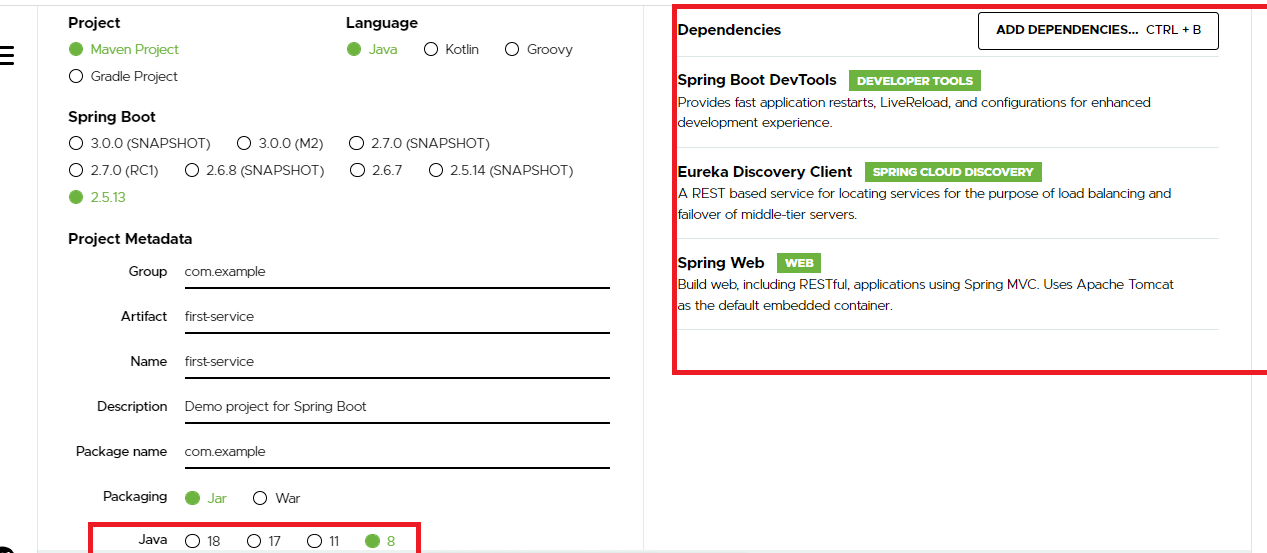
Note: Now you can run this project, Eureka Server gives a dashboard that you can use to see the microservices list, you can access it in 8761 port as its in localhost you can use

