**Querying**

**with HQL and JPA QL**

Contents

[I/ Understanding the various query options 3](#_Toc520699700)

[II/ Writing HQL and JPA QL queries 4](#_Toc520699701)

[1/ Configuration and preparing a query 4](#_Toc520699702)

[1.1/ HQL 4](#_Toc520699703)

[1.2/ JPA QL 4](#_Toc520699704)

[2/ Creating a query 6](#_Toc520699705)

[2.1/ Creating a query object 6](#_Toc520699706)

[2.2/ Paging the result 7](#_Toc520699707)

[2.3/ Considering parameter binding 7](#_Toc520699708)

[2.4/ Using positional parameters 8](#_Toc520699709)

[3/ Executing a query 8](#_Toc520699710)

[III/ Named Query and Named Stored Procedure 9](#_Toc520699711)

[1/ Named query 9](#_Toc520699712)

[1.1/ Benefits 9](#_Toc520699713)

[1.2/ Usage 9](#_Toc520699714)

[2/ Named stored procedure 10](#_Toc520699715)

[IV/ Joins, reporting queries, subselects 12](#_Toc520699716)

[1/ Joins 12](#_Toc520699717)

[1.1/ Implicit association joins 14](#_Toc520699718)

[1.2/ Joins expressed in the FROM clause 14](#_Toc520699719)

[1.3/ Dynamic fetching strategies with joins 15](#_Toc520699720)

[1.4/ Theta-style joins 16](#_Toc520699721)

[2/ Reporting queries 17](#_Toc520699722)

[3/ Subselects 17](#_Toc520699723)

# I/ Understanding the various query options

* There are three ways to express queries in Hibernate:
  + Hibernate Query Language (HQL) and the subset standardized as JPA QL:

|  |
| --- |
| session.createQuery("from EmployeeEntity e where e.firstName like 'Hai%' ");  entityManager.createQuery("from EmployeeEntity e where e.firstName like 'Hai%' "); |

* + Criteria API for query by criteria (QBC) and query by example (QBE):

|  |
| --- |
| session.createCriteria(EmployeeEntity.class)  .add(Restrictions.like("firstName", "Hai%")); |

* + Direct SQL with or without automatic mapping of resultsets to objects:

|  |
| --- |
| session.createSQLQuery("select {e.\*} from EmployeeEntity {e}  where firstName like 'Hai%'")  .addEntity("e", EmployeeEntity.class); |

# II/ Writing HQL and JPA QL queries

## 1/ Configuration and preparing a query

### 1.1/ HQL

* **Step 1:** Add this code into file pom.xml of the project:

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>5.1.2.Final</version>  <scope>provided</scope>  </dependency> |

* **Step 2:** The **org.hibernate.Query** interface defines several methods for controlling execution of a query. In addition, Query provides methods for binding concrete values to query parameters. To execute a query in your application, you need to obtain an instance of one of these interfaces, using the **Session**.

So, import:

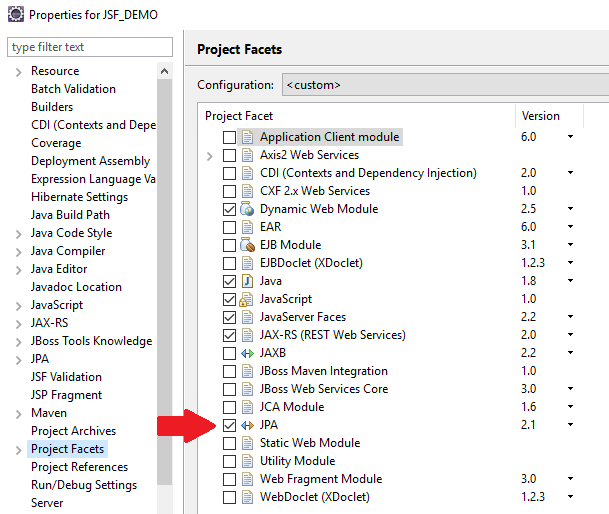
|  |
| --- |
| **import** javax.persistence.PersistenceContext;  **import** org.hibernate.Query;  **import** org.hibernate.Session; |

