**Basics Java & Spring Boot Microservices**

**Pre-requisites:**

* Participants must have good practical knowledge on Core Java especially like
  + Fundamentals - Control Statements, Loops, Arrays, Operators
  + OOPS concepts - Inheritance, Encapsulation, Abstraction, Polymorphism
  + Exception Handling
  + Collection Framework and Map
  + File Handling
  + Lambda Expression
* Participants can go through the below Udemy course

https://www.udemy.com/course/java-the-complete-java-developer-course/

**Training Methodology**

* Training will be on TDD & DDD approach
* The training will be based on a real-world use case and will be built from ground up in an iterative manner.
* All the sessions will be workshop based, where participants will be coding along with the instructor for the entire session.
* Concepts learnt will be added to the case study project in an iterative manner so that it must be an application at the end of the training program.

**Deliverables**:

* Codebase developed during workshop will be shared with the participants
* Reference materials will be shared with the participants

**Hardware & Software:**

* JDK 8 or later
* Eclipse IDE for enterprise / STS
* Database - PostgreSQL
* VS Code
* Postman
* Open internet connection for Maven and Node Module downloads

**Course Contents**

**Self Paced Learning (Mandatory)**

**Udemy Course link**

**https://www.udemy.com/course/java-the-complete-java-developer-course/**

**Topics to be learnt**

* Java Fundamentals
  + Operators
  + Conditional statements - if, if-else, switch
  + Loops - for, while, do-while
  + Arrays
  + break & return
* Encapsulation
  + private, public keywords
  + setters & getters method
* Inheritance
  + extends
  + super
* Polymorphism
  + overloading
  + overriding
* Abstraction
  + abstract class
  + interface
* Inbuilt classes
  + Object
  + String
  + LocalDate & LocalTime
* Functional Interface
  + Lambda expression
  + Method reference
* Exception Handling
  + try, catch, finally, throw & throws
  + Custom Exceptions
* Collection Framework & Map
  + List - ArrayList, LinkedList
  + Set - HashSet, TreeSet, LinkedHashSet
  + Map - HashMap, TreeMap, LinkedHashMap
* File Handling
  + Input & Output Streams
  + Reader & Writer

**Instructor Led Training - Basics**

**Day 1**

* **Pre-requisites** 
  + Collection Framework
  + Streams & Lambda expressions
* **PostgreSQL Database**
  + Queries
  + Datatypes
  + Functions & Operators
  + Type conversion
  + Parent and Child tables relationship
  + Indexes
  + Full Text Search
  + Performance Tips
  + Transaction Isolation

**Day 2**

**Spring & Spring Boot**

* **What is Spring**
  + Features of Spring
  + Dependency injection
  + Bean Configurations
  + Application Context
  + ClassPathXmlApplicationContext
  + Setter injection
  + Constructor injection
* **Spring Boot**
  + Advantages of Spring Boot
  + @SpringBootApplication
  + Auto-Configurations
  + Spring Boot starters
  + Component scanning
  + Spring boot configuration file
  + YAML
  + Properties

**Day 3**

* **Configuring Spring Boot Application**
  + Adding Jetty server
  + Adding Undertow server
  + Creating an executable Jar
  + Creating a WAR
  + Deploying to the external server
  + Adding Dev-tools
  + Adding Actuator
* **Spring Boot Web**
  + Front Controller
  + View Resolver
  + Controller
* **Multilayer architecture**
  + View Layer
  + Controller Layer
  + Service Layer
  + DAO Layer
* **Spring Boot with REST**
  + @RestController
  + @RequestMapping
  + JSON & XML data
  + Path parameters
  + Request body
  + HTTP status codes
  + HTTP methods mapping

**Day 4**

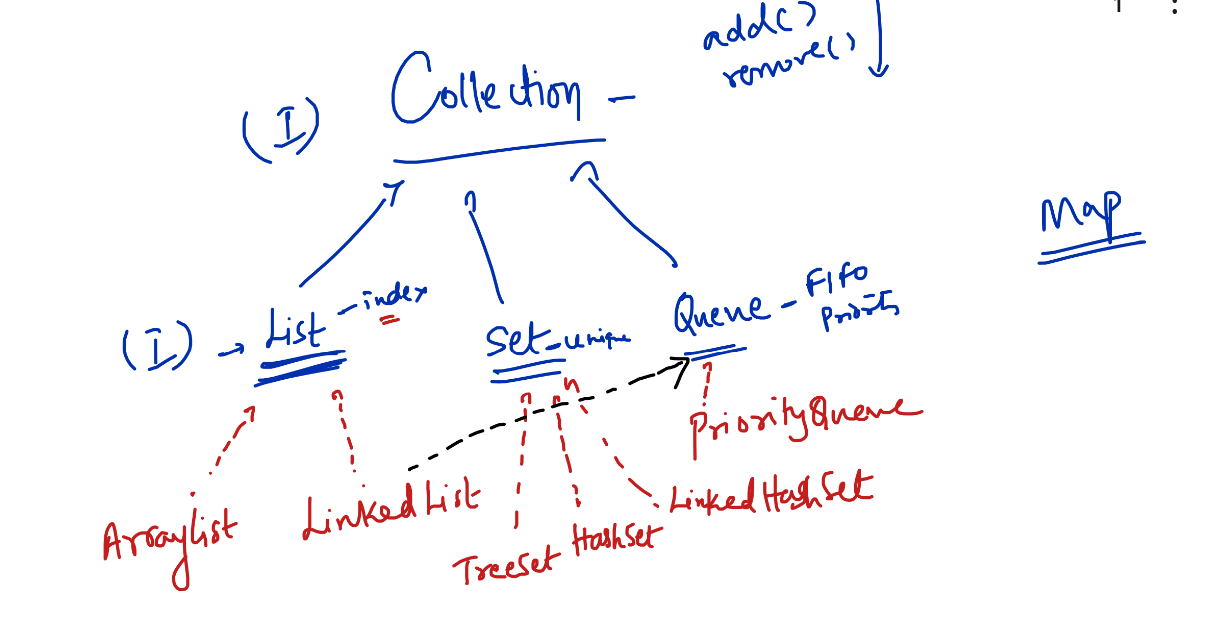
* **JDBC & ORM Framework**
  + JDBC vs ORM
  + Benefits of ORM Framework
  + JPA Specification & Implementations
  + Entities
  + Id & Generators strategy
  + SessionFactory
* **Spring Data JPA**
  + Spring Data JPA
  + Entity
  + Repository
  + CrudRepository
  + JpaRepository
  + Query & Modify
  + Parent & Child mappings

