**Spring Data JPA**

**1.Spring Data JPA**

Spring Data JPA is a part of the larger Spring Data project, which aims to simplify data access and database interactions in Java applications using Spring. Spring Data JPA provides an abstraction over JPA (Java Persistence API), making it easier to implement JPA-based repositories and handle relational data.

Here are the key concepts and features of Spring Data JPA:

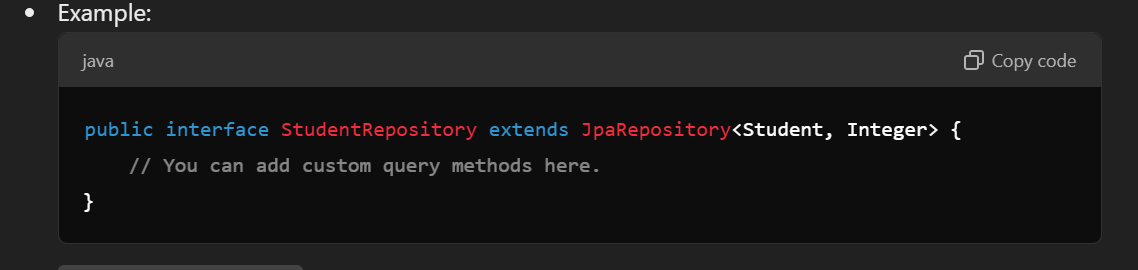
**1. Simplifies Database Operations**

Spring Data JPA reduces boilerplate code by providing simple CRUD (Create, Read, Update, Delete) operations with minimal coding. You no longer need to write complex implementations for basic database operations.

**2. Repository Interfaces**

* You define an interface that extends JpaRepository, CrudRepository, or another repository interface.
* Spring Data JPA provides the implementation automatically.

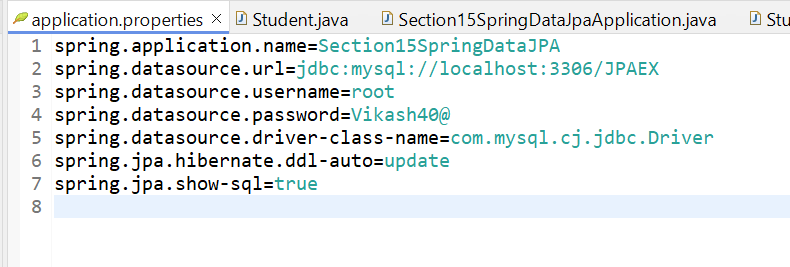
**Example:**

****

* JpaRepository<T, ID>:
* T: The entity type (e.g., Student).
* ID: The type of the primary key (e.g., Integer).