And declare:

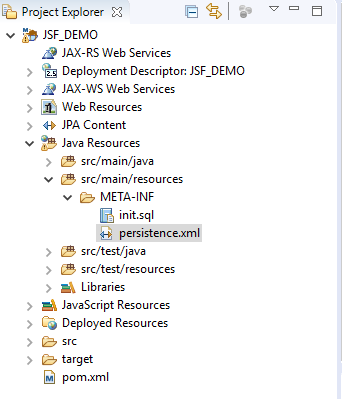
|  |
| --- |
| @PersistenceContext  Session session; |

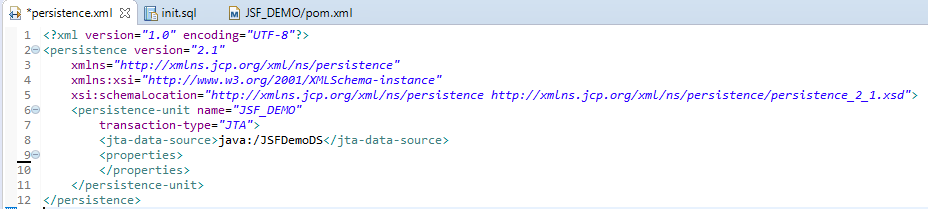
### 1.2/ JPA QL

* **Step 1:** Remember how to create a maven project and the first steps to configurate necessities relating to JPA to work on it:
  + Right click on the project, choose **Properties** > **Project Facets**, then check **JPA** (you can also select the version of JPA):



* + And remember how to create the file **persistence.xml** in **src/main/resources**:





* **Step 2:** A little bit same as HQL, we import and declare:

|  |
| --- |
| **import** javax.persistence.EntityManager;  **import** javax.persistence.PersistenceContext;  **import** javax.persistence.Query;  @PersistenceContext  EntityManager em; |

## 2/ Creating a query

### 2.1/ Creating a query object

* HQL query:

|  |
| --- |
| Query hqlQuery = session.createQuery("from User"); |

* SQL query:

|  |
| --- |
| Query sqlQuery = session.createSQLQuery(  "select {user.\*} from USERS {user}"  ).addEntity("user", User.class); |

* Criteria query:

|  |
| --- |
| Criteria crit = session.createCriteria(User.class); |

* JPA QL query:

|  |
| --- |
| Query ejbQuery = em.createQuery("select u from User u"); |

* Native query:

|  |
| --- |
| Query sqlQuery = em.createNativeQuery(  "select u.USER\_ID, u.FIRSTNAME, u.LASTNAME from USERS u",  User.class  ); |

### 2.2/ Paging the result

* HQL and JPA QL are the same in the way using **setMaxResults**:
  + HQL:

|  |
| --- |
| Query query = session.createQuery("from EmployeeEntity e order by e.firstName asc");  query.setMaxResults(10); |

* + JPA QL:

|  |
| --- |
| Query query = em.createQuery("select e from EmployeeEntity e order by e.firstName asc").setFirstResult(40); |

### 2.3/ Considering parameter binding

* HQL:

|  |
| --- |
| String queryString = "from Item item"  + " where item.description like :search"  + " and item.date > :minDate";  Query q = session.createQuery(queryString)  .setString("search", searchString)  .setDate("minDate", mDate); |

* JPA QL:

|  |
| --- |
| String queryString = "from Item item"  + " where item.description like :search"  + " and item.date > :minDate";  Query q = em.createQuery(queryString)  .setParameter("search", searchString)  .setParameter("minDate", mDate, TemporalType.DATE); |

### 2.4/ Using positional parameters

* HQL:

|  |
| --- |
| String queryString = "from Item item"  + " where item.description like ?"  + " and item.date > ?";  Query q = session.createQuery(queryString)  .setString(0, searchString)  .setDate(1, minDate); |

