Spring Boot

--------------

Why Spring Boot?

--------------

Reasons:

1. Spring Boot has provided starter packs, for dependencies.

If you are working with Spring Framework, then the programmer has to add multiple dependencies with compatible versions to implement a feature, in the pom.xml

But with Spring Boot, the programmer is no need to search for the compatible versions of the dependencies and no need to add the multiple dependencies in pom.xml file.

Instead, the programmer can add a starter dependency provided by spring boot to implement a feature.

For example,

. To develop Web and RESTful applications, the starter is,

spring-boot-starter-web

. To provide Hibernate integration, the starter is,

spring-boot-starter-data-jpa

. To provide mongodb integration, the starter is,

spring-boot-starter-data-mongodb

1. Spring Boot has provided auto configuration feature.

In this, Spring Boot itself will provide the most of the configurations required for an application, based on the spring boot starters added in pom.xml as dependencies.

1. Spring Boot is mainly introduced to implement Microservices architecture.

Spring Boot supports Spring Cloud and also integrates with the tools like Docker, Kubernetes and cloud environments like AWS, Azure and GCP.

1. Spring Boot applications embed servers like **Tomcat, Jetty, or Undertow** inside the application itself. You don’t need to deploy a .war file to an external server.

Run main() method, and it launches its own Tomcat. Makes deployment and Testing easier.

1. Spring Boot Actuator is a module which provides pre-defined REST endpoints, for monitoring and inspecting the various aspects of the spring boot application.

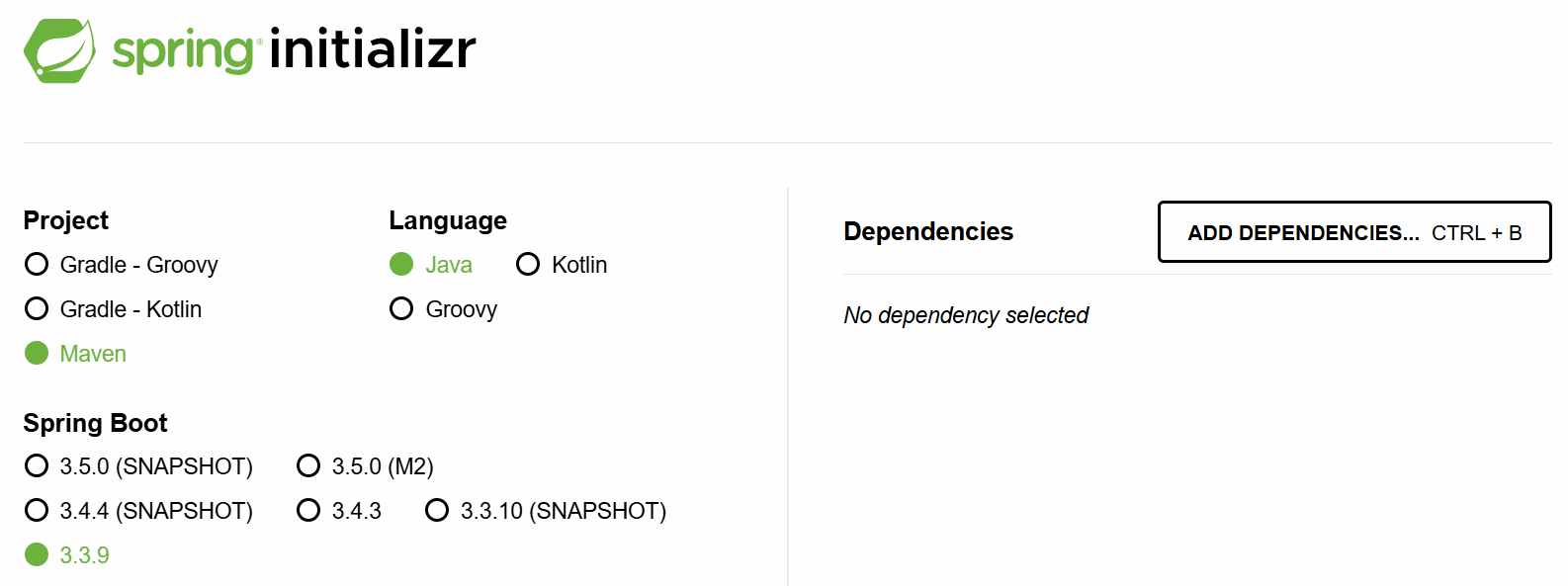
It means, like we can get the health checks, metrics, configurations, runtime details of the currently executing application.

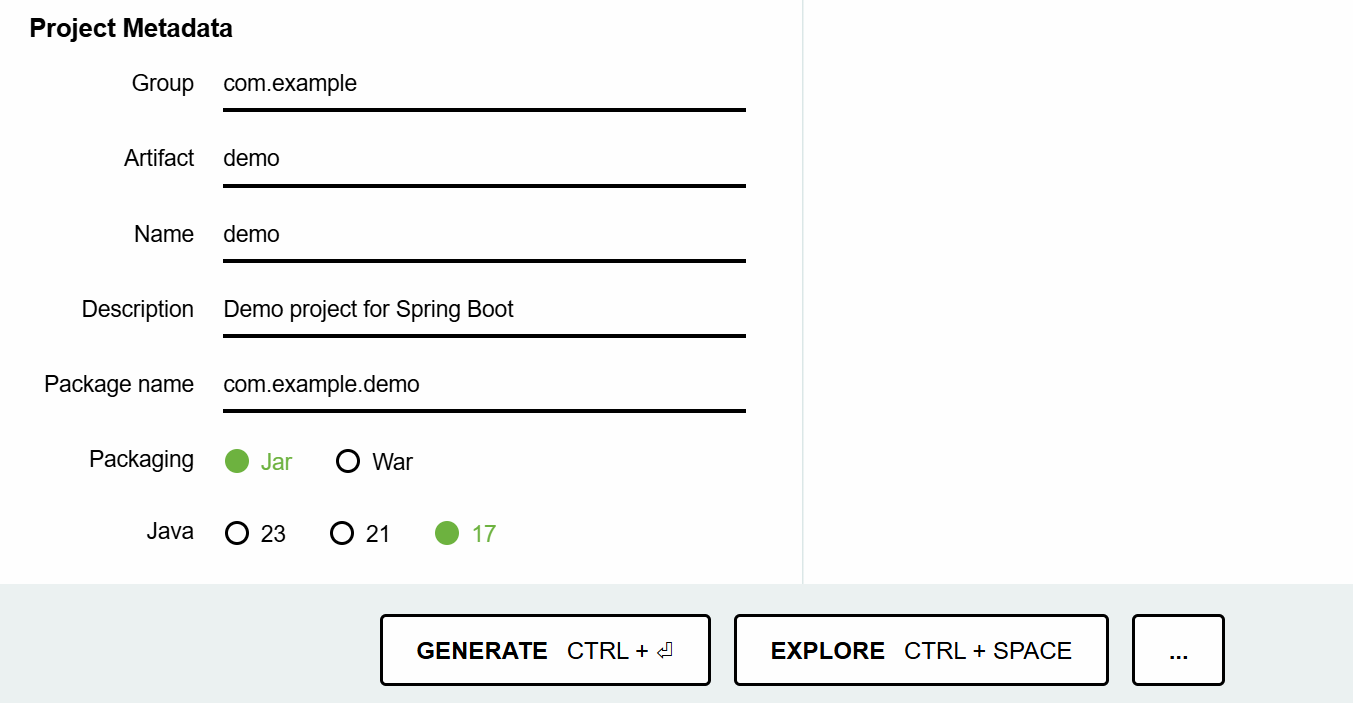
1. Spring Boot provides **JUnit, Mockito, and TestContainers** integrations.

