**Spring Boot Database Support and CRUD operations**

* Basic Auto-configuration - DataSource
* Configuration Properties
* Spring Boot's JPA Support - spring-boot-starter-data-jpa
* Using Spring Boot Data - CrudRepository/JpaRepository
* Defining Queries with Spring Boot Data
* @Query annotation

**Spring Boot JPA:**

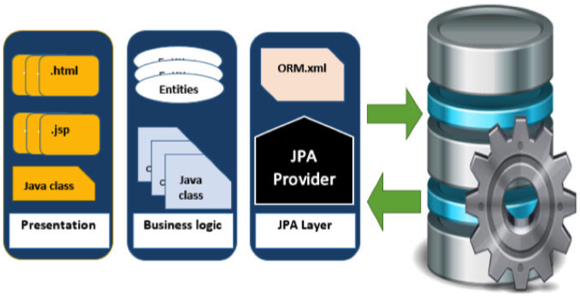
JPA is a specification which specifies how to access, manage and persist information/data between java objects and relational databases. It provides a standard approach for ORM, Object Relational Mapping. Spring Boot provides a seamless integration with JPA.

What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

Where to use JPA?

To reduce the burden of writing codes for relational object management, a programmer follows the ‘JPA Provider’ framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



JPA

JPA is a specification which specifies how to access, manage, and persist information/data between java objects and relational databases. It provides a standard approach for ORM, Object Relational Mapping. It provides EntityManager and EntityManagerFactory to perform CRUD operations.

Hibernate

Hibernate is an implementation of JPA. It provides a lightweight framework and is one of the most popular ORM tool used.

JPA Vs Hibernate

Following table summerises the differences between JPA and Hibernate.

|  |  |  |
| --- | --- | --- |
| **Category** | **JPA** | **Hibernate** |
| **Type** | JPA is a specification and defines the way to manage relational database data using java objects. | Hibernate is an implementation of JPA. It is an ORM tool to persist java objects into the relational databases. |
| **Package** | JPA uses javax.persistence package. | Hibernate uses org.hibernate package. |
| **Factory** | JPA uses EntityManagerFactory interface to get the entity manager to persist objects. | Hibernate uses SessionFactory interface to create session object which is then used to persist objects. |
| **CRUD Operations** | JPA uses EntityManager interface to create/read/delete operation and maintains the persistence context. | Hibernate uses Session interface to create/read/delete operation and maintains the persistence context. |
| **Language** | JPA uses JPQL (Java Persistence Query Language) as Object Oriented Query language for database operations. | Hibernate uses HQL (Hibernate Query Language) as Object Oriented Query language for database operations. |

**Spring Boot – CrudRepository**

Spring Boot is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time. Following are some of the features of Spring Boot:

It allows avoiding heavy configuration of XML which is present in spring

It provides easy maintenance and creation of REST endpoints

It includes embedded Tomcat-server

Deployment is very easy, war and jar files can be easily deployed in the tomcat server

For more information please refer to this article: Introduction to Spring Boot. In this article, we are going to discuss how to use CrudRepository to manage data in a Spring Boot application.

**CrudRepository**

There is an interface available in Spring Boot named as CrudRepository that contains methods for CRUD operations. It provides generic Crud operation on a repository. It is defined in the package org.springframework.data.repository and It extends the Spring Data Repository interface. If someone wants to use CrudRepository in the spring boot application he/she has to create an interface and extend the CrudRepository interface.

In order to implement Spring ORM we will have to add spring data jpa starter POM in our project. Step By Step Implementation of Data JPA project.