**Day 5**

* **Spring Cloud & Microservices with Spring Boot**
  + Microservices Introduction
  + Why and When to use
  + Monolithic vs Microservice Architecture
  + Benefits of Microservices
  + Challenges in using Microservices Architecture
  + Breaking down a Monolithic app to Microservice app
* **Microservices with Spring** 
  + Spring Cloud & Spring Boot Projects
  + Eureka Server
  + Eureka Client
  + Disabling the client features in Service Discovery
  + Eureka Dashboard
  + Locating the multiple instance of same Microservices

Collection Framework:

Maintain complex data in various formats – it provides many APIs that can manage the complex data like storing in insertion order, sorted order, keeping only unique data



List: It maintains elements in insertion order, it is index based

ArrayList: Maintains elements in contiguous memory address

LinkedList: Maintains elements in non-contiguous memory address

Set: It maintains only unique elements

TreeSet: It maintains elements in sorted order

HashSet: It maintains elements in random order, its retrieval is faster compare to other algorithms

LinkedHashSet: It maintains elements in insertion order

Queue: It is used when you want to remove the element and process it

Priority Queue: It removes elements in sorted order

List sorting: You can sort the elements in the List using the sort method, it takes Comparator as an argument

Comparator<T>: It is a functional interface which has one abstract method compare(T t1, T t2)

class SortById implements Comparator<Employee> {

// returns +ve, 0 or -ve  
 public int compare(Employee e1, Employee e2) {   
 100 – 100 : 0  
 return e1.getId() – e2.getId(); // it can return 0 or -ve or +ve  
 }  
}

list.sort( new SortById() ); // old approach of implementing Comparator.

You can use anonymous implementation to avoid creating multiple classes

list.sort(new Compator<Employee>() {   
 public int compare(Employee e1, Employee e) { …. return int …}  
});

In Java 8 lambda expression is introduced to minimize writing anonymous implementation: it can be applied only for functional interfaces (having only one abstract method)

list.sort( (e1, e2) -> e1.getId() – e2.getId() );  
// sorting in descending order  
list.sort( (e1, e2) -> e2.getId() – e1.getId() );   
// sorting based on the salary  
list.sort( (e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary() ) );   
// sorting based on the date of joining – LocalDate has a compare method  
list.sort( (e1, e2) -> e1.getDoj().compareTo(e2.getDoj()));   
// sorting based on the name – String has a compareTo method  
list.sort( (e1, e2) -> e1.getName().compareTo(e2.getName()));

Java Streams: These are data in the collections which you can process in an efficient way(without changing the existing datastructure) with less code like filtering, sorting, transforming, iterating

In Java Streams we have two types of operations:

intermediate operation: filter, sort, map – they give a new stream so that you can chain the methods

ex: stream().filter().filter().sorted().map()

terminal operation: forEach, collect – they are the last operation in the stream, you can’t chain the stream methods

ex: stream().forEach()

ex: stream().collect()

filter(): it is used to apply the conditions on the stream to get only those data that matches to the condition

sorted(): it is used to apply the sorting logic using Comparator

map(): It is used to transform a stream into another form, like transform employee to string or numbers or to person

collect(): It is used to convert from one collection to another

forEach(): It is used to iterate the stream

Comparable vs Comparator

Comparable is used when you want to apply default sorting technique / natural ordering

ex: String, Integer, Double, LocalDate they all have default sorting technique i.e., ascending order

Comparable must be implemented in the same class that you want to compare, it has a method called compareTo(T t)

Comparator is used to override the default sorting technique, it has a method called compare(T t1, T t2), it is implemented in a separate class

|  |  |
| --- | --- |
| Comparable<T> | Comparator<T> |
| Class Employee implements Comparable<Employee> { …. } | Class Employee { }  class SeparateClass { lambdas for Comparator } |

Comparable implementation within the same class:   
class Employee implements Comparable<Employee> {  
 public int compareTo(Employee emp) { return this.id – emp.getId(); }  
}  
Comparator implementation outside the class:  
Comparator<Employee> compareSalary = (x, y) -> Double.compare(x.salary,y.salary);

Note: If you don’t have Comparable or Comparator and try to sort you get an exception at runtime

list.stream().sorted(): Gives exception if Employee doesn’t have Comparable, it has Comparable then sorts   
list.stream.sorted( (x, y) -> x.id – y.id ): Uses Comparator, if Employee has Comparable then it will be overridden by Comparator

Postgres

Stores all the data in a pgsql\_data, you must create this folder to initialize the database

Steps to perform

* + - 1. creating a folder(done only once): pgsql\_data
      2. Initializing the database(done only once): initdb.exe -D path-of-pgsql\_data -U postgres -W -E UTF8 -A scram-sha-256

>>> it asks you to enter password and also to confirm password : Welcome@123

* + - 1. Starting the database(done whenever you want to start the database):   
         *pg\_ctl -D path-of\_pgsql\_data -l logfile start*
      2. Login to the database with username & password:  
         psql.exe -U postgres

Note: All the above steps are performed in the terminal

Datatypes in postgres

1. int
2. bigint
3. varchar
4. date
5. timestamp
6. json

Creating & connecting to the database

create database npci\_db;

\c npci\_db;

Creating serials: generate numbers automatically starting from 1

create table employee(id serial, name varchar(20));  
insert into employee(name) values(‘Alex’);

Creating sequence: When you want to start from a particular number

create sequence emp\_seq start 500;

Generating the sequence

insert into employee values(nextval(‘emp\_seq’), ‘Alex’);

Parent & Child table relationships:

primary key & foreign key relationships

customer table linked with account table

employee table linked with department table

student table linked with marks table

Full text search

It is mainly used to perform complex searches where it can search the derived words from the base words by ignoring the stop words like to, from, but, between, and

inspire -> inspires, inspiration, inspired, inspirational

friend -> friendly, friends, friendship

satisfy -> satisfies, satisfied, satisfaction

postgres uses tsvector to identify all the base & derived words from the dictionary

In Full text search you will use 2 main functions

1. to\_tsvector(column): It takes the column and identifies all the base & derived words in the dictionary
2. to\_tsquery(text): it takes the text that needs to be searched against the tsvector.

Index: It is used to increase the performance when you are searching

Transaction isolation level

By default transaction is read committed for all the users, but we can create a transaction so that when multiple users are in multiple transactions they don’t see other users modification even if they commit, this is called as repeatable read

Spring Framework

Framework simplifies developing applications by providing all the common features every application needs like type conversion, design patterns, exception handling so that within less time developers can get more work done.

