JPA Query API

Queries are represented in JPA 2 by two interfaces - the old `Query` interface, which was the only interface available for representing queries in JPA 1, and the new `TypedQuery` interface that was introduced in JPA 2. The `TypedQuery` interface extends the `Query` interface.

In JPA 2 the `Query` interface should be used mainly when the query result type is unknown or when a query returns polymorphic results and the lowest known common denominator of all the result objects is `Object`. When a more specific result type is expected queries should usually use the `TypedQuery` interface. It is easier to run queries and process the query results in a type safe manner when using the `TypedQuery` interface.

Building Queries with createQuery

As with most other operations in JPA, using queries starts with an `EntityManager` (represented by `em` in the following code snippets), which serves as a factory for both `Query` and `TypedQuery`:

```
Query q1 = em.createQuery("SELECT c FROM Country c");

TypedQuery<Country> q2 =
    em.createQuery("SELECT c FROM Country c", Country.class);
```

In the above code, the same JPQL query which retrieves all the `Country` objects in the database is represented by both `q1` and `q2`. When building a `TypedQuery` instance the expected result type has to be passed as an additional argument, as demonstrated for `q2`. Because, in this case, the result type is known (the query returns only `Country` objects), a `TypedQuery` is preferred.

There is another advantage of using `TypedQuery` in ObjectDB. In the context of the queries above, if there are no Country instances in the database yet and the `Country` class is unknown as a managed entity class - only the `TypedQuery` variant is valid because it introduces the `Country` class to ObjectDB.

Dynamic JPQL, Criteria API and Named Queries

Building queries by passing JPQL query strings directly to the `createQuery` method, as shown above, is referred to in JPA as dynamic query construction because the query string can be built dynamically at runtime.

The JPA Criteria API provides an alternative way for building dynamic queries, based on Java objects that represent query elements (replacing string based JPQL).

JPA also provides a way for building static queries, as named queries, using the `@NamedQuery`and `@NamedQueries` annotations. It is considered to be a good practice in JPA to prefer named queries over dynamic queries when possible.

Organization of this Section

The following pages explain how to define and execute queries in JPA:

- Running JPA Queries
- Query Parameters in JPA
- JPA Named Queries
- JPA Criteria API Queries
- Setting and Tuning of JPA Queries

In addition, the syntax of the JPA Query Language (JPQL) is described in:

- JPA Query Structure (JPQL / Criteria)
- JPA Query Expressions (JPQL / Criteria)

JPA Named Queries

A named query is a statically defined query with a predefined unchangeable query string. Using named queries instead of dynamic queries may improve code organization by separating the JPQL query strings from the Java code. It also enforces the use of query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

This page covers the following topics:

@NamedQuery and @NamedQueries Annotations

The following `@NamedQuery` annotation defines a query whose name is `"Country.findAll"` that retrieves all the `Country` objects in the database:

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
```

The `@NamedQuery` annotation contains four elements - two of which are required and two are optional. The two required elements, `name` and `query` define the name of the query and the query string itself and are demonstrated above. The two optional elements, `lockMode` and `hints`, provide static replacement for the `setLockMode` and `setHint` methods.

Every `@NamedQuery` annotation is attached to exactly one entity class or mapped superclass - usually to the most relevant entity class. But since the scope of named queries is the entire persistence unit, names should be selected carefully to avoid collision (e.g. by using the unique entity name as a prefix).

It makes sense to add the above `@NamedQuery` to the `Country` entity class:

```
@Entity
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
public class Country {
  ...
}
```

Attaching multiple named queries to the same entity class requires wrapping them in a `@NamedQueries` annotation, as follows:

```
@Entity
@NamedQueries({
    @NamedQuery(name="Country.findAll",
                query="SELECT c FROM Country c"),
    @NamedQuery(name="Country.findByName",
                query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
  ...
}
```

Note: Named queries can be defined in JPA XML mapping files instead of using the `@NamedQuery`annotation. ObjectDB supports JPA XML mapping files, including the definition of named queries. But, because mapping files are useful mainly for Object Relational Mapping (ORM) JPA providers and less so when using ObjectDB, this alternative is not covered in this manual.

Using Named Queries at Runtime

Named queries are represented at runtime by the same `Query` and `TypedQuery` interfaces but different `EntityManager` factory methods are used to instantiate them. The `createNamedQuery`method receives a query name and a result type and returns a `TypedQuery` instance:

```
  TypedQuery<Country> query =
      em.createNamedQuery("Country.findAll", Country.class);
  List<Country> results = query.getResultList();
```

Another form of `createNamedQuery` receives a query name and returns a `Query` instance:

```
  Query query = em.createNamedQuery("SELECT c FROM Country c");
```

```
List results = query.getResultList();
```

One of the reasons that JPA requires the listing of managed classes in a persistence unitdefinition is to support named queries. Notice that named queries may be attached to any entity class or mapped superclass. Therefore, to be able to always locate any named query at runtime a list of all these managed persistable classes must be available.

ObjectDB makes the definition of a persistence unit optional. Named queries are automatically searched for in all the managed classes that ObjectDB is aware of, and that includes all the entity classes that have objects in the database. However, an attempt to use a named query still might fail if that named query is defined on a class that is still unknown to ObjectDB.

As a workaround, you may introduce classes to ObjectDB before accessing named queries, by using the JPA 2 `Metamodel` interface. For example:

```
em.getMetamodel().managedType(MyEntity.class);
```

Following the above code ObjectDB will include `MyEntity` in searching named queries.

JPA Criteria API Queries

The JPA Criteria API provides an alternative way for defining JPA queries, which is mainly useful for building dynamic queries whose exact structure is only known at runtime.

This page covers the following topics:

JPA Criteria API vs JPQL

JPQL queries are defined as strings, similarly to SQL. JPA criteria queries, on the other hand, are defined by instantiation of Java objects that represent query elements.

A major advantage of using the criteria API is that errors can be detected earlier, during compilation rather than at runtime. On the other hand, for many developers string based JPQL queries, which are very similar to SQL queries, are easier to use and understand.

For simple static queries - string based JPQL queries (e.g. as named queries) may be preferred. For dynamic queries that are built at runtime - the criteria API may be preferred.

For example, building a dynamic query based on fields that a user fills at runtime in a form that contains many optional fields - is expected to be cleaner when using the JPA criteria API, because it eliminates the need for building the query using many string concatenation operations.

String based JPQL queries and JPA criteria based queries are equivalent in power and in efficiency. Therefore, choosing one method over the other is also a matter of personal preference.

First JPA Criteria Query

The following query string represents a minimal JPQL query:

```
SELECT c FROM Country c
```

An equivalent query can be built using the JPA criteria API as follows:

```
  CriteriaBuilder cb = em.getCriteriaBuilder();

  CriteriaQuery<Country> q = cb.createQuery(Country.class);
  Root<Country> c = q.from(Country.class);
  q.select(c);
```

The `CriteriaBuilder` interface serves as the main factory of criteria queries and criteria query elements. It can be obtained either by the `EntityManagerFactory`'s `getCriteriaBuilder`method or by the `EntityManager`'s `getCriteriaBuilder` method (both methods are equivalent).

In the example above a `CriteriaQuery` instance is created for representing the built query.
Then a `Root` instance is created to define a range variable in the FROM clause. Finally, the range variable, `c`, is also used in the SELECT clause as the query result expression.

A `CriteriaQuery` instance is equivalent to a JPQL string and not to a `TypedQuery` instance.
Therefore, running the query still requires a `TypedQuery` instance:

```
  TypedQuery<Country> query = em.createQuery(q);
  List<Country> results = query.getResultList();
```

Using the criteria API introduces some extra work, at least for simple static queries, since the equivalent JPQL query could simply be executed as follows:

```
  TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Because eventually both types of queries are represented by a `TypedQuery` instance - query execution and query setting is similar, regardless of the way in which the query is built.