* JPA QL:

|  |
| --- |
| String queryString = "from Item item"  + " where item.description like ?1"  + " and item.date > ?2";  Query q = em.createQuery(queryString)  .setParameter(1, searchString)  .setParameter(2, minDate, TemporalType.DATE); |

**\* Notice:** If you have to use positional parameters, remember that Hibernate starts counting at 0, but Java Persistence starts at 1, and that you have to add a number to each question mark in a JPA QL query string. They have different legacy roots: Hibernate in JDBC, Java Persistence in older versions of EJB QL.

## 3/ Executing a query

* In this document, we only consider the case that we list all the results:
  + HQL: In Hibernate, the **list()** method executes the query and returns the results as a **java.util.List**:

|  |
| --- |
| List result = myQuery.list(); |

* + JPA QL: Java Persistence offers a method with the same semantics, but a different name:

|  |
| --- |
| List result = myJPAQuery.getResultList(); |

# III/ Named Query and Named Stored Procedure

## 1/ Named query

### 1.1/ Benefits

* Compiled and validated at app start-up time: That means if anything is wrong with it, it will be reported immediately, and the application won't start.
* Easier to maintain than string literals embedded in your code: HQL (or JPA SQL) and Native SQL queries can be used and replaced without code changes (no need to re-compile your code).
* Good for performance: The query is prepared once and ready for usage anywhere in the app. So, any subsequent usage of named query will not cause any additional processing.

### 1.2/ Usage

* We can define a named query in XML metadata or define a named query with annotations, but in this document, we only consider defining a named query with annotations.
* The Java Persistence standard specifies the @NamedQuery and @NamedNativeQuery annotations. You can either place these annotations into the metadata of a particular class or into JPA XML descriptor file. Note that the query name must be globally unique in all cases; no class or package name is automatically prefixed.
* Let’s assume you consider a particular named query to belong to a particular entity class:

|  |
| --- |
| package auction.model;  import ...;  @NamedQueries({  @NamedQuery(  name = "getSpecialEmployeeListWithNamedQuery",  query = "select e from EmployeeEntity e where e.gender like :gender and e.firstName like :firstName"  )  })  @Entity  @Table(name = "ITEM")  public class Item { ... } |

* And you can call it:
  + HQL:

|  |
| --- |
| Query query = session.getNamedQuery("getSpecialEmployeeListWithNamedQuery")  .setString("gender", gender)  .setString("firstName", firstName); |

* + JPA QL:

|  |
| --- |
| Query query = em.createNamedQuery("getSpecialEmployeeListWithNamedQuery")  .setParameter("gender", gender)  .setParameter("firstName", firstName); |

## 2/ Named stored procedure

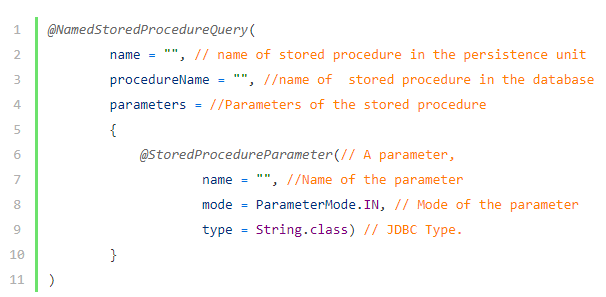
### 2.1/ Working with stored procedure that return a refcursor

* Step 1: Firstly, we have a function (because we are using PostgreSQL and there are only functions supported in PostgreSQL, no stored procedures) as below. This function returns a list of employees of a specific department:

**In Postgres database**, please create one **Function that returns a refcursor**

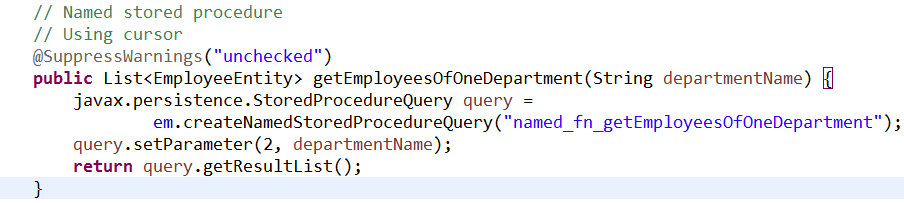
|  |
| --- |
| CREATE OR REPLACE FUNCTION fn\_getEmployeesOfOneDepartment(departmentName varchar)  RETURNS refcursor AS  $BODY$  DECLARE employees refcursor; -- Declare cursor variables  BEGIN  OPEN employees FOR  SELECT e.\*  FROM employee e, department d  WHERE e.department = d.id and d.name = departmentName;  RETURN employees;  END;  $BODY$  LANGUAGE plpgsql |

* Step 2: Secondly, we define or named stored procedure call via annotation:



|  |
| --- |
| @NamedStoredProcedureQuery(  name = "named\_fn\_getEmployeesOfOneDepartment",  procedureName = "fn\_getEmployeesOfOneDepartment",  resultClasses = EmployeeEntity.**class**,  parameters = {  @StoredProcedureParameter(mode = ParameterMode.***REF\_CURSOR***, type = **void**.**class**),  @StoredProcedureParameter(mode = ParameterMode.***IN***,  type = String.**class**)  }  )  @Entity  @Table(name = "employee")  **public** **class** EmployeeEntity **implements** IEntity {  …  } |

* + Step 3:



### 2.2/ Working with stored procedure that return a table

* + Step 1: Create a function in Postgres database named as following:

CREATE OR REPLACE FUNCTION find\_Emplyee\_By\_Department\_ID(\_Department\_ID int)

RETURNS TABLE (

first\_name character varying(255) -- visible as OUT parameter inside and outside function

, last\_name character varying(255)

, gender character varying(255)

, email character varying(255)

, name character varying(255)) AS

$func$

BEGIN

RETURN QUERY

SELECT e.first\_name, e.last\_name, e.gender, e.email, d.name

FROM employee e, department d

WHERE e.department = d.id AND

d.id = \_Department\_ID

ORDER BY e.first\_name DESC;

END

$func$ LANGUAGE plpgsql;

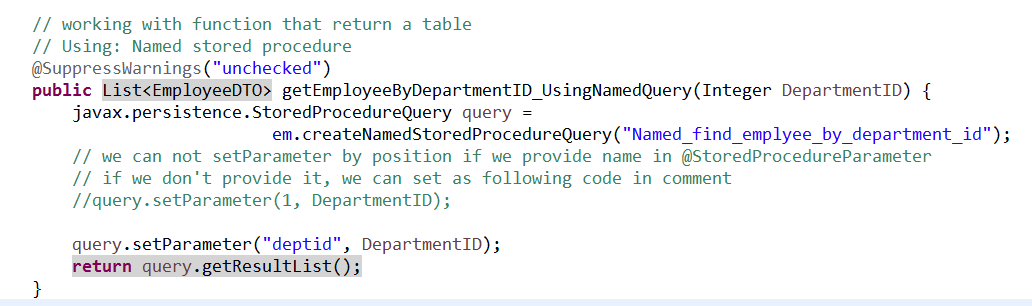
Note: you can check this function by following query:

select \* from find\_Emplyee\_By\_Department\_ID(1);

* + Step 2: Create named query in an Entiy and @SqlResultMapping (Because this function not return an Entity, it returns a DTO or columns from many database table)