<http://localhost:8761>

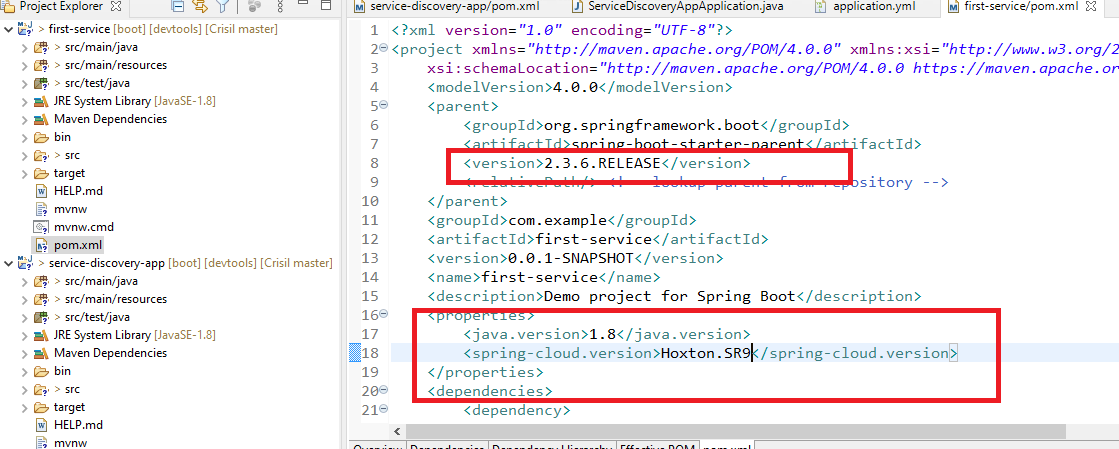


You can create Microservices and see its registered or not here

Creating a microservice

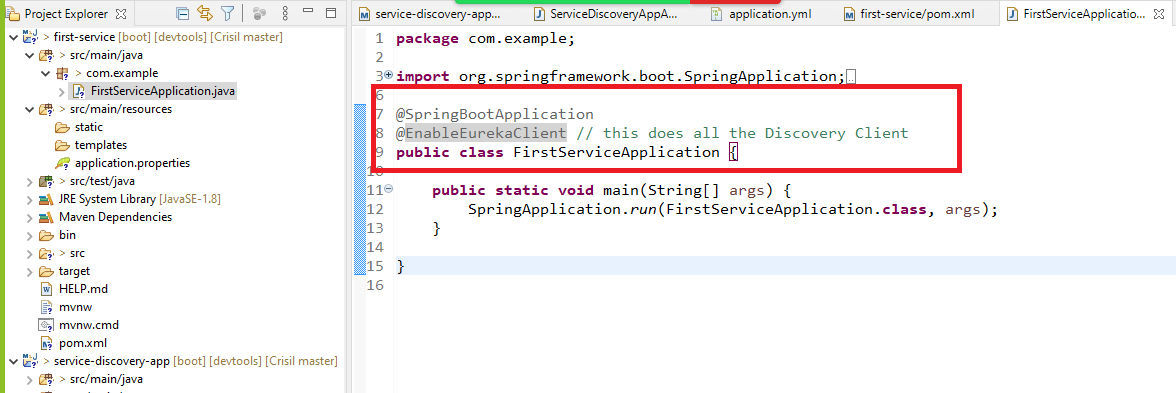


Note: Change pom.xml accordingly



Note: Update the project if you see cross mark

FirstServiceApplication.java

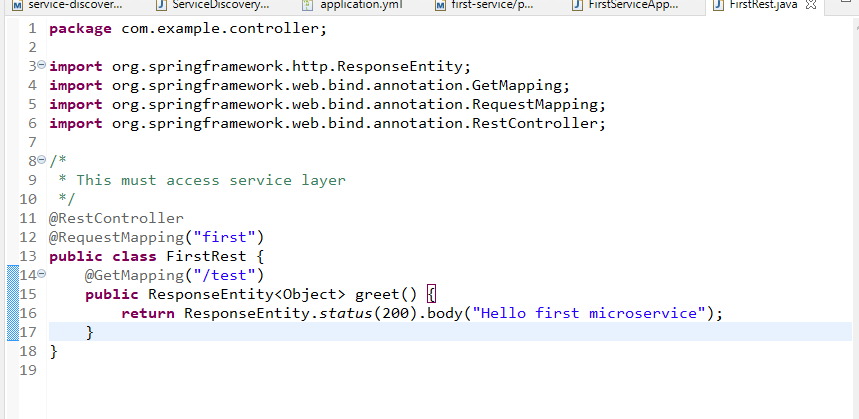


@EnableEurekaClient: It takes care of

* Registering it in Service Discovery as a microservice
* Fetching Service Discovery
* Sending Heart beats every 30s
* Looking for Service Discovery in 8761 port by default.
* Having a logical name in its property file

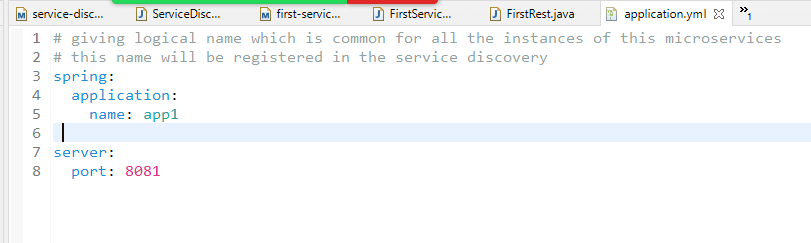
You can create services that can be accessed by other microservices