Parameters in Criteria Queries

The following query string represents a JPQL query with a parameter:

```
SELECT c FROM Country c WHERE c.population > :p
```

An equivalent query can be built using the JPA criteria API as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
ParameterExpression<Integer> p = cb.parameter(Integer.class);
q.select(c).where(cb.gt(c.get("population"), p));
```

The `ParameterExpression` instance, `p`, is created to represent the query parameter. The wheremethod sets the WHERE clause. As shown above, The `CriteriaQuery` interface supports method chaining. See the links in the next sections of this page for detailed explanations on how to set criteria query clauses and build criteria expressions.

Running this query requires setting the parameter:

```
TypedQuery<Country> query = em.createQuery(q);
query.setParameter(p, 10000000);
List<Country> results = query.getResultList();
```

The `setParameter` method takes a `Parameter` (or a `ParameterExpression`) instance as the first argument instead of a name or a position (which are used with string based JPQL parameters).

Criteria Query Structure

Queries in JPA (as in SQL) are composed of clauses. Because JPQL queries and criteria queries use equivalent clauses - they are explained side by side in the Query Structure pages.

Specific details about criteria query clauses are provided in the following page sections:

- SELECT clause (`select`, `distinct`, `multiselect`, `array`, `tuple`, `construct`).
- FROM clause (`from`, `join`, `fetch`).
- WHERE clause (`where`).
- GROUP BY / HAVING clauses (`groupBy`, `having`, `count`, `sum`, `avg`, `min`, `max`, ...).
- ORDER BY clause (`orderBy`, `Order`, `asc`, `desc`).

The links above are direct links to the criteria query sections in pages that describe query structure in general, including in the context of string based JPQL queries.

Criteria Query Expressions

JPA query clauses are composed of expressions. Because JPQL queries and criteria queries use equivalent expressions - they are explained side by side in the Query Expressions pages.

Specific details about criteria query expressions are provided in the following page sections:

- Literals and Dates (`literal`, `nullLiteral`, `currentDate`, ...).
- Paths, navigation and types (`get`, `type`).

type()
Path's method
Create an expression corresponding to the type of the path.
See JavaDoc Reference Page...

- Arit d, `quot`, `mod`, `abs`, `neg`, `sqrt`).
- Stri te, `lower`, `upper`, `concat`, `substring`, ...).
- Collection expressions (`isEmpty`, `isNotEmpty`, `isMember`, `isNotMember`, `size`).
- Comparison expressions (`equal`, `notEqual`, `gt`, `ge`, `lt`, `le`, `between`, `isNull`, ...)
- Logical expressions (`and`, `or`, `not`, `isTrue`).

The links above are direct links to the criteria query sections in pages that describe expressions in general, including in the context of string based JPQL queries.

Running JPA Queries

The `Query` interface defines two methods for running SELECT queries:

- `Query.getSingleResult` - for use when exactly one result object is expected.
- `Query.getResultList` - for general use in any other case.

Similarly, the `TypedQuery` interface defines the following methods:

- `TypedQuery.getSingleResult` - for use when exactly one result object is expected.
- `TypedQuery.getResultList` - for general use in any other case.

In addition, the Query interface defines a method for running DELETE and UPDATE queries:

- `Query.executeUpdate` - for running only DELETE and UPDATE queries.

This page covers the following topics:

Ordinary Query Execution (with getResultList)

The following query retrieves all the Country objects in the database. Because multiple result objects are expected, the query should be run using the getResultList method:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Both `Query` and `TypedQuery` define a `getResultList` method, but the version of `Query` returns a result list of a raw type (non generic) instead of a parameterized (generic) type:

```
Query query = em.createQuery("SELECT c FROM Country c");
List results = query.getResultList();
```

An attempt to cast the above `results` to a parameterized type (`List<Country>`) will cause a compilation warning. If, however, the new `TypedQuery` interface is used casting is unnecessary and the warning is avoided.

The query result collection functions as any other ordinary Java collection. A result collection of a parameterized type can be iterated easily using an enhanced for loop:

```
for (Country c : results) {
    System.out.println(c.getName());
}
```

Note that for merely printing the country names, a query that uses projection and retrieves country names directly instead of fully built Country instances would be more efficient.

Single Result Query Execution (with getSingleResult)

The `getResultList` method (which was discussed above) can also be used to run queries that return a single result object. In this case, the result object has to be extracted from the result collection after query execution (e.g. by `results.get(0)`). To eliminate this routine operation JPA provides an additional method, `getSingleResult`, as a more convenient method when exactly one result object is expected.

The following aggregate query always returns a single result object, which is a `Long` object reflecting the number of `Country` objects in the database:

```
TypedQuery<Long> query = em.createQuery(
    "SELECT COUNT(c) FROM Country c", Long.class);
long countryCount = query.getSingleResult();
```

Notice that when a query returns a single object it might be tempting to prefer Query overTypedQuery even when the result type is known because the casting of a single object is easy and the code is simple:

```
Query query = em.createQuery("SELECT COUNT(c) FROM Country c");
long countryCount = (Long)query.getSingleResult();
```

An aggregate COUNT query always returns one result, by definition. In other cases our expectation for a single object result might fail, depending on the database content. For example, the following query is expected to return a single Country object:

```
Query query = em.createQuery(
    "SELECT c FROM Country c WHERE c.name = 'Canada'");
Country c = (Country)query.getSingleResult();
```

However, the correctness of this assumption depends on the content of the database. If the database contains multiple `Country` objects with the name 'Canada' (e.g. due to a bug) aNonUniqueResultException is thrown. On the other hand, if there are no results at all aNoResultException is thrown. Therefore, using getSingleResult requires some caution and if there is any chance that these exceptions might be thrown they have to be caught and handled.

DELETE and UPDATE Query Execution (with executeUpdate)

DELETE and UPDATE queries are executed using the `executeUpdate` method.

For example, the following query deletes all the `Country` instances:

```
int count = em.createQuery("DELETE FROM Country").executeUpdate();
```

and the following query resets the area field in all the Country instances to zero:

```
int count = em.createQuery("UPDATE Country SET area = 0").executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.

On success - the executeUpdate method returns the number of objects that have been updated or deleted by the query.

The Query Structure section explains DELETE and UPDATE queries in more detail.

Query Parameters in JPA

Query parameters enable the definition of reusable queries. Such queries can be executed with different parameter values to retrieve different results. Running the same query multiple times with different parameter values (arguments) is more efficient than using a new query string for every query execution, because it eliminates the need for repeated query compilations.

This page covers the following topics:

Named Parameters (:name)

The following method retrieves a `Country` object from the database by its name:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = :name", Country.class);
  return query.setParameter("name", name).getSingleResult();
}
```

The WHERE clause reduces the query results to `Country` objects whose name field value is equal to `:name`, which is a parameter that serves as a placeholder for a real value. Before the query can be executed a parameter value has to be set using the `setParameter` method. The `setParameter` method supports method chaining (by returning the same `TypedQuery` instance on which it was invoked), so invocation of `getSingleResult` can be chained to the same expression.

Named parameters can be easily identified in a query string by their special form, which is a colon (:) followed by a valid JPQL identifier that serves as the parameter name. JPA does not provide an API for defining the parameters explicitly (except when using criteria API), so query parameters are defined implicitly by appearing in the query string. The parameter type is inferred by the context. In the above example, a comparison of `:name` to a field whose type is `String`indicates that the type of `:name` itself is `String`.

Queries can include multiple parameters and each parameter can have one or more occurrences in the query string. A query can be run only after setting values for all its parameters (in no matter in which order).

Ordinal Parameters (?index)

In addition to named parameter, whose form is `:name`, JPQL supports also ordinal parameter, whose form is `?index`. The following method is equivalent to the method above, except that an ordinal parameter replaces the named parameter:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = ?1", Country.class);
  return query.setParameter(1, name).getSingleResult();
}
```

The form of ordinal parameters is a question mark (?) followed by a positive int number. Besides the notation difference, named parameters and ordinal parameters are identical.

Named parameters can provide added value to the clarity of the query string (assuming that meaningful names are selected). Therefore, they are preferred over ordinal parameters.

Criteria Query Parameters

In a JPA query that is built by using the JPA Criteria API - parameters (as other query elements) are represented by objects (of type `ParameterExpression` or its super interface `Parameter`) rather than by names or numbers.

See the Parameters in Criteria Queries section for more details.