Spring framework: It is an application framework that helps you to develop various types of applications like:-

1. web application
2. enterprise application
3. desktop application
4. cloud based applications / microservices

Dependency Injection:

It is a process where an object is supplied to another object, in spring framework there will be a spring container that creates object and supplies them to their dependencies

ex: DAO object supplying to service object, service object supplying to controller object

Spring Projects

XML file: This will configure the classes that spring framework must instantiate

Libraries: You must add spring libraries in your project, you can use maven dependencies for this.

Java Project => Convert to Maven Project => Add spring context in pom.xml

interface EmployeeDao { }  
class EmployeeDaoV1 implements EmployeeDao { }   
class EmployeeDaoV2 implements EmployeeDao { }   
class EmployeeDaoV3 implements EmployeeDao { }

XML File  
-------------  
<bean id = “x” class = “com.npci.dao.EmployeeDaoV3”> </bean>

How to get the object from the spring container

ApplicationContext context = new ClassPathXmlApplicationContext(xmlFile);  
EmployeeDao dao = context.getBean(“x”)

Dependency Injection:

It is a process where an object is injected (supplied) to another object, it can be done in two ways in spring framework

1. Setter Injection: Spring uses setter method to supply the object
2. Constructor Injection: Spring uses constructor argument to supply the object

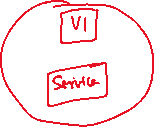
class EmployeeDaoV1 & class EmployeeDaoV2

class EmployeeService {   
 EmployeeDao dao; // can get V1 or V2

public void setDao(EmployeeDao dao) { this.dao = dao; }// spring way  
 this.dao = new EmployeeDaoV1(); // without spring  
}  
XML file



<bean id = “b1” class = “com.npci.dao.EmployeeDaoV1”></bean>  
<bean id = “b2” class = “com.npci.service.EmployeeService>  
 <property name = “dao” ref = “b1” /> #spring supplies V1 or V2  
</bean>



Activity:

Create a class called Datasource with 3 properties: url, username & password, configure this class in XML by providing value to url, username & password (some dummy values), in the main class get the object of the Datasource and print their properties.

Spring Boot:

It simplifies developing spring applications with the help of automated features provided by spring boot libraries(spring boot starter libraries), spring boot doesn’t need any XML file at all instead it uses properties file.

Spring boot auto configures the application based on the spring boot starter library you add in the project.

1. Spring boot starter web: It autoconfigures the application to work in web environment, it provides
   1. Embedded tomcat server
   2. Front controller configuration
   3. Creating the objects in the spring container

Note: All these thing happens without using XML file

1. Spring boot starter data jpa: It autoconfigures the application to interact with any database, it does following tasks
   1. Application will be automatically connected to the database by reading datasource details from property file (application.properties)

Spring boot uses annotations to perform various operations in the application

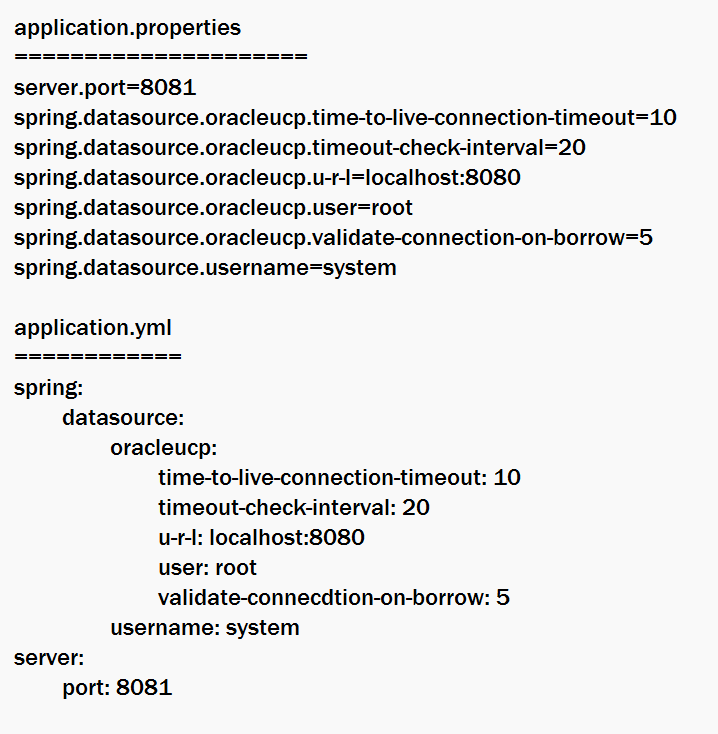
@Component  
@Service  
@Repository  
@RestController  
@Autowired  
@Qualifier  
@Controller  
@Bean

@Component: This annotation lets spring container to create the object of the class, it is written on top of the class, it is a replacement to <bean> tag

@Service, @Repository, @RestController are all sub-types of @Component, means any class having these annotations will be instantiated automatically  
@Autowired: This is used to supply the object to another object i.e., for dependency inject

@Controller & @RestController: It comes in controller layer  
@RestController is used in webservice: view will not be part of the application  
@Controller is used in web application: view will be part of the application

Properties & YAML: These are text files that can have application properties



Building jar / war:

Your application can be packaged into jar/war so that you can run outside

jar: you can run using java command

java -jar filename.jar [arguments]

war: you need to deploy in an external server environment

@Qualifier: It is used to supply the object when there are multiple instances matching

Try it:

Remove @Qualifier from the Service, and use @Primary on top any one Dao implementation and check what is the output.

@Bean: It is used when spring container wants to register the object created by the user in the spring container, spring automatically calls the method having @Bean

Note: Write @Bean in a class that has @Configuration

@Configuration: It is used when a class is creating an object and want spring container to maintain that object with @Bean

@Configuration  
public class Config {   
 @Bean  
 public A createA() { return new A(); }  
 @Bean  
 public B create() { return B(); }  
}

@Service  
class C {   
 @Autowired  
 private A obj;  
}

Summary

1. Dependency Injection
2. Annotations - @Service, @Repository, @RestController, @Autowired, @Component
3. Layered architecture – View <-> Controller <-> Service <-> Repository

Webservices: These are online services which are available for different technologies which can share the data in a common format(JSON/XML) that can be converted to the structure the technologies understand

ex: IRCTC -> Phone Pay -> Banking

ReST Webservices: ReST stands for Representational State Transfer

Representational: JSON/XML/CSV representation  
State: Data  
Transfer: Transferring the data.

Principles of Restful webservices

1. Location: Represented using URL, every service must have some URL
2. HTTP protocol: They provide various mappings/operations to access the service
   1. CRUD operations : Represented by http methods
      1. GET: Retrieving the data
      2. POST: Creating the new data
      3. PUT: Updating the existing data
      4. DELETE: Deleting the existing data

ATM >> 10000 >> Banking

PUT PUT

ATM >> Check balance >> Banking

GET GET

How to create webservices in spring boot

@RestController: You must use a class with @RestController, webservice code comes in controller layer, it handles the request and generates the response

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping: These annotations are used on the controller class methods to represent the HTTP methods

@RequestMapping: This is an annotation used when you want to map the request for the entry point class or when you want to access a resource from any http methods.