FirstRest.java

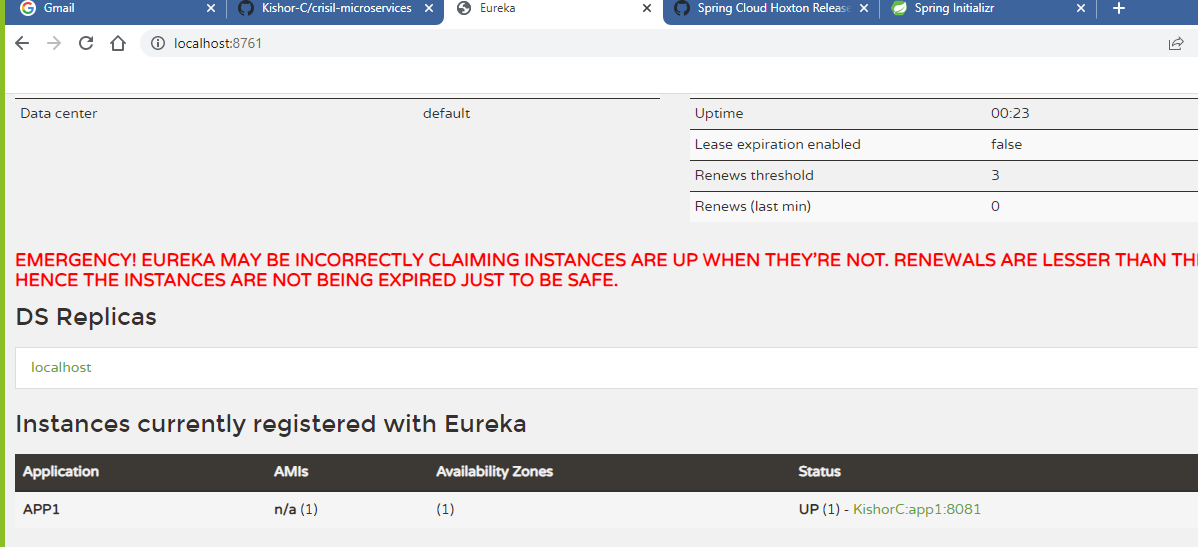


Note: Any microservices who wants to access this first/test must use the logical name & access, but if you are testing this microservice then you must use physical address

application.yml

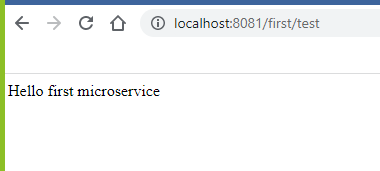


Now you can run the same microservice in different port but they all register with the same name app1 in the service discovery.



Now the microservices who wants to communicate must use APP1 to access this microservice, i.e., <http://APP1>

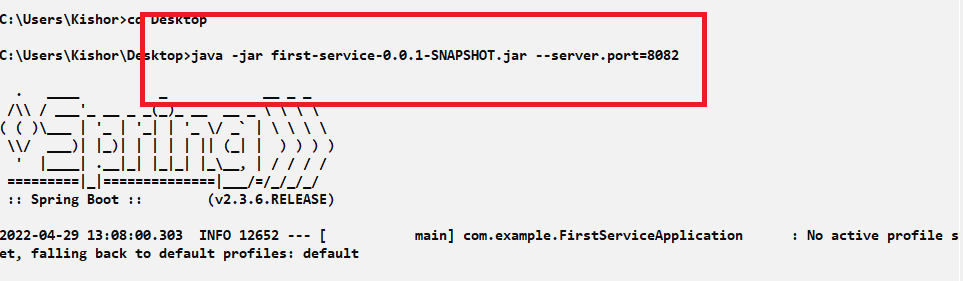
However if you want to test your microservice you must use its physical address only.



