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.

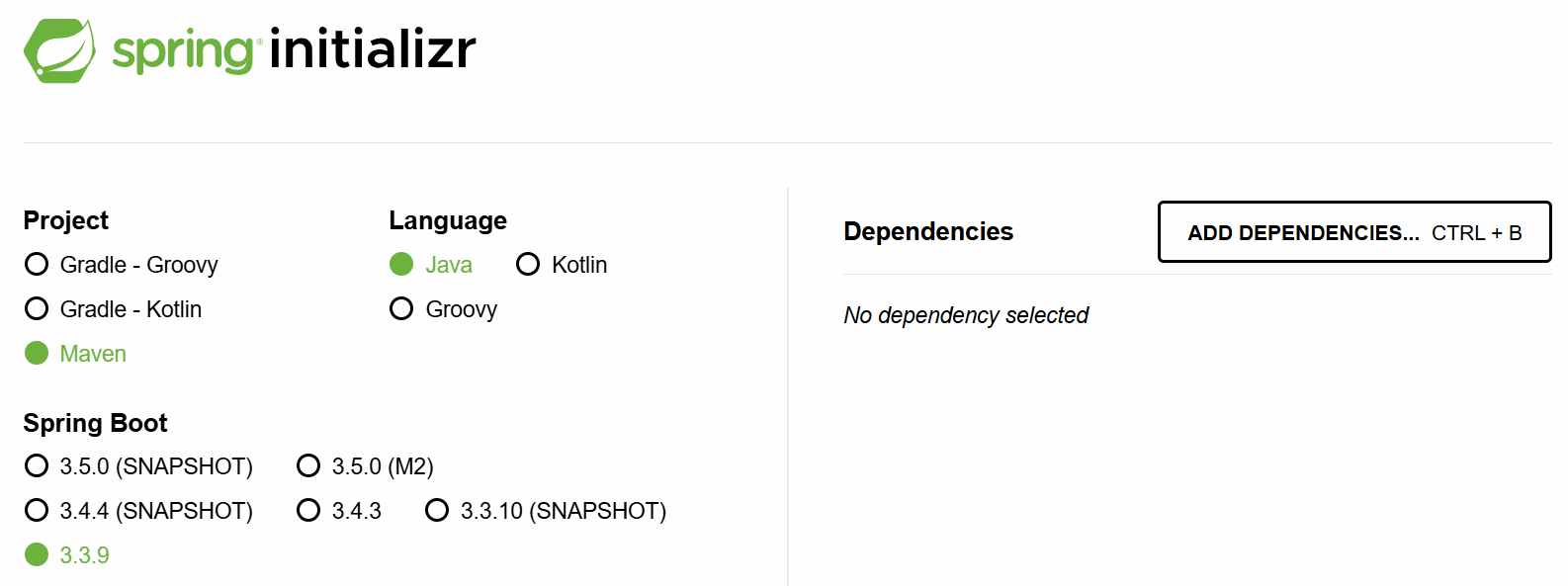
It means, like we can get the health checks, metrics, configurations, runtime details of the currently executing application.

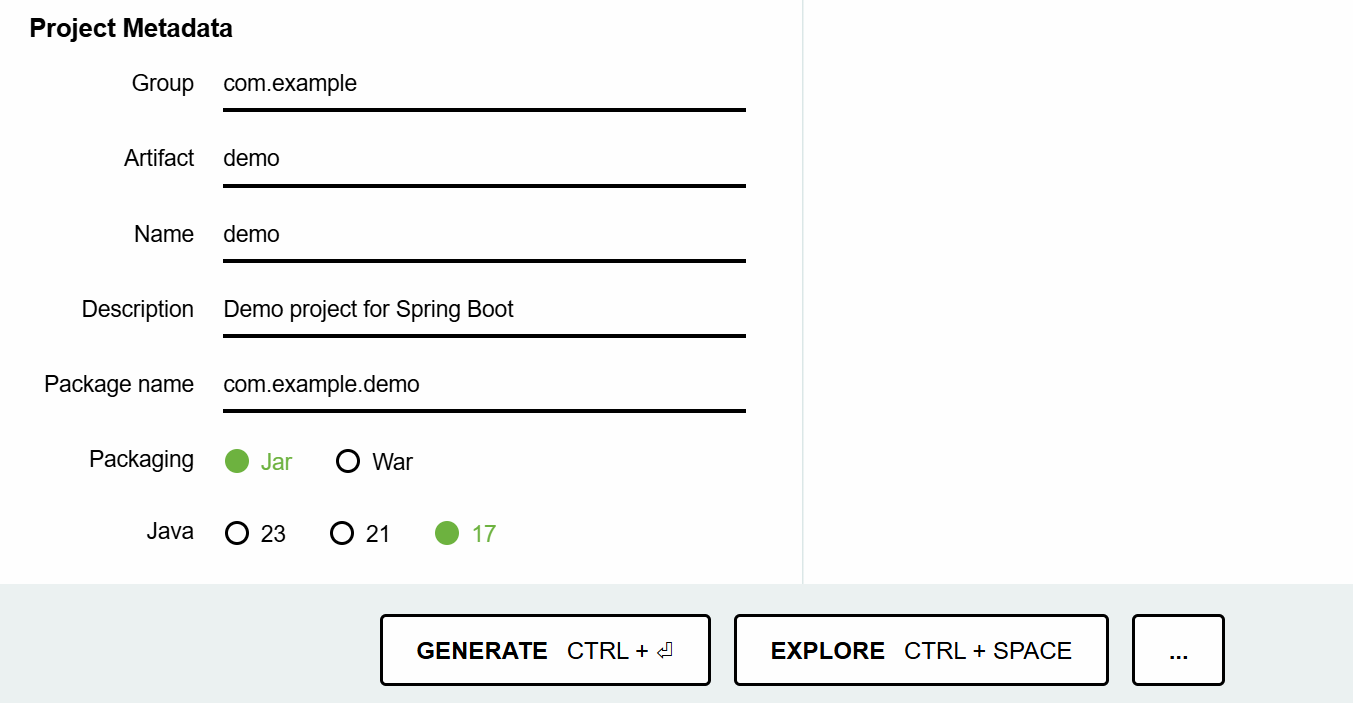
1. Spring Boot provides **JUnit, Mockito, and TestContainers** integrations.

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>

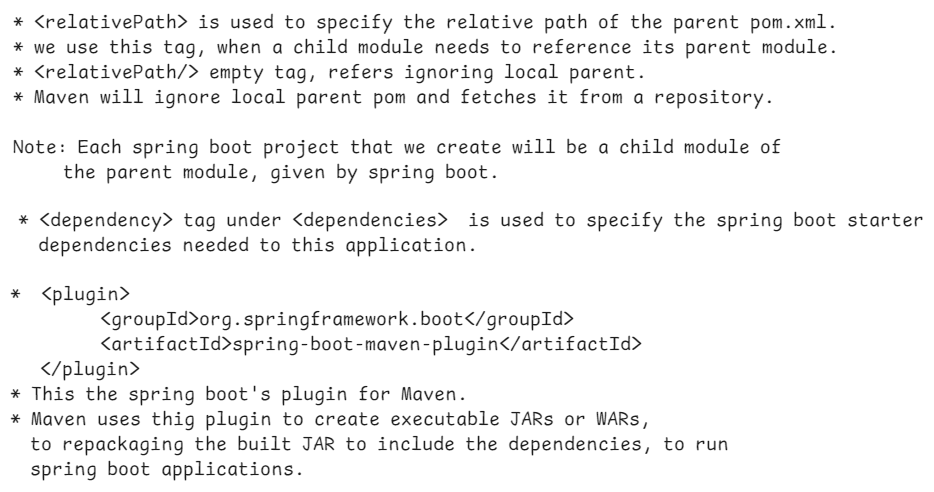
<scope>test</scope>

</dependency>

</dependencies>

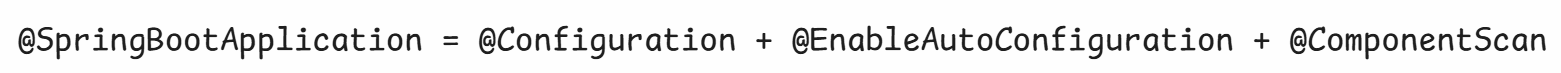
what are the dependency scopes in maven?

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.

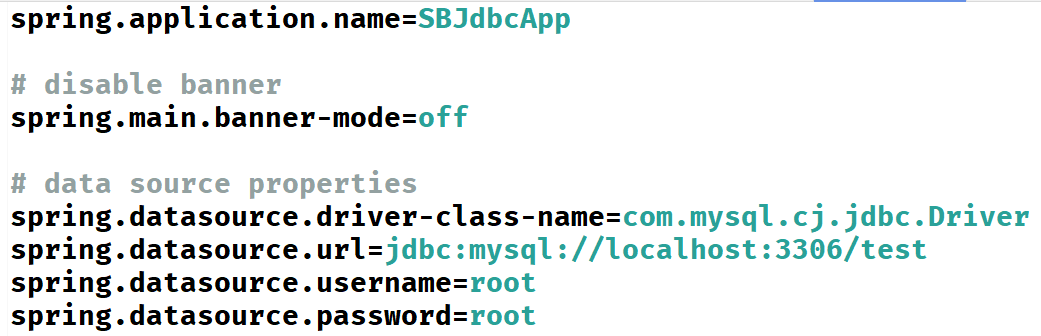
* You need to add JDBC API and MySQL driver dependencies while creating Spring Starter project in STS.
* Note: Before executing the below application, run the below commands from MySQL Workbench.

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 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 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();**

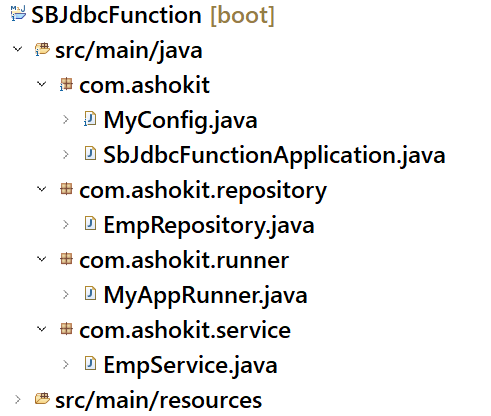
**System.*out*.println("\*+\*+".repeat(20));**

**}**

**}**

**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 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);**

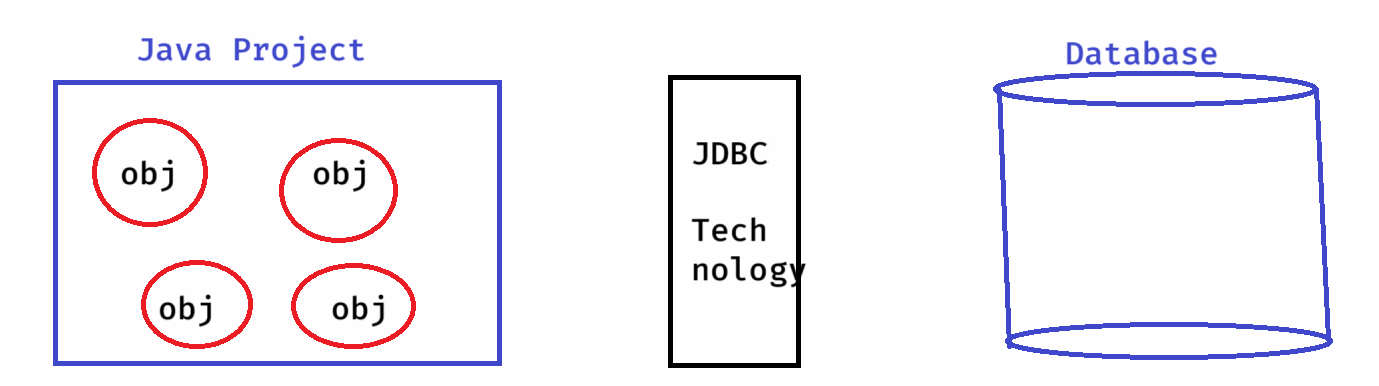
**}**

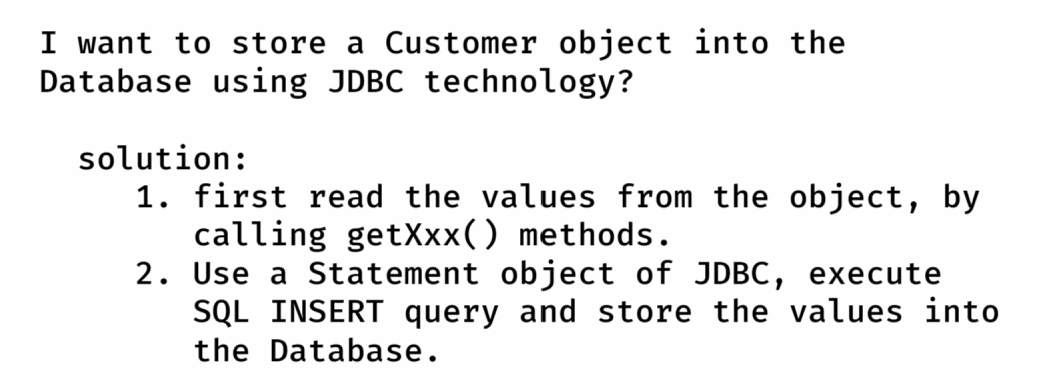
**}**

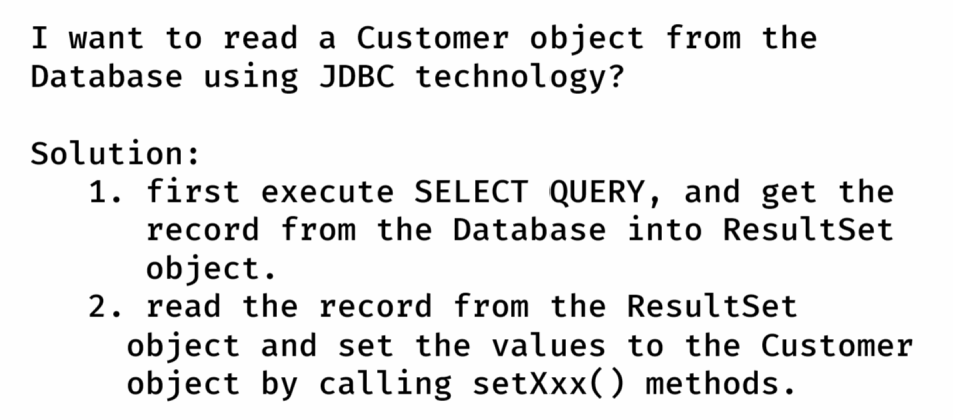
Spring Boot Data JPA

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.

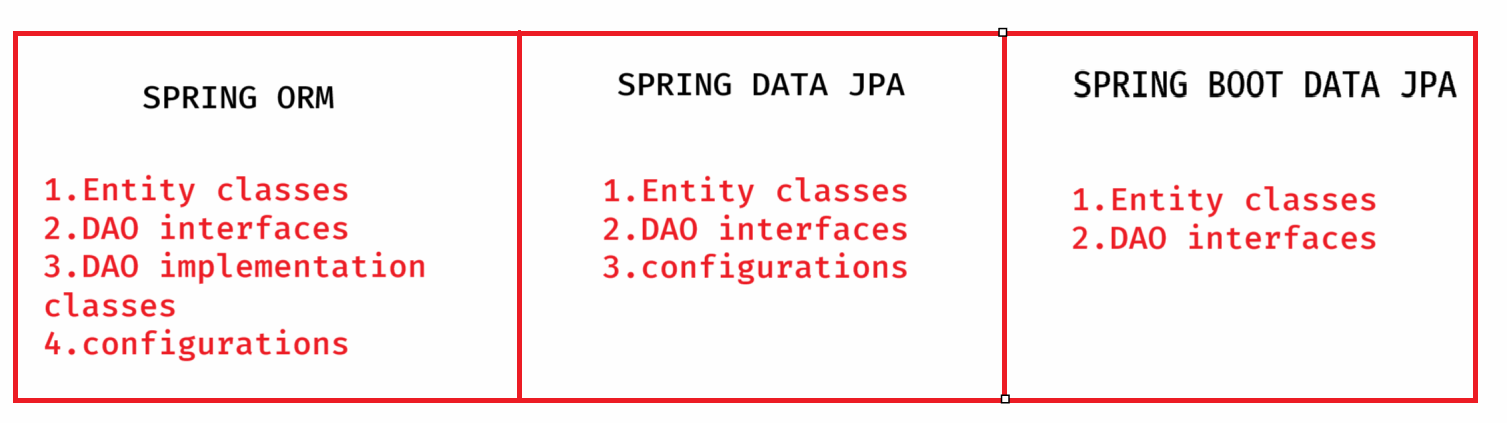
WHY SPRING BOOT DATA JPA?

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

* 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.

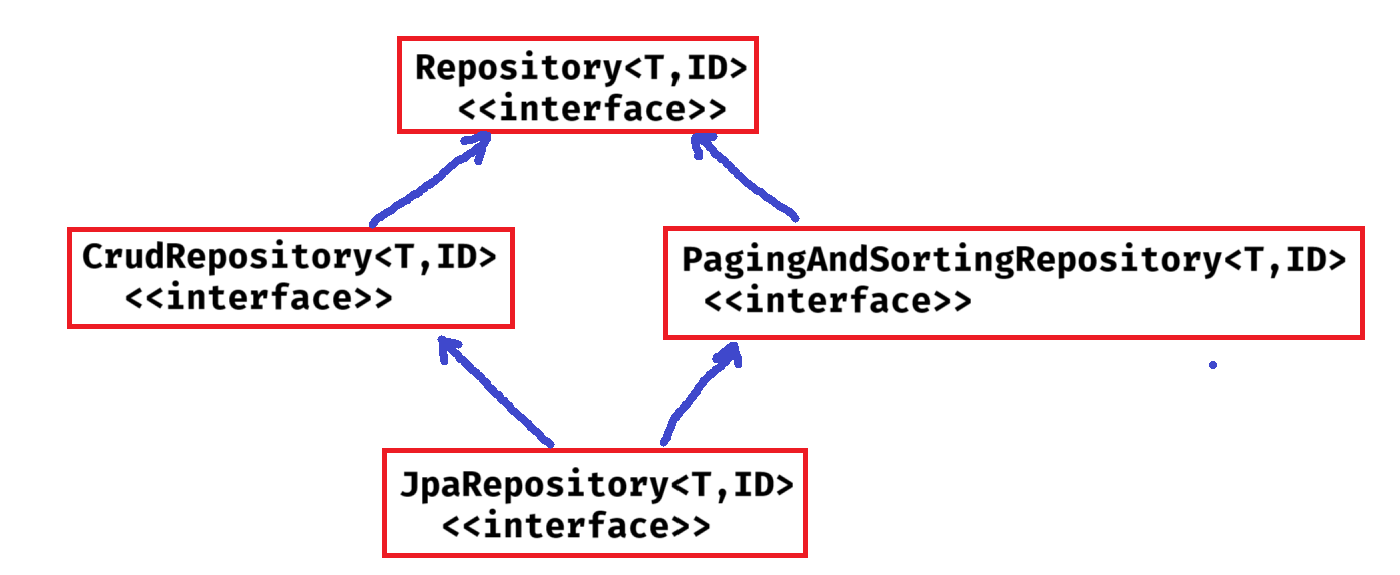
(C:\Users\WINDOWS\.m2\repository\org\projectlombok\lombok\1.18.34)

* 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**

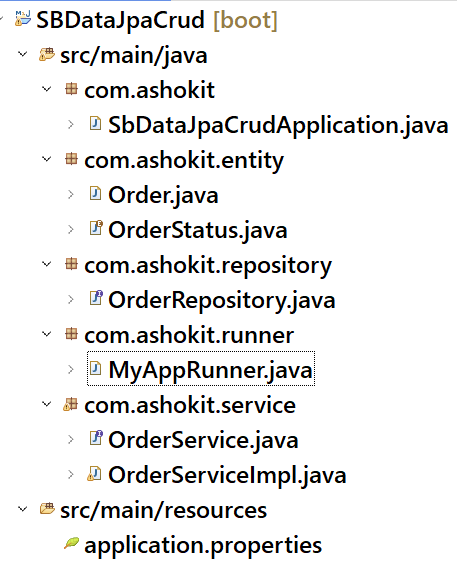
**spring.jpa.hibernate.ddl-auto=update**

* 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)

**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 **JPA** **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;



**Download the application from the below link:**

[**https://github.com/ShekherJava/SBDataJpaQueryApproach1**](https://github.com/ShekherJava/SBDataJpaQueryApproach1)

**@Query annotation**:

* **@Query annotation can be used to attach a query to the method declaration.**
* **Here, we can add a custom method declaration to the repository interface and also we can attach a query to that method.**
* **Here, the method declaration is no need to follow any structure.**
* **This annotation in Spring Data JPA allows developers to define custom queries using JPQL or in SQL.**
* **JPQL – Jakarta Persistence Query Language**
* **SQL – Structured Query Language**
* **JPQL queries looks like SQL queries. But in place of column names, we have to write property names and in place of table name, we have to write entity class reference.**
* **If we write a JPQL query, it will be translated to a tuned SQL query and then that query will run on the database.**
* **JPQL queries are database indendent and they improve the application performance.**

**example1:**

**sql : SELECT \* FROM EMP**

**jpql: SELECT e FROM EmployeeEntity e**

**example2:**

**sql : SELECT EMPNO, SAL FROM EMP**

**jpql: SELECT e.empNumber, e.empSal FROM EmployeeEntity e**

**example3:**

**sql : SELECT \* FROM EMP WHERE SAL > ?**

**jpql: SELECT e FROM EmployeeEntity e WHERE**

**e.empSal > ?1**

**(or)**

**SELECT e FROM EmployeeEntity e WHERE**

**e.empSal > :salary**

**example4:**

**sql : SELECT \* FROM EMP WHERE ENAME LIKE ‘%s%’ AND**

**SAL > ?**

**jpql: SELECT e FROM EmployeeEntity e WHERE**

**e.empName LIKE ‘%s%’ AND e.empSal > ?1**

****

**Download the application from the below link.**

[**https://github.com/ShekherJava/SBDataJpaQueryApproach2.git**](https://github.com/ShekherJava/SBDataJpaQueryApproach2.git)

**==================================================**

**@NamedQuery:**

**This annotation can be used to define a named query that can be reused across mutliple places in our application.**

**we define a named query at entity level and it is referred in the repository, with its name.**

**Named query improves the query readability as the query and entity class used in the query are at one place.**

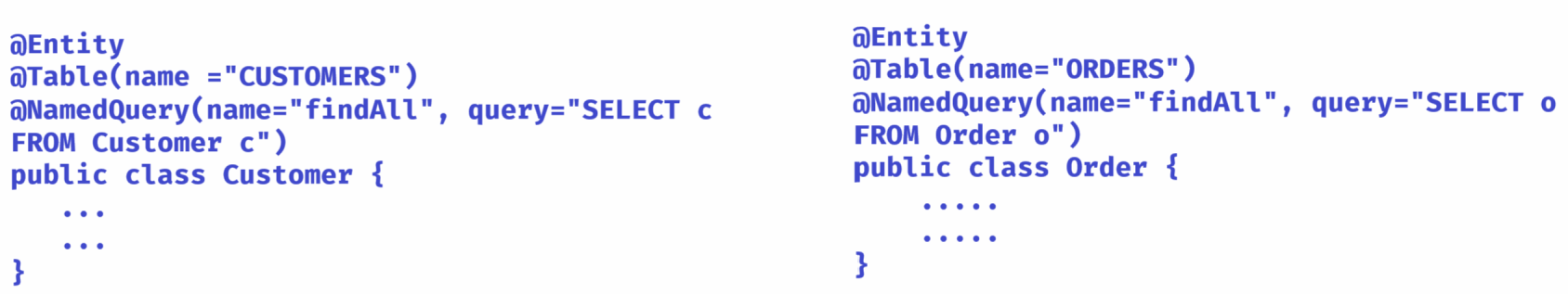
**Without rewriting a query, it can be used at different parts of the application.**

**Named queries are precompiled by the JPA provider. So, it improves performance.**

**Like @NamedQuery, we also have @NamedNativeQuery to configure native SQL query at entity level.**

* **we can define multiple @NamedQuery’s or @NamedNativeQuery’s at entity class level.**
* **The naming convention to be followed to provide a name to the query is “entityclassname.methodname”**
* **JPA will store the named queries in PeristenceContext object, with their names. If multiple queries are configured with the same name, then one query will override another query.**
* **So, it is recommended to name the query with naming convention.**

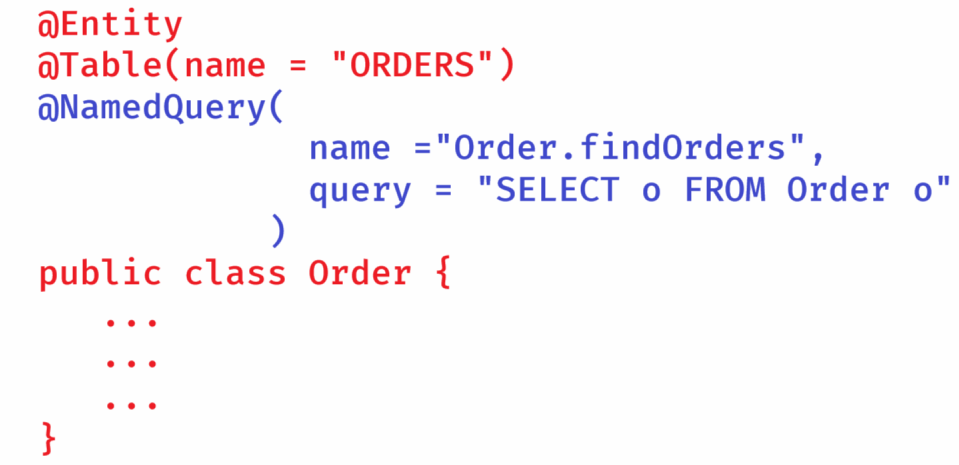
**For ex:**

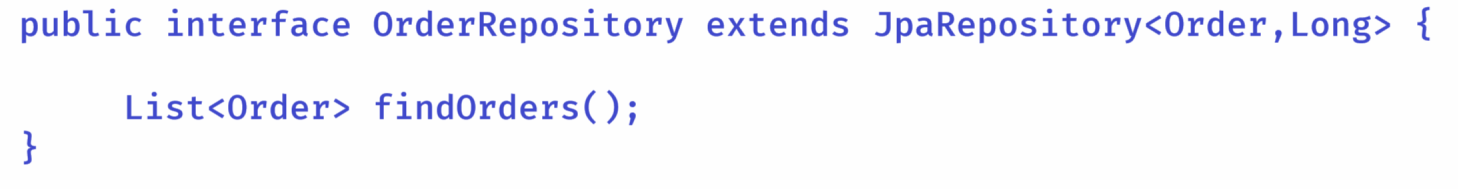
****

****

**Here, the two queries have the same name. So, one query will override another.**

**ex:**

****

****

**@Query annotation can also be used to perform insert/delete/update operations.**

**In this case, to mark that this query method is performing modifying operation, we should add @Modifying annotation.**

**JPA provider/Hibernate can execute INSERT/DELETE/UPDATE operations with in a Database transaction only.**

**So, we should mark that method as transactional method with @Transactional annotation.**

**ex:**

**@Query( value = "DELETE FROM Order o where o.orderStatus = ?1")**

**@Modifying**

**@Transactional**

**void deleteOrdersByStatus(OrderStatus orderStatus);**

**Identifier Generator strategies of JPA:**

* **Identifier generator strategies are used for generating the primary key values for the entity class objects automatically.**
* **JPA has provided 4 pre-defined identifier generation strategies.**

1. **GenerationType.IDENTITY**
2. **GenerationType.SEQUENCE**
3. **GenerationType.TABLE**
4. **GenerationType.AUTO**

**Note: GenerationType is an enum with 4 constants.**

* **To attach a primary key generation strategy, we have to use @GeneratedValue annotation.**

**For ex:**

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

**@Column(name = “ORDER\_ID”)**

**private Long orderId;**

**GenerationType.IDENTITY:**

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

 **JPA uses an auto-increment column in the database.**

* **The database generates the primary key upon INSERT, and JPA fetches it afterward.**
* **It is Simple and efficient when using databases like MySQL, PostgreSQL, or SQL Server.**

**GenerationType.SEQUENCE:**

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

 **Uses a database sequence to generate primary keys.**

* **Requires defining a sequence in the database.**
* **The sequence is an independent database object that maintains a counter.**
* **A sequence can be created in the database with the below command.**

**CREATE SEQUENCE order\_seq START WITH 1 INCREMENT BY 1;**

**This strategy works like below.**

1. **JPA queries the sequence**
2. **The database returns the next available ID.**
3. **The new record is inserted into the ORDERS table with that ID.**

**ex:**

**@Id**

**@SequenceGenerator(name = "seq1", sequenceName = "order\_seq", allocationSize = 1)**

**@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq1")**

**private Long id;**

**GenerationType.TABLE:**

* **Uses a separate table to store and generate unique primary key values.**
* **The table maintains a counter, similar to sequences, but is implemented at the application level.**
* **Works with databases that don’t support sequences or auto increment.**
* **To use this strategy, first we need to create a id generator table in the database and insert one record like below.**

**CREATE TABLE id\_generator (**

**gen\_name VARCHAR(50) PRIMARY KEY,**

**gen\_value LONG**

**);**

**INSERT INTO id\_generator (gen\_name, gen\_value) VALUES ('ord\_id', 0);**

* **This strategy can be used with primary key like below.**

**@Id**

**@GeneratedValue(strategy = GenerationType.TABLE, generator = "gen")**

**@TableGenerator(**

**name = "gen",**

**table = "id\_generator",**

***pkColumnName = "gen\_name",***

***valueColumnName = "gen\_value",***

**pkColumnValue = "ord\_id",**

**allocationSize = 1**

**)**

**private Long id;**

* **This strategy works like below.**

1. **JPA selects the pkColumn value from the id\_generator table.**
2. **JPA increments its value by 1 and updates the value to the id\_generator table.**
3. **JPA will insert the record into ORDERS table with that id.**

**GenerationType.AUTO:**

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

* **JPA delegates key generation to the underlying database.**
* **The database picks the best strategy (SEQUENCE, IDENTITY, or TABLE) based on the database dialect.**

**ex:**

**@Id**

**@GeneratedValue(strategy = GenerationType.AUTO)**

**private Long id;**

**Download the example application from the below link.**

https://github.com/ShekherJava/SBDataJpaIdGenerator.git

**Custom ID Generator:**

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

* **If the pre-defined id generators of JPA are not suitable for your application’s requirement then you can create a custom ID Generator and you can use it to generate the primary key values for the entity class objects.**
* **The below are the steps to use a Custom ID Generator in application.**
* **1. create a class by implementing IdentifierGenerator interface, define the custom logic.**
* **2. Use @GenericGenerator to integrate it with JPA.**
* **3. Use @GeneratedValue, to add this custom strategy to the primary key.**

**Download the application from the below link:**

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

**JPA Relationships**

* **JPA relationships are used to model the real-world mappings between entity classes.**
* **These relationships are also called as associations.**
* **We can able to provide the connections between the entity classes with the following relationships.**

1. **one-to-many**
2. **many-to-one**
3. **many-to-many**
4. **one-to-one**

**one-to-many:**

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

* **one-to-many relationship can be provided to navigate from a parent entity to its child entities.**
* **For example, we can provide one-to-many mapping from customer to his orders. So that, we can navigate from a customer to see his orders.**
* **To provide one-to-many relationship, the parent class should have a collection reference, to hold the child class objects.**
* **For example, in Customer class, you need a List<Order> to hold the orders of a customer.**

**ex:**

**public class Customer {**

**@OneToMany**

**List<Order> orders = new ArrayList<>();**

**}**

* **The parameters of @OneToMany are,**

1. **cascade**
2. **fetch**

* **T**he cascade attribute in the @OneToMany annotation is used to define which operations should be cascaded from the parent entity to the child entities. That means, if an operation (like persist, merge, remove, etc.) is performed on the parent, it will also be performed on its associated child entities.
* JPA provides several cascade types through the CascadeType enum.

|  |  |
| --- | --- |
| * CascadeType.PERSIST | : If you persist the parent, the children are persisted too. |

|  |  |
| --- | --- |
| * CascadeType.MERGE | : If you merge (update) the parent, the children are merged too. |

|  |  |
| --- | --- |
| * CascadeType.REMOVE | : If you remove the parent, the children are removed too. |

|  |  |
| --- | --- |
| * CascadeType.REFRESH | : If you refresh the parent, the children are refreshed too. |

|  |  |
| --- | --- |
| * CascadeType.DETACH | : If you detach the parent, the children are detached too. |

|  |  |
| --- | --- |
| * CascadeType.ALL | :Applies all of the above cascade operations. |
| * The fetch attibute of @OneToMany tells the JPA that it Should load the related entity **immediately** when the parent is loaded? Or should I wait until it's accessed?   **FetchType.EAGER**   * Loads the related entity **immediately**, along with the parent. * Uses a JOIN behind the scenes.   **FetchType.LAZY**   * Loads the related entity **only when accessed** for the first time. * Improves performance, avoids unnecessary data.   Download the application from the below link.  <https://github.com/ShekherJava/SBDataJpaOneToMany.git>  one-to-many Bi direction:   * One-to-many bi direction is a combination of one-to-many along with many-to-one from the other side. * With bi-direction, you can navigate from parent to child objects and also from a child to its parent object. * To connect a child object with its parent object, in child entity class, you should create a reference to parent class, and mark it as a many-to-one field. * The joincolumn is specified with @ManyToOne, and mappedBy attribute is used with @OneToMany. * The mappedBy attribute in the @OneToMany annotation is used to specify the **inverse side** of a bidirectional relationship. It tells JPA that this entity does not own the relationship and that the foreign key mapping is handled by the **other entity**. * The field inside mappedBy must match the **variable name** in the owning entity. * The **owning side** (with @ManyToOne) manages the relationship, while the **inverse side** (with @OneToMany(mappedBy)) just reads it.   Download the example code from the below link:  <https://github.com/ShekherJava/SBDataJpaOneToManyBiDirection.git>  Many-to-Many relationship:  A many-to-many relationship is a type of association between two entities where:   1. One record in the first table can be associated with many records in the second table, and 2. One record in the second table can be associated with many records in the first table.   ex:  A student can enroll in many courses.  A course can have many students enrolled.  Relational databases don’t support many-to-many relationships directly. Instead, you use a third (junction or join) table to break it into two one-to-many relationships.  This junction table holds the foreign keys of the relationship.  The JPA annotations for Many to Many relationship are,  @ManyToMany  @JoinTable  You can define Many to Many as uni-directional or bi-directional.  Suppose, if you want to navigate from student to courses, but not from vice-versa then you can define uni-directional relationship.  To navigate from both sides, define bi-directional relationship.  Download the example from the below link:  <https://github.com/ShekherJava/SBDataJpaManyToMany.git>  @Embeddable annotation:  In JPA (Java Persistence API), an @Embeddable class is a reusable piece of data that you can embed inside entities, rather than treat as a separate table.  It does not have its own table or primary key.  It's stored as part of the owning entity's table.  @Embeddable Marks a class as embeddable inside entities  @Embedded Tells JPA to include the embeddable in this entity  No @Id Embeddables don’t have primary keys.  Download the application from the below link:  <https://github.com/ShekherJava/SBDataJpaEmbeddable.git>  Spring Boot MVC  Design pattern?  \*\* A design pattern is a document, which contains the context, the recursive problem occurred and the best solution to resolve the problem.  \*\* So, we can define a desing patern as a best solution provided to solve a recursive problem.  \*\* GoF design patterns are the patterns identified on Java SE. They are divided 3 categories.   1. creational design patterns 2. structural design patterns 3. behavioral design patterns.   \*\* Apart from GOF design patterns, we also have Java EE design patterns like, IoC, MVC, DAO, DTO, Business, Delegate, Service Façade, etc..  \*\* A web application is a an application that runs on a server environment, which is accessible to the users over internet, and it knows how to receive the request from the users, and it knows how to process the requests and also it knows how to send to the response to the users.  \*\* Web applications are 2 types.   1. static web applications 2. dynamic web applications.   \*\* static web applications can be developed by using HTML, CSS, Javascript and Bootstrap.  \*\* dynamic web applications can be developed by using static web technologies and also with server-side technologies like, servlets, JSP, Spring MVC, Struts, JSF, etc..  MVC design pattern:   * Java dynamic web application development was started with Servlet technology. * In a Servlet class, a developer has to define the following logics to send a dynamic response to a client’s request. * 1. accepting the request * 2. parsing the request parameters. * 3. validations, conversion, null checks * 4. fetching the data from the database * 5. implementing business logics. * 6. Building HTML for response.   \* Later web application development in Java was started with Servlet and Java Bean together.  \* Here, Java Beans were used for fetching the data from a database and implementing business logics.  \* All the remaining tasks are need to be done by a servlet only and also it has to decide the appropriate Java Bean for the request.  \* For smaller applications it is good, but when applicaton grew up, it would be very painful.  \* So, to develop large scale web applications with clear separation of concerns, MVC design pattern was provided.  \* In MVC,  Model – Contains business logic or the data. It returns the information to the controller asked for.  View – Contains the presentaton logic for the data generated by the model.  Controller – Accepts the user request, calls the appropriate model, after receiving the info from model, then calls appropriate view, provides the model data to the view.    Advantages of MVC:   1. separation of concerns like business logic, presentation logic and the control flow. 2. You can test model without UI. 3. Different Dev teams can work on Model, View and Controller. 4. Same model can be reused for UI clients and API clients.   \* For developing Enterprise level web applications using MVC frameworks like Spring, MVC is combined with another pattern called Front Controller.  \* This Front controller is the main entry gate for all the requests to go through and routes the requests to the appropriate controllers and also provides the system level services like logging, security and other pre-processing operations.   * In Spring MVC, this Front controller is a pre-defined servlet class called DispatcherServlet.      |  |  | | --- | --- |  |  |  |  | | --- | --- | --- | | 1️⃣ | **DispatcherServlet** | Receives the HTTP request |  |  |  |  | | --- | --- | --- | | 2️⃣ | **HandlerMapping** | Finds which controller handles the request |  |  |  |  | | --- | --- | --- | | 3️⃣ | **Controller** | Runs the logic, prepares data, returns view name |  |  |  |  | | --- | --- | --- | | 4️⃣ | **ModelAndView** | Combines data (Model) + View name |  |  |  |  | | --- | --- | --- | | 5️⃣ | **ViewResolver** | Converts view name like "home" to home.jsp or home.html |  |  |  |  | | --- | --- | --- | | 6️⃣ | **View** (JSP/Thymeleaf) | Final HTML is generated using the model data |  |  |  |  | | --- | --- | --- | | 7️⃣ | **DispatcherServlet** | Sends the rendered result to the client |   **Defining the controller:**  **-----------------------**   * **The two primary annotations for defining a controller component are,** * **@Controller** * **@RequestMapping** * **The controller component contains handler methods, where each handler method is mapped to a request path, to handle that request.** * **@Controller is a stereo type annotation, which can be used at class-level.** * **@RequestMapping can be used at class-level and also at method-level.** * **@RequestMapping is used to map a request path and HTTP method on to a handler method.**   **ex:**  **@Controller**  **public class UserController {**    **@RequestMapping(**  **value = “/add”,**  **method = RequestMethod.POST**  **)**  **public String addUser(User user) { //handler method**  **//logic**  **}**  **@RequestMapping(**  **value = “/user/{id}”**  **method = RequestMethod.GET**  **)**  **public String getUser(Long id) { //handler method**  **//logic**  **}**  **}**  **@RequestMapping(value = “/add”, method =RequestMetho.POST) = @PostMapping(value = “/add”)**  **@RequestMapping(value = “/user/{id}”, method = RequestMethod.GET) = @GetMapping(value=”/user/{id}”)**   * **The handler methods of a controller class should contain the return type as either String or ModelAndView** * **If a handler method wants to return a logical view name to the DispatcherServlet then its return type should be String.** * **If a handler method wants to return logical view name + data to the DispatcherServlet then its return type should be ModelAndView.** * **A handler method has to share the data received from service component to the views.** * **It can use ModelAndView object or it can use Model object of spring, to share the data with the view.**   **ex1:**  **@Controller**  **public class EmployeeController {**  **@GetMapping(value = “/employees”)**  **public ModelAndView getEmployees() {**  **// calls Service component**  **// received List<Employee>**  **return new ModelAndView(“display”,**  **“employees”, list);**  **}**  **}** |  |

**ex2:**

**@Controller**

**public class EmployeeController {**

**@GetMapping(value = “/employees”)**

**public String getEmployees(Model model) {**

**//calls Service component**

**//received List<Employee>**

**model.addAttribute(“emplyoees”, list);**

**return “display”;**

**}**

**}**

**Why a handler method in Controller should return logical view name, why not the actual view?**

**ANS: 1. If a handler method returns actual view, then when the location of view is changed then we need to do the changes in the controller also.**

**2. If view technology is changed like jsp to freemarker pages then also we need to do the changes in the controller class.**

**\* So, we can say that the controller is tightly coupled with the view.**

**\* That’s why the handler method returns the logical view name but not the actual view.**

**@RequestParam:**

**.** It extracts values from the **query string** or **form fields** of an HTTP request.

**.** Imagine a form where a user types their name:

**<form action="/greet">**

**<input type="text" name="userName" />**

**<input type="submit" />**

**</form>**

**.** Now in your controller:

@GetMapping("/greet")

public String greetUser(@RequestParam String userName) {

return "Hello, " + userName;

}

Model class in spring:

* Think of it as a **container or basket** where you put data that you want to show on the webpage.
* **It p**asses data from the controller to the view.

**for example,**

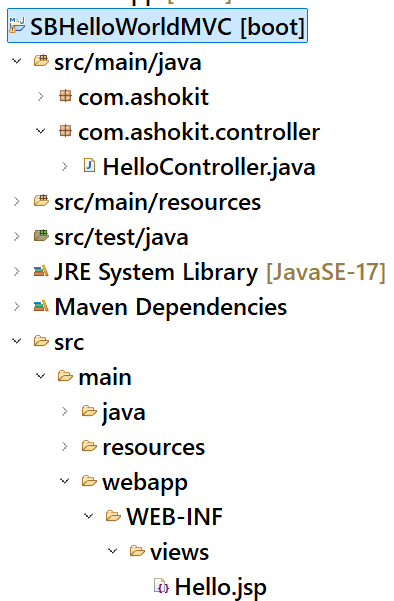
**@GetMapping("/showUser")**

**public String showUser(@RequestParam String userName, Model model) {**

**model.addAttribute("name", userName);**

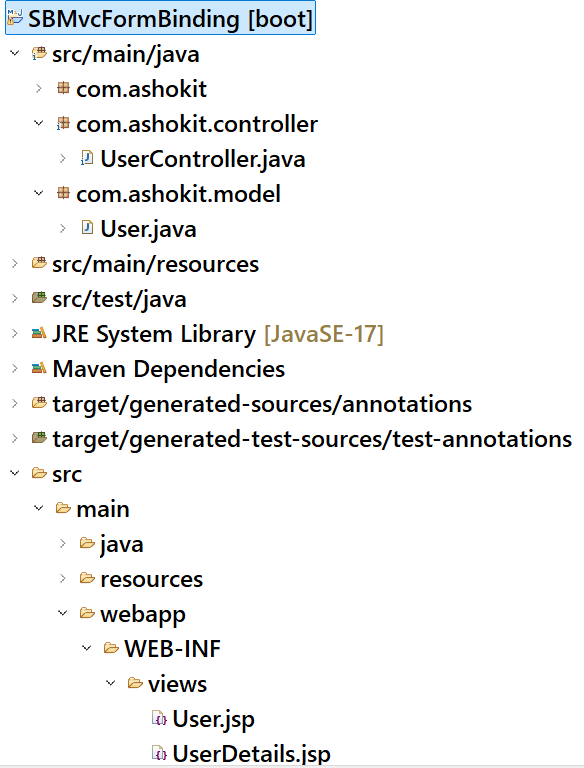
**return "welcome"; // refers to welcome.html**

**}**

****

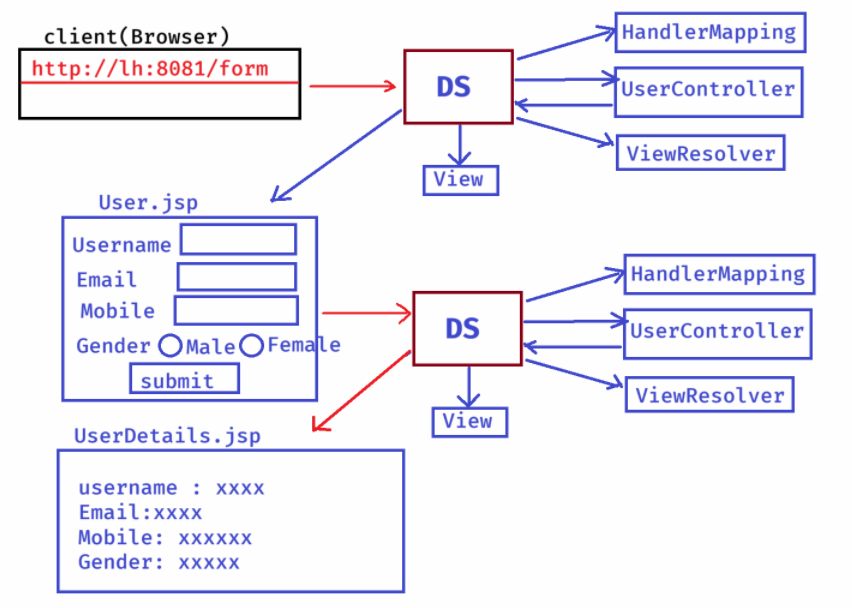
**Download the source code from the below link:**

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

****

**Download the source code from the below link:**

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

****

**Spring form tags:**

Spring provides a special set of **HTML-like tags** (inside <form:...>) to **build forms easily** and **bind form fields directly** to your Java objects ("model attributes").

You don’t have to manually match form input names to Java object fields anymore — Spring does it **automatically** with these tags!

You can also design the forms using plain HTML form tags. But you can’t preload the data from form object into the form and also you can’t display the validation error messages. So, we prefer using spring form tags.

**Bean Validation API** is a **Java standard** (part of Jakarta EE, originally Java EE) that provides a **way to validate Java objects automatically** by just adding **annotations**.

**No need** to manually write lots of if...else checks.  
**Rules are attached** directly to the Java fields.

| **Annotation** | | **Description** | |
| --- | --- | --- | --- |
| @NotNull | | The value must not be null. | |
| @NotEmpty | | The value must not be null or empty (works for Strings, Collections, etc.). | |
| @NotBlank | | The string must not be null and must contain at least one non-whitespace character. | |
| @Size(min, max) | | The size (length or number of elements) must be within the specified bounds. | |
| @Min(value) | | The numeric value must be greater than or equal to the specified minimum. | |
| @Max(value) | | The numeric value must be less than or equal to the specified maximum. | |
| @DecimalMin(value) | | The decimal number must be ≥ specified value. | |
| @DecimalMax(value) | | The decimal number must be ≤ specified value. | |
| @Positive | | Must be a positive number (greater than 0). | |
| @PositiveOrZero | | Must be 0 or a positive number. | |
| @Negative | | Must be a negative number (less than 0). | |
| @NegativeOrZero | | Must be 0 or a negative number. | |
| @Email | The string must be a valid email address. | |

|  |  |  |  |
| --- | --- | --- | --- |
| @Pattern(regexp=...) | | The string must match the specified regular expression. | |
| @Past | Date must be in the past. | |

|  |  |
| --- | --- |
| @PastOrPresent | Date must be in the past or present. |

|  |  |
| --- | --- |
| @Future | Date must be in the future. |

* For example,

public class User {

@NotBlank

private String name;

@Min(18)

@Max(99)

private int age;

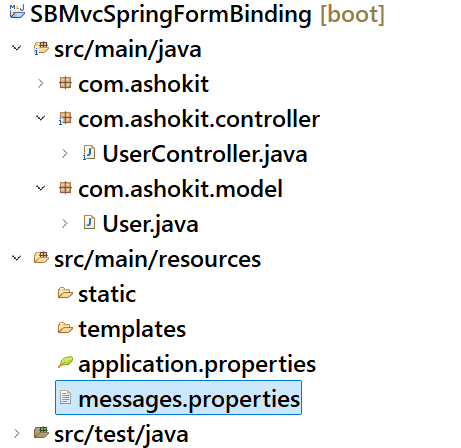
@Email

private String email;

// getters and setters

}

* In spring boot applications, to validate the form objects using above constaints/annotations, you have to add
* spring-boot-starter-validation dependency in pom.xml file.
* The error messages for each field should be defined in a messages.properties file.



Download the application from the below url:

<https://github.com/ShekherJava/SBMvcSpringFormBinding.git>

Using thymeleaf:

Thymeleaf is a **modern server-side Java template engine** used for web and standalone environments. It's especially popular in **Spring Boot** applications for rendering dynamic HTML content.

Thymeleaf allows you to create **HTML templates** that contain both **static** and **dynamic content**. It works like JSP or FreeMarker, but with better integration with **Spring MVC.**

Thymeleaf replaces special attributes with real data at runtime.

| **Attribute** | **Description** | **Example Usage** |
| --- | --- | --- |
| th:text | Replaces content inside the tag | <p th:text="${employee.name}">Name</p> |
|  |  |  |
| th:if | Conditionally renders the element if the expression is true | <div th:if="${employee.active}">Active</div> |
| th:unless | Opposite of th:if, renders only if the expression is false | <div th:unless="${employee.active}">Inactive</div> |
| th:each | Loops through a collection | <tr th:each="emp : ${employees}"> |
| th:object | Binds a form or group of fields to an object | <form th:object="${employee}"> |
| th:field | Binds form fields to object properties (with th:object) | <input th:field="\*{name}" /> |
| th:href | Dynamically sets the href attribute | <a th:href="@{/edit/{id}(id=${emp.id})}">Edit</a> |
| th:src | Dynamically sets the src attribute of images/scripts | <img th:src="@{/images/logo.png}" /> |
| th:value | Dynamically sets the value attribute of an input field | <input th:value="${employee.name}" /> |
| th:action | Sets the action URL for a form | <form th:action="@{/save}" method="post"> |

Pagination:

**Pagination** is used to split large results into smaller chunks (pages), improving performance and usability.

Spring Data JPA makes pagination **very easy** using the PagingAndSortingRepository or simply JpaRepository.

In **Spring Data JPA**, Pageable and Page are two key interfaces that support **pagination.**

**Pageable is an interface used to specify:**

* **Page number**
* **Page size (number of items per page)**
* **Sorting order (optional)**

You can create an instance using PageRequest.of(...):

Pageable pageable = PageRequest.of(0, 5); // Page 0, 5 items per page

* **Page<T> is a result container that holds:**
* A **list of entities** (e.g., List<Employee>)
* **Metadata** like:
  + Total elements
  + Total pages
  + Current page number
  + Whether there is a next/previous page

**Example:**

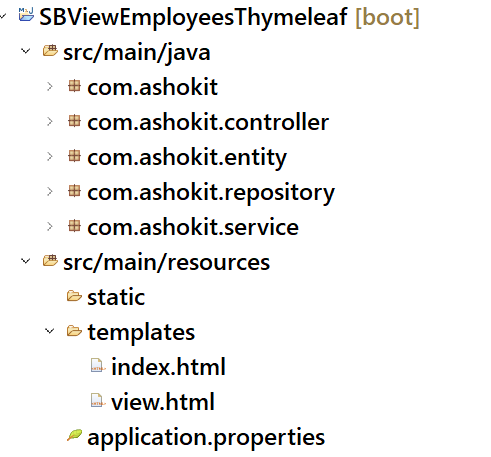
Page<Employee> page = employeeRepository.findAll(pageable);

List<Employee> employees = page.getContent(); // Data

int totalPages = page.getTotalPages(); // Total pages

long totalElements = page.getTotalElements(); // Total items

boolean hasNext = page.hasNext(); // Navigation info



Download the source code of this example from the below link.

<https://github.com/ShekherJava/SBViewEmployeesThymeleaf.git>