Note: You can use Postman client / Thunder bolt client to test the webservices you have created, because its impossible to access the webservice from all the technologies

Our first webservice program with spring boot

1. Libraries
   1. Web: This provides embedded server and auto-scanning features
   2. Dev tools: It auto-reloads the server when you do changes in the app
2. @SpringBootApplication: It will be present by default when you get the project from spring initializer
   1. This takes care of auto-configuring the application based on the library you have

How to pass data to the webservice

Path Parameter: It is used to parameterize the path, so that client can send dynamic value via path, it will extracted using @PathVariable(“name”)

ex:   
/{x}/{y} >> @PathVariable(“x”) int id, @PathVariable(“y”) long phone;

if the url is /123/88998, then id is 123 & phone is 88998

Request Body: It is used to send the data via body of the request, mainly the complex data is sent because path parameter is limited to 256 characters

Suppose you have a Java class (Profile) with name, phone, dob, email properties, then a JSON having these 4 properties is mapped as below

Request Body: {“name”:”Raj”, “phone”:9999, “dob”:”2000-11-25”, “email”:”abc”}

public ResponseEntity<Object> store(@RequestBody Profile p) {   
 profile object [name = Raj, phone = 9999, dob = 2000-11-25, email = abc ]  
}

Request Parameter: It is also used to send the data to the webservice via query parameter of the URL, but it is sent at the end of the URL with ? and &

ex: http:ip:port/path/subpath?key=value&key=value

Here you must have a webservice with a path & a parameter to read the request parameter

@GetMapping(path = “/subpath”)  
public ResponseEntity<T> fetch(@RequestParam(“key”) Type variable) { }  
Above webservice can be accessed without passing request parameter/query parameter

i.e., it works for both /subpath?key=value

or /subpath

Sending complex data to the webservice using JSON and Java bean

Suppose we are sending profile data: id, name, dob, phone in json format as  
{ “id”:15, “name”:”Vijay”, “dob”:”2000-10-25”, “phone”:999888233}

The java bean must have 4 properties: id, name, dob, phone

Layered architecture

1. View: It will have the presentation logic it could be for Mobile, web, desktop, terminal or etc
2. Front Controller: It is the main entry point for all the incoming requests, it will take care of routing the request to the appropriate controller
3. Controller: It is the one who handles the request & will access the service layer and generate the response
4. Service: It will have business logic & will access database layer (DAO)
5. DAO: It will have database logic

Things to create now

1. Profile: id, name, dob, phone
2. ProfileService: an interface with business methods
3. ProfileServiceListImpl: an implementation to the business interface
4. ProfileController: will have webservices
5. ProfileNotFoundException: This is thrown when id is not found in the database

Embedded server spring boot supports

1. Tomcat ( Default server)
2. Undertow (JBoss): You must add this explicitly by disabling tomcat
3. Jetty (Eclipse): You must add this explicitly by disabling tomcat

How to disable tomcat:

In pom.xml: We must use <exclusion> tag and exclude tomcat artifact-id, and then add <dependency> of undertow or jetty server

Packaging:

1. jar: where server is part of the application embedded & you just need java to run this jar

jar is a collection of classes, xml, property files

1. war: where server is not part of the application, you must run this using an external server

war is a collection of jar + html, css, js, jsp

Spring Boot 3.x minimum needs tomcat 10.x or later when you want to deploy

JDBC

Java Database Connectivity, provides set of API’s(interfaces & classes) to interact with any relational database.

Steps to interact with the database

1. Loading the JDBC driver (every database will have JDBC drivers)

Class.forName(fullyQualifiedDriverClass)

1. Connecting to the database

Connection connection = DriverManager.getConnection(url, username, password)

1. Create Statements to run SQL queries

PreparedStatement statement = connection.prepareStatement(sqlQuery)

sqlQuery = (insert, update, delete, select)

ex 1: sqlQuery = insert into profile values(?, ?, ?, ?)

statement.setInt(1, 200); // converts java int to SQL int  
statement.setString(2, “Alex”); // converts java string to SQL varchar  
statement.setLong(3, 99980008L); // converts java long to SQL bigint or big number  
statement.setDate(4, LocalDate.of(2000, 10, 11)); // converts LocalDate to Date

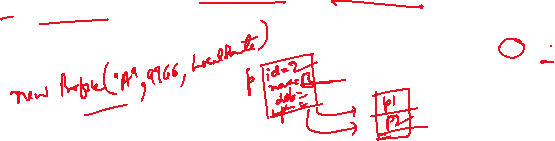
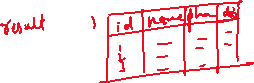
ex 2: sqlQuery = “select \* from profile where name = ?”;  
 statement.setString(1, “Alex”);

1. Execute Queries / Run Queries

// DML queries use executeUpdate, DRL queries use executeQuery  
int count = statement.executeUpdate(); // for DML (insert,update,delete) queries  
ResultSet result = statement.executeQuery(); // for DRL (select)

1. Closing the resources
   1. result.close()
   2. statement.close()
   3. connection.close()

How to use the ResultSet to convert SQL result to Java object  
String query = “select \* from profile”;  
PreparedStatement statement = connection.prepareStatement(query);  
ResultSet result = statement.executeQuery();  
while(result.next()) {  
 Profile p =   
 new Profile(result.getString(2), result.getLong(3), result.getDate(4).toLocalDate());  
 p.setId(result.getInt(1);   
 list.add(p);  
}



ORM:

Object Relational Mapping, it is a framework for the database, it can directly map java objects to the table and vice versa.