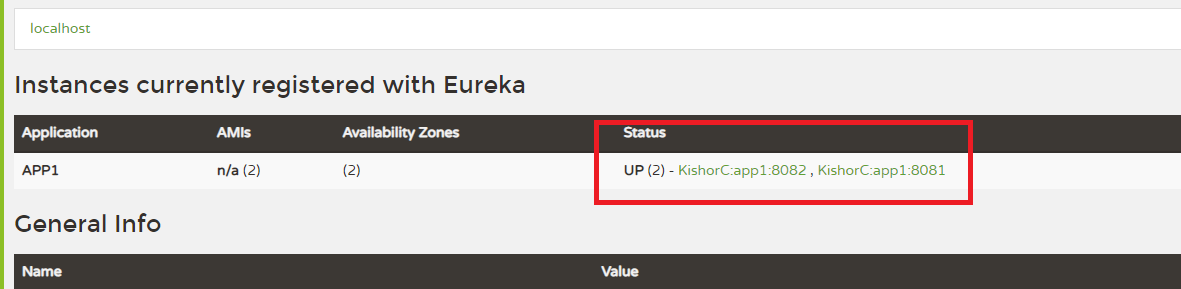
But the microservices must use <http://APP1/first/test>

You can create another instance of same microservice, for that you can create an executable jar & run that jar in command prompt.

To create an executable jar Goto File -> Run As -> Maven -> build -> Type package in goals

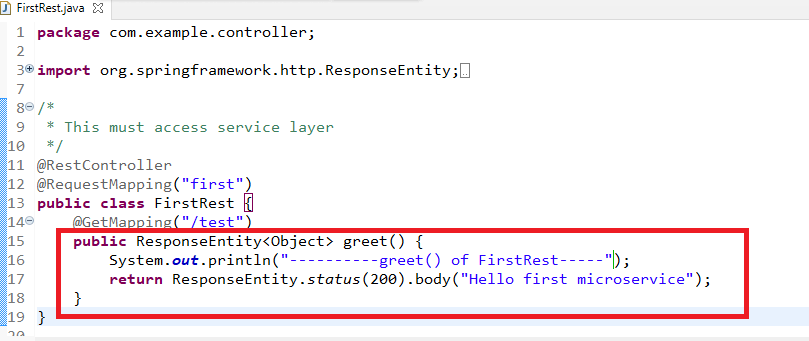


Note: Now totally there are 2 applications running in 8081 & 8082, but both are same microservice, you will see 2 instances in Eureka dashboard.



Note: Though you have 2 instances their logical name is still APP1, which is used by the microservice to communicate with APP1, through LoadBalancer it will distribute the load

You can write an System.out.println() statement in First Microservice so that you will see the print statement in all the instances whenever the client side load balancer distributes the request.



Note: You need to rebuilt the jar

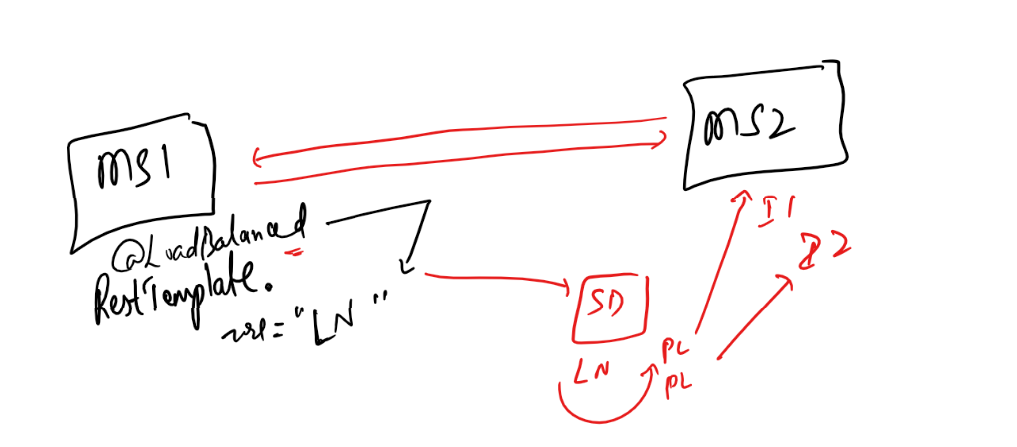
Communication from one microservice to another microservice