How to kickstart a spring boot project?

------------------------------------

* To kickstart/intial setup, a spring boot project, we have 2 options.
* 1. we can use spring initializer tool(web-based tool)
* 2. we can use spring starter project option in STS IDE
* You can visit the url, start.spring.io to open spring initializer.
* Fill the required details, and click on Generate.
* A Zip file is downloaded. Extract it.





* Now you can import the spring boot project into your workspace.
* Suppose, if you are working with STS IDE, then you can choose Spring Starter Project, to create a spring boot project.
* Note: STS also internally uses Spring Initializer tool.

understanding pom.xml:

-------------------

* we can able to observe the below 3 changes in the pom.xml file, when a spring boot project is created.

1. <parent> tag
2. starter dependencies
3. <plugin> tag.

* Every spring boot project that we create, will be a child module of a parent module given by spring boot called,

spring-boot-stater-parent.

* In a child module’s pom.xml file, to configure the parent module, we have to use <parent> tag.
* In Spring Boot, our project will inherit the default configurations from the parent project.
* The parent project provided by Spring Boot, will provide the required maven plugin configurations automatically like,

maven-compiler-plugin

maven-clean-plugin

maven-resources-plugin

maven-surefile-plugin, etc.

* The starter dependencies in the pom.xml will use the version of the parent project. So, you no need to specify the version for the starter depedencies.

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.3.9</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

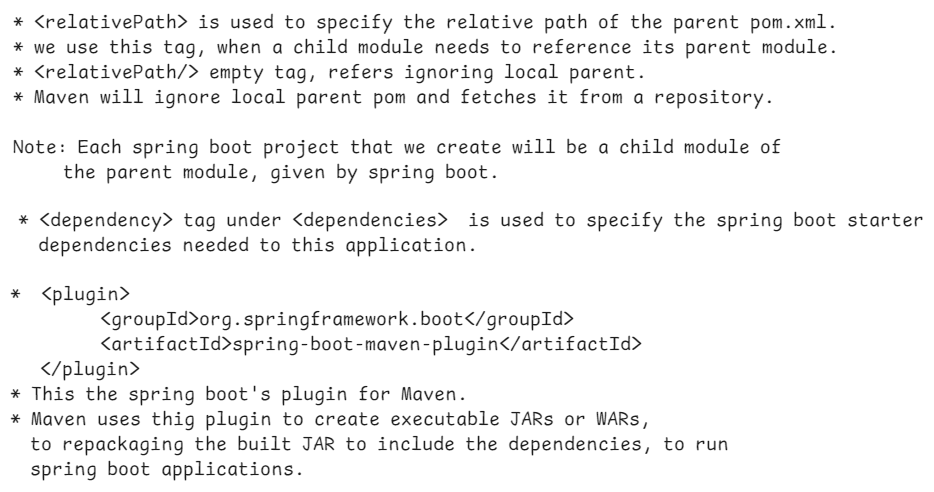
<scope>test</scope>

</dependency>

</dependencies>

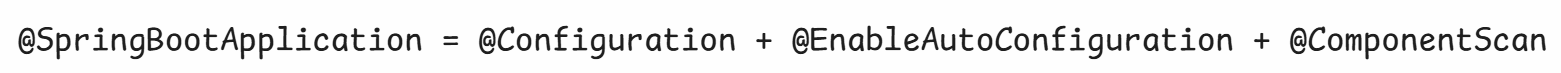
what are the dependency scopes in maven?

1. compile(default)
2. test
3. runtime
4. provided
5. system



@SpringBootApplication:

--------------------



* @SpringBootApplication represents that, this class is the entry point to execute the spring boot application.
* A class with this @SpringBootApplication annotation, contains Java main() method.
* Every spring boot application’s execution starts from main() method only.
* A class with @SpringBootApplication is a Java configuration class. So, if any addition bean configurations are required, you can define them in this class.
* In Spring Boot, a module is provided called spring-boot-autoconfigure module.
* This module goes to project classpath, then identifies the starter dependencies added to the project and then defines all the default configurations required.
* @EnableConfiguration annotation, enables this spring-boot-autoconfigure module’s support for the project.
* @SpringBootApplication annotation also contains @ComponentScan. So, by default, Spring will start components autoscanning from the current package in which boot application class is available. Further, Spring will also enters into its sub-packages.
* In a Spring Boot project, the main class looks like below.

@SpringBootApplication

**public** **class** SampleApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SampleApplication.**class**, args);

}

}

what SpringApplication.*run*(SampleApplication.**class**, args); statement will do?

----------------------------------------------

* Spring Boot has provided SpringApplication class with a static method called run() to do the below tasks.

1. starts the ApplicationContext container of spring.
2. loads the command-line arguments into ApplicationContext container.
3. executes the Spring Boot Runner components.

Creating a First Spring Boot application in STS :

-------------------------------------------

1. File 🡪 New 🡪 Spring Starter Project
2. Name : SBHelloApp

Project : Maven packaging: jar

Java version : 17 language: Java

GroupId: in.ashokit

ArtifactId: SBHelloApp

version: 1.0

package: com.ashokit

Next 🡪 Next 🡪 Finish

1. Right click on src/main/java and create a new package.

com.ashokit.bean

1. right click on this bean package and create a new class, HelloBean.