Parameters vs. Literals

Following is a third version of the same method. This time without parameters:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = '" + name + "'",
      Country.class);
  return query.getSingleResult();
}
```

Instead of using a parameter for the queried name the new method embeds the name as a `String` literal. There are a few drawbacks to using literals rather than parameters in queries.

First, the query is not reusable. Different literal values lead to different query strings and each query string requires its own query compilation, which is very inefficient. On the other hand, when using parameters, even if a new `TypedQuery` instance is constructed on every query execution, ObjectDB can identify repeating queries with the same query string and use a cached compiled query program, if available.

Second, embedding strings in queries is unsafe and can expose the application to JPQL injection attacks. Suppose that the name parameter is received as an input from the user and then embedded in the query string as is. Instead of a simple country name, a malicious user may provide JPQL expressions that change the query and may help in hacking the system.

In addition, parameters are more flexible and support elements that are unavailable as literals, such as entity objects.

API Parameter Methods

Over half of the methods in `Query` and `TypedQuery` deal with parameter handling. The `Query`interface defines 18 such methods, 9 of which are overridden in `TypedQuery`. That large number of methods is not typical to JPA, which generally excels in its thin and simple API.

There are 9 methods for setting parameters in a query, which is essential whenever using query parameters. In addition, there are 9 methods for extracting parameter values from a query. These get methods, which are new in JPA 2, are expected to be much less commonly used than the set methods.

Two set methods are demonstrated above - one for setting a named parameter and the other for setting an ordinal parameter. A third method is designated for setting a parameter in a Criteria API query. The reason for having nine set methods rather than just three is that JPA additionally provides three separate methods for setting `Date` parameters as well as three separate methods for setting `Calendar` parameters.

`Date` and `Calendar` parameter values require special methods in order to specify what they represent, such as a pure date, a pure time or a combination of date and time, as explained in detail in the Date and Time (Temporal) Types section.

For example, the following invocation passes a `Date` object as a pure date (no time):

```
query.setParameter("date", new java.util.Date(), TemporalType.DATE);
```

Since `TemporalType.Date` represents a pure date, the time part of the newly constructed `java.util.Date` instance is discarded. This is very useful in comparison against a specific date, when time should be ignored.

The get methods support different ways to extract parameters and their values from a query, including by name (for named parameter), by position (for ordinal parameters) by Parameter object (for Criteria API queries), each with or without an expected type. There is also a method for extracting all the parameters as a set (`getParameters`) and a method for checking if a specified parameter has a value (`isBound`). These methods are not required for running queries and are expected to be less commonly used.

JPA Query Structure (JPQL / Criteria)

The syntax of the Java Persistence Query Language (JPQL) is very similar to the syntax of SQL. Having a SQL like syntax in JPA queries is an important advantage because SQL is a very powerful query language and many developers are already familiar with it.

The main difference between SQL and JPQL is that SQL works with relational database tables, records and fields, whereas JPQL works with Java classes and objects. For example, a JPQL query can retrieve and return entity objects rather than just field values from database tables, as with SQL. That makes JPQL more object oriented friendly and easier to use in Java.

JPQL Query Structure

As with SQL, a JPQL SELECT query also consists of up to 6 clauses in the following format:

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

The first two clauses, SELECT and FROM are required in every retrieval query (update and delete queries have a slightly different form). The other JPQL clauses, WHERE, GROUP BY, HAVING andORDER BY are optional.

The structure of JPQL DELETE and UPDATE queries is simpler:

```
DELETE FROM ... [WHERE ...]

UPDATE ... SET ... [WHERE ...]
```

Besides a few exceptions, JPQL is case insensitive. JPQL keywords, for example, can appear in queries either in upper case (e.g. SELECT) or in lower case (e.g. select). The few exceptions in which JPQL is case sensitive include mainly Java source elements such as names of entity classes and persistent fields, which are case sensitive. In addition, string literals are also case sensitive (e.g. "ORM" and "orm" are different values).

A Minimal JPQL Query

The following query retrieves all the `Country` objects in the database:

```
SELECT c FROM Country AS c
```

Because SELECT and FROM are mandatory, this demonstrates a minimal JPQL query.

The FROM clause declares one or more query variables (also known as identification variables). Query variables are similar to loop variables in programing languages. Each query variable represents iteration over objects in the database. A query variable that is bound to an entity class is referred to as a range variable. Range variables define iteration over all the database objects of a binding entity class and its descendant classes. In the query above, `c` is a range variable that is bound to the `Country` entity class and defines iteration over all the `Country`objects in the database.

The SELECT clause defines the query results. The query above simply returns all the `Country`objects from the iteration of the c range variable, which in this case is actually all the `Country`objects in the database.

Organization of this Section

This section contains the following pages:

Detailed explanations on how to set criteria query clauses are provided as follows:

- Criteria SELECT (`select`, `distinct`, `multiselect`, `array`, `tuple`, `construct`).
- Criteria FROM (`from`, `join`, `fetch`).
- Criteria WHERE (`where`).
- Criteria GROUP BY / HAVING (`groupBy`, `having`, `count`, `sum`, `avg`, `min`, `max`, ...).
- Criteria ORDER BY (`orderBy`, `Order`, `asc`, `desc`).

DELETE Queries in JPA/JPQL

As explained in chapter 2, entity objects can be deleted from the database by:

- Retrieving the entity objects into an `EntityManager`.
- Removing these objects from the `EntityManager` within an active transaction, either explicitly by calling the `remove` method or implicitly by a cascading operation.
- Applying changes to the database by calling the `commit` method.

JPQL DELETE queries provide an alternative way for deleting entity objects. Unlike `SELECT`queries, which are used to retrieve data from the database, DELETE queries do not retrieve data from the database, but when executed, delete specified entity objects from the database.

Removing entity objects from the database using a DELETE query may be slightly more efficient than retrieving entity objects and then removing them, but it should be used cautiously because bypassing the `EntityManager` may break its synchronization with the database. For example, the `EntityManager` may not be aware that a cached entity object in its persistence context has been removed from the database by a DELETE query. Therefore, it is a good practice to use a separate `EntityManager` for DELETE queries.

As with any operation that modifies the database, DELETE queries can only be executed within an active transaction and the changes are visible to other users (which use other `EntityManager`instances) only after `commit`.

This page covers the following topics:

Delete All Queries

The simplest form of a DELETE query removes all the instances of a specified entity class (including instances of subclasses) from the database.

For example, the following three equivalent queries delete all the `Country` instances:

```
DELETE FROM Country       // no variable
DELETE FROM Country c     // an optional variable
DELETE FROM Country AS c  // AS + an optional variable
```

ObjectDB supports using the `java.lang.Object` class in queries (as an extension to JPA), so the following query can be used to delete all the objects in the database:

```
DELETE FROM Object
```

DELETE queries are executed using the `executeUpdate` method:

```
  int deletedCount = em.createQuery("DELETE FROM Country").executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.

On success - the `executeUpdate` method returns the number of objects that have been deleted by the query.

Selective Deletion

The structure of DELETE queries is very simple relative to the structure of SELECT queries. DELETE queries cannot include multiple variables and JOIN, and cannot include the GROUP BY, HAVING and ORDER BY clauses.

A WHERE clause, which is essential for removing selected entity objects, is supported.

For example, the following query deletes the countries with population size that is smaller than a specified limit:

```
DELETE FROM Country c WHERE c.population < :p
```

The query can be executed as follows:

```
  Query query = em.createQuery(
    "DELETE FROM Country c WHERE c.population < :p");
int deletedCount = query.setParameter(p, 100000).executeUpdate();
```

Setting and Tuning of JPA Queries

The `Query` and `TypedQuery` interfaces define various setting and tuning methods that may affectquery execution if invoked before a query is run using getResultList or getSingleResult.

This page covers the following topics:

Result Range (setFirstResult, setMaxResults)

The `setFirstResult` and `setMaxResults` methods enable defining a result window that exposes a portion of a large query result list (hiding anything outside that window). The `setFirstResult`method is used to specify where the result window begins, i.e. how many results at the beginning of the complete result list should be skipped and ignored.
The `setMaxResults` method is used to specify the result window size. Any result after hitting that specified maximum is ignored.

These methods support the implementation of efficient result paging. For example, if each result page should show exactly `pageSize` results, and `pageId` represents the result page number (`0`for the first page), the following expression retrieves the results for a specified page:

```
List<Country> results =
    query.setFirstResult(pageIx * pageSize)
         .setMaxResults(pageSize)
         .getResultList();
```

These methods can be invoked in a single expression with `getResultList` since the setter methods in `Query` and `TypedQuery` support method chaining (by returning the query object on which they were invoked).

Flush Mode (setFlushMode)

Changes made to a database using an `EntityManager` `em` can be visible to anyone who uses `em`, even before committing the transaction (but not to users of other `EntityManager` instances). JPA implementations can easily make uncommitted changes visible in simple JPA operations, such as `find`. However, query execution is much more complex. Therefore, before a query is executed, uncommitted database changes (if any) have to be flushed to the database in order to be visible to the query.