You can communicate using the standard way through RestTemplate and another way is to use FeignClient

RestTemplate: This is an instance used to access the webservices, you need to add the load balancer to it when you want to access microservice.

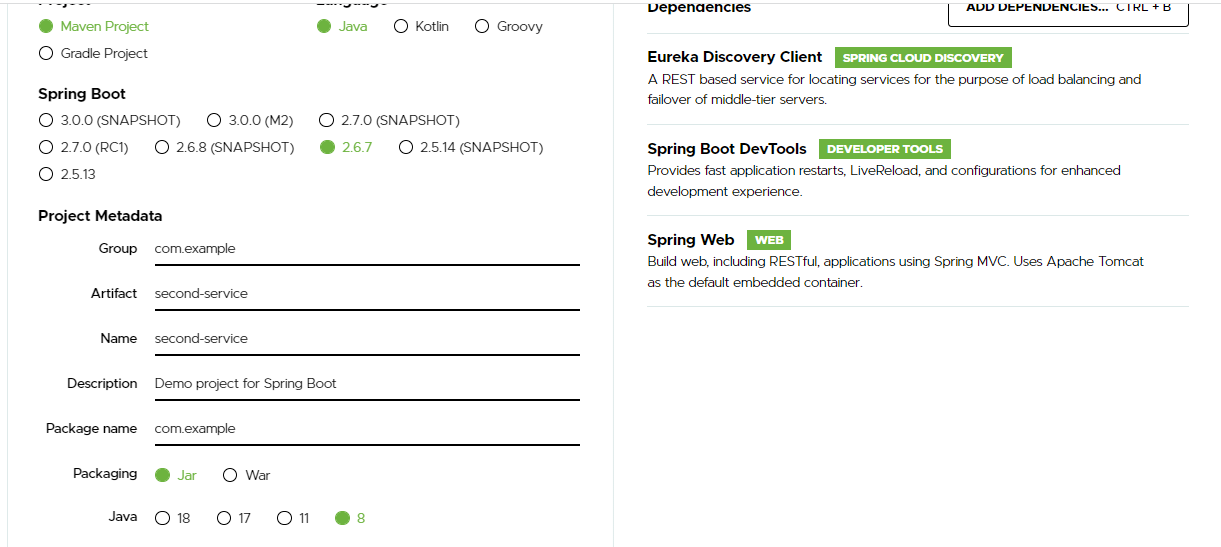
FeignClient: This is especially when you to access microservices through load balancer inbuilt

Accessing Microservices with RestTemplate



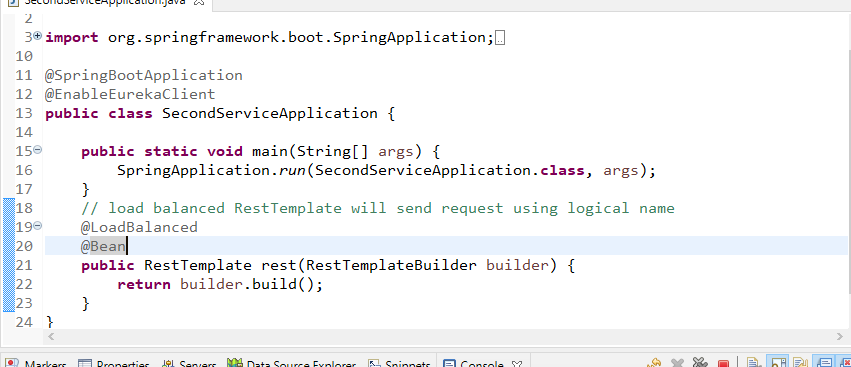
You need to use @LoadBalanced to access the microservice so that it knows all the instances of same microservices.