@Component

**public** **class** HelloBean {

**public** **void** sayHello() {

System.***out***.println("Hello, Welcome to Spring Boot!!!");

}

}

1. Open SbHelloAppApplication.java from com.ashokit package, and add the required chanages.

@SpringBootApplication

**public** **class** SbHelloAppApplication {

**static** HelloBean *helloBean*;

@Autowired

**public** SbHelloAppApplication(HelloBean helloBean) {

**this**.*helloBean* = helloBean;

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SbHelloAppApplication.**class**, args);

*helloBean*.sayHello();

}

}

1. Right click on the project -🡪 RunAs --🡪 Maven Build…

-🡪 Goals: package ---🡪 Run.

7. Right click on the project ---🡪 Refresh.

8. Right click on Application class 🡪 RunAs 🡪 SpringBootApp

output:

Hello, Welcome to Spring Boot.

Q) what is the drawback of field injection?

A) field injection doesn’t work for static references.

Q) what is the drawback of dependency injection?

A) dependecy injection doesn’t work for static final references.

How to execute a spring boot project from command line:

----------------------------------------------------

* A spring boot project’s jar file is required for executing the project from command line.
* we can execute the project from command line in two ways.

1. using java command
2. using mvn command

* open a command prompt and move to the project folder, and execute the below command.

D:\Workspaces\SB-Workspace\SBHelloApp> java -jar target/SBHelloApp-1.0.jar

(or)

D:\Workspaces\SB-Workspace\SBHelloApp>mvn spring-boot:run

==========================================================

Spring Boot Runner component:

---------------------------

* If you want to execute a task for one time in a spring boot application, immediately when it starts, then you have to define a Spring Boot runner component.
* For example, if you want to load all the Job Schedules from the Database, immediately when the application starts, then you can define a Spring Boot Runner component for this.
* For example, if you want to insert the application admin’s details to the database, immediately when the application starts, then you can define a Spring Boot Runner component for this.
* To define a Spring Boot Runner component, the Spring Boot has provided two runner interfaces.

1. CommandLineRunner interface
2. ApplicationRunner interface

Note: CommandLineRunner is a legacy interface, and to define runner components, we got a new one called ApplicationRunner.

* The runner components are executed immediately when application starts, if they are registered into the Spring Container.
* So, we should add @Component annotation, to register a runner component into the spring container.

ex:

@Component

**public** **class** MyAppRunner **implements** ApplicationRunner {

@Override

**public** **void** run(ApplicationArguments args) **throws** Exception {

// **TODO** Auto-generated method stub

System.***out***.println("Hello, Welcome To Spring Boot!!!");

}

}

* You can define multiple runner components, for multiple tasks.
* To specify the order of the runner components execution, add @Order annotation, before the class.

ex1:

@Component

@Order( value = 1 )

**public** **class** MyAppRunner **implements** ApplicationRunner {

@Override

**public** **void** run(ApplicationArguments args) **throws** Exception {

// **TODO** Auto-generated method stub

System.***out***.println("Hello, Welcome To Spring Boot!!!");

}

}

ex2:

@Component

@Order(value = 2)

**public** **class** MyAppRunner2 **implements** ApplicationRunner {

@Override

**public** **void** run(ApplicationArguments args) **throws** Exception {

// **TODO** Auto-generated method stub

System.***out***.println("Task2.......");

}

}

Note: If we add the same order for the two or more runner components, then we don’t get any error or exception. But they are executed in the random order.

Q) when you package a spring boot project, in target folder, how many jars are created?

A) 2 jars

<projectname>-<projectversion>.jar.original

<projectname>-<projectversion>.jar

ex:

SBHelloApp-1.0.jar.original

SBHelloApp-1.0.jar

Q) what is the difference between .jar.orginal and .jar?

A) .jar.original contains only application classes without libraries.

.jar contains application classes and also the libraries(jars).

So, we can execute .jar file, not .jar.original file.

Q) Who will convert .jar.original to .jar?

A) When you build a spring application, first maven creates .jar.original file then it repackages the application into .jar file, by using spring-boot-maven-plugin.

=================================================

Spring Boot Banner:

-----------------

In Spring Boot, the **banner** is the ASCII text displayed in the console when the application starts. You can customize or disable it using various **banner properties** in application.properties.

To turn off the banner display:

spring.main.banner-mode=off

By default, Spring Boot looks for a banner.txt file in the src/main/resources directory. You can replace it with your own ASCII art.

**Spring Boot does not natively support images as banners**. The spring.banner.image.\* properties were **removed in Spring Boot 2.4**. Instead, you need to **convert an image into ASCII text manually** and save it in banner.txt.

Use online tool to convert image to ASCII.

<https://www.ascii-art-generator.org/>

command line arguments:

=======================

* Command-line arguments are parameters passed to a Spring Boot application when it starts.
* They allow users to provide dynamic values at runtime without modifying the code.
* These arguments are typically passed using the --key=value format.
* A **non-option argument** is a command-line argument **without a named key**.
* Unlike **option arguments** (--key=value), non-option arguments are simple positional values passed when starting a Spring Boot application.

ex1:

* A Spring Boot application is deployed in multiple environments like **dev, test, and production**. Instead of changing the application.properties file for each environment, we can use command-line arguments.

java -jar myapp.jar --spring.profiles.active=prod

ex2:

* A **file processing service** that accepts a filename as an argument.

java -jar file-processor.jar input.csv

* The following spring boot runner component reads the command line argument dark.mode and displays the output statement accordingly.

@Component

**public** **class** MyAppRunner **implements** ApplicationRunner {

@Override

**public** **void** run(ApplicationArguments args) **throws** Exception {

// **TODO** Auto-generated method stub

System.***out***.println("Hello, Welcome To Spring Boot!!!");

**if**(args.containsOption("dark.mode")) {

List<String> lst = args.getOptionValues("dark.mode");

**if**(lst.get(0).equals("true")) {

System.***out***.println("Dark mode is enabled");

}

**else** {

System.***out***.println("Dark mode is disabled");

}

}

**else** {

System.***out***.println("Dark mode is disabled");

}

}

}

Spring Boot JDBC

------------------

\* JDBC : Java Database Connectivity

\* JDBC is a technology which could be used to connect a Java application with a Database, to perform CRUD operations.

\* JDBC Driver is a sotware, which makes Java Environment and Database Environment to work together.

\* If we directly use JDBC API in an application, we are going get the below issues.

1. boiler-plate code problem.

2. memory-leak problem.

3. handling the exceptions(checked)

\* boilter plate code means, repeated code like load the driver, establish a connection, create a statement, close the connection, etc..

\* If you do not clean up the connection objects or statement objects properly, then memory leak problem will occur.

\* In every method, where a Jdbc code is written, you have to handle the execeptions with try and catch blocks.

* To avoid these issues, Spring Framework has provided Spring JDBC module and this module has an important class called “JdbcTemplate” class.
* JdbcTemplate class will internally take care of the above issues, and as a developer, we are going to use JdbcTemplate class to execute the SQL queries.
* When you are creating a repository class, you have to use JdbcTemplate class object as dependency object.
* JdbcTemplate class uses DataSource object internally to obtain a connection with a Database.
* You need to configure the below datasource properties in the application.properties file.

spring.datasource.driver-class-name

spring.datasource.url

spring.datasource.username

spring.datasource.password

* JdbcTemplate class has provided the below methods to perform CRUD operations.

update(query) or update(query, values) For INSERT/UPDATE/DELETE commands

queryForMap(query) or queryForMap(query,values) For SELECT to fetch a single row.

queryForList(query) or queryForList(query, values) For SELECT to fetch multiple rows.

queryForObject(query) or queryForObject(query, values) For SELECT to fetch a single value.

execute(query) for CREATE/ALTER/TRUNCATE/DROP commands.

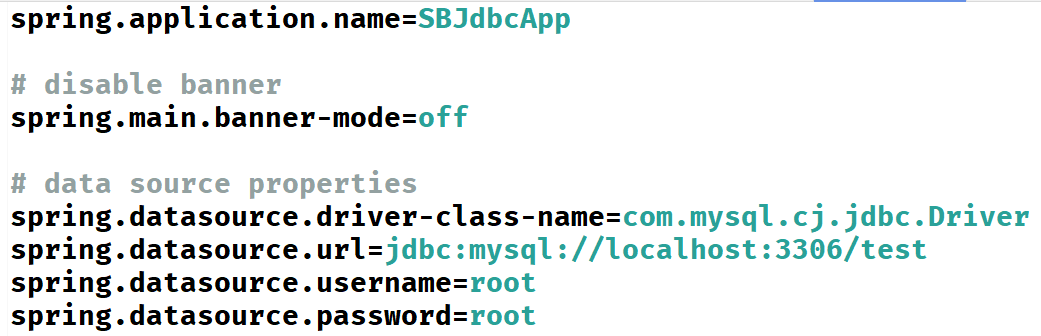
* You need to add JDBC API and MySQL driver dependencies while creating Spring Starter project in STS.
* Note: Before executing the below application, run the below commands from MySQL Workbench.

1. create database test;
2. use test;
3. grant drop on test.\* to ‘root’@’localhost’;
4. flush previleges;



application.properties

--------------------



Employee.java

-----------

**package com.ashokit.model;**

**public class Employee {**

**private int empno;**

**private String ename;**

**private double sal;**

**private int deptno;**

**public Employee() {**

**// TODO Auto-generated constructor stub**

**}**

**public Employee(int empno, String ename, double sal, int deptno) {**

**super();**

**this.empno = empno;**

**this.ename = ename;**

**this.sal = sal;**

**this.deptno = deptno;**

**}**

**public int getEmpno() {**

**return empno;**

**}**

**public void setEmpno(int empno) {**

**this.empno = empno;**

**}**

**public String getEname() {**

**return ename;**

**}**

**public void setEname(String ename) {**

**this.ename = ename;**

**}**

**public double getSal() {**

**return sal;**

**}**

**public void setSal(double sal) {**

**this.sal = sal;**

**}**

**public int getDeptno() {**

**return deptno;**

**}**

**public void setDeptno(int deptno) {**

**this.deptno = deptno;**

**}**

**@Override**

**public String toString() {**

**return "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", deptno=" + deptno + "]";**

**}**

**}**

EmployeeRepository.java

---------------------

**package com.ashokit.repository;**

**import java.util.List;**

**import java.util.Map;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.jdbc.core.JdbcTemplate;**

**import org.springframework.stereotype.Repository;**

**import com.ashokit.model.Employee;**

**@Repository**

**public class EmployeeRepository {**

**@Autowired**

**JdbcTemplate;**

**//insert an employee**

**public void save(Employee e) {**

**jdbcTemplate.update("INSERT INTO EMP VALUES(?, ?, ?, ?)", e.getEmpno(), e.getEname(), e.getSal(), e.getDeptno());**

**}**

**//Get an employee by empno**

**public void findById(int empno) {**

**Map<String, Object> map = jdbcTemplate.queryForMap("SELECT \* FROM EMP WHERE EMPNO = ?", empno);**

**map.forEach( (k,v) -> System.*out*.println(k + " : " + v) );**

**}**

**//Get all employees**

**public void findAll() {**

**List<Map<String, Object>> employees = jdbcTemplate.queryForList("SELECT \* FROM EMP");**

**//for each loop**

**for(Map<String, Object> map : employees) {**

**map.forEach( (k,v) -> System.*out*.println(k + " : " + v) );**

**}**

**}**

**//updates an employee sal by empno**

**public void update(int empno, double sal) {**

**jdbcTemplate.update("UPDATE EMP SET SAL = ? WHERE EMPNO = ?", sal, empno);**

**}**

**//delete an employee by empno**

**public void deleteById(int empno) {**

**jdbcTemplate.update("DELETE FROM EMP WHERE EMPNO = " + empno);**

**}**

**}**

**EmployeeService.java**

**--------------------**

**package com.ashokit.service;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.stereotype.Service;**

**import com.ashokit.model.Employee;**

**import com.ashokit.repository.EmployeeRepository;**

**@Service**

**public class EmployeeService {**

**@Autowired**

**EmployeeRepository repository;**

**public void saveEmployee(Employee e) {**

**repository.save(e);**

**System.*out*.println("Employee is saved successfully!");**

**}**

**public void updateEmployee(int empno, double sal) {**

**repository.update(empno, sal);**

**System.*out*.println("Employee is updated successfully");**

**}**

**public void deleteEmployee(int empno) {**

**repository.deleteById(empno);**

**System.*out*.println("Employee is deleted successfully");**

**}**

**public void selectById(int empno) {**

**repository.findById(empno);**

**}**

**public void selectAll() {**

**repository.findAll();**

**}**

**}**

**MyAppRunner.java**

**--------------**

**package com.ashokit.runner;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.boot.ApplicationArguments;**

**import org.springframework.boot.ApplicationRunner;**

**import org.springframework.core.annotation.Order;**

**import org.springframework.jdbc.core.JdbcTemplate;**

**import org.springframework.stereotype.Component;**

**@Component**

**@Order(1)**

**public class MyAppRunner implements ApplicationRunner {**

**@Autowired**

**JdbcTemplate;**

**@Override**

**public void run(ApplicationArguments args) throws Exception {**

**jdbcTemplate.execute("DROP TABLE IF EXISTS EMP");**

**jdbcTemplate.execute("CREATE TABLE EMP (EMPNO INT, ENAME VARCHAR(20), SAL DOUBLE, DEPTNO INT, PRIMARY KEY(EMPNO))");**

**System.*out*.println("MyAppRunner :: Table is created");**

**jdbcTemplate.update("INSERT INTO EMP VALUES(7209, 'SMITH', 6000.0, 20)");**

**jdbcTemplate.update("INSERT INTO EMP VALUES(7708, 'ALLEN', 4000.0, 10)");**

**jdbcTemplate.update("INSERT INTO EMP VALUES(7652, 'SOPHIA', 8000.0, 30)");**

**jdbcTemplate.update("INSERT INTO EMP VALUES(7788, 'SCOTT', 5000.0, 10)");**

**jdbcTemplate.update("INSERT INTO EMP VALUES(7186, 'MILLER', 6000.0, 20)");**

**System.*out*.println("MyAppRunner :: Rows are inserted");**

**System.*out*.println("\*\*\*\*\*\*\* Database Initial Setup is completed \*\*\*\*\*\*\*\*\*");**

**}**

**}**

**ServiceCaller.java**

**-----------------**

**package com.ashokit.runner;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.boot.ApplicationArguments;**

**import org.springframework.boot.ApplicationRunner;**

**import org.springframework.core.annotation.Order;**

**import org.springframework.stereotype.Component;**

**import com.ashokit.model.Employee;**

**import com.ashokit.service.EmployeeService;**

**@Component**

**@Order(2)**

**public class ServiceCaller implements ApplicationRunner {**

**@Autowired**

**EmployeeService service;**

**@Override**

**public void run(ApplicationArguments args) throws Exception {**

**Employee e = new Employee(7065, "DAVID", 7000, 30);**

**service.saveEmployee(e);**

**System.*out*.println("\*+\*+".repeat(20));**

**service.updateEmployee(7788, 6999.0);**

**System.*out*.println("\*+\*+".repeat(20));**

**service.deleteEmployee(7186);**

**System.*out*.println("\*+\*+".repeat(20));**

**service.selectById(7788);**

**System.*out*.println("\*+\*+".repeat(20));**

**service.selectAll();**

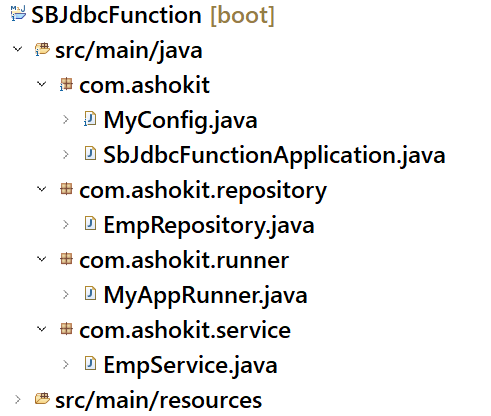
**System.*out*.println("\*+\*+".repeat(20));**

**}**

**}**

**Calling a function:**

* **Spring JDBC API has provided SimpleJdbcCall class to invoke a procedure or a function of a database.**
* **In a database we can define a procedure or a function to implement a task/logic.**
* **A Java application can call that procedure/function, so that the task will be completed in a single trip to the database.**
* **This will reduce the trips b/w application and database and can enhance the performance of an application.**
* **When we add spring-boot-starter-jdbc dependency, the spring boot will autoconfigure JdbcTemplate class into the contianer, but not SimpleJdbcCall.**
* **So, we need to explicitly configure SimpleJdbcCall class, in a configuration class, by creating a @Bean method.**

****

**MyConfig.java**

**--------------**

**package com.ashokit;**

**import javax.sql.DataSource;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.context.annotation.Bean;**

**import org.springframework.context.annotation.Configuration;**

**import org.springframework.jdbc.core.simple.SimpleJdbcCall;**

**@Configuration**

**public class MyConfig {**

**@Autowired**

**DataSource;**

**@Bean**

**public SimpleJdbcCall simpleJdbcCall() {**

**SimpleJdbcCall sjc = new SimpleJdbcCall(dataSource);**

**return sjc;**

**}**

**}**

EmpRepository.java

----------------

package com.ashokit.repository;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.simple.SimpleJdbcCall;

import org.springframework.stereotype.Repository;

@Repository

public class EmpRepository {

@Autowired

SimpleJdbcCall jdbcCall;

public void getBonus(int empid) {

//specify the function name of the database

jdbcCall.withFunctionName("retun\_bonus");

Map<String, Object> output = jdbcCall.execute(empid);

output.forEach((k,v) -> System.out.println(k + " : " + v));

}

}

EmpService.java

----------

**@Service**

**public class EmpService {**

**@Autowired**

**EmpRepository empRepo;**

**public void readBonus(int empid) {**

**empRepo.getBonus(empid);**

**}**

**}**

**MyAppRunner.java**

**-------------**

**@Component**

**public class MyAppRunner implements ApplicationRunner {**

**@Autowired**

**EmpService service;**

**@Override**

**public void run(ApplicationArguments args) throws Exception {**

**// TODO Auto-generated method stub**

**List<String> lst = args.getOptionValues("empid");**

**String str = lst.get(0);**

**int empid = Integer.*parseInt*(str);**

**service.readBonus(empid);**

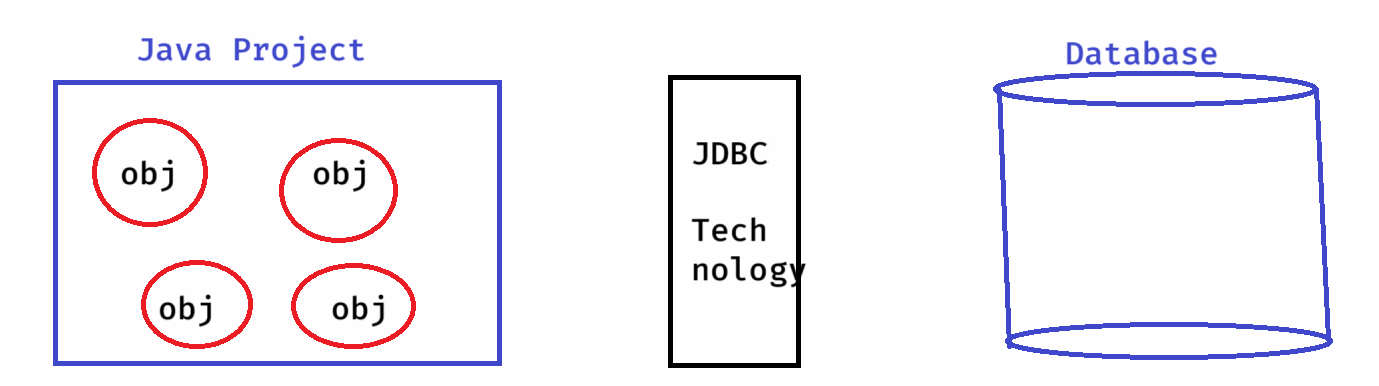
**}**

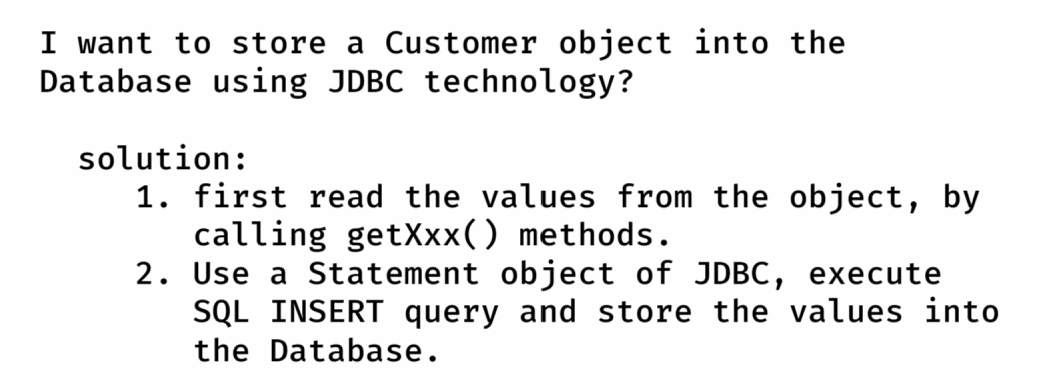
**}**

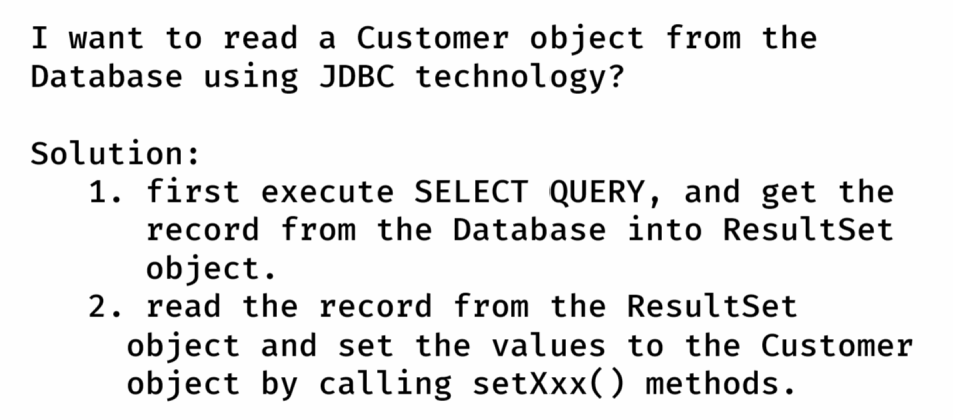
Spring Boot Data JPA

WHY ORM?

------







* JDBC technology is not convinient for storing the objects or retrieving the objects from the database, because manually a developer has to take care about object to values and values to object conversion.
* As a solution, ORM(Object Relational Model) technology was provided to perform the CRUD operations on Java objects, by avoiding the mismatches between Object model and Relational model.
* Based on ORM technology, multiple ORM Tools/Frameworks were developed like TopLink, DataNucleus, ORMLite, MyBatis, Hibernate, etc..
* The most popular ORM Tool is Hibernate.

WHAT IS JPA?

----------

JPA : Java Persistence API

* Before JPA, there are many different ORM Tools/Frameworks are available, to develop Data Accees Layer/Persistence Layer in a Java Project.
* When a developer is switching from One Project to another Project, if a different ORM Tool was used in the other project, then it took some time for the developer to understand and work with that different ORM Tool.
* In this way, the developers has got a burden on using the ORM Frameworks.
* As a solution, Sun Microsystems has provided a Common specification for the ORM Tools, and this specification is called JPA.
* The ORM Tools were re-defined according to the JPA specification.
* JPA specification has provided the below things for the developers to code Data Access Layer.

1. Annotations to define the mappings between the Java classes and Database tables.
2. API to Perform CRUD operations.
3. JPQL(Java Persistence Query Language)

* As a developer, we use JPA specification in the Data Access Layer classes and we configure the JPA Provider like Hibernate or MyBatis,etc.. in the configuration.

WHY SPRING DATA JPA?

* When you are implementing the DAO Layer/Repository Layer,

you have to define the following things.

1. entity classes
2. interfaces
3. implmentation classes
4. configurations.

* The implementation classes contains CRUD operations logic.
* The same CRUD operations logic is duplicated in all the implementation classes. This is code redundency.
* So, to avoid writing the implementation classes, Spring Data JPA module is provided.
* Spring Data JPA eliminates writing the DAO implementation classes from a Java Project.
* You have to write only interfaces, and the Spring will automatically generate the implementation classes at runtime.
* So, in Spring Data JPA, you have to write,

1. entity classes
2. interfaces
3. configurations.

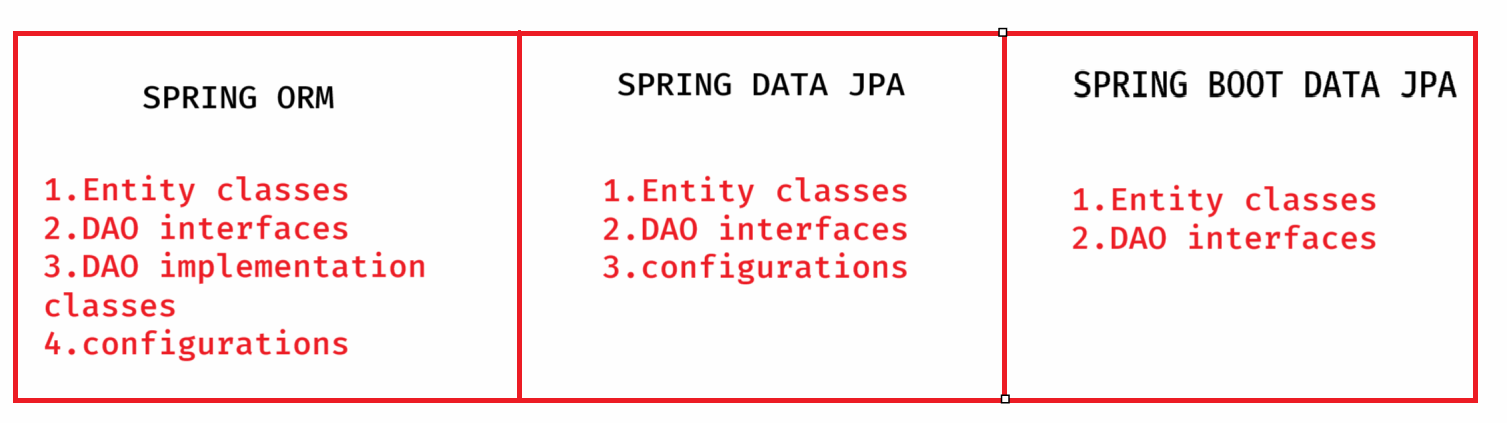
WHY SPRING BOOT DATA JPA?

----------------------

* In Spring Data JPA, we only create interfaces to perform the Database operations. But, we need to define the below configurations.

1. DatSource configuration
2. JpaProvider configuration
3. EntityManager configuration
4. TransactionManager configuration

* In Spring Boot Data Jpa, you no need to define the above configurations, it will be automatically done by Spring Boot only.
* Just you need to add spring-boot-starter-data-jpa dependency in the pom.xml file.
* So, in Spring Boot Data JPA, we only entity classes and create repository interfaces



Lombok api:

* It is a popular java library used to reduce boiler plate code by providing annotations.
* It avoids writing getter and setters, constructors, toString, hashCode and equals methods.
* To use Lombok, first we need to add the dependency to the pom.xml file.

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.34</version>

</dependency>

* annotations in lombok.

@Getter: generates getter methods for each field.

@Setter: generates setter method for each field

@ToString: generates toString() method.

@EqualsAndHashCode: generates equals() and hashCode() methods.

@NoArgsConstructor : generates no-args constructor

@AllArgsConstructor: generates all args constructor

@Data: A convenient annotation that bundles

@Getter, @Setter, @ToString and @EqualsAndHashCode

configuring lombok for IDE:

* first add the dependency in pom.xml
* Goto lombok jar file stored in local repository of maven.

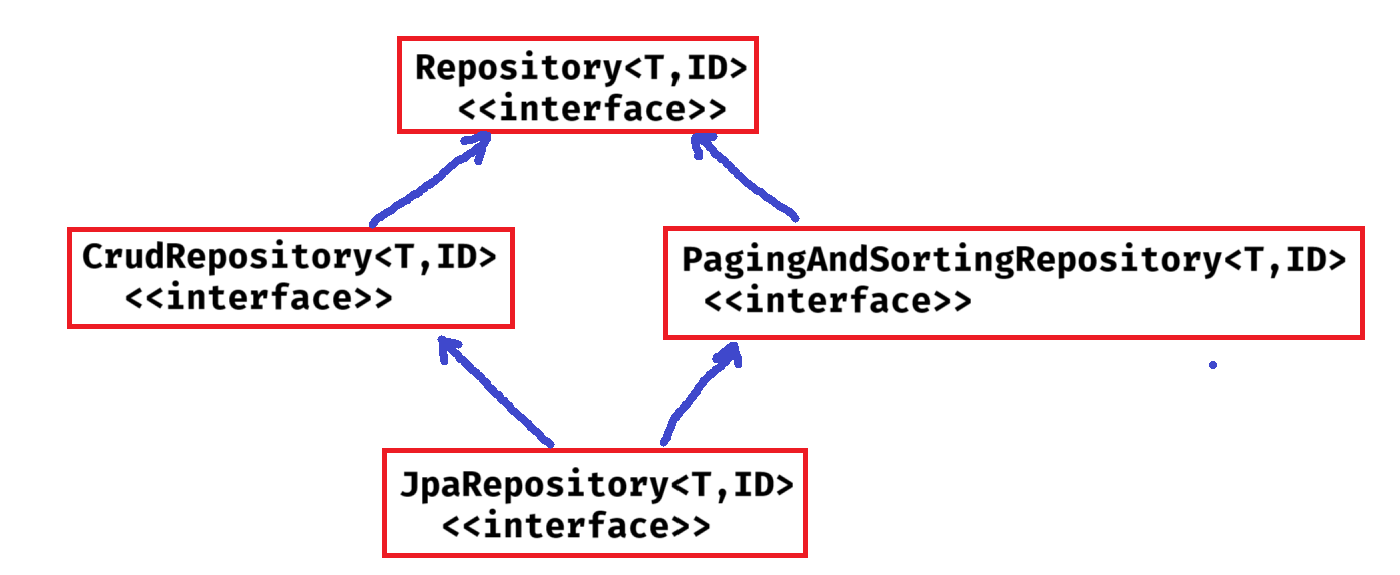
(C:\Users\WINDOWS\.m2\repository\org\projectlombok\lombok\1.18.34)

* type cmd in the location bar
* in the command prompt, run the jar file.

java -jar lombok-1.18.34.jar

* A window is opened, click on specify location and choose SpringToolSuite4.exe, then click on install button. Finally click on finish button.
* Now restart the STS ide.

creating a repository interface:



* Repository is a marker interface, means an empty interface.

**CrudRepository methods**:

1. save(T entity) : saves the given entity. If entity already exist, it updates it.
2. findById(ID id): retrieves entity by its id.
3. findAll() : returns all entities
4. deleteById(ID id): deletes an entity with the given id.
5. delete(T entity): deletes a given entity.
6. deleteAll(): delets all entities.
7. existsById(ID id): checks if an entity exist with the given id.

**PagingAndSortingRepository methods:**

1. findAll(Sort sort): returns all entities sorted according to the provied sort object.
2. findAll(Pageable pageable): returns a Page object, with entities meeting the restriction provided in the pageable object.

**JpaRepository methods**:

* JpaRepository is the most commonly used interface, because it provides crud operations, paging and sorting operations and also some JPA-specific operations.

1. saveAndFlush(T entity): saves and entity and flushes the changes instantly.
2. deleteAllInBatch(): deletes all entities in a batch.
3. getOne(ID id): returns a reference to the entity with the given id, without actually loading it.

* **When you are creating your repository interface, you should extend CrudRepository to perform CRUD operations on entity objects.**
* **If you want to perform pagination and/or sorting operations also, then extend PagingAndSortingRepository.**
* **Instead of extending CrudRepository and PagingAndSortingRepository, you can also extend JpaRepository.**

**ex:**

**public interface EmployeeRepository extends JpaRepository<Employee, Integer> {**

**}**



**entity class PK field datatype**

* **In application.properties, you should configure Data Source properties and JPA properties.**
* **ex: application.properties**

**------------------------**

**# DataSource properties**

**spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver**

**spring.datasource.url=jdbc:mysql://localhost:3306/test**

**spring.datasource.username=root**

**spring.datasource.password=root**

**# JPA properties**

**spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect**

**spring.jpa.show-sql=true**

**spring.jpa.properties.hibernate.format\_sql=true**

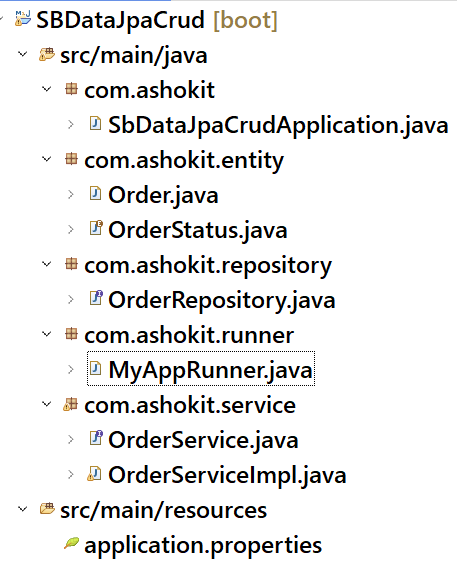
**spring.jpa.hibernate.ddl-auto=update**

* Hibernate has provided dialect classes for each database to generate SQL queries according to the underlying database.
* **spring.jpa.hibernate.ddl-auto defines how Hibernate handles database schema updates. The possible values are,**

1. none : no changes to the database.
2. create : creates tables, drops existing ones.
3. create-drop: creates tables on appln startup and drops them when the appln stops.
4. update: updates the schema without dropping the data
5. validate: validates the schema, but doesn’t change it.

Example on Data JPA CRUD operations:

------------------------------------



**Download the application from the below URL:**

[**https://github.com/ShekherJava/SBDataJpaCrud.git**](https://github.com/ShekherJava/SBDataJpaCrud.git)

1)**Entity class creation**

==================

package com.ashokit.entity;

import java.time.LocalDate;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.EnumType;

import jakarta.persistence.Enumerated;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Entity

@Table( name = "ORDERS" )

@NoArgsConstructor

@AllArgsConstructor

@Data

public class Order

@Id

@Column( name = "ORDER\_ID")

private Long orderId;

@Column( name = "CUSTOMER\_ID")

private Long customerId;

@Column( name = "DATE\_PURCHASED")

private LocalDate datePurchased;

@Enumerated(value = EnumType.STRING)

@Column( name = "STATUS")

private OrderStatus;

}

**OrderStatus Enum**

package com.ashokit.entity;

public enum OrderStatus {

ORDERED("ordered"),

PROCESSED("processed"),

DELIVERED("delivered"),

REFUNDED("refunded"),

CANCELED("canceled"),

;

private String value;

private OrderStatus(String value) {

this.value = value;

}

public String getValue() {

return value;

}}

Step2:

Repository class creation

package com.ashokit.repository;

import org.springframework.data.repository.CrudRepository;

import com.ashokit.entity.Order;

public interface OrderRepository extends CrudRepository<Order, Long> {

}

3)Service Interface

package com.ashokit.service;

import com.ashokit.entity.Order;

public interface OrderService {

Order saveOrUpdate(Order order);

Order getOrder(Long orderId);

Iterable<Order> getAllOrders();

void deleteOrder(Long orderId);

}

**Implement class**

package com.ashokit.service;

import java.util.List;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.ashokit.entity.Order;

import com.ashokit.repository.OrderRepository;

@Service( "orderService" )

public class OrderServiceImpl implements OrderService {

@Autowired

OrderRepository repository;

@Override

public Order saveOrUpdate(Order order) {

return repository.save(order);

}

@Override

public Order getOrder(Long orderId) {

Optional<Order> opt = repository.findById(orderId);

if(opt.isPresent())

return opt.get();

return null;

}

@Override

public Iterable<Order> getAllOrders() {

return repository.findAll();

}

@Override

public void deleteOrder(Long orderId) {

repository.deleteById(orderId);

}

}

**4)Runner Component**

package com.ashokit.runner;

import java.time.LocalDate;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.ApplicationArguments;

import org.springframework.boot.ApplicationRunner;

import org.springframework.stereotype.Component;

import com.ashokit.entity.Order;

import com.ashokit.entity.OrderStatus;

import com.ashokit.service.OrderService;

@Component

public class MyAppRunner implements ApplicationRunner {

@Autowired

OrderService;

@Override

public void run(ApplicationArguments args) throws Exception {

//calling saveOrUpdate(order)

Order order = new Order(19002L, 102L, LocalDate.now(), OrderStatus.PROCESSED );

orderService.saveOrUpdate(order);

System.out.println("Order saved successfully!");

System.out.println("=========================");

//calling getOrder(orderId)

// Order o = orderService.getOrder(19001L);

// System.out.println(o);

//calling getAllOrders()

Iterable<Order> iterable = orderService.getAllOrders();

iterable.forEach(ord -> System.out.println(ord));

}

}

**In resources ->application.propertices inside we can use(propertices)**

# switch off the banner

spring.main.banner-mode=off

# DataSource properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=root

# JPA properties

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.hibernate.ddl-auto=update

Query approaches in Data JPA:

-----------------------------

* When we extend JpaRepository, our DAO layer will get data retrieval operations like findById() and findAll().
* Suppose, if we want more data retrieval operations on our DAO layer then we have to add custom method declarations to our repository interface.
* To add custom method declarations, we have to use Query approaches.
* Query approaches are 3 types.

1. query creation by method name
2. using @Query annotation
3. using @NamedQuery annotation.

query creation by method name:

* we add a method declaration to our repository interface.
* Spring Data Jpa, will automatically generates a query based on method name, and it executes that query to return the entities.
* The method name should follow a structure.

**findBy or readBy or getBy + PropertyName + Condition(optional)**

example1:

List<Employee> findByEname(String ename);

The above method generates a query like this;

SELECT \* FROM EMP WHERE ENAME = ?

example2:

List<Employee> findByFirstNameOrLastName(String firstName, String lastName);

The above method generates a query like this;

SELECT \* FROM EMP WHERE FIRST\_NAME=? OR LAST\_NAME=?

example3:

List<Employee> findBySalaryGreaterThan(double sal);

The above method generates a query like this;

SELECT \* FROM EMP WHERE SAL > ?;

example4:

List<Employee> findByEnameContaining(String namePart);

The above method generates a query like this;

SELECT \* FROM EMP WHERE ENAME LIKE ‘%?%’;

example5:

List<Employee> findTop5BySalaryGreaterThan(double sal);

The above method generates a query like this;

SELECT \* FROM EMP WHERE SAL > ? LIMIT 5;