Flush policy in JPA is represented by the `FlushModeType` enum, which has two values:

- `AUTO` - changes are flushed before query execution and on commit/flush.
- `COMMIT` - changes are flushed only on explicit commit/flush.

In most JPA implementations the default is `AUTO`. In ObjectDB the default is `COMMIT` (which is more efficient). The default mode can be changed by the application, either at the `EntityManager` level as a default for all the queries in that `EntityManager` or at the level of a specific query, by overriding the default `EntityManager` setting:

```
// Enable query time flush at the EntityManager level:
em.setFlushMode(FlushModeType.AUTO);

// Enable query time flush at the level of a specific query:
query.setFlushMode(FlushModeType.AUTO);
```

Flushing changes to the database before every query execution affects performance significantly. Therefore, when performance is important, this issue has to be considered.

Lock Mode (setLockMode)

ObjectDB uses automatic optimistic locking to prevent concurrent changes to entity objects by multiple users. JPA 2 adds support for pessimistic locking. The `setLockMode` method sets a lock mode that has to be applied on all the result objects that the query retrieves. For example, the following query execution sets a pessimistic WRITE lock on all the result objects:

```
List<Country> results =
        query.setLockMode(LockModeType.PESSIMISTIC_WRITE)
             .getResultList();
```

Notice that when a query is executed with a requested pessimistic lock mode it could fail if locking fails, throwing a `LockTimeoutException`.

Query Hints

Additional settings can be applied to queries via hints.

Supported Query Hints

ObjectDB supports the following query hints:

- `"javax.persistence.query.timeout"` - sets maximum query execution time in milliseconds. A `QueryTimeoutException` is thrown if timeout is exceeded.
- `"javax.persistence.lock.timeout"` - sets maximum waiting time for pessimistic locks, when pessimistic locking of query results is enabled. See the Lock Timeout section for more details about lock timeout.
- `"objectdb.query-language"` - sets the query language, as one of `"JPQL"` (JPA query language), `"JDOQL"` (JDO query language) or `"ODBQL"` (ObjectDB query language). The default is ODBQL, which is a union of JPQL, JDOQL and ObjectDB extensions. Setting `"JPQL"`is useful to enforce portable JPA code by ObjectDB.
- `"objectdb.result-fetch"` - sets fetch mode for query result as either `"EAGER"` (the default) or `"LAZY"`. When LAZY is used result entity objects are returned as references (with no content). This could be useful when the shared L2 cache is enabled and entity objects may already be available in the cache.

Setting Query Hint (Scopes)

Query hints can be set in the following scopes (from global to local):

For the entire persistence unit - using a `persistence.xml` property:

```
     <properties>
        <property name="javax.persistence.query.timeout" value="3000"/>
     </properties>
```

For an `EntityManagerFactory` - using the `createEntityManagerFacotory` method:

```
  Map<String,Object> properties = new HashMap();
  properties.put("javax.persistence.query.timeout", 4000);
  EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("pu", properties);
```

For an `EntityManager` - using the `createEntityManager` method:

```
  Map<String,Object> properties = new HashMap();
  properties.put("javax.persistence.query.timeout", 5000);
  EntityManager em = emf.createEntityManager(properties);
```

or using the `setProperty` method:

```
  em.setProperty("javax.persistence.query.timeout", 6000);
```

For a named query definition - using the `hints` element:

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c",
    hints={@QueryHint(name="javax.persistence.query.timeout", value="7000")})
```

For a specific query execution - using the `setHint` method (before query execution):

```
  query.setHint("javax.persistence.query.timeout", 8000);
```

A hint that is set in a global scope affects all the queries in that scope (unless it is overridden in a more local scope). For example, setting a query hint in an `EntityManager` affects all the queries that are created in that `EntityManager` (except queries with explicit setting of the same hint).

SELECT clause (JPQL / Criteria API)

The ability to retrieve managed entity objects is a major advantage of JPQL. For example, the following query returns `Country` objects that become managed by the `EntityManager` `em`:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Because the results are managed entity objects they have all the support that JPA provides for managed entity objects, including transparent navigation to other database objects, transparent update detection, support for delete, etc.

Query results are not limited to entity objects. JPA 2 adds the ability to use almost any valid JPQL expression in SELECT clauses. Specifying the required query results more precisely can improve performance and in some cases can also reduce the amount of Java code needed. Notice that query results must always be specified explicitly - JPQL does not support the "SELECT *" expression (which is commonly used in SQL).

This page covers the following topics:

Projection of Path Expressions

JPQL queries can also return results which are not entity objects. For example, the following query returns country names as `String` instances, rather than `Country` objects:

```
SELECT c.name FROM Country AS c
```

Using path expressions, such as `c.name`, in query results is referred to as projection. The field values are extracted from (or projected out of) entity objects to form the query results.

The results of the above query are received as a list of `String` values:

```
TypedQuery<String> query = em.createQuery(
    "SELECT c.name FROM Country AS c", String.class);
List<String> results = query.getResultList();
```

Only singular value path expressions can be used in the SELECT clause. Collection and map fields cannot be included in the results directly, but their content can be added to the SELECT clause by using a bound JOIN variable in the FROM clause.

Nested path expressions are also supported. For example, the following query retrieves the name of the capital city of a specified country:

```
SELECT c.capital.name FROM Country AS c WHERE c.name = :name
```

Because construction of managed entity objects has some overhead, queries that return non entity objects, as the two queries above, are usually more efficient. Such queries are useful mainly for displaying information efficiently. They are less productive with operations that update or delete entity objects, in which managed entity objects are needed.

Managed entity objects can, however, be returned from a query that uses projection when a result path expression resolves to an entity. For example, the following query returns a managed`City` entity object:

```
SELECT c.capital FROM Country AS c WHERE c.name = :name
```

Result expressions that represent anything but entity objects (e.g. values of system types and user defined embeddable objects) return as results value copies that are not associated with the containing entities. Therefore, embedded objects that are retrieved directly by a result path expression are not associated with an EntityManager and changes to them when a transaction is active are not propagated to the database.

Multiple SELECT Expressions

The SELECT clause may also define composite results:

```
SELECT c.name, c.capital.name FROM Country AS c
```

The result list of this query contains `Object[]` elements, one per result. The length of each result`Object[]` element is 2. The first array cell contains the country name (`c.name`) and the second array cell contains the capital city name (`c.capital.name`).

The following code demonstrates running this query and processing the results:

```
  TypedQuery<Object[]> query = em.createQuery(
      "SELECT c.name, c.capital.name FROM Country AS c", Object[].class);
  List<Object[]> results = query.getResultList();
  for (Object[] result : results) {
      System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
  }
```

As an alternative to representing compound results by `Object` arrays, JPA supports using custom result classes and result constructor expressions.

Result Classes (Constructor Expressions)

JPA supports wrapping JPQL query results with instances of custom result classes. This is mainly useful for queries with multiple SELECT expressions, where custom result objects can provide an object oriented alternative to representing results as `Object[]` elements.

The fully qualified name of the result class is specified in a NEW expression, as follows:

```
SELECT NEW example.CountryAndCapital(c.name, c.capital.name)
FROM Country AS c
```

This query is identical to the previous query above except that now the result list contains `CountryAndCapital` instances rather than `Object[]` elements.

The result class must have a compatible constructor that matches the SELECT result expressions, as follows:

```
package example;

public class CountryAndCapital {
    public String countryName;
    public String capitalName;

    public CountryAndCapital(String countryName, String capitalName) {
        this.countryName = countryName;
        this.capitalName = capitalName;
    }
}
```

The following code demonstrates running this query:

```
  String queryStr =
      "SELECT NEW example.CountryAndCapital(c.name, c.capital.name) " +
      "FROM Country AS c";
  TypedQuery<CountryAndCapital> query =
      em.createQuery(queryStr, CountryAndCapital.class);
  List<CountryAndCapital> results = query.getResultList();
```

Any class with a compatible constructor can be used as a result class. It could be a JPA managed class (e.g. an entity class) but it could also be a lightweight 'transfer' class that is only used for collecting and processing query results.

If an entity class is used as a result class, the result entity objects are created in the NEW state, which means that they are not managed. Such entity objects are missing the JPA functionality of managed entity objects (e.