Create another microservice to access first microservice



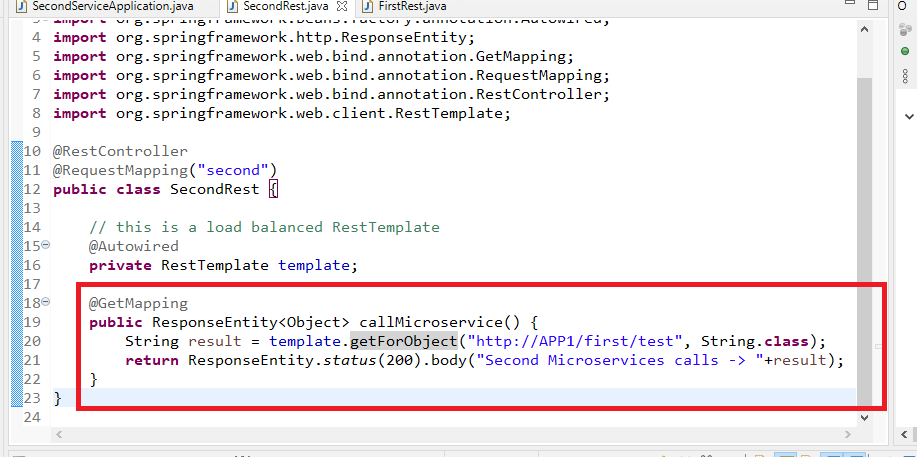
Since RestTemplate needs to be used to call a microservice we need to register it as a bean in the spring container so that you can use @Autowired wherever need

SecondServiceApplication.java



Now you can use @Autowired for RestTempalte and call the microservices using the logical name.

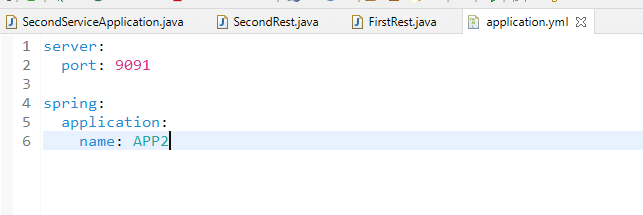
SecondRest.java



Note: Since the Microservice APP1 returns string we are using String.class, however if the Microservice response was complex type i.e., some JSON like {“name”:”Alex”, age:35 }, then in Second Microservice we can have a Java Bean that matches to the JSON structure and mention that in the getForObject(url, class).

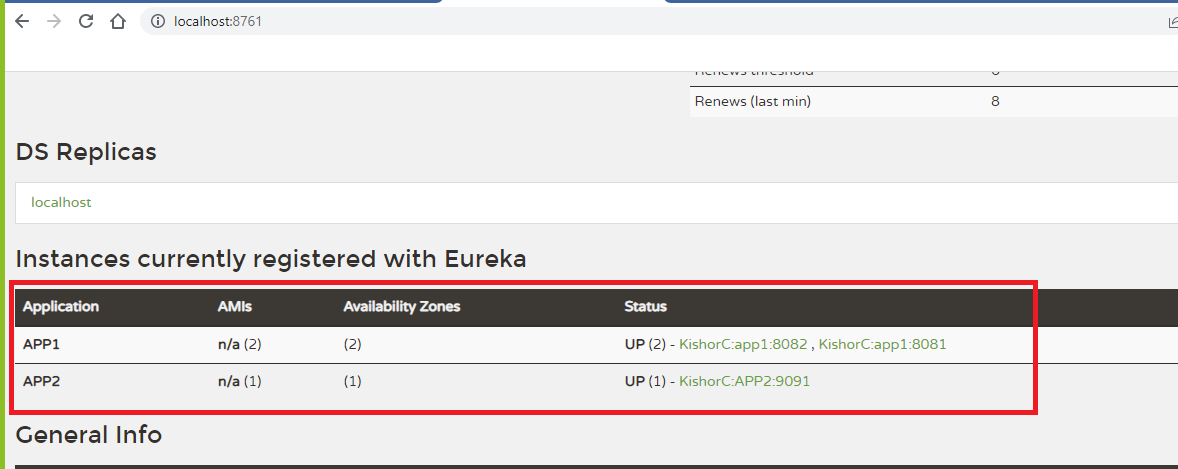
i.e., Suppose Person is the model that can be mapped with name & age of JSON structure then you can use getForObject as getForObject(url, Person.class)

application.yml



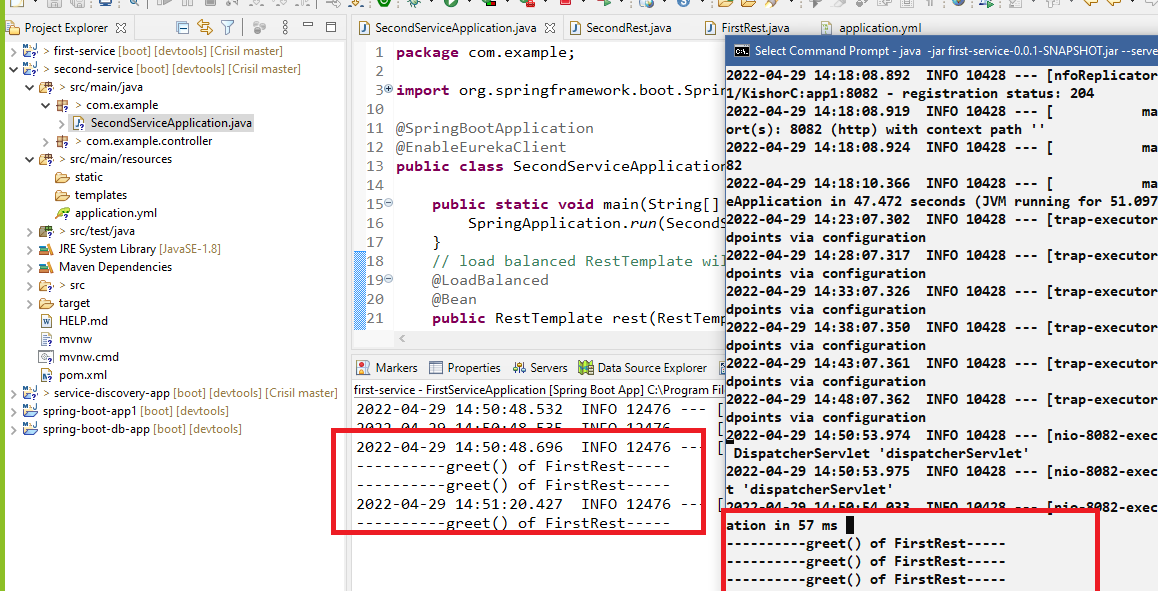
Now you can register this in Service Discovery, but if you access second microservice it internally access first microservice using logical name.

You can see 2 microservices in Eureka Dashboard



Output:

When you send multiple requests to second microservice you must see alternatively the load balancer is distributing to both the instances



Activity

1. Try the above activities
2. Create a Java bean in first microservice and instantiate manually i.e., using constructors / setters and return the JSON format through @GetMapping, from second Microservice you must able to get the JSON and covert that JSON to the Javaobject & show the JSON format as a response data.

i.e,. in first microservice if you have Account data i..e, { “no”:1324, “amount”:5000}, then the second microservice should able to use this response and convert to a Java Bean & use their value and same Java Bean must be returned in JSON format when you send request to the second microservice.

