|  |
| --- |
| **JPA Query API** |

|  |
| --- |
|  |

Once you have persisted objects you need to query them. For example if you have a web application representing an online store, the user asks to see all products of a particular type, ordered by the price. This requires you to query the datastore for these products. JPA specifies support for [a semi-object-oriented query language (JPQL)](http://www.datanucleus.org/products/accessplatform_1_1/jpa/jpql.html) and [a relational query language (SQL)](http://www.datanucleus.org/products/accessplatform_1_1/jpa/sql.html). DataNucleus supports the following

* [JPQL](http://www.datanucleus.org/products/accessplatform_1_1/jpa/jpql.html) - language based around the objects that are persisted **but**with syntax very similar to SQL
* [SQL](http://www.datanucleus.org/products/accessplatform_1_1/jpa/sql.html) - language found on almost all RDBMS, which JPA is targetted at.

Which query language is used is down to the developer. The data-tier of an application could be written by a primarily Java developer, who would typically think in an object-oriented way and so would likely prefer **JPQL**. On the other hand the data-tier could be written by a datastore developer who is more familiar with SQL concepts and so could easily make more use of **SQL**. This is the power of an implementation like DataNucleus in that it provides the flexibility for different people to develop the data-tier utilising their own skills to the full without having to learn totally new concepts.

There are 2 categories of queries with JPA :-

* [Named Query](http://www.datanucleus.org/products/accessplatform_1_1/jpa/named_query.html) where the query is defined in MetaData and referred to by its name at runtime.
* **Programmatic Query**where the query is defined using the JPA1 Query API.

Let's now try to understand the Query API in JPA [http://www.datanucleus.org/products/accessplatform_1_1/images/javadoc.gif](http://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html), We firstly need to look at a typical Query. We'll take 2 examples

|  |
| --- |
| **JPQL Query** |

|  |
| --- |
|  |

Let's create a JPQL query to highlight its usage

Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");

query.setParameter("threshold", my\_threshold);

List results = query.getResultList();

In this Query, we implicitly select JPQL by using the method *EntityManager.createQuery()*, and the query is specified to return all objects of type *Product*(or subclasses) which have the field *param2*less than some threshold value ordering the results by the value of field *param1*. We've specified the query like this because we want to pass the threshold value in as a parameter (so maybe running it once with one value, and once with a different value). We then set the parameter value of our *threshold*parameter. The Query is then executed to return a List of results. The example is to highlight the typical methods specified for a (JPQL) Query.

|  |
| --- |
| **SQL Query** |

|  |
| --- |
|  |

Let's create an SQL query to highlight its usage

Query query = em.createNativeQuery("SELECT \* FROM Product p WHERE p.param2 < ?1");

query.setParameter(1, my\_threshold);

List results = query.getResultList();

So we implicitly select SQL by using the method *EntityManager.createNativeQuery()*, and the query is specified like in the JPQL case to return all instances of type *Product*(using the table name in this SQL query) where the column *param2*is less than some threshold value.

|  |
| --- |
| **setFirstResult(), setMaxResults()** |

|  |
| --- |
|  |

In JPA to specify the range of a query you have two methods available. So you could do

Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");

query.setFirstResult(1);

query.setMaxResults(3);

so we will get results 1, 2, and 3 returned only. The first result starts at 0 by default.

|  |
| --- |
| **setHint()** |

|  |
| --- |
|  |

JPA's query API allows implementations to support extensions ("hints") and provides a simple interface for enabling the use of such extensions on queries.

q.setHint("extension\_name", value);

DataNucleus provides various extensions for different types of queries.

|  |
| --- |
| **setParameter()** |

|  |
| --- |
|  |

JPA's query API supports named and numbered parameters and provides method for setting the value of particular parameters. To set a named parameter, for example, you could do

Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");

q.setParameter("threshold", value);

To set a numbered parameter you could do

Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < ?1 ORDER BY p.param1 ascending");

q.setParameter(1, value);

Numbered parameters are numbered from 1.

|  |
| --- |
| **getResultList()** |

|  |
| --- |
|  |

To execute a JPA query you would typically call *getResultList*. This will return a List of results. This should not be called when the query is an "UPDATE"/"DELETE".

Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 < :threshold ORDER BY p.param1 ascending");

q.setParameter("threshold", value);

List results = q.getResultList();

|  |
| --- |
| **getSingleResult()** |

|  |
| --- |
|  |

To execute a JPA query where you are expecting a single value to be returned you would call *getSingleResult*. This will return the single Object. If the query returns more than one result then you will get an Exception. This should not be called when the query is an "UPDATE"/"DELETE".

Query query = em.createQuery("SELECT p FROM Product p WHERE p.param2 = :value");

q.setParameter("value", val1);

Product prod = q.getSingleResult();

|  |
| --- |
| **executeUpdate()** |

|  |
| --- |
|  |

To execute a JPA UPDATE/DELETE query you would call *executeUpdate*. This will return the number of objects changed by the call. This should not be called when the query is a "SELECT".

Query query = em.createQuery("DELETE FROM Product p");

int number = q.executeUpdate();

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | |  | | --- | | **JPA Named Queries** |  |  | | --- | |  |   http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif  With the JPA1 API you can either define a query at runtime, or define it in the MetaData/annotations for a class and refer to it at runtime using a symbolic name. This second option means that the method of invoking the query at runtime is much simplified. To demonstrate the process, lets say we have a class called *Product*(something to sell in a store). We define the JPA Meta-Data for the class in the normal way, but we also have some query that we know we will require, so we define the following in the Meta-Data.  <entity class="Product">  ...  <named-query name="SoldOut"><![CDATA[  SELECT p FROM Product p WHERE p.status == "Sold Out"  ]]></named-query>  </entity>  So we have a JPQL query called "SoldOut" defined for the class *Product*that returns all Products (and subclasses) that have a *status*of "Sold Out". Out of interest, what we would then do in our application to execute this query woule be  Query query = em.createNamedQuery("SoldOut");  List results = query.getResultList();  The above example was for the JPQL query language. We can do a similar thing using SQL, so we define the following in our MetaData for our *Product*class  <entity class="Product">  ...  <named-native-query name="PriceBelowValue"><![CDATA[  SELECT NAME FROM PRODUCT WHERE PRICE < ?1  ]]></named-native-query>  </entity>  So here we have an SQL query that will return the names of all Products that have a price less than a specified value. This leaves us the flexibility to specify the value at runtime. So here we run our named query, asking for the names of all Products with price below 20 euros.  Query query = em.createNamedNativeQuery("PriceBelowValue");  query.setParameter(1, new Double(20.0));  List results = query.getResultList();  All of the examples above have been specifed within the <entity> element of the MetaData. You can also define these named queries in annotations in the class itself, but clearly that is polluting your model.  Please proceed to the sections specific to [**JPQL**](http://www.datanucleus.org/products/accessplatform_1_1/jpa/jpql.html)and [**SQL**](http://www.datanucleus.org/products/accessplatform_1_1/jpa/sql.html)for details on the precise nature of the query for the querying languages supported by DataNucleus with JPA. | | |
| **Query Caching** |

|  |
| --- |
|  |

JPA doesn't currently define a mechanism for caching of queries. DataNucleus provides the following level of caching

* [Generic Compilation](http://www.datanucleus.org/products/accessplatform_1_1/jpa/query_cache.html#genericCompilation) : when a query is compiled it is initially compiled *generically*into expression trees. This generic compilation is independent of the datastore in use, so can be used for other datastores. This can be cached.

|  |
| --- |
| **Generic Query Compilation Cache** |

|  |
| --- |
|  |

This cache is by default set to *weak*, meaning that the generic query compilation is cached using weak references. This is set using the persistence property**datanucleus.cache.query.type**. You can also set it to *hard*meaning that strong references are used, or *soft*meaning that soft references are used, or finally to *none*meaning that there is no caching of generic query compilation information

You can turn caching on/off (default = on) on a query-by-query basis by specifying the query extension **datanucleus.query.cached**as true/false.

|  |
| --- |
| **JPQL SELECT Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif

The JPA specification defines JPQL, for selecting objects from the datastore. To provide a simple example, this is what you would do

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = 'Jones'");

List results = (List)q.getResultsList();

This finds all "Person" objects with surname of "Jones". You specify all details in the query.

|  |
| --- |
| **SELECT Syntax** |

|  |
| --- |
|  |

In JPQL queries you define the query in a single string, defining the result, the candidate class(es), the filter, any grouping, and the ordering. This string has to follow the following pattern

SELECT [<result>]

[FROM <candidate-class(es)>]

[WHERE <filter>]

[GROUP BY <grouping>]

[HAVING <having>]

[ORDER BY <ordering>]

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

|  |
| --- |
| **Entity Name** |

|  |
| --- |
|  |

In the example shown you note that we did not specify the full class name. We used *Person p*and thereafter could refer to *p*as the alias. The *Person*is called the **entity name**and in JPA MetaData this can be defined against each class in its definition. For example

<entity class="org.datanucleus.company.Person" name="Person">

...

</entity>

|  |
| --- |
| **Input Parameters** |

|  |
| --- |
|  |

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for different values. Let's take two examples

Named Parameters :

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :surname AND o.firstName = :forename");

q.setParameter("surname", theSurname);

q.setParameter("forename", theForename");

Numbered Parameters :

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND p.firstName = ?2");

q.setParameter(1, theSurname);

q.setParameter(2, theForename);

So in the first case we have parameters that are prefixed by **:**(colon) to identify them as a parameter and we use that name when calling *Query.setParameter()*. In the second case we have parameters that are prefixed by **?**(question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*.

|  |
| --- |
| **Range of Results** |

|  |
| --- |
|  |

With JPQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

Query q = em.createQuery("SELECT p FROM Person p WHERE p.age &gt; 20");

q.setFirstResult(0);

q.setMaxResults(20);

So with this query we get results 0 to 19 inclusive.

|  |
| --- |
| **Query Execution** |

|  |
| --- |
|  |

There are two ways to execute a JPQL query. When you know it will return 0 or 1 results you call

Object result = query.getSingleResult();

If however you know that the query will return multiple results, or you just don't know then you would call

List results = query.getResultList();

|  |
| --- |
| **JPQL DELETE Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif

The JPA specification defines a mode of JPQL for deleting objects from the datastore. DataNucleus AccessPlatform supports these for db4o, Excel, LDAP, NeoDatis and XML datastores.

|  |
| --- |
| **DELETE Syntax** |

|  |
| --- |
|  |

The syntax for deleting records is very similar to selecting them

DELETE FROM [<candidate-class>]

[WHERE <filter>]

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

|  |
| --- |
| **JPQL UPDATE Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif

The JPA specification defines a mode of JPQL for updating objects in the datastore. DataNucleus doesnt currently support this but will soon.

|  |
| --- |
| **UPDATE Syntax** |

|  |
| --- |
|  |

The syntax for updating records is very similar to selecting them

UPDATE [<candidate-class>] SET item1=value1, item2=value2

[WHERE <filter>]

The "keywords" in the query are shown in UPPER CASE are case-insensitive.

|  |
| --- |
| **JPQL Subqueries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif

With JPQL the user has a very flexible query syntax which allows for querying of the vast majority of data components in a single query. In some situations it is desirable for the query to utilise the results of a separate query in its calculations. JPQL also allows the use of subqueries. Here's an example

SELECT Object(e) FROM org.datanucleus.Employee e

WHERE e.salary > (SELECT avg(f.salary) FROM org.datanucleus.Employee f)

So we want to find all Employees that have a salary greater than the average salary. The subquery must be in parentheses (brackets). Note that we have defined the subquery with an alias of "f", whereas in the outer query the alias is "e".

|  |
| --- |
| **All or Any Expressions** |

|  |
| --- |
|  |

One use of subqueries with JPQL is where you want to compare with some or all of a particular expression. To give an example

SELECT emp FROM Employee emp

WHERE emp.salary > ALL (SELECT m.salary FROM Manager m WHERE m.department = emp.department)