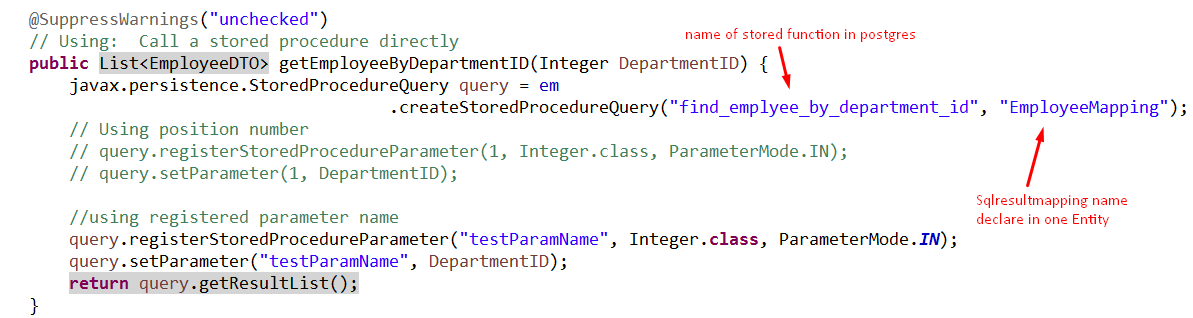


* + Step 3:

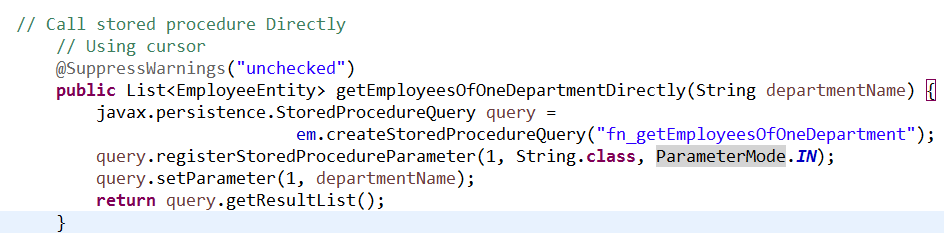


## 3/ Call stored procedure directly

## 3.1/ call a stored proc that return a table



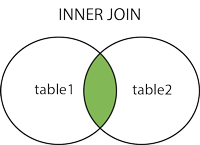
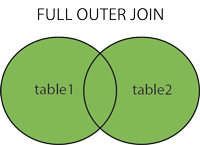
## 3.2/ call a stored proc that return a cursor

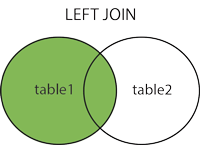
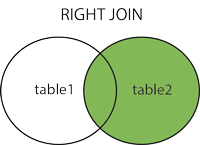


# IV/ Joins, reporting queries, subselects

## 1/ Joins

* We use a join to combine data in two (or more) relations.
* We have 4 types of join: **join** (inner join), **left join** (left outer join), **right join** (right outer join) and **full join** (full outer join).

* HQL and JPA QL provide 4 ways of expressing (inner and outer) joins:
  + An implicit association join
  + An ordinary join in the FROM clause
  + A fetch join in the FROM clause
  + A theta-style join in the WHERE clause
* Before we go to each way how to use joins, we should first take a look at the database structure given for example:

We have 2 tables: **employee** and **department**

* + In Java:

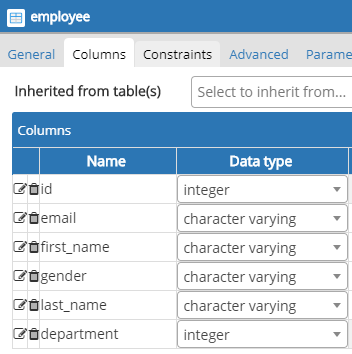
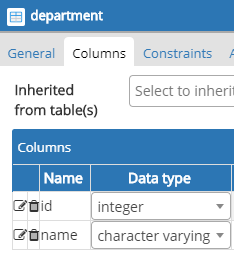
Employee:

|  |
| --- |
| @Entity  @Table(name = "employee")  **public** **class** EmployeeEntity **implements** IEntity {  @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **private** **int** id;  @Column(name = "first\_name", nullable = **false**)  **private** String firstName;  @Column(name = "last\_Name", nullable = **false**)  **private** String lastName;  @Column(name = "gender", nullable = **false**)  **private** String gender;  @Column(name = "email", nullable = **false**)  **private** String email;  @ManyToOne  @JoinColumn(name = "department", nullable = **true**)  **private** DepartmentEntity department; |