It takes care of handling the exceptions, type conversion, connection pooling so that developers can directly perform CRUD operations in less code

ORM provides table information’s in the java bean itself which are called as entities

@Entity  
@Table(name = “profile”)  
class Profile {  
 @Id  
 @Column(name = “id”) // maps to id column  
 private int profileId;  
 @Column(name = “name”) // maps to name column  
 private String profileName;  
 // ORM maps this to dob column automatically  
 private LocalDate dob;  
}

ORM provides predefined methods like save(), delete(), get():

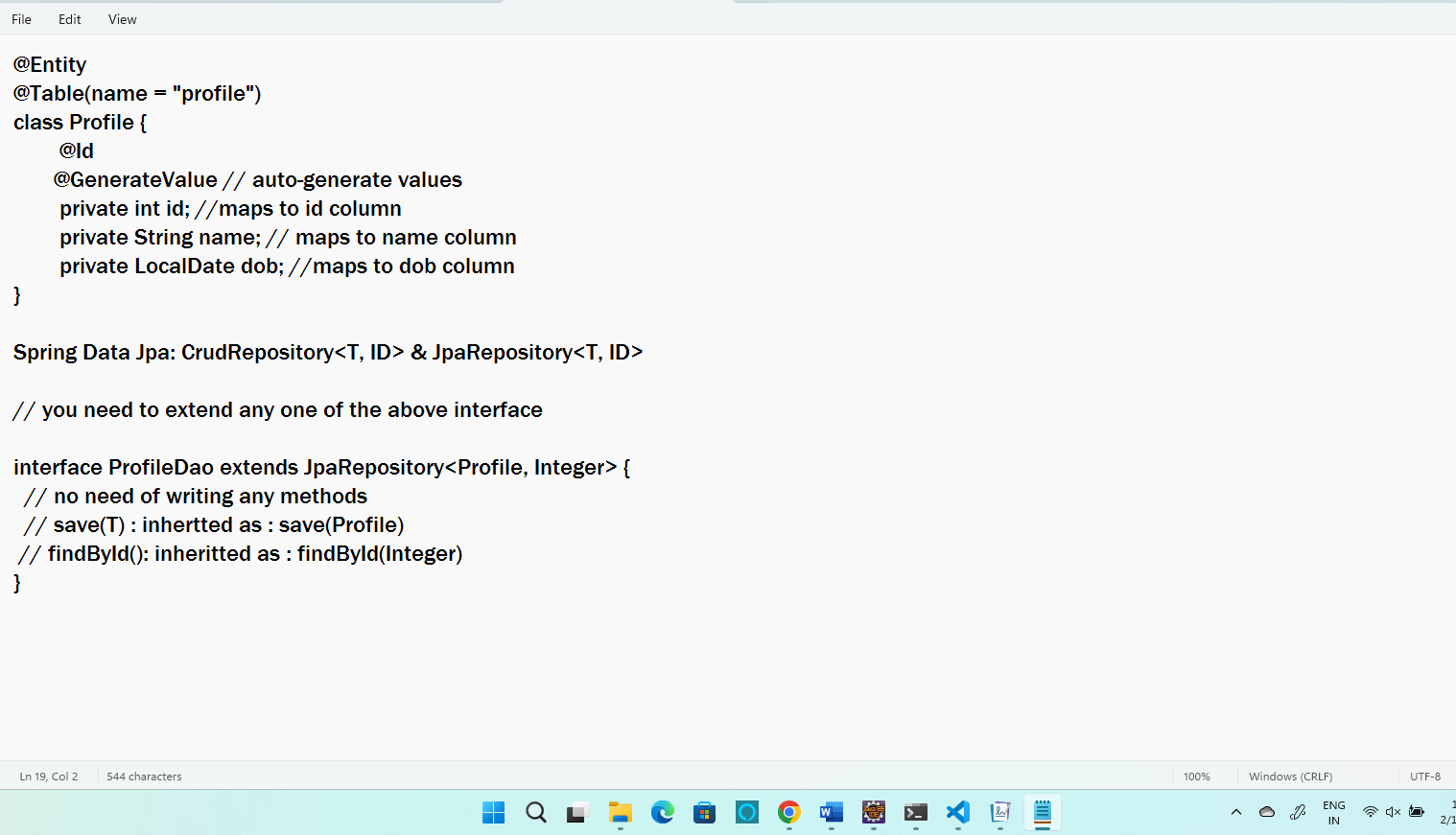
save(profile); // profile objects will be mapped  
delete(200); // deletes the records whose primary key is 200  
get(201); // selects the record whose primary key is 201, returns the profile object will all the properties initialized

JDBC vs ORM

|  |  |
| --- | --- |
| JDBC | ORM |
| You need to write SQL queries | You don’t need to write SQL queries, inbuilt methods auto-generate SQL queries |
| For Exception handling, type conversion you need to write code | ORM handles all the exceptions, auto-convers the java types to sql types using entity class |
| Query you write is database dependent because its SQL | Query you write is not SQL, its JPQL, which is a query written for an entity, ORM converts JPQL to SQL  i.e., select x from com.Profile x; // JPQL  select id, name, dob, phone from profile; //SQL |

Spring Boot provides ORM feature using Spring Data JPA: Its advantage is you don’t have to create a DAO layer, you only need to extend the interfaces provided by Spring Data Jpa, these interfaces will have CRUD methods which are inherited to your interface & will be implemented automatically.

Those interfaces are   
1. CrudRepository<T, ID>: save(T), deleteById(ID), findById(ID)  
2. JpaRepository<T, ID> extends CrudRepository<T, ID>: saveAll(List<T>), sort(…)  
  
T: is an entity class name  
ID: is the primary key type



Spring Data Jpa gives an auto-implementation to the ProfileDao, your job is to auto-wire this in the service layer, there are 2 steps you need to do when you want to interact with the DB using spring boot.

1. create an interface & extend either CrudRepository or JpaRepository interface
2. Autowire the created interface in the service layer.

ORM framework is specified by JPA which has many implementations

1. Hibernate: It can be used in a non-spring framework project
2. JPA
3. Spring Data JPA
4. Toplink
5. iBatis

What JPA specifies

1. Annotations are used to map java objects to sql types
2. Connections are automatically established using a factory pattern called SessionFactory (maintains pool of connection objects)
3. Primary key must be present in the entity – because using primary key many operations are done like delete, get, update based on the primary key
4. JPQL must be used to query (It is a database independent language) instead of SQL – when you want to create custom queries to perform some operations when inbuilt methods are not available then you can use JPQL.

Spring Data JPA: It autoconfigures the database by providing

1. Connections
2. Implementations to the DAO layer
3. All the JPA standards like annotations, primary annotations, and etc.