**4. Creating Table and inserting data.**

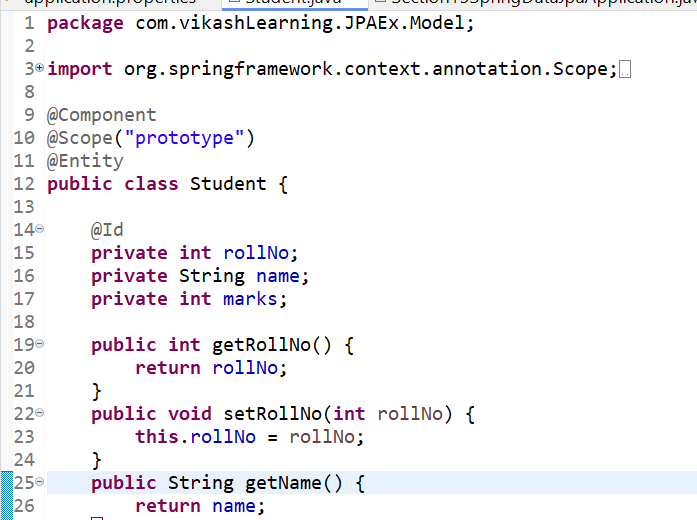
**4.1.application.properties**

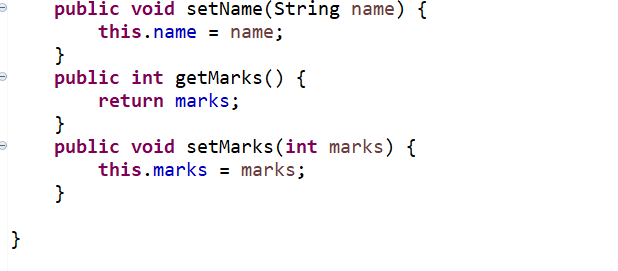
****

**4.1.1 Explanation:**

* **spring.application.name**: Specifies the name of the application.
* **spring.datasource.url**: The JDBC URL used to connect to the MySQL database. Here, the database name is JPAEX.
* **spring.datasource.username** and **spring.datasource.password**: Credentials used to connect to the MySQL database.
* **spring.datasource.driver-class-name**: Specifies the JDBC driver class for MySQL.
* **spring.jpa.hibernate.ddl-auto=update**: Tells Hibernate to update the database schema automatically when needed. It creates tables if they don’t exist and updates their structure if they do.
* **spring.jpa.show-sql=true**: Shows the SQL queries being executed in the logs for debugging purposes.

**4.2 Student.java (Entity class)**

****

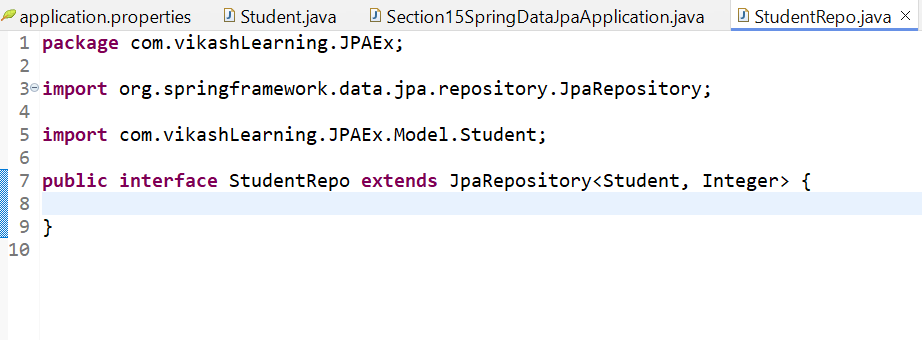
****

**Annotations and Their Use:**

* **@Component**: Marks the class as a Spring-managed bean. This means Spring will create and manage instances of Student in the application context.
* **@Scope("prototype"):** Indicates that a new instance of Student should be created every time it is requested from the Spring container, instead of reusing the same instance (which would be the case with the default singleton scope).
* **@Entity**: Specifies that the Student class is a JPA entity and is mapped to a database table. The table name will be Student by default, or you can customize it using the @Table annotation.
* **@Id:** Marks the field rollNo as the primary key for the entity.

