Spring Microservices

Pre-requisites

* Core Java – OOPS & Design pattern (Factory pattern)
* Comparator & Lambda expressions
* Java 8 Streams – stream(), forEach(), filter(), collect(), sorted()
* Spring Framework – Dependency Injection & Annotations
* Spring Boot – Webservices & Configurations

Factory Design pattern

It is to abstract object creation at the client side

interface DBOperations {   
 void store();  
 void delete();  
}

class One implements DBOperations { }   
class Two implements DBOperations { }  
class Three implements DBOperations { }

Scenario1: Client creates object – leads to tightly coupled code

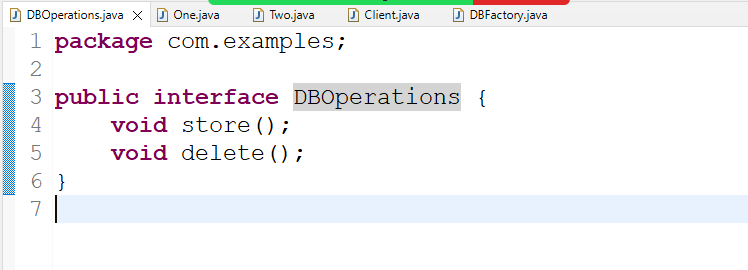
DBOperations db = new One(); // if new implementation must be used, then code must be modified here  
db.store();  
db.delete();

Scenario2: Client doesn’t create object – they use factory pattern to get the object – makes code loosely coupled

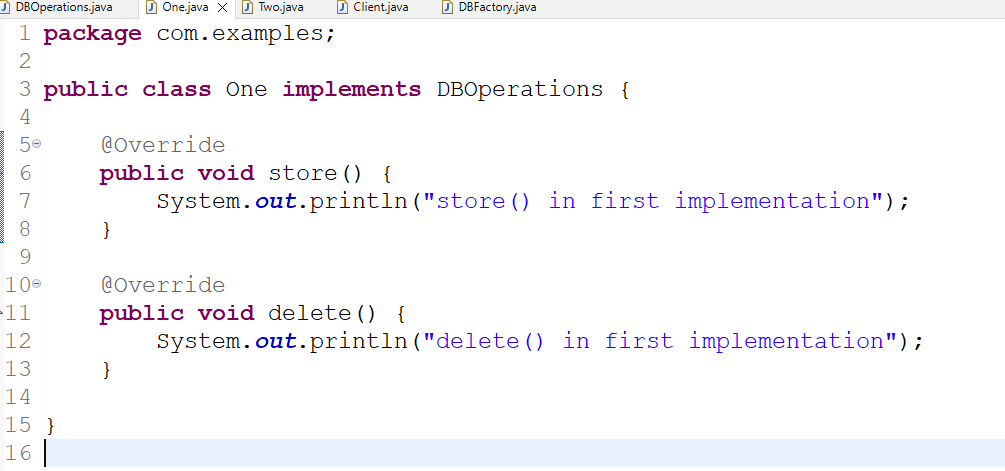
class DBFactory {   
 public static DBOperations getInstance() {   
 return new Two();  
 }  
}

DBOperations db = DBFactory.getInstance(); // client doesn’t know which implementation object is returned.  
db.store();  
db.delete();

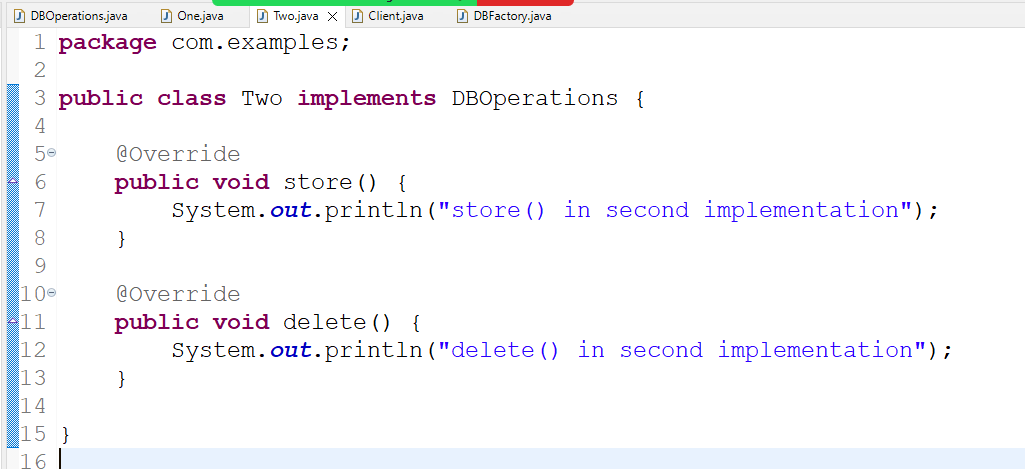
DBOperations.java



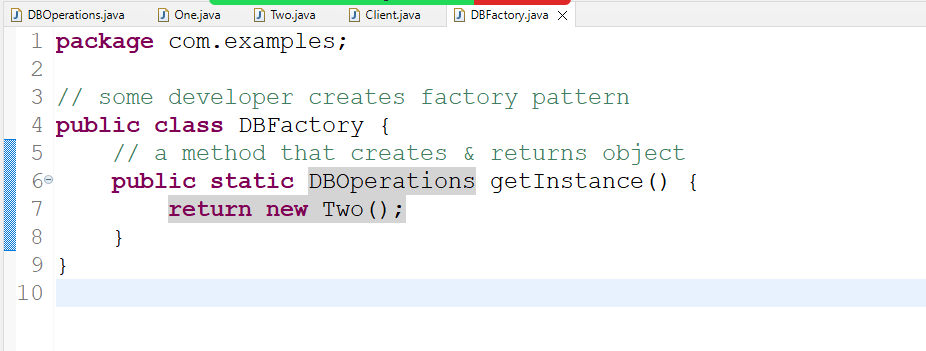
One.java



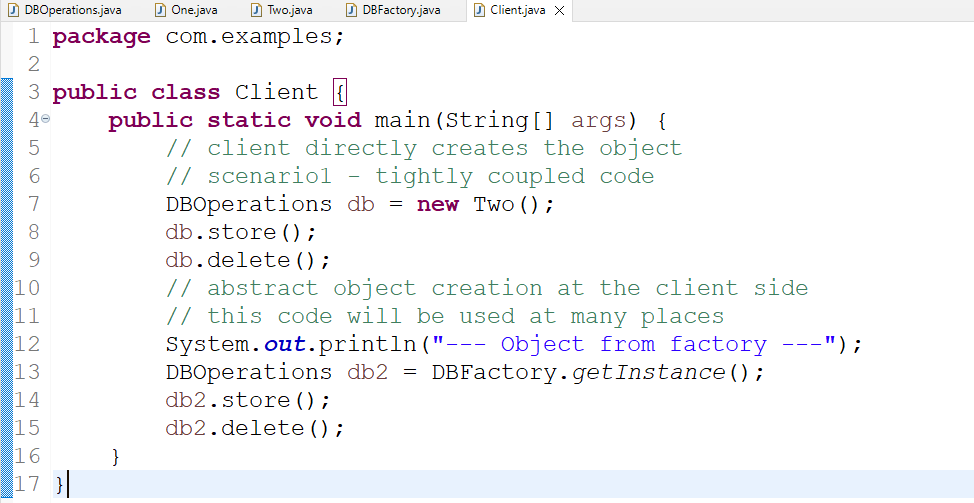
Two.java



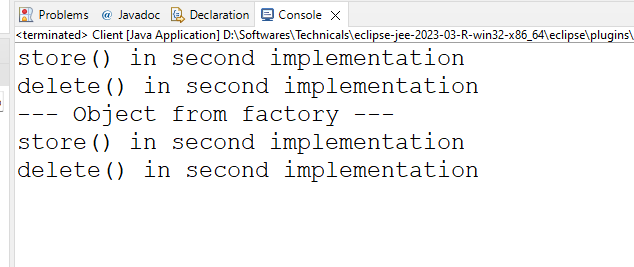
DBFactory.java



Client.java



Output:



Summary:

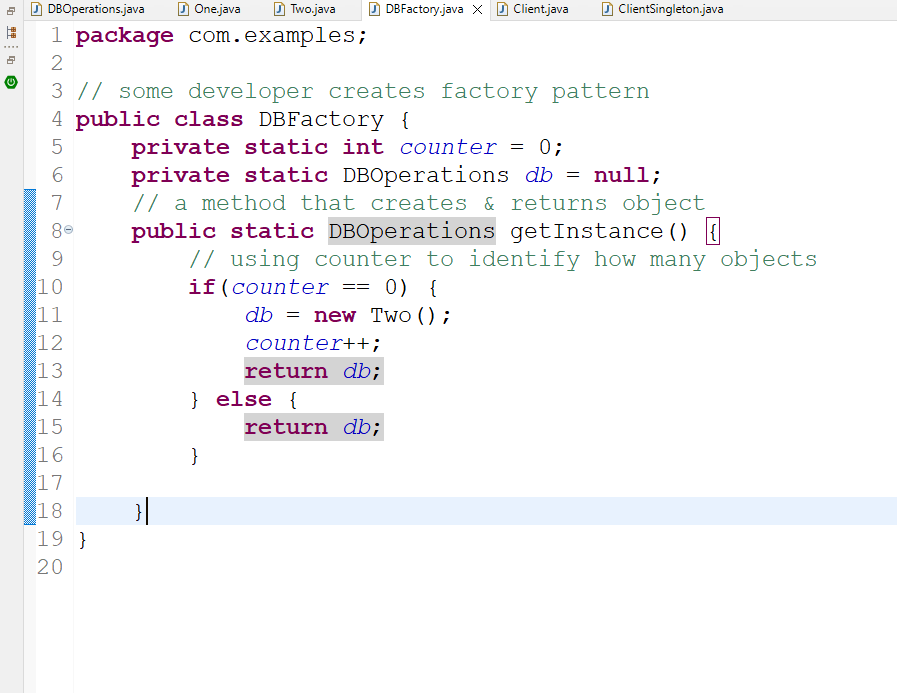
Developers who create object directly need to change their code when a new implementation needs to be used, developers who gets the object from the factory pattern need not to change the code because factory pattern takes care of giving the object.

Factory pattern vs Singleton pattern

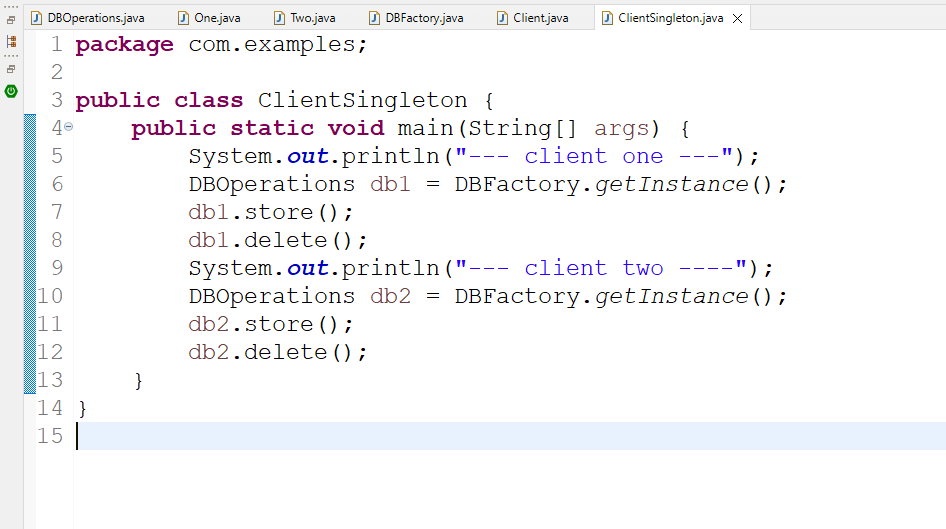
Factory pattern just creates the object, however singleton pattern is a factory pattern where on a class only one object will be created even if you call the factory method more than once

Previous factory pattern creates object more than once based on the how many time you call the getInstance() factory method, this leads to more number of object creation, to avoid this we can change the factory method to give only one object regardless of how many times you call the factory method

Modifying the factory to return a singleton object



ClientSingleton.java



Spring Framework:

Framework is like a semi implemented application which provides all the common features every application needs like

1. Type conversion: String to Number to String & Java types to SQL & vice versa
2. Design patterns:
3. Object Creation & Initialization
4. Exception Handling
5. Transaction Management
6. Connection Pooling

Spring Framework is an application framework which helps you to create various types of applications

1. Web / Webservices
2. Desktop
3. Cloud based application
4. Console based application

Spring Framework provides many modules to develop the above applications

1. Spring Core
2. Spring Web
3. Spring Boot
4. Spring Cloud
5. Spring Batch

Spring Core: It is the base modules that takes care of all the features every application needs like

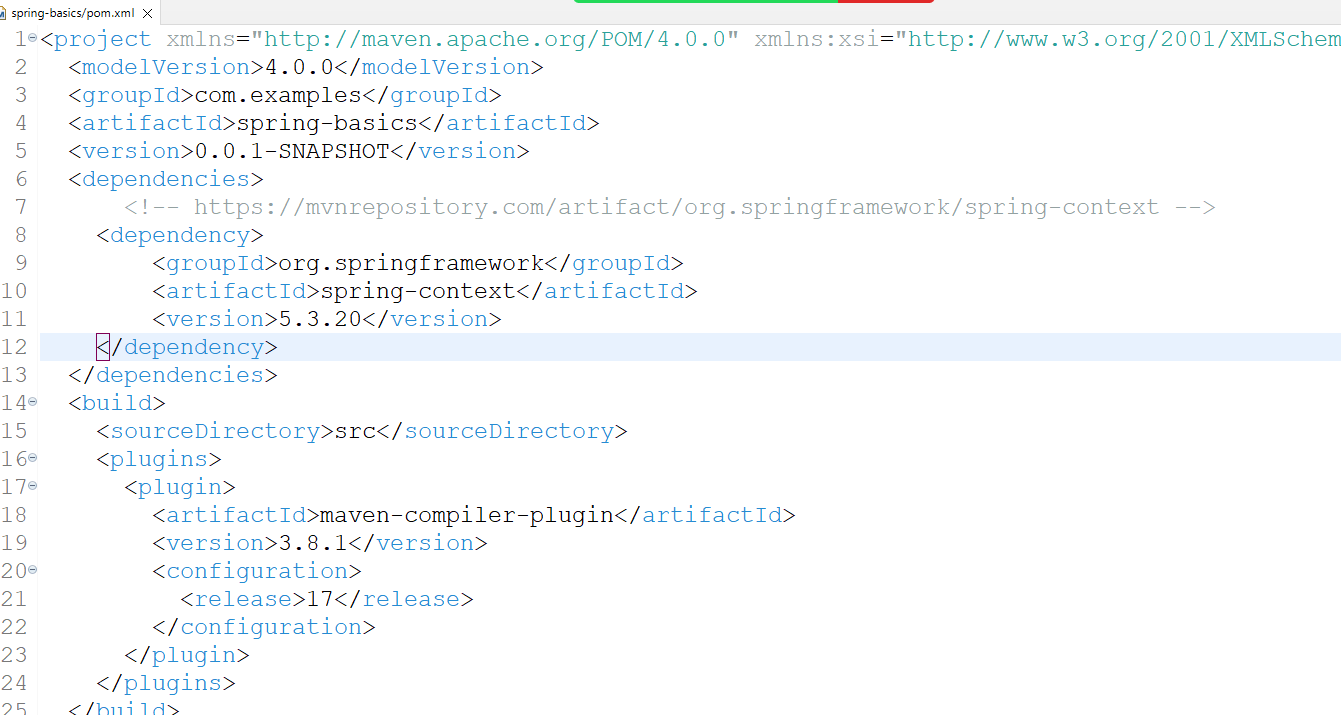
1. Design pattern
2. Object creation

Dependency Injection: It is a process of supplying an object to another object

Spring framework uses factory pattern internally to create objects so that developers don’t have to implement factory pattern

Spring Context: This is the library which you need to use to get the benefits of spring framework for Dependency Injection

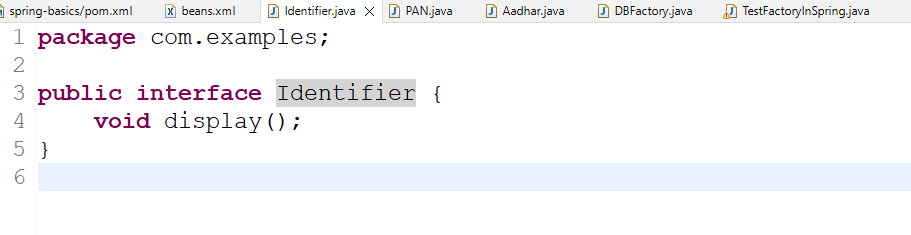
pom.xml



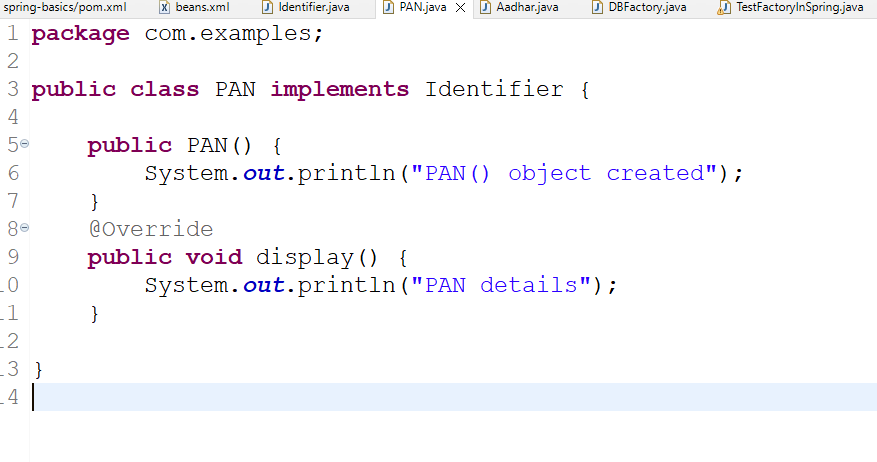
Project Structure

1. classes
2. XML – declare all the beans (java classes) whose object must be created by spring container