So this returns all employees that earn more than all managers in the same department! You can also compare with some/any, like this

SELECT emp FROM Employee emp

WHERE emp.salary > ANY (SELECT m.salary FROM Manager m WHERE m.department = emp.department)

So this returns all employees that earn more than any one Manager in the same department.

|  |
| --- |
| **Existence Expressions** |

|  |
| --- |
|  |

Another use of subqueries in JPQL is where you want to check on the existence of a particular thing. For example

SELECT DISTINCT emp FROM Employee emp

WHERE EXISTS (SELECT emp2 FROM Employee emp2 WHERE emp2 = emp.spouse)

So this returns the employees that have a partner also employed.

|  |
| --- |
| **JPQL : In-Memory queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/nucleus_extension.gif

The typical use of a JPQL query is to translate it into the native query language of the datastore and return objects matched by the query. For many datastores it is simply impossible to support the full JPQL syntax in the datastore *native query language*and so it is necessary to evaluate the query in-memory. This means that we evaluate as much as we can in the datastore and then instantiate those objects and evaluate further in-memory. Here we document the current capabilities of *in-memory evaluation*in DataNucleus.

* Query methods **CONTAINS(Collection)**, **CONTAINS(Map)**are not currently supported
* Subqueries using ALL, ANY, SOME, EXISTS are not currently supported
* MEMBER OF syntax is not currently supported.

To enable evaluation in memory you specify the query hint **datanucleus.query.evaluateInMemory**to *true*as follows

query.setHint("datanucleus.query.evaluateInMemory","true");

|  |
| --- |
| **SQL Queries** |

|  |
| --- |
|  |

http://www.datanucleus.org/products/accessplatform_1_1/images/jpa1.gif

The JPA specification defines its interpretation of SQL, for selecting objects from the datastore. To provide a simple example, this is what you would do

Query q = em.createNativeQuery("SELECT p.id, o.firstName, o.lastName FROM Person p, Job j " +

"WHERE (p.job = j.id) AND j.name = 'Cleaner'");

List results = (List)q.getResultsList();

This finds all "Person" objects that do the job of "Cleaner". The syntax chosen has to be runnable on the RDBMS that you are using (and since SQL is anything but "standard" you will likely have to change your query when moving to another datastore).

|  |
| --- |
| **Input Parameters** |

|  |
| --- |
|  |

In JPQL queries it is convenient to pass in parameters so we dont have to define the same query for different values. Here's an example

Numbered Parameters :

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = ?1 AND p.firstName = ?2");

q.setParameter(1, theSurname);

q.setParameter(2, theForename);

So we have parameters that are prefixed by **?**(question mark) and are numbered starting at 1. We then use the numbered position when calling *Query.setParameter()*. With SQL queries we can't use named parameters. This is known as *numbered*parameters.

DataNucleus also supports use of *named*parameters where you assign names just like in JPQL. This is not defined by the JPA specification so dont expect other JPA implementations to support it. Let's take the previous example and rewrite it using *named*parameters, like this

Named Parameters :

Query q = em.createQuery("SELECT p FROM Person p WHERE p.lastName = :firstParam AND p.firstName = :otherParam");

q.setParameter("firstParam", theSurname);

q.setParameter("otherParam", theForename);

|  |
| --- |
| **Range of Results** |

|  |
| --- |
|  |

With SQL you can select the range of results to be returned. For example if you have a web page and you are paginating the results of some search, you may want to get the results from a query in blocks of 20 say, with results 0 to 19 on the first page, then 20 to 39, etc. You can facilitate this as follows

Query q = em.createNativeQuery("SELECT p FROM Person p WHERE p.age &gt; 20");

q.setFirstResult(0);

q.setMaxResults(20);

So with this query we get results 0 to 19 inclusive.

|  |
| --- |
| **Query Execution** |

|  |
| --- |
|  |

There are two ways to execute a SQL query. When you know it will return 0 or 1 results you call

Object result = query.getSingleResult();

If however you know that the query will return multiple results, or you just don't know then you would call

List results = query.getResultList();