**4.**3

**StudentRepo.java**



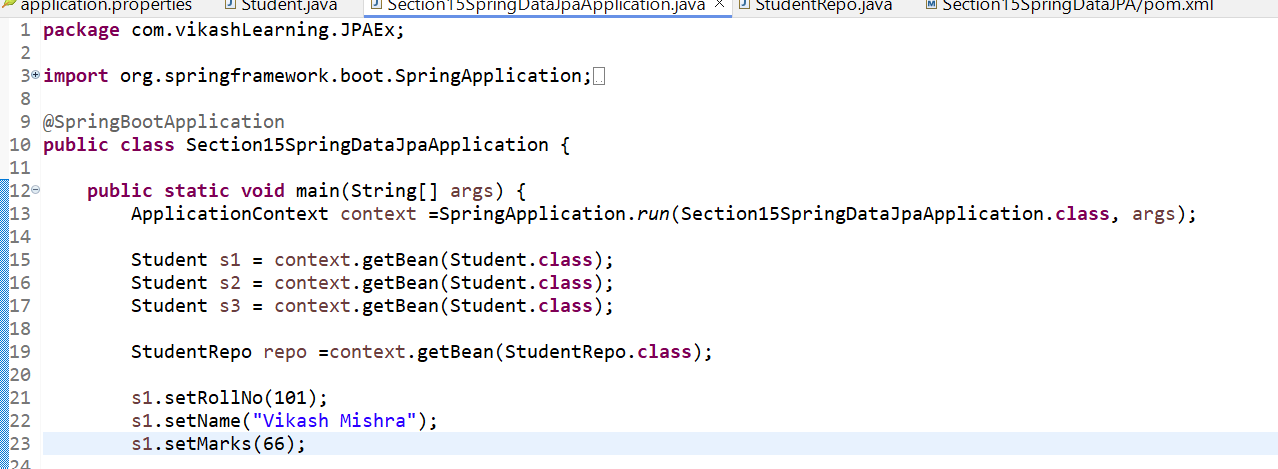
**Explanation:**

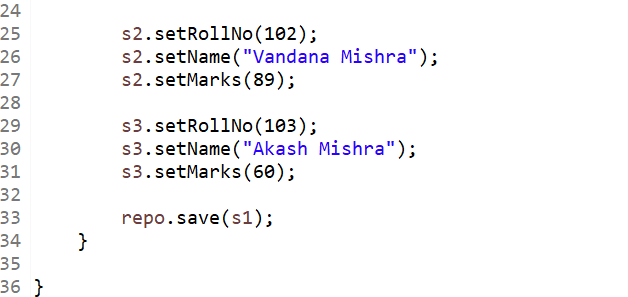
4.3.1 JpaRepository<Student, Integer>:

* Student: This is the entity type the repository will manage.
* Integer: This is the type of the primary key (i.e., rollNo).

4.3.2 JpaRepository provides built-in methods for CRUD operations such as save(), findAll(), findById(), delete(), etc., without requiring additional implementation code.

**4.4 Section15SpringDataJpaApplication**

****

****

**4.4. Flow of Execution:**

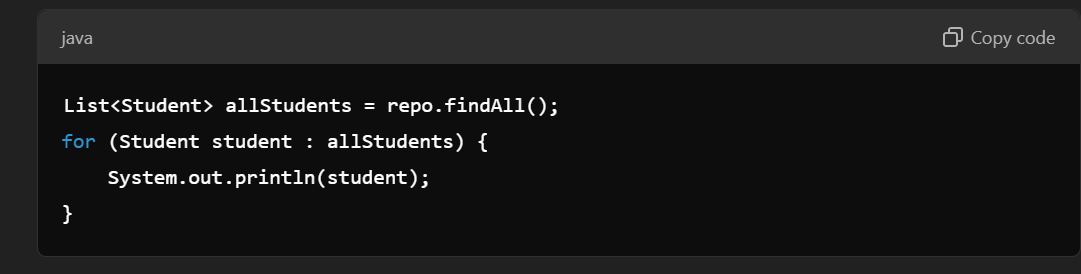
* **Application Start (main Method):**
  + The SpringApplication.run() method boots the application by creating an ApplicationContext and starting the embedded server.
  + It also scans the packages for any components (@Component, @Entity, @Repository, etc.) to manage.
* **Bean Instantiation:**
  + The context.getBean(Student.class) is called thrice to create three new instances of Student. Since @Scope("prototype") is used, each call creates a new instance of Student.
  + The Prototype Scope ensures that every time you request a Student bean from the context, a new instance is created.
* **Populating Data:**
  + Values are assigned to the fields of Student objects (s1, s2, s3).
* **Saving the Entity (repo.save(s1)):**
  + The StudentRepo bean is retrieved using context.getBean(StudentRepo.class).
  + repo.save(s1) is called to save s1 to the database.
  + The StudentRepo interface extends JpaRepository, so it already has methods to handle saving, retrieving, and deleting entities.
  + Since @Entity is used, s1 is saved as a row in the corresponding table in the MySQL database.
* **Database Operations:**
  + The spring.jpa.hibernate.ddl-auto=update configuration makes sure that if the Student table doesn't exist, Hibernate will create it automatically.
  + Spring JPA uses Hibernate to handle the persistence layer, and the save() method from JpaRepository persists s1 into the database.