Project Setup

1. Libraries: Spring Web, Spring Data JPA, Postgres Driver, Devtools
2. Classes: Controller, Service, DAO interface(you don’t implement DAO interface)
3. Dao interface must extend either JpaRepository or CrudRepository
4. Entity class: Represent the table structure

Methods available in CrudRepository<T, ID> / JpaRepository<T, ID>

1. public T save(T t): returns the saved entity, it can be used to save / update
2. public Optional<T> findById(ID id): returns the Optional<T> based on the id
3. public void deleteById(ID id): deletes the entity based on the id
4. public List<T> findAll(): returns all the entities

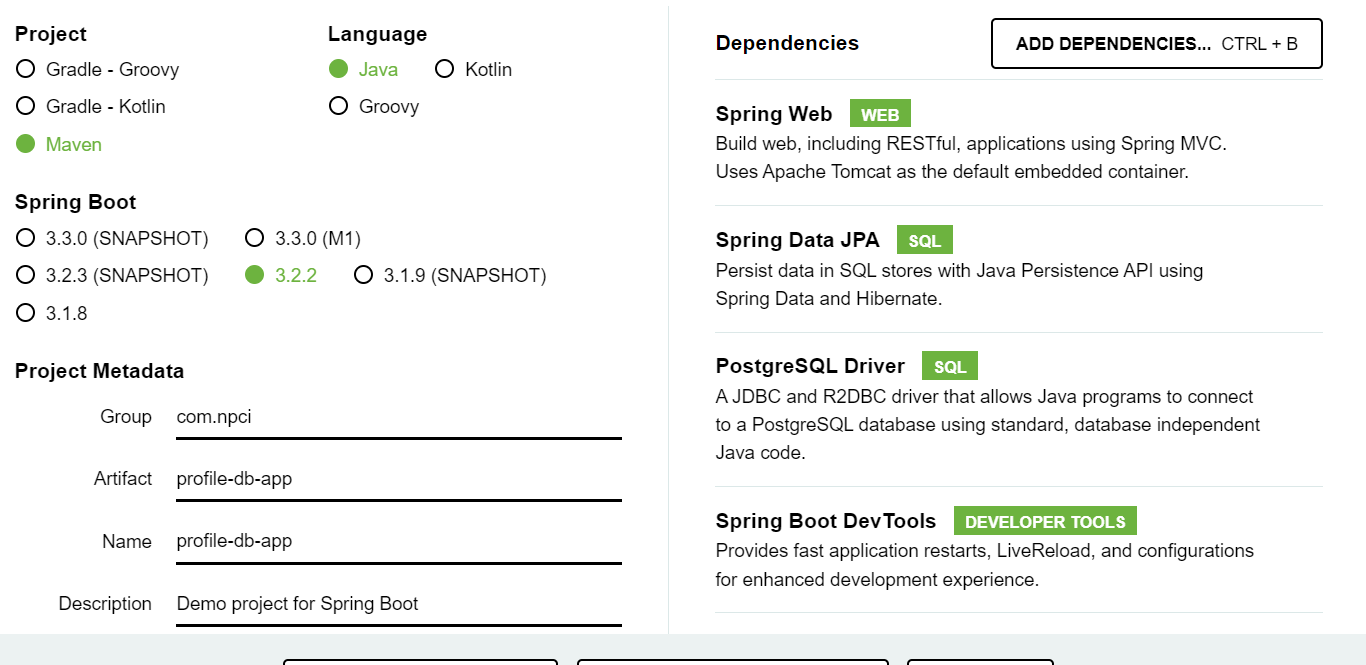
T: is an entity type like Employee, Profile, Customer, Account

ID: is a primary key type like Integer, Long

List of interfaces & classes we are going to create

1. class Profile : @Entity, @Id, @Column, @Table
2. class ProfileNotFoundException: throws exception when a id not found
3. interface ProfileDao extends JpaRepository<Profile, Integer>: Autoimplemented by spring
4. interface ProfileService: store, delete, find, update and so on
5. class ProfileServiceImpl implements ProfileService: @Service, autowire ProfileDao
6. class ProfileController: @RestController, autowire ProfileService
7. application.properties: Provide datasource information’s

Project Setup



Activity:

1. Implement webservices to call other methods of ProfileService
2. Create a method in DAO that updates name by phone number
   1. You must use @Query to write update query in JPQL
   2. You must use @Modify on top the same method

Note: @Modify is used when you write custom JPQL that wants to update the entity

1. Create a webservice to update the name using the phone number with appropriate HTTP methods.

Summary:

JDBC vs ORM

Spring Data Jpa – CrudRepository, JpaRepository

@Query, @Modifying

Mappings:

In ORM you can map an entity to another entity that helps to join the tables, ORM provides annotations to join the tables

1. @OneToOne
2. @OneToMany
3. @ManyToOne
4. @ManyToMany

One to One

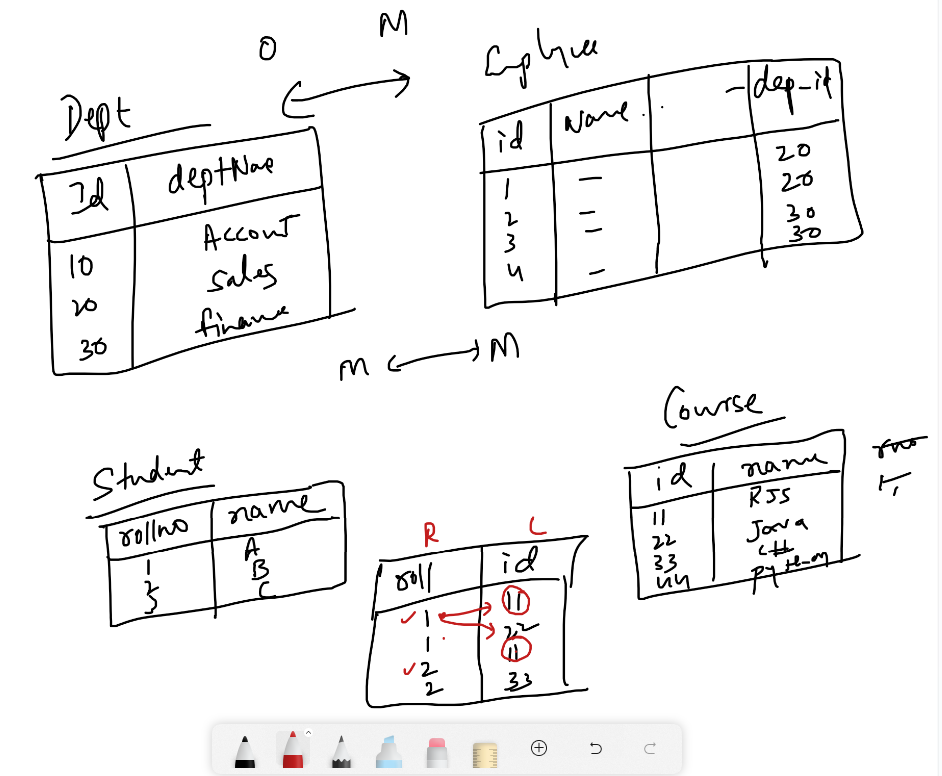
Employee & Address

Employee

|  |  |  |
| --- | --- | --- |
| ID(PK) | Name | Salary |
| 1001 | A | 30500 |
| 1002 | B | 45000 |
|  |  |  |