Department:

|  |
| --- |
| @Entity  @Table(name = "department")  **public** **class** DepartmentEntity **implements** IEntity {  @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **private** **int** id;  @Column(name = "name", nullable = **false**)  **private** String name; |

* + In PostgreSQL:

### 1.1/ Implicit association joins

* So far, you use simple qualified property name like department.name in query. HQL and JPA QL support multipart property path expressions with a dot notation for two different purposes:
  + Querying components
  + Expressing implicit association joins
* Example:
  + In HQL or JPA QL:

|  |
| --- |
| from EmployeeEntity e where e.department.name like 'C-level' |

* + In Native SQL:

|  |
| --- |
| select e.\*  from employee e join department d on e.department = d.id  where d.name like 'C-level' |

### 1.2/ Joins expressed in the FROM clause

* Example:
  + Using foreign keys:
    - In HQL or JPA QL:

|  |
| --- |
| select e  from EmployeeEntity e right join e.department d  where d.id > 2 |

* + - In Native SQL:

|  |
| --- |
| select e.\*  from employee e right join department d on e.department = d.id  where d.id > 2 |

* + Not using foreign keys:
    - In HQL or JPA QL:

|  |
| --- |
| select e  from EmployeeEntity e right join DepartmentEntity d  on e.department.id = d.id  where d.id > 2 |

* + - In Native SQL:

|  |
| --- |
| select e.\*  from employee e right join department d on e.department = d.id  where d.id > 2 |

### 1.3/ Dynamic fetching strategies with joins

* In One-To-Many relationship, the returned instances have a collection (for example: Employees collection in Department) . This collection, if mapped as lazy="true" (default), isn’t initialized, and an additional SQL statement is triggered as soon as you access it. The same is true for all single-ended associations. By default, Hibernate generates a proxy and loads the associated Department instance lazily and only on-demand.
* By default, we have:
  + One-To-Many: LAZY
  + Many-To-One: EAGER
  + Many-To-Many: LAZY
  + One-To-One: EAGER
* In HQL and JPA QL you can specify that an associated entity instance or a collection should be eagerly fetched with the FETCH keyword in the FROM clause. You can also prefetch many-to-one or one-to-one associations, using the same syntax:
  + In HQL or JPA QL:

|  |
| --- |
| select e  from EmployeeEntity e left join fetch e.department d  where e.gender like 'female' |

* + In Native SQL:

|  |
| --- |
| select e.\*  from employee e left join department d on e.department = d.id  where e.gender like 'female' |

### 1.4/ Theta-style joins

* In traditional SQL, a theta-style join is a Cartesian product together with a join condition in the WHERE clause, which is applied on the product to restrict the result. So, it has worse performance than putting join condition in the FROM clause.
* In HQL and JPA QL, the theta-style syntax is useful when your join condition isn’t a foreign key relationship mapped to a class association.
* For example:
  + In HQL or JPA QL:

|  |
| --- |
| select e  from EmployeeEntity e, DepartmentEntity d  where e.department.id = d.id and d.id > 2 |

* + In Native SQL:

|  |
| --- |
| select e.\*  from employee e, department d  where e.department = d.id and d.id > 2 |

## 2/ Reporting queries

* The aggregate functions that are recognized by HQL and standardized in JPA QL are **count(), min(), max(), sum()** and **avg().**
* For example: This query counts all the Items:

|  |
| --- |
| select count(e) from EmployeeEntity e |

And the result is returned as a Long:

|  |
| --- |
| Long count = (Long) session  .createQuery("select count(e) from EmployeeEntity e ") .uniqueResult(); |

## 3/ Subselects

* For example:
  + In HQL and JPA QL:

|  |
| --- |
| from EmployeeEntity e  where e.department.id >= (select max(d.id) from DepartmentEntity d) |

* + In Native SQL:

|  |
| --- |
| select \*  from employee e  where e.department >= (select max(d.id) from department d) |

<https://www.programcreek.com/java-api-examples/index.php?api=javax.persistence.StoredProcedureQuery>

<https://stackoverflow.com/questions/46786528/error-in-namedstoredprocedurequery-in-spring-jpa-found-named-stored-procedure>

==========END==========

<https://docs.spring.io/spring-data/jpa/docs/2.3.2.RELEASE/reference/html/#reference>

<https://spring.io/guides/gs/accessing-data-jpa/>

Cascading docs and example:

<https://dzone.com/articles/beginner%E2%80%99s-guide-jpa-and>

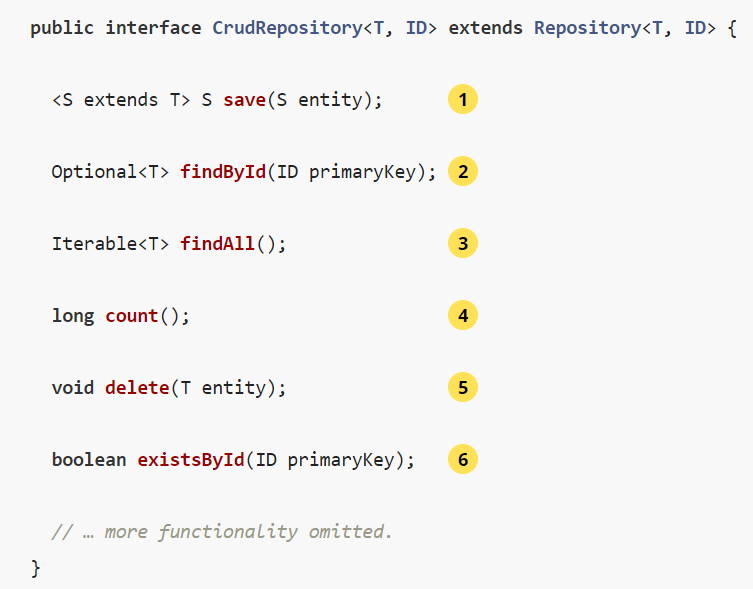
# Spring Data JPA

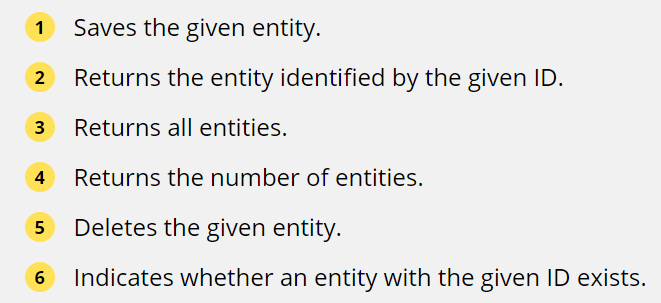
Spring Data JPA provides repository support for the Java Persistence API (JPA)

Spring Data modules require Spring Framework in version 5.2.8.RELEASE or better

## Spring Data Repositories

CrudRepository**interface**

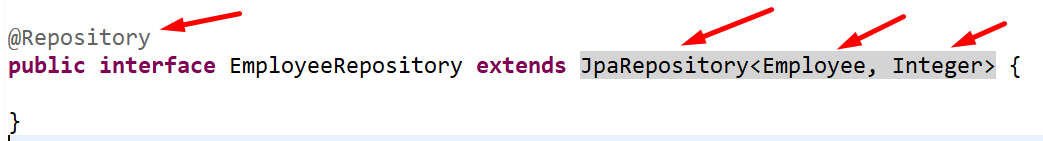




CrudRepository

PagingAndSortingRepository

JpaRepository



Spring Data JPA so powerful: You need not write an implementation of the repository interface. Spring Data JPA creates an implementation when you run the application.