**5. Find All**

**findAll()**

* **Purpose**: This method retrieves all records from the corresponding database table.
* **Usage**: It returns a list of all entities (List<Student> in your case).

**Example**:



* The findAll() method does not require any input parameters.
* The method will generate a SQL query that retrieves all rows from the database table representing the Student entity.
* **Return Type**: List<Student>, meaning it will return all Student records as a list.

**6. FindById**

**FindById()**

* Purpose: This method retrieves a specific record from the database table based on the provided ID (the primary key).
* Usage: It returns an Optional<Student>, which can either contain the result or indicate that no result was found.

Example

****

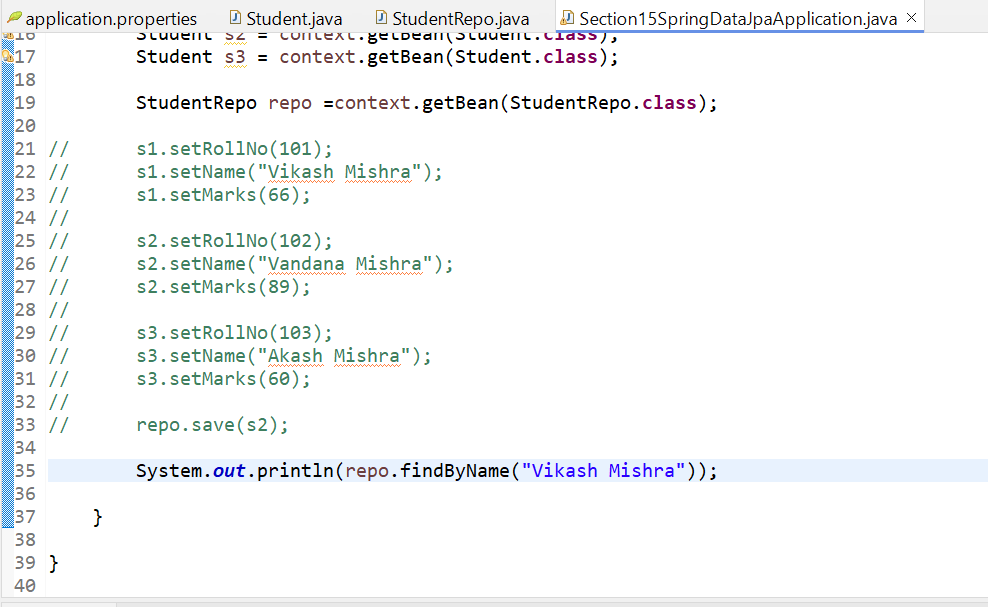
* The findById() method takes one argument (the primary key) and retrieves the matching record.
* Return Type: Optional<Student>. This means the method returns an Optional, which is a container object that can hold either a value or be empty if no record was found.

**Optional Explained**

* Optional<T> is a container used to handle the possibility of null values more effectively and prevent NullPointerException.
* In the example, findById(102) returns an Optional<Student>. If a Student with id = 102 exists, the Optional will contain that value; if not, it will be empty.