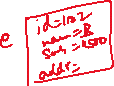
Address

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address\_ID | State | City | Pin | Employee\_ID(FK) |
| 1 | MH | MBI | 4000 | 1002 |
| 2 | KA | BLR | 5000 | 1001 |



ORM takes care of joining the tables and representing in the form entities, you must only create a variable of child table & on top of that you must use mapping annotations

@Entity  
class Employee {   
 id, name, salary;  
 @OneToOne  
 private Address addr;  
}  
  
@Entity  
class Address {   
  
}



Employee

|  |  |  |
| --- | --- | --- |
| ID(PK) | Name | Salary |
| 1001 | A | 30500 |
| 1002 | B | 45000 |
|  |  |  |

Address

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address\_ID | State | City | Pin | Employee\_ID(FK) |
| 1 | MH | MBI | 4000 | 1002 |
| 2 | KA | BLR | 5000 | 1001 |

Things to create

1. Contact class: @Entity, @Id
2. Profile class: create a List<Contact> with @OneToMany, then setters & getters for List<Contact>

@Entity  
public class Contact {   
 // id, name, phone, pid;  
}

@Entity  
public class Profile {   
 // id, name, phone : setters & getters  
 @OneToMany(cascade = CascadeType.ALL)  
 @JonColumn(“pid”)  
 List<Contact> contacts;  
}

Activity: Saving the contact for a particular profile

Create a ContactDao that extends Crud/JpaRepository

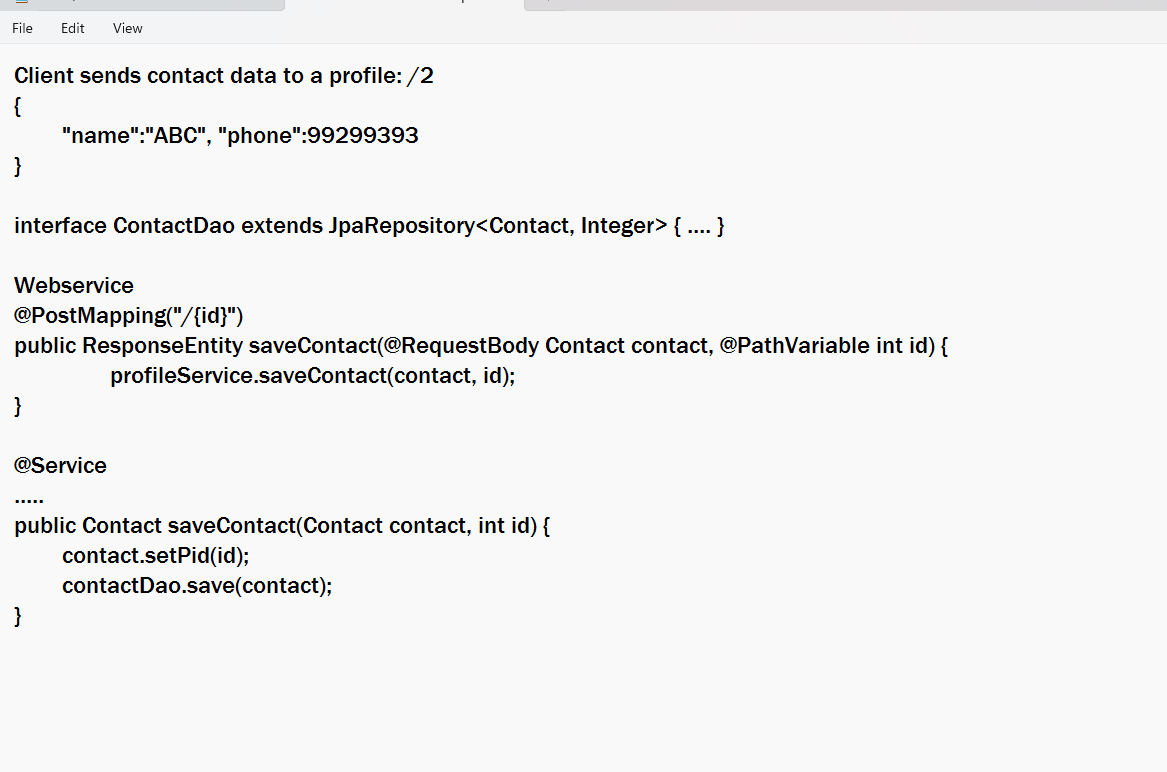
Create a method in a ProfileService as: public Contact saveContact(Contact contact, pid int)

Create a webservice that will use /{profileId} and extracts the contact data in json format & calls saveContact(contact, pid);

From the client you must send only contact data except pid & id, because pid is sent in the path and id is auto-generated

Note: when you save contact, it must save pid also

Solution Hint:



Microservices:

They are small independent services which are loosely coupled from other services of either same application or different applications

Monolithic architecture: All the services are created in the same application & they are tightly coupled, below are the drawbacks of monolithic architecture

1. Changes done in one service requires to test other services
2. Failure of a service might bring down other services
3. Scaling of a particular service is not possible, you must scale all the services
4. Language dependency – you must implement the new feature in the same technology what other services are using.

Microservice architecture: All the services are independent, Advantages you get

1. Changes done in one service doesn’t force to test other services
2. Failure in one service doesn’t impact other services
3. You can scale only a particular service
4. You can use different languages & technologies to create the services

Note: One microservice can call another microservice

When to use microservice and when not

You need to use microservice when your application is used by global customers or has other competitors with similar features

Microservices can be created using any technology

Java – Spring Boot & Spring Cloud projects

Javascript – Node.js

Python – Fast API

Spring Cloud: It provides libraries to develop microservices

Spring Boot: Does auto-configuration based on the spring cloud libraries

Design principles or patterns to create microservices

1. Service Discovery: to register the microservices with their instance – id and physical address
2. Client side load balancer: This is a program who queries the service discovery to send the request to the remote service
3. Distributed Cloud configuration: It is a centralized configuration for the multiple microservices
4. Client side resiliency:

and many more patterns

Spring Cloud: It has implemented all the design principles for microservices, you just need to use those patterns with simple annotations

ex: @EnableEurekaServer : Service discovery   
ex: @LoadBalanced: Client side load balancer  
ex: @EnableConfigServer: This is for distributed cloud configuration

Things to create

1. Service Discovery
   1. Libraries : Eureka Server(spring cloud), Devtools
2. Microservice
   1. Libraries: Eureka Client(spring cloud), Web, Devtools

Eureka Server: This is the library that creates service discovery, run in port:8761, it gives a dashboard to see the microservice instances

Eureka Client: This is the library that registers the microservice in the Eureka Server by default in searching the eureka server localhost:8761 port.