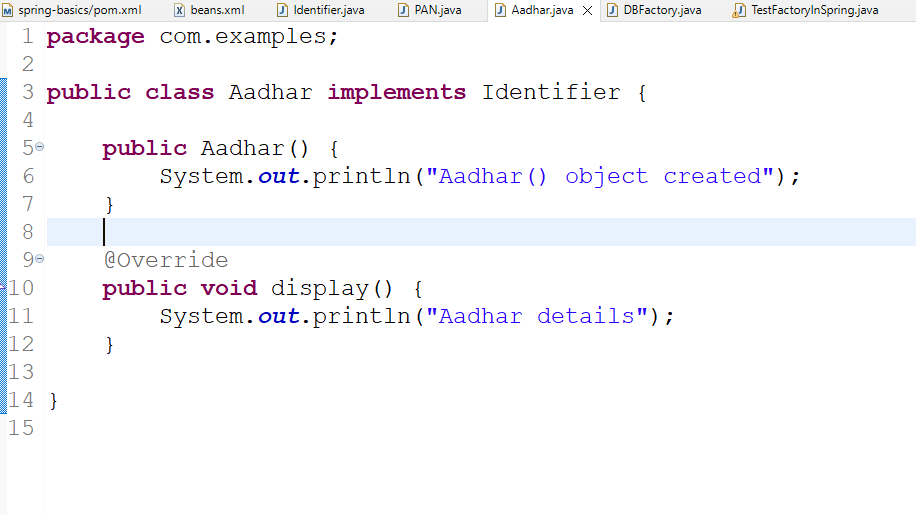
Identifier.java



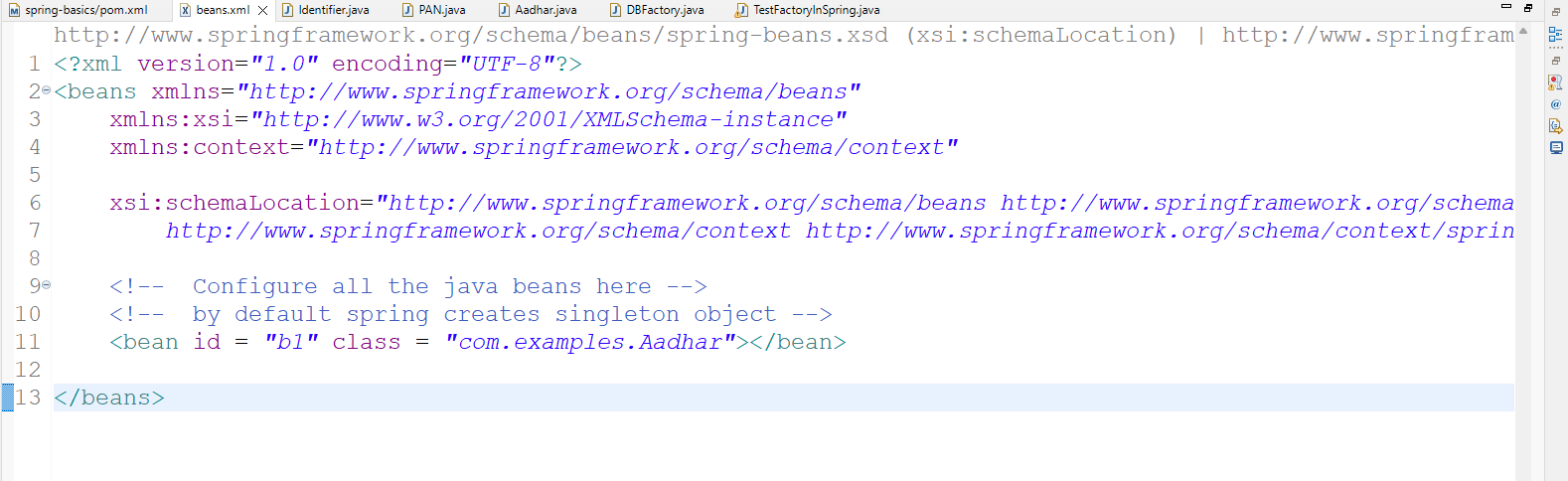
PAN.java



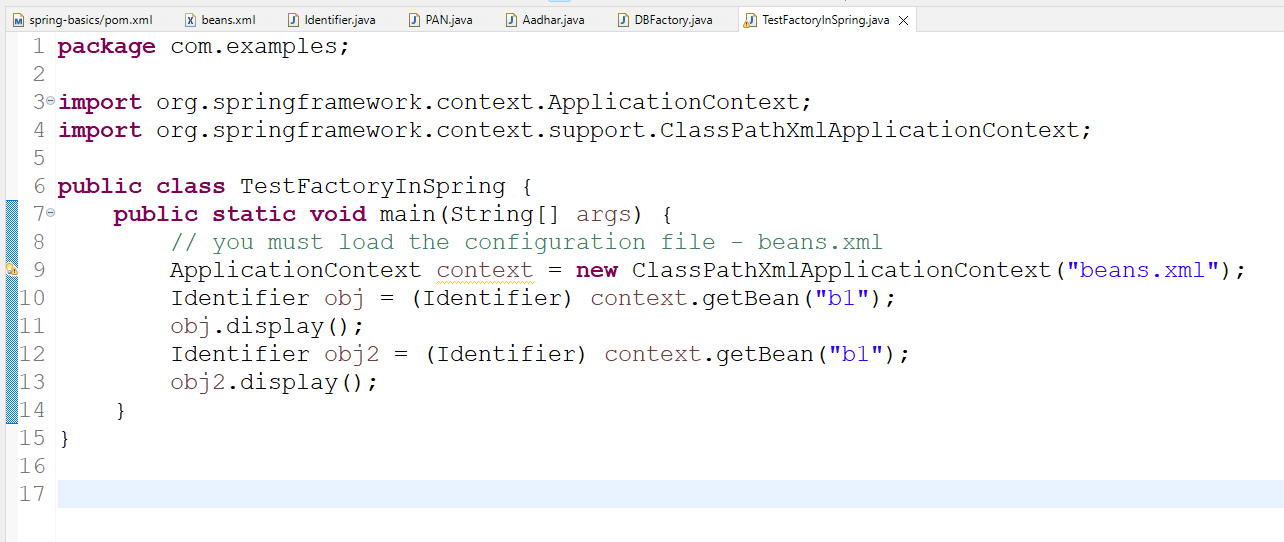
Aadhar.java



beans.xml

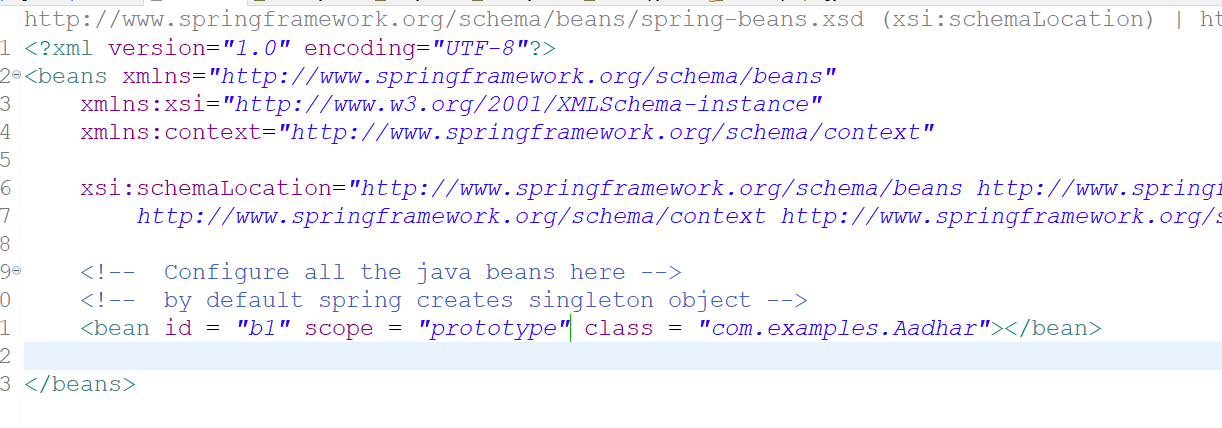


TestFactoryInSpring.java



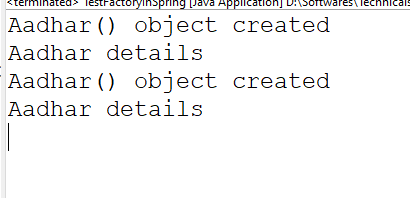
Note: At present spring container is creating singleton object, but you can make it to create multiple objects, for that you must use scope in the xml

beans.xml



scope is singleton by default, when it is prototype spring creates multiple objects for each getBean() invocation.

Output:



Annotation based configuration

It simplifies configuring the spring beans with simple annotations so that you can avoid declaring beans in the XML file

List of annotations spring provides to create the object

@Component  
@Service  
@Repository  
@RestController

@Service  
public class EmployeeService { } // id will be class name but first letter will be lowercase

This is equal to <bean id = “employeeService” class = “com.examples.EmployeeService”>

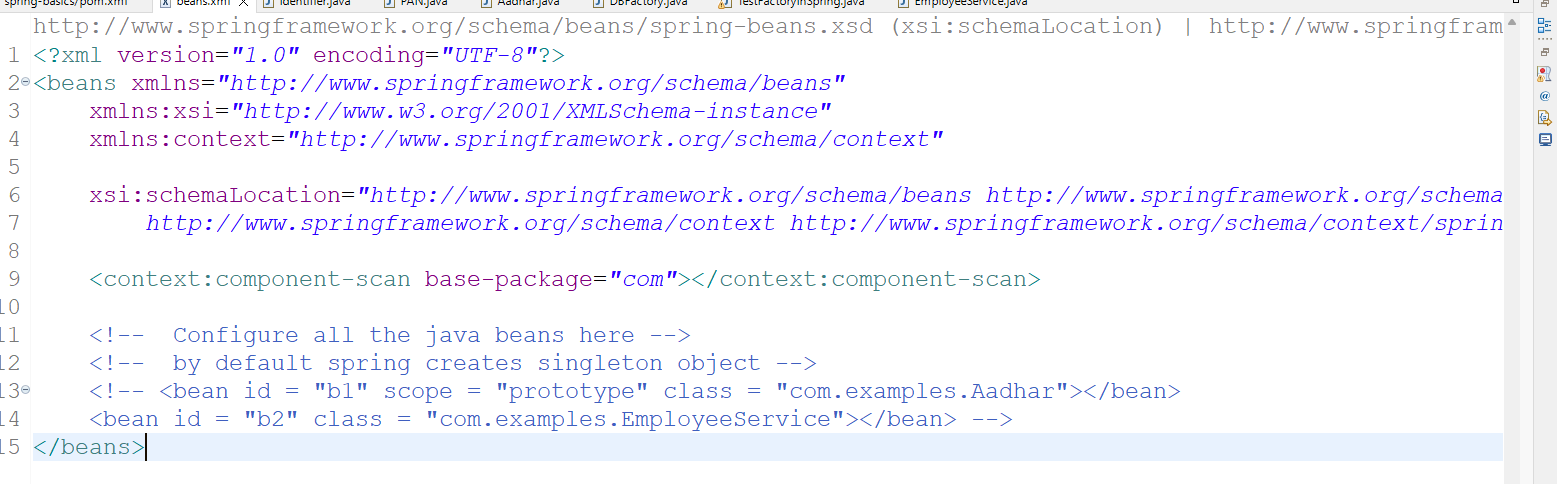
@Repository  
public class EmployeeDao { }   
This is equal to <bean id = “employeeDao” class = “com.examples.EmployeeDao”>

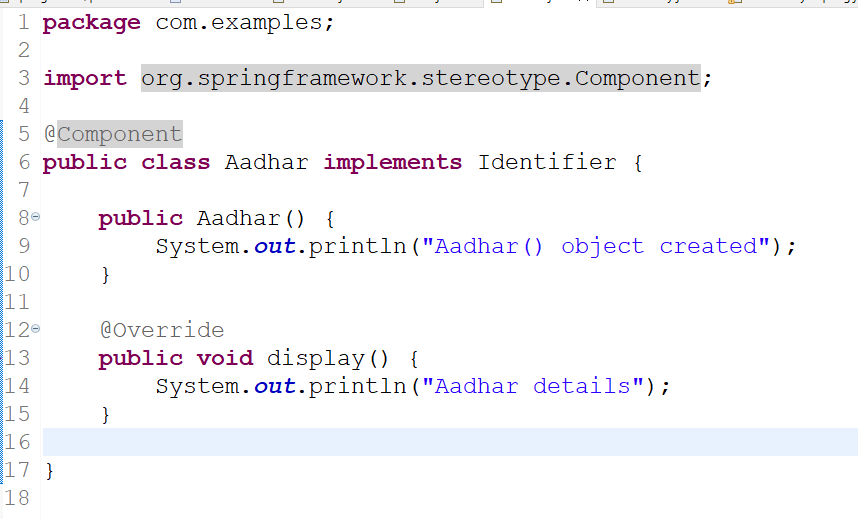
@Component is a base annotation where as @Service, @Repository, @RestController, @Configuration, @Controller are all derived from @Component

Note: If spring needs to search the classes having these annotation then in XML you need to use one tag to scan all the classes having these annotation

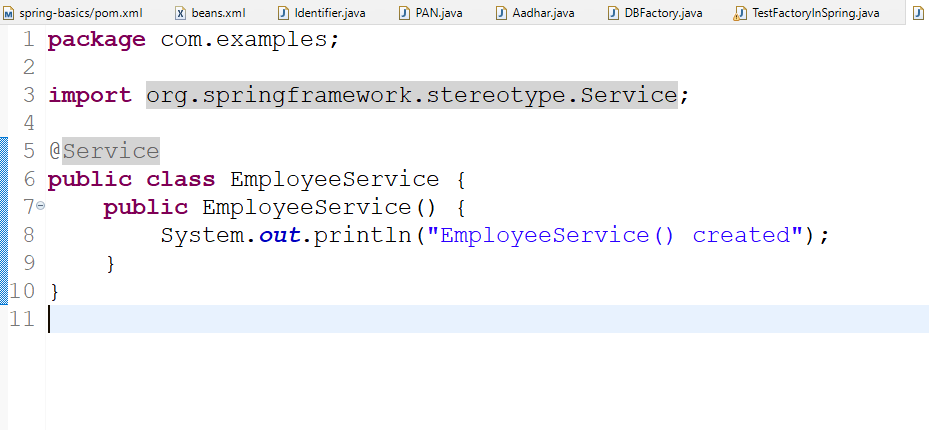
<context: component-scan base-package = “com.examples” />

beans.xml

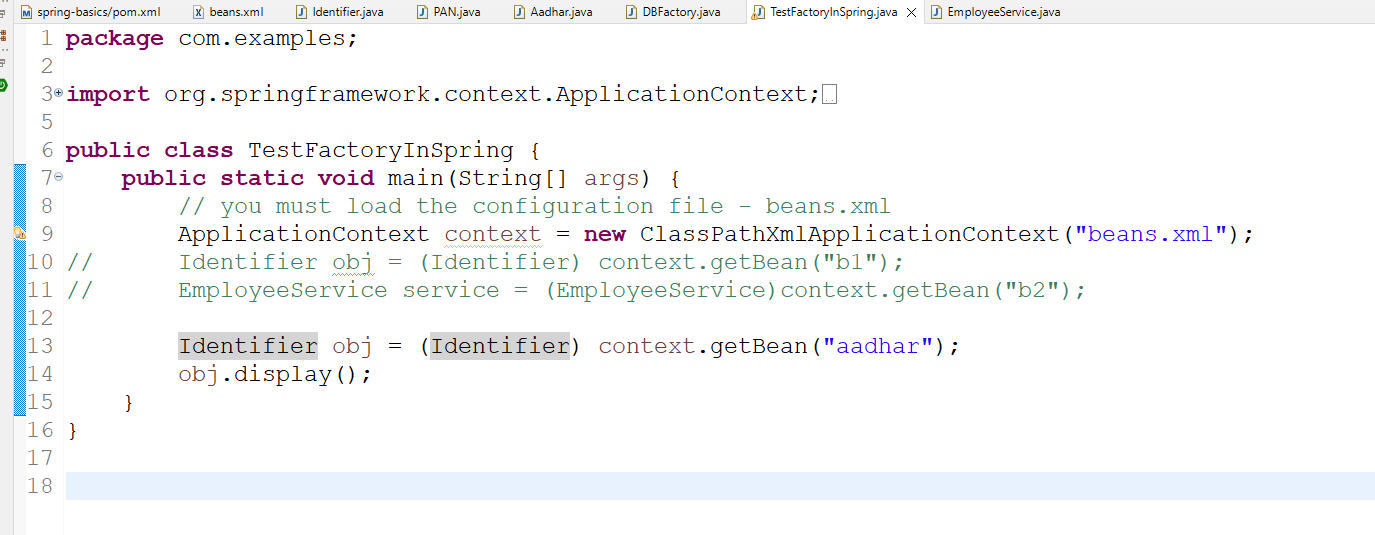




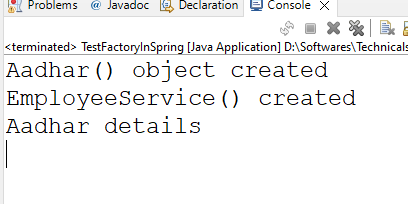
EmployeeService.java



TestFactoryInSpring.java



Output:



Spring is going to scan all the classes having @Component or their derived annotations to create the object.

Spring Boot

It simplifies developing the spring applications by auto-configuring your application based on the library you add

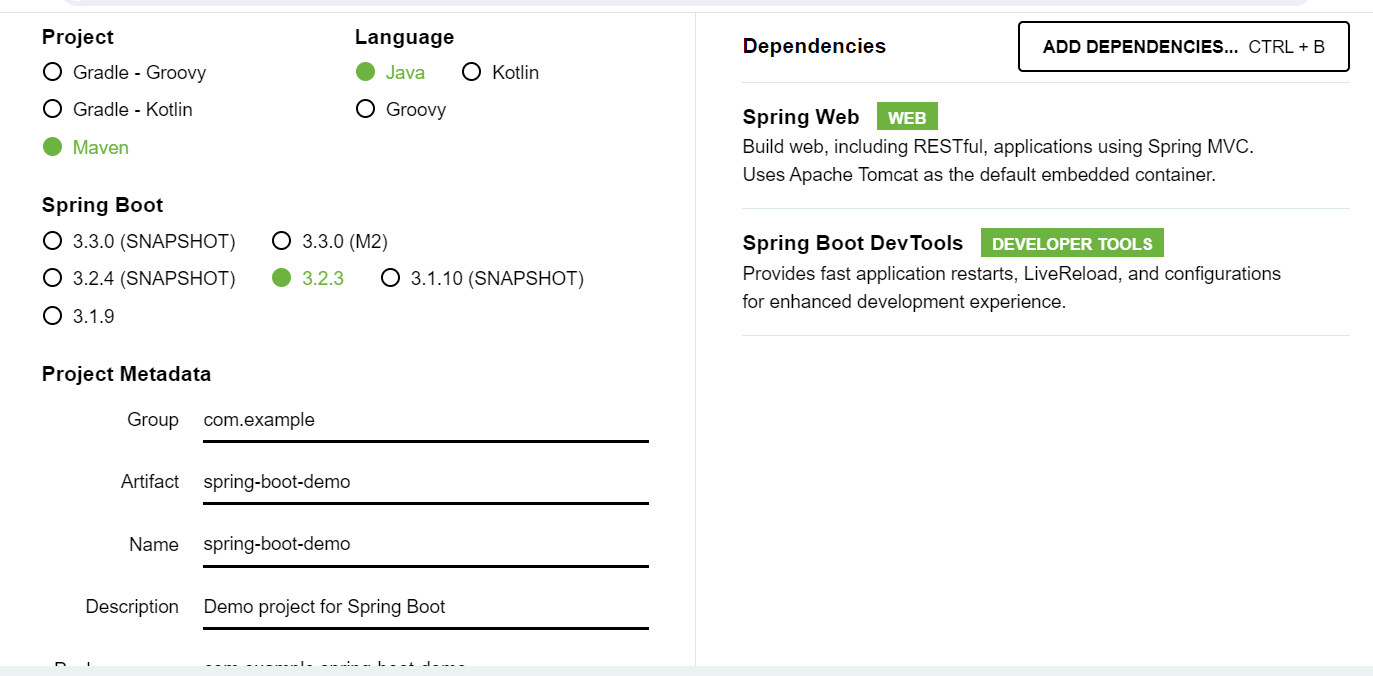
Spring boot takes care of configuring all the generic setup

* Server configurations
* Database configuration
* Component scanning
* Front controller configuration

Spring boot does this using the libraries which will have name as spring boot starter

ex: spring boot starter web, spring boot data jpa,

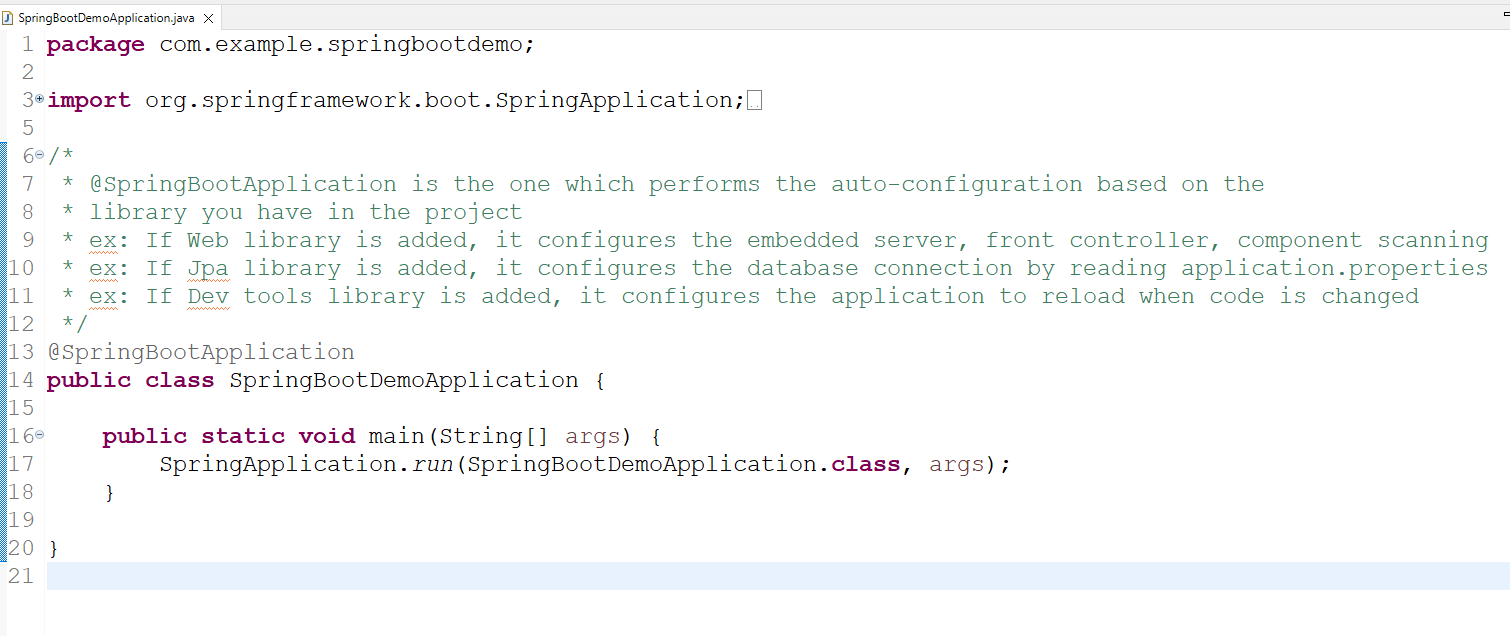
spring initializr: it is a official website to download empty spring boot project



When you download you will get a zip file, which you can directly open from eclipse, by following these steps

File -> Open Projects from file system -> In Import Source -> Archive -> zip file -> Select maven project & uncheck the folder not having maven

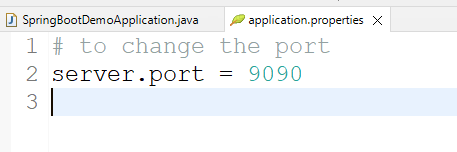
Every project of spring boot will have a main class with @SpringBootApplication that does the following job which is mentioned in the comments



You can get all the properties of spring boot in the below URL

<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.data>

application.properties



Webservices:

These are online services that helps heterogenous applications to share the data

ReSTful webservice exchanges the data in a common format (JSON)