**Step 1.** Create a Spring Starter Project and add three starter POMs

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.microsoft.sqlserver</groupId>

<artifactId>mssql-jdbc</artifactId>

<scope>runtime</scope>

</dependency>

**Step 2:** Create a package com.controller and create a class (Entity).

package com.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

*@Entity*

public class Students {

*@Id*

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

private int student\_id;

private String student\_name;

private String student\_address;

private String student\_phone;

public int getStudent\_id() {

return student\_id;

}

public void setStudent\_id(int student\_id) {

this.student\_id = student\_id;

}

public String getStudent\_name() {

return student\_name;

}

public void setStudent\_name(String student\_name) {

this.student\_name = student\_name;

}

public String getStudent\_address() {

return student\_address;

}

public void setStudent\_address(String student\_address) {

this.student\_address = student\_address;

}

public String getStudent\_phone() {

return student\_phone;

}

public void setStudent\_phone(String student\_phone) {

this.student\_phone = student\_phone;

}

public Students() {

super();

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

}

public Students(int student\_id, String student\_name, String student\_address, String student\_phone) {

super();

this.student\_id = student\_id;

this.student\_name = student\_name;

this.student\_address = student\_address;

this.student\_phone = student\_phone; } }

**Step 3:** Create package for DAO. Add an interface StudentRepository which extends CRUDRepository

Definition for CrudRepository

public interface CrudRepository<T, ID> extends Repository<T, ID>

Where:

* T: Domain type that repository manages (Generally the Entity/Model class name)
* ID: Type of the id of the entity that repository manages (Generally the wrapper class of your @Id that is created inside the Entity/Model class)

Code for Interface:

package com.dao;

import org.springframework.data.repository.CrudRepository;

import com.model.Students;

public interface StudentRepository extends CrudRepository<Students, Integer>{

}

**Step 4:** Add database Configuration in application.properties.

server.port=9090

spring.mvc.view.prefix=/views/

spring.mvc.view.suffix=.jsp

spring.datasource.url=jdbc:sqlserver://localhost;databaseName=Citiustech;encrypt=true;trustServerCertificate=true;instanceName=SQLEXPRESS

spring.datasource.username=sa

spring.datasource.password=Password\_123

spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.SQLServerDialect

spring.jpa.hibernate.ddl-auto=update

**Step 5:** To test whether application able to connect with database or not we will simply add given below code in main method.

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(Demo2DataJpaApplication.class, args);

StudentRepository studentRepository = context.getBean(StudentRepository.class);

Students student = new Students();

student.setStudent\_name("Sam");

student.setStudent\_address("Mumbai");

student.setStudent\_phone("+91-12345678");

System.out.println(studentRepository.save(student));

}

Above code will create a table and data in table.

**Step 6:** Now we will perform CRUD operations

**Operation 1.** We have seen first operation create rather insert using save method. But in case if we have multiple users to add in one go then we can use saveAll method which takes object of Iterator (List Object or set object).

package com;

import java.util.List;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import com.dao.StudentRepository;

import com.model.Students;

@SpringBootApplication

public class Demo2DataJpaApplication {

public static void main(String[] args) {

ApplicationContext context =SpringApplication.run(Demo2DataJpaApplication.class, args);

StudentRepository studentRepository = context.getBean(StudentRepository.class);

Students student = new Students();

student.setStudent\_name("Sam");

student.setStudent\_address("Mumbai");

student.setStudent\_phone("+91-12345678");

// Add single student

// studentRepository.save(student);

System.out.println(studentRepository.save(student. getStudent\_name());

Students student2 = new Students();

student2.setStudent\_name("Tom");

student2.setStudent\_address("Chennai");

student2.setStudent\_phone("+91-453526728");

Students student3 = new Students();

student3.setStudent\_name("Tina");

student3.setStudent\_address("Bangalore");

student3.setStudent\_phone("+91-453526728");

List<Students> students = List.of(student2,student3);

// Add multiple students in one go

// studentRepository.saveAll(students);

studentsAdded.forEach(stud->{

System.out.println(stud.getStudent\_name());

});

}

}

**Operation 2:**  Update Student. In order to perform the update operation first we will have to get the employee from table. For this we will use the findById method which will return us object of Optional and from that we can get object of student by using method **get()** and after getting the student we will have to save new values.