**7. Query DSL**

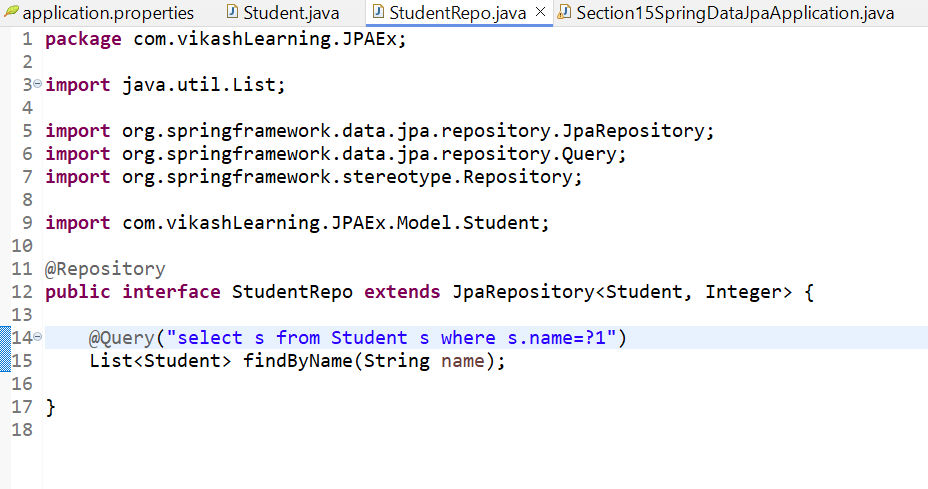
Derived query methods are created by defining the method names in the repository interface, following a specific naming convention that tells Spring Data JPA to derive the query from the method name.



**7.1. JPQL (Java Persistence Query Language)**

JPQL is a query language similar to SQL but operates on entity objects rather than directly on database tables. It allows you to write more complex queries that can't be expressed using query methods alone.

Example:



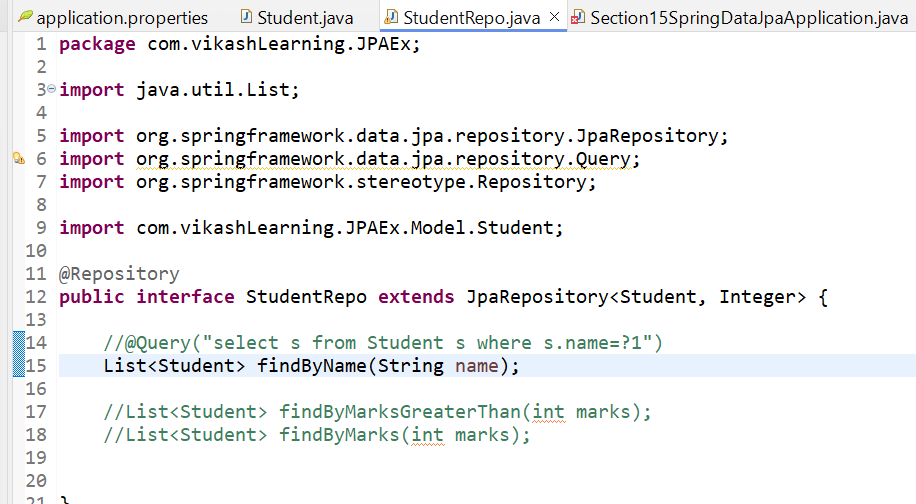
* @Query: Annotates a custom query that operates at the entity level.

Now if u don’t write the Query it will still work because code works even though you haven’t explicitly written the @Query("select s from Student s where s.name=?1") is because Spring Data JPA uses **method name conventions** to automatically generate the necessary query for you. This feature is known as **derived query methods**.