Note: Every microservice is a webservice hence we need a web library

We need to create 2 projects

1. Service Discovery : @EnableEurekaServer on the main class

application.properties

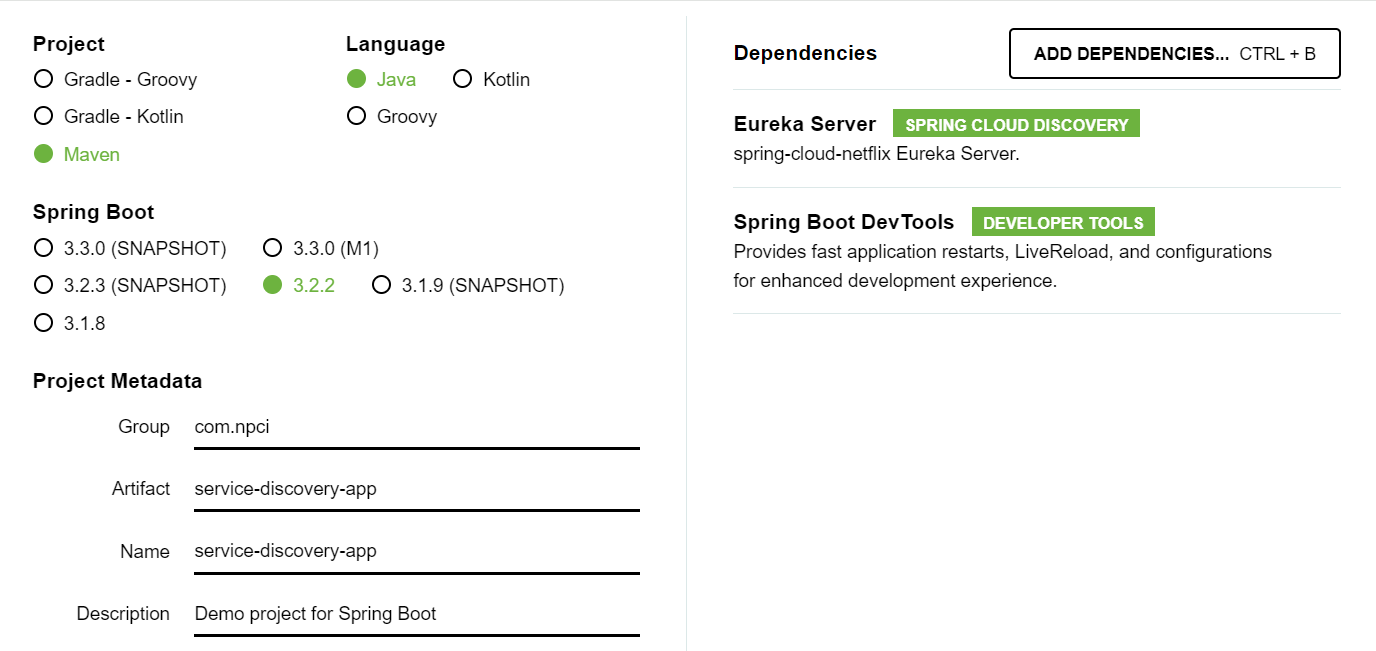
server.port=8761  
# by default eureka server downloads eureka client – we must disable client features using some properties because spring boot auto-configures this as a client

1. Microservice: You add eureka client library that is sufficient

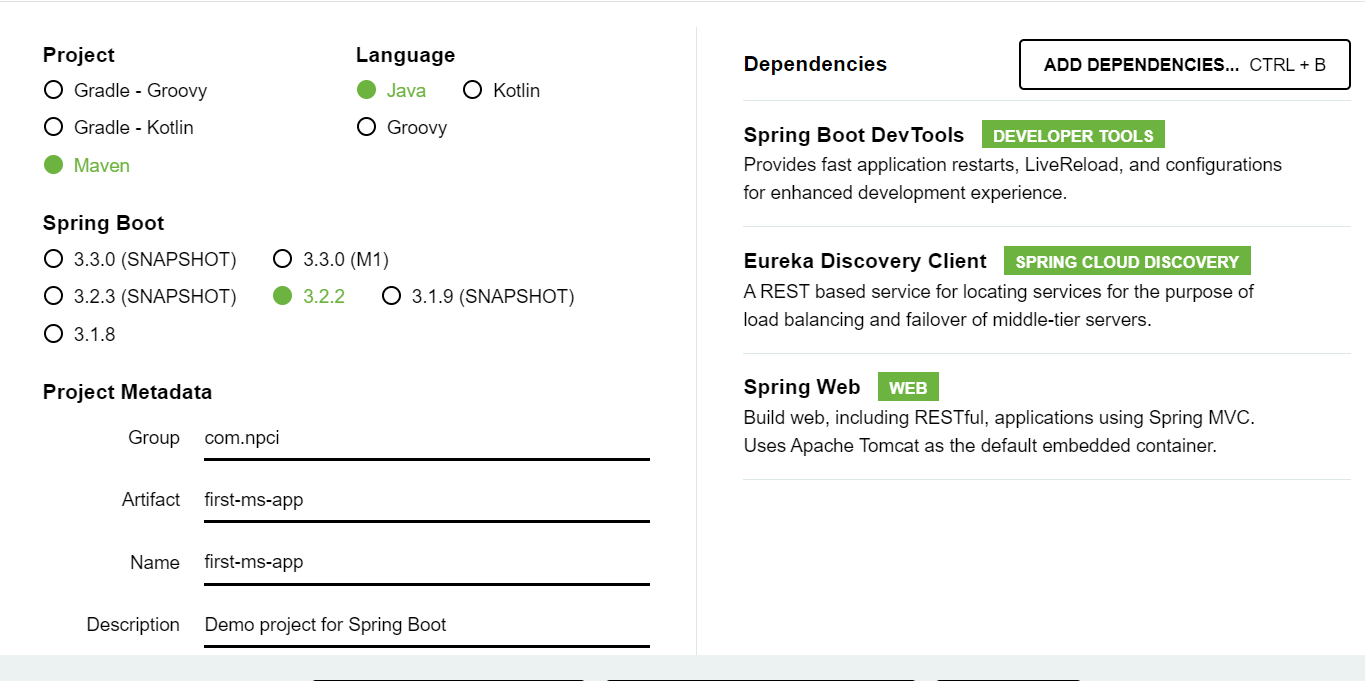
application.properties

spring.application.name=instance-id  
server.port=8081

Service Discovery



Microservice



Activity:

1. Configure the profile-db-app to get register in the service discovery

Solution:

1. Add the eureka client dependency: you must add some extra properties in the pom.xml like spring-cloud.version & dependency management tag
2. in application.properties give instance-id using spring.application.name

Post Test MCQ: After the snack break : 5:00PM