// Update Student

Optional<Students> stud2 = studentRepository.findById(153);

Students st = stud2.get();

st.setStudent\_name("TinaBull");

studentRepository.save(st);

System.out.println("Table Updated...");

**Operation 3:**  In order to get data from table we have two methods provided by CRUDRepository

1. findAll()
2. findById(int id)

Code for displaying data from table:

// Getting data from table - single student

Optional<Students> stud1 = studentRepository.findById(153);

System.out.println(stud1.get().getStudent\_name()+"--"+stud1.get().getStudent\_address());

// Getting data from table - all student

Iterable<Students> itr2 = studentRepository.findAll();

Iterator<Students> irtr = itr2.iterator();

for (Students std : itr2) {

System.out.println(std.getStudent\_name()+"--"+std.getStudent\_address());

}

**Operation 4:** In order to delete data from table CRUDRepository has provided us two methods

1. deleteById(int id)
2. deleteAll()

// -----------------------Delete------------------------------------

// delete single student

// studentRepository.deleteById(152);

// System.out.println("Deleted");

// Delete All Employee

Iterable<Students> allUSer = studentRepository.findAll();

studentRepository.deleteAll();

System.out.println("All Students Deleted");

}

**Custom Finder:**

We have used two methods for fetching data from database.

1. findAll()
2. findById(int id)
3. findAllById(Iterable<Integer> id)

But here we can get data only with respect to Id only but in case if our requirement is that we have to get the data by using student name or any other condition then these methods won’t be useful So, in order to achieve this requirement, we have to use custom finder method.

Let’s see how to use them

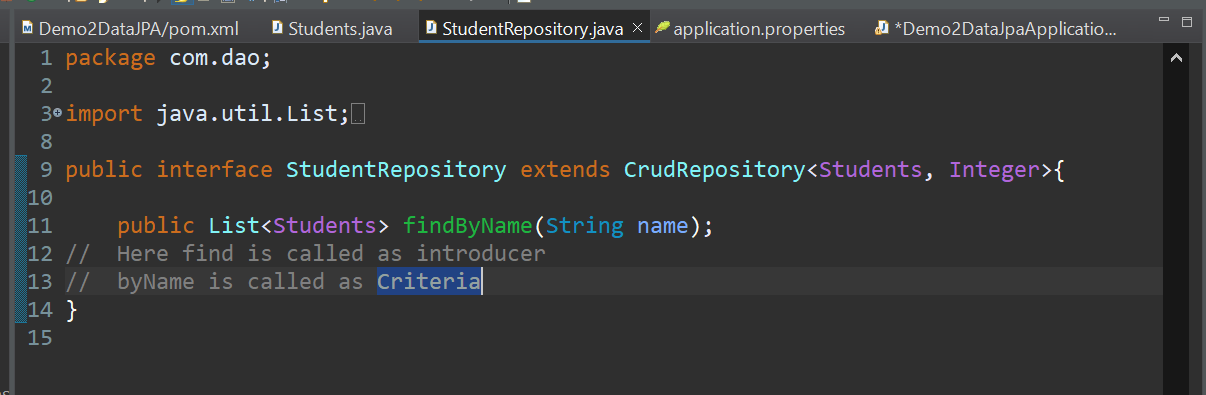
Step 1. Let’s consider we have to fire command like given below:

Select \* from student where student\_name=”name”

So, in order to get this query data in Spring JPA we have to create a method which can gives us data. Method definition should follow some standards.

Public List<Students> findByName(String name).

This method we have to specify in interface where we have extended CRUDRepository.



package com.dao;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

import com.model.Students;

public interface StudentRepository extends CrudRepository<Students, Integer>{

public List<Students> findByName(String name);

// Here find is called as introducer

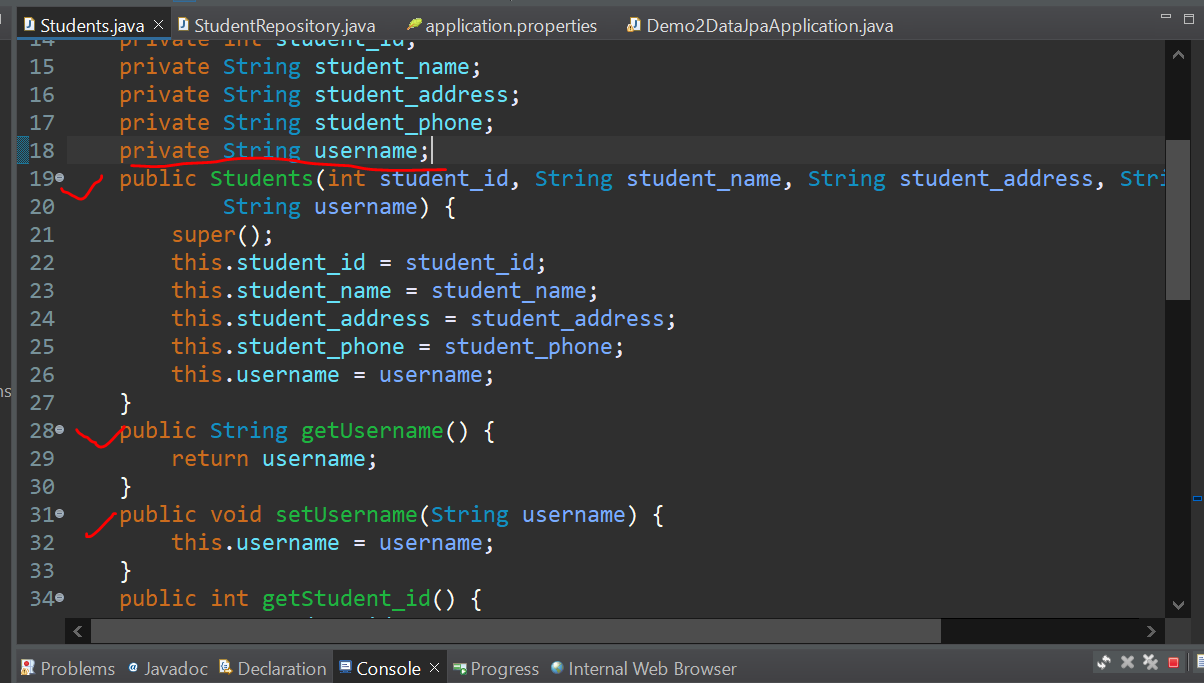
// byName is called as Criteria

}

Now, we can use this method wherever there we have to fetch data on basis of Name.

Note:- Please try to provide the name properly.

In our case we have named as student\_name. This won’t be recognized so I will create a new property in Students as username. Create getter and setters. Also create new constructor with new property.



package com.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name="Student")

public class Students {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private int student\_id;

private String student\_name;

private String student\_address;

private String student\_phone;

private String username;

public Students(int student\_id, String student\_name, String student\_address, String student\_phone,

String username) {

super();

this.student\_id = student\_id;

this.student\_name = student\_name;

this.student\_address = student\_address;

this.student\_phone = student\_phone;

this.username = username;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public int getStudent\_id() {

return student\_id;

}

public void setStudent\_id(int student\_id) {

this.student\_id = student\_id;

}

public String getStudent\_name() {

return student\_name;

}

public void setStudent\_name(String student\_name) {

this.student\_name = student\_name;

}

public String getStudent\_address() {

return student\_address;

}

public void setStudent\_address(String student\_address) {

this.student\_address = student\_address;

}

public String getStudent\_phone() {

return student\_phone;

}

public void setStudent\_phone(String student\_phone) {

this.student\_phone = student\_phone;

}

public Students() {

super();

// TODO Auto-generated constructor stub

}

public Students(int student\_id, String student\_name, String student\_address, String student\_phone) {

super();

this.student\_id = student\_id;

this.student\_name = student\_name;

this.student\_address = student\_address;

this.student\_phone = student\_phone;

}

}

First add the data in table with new property.

Students student4 = new Students();

student4.setStudent\_name("James");

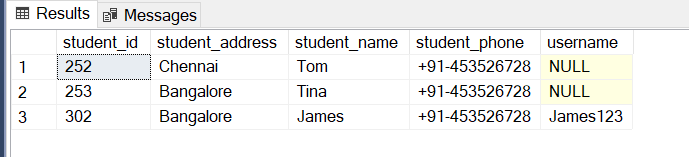
student4.setStudent\_address("Bangalore");

student4.setStudent\_phone("+91-453526728");

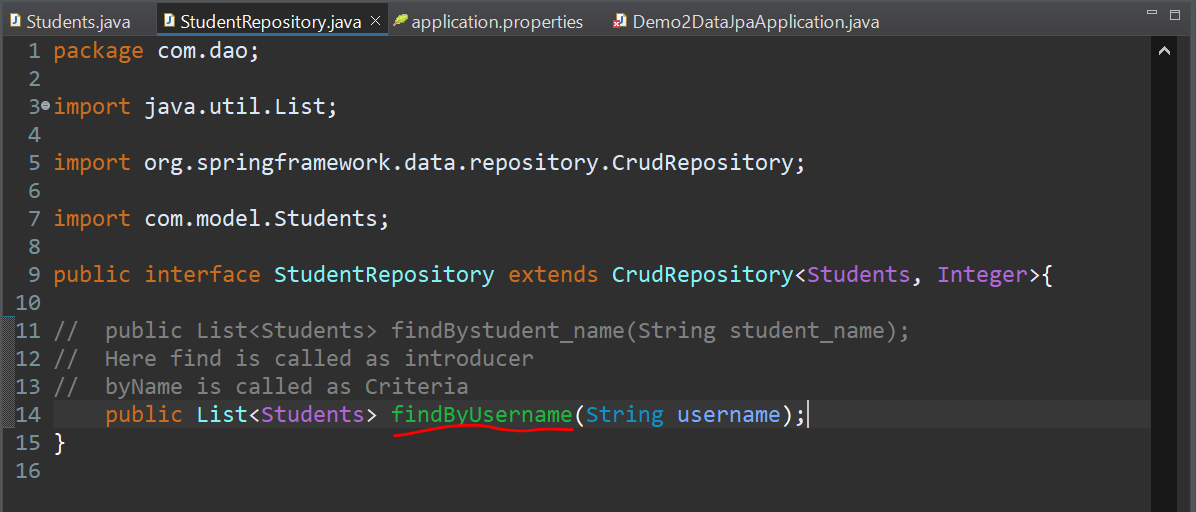
student4.setUsername("James123");

studentRepository.save(student4); 

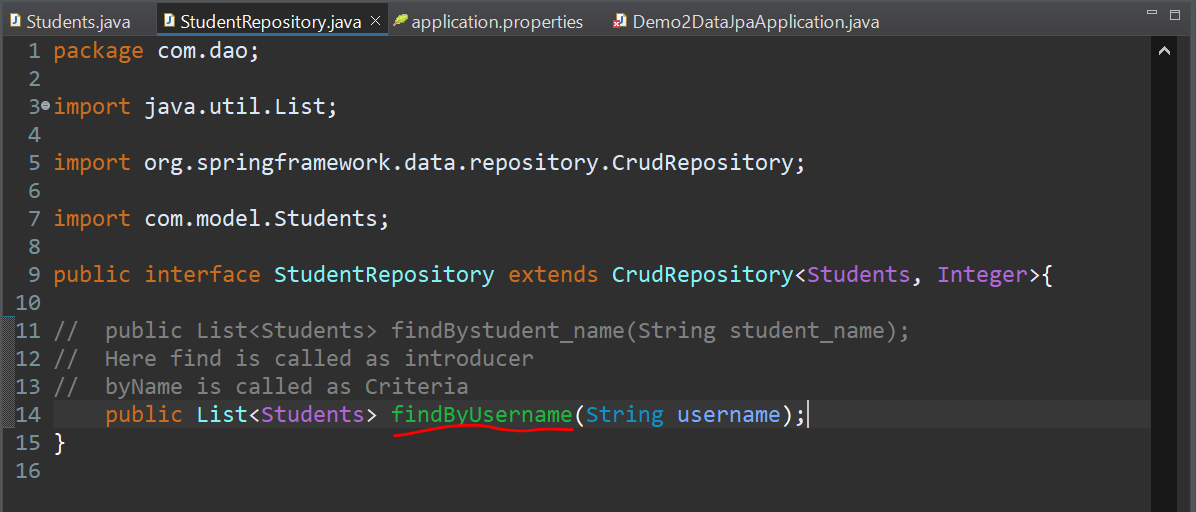
New column will be added to our table



Now, create a custom finder method in StudentRepository interface.



Call this method in our main method in order to get data with respect to username



List<Students> s1 = studentRepository.findByUsername("James123");

for (Iterator iterator = s1.iterator(); iterator.hasNext();) {

Students st1 = (Students) iterator.next();

System.out.println(st1.getUsername());

}

For details:

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords>

So, now we have seen how we can create custom finder.

But in case you have to fire your own query then this can also be achieved using JPA.

We can use @Query annotation for same.

We can write

1. JPQL – Java persistence Query language
2. Native – native SQL queries

Scenario: Let’s consider that here we have to fire our own queries then we can’t use custom finder methods. So, how we can achieve this?

*@Query*(value="select \* FROM student", nativeQuery=true)

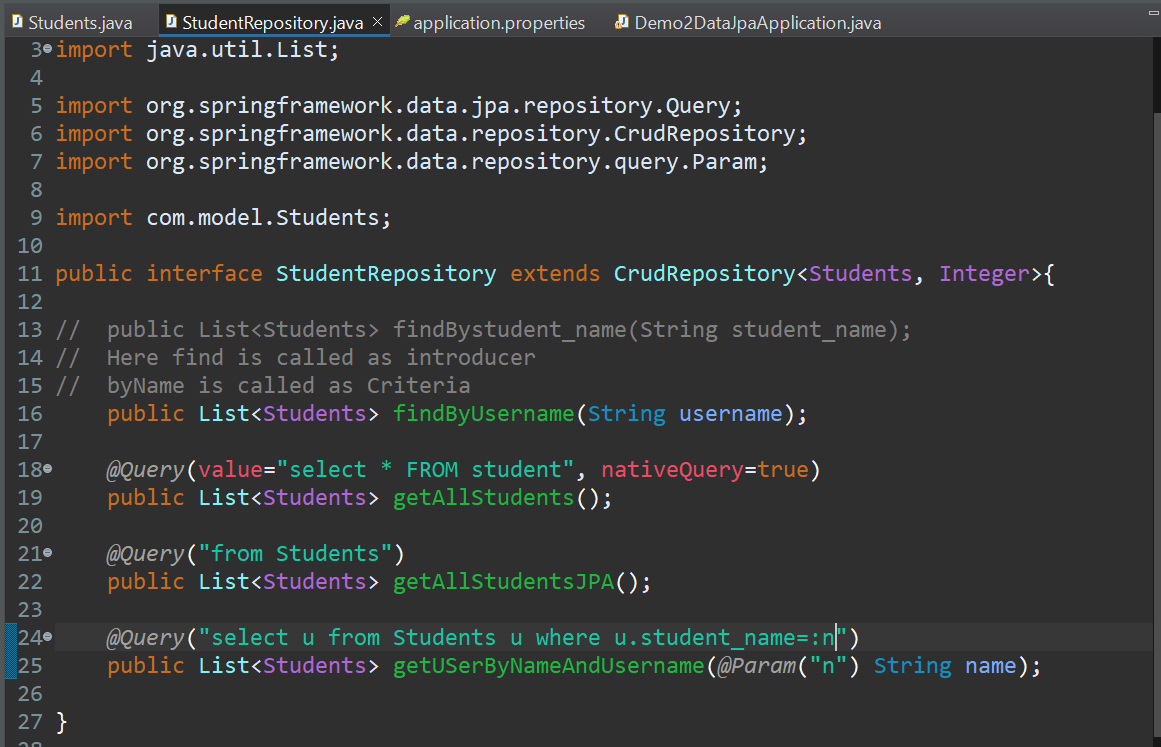
public List<Students> getAllStudents();

*@Query*("from Students")

public List<Students> getAllStudentsJPA();

*@Query*("select u from Students u where u.student\_name=:n")

public List<Students> getUSerByNameAndUsername(*@Param*("n") String name);



How to call these methods?

List<Students> ss= studentRepository.getAllStudents();

// System.out.println(ss.toString());

for (Iterator<Students> iterator = ss.iterator(); iterator.hasNext();) {

Students ss1 = (Students) iterator.next();

System.out.println(ss1.getStudent\_name());

}

List<Students> ss1= studentRepository.getAllStudentsJPA();

// System.out.println(ss.toString());

for (Iterator<Students> iterator = ss1.iterator(); iterator.hasNext();) {

Students ss2 = (Students) iterator.next();

System.out.println(ss2.getStudent\_name());

}

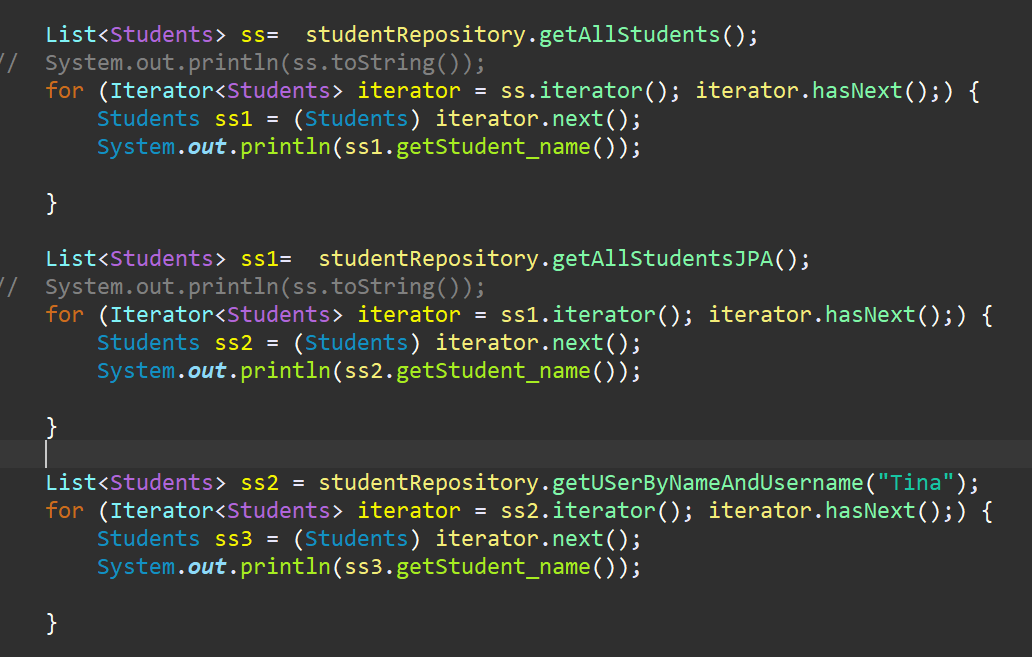
List<Students> ss2 = studentRepository.getUSerByNameAndUsername("Tina");

for (Iterator<Students> iterator = ss2.iterator(); iterator.hasNext();) {

Students ss3 = (Students) iterator.next();

System.out.println(ss3.getStudent\_name());

}



In this way we can use @Query.

H2 Database:

You can try with H2 database, Copy your project and paste. Add H2 starter POM and add configuration in application.properties file.

Pom.xml file dependency:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

Application.properties:

spring.datasource.url=jdbc:h2:mem:student

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true