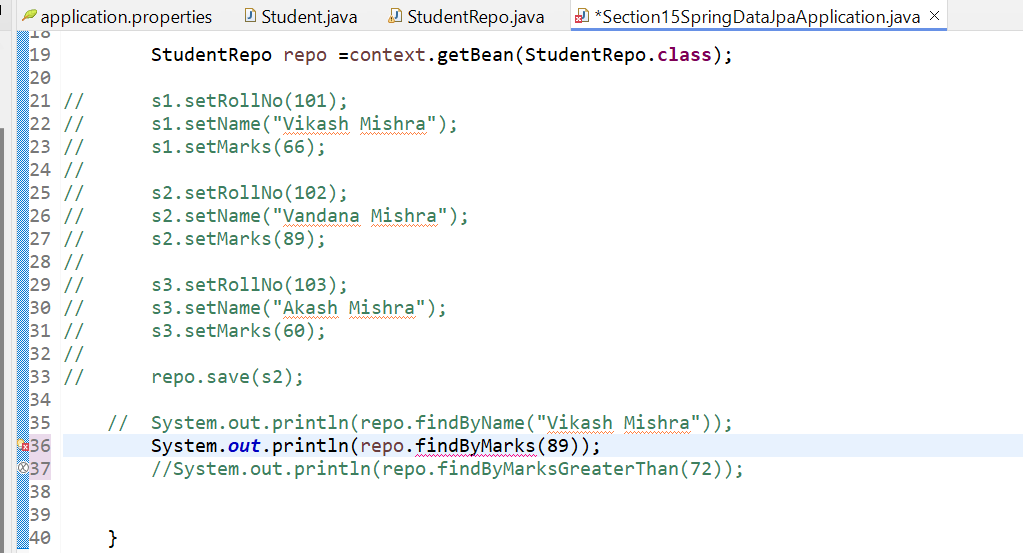
**How Derived Query Methods Work**

Spring Data JPA can create queries automatically by parsing the method names you define in your repository interface. This feature saves you from having to write repetitive or simple queries manually. It works based on a specific naming convention.