ReST stands for Representational State Transfer

ReSTful uses two things to allow applications to exchange the data

1. URL: To locate the webservices
2. HTTP methods: To map the operations using GET, POST, PUT & DELETE

To create webservices in Spring we have some annotations

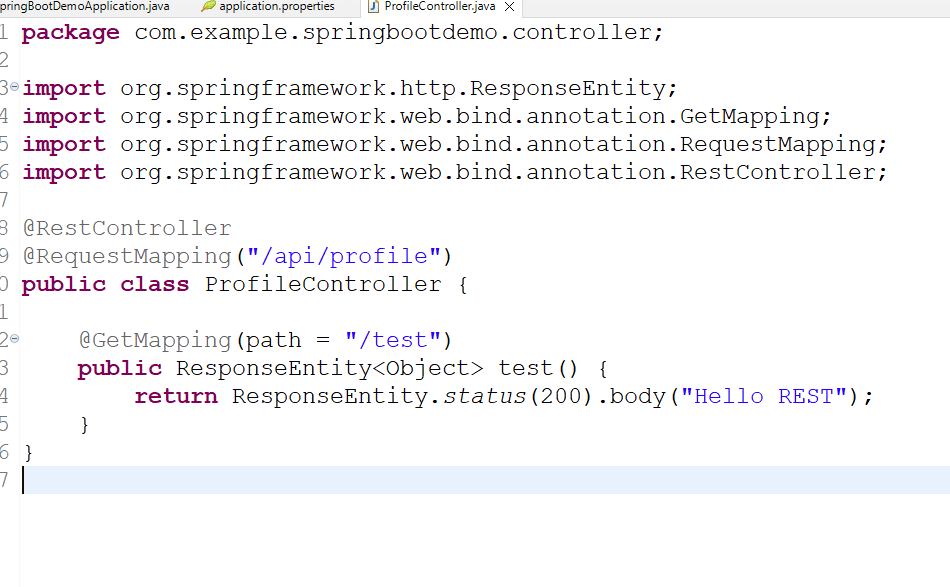
1. @RestController: This is to create a controller class which will have webservice code
2. @RequestMapping: This is to configure the URL for your controller
3. @GetMapping: This is to map HTTP GET
4. @PostMapping: This is to map HTTP POST
5. @PutMapping: This maps HTTP PUT
6. @DeleteMapping: This maps HTTP DELETE

@RestController  
@RequestMapping(“/api/profile”)

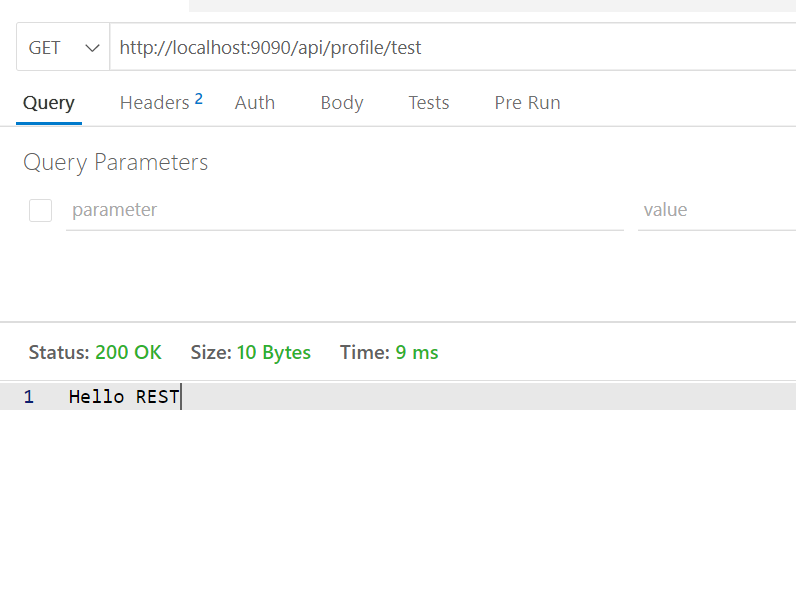
class ProfileController {   
 @GetMapping(path = “/test”)  
 public ResponseEntity test() {   
 return ResponseEntity.status(200).body(“some data”);  
 }  
}

Client uses: GET http://ip:port/api/profile/test

ProfileController.java



Output:



Different ways of passing the data to the webservice

1. URL: You can pass data via url path – simple data
2. Body: You can pass data via request body – JSON

api/profile/100

api/profile/200

api/profile/300

@GetMapping(“/api/profile/{x}”), @PostMapping(“/api/profile/{x}”)  
@PathVariable(“x”) int id;

@PathVariable(“x”) is going to extract the value associated with x and injects to the variable

The above code is sending the data via URL path

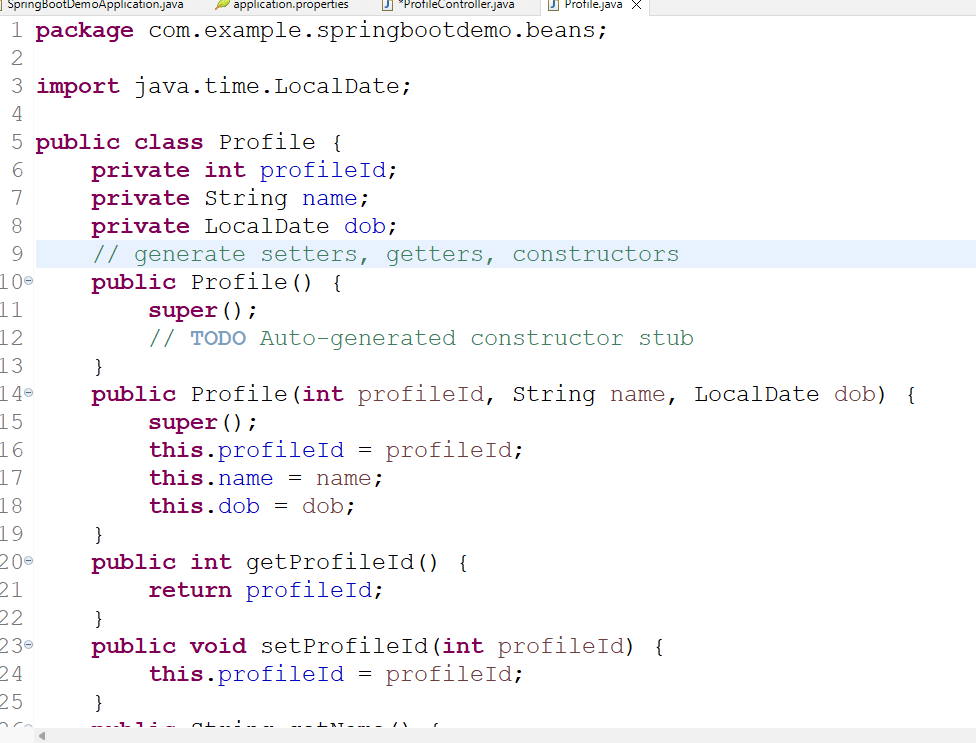
When you are entering complex data it is send in JSON format  
{ “name” : “Abc”, “phone”:99999, “dob”:”1998-10-25” }

@RequestBody: This annotation extracts the data from the request body, and maps to the java object

store(@RequestBody Profile pr) { }

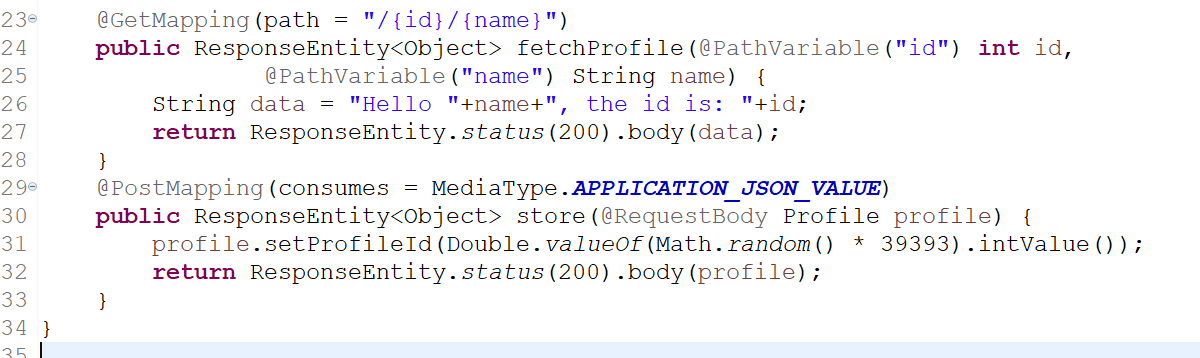
class Profile { name, phone, dob }

Profile.java

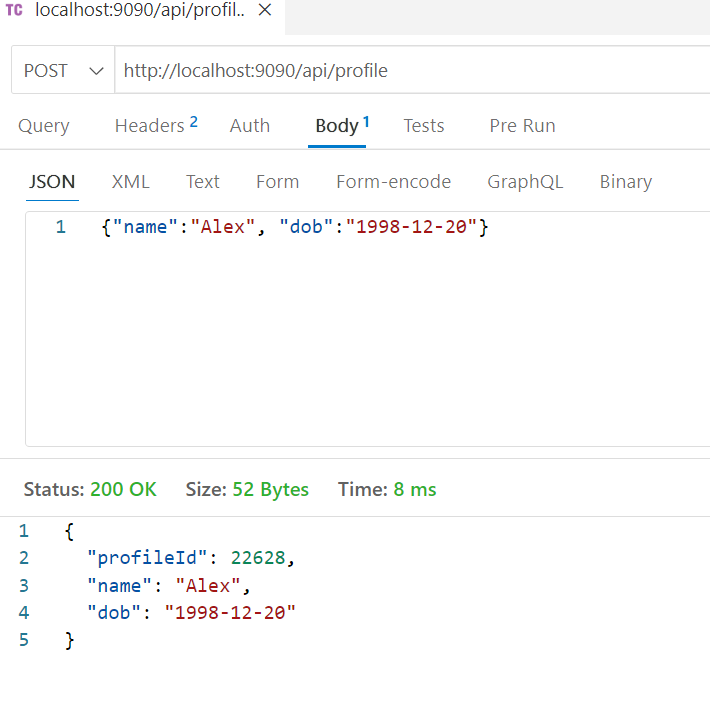


Create a webservice that accepts data in the url & the request body

ProfileController.java



Output:



Spring Data Jpa

It is a library which can automate DB connections & DB logics

For DB connections you must provide DB credentials in the application.properties

For DB logics it provides some inbuilt interfaces called Repository interfaces to that you must provide the entity class details

Entity Class; It is a class that provides table & primary information’s

Profile table : id, name, dob

@Entity  
@Table(name = “profile”) // optional if class name & tables names are same  
class Profile {   
 @Column(name = “id”) private int pid;  
 @Column(name = “name”) private String pname;  
 @Column(name = “dob”) private LocalDate pdob;  
}

You can name your properties same as column names to directly map the object to the table without using @Column

@Entity  
@Table(name = “profile”)  
class Profile {   
 @Id  
 private int id; // maps to the column having the name id  
 private String name; // maps to the column having the name ‘name’  
 private LocalDate dob; // maps to dob

}

@Id: We need to use this on a property that maps to the primary key column, because in spring data jpa DB logics are automated, many queries are generated based on the primary key like find by primary key, delete by primary key, update by primary key

Repository interfaces: These are the inbuilt interfaces that will make Spring Data JPA to automate the DB logics like CRUD operations based on the entity, these interfaces provide some inbuilt methods which can perform CRUD operations on the entity which reflects to the table

1. CrudRepository<T, ID>: gives methods like save, findById, deleteById, findAll
2. JpaRepository<T, ID>: extends CrudRepository and provides methods to sort, saving multiple objects and etc.

T: Is an entity class type

ID: Is a primary key class type

CrudRepository<T, ID> has following methods

1. save(T): Here it saves the entity
2. findById(ID): Here it accepts the id and returns the entity matching to the id
3. deleteById(ID): Here it accepts the id & deletes the entity matching to the id
4. findAll(): It returns all the entities in List<T>

Our job is to inherit either CrudRepository or JpaRepository, but spring boot automates the implementation of the interface you create.

interface ProfileRepository extends CrudRepository<Profile, Integer> {   
   
}

Now ProfileRepository gets all the methods of CrudRepository in the form as

save(T) = save(Profile)

findById(ID) = findById(Integer)

All the methods of ProfileRepository will auto-implemented so that when you call the methods like save, deleteById, findById the reflect to the profile entity i.e., profile table.

Service layer must only use the object of this auto-implemented class

@Autowired  
private ProfileRepository profileDao;

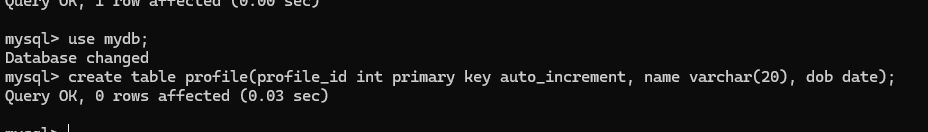
Things to create

1. Profile entity: Using @Entity, @Id, @GeneratedValue
2. ProfileRepository to extend CrudRepository<Profile, Integer> or JpaRepository<Profile, Integer>
3. Checked Exception: To throw from the business layer so that controller will hande
4. Autowire the ProfileRepository in the service layer
5. @Service: Service layer must have methods to call ProfileRepository methods like save, deleteById, findById, findAll
6. Controller layer must autowire service layer
7. application.properties: Must have database credentials

Libraries

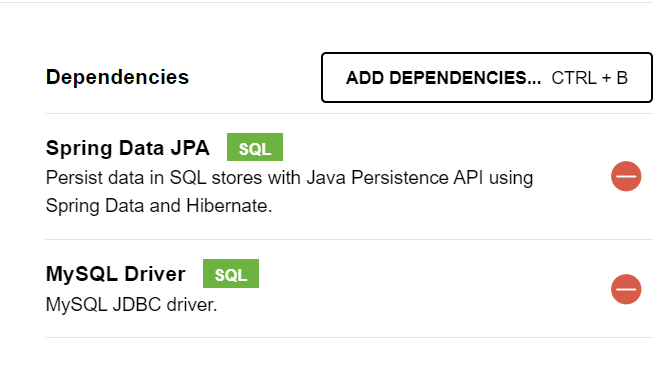
1. Web
2. JPA
3. MySQL

Create a table with primary key in mysql



Since database is in localhost, the url will be jdbc:mysql://localhost:3306/mydb

Adding the JPA & Mysql library



Explore pom.xml & copy the dependency tags to your existing pom



pom.xml

..



…

Firstly we will configure application.properties

application.properties



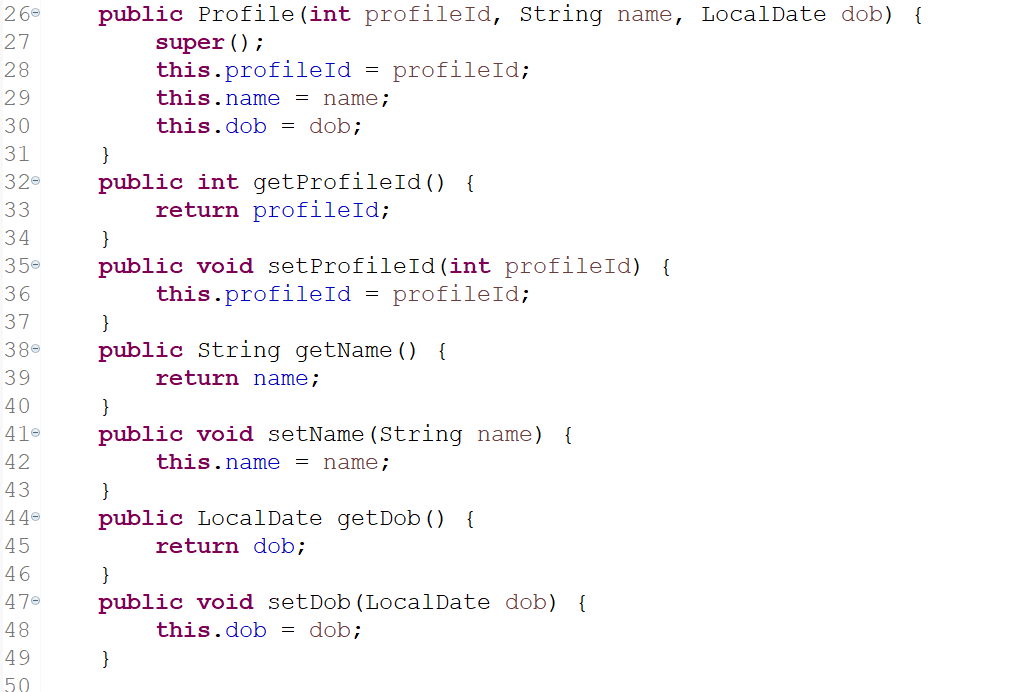
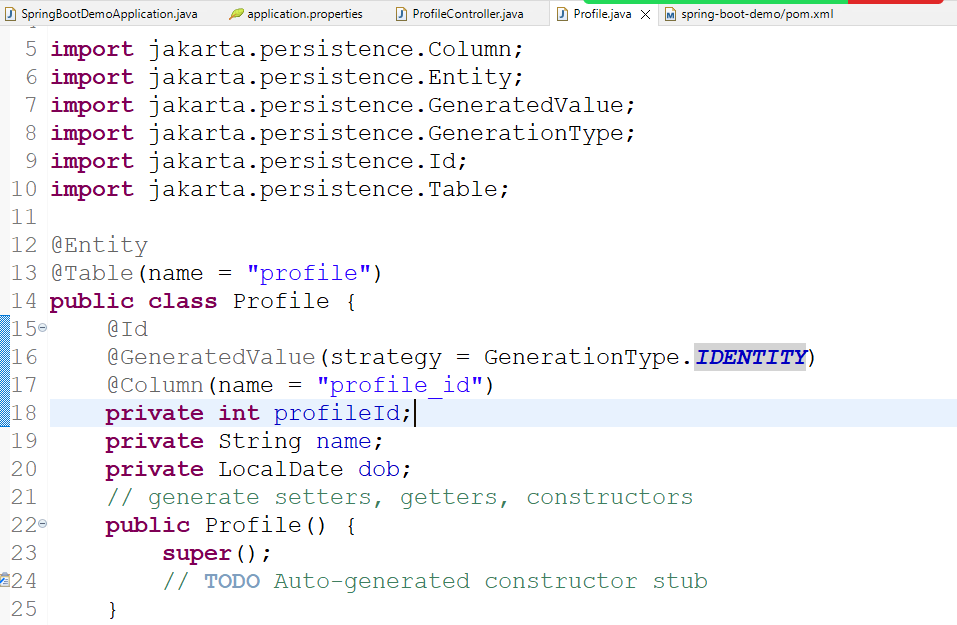
You can run the program to confirm the db connection, this connection will be present in the spring container which will be automatically wired to the DAO layer

In the console you see connected added



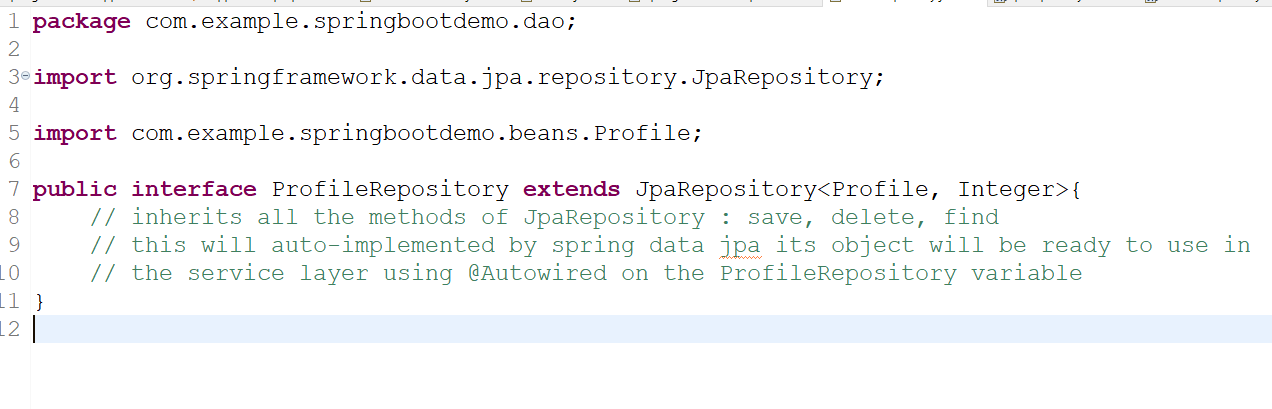
Creating an entity class with @Entity, @Table, @Column, @Id, @GeneratedValue

Profile.java



You need to create an interface in the DAO layer that will be auto-implemented & instantiated in the spring container, it must extend either CrudRepository or JpaRepository<Entity, ID>

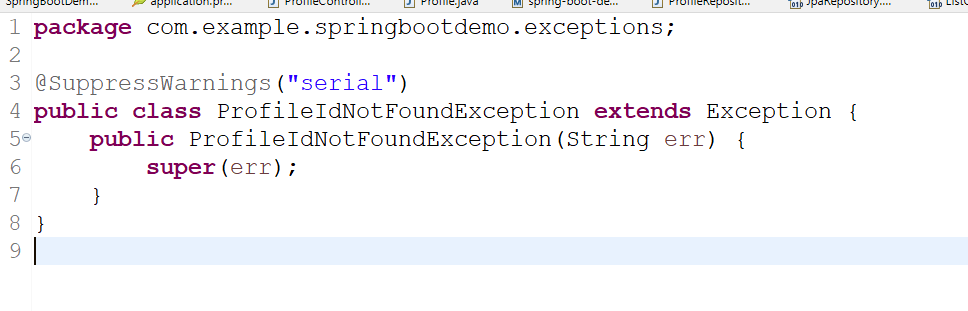
ProfileRepository.java



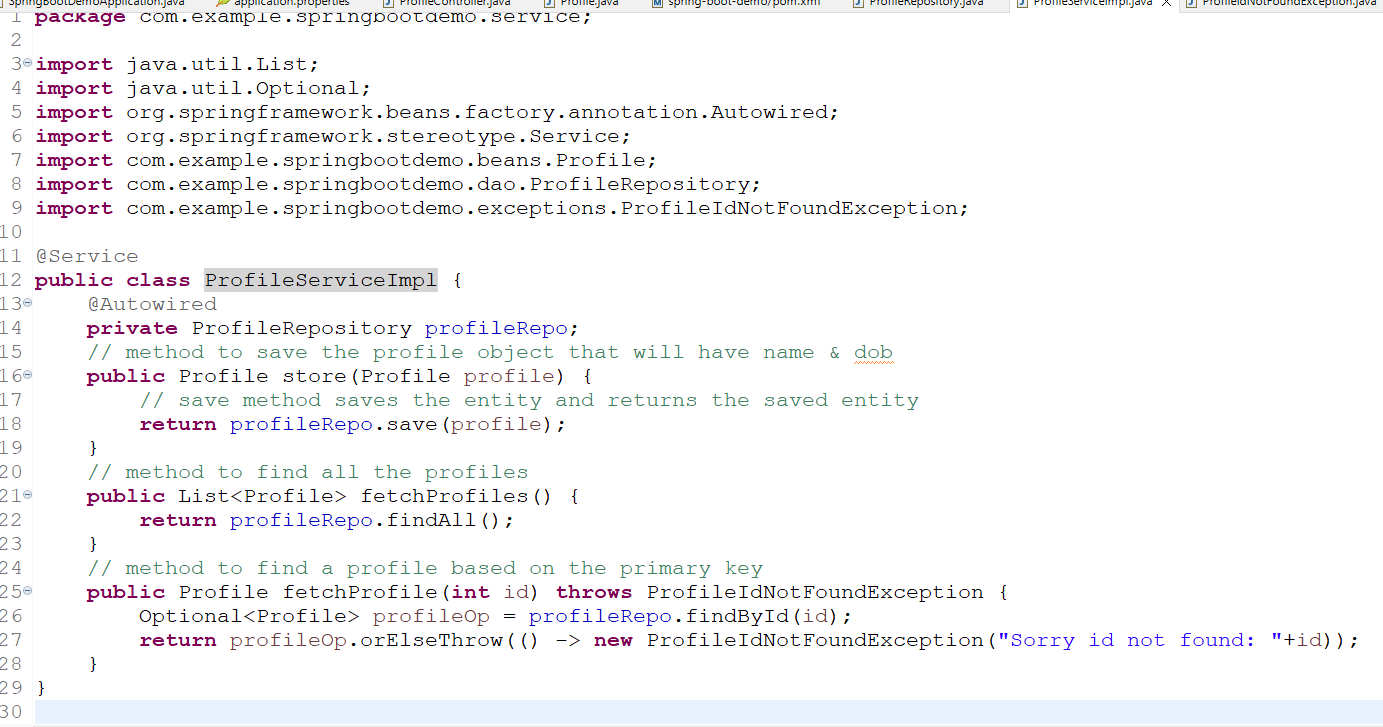
Now we can create a service layer that can access methods of DAO, its advisable to create an interface in the service layer & implement so that controller will not have an idea about the service layer implementation

Note: You can create a class and implement the methods you want and then you can extract the interface from the class using eclipse

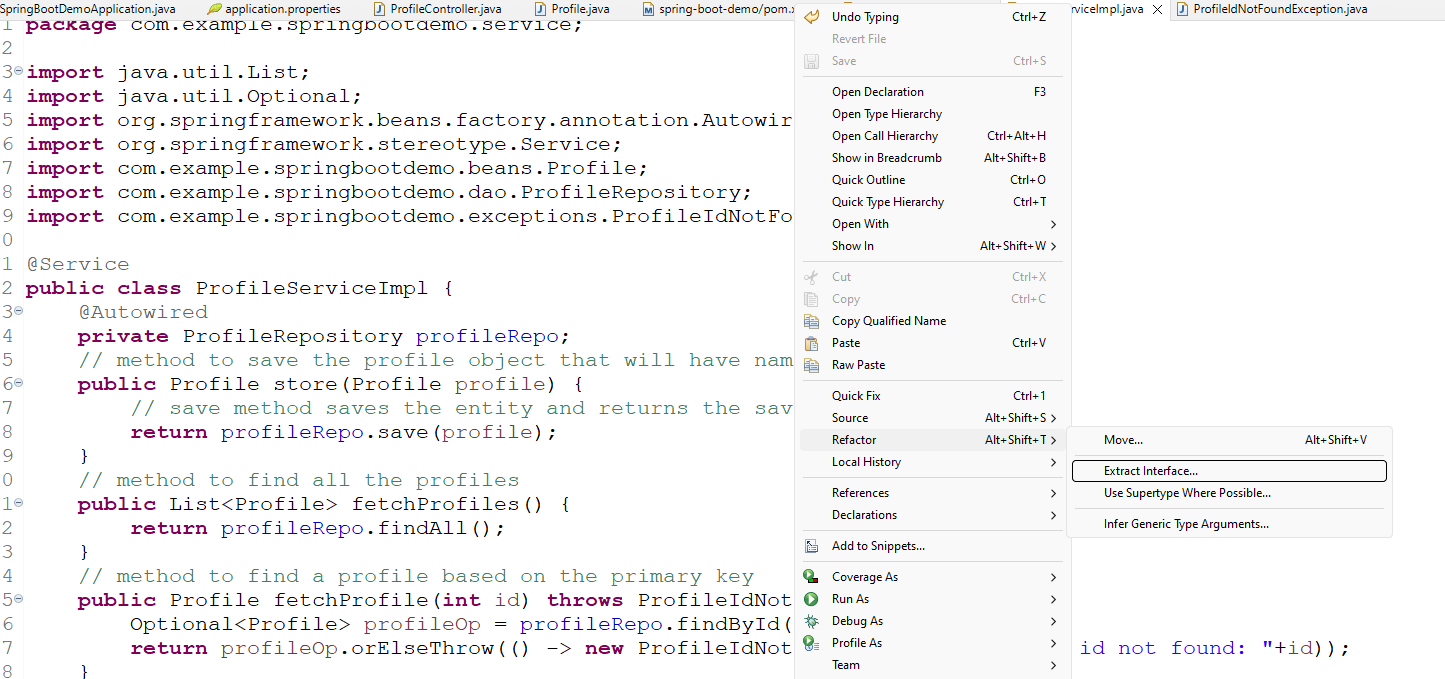
ProfileIdNotFoundException.java



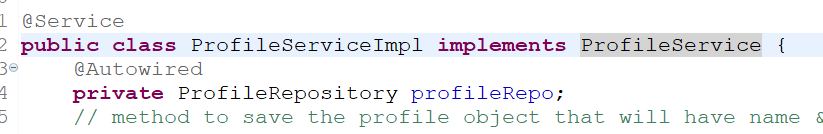
ProfileServiceImpl.java



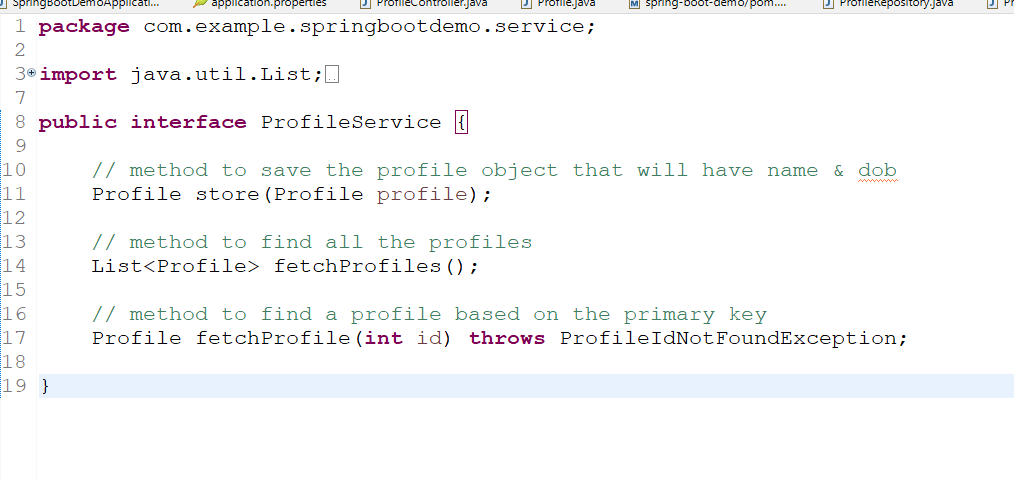
We must give the interface to the client code (i.e., controller) hence we must create interface for this class using Refactor -> Extract Interface



You can name the interface like ProfileService so that a ProfileService.java will be created and the class will have implements in the ProfileServiceImpl.

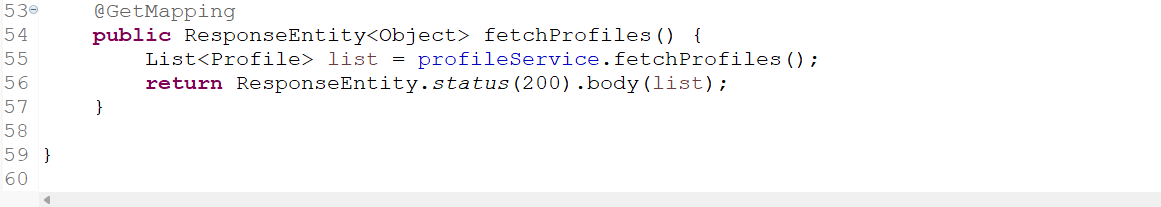
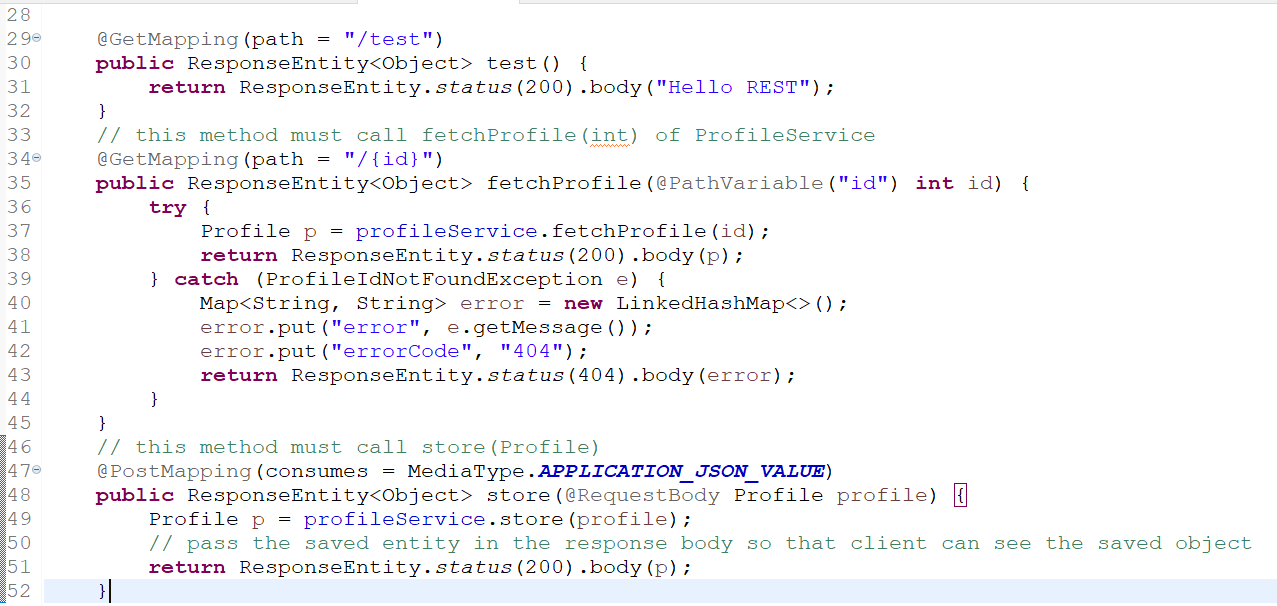
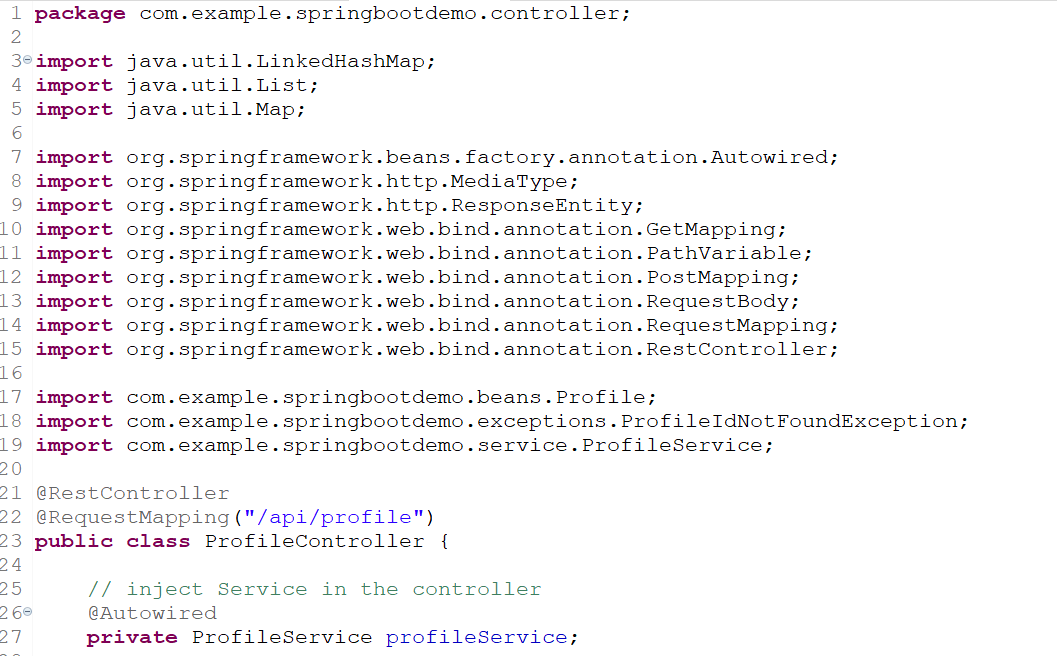


You can also see ProfileService.java

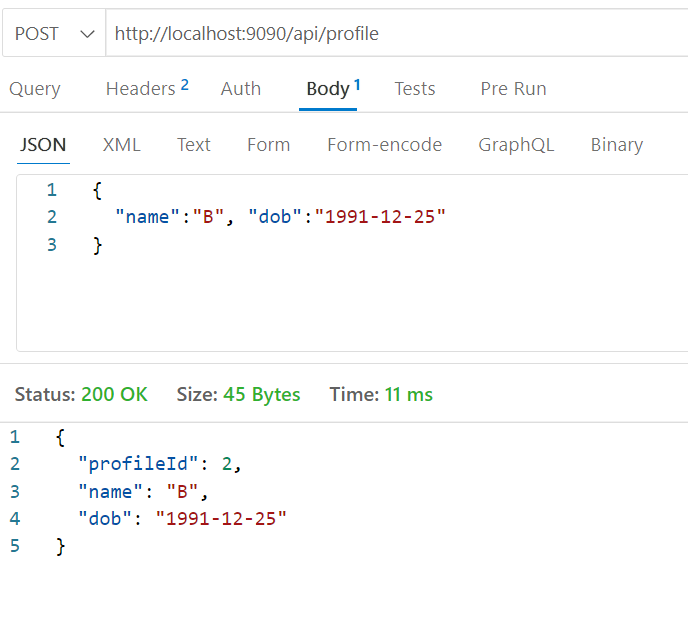


Now RestController must access these methods by @Autowired on ProfileService

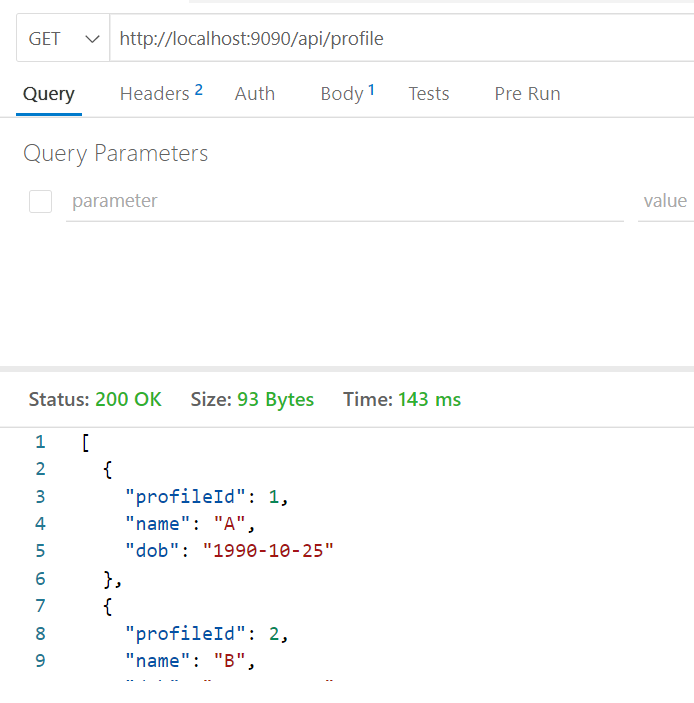
ProfileController.java



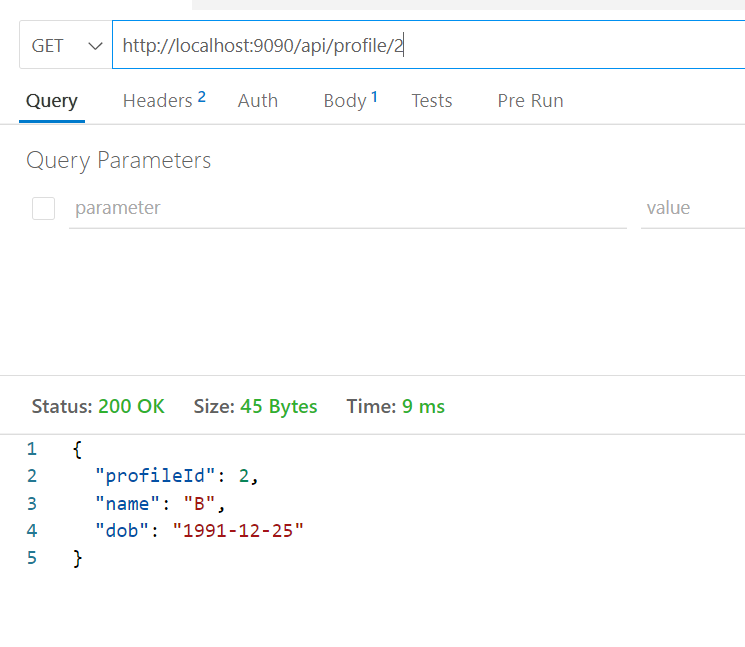
Now you can perform save & retrieve operation

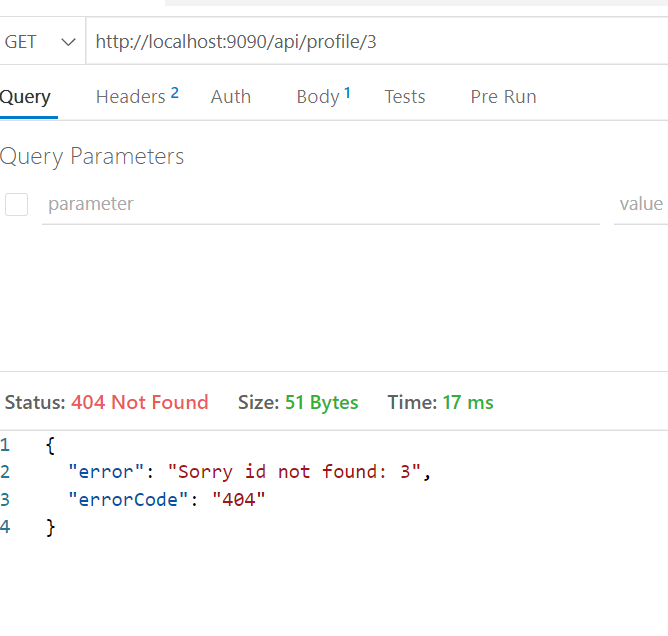


See all the profiles



See profile by id





Activity:

Create deleteById method in the service and call that method from the controller so that it should delete the profile or show profile id not found

How to apply joins: In Spring Data JPA you have annotations that can automatically join the entities which in turn joins table, it eliminates user to write complex join queries.

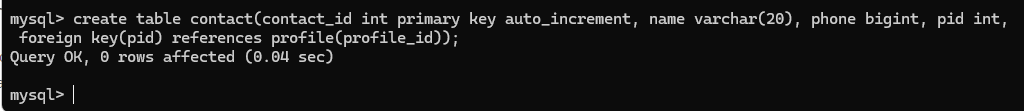
@OneToOne  
@OneToMany  
@ManyToOne  
@ManyToMany

These annotations must be used on a complex type so that spring data jpa automatically creates a join query & initializes the complex type.

ex:

@Entity  
class Profile {   
 // profileId, name, dob  
 @OneToMany  
 private List<Contact> contacts;  
}  
  
@Entity  
class Contact {   
 // contactId, name, phone, pid;  
}

Create contacts table

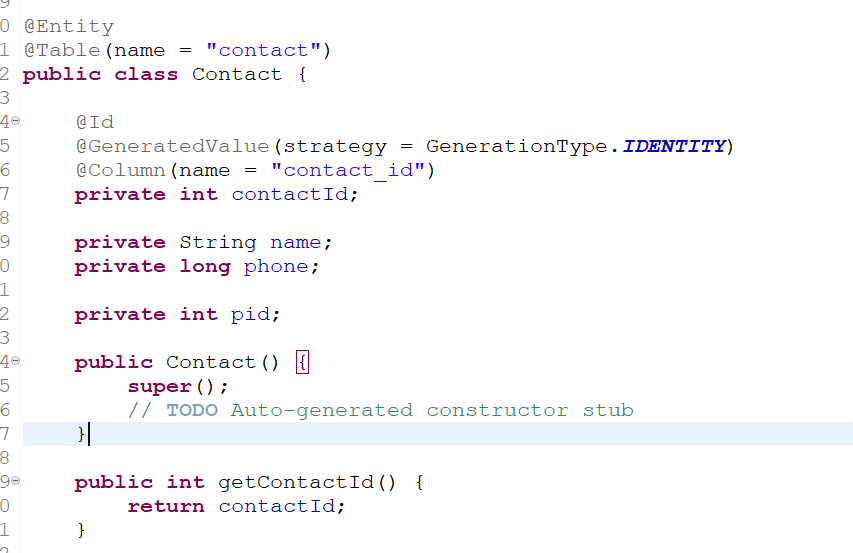


Now we must understand that each profile can have more than one contacts, and when profile wants to add a contact the id of the profile must also be stored, but for a new profile contact list will be empty

Things to create

1. Contact: An entity class for a contact table
2. Profile: A List<Contact> with @OneToOne annotation
3. ContactRepository: CRUD methods to manage contact entity
4. ProfileService: create addContact, deleteContact based on the profile id
5. ProfileServiceImpl: auto-wired ContactRepository so that addContact, deleteContact methods can save or delete the contact entity
6. ProfileController: Methods to add contact based on the profile and also delete contact based on the profile

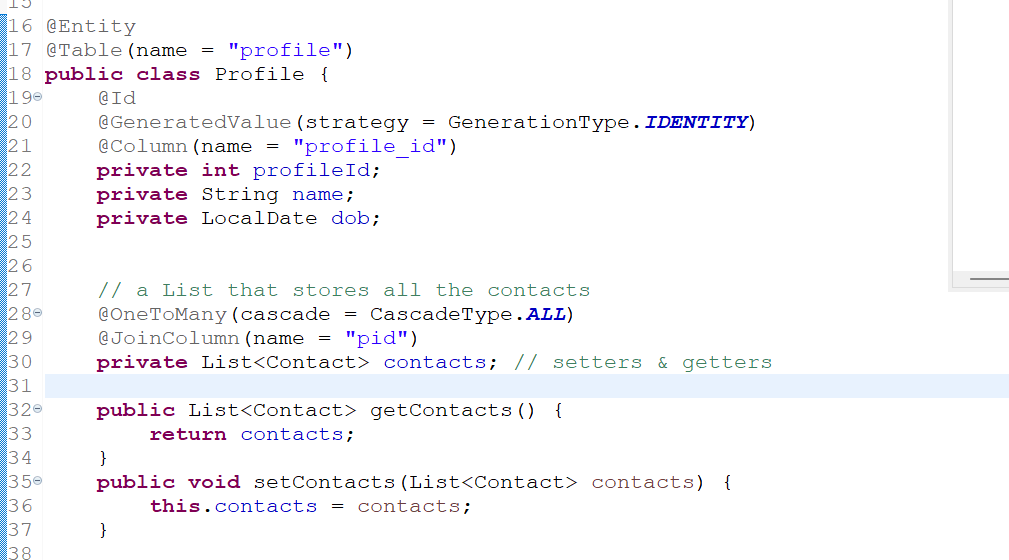
Contact.java



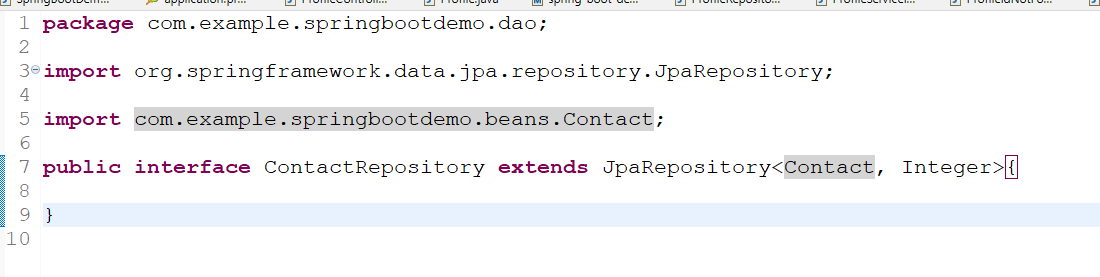
pid is a foreign key whose value is initialized when a particular profile wants to add contact

ex: /profile/100/addContact with json { “name”:”A”, “phone”:999} now webservice will initialize contact object with name = A, phone = 999 & pid = 100, passes to the ContactRepository

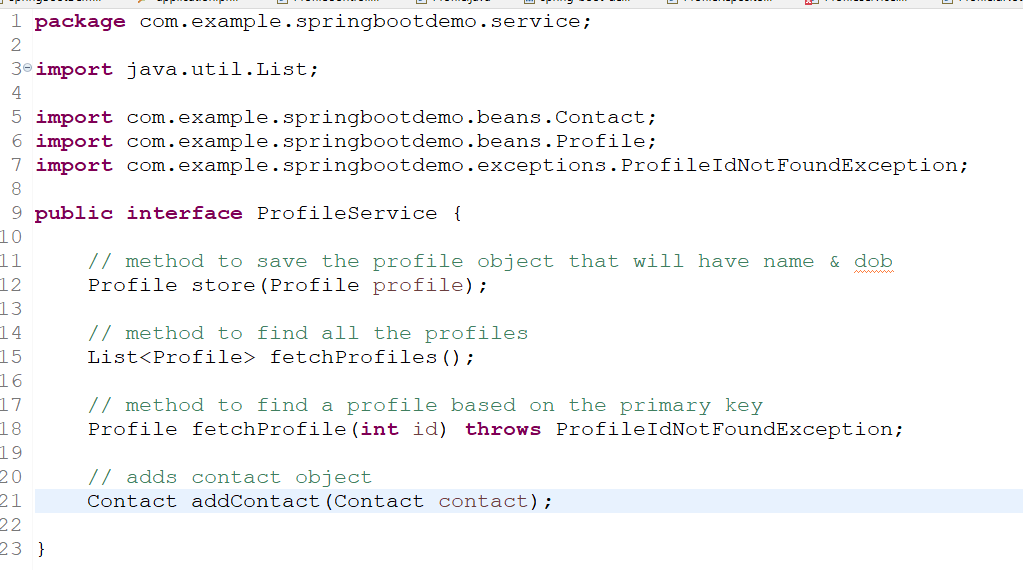
Profile.java Must have the List<Contact> with @OneToOne & @JoinColumn(name = “pid”) this is used to apply a condition during join table



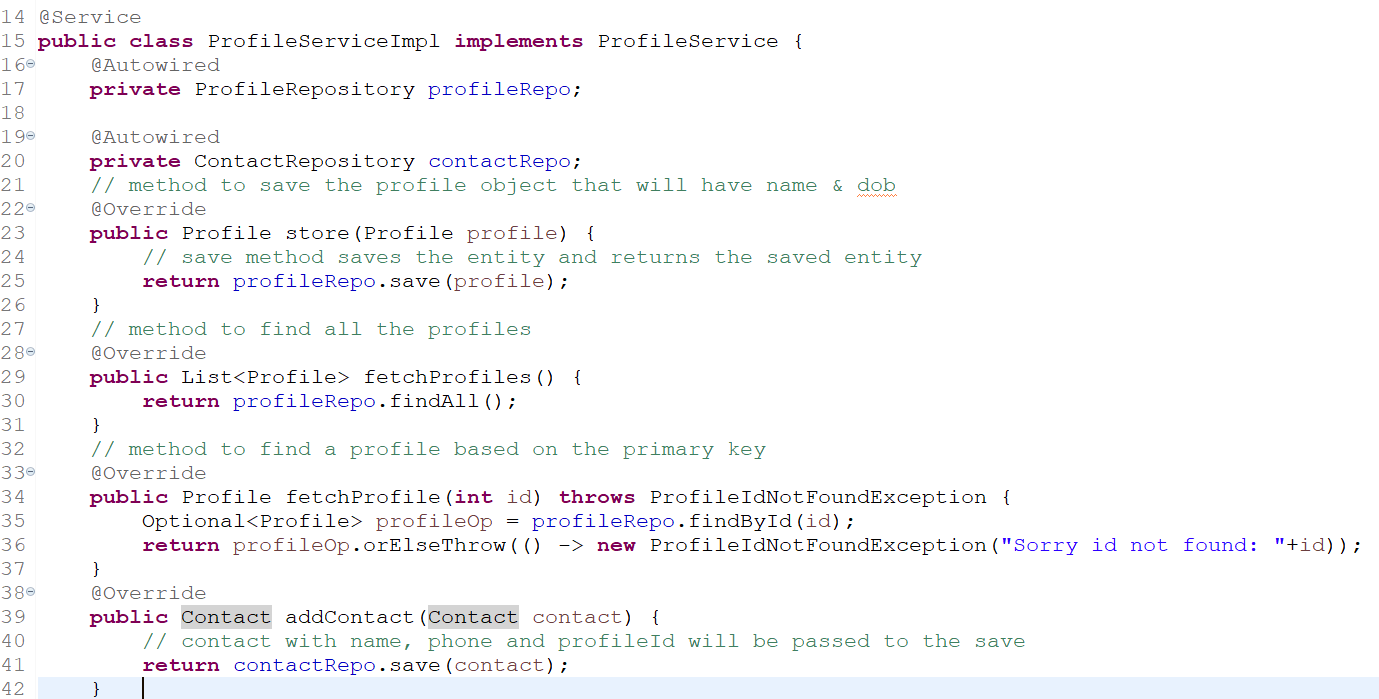
ContactRepository.java



ProfileService.java



ProfileServiceImpl.java



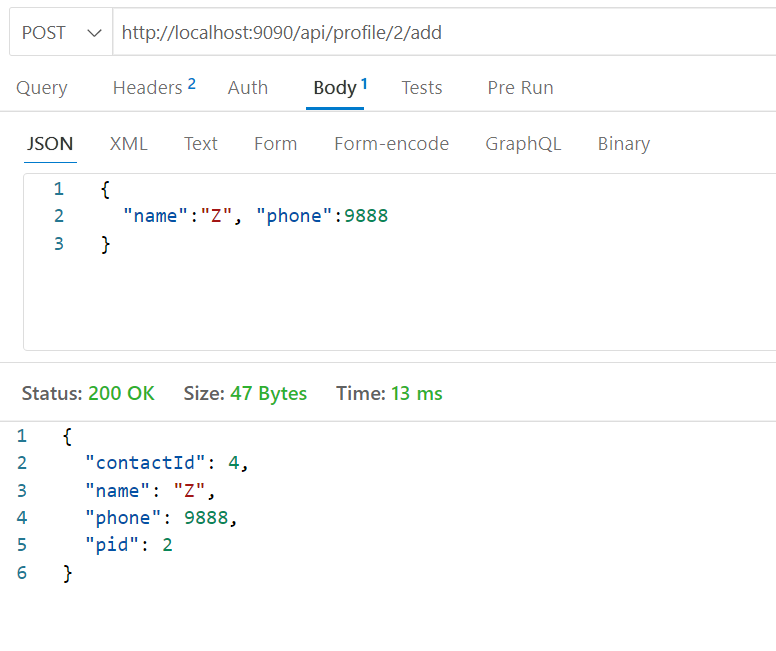
Note: We have auto-wired ContactRepository to call the save() method inside the addContact method

Now controller must call the addContact when a request comes to the addContact it uses profileId in the path & contact data in the form JSON

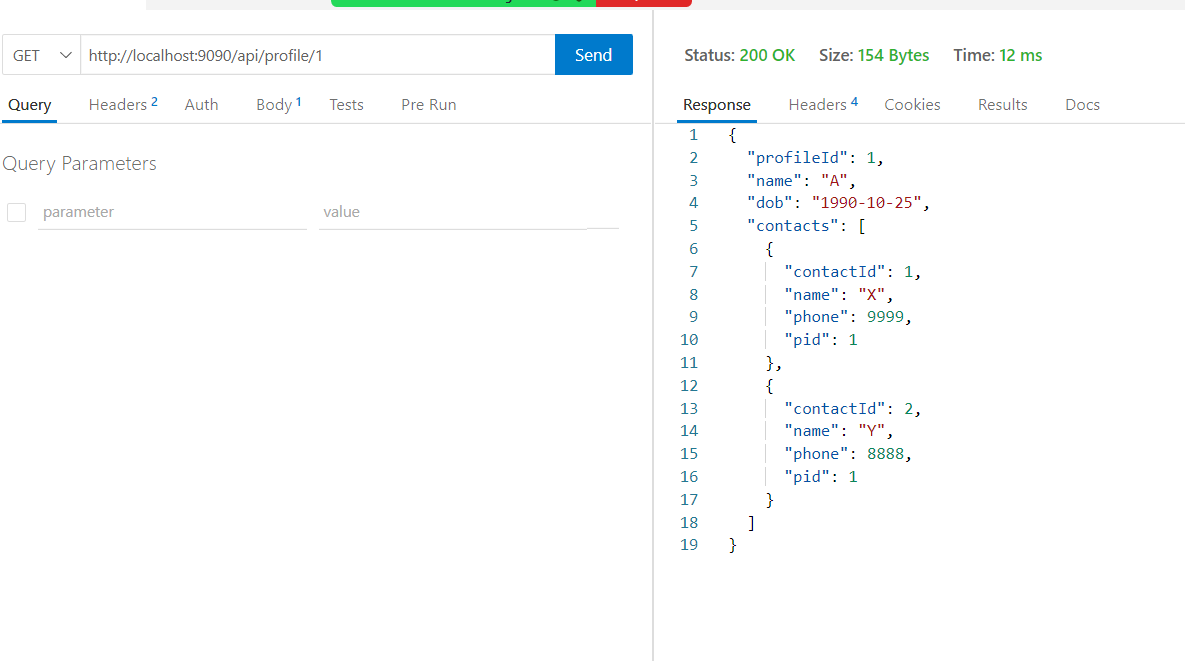
ProfileController.java



Now you can use /profile/1/add and send contact data



Get the profile where you will see the contacts belonging to the profile



Activity:

1. Perform delete contact based on the contact-id for a particular profile-id
2. Update contact phone number based on the contact-id for a particular profile-id

Summary:

@PostMapping, @GetMapping, @PutMapping, @DeleteMapping  
@PathVariable  
@RequestBody  
@RequestMapping  
@RestController  
@Autowired  
@Service  
@Repository  
@Configuration

Microservices

It is a small independent services that you can independently develop, build, test & deploy.

Advantages

1. Developers can independently create the services without having any technology barrier
2. Testers doesn’t need to test all the services when changes happen to any one of the services
3. You can scale whichever the service you want without scaling the entire application
4. Release of new feature wouldn’t take much time
5. Failure of any services doesn’t affect other services
6. One microservice can share data with another microservice just like a REST based communication

Design patterns to implement microservice

1. Service Discovery
2. Discovery Client / Microservices
3. Client side load balancer
4. Distributed configuration
5. Circuit Breaker

Service Discovery: It is a program that registers all the microservices with its physical address & instance – id, physical address can change however the instance – id will not be changed

Discover Client / Microservice: These are programs which are registered in the service discovery, they need to constantly ping the service discovery to give its health status (Generally every microservice sends heart beats to the service discovery every 30s)

Client side load balancer: This is a program which can resolve the physical address of the microservice and also can distribute the load if there are multiple microservice instance.

Distributed Configuration: This is a program which can have a centralized configurations over the cloud like AWS/GIT that can be shared across the multiple microservices, the configurations can also be kept in encrypted format (some sensitive information’s) so that only the microservices must able to get the right data after decrypting.

Circuit Breaker: This is a program which will open the circuit when a remote service is down so that it avoids cascading the failures to their client service

Note: All these design patterns are common things every microservice must have

Spring Cloud: This is a project by a spring which provides all the necessary tools & design patterns to develop the microservice with a simple annotation

Below are the libraries of spring cloud for each design patterns

1. Service Discovery: Eureka Server library
2. Discovery Client: Eureka Client library
3. Client Side Load Balancer: Ribbon Client which part of eureka client
4. Distributed Configuration: Configuration Server library
5. Circuit Breaker: Resilience4j library

Spring Microservices uses 2 projects

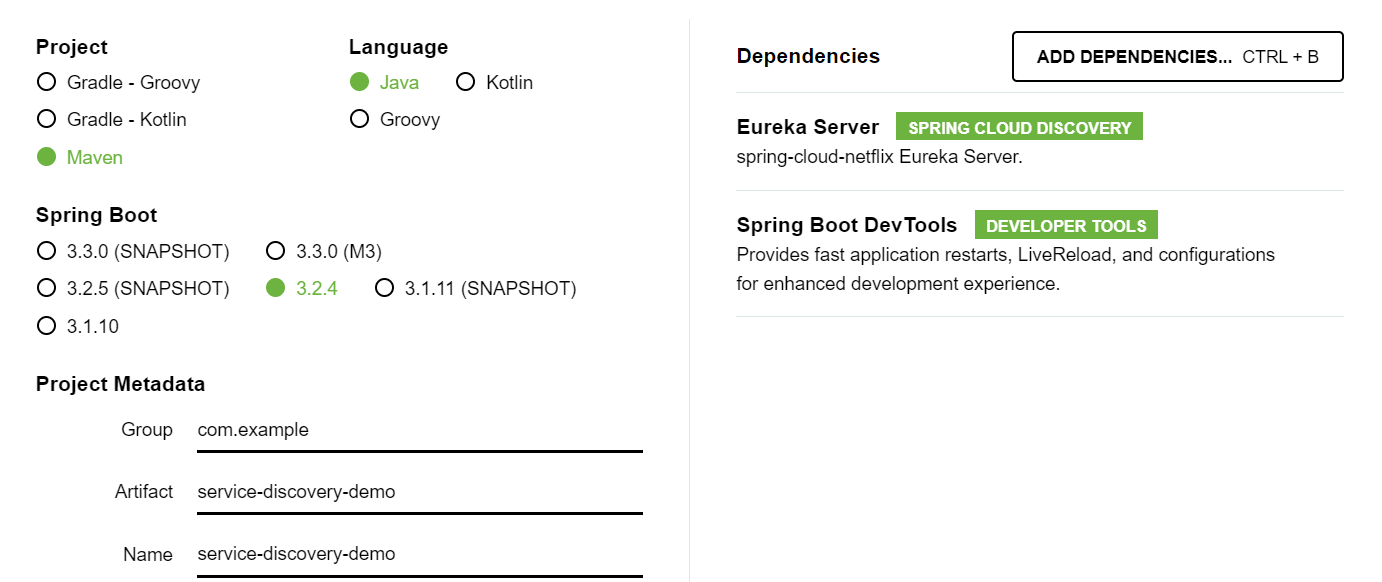
1. Spring Cloud: It provides all the common tools & design patterns to develop microservice
2. Spring Boot: It autoconfigures the project based on the library you add

Note: Spring Initializr provides the compatible versions of spring cloud & spring boot which we must not change

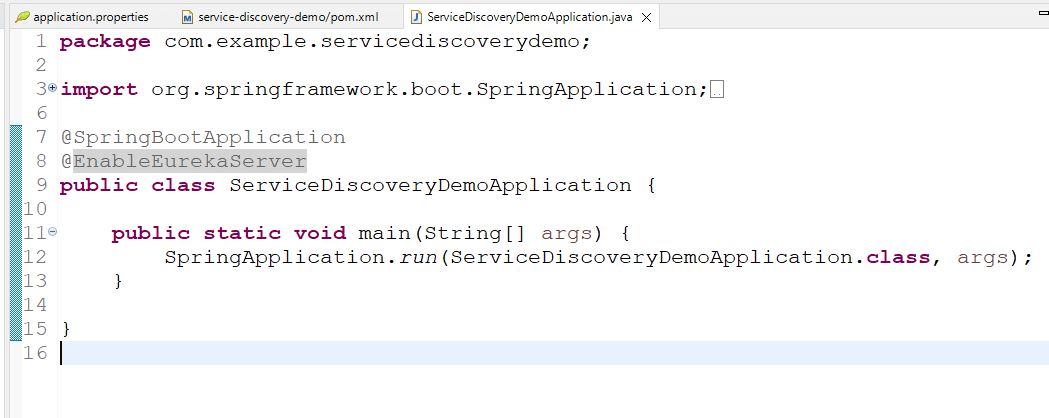
We need to create 2 projects

1. Service Discovery
2. Microservice

Service Discovery Dependencies



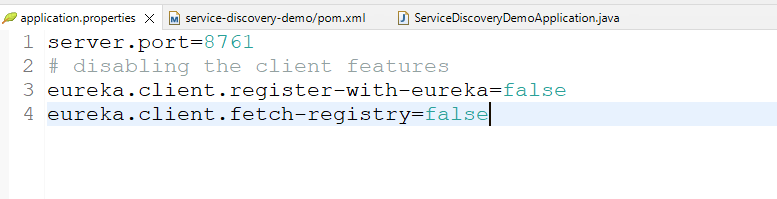
Eureka Server provides an annotation @EnableEurekaServer to add the service discovery to your program



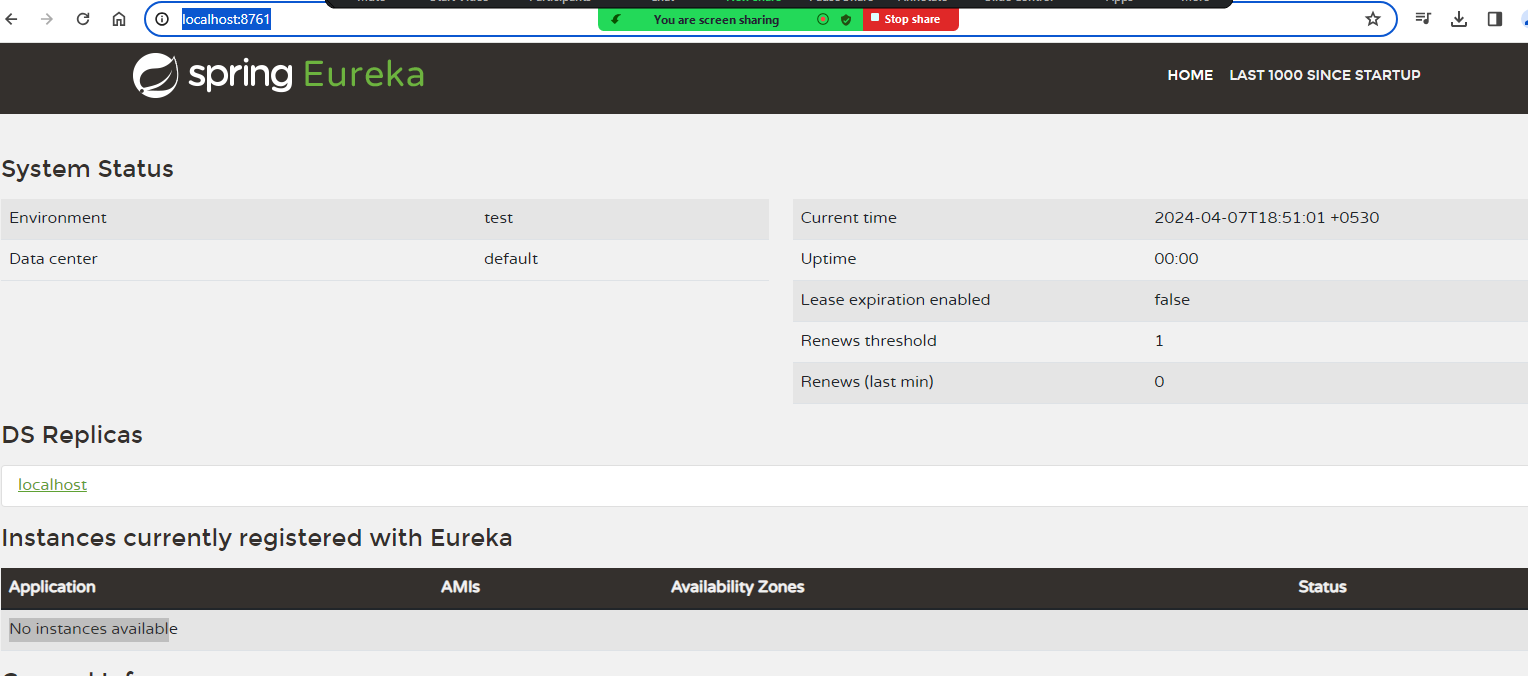
Note: When you add Eureka Server library it also downloads Eureka Client library, which makes the service discovery act like microservice, which must be disabled because spring boot auto-configures server as client also, we don’t want this to act like client

Note: Eureka clients always registers automatically to Eureka server by searching them in 8761 by default

application.properties

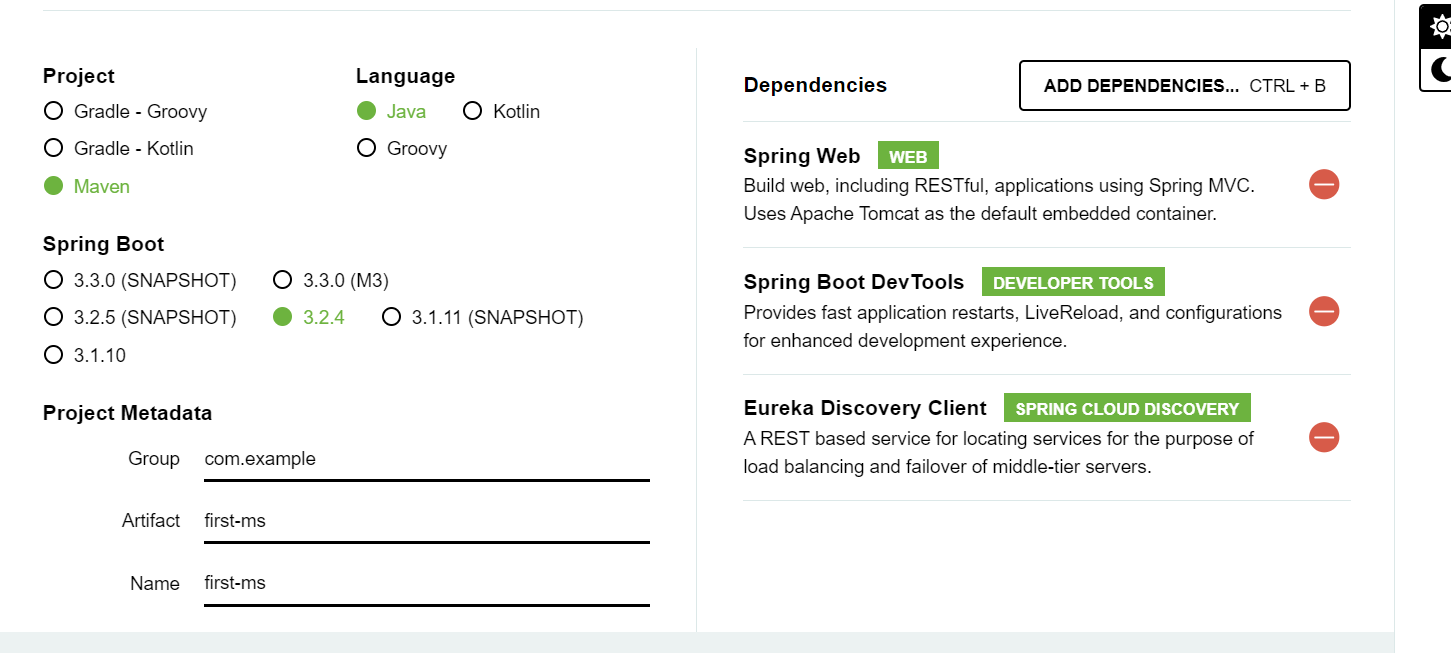


Eureka Server provides a dashboard which is like a website so that admins can see the list of clients/microservices registered through the URL



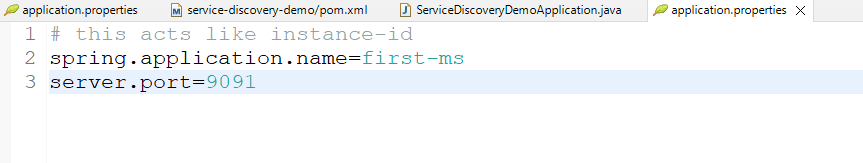
Microservice Dependencies

1. Web
2. Dev tools
3. Eureka Client

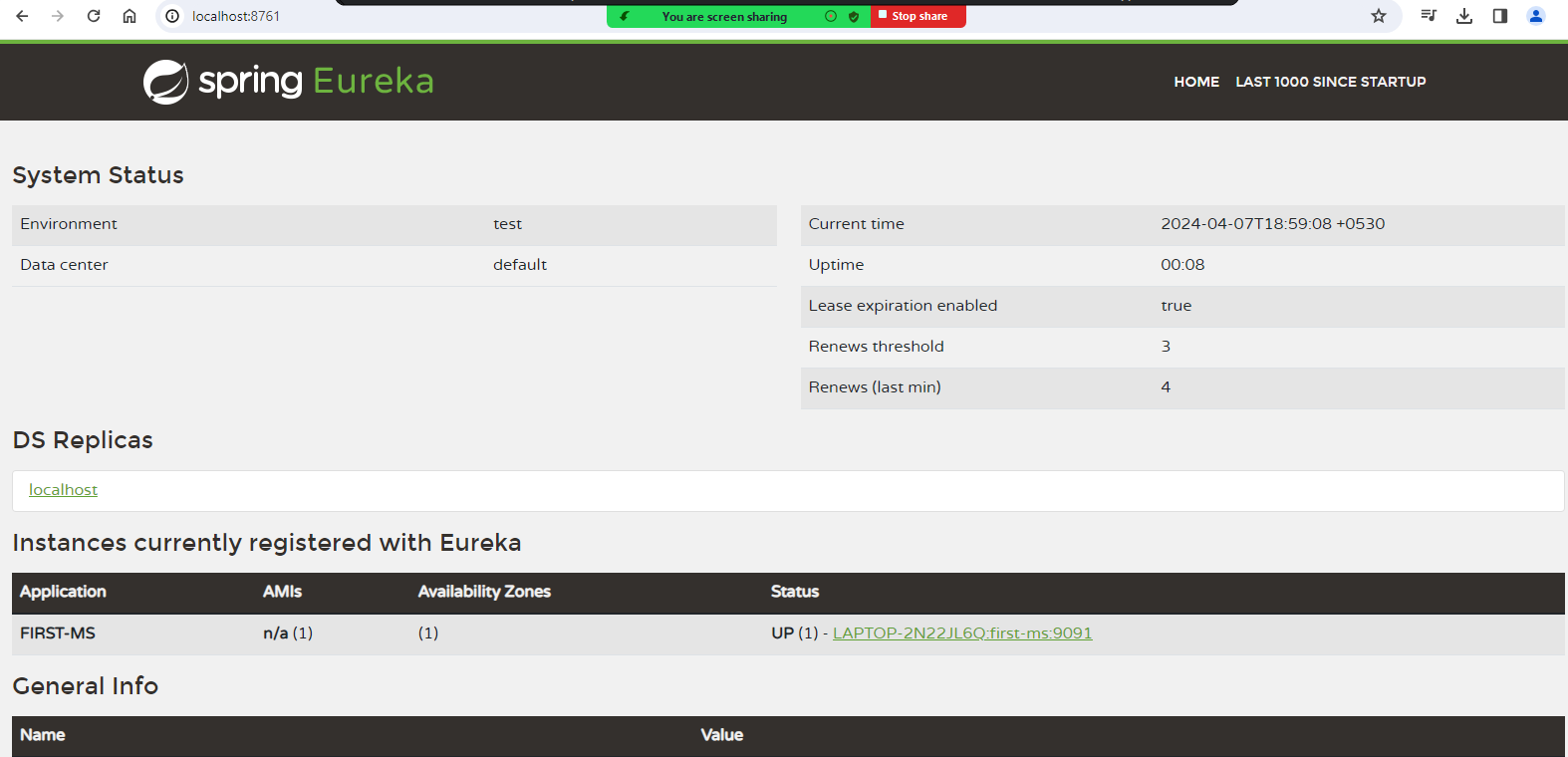


Note: You don’t have to use any annotations to register this as a microservice, because spring boot will automatically register this as a microservice using the eureka client library

application.properties



You just need to launch the application that will register the service in the service discovery.



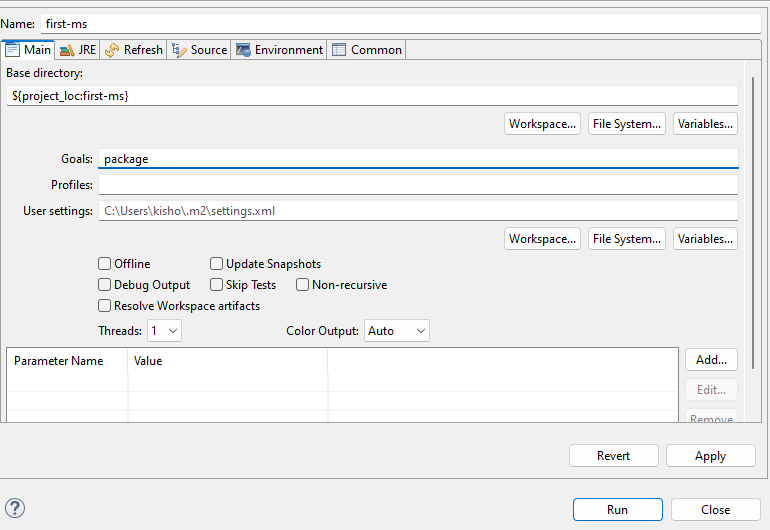
You can see the registered service, you can see UP(1) which means only one instance of FIRST-MS is running.

How to create another instance of the same microservice

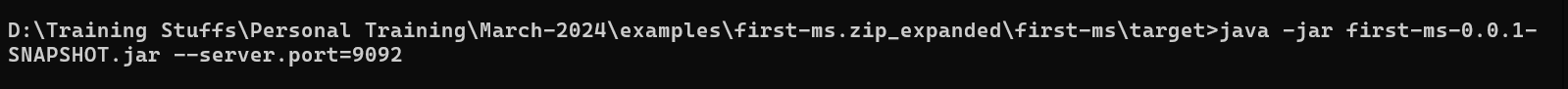
We must create a jar and run using command prompt, there we must change the port number

Right Client -> Run As -> Maven Build… -> Goals : package

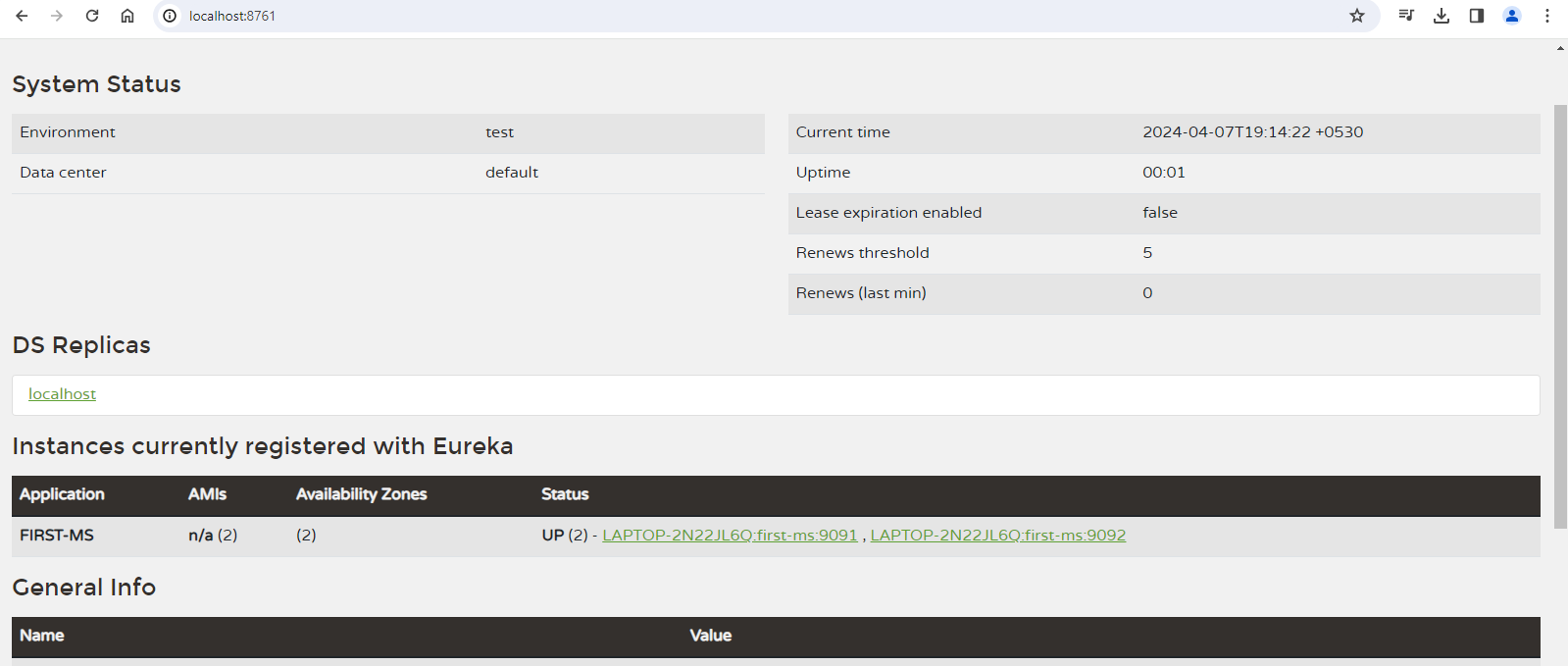




The jar file will be created in target folder, you must run the jar from the terminal by using –server.port=9092 that overrides the server.port in application.properties



Now you can see 2 instances of the same service

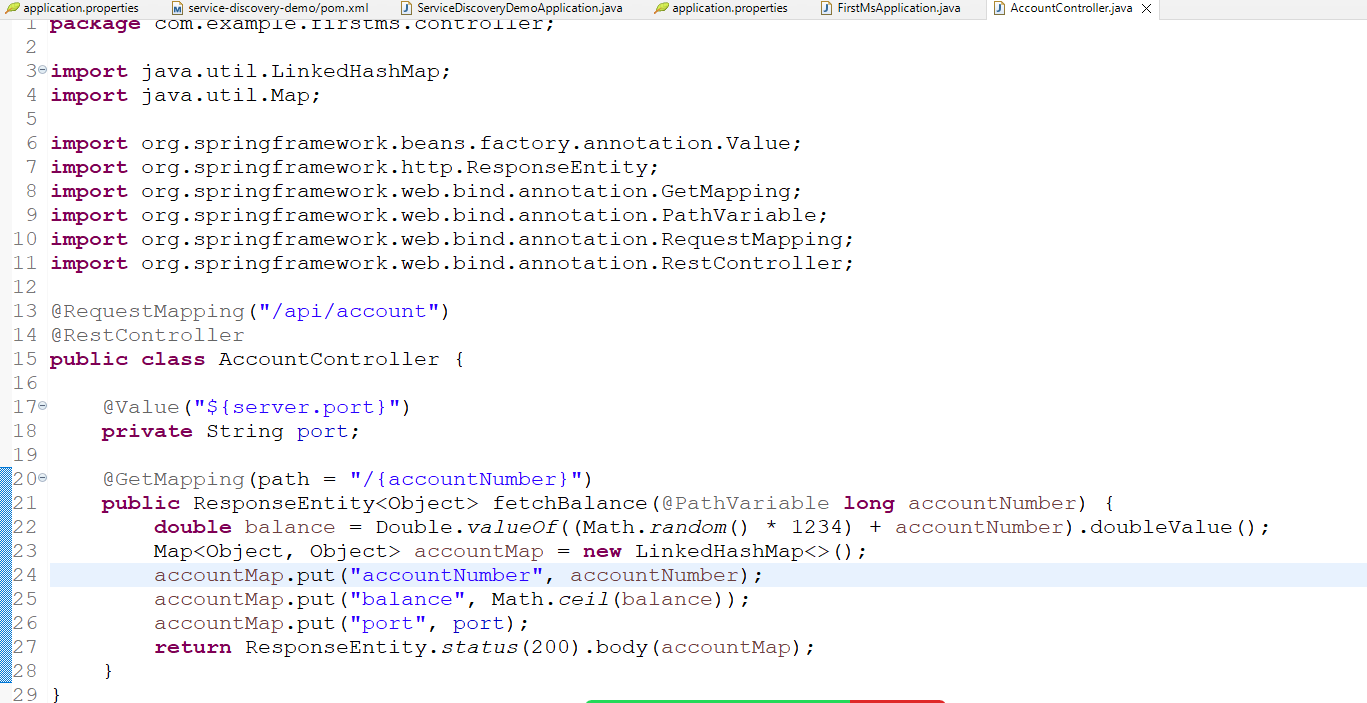


Summary:

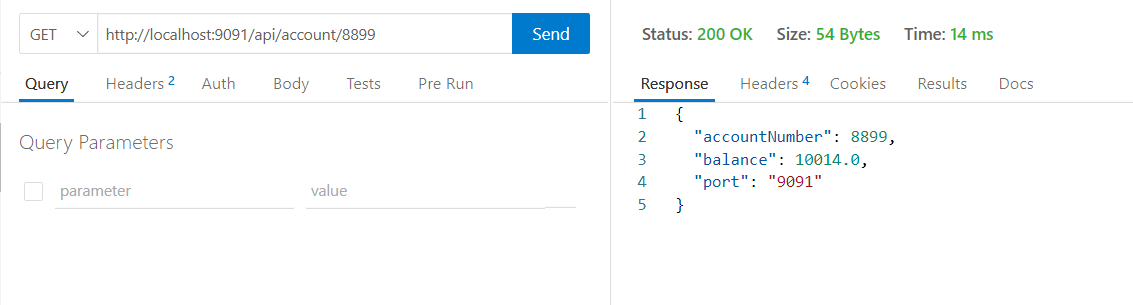
1. Eureka Server Project
2. Eureka Client Project
3. java -jar filename.jar --server.port=9091

Creating a webservice that can return the account-balance in the first-ms

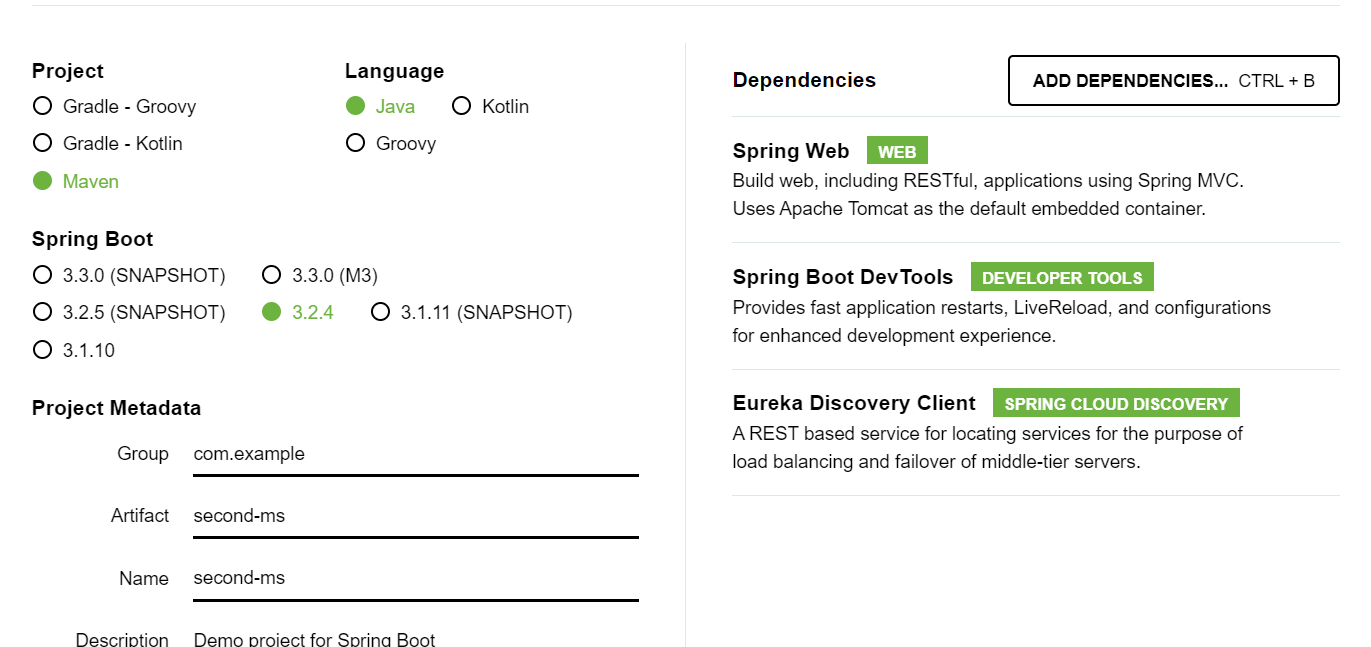
AccountController.java



You can test in the thunder client for different account number

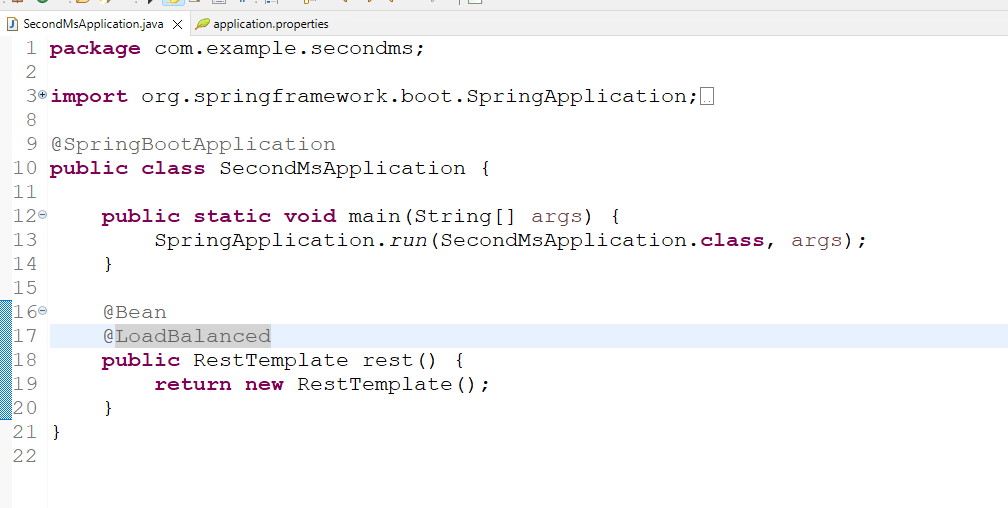


Creating a second microservice which is a wallet that can show the wallet name and the amount it can spend by getting the data from the account microservice (first microservice) this is called as communication between the microservices.



RestTemplate: It is an instance that can access a remote service using the URL, it can make HTTP calls using get, post, put & delete methods, it is mainly used to access webservices from a spring program

Create RestTemplate in the main class and annotate with @Bean & @LoadBalanced



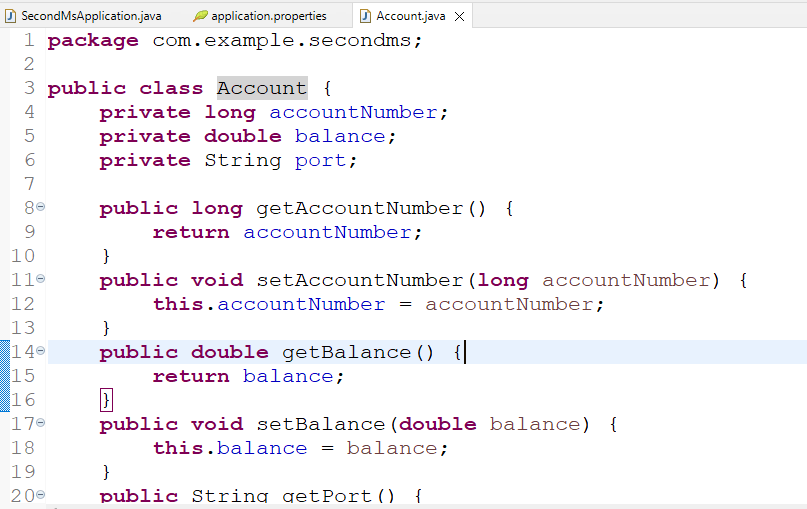
@Bean: This will register the object in the spring container so that you can use @Autowired in any place, because RestTemplate is not by default present in the spring container we must register that in the container

@LoadBalanced: It is the client side load balancer, takes care of resolving the physical address of the remote service when RestTemplate uses the instance-id

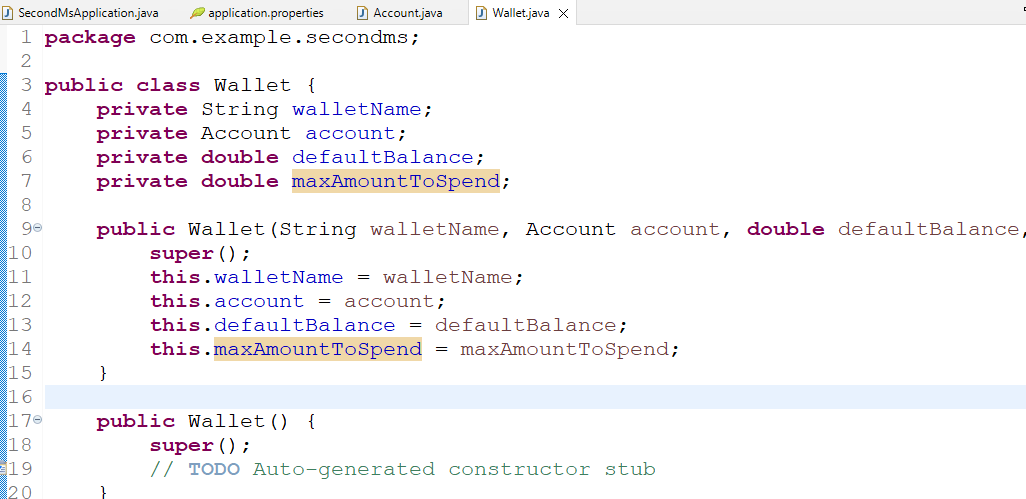
ex: <http://first-ms/api/account/8899> this URL physical address will be resolved by the load balancer

We need to create two java beans to represent the account & wallet details.

Account.java

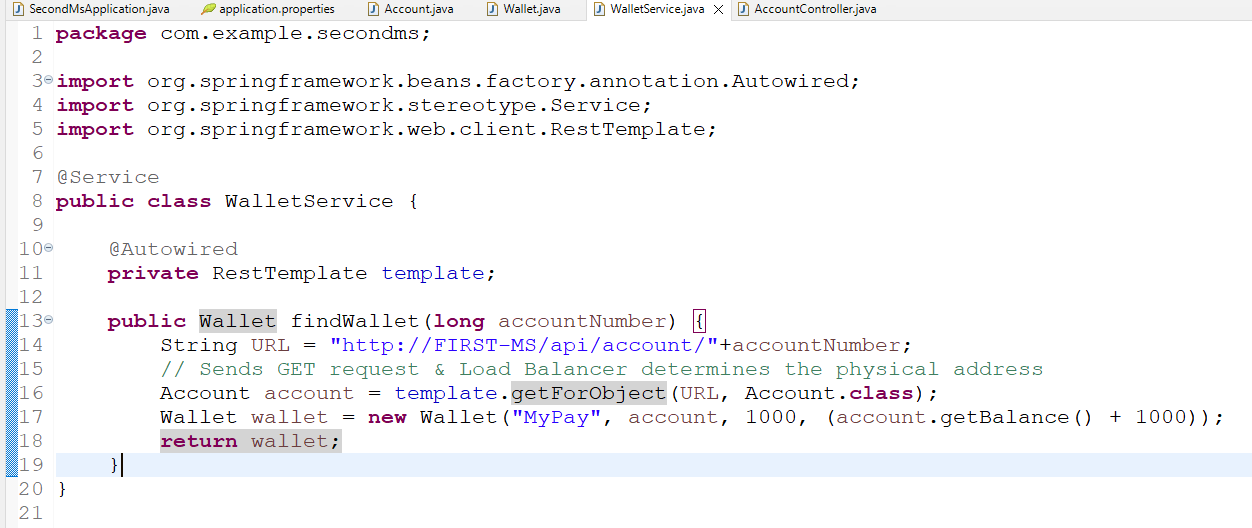


Wallet.java

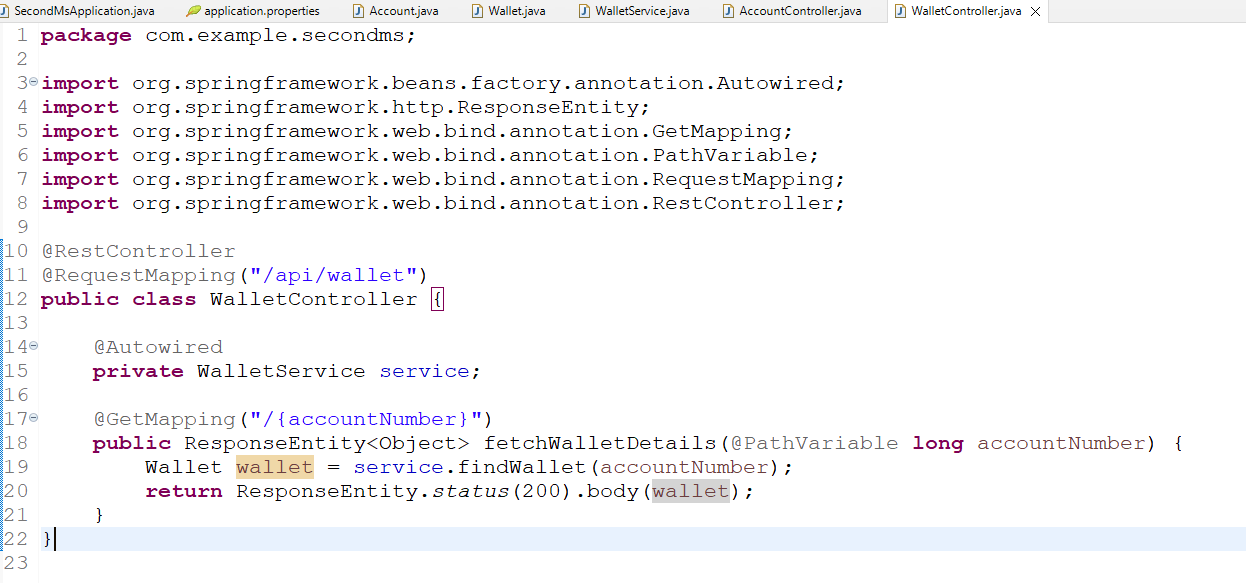


Create a service layer that will access the remote microservice using the RestTemplate

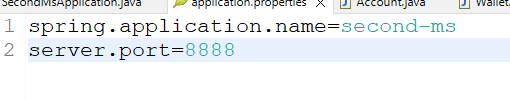
WalletService.java



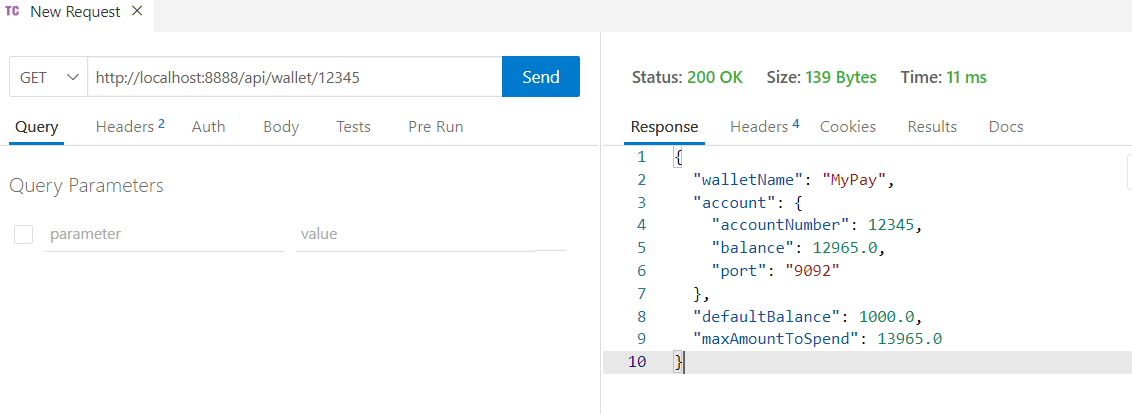
WalletController.java



application.properties



Output:



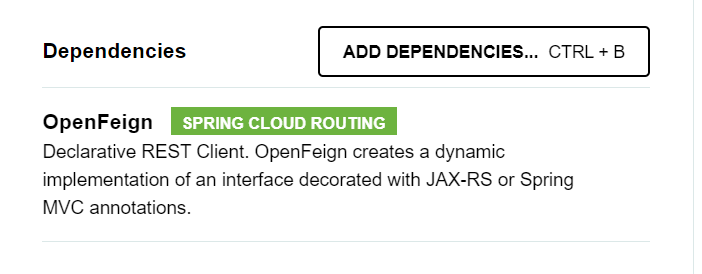
Feign Client: It is a library used to access the remote service just like the RestTemplate but it has some benefits over the RestTemplate like

* It has inbuilt load balancer so that you don’t need to use @LoadBalanced
* It allows you to create a reusable interface which can be used to provide methods that can make remote calls using HTTP methods
* It also implements the interface & registers its object in the spring container so that you can directly autowire and access the methods of the interface

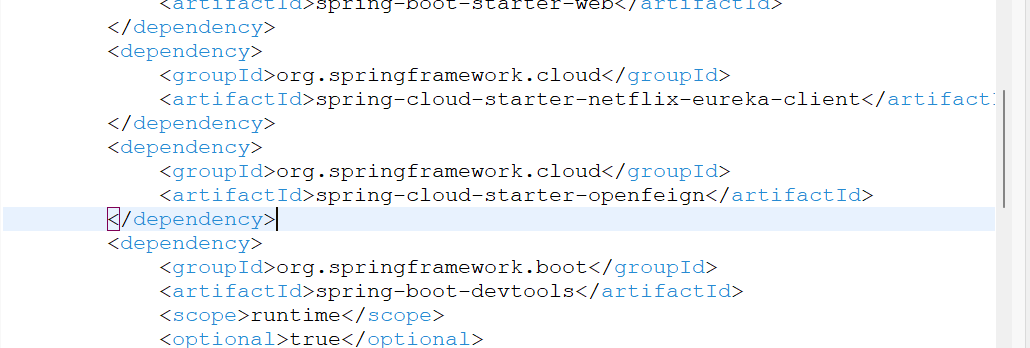
@FeignClient(“http://FIRST-MS/api”)  
interface AccountClient {   
 @GetMapping(“/first/account/{accountNumber}”)  
 public Account fetchAccountDetails(@PathVariable long accountNumber);  
}

@Autowired  
AccountClient client;  
  
Account account = client.fetchAccountDetails(1234);   
// GET request to http://FIRST-MS/api/first/account/1234

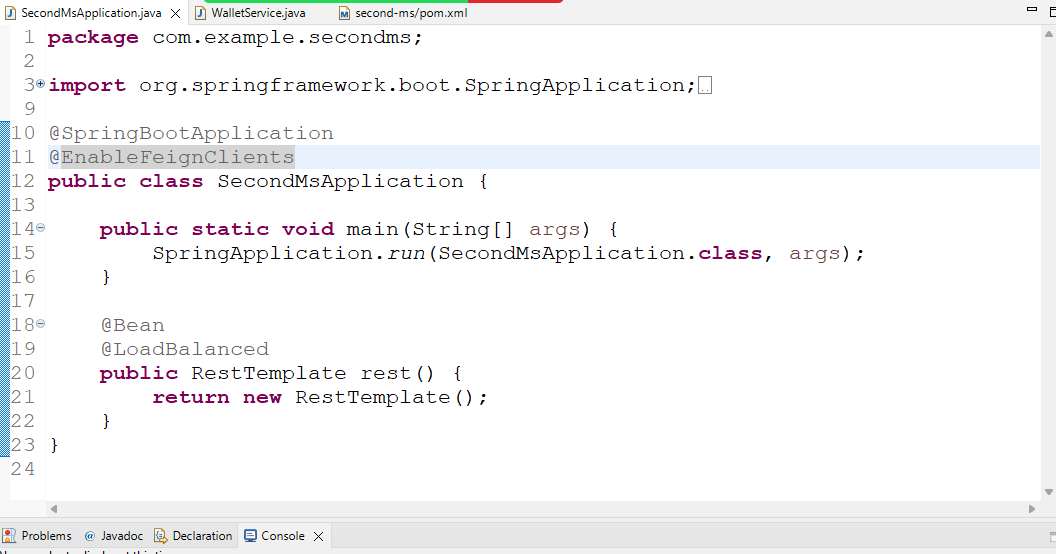
Library for feign client



Modify pom.xml of the second project

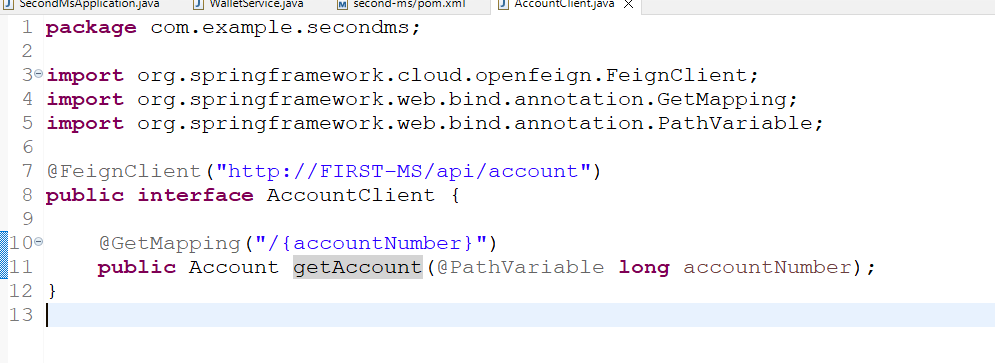


@EnableFeignClients: This annotation must be used on top of the main class, this scans the @FeignClient interface to auto-implement the interface

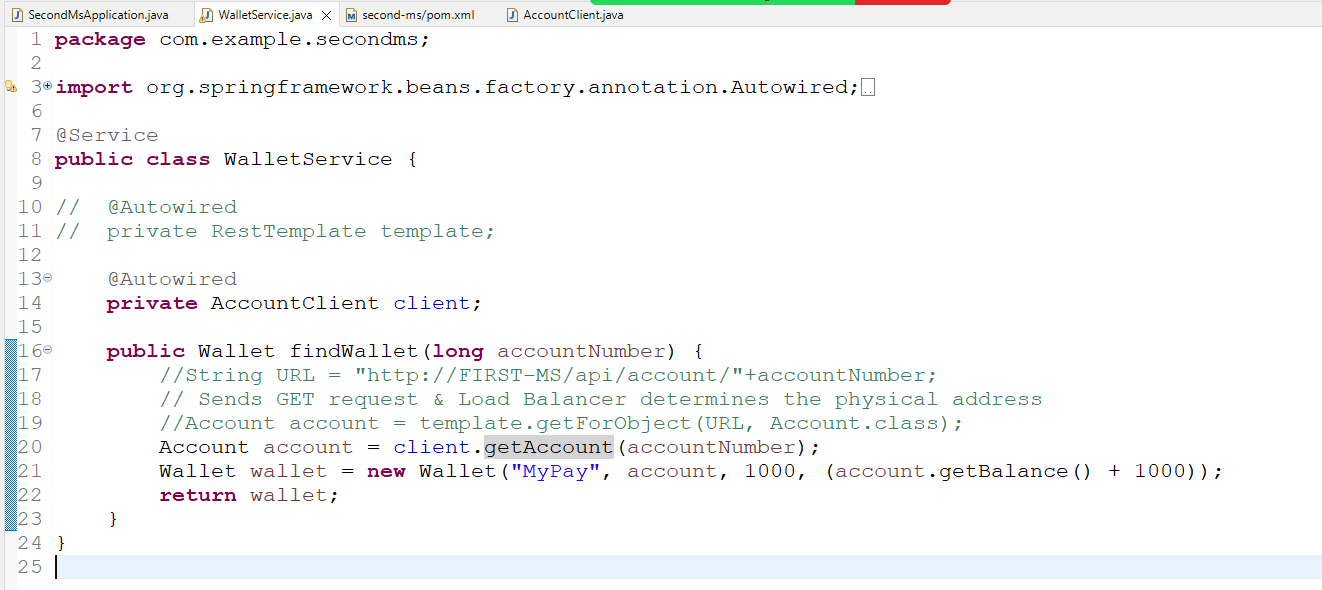


Create an interface to make GET calls to the first microservice

AccountClient.java



Note: Here AccountClient will be auto-implemented and it is reusable, you can call getAccount at multiple places without using the URL, behind the scene a GET request will be sent and LoadBalancer resolves the physical address



Now when the getAccount is called a GET request to the first microservice is sent and behind the scene load balancer determines the physical address.

Output:



Circuit Breaker: It is a program that will block the request reaching to the remote service when it is down, this avoids the cascade of failures to the client microservice, Circuit breaker will have 3 states

1. Close
2. Half Open
3. Open

Close: It will send request to the remote service

Half Open: It will also send request to the remote service, but it is mainly to check the remote service status whether its up / down, so that the state can be changed to close or open

Open: It will not send request to the remote service, instead it returns the alternate response.

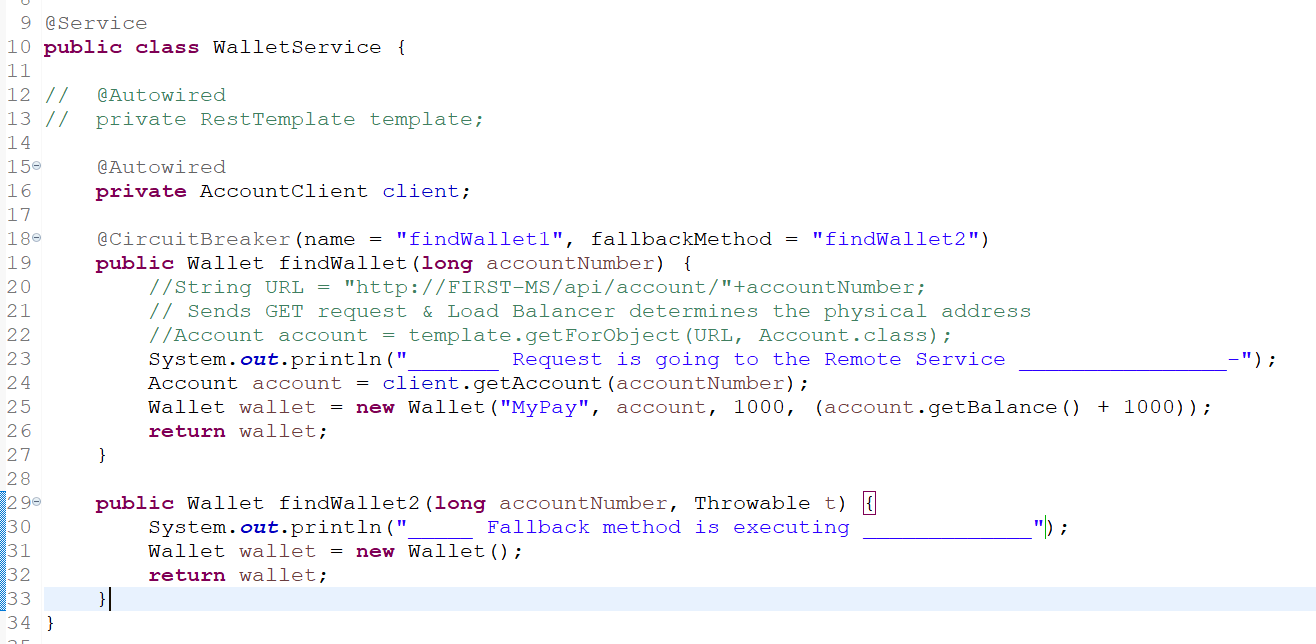
Client Microservice configuration

1. Resilience4j: For circuit breaker configurations
2. AOP: To automatically access the fallback methods
3. Actuator: To see the application health/status like circuit breaker status



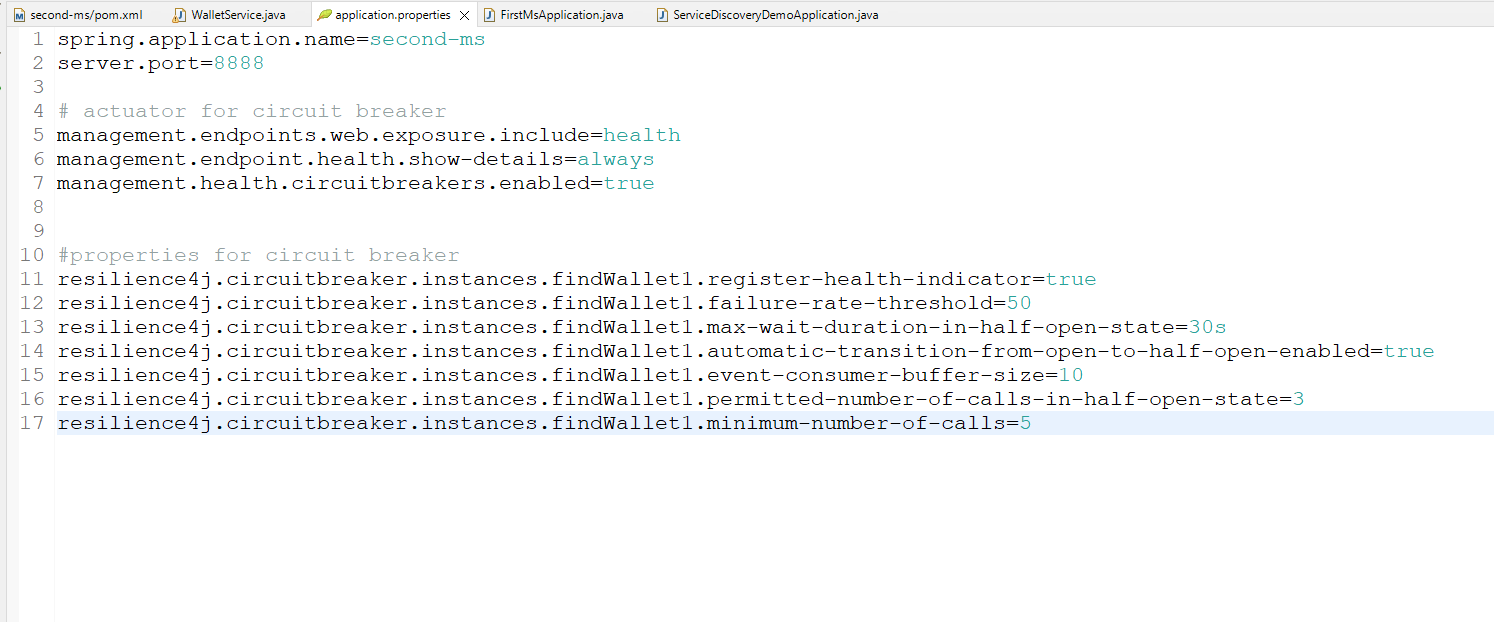
Note: spring boot starter aop is available from maven repository .

WalletService.java

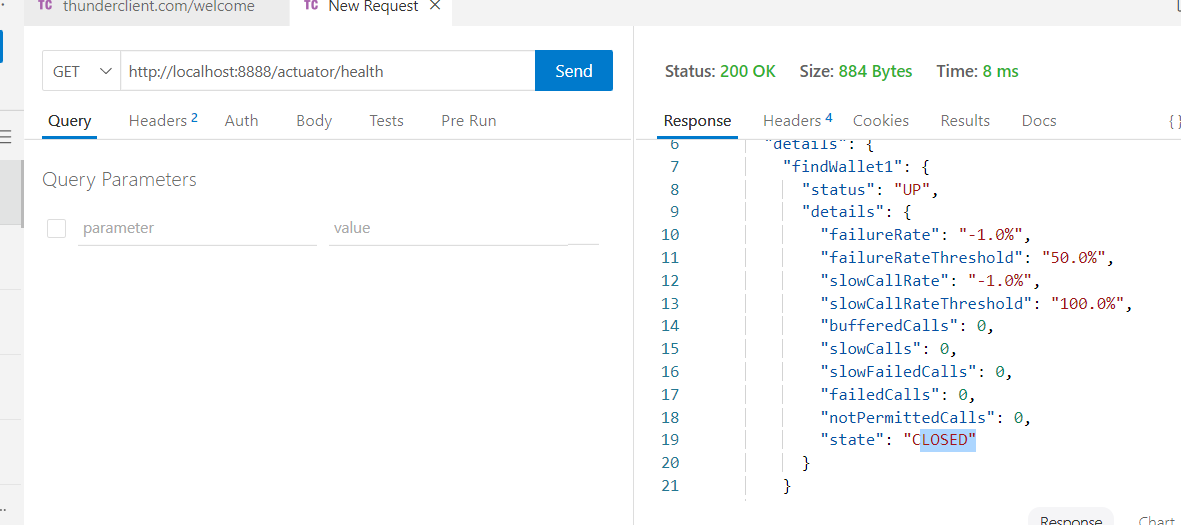


There are two methods one is sending request to the remote service & other is executed when as a fallback method for the alternate response when the circuit is open.

application.properties



You can see the circuit breaker closed, half\_open & open state in the actuator/health

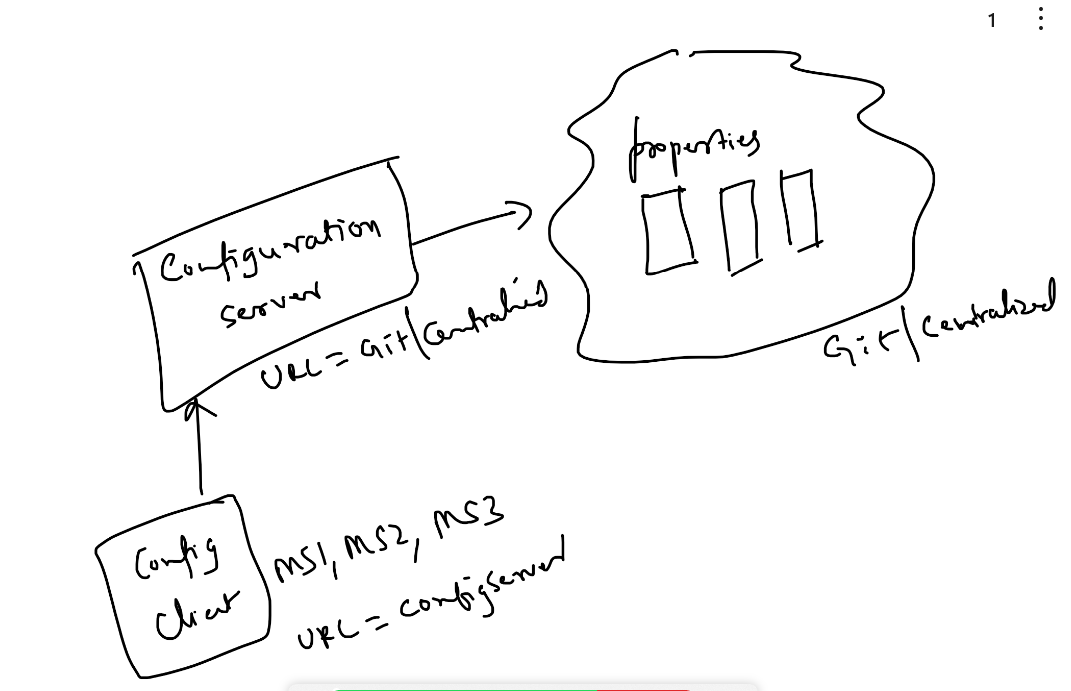


Distributed Cloud Configuration

It allows multiple microservices to use the common configurations stored in a centralized location like GIT/AWS/SVN.

Configuration Server: it is a program that connects the centralized repository, it will have the URL of the centralized repository, it will give the configurations required for the client

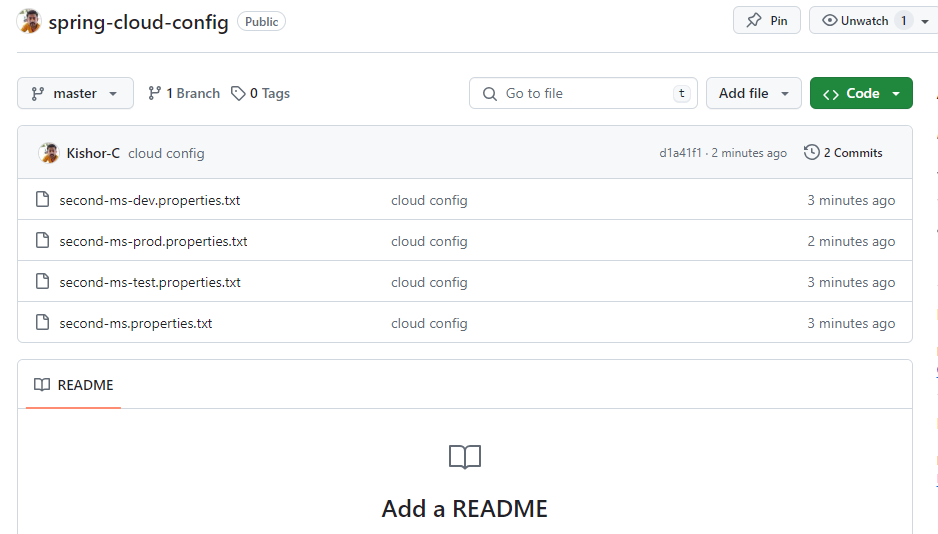
Configuration Client: It is a program that connects to the configuration server to fetch the configurations, it will have the URL of the configuration server



Things to create

1. Configuration Server program: This uses @EnableConfigServer that connects the centralized repository & pulls the configuration to the client
2. Configuration Client program: This is a microservice that connects to the configuration server & specifies which file it needs

Push some configurations to the git, whose filenames must be spring.application.name



second-ms-dev.properties

database.username=devroot

database.password=12345

second-ms-test.properties

database.username=testroot

database.password=12345

second-ms-prod.properties

database.username=prodroot

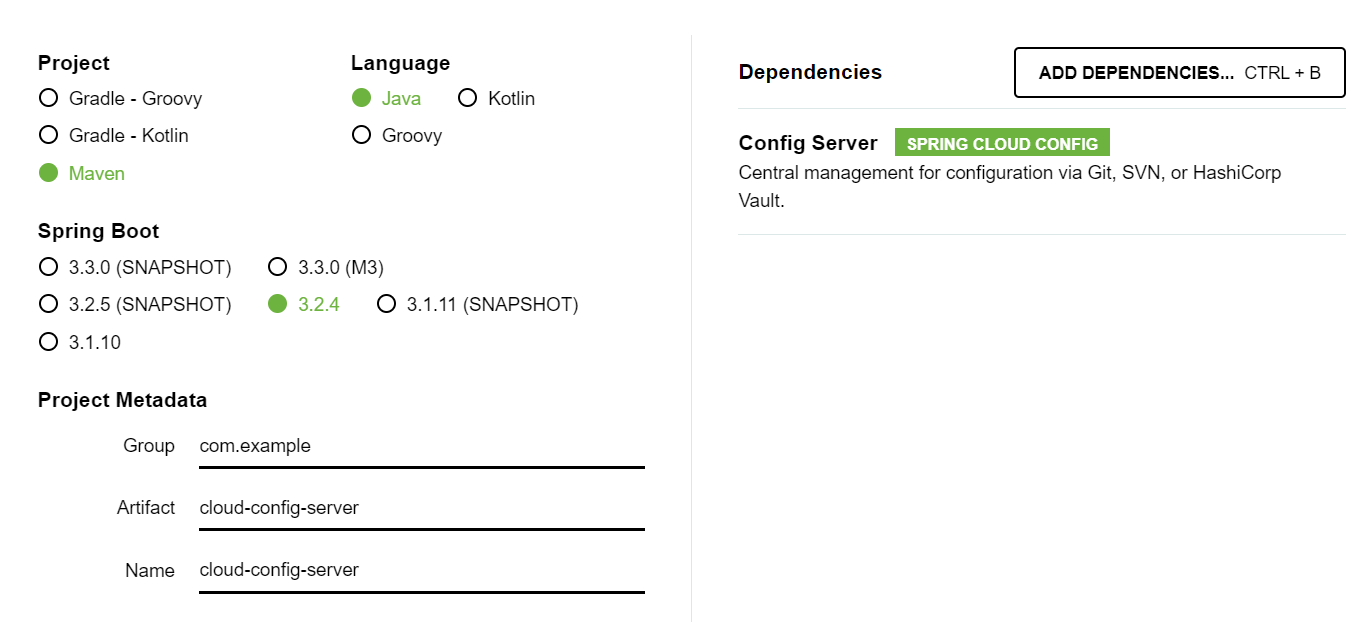
database.password=12345

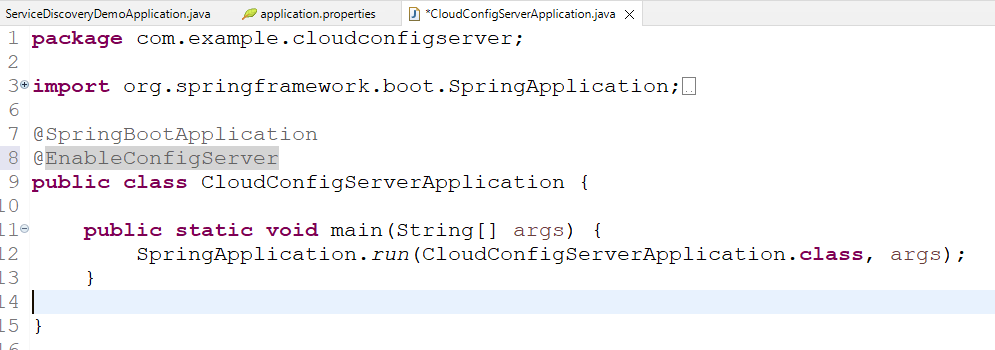
second-ms.properties

database.username=root

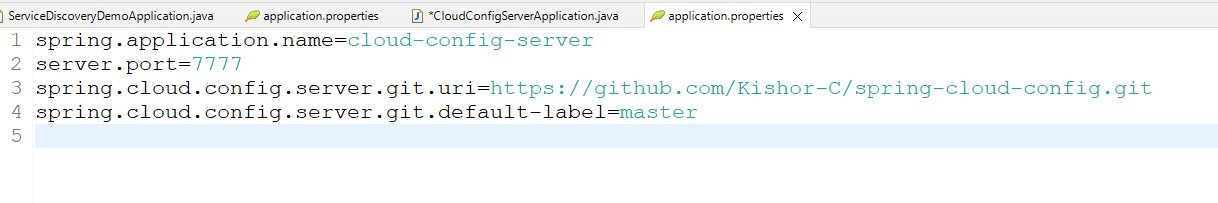
database.password=12345

We need to create a configuration server program that connects to the GIT





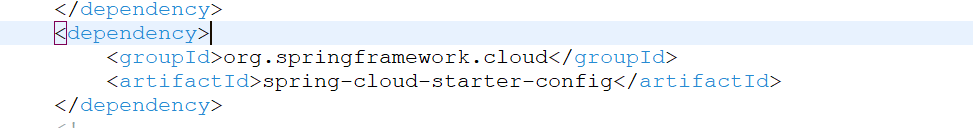
application.properties



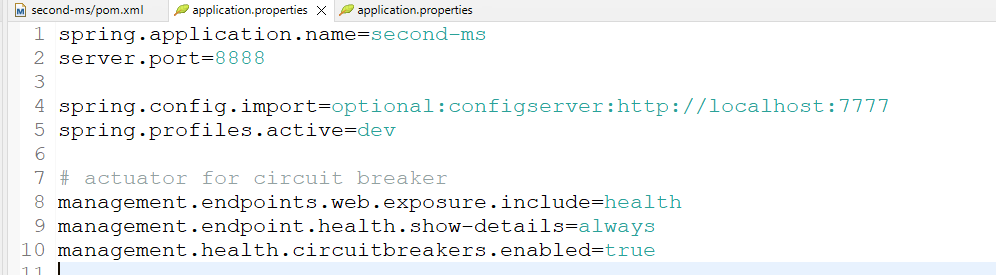
Now config client needs to use the configuration server url

Note: You must add config client library to the existing microservice

pom.xml of second-ms



second-ms/application.properties



WalletController.java



We have created username & password variables which gets the value from the property file that is fetched from the git



Output:

