Spring Boot

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

* Spring Boot is not a another framework.
* Spring Boot is a new model to develop spring applications.
* With Spring Boot model, rapid application development is possible.

why spring boot?

1. In spring framework, the configuration becomes complex and hard to maintain for large projects.

With Spring Boot, the configuration will be provided by Spring Boot. So, very little or zero configuration is required.

1. In spring framework, the spring dependencies and the third-party dependencies with compatible versions needs to be added in pom.xml. If we upgrade spring dependency version, then we also need to search and update a matching third-party dependency in pom.xml. This is a huge task.

With Spring Boot, we get starter dependencies, where a starter is internally spring and third-party dependencies with compatible versions. So, dependency management is simple.

1. In spring framework, we need to explicity deploy our application(war) into a server like Tomcat. If any changes are made to the source code then again we need to deploy the application(war) into a server.

With Spring Boot, no deployment is required, because of embedded server support.

1. In spring framework, if you want to know the metrics of an application, the you have to use an external server support.

With spring boot, we have an internal module called actuator, to know the metrics.

metrics?

* metrics refers the measures to monitor the performance and health of an application.

How can we start the spring boot project?

1. we can use spring initializer tool
2. we can use IDE like STS.

To access spring initializer tool, open the browser, enter start.spring.io

choose the parameters like project as maven, language as Java, and other details, then click Generate.

A zip file is downloaded, say demo.zip.

Extract the zip file.

Now you can import the spring boot project into a workspace.

For ex, in Eclipse/STS,

File 🡪 Import 🡪 expand Maven 🡪 existing maven projects 🡪 choose the root directory(demo) 🡪 finish

In STS IDE, we have Spring starter project, to create start with a spring boot project.

what are the changes observed in pom.xml file.

1. parent tag
2. start dependencies
3. spring boot plugin

The spring-boot-starter-parent is a special starter and it provides the default configurations for a Spring Boot application.

This parent, provides a spring boot version used in a project, so that we need to specify the version manually in the dependencies.

This parent, provides the default maven plugin confiugrations like, compiler plugin, surefire plugin, jar plugin,… It reduces the amount of configuration in pom.xml.

This parent, will set the default Java version for the project.

<relativePath> tag, tells maven about where to find the parent pom.xml.

If it is <relativePath/> (empty string), then it tells maven that locate the parent pom.xml in the maven’s central repository.

The plugin spring-boot-maven-plugin, will be used by maven to create a jar/war file for the spring boot project and also to run the spring boot project.

|| DATE: 6 – 9 – 24||

@SpringBootApplication:

* This annotation declares that a class has the main method to kick start the spring boot project execution.
* This @SpringBootApplication annotation combines three annotations together.

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

* @Configuration marks that a class is a Java configuration class and in this class, if required we can create @Bean methods.
* @EnableAutoConfiguration tells the spring boot that, automatically configure all the required beans into the spring container based on the starter dependencies available at classpath.
* @ComponentScan tells the spring to scan the current package and its sub-packages, identify the spring components like @Component or @Controller or @RestController or @Service or @Repository annotations and register them as beans in the spring container.

what does SpringApplication.run() statement will do in spring boot?

* 1. It will start the Spring application context(spring container).
* 2. It will convert command line arguments if any, into spring environment properties.
* 3. It will invoke the spring boot runner components.

spring boot runners:

* These are the components/beans that we create in a spring boot application, to run a piece of code only once when spring boot application starts.
* The runner components are useful in performing one time tasks of a spring boot application.
* ex1: Loading initial data into the database at the startup of the application.
* ex2: Validating the configuration settings at the startup of the application.
* ex3: Loading the Job schedules of the application from the database at the startup of the application.

How do we create a runner component in spring boot?

* Spring Boot has provided 2 functional interfaces.

1. CommandLineRunner interface.
2. ApplicationRunner interface.

|| DATE: 9-9-24 ||

* For example, we can create a CommandLineRunner component, like below.

@Component

public class MyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String… args) throws Exception {

//code to execute at startup

}

}

First spring boot application:

1. Launch STS
2. File 🡪 Spring starter project 🡪 provide the parameters values -> next -> finish
3. create a subpackage under demo, say in.ashokit.demo.bean
4. create HelloWorld class in the bean package.

@Component

**public** **class** HelloWorld {

**public** String sayHello() {

**return** "hello";

}

}

1. create another subpackage under demo, say

in.ashokit.demo.runner

1. Create MyCommandLineRunner class in the runner package.

@Component

**public** **class** MyCommandLineRunner **implements** CommandLineRunner {

@Autowired

HelloWorld helloWorld;

@Override

**public** **void** run(String... args) **throws** Exception {

System.***out***.println( helloWorld.sayHello() );

}

}

1. We can execute the application with in IDE, just like a Java application.

* Suppose, if you want to run this spring boot application from CommandLine, then create a jar file for the application.
* Right click on the project(Application1) 🡪 RunAs --🡪 Maven build… --🡪 Goals : package

--🡪 Run

\* Refresh the project once.

\* Now expand target folder, and verify that jar file created.

\* open command prompt, place the cursor in the project directory.

\* execute the below command.

> java -jar target/Application1-0.0.1-SNAPSHOT.jar

* another way to run the application from the command prompt is, with maven command.
* mvn spring-boot:run

|| DATE: 10-9-24 ||

command line arguments:

1. option arguments
2. non-option arguments

* option arguments are like options provided to the application from the execution line, to modify the behavior of an application.
* option arguments are prefixed with a dash(-) or double dash(--) and it has key=value syntax.
* non-option arguments do not start with a dash or double dash, and it is a value provided to the application from the execution line and it usually represents a main inputs to the application.

for ex:

java -jar xxxx.jar --user=admin --pwd=admin input.csv

CommandLineRunner and ApplicationRunner:

* Both are functional interfaces, and they a single abstract method called run().
* CommandLineRunner’s run() method has String… args parameter, it is a String array and it provides the access to command line arguments as raw arguments.
* We need to write the code manually in the run() method, to segregate option and non-option arguments separately.
* ApplicationRunner’s run() method has ApplicationArguments object as parameter and it provides parsed command line arguments.
* So, we no need to write the code manually to segregate option and non-option arguments separately.

for example:

@Component

**public** **class** MyCommandLineRunner **implements** CommandLineRunner {

@Override

**public** **void** run(String... args) **throws** Exception {

System.***out***.println("The option and non-option arguments provided from execution line");

**for** ( String arg : args ) {

System.***out***.println( arg );

}

}

}

* To provide run time arguments to the application in IDE,

right click on the project 🡪 Run As 🡪 Run configurations…

* choose the project 🡪 click on search button for main type
* click on arguments tab 🡪 program arguments textbox, enter the arguments here.

For ex,

input.csv --username=admin --password=admin output.log

|| DATE: 11-9-24 ||

* If you want to execute the spring boot application from the command line, build the application into a jar file.
* Now you can run the jar file from the command prompt like below.

java -jar target/xxxx.jar input.csv out.log

--user=admin --pwd=tiger

\* If you want to execute the application using maven command, then use the below.

mvn spring-boot:run -Dspring-boot.run.arguments=” input.csv out.log --user=admin --pwd=tiger”

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

@Override

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

System.***out***.println("Non option arguments passed to the application");

args.getNonOptionArgs().forEach(System.***out***::println);

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

System.***out***.println("option arguments passed to the application");

Set<String> optionNames = args.getOptionNames();

**for**( String name : optionNames ) {

List<String> values = args.getOptionValues(name);

System.***out***.print(name + " - ");

values.forEach(str -> System.***out***.print(str+", "));

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

}

}

}

application.properties

* It is located at src/main/resources in a maven project.
* This file is used to configure various settings, to control the behaviour of a spring boot application.
* The settings like database settings, server settings, logging settings, connection pool settings, etc.. are configured in this file.
* By default, Spring boot will load the settings from application.properties file.
* we can change the file name application.properties to some custom name.
* In this case, we need to add @PropertySource annotation with that filename as parameter at spring boot application class.

for ex:

@SpringBootApplication

@PropertySource(“myapp.properties”)

public class MyApplication {

p s v m(String[] args) {

…..

}

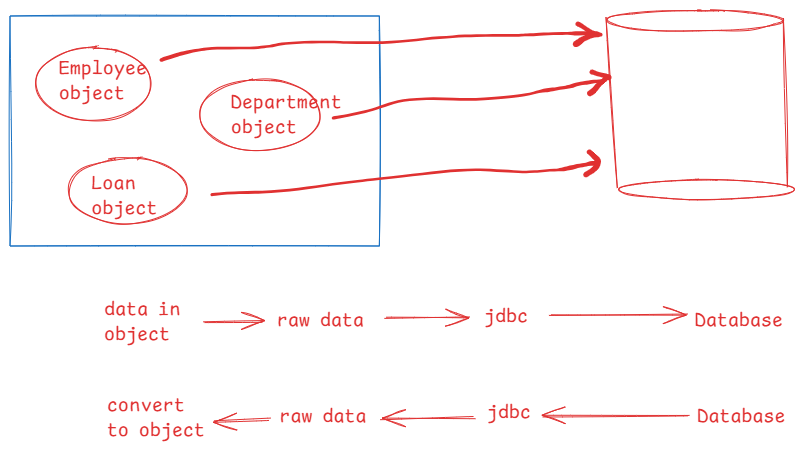
}

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

Spring Boot JPA

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

ORM(Object Relational Mapping):



|| DATE: 13-9-24 ||

* JDBC can only execute SQL queries with values(raw data) on a database. But it can’t perform operations directly on Objects.
* So, we got ORM technology, where it allows a Java application to send/receive the Java objects from the database.
* ORM tools, will fill the gaps between object model and the relational model and makes a Java application and a Database can work together.

JPA? (Java Persistence API)

* Different vendors have followed different approaches and provided ORM frameworks, to make Object model and Relational model to work together.
* When a developer is moving from one framework to another, it made difficult for the developer, as it followed a different approach.
* So, to provide a common way, and to allow the developers to easily migrate from one ORM framework to another, Sun Microsystems has provided JPA specification.
* The ORM frameworks like Hibernate, Eclipse Link, ORMLite, OpenJPA, etc.. have provided JPA implementation. So, we call them as JPA vendors.
* By creating repository classes/DAO classes with JPA specification, we can change the ORM framework through project configuration, by without doing the changes to source code.
* It means, our DAO classes are loosely coupled with the ORM framework.

Spring Data JPA?

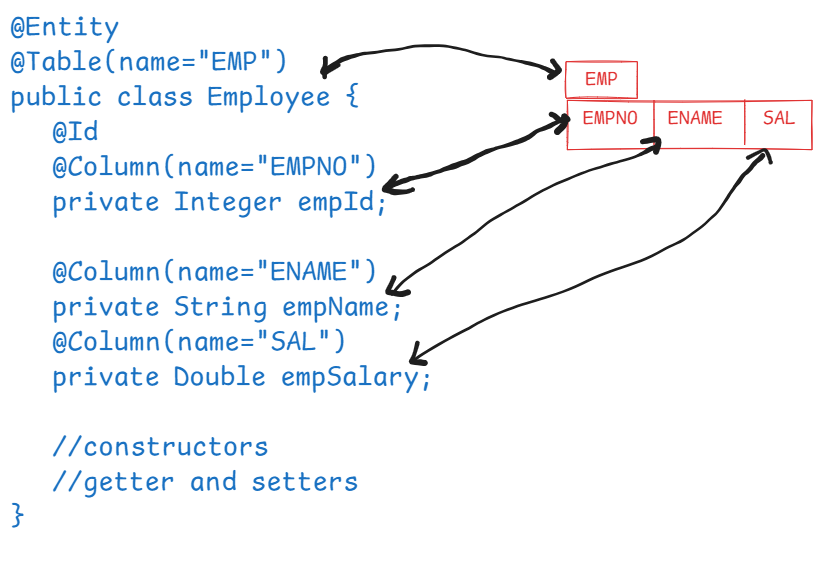
* In a repository layer, the repository classes/dao classes, performs the common operations like CRUD operations, pagination and sorting operations.
* So, Spring Data JPA, has removed writing the dao implementation classes from the repository layer.
* We write only repository interfaces and Spring will create implementation classes at runtime.
* Along with repository interfaces, we also need to define some JPA related configurations into the spring configuration and also we should add the dependencies like spring and jpa dependencies in pom.xml file.
* The spring and the JPA related dependencies must be compatible versions.

Spring Boot Data JPA?

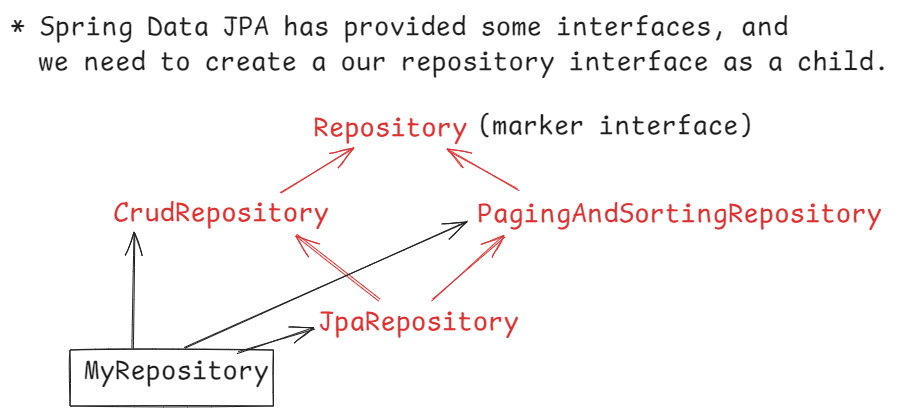
* Spring Boot Data JPA avoid writing the JPA related configurations and adding multiple compatible version of dependencies in the pom.xml file.
* Just we need to write only repository interface and Spring Boot will do the rest.
* we need to add the spring-boot-starter-data-jpa and a driver dependency.

Entity class:

* It is a Java class which is mapped with a database table.



creating a repository interface:



CrudRepository operations:

1. save(entity) : performs save/update
2. findById(id) : retrieves an entity by id
3. findAll(): retrieves all entities
4. deleteById(id): deletes an entity by id
5. existsById(id): checks for an entity by id.

etc…

PagingAndSorting operations:

1. findAll(Pageable p) : for pagenation
2. findAll(Sort sort): for sorting

JpaRepository operations:

1. saveAndFlush(entity)
2. getReferenceById(id)
3. deleteAllInBatch()
4. findAll(Example e),

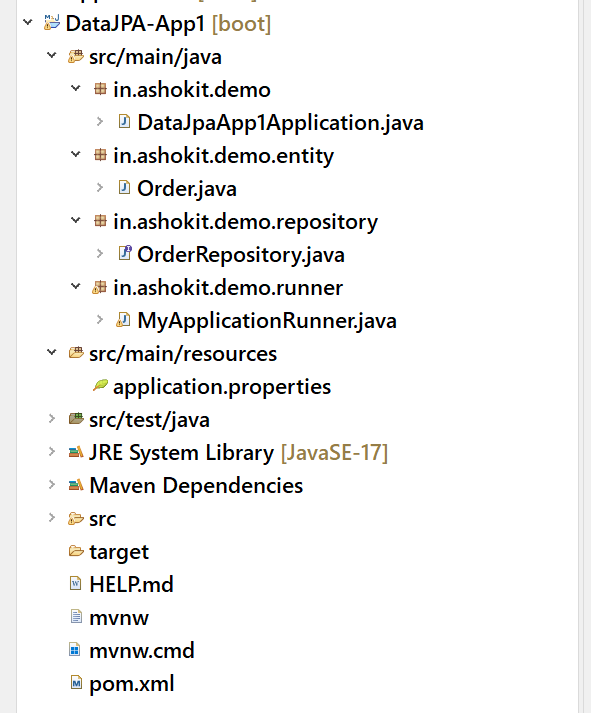
etc..

@Repository //optional

public interface EmpRepository extends JpaRepository<Employee, Integer> {

}

* To write a first spring boot data jpa application, we need 3 files.
* 1. entity class
* 2. repository interface
* 3. application.properties file
* 4. a runner component.



DataJpaApp1Application.java

@SpringBootApplication

**public** **class** DataJpaApp1Application {

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

SpringApplication.*run*(DataJpaApp1Application.**class**, args);

}

}

Order.java

@Entity

@Table(name="ORDERS")

**public** **class** Order {

@Id

**private** String id;

**private** LocalDate orderDate;

**private** String status;

**public** Order() {

}

**public** Order(String id, LocalDate orderDate, String status) {

**super**();

**this**.id = id;

**this**.orderDate = orderDate;

**this**.status = status;

}

**public** String getId() {

**return** id;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** LocalDate getOrderDate() {

**return** orderDate;

}

**public** **void** setOrderDate(LocalDate orderDate) {

**this**.orderDate = orderDate;

}

**public** String getStatus() {

**return** status;

}

**public** **void** setStatus(String status) {

**this**.status = status;

}

@Override

**public** String toString() {

**return** "Order [id=" + id + ", orderDate=" + orderDate + ", status=" + status + "]";

}

}

OrderRepository.java

**public** **interface** OrderRepository **extends** CrudRepository<Order, String>, PagingAndSortingRepository<Order, String>{

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

@Autowired

OrderRepository orderRepo;

@Override

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

/\*

Order order1 = new Order("orderid#288701", LocalDate.of(2024, 9, 14), "Out for delivery");

Order order2 = new Order("orderid#198772", LocalDate.of(2024, 9, 11), "Delivered");

Order order3 = new Order("orderid#652093", LocalDate.of(2024, 9, 12), "Shipped");

Order order4 = new Order("orderid#300876", LocalDate.of(2024, 9, 10), "Out for delivery");

orderRepo.saveAll(Arrays.asList(order1, order2, order3, order4));

\*/

/\*

\* findById() : retrieves an entity from DB, by id

\* : returns Optional class object

\* : Optional object is a container object and

\* : it may or may not contain non null value.

\* : isPresent(): returns true, if value is present, otherwise false.

\* : get(): returns the value

\*/

/\*

Optional<Order> opt = orderRepo.findById("orderid#101001");

if(opt.isPresent()) {

Order order = opt.get();

System.out.println(order.getId() + " " + order.getOrderDate() + " " + order.getStatus());

}

orderRepo.deleteById("orderid#288701");

\*/

/\*

\* findAll(Sort sort): Sort is a class and we can create

\* an object by calling a static factory

\* method, by();

\*/

Sort sort = Sort.*by*("status");

orderRepo.findAll(sort).forEach(System.***out***::println);

}

}

application.properties

spring.application.name=DataJPA-App1

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

#logs SQL queries on console

spring.jpa.show-sql=true

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

spring.jpa.hibernate.ddl-auto=update

Query approaches:

* when JpaRepository is extended, we will two query operations.

1. findById() : returns a specific entity based on id
2. findAll(): returns all entities.

* If you need more query operations like return the orders based on status or based on order date, then you have to use query approach.
* there are 3 query approaches.

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

* query creation by method name means, based on the method added to the repository interface, automatically a query is generated and executed to return the result.
* The method name should follow a specific pattern.

findBy or getBy+Property Name+Condition(optional)

ex1: find the orders based on status.

List<Order> findByStatus(String status);

ex2: find the orders based on orderdate and status

List<Order> findByOrderDateAndStatus(LocalDate orderDate, String status);

ex3: find the orders based on orderdate greater than a specifi date

List<Order> findByOrderDateGreaterThan(LocalDate orderDate);



Order.java

@Entity

@Table(name="ORDERS")

**public** **class** Order {

@Id

**private** String id;

**private** LocalDate orderDate;

**private** String status;

**public** Order() {

}

**public** Order(String id, LocalDate orderDate, String status) {

**super**();

**this**.id = id;

**this**.orderDate = orderDate;

**this**.status = status;

}

**public** String getId() {

**return** id;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** LocalDate getOrderDate() {

**return** orderDate;

}

**public** **void** setOrderDate(LocalDate orderDate) {

**this**.orderDate = orderDate;

}

**public** String getStatus() {

**return** status;

}

**public** **void** setStatus(String status) {

**this**.status = status;

}

@Override

**public** String toString() {

**return** "Order [id=" + id + ", orderDate=" + orderDate + ", status=" + status + "]";

}

}

OrderRepository.java

**public** **interface** OrderRepository **extends** JpaRepository<Order, String>{

List<Order> findByStatus(String status);

List<Order> findByOrderDateGreaterThan(LocalDate orderDate);

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

@Autowired

OrderRepository repo;

@Override

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

//findById

repo.findById("orderid#101001").ifPresent(System.***out***::println);

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

//findAll

repo.findAll().forEach(System.***out***::println);

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

//findByStatus

repo.findByStatus("Shipped").forEach(System.***out***::println);

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

//findByOrderDateGreaterThan

repo.findByOrderDateGreaterThan(LocalDate.*of*(2024, 9, 11)).forEach(System.***out***::println);

}

}

@Query annotation allows us to define custom queries using JPQL or SQL within our repository interface.

JPQL – Java Persistence Query Language.

JPQL is a database independent query language. When we write a JPQL query, it is translated to the SQL query based on the underlying database, then that SQL query will be executed on the database.

examples of writing JPQL queries:

ex1:

sql: select \* from orders

jpql: select o from Order o

ex2:

sql: select order\_date,status from orders

jpql: select o.orderDate, o.status from Order o

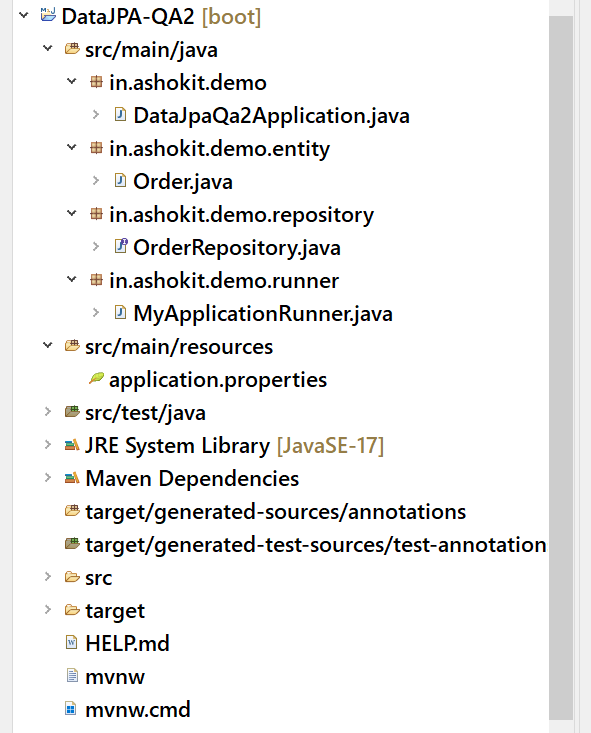
ex3:

sql: select \* from orders where id = ?

jpql: select o from Order o where o.id = ?1

or

select o from Order o where o.id = :id



Order.java

**package** in.ashokit.demo.entity;

**import** java.time.LocalDate;

**import** jakarta.persistence.Entity;

**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

**private** String id;

**private** LocalDate orderDate;

**private** String status;

}

OrderRepository.java

**package** in.ashokit.demo.repository;

**import** java.time.LocalDate;

**import** java.util.List;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** org.springframework.data.jpa.repository.Query;

**import** org.springframework.data.repository.query.Param;

**import** in.ashokit.demo.entity.Order;

**public** **interface** OrderRepository **extends** JpaRepository<Order, String>{

@Query( value = "SELECT O FROM Order O WHERE O.status = :status")

List<Order> getOrdersByStatus(@Param("status") String status);

@Query( nativeQuery = **true**, value = "select \* from orders where order\_date = ?")

List<Order> chooseOrdersByOrderDate(LocalDate orderDate);

}

MyApplicationRunner.java

**package** in.ashokit.demo.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** in.ashokit.demo.repository.OrderRepository;

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

@Autowired

OrderRepository repo;

@Override

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

repo.getOrdersByStatus("Shipped").forEach(System.***out***::println);

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

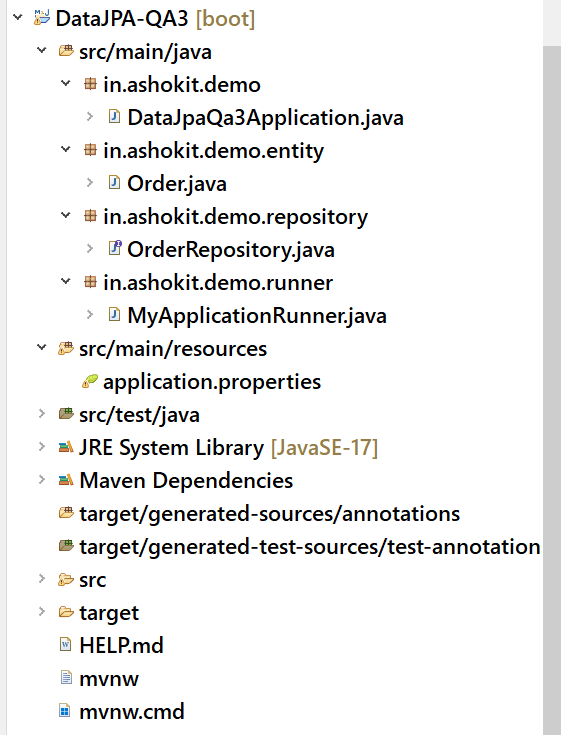
repo.chooseOrdersByOrderDate(LocalDate.*of*(2024, 9, 14)).forEach(System.***out***::println);

}

}

Named query:

* when you want to reuse a query/queries across multiple places in your application, then use named query.
* Named query is defined at the entity level and a name is given to it.
* In the repository, the query is referenced by its name.
* @NamedQuery annotation is used to define the query at the entity level.
* The name of the Named query should follow a naming convention that, entityclassname.name



Order.java

package in.ashokit.demo.entity;

import java.time.LocalDate;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.NamedQuery;

import jakarta.persistence.Table;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Entity

@Table(name="ORDERS")

@NoArgsConstructor

@AllArgsConstructor

@Data

@NamedQuery( name = "Order.chooseOrders",

query = "SELECT o from Order o ORDER BY o.status")

public class Order {

@Id

private String id;

private LocalDate orderDate;

private String status;

}

OrderRepository.java

**package** in.ashokit.demo.repository;

**import** java.util.List;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** in.ashokit.demo.entity.Order;

**public** **interface** OrderRepository **extends** JpaRepository<Order, String>{

//no need to define the query again

//just use the name of the named query

List<Order> chooseOrders();

}

MyApplicationRunner.java

**package** in.ashokit.demo.runner;

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

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

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

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

**import** in.ashokit.demo.repository.OrderRepository;

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

@Autowired

OrderRepository repo;

@Override

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

repo.chooseOrders().forEach(System.***out***::println);

}

}

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