Updated repo class

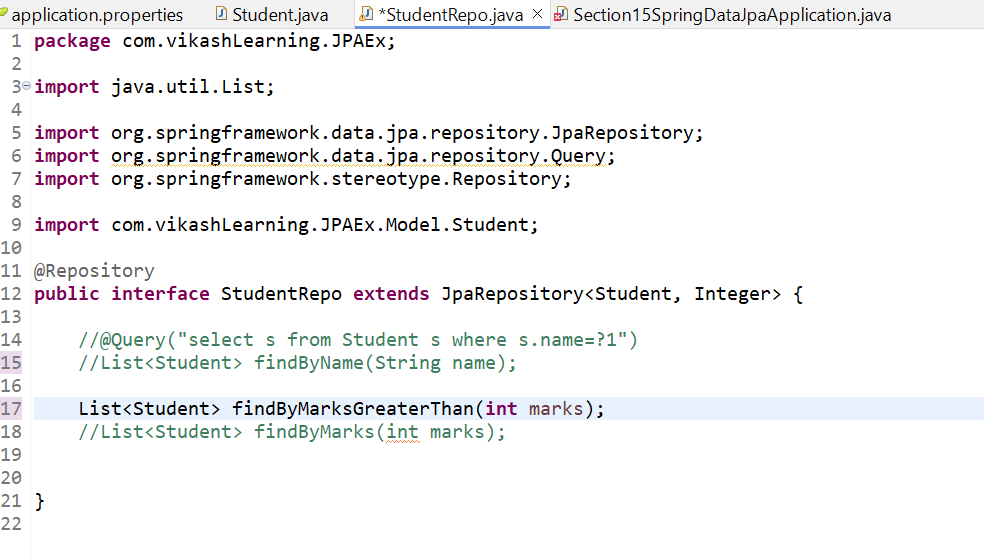


Updated Section15SpringDataJpaApplication class

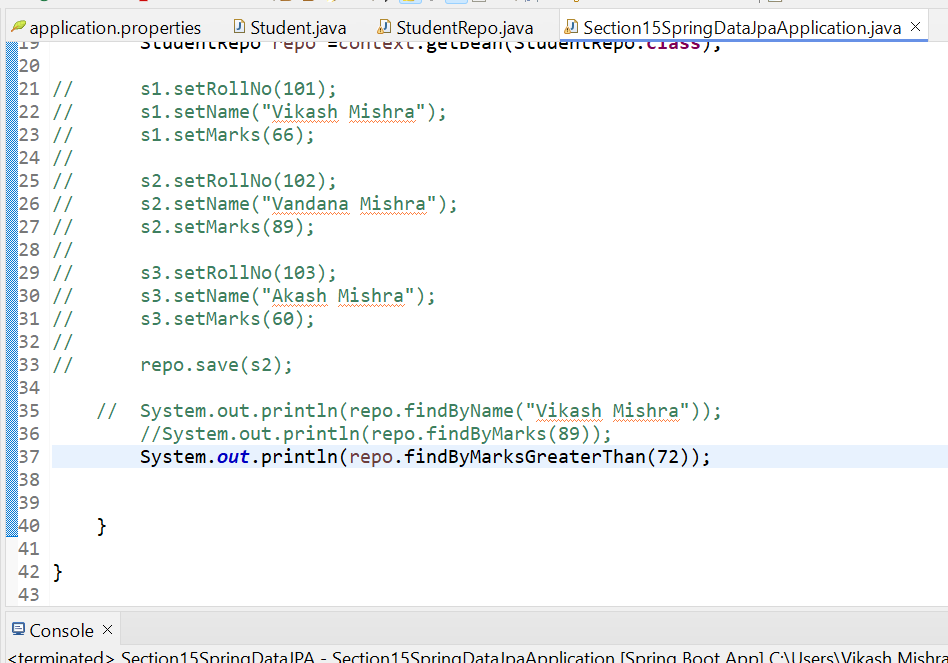


We can write it for complex query as well for example :

Updated repo class

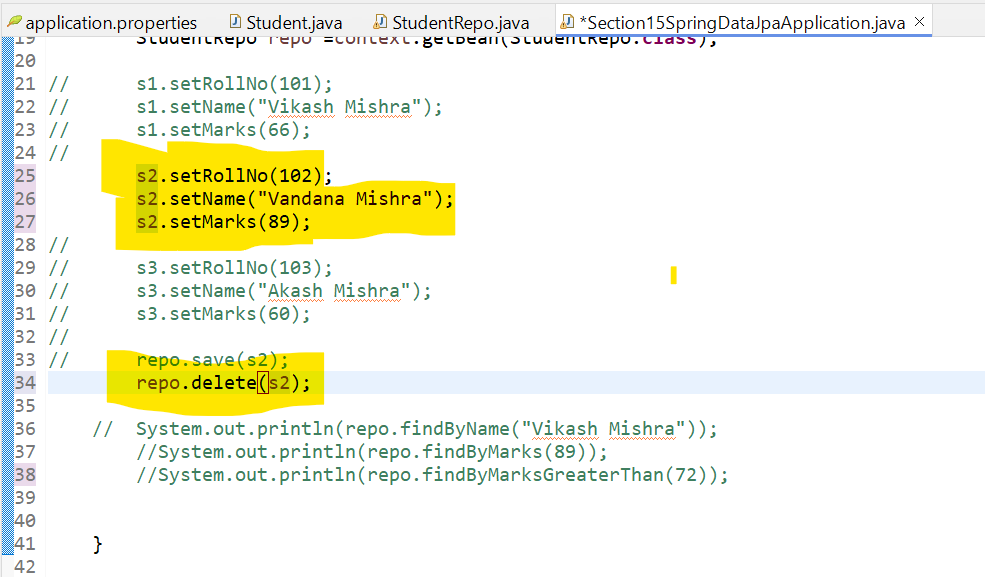


Updated Section15SpringDataJpaApplication class



**8. Update and Delete**

This is for Delete operation



**9. JPA in Job app and Search By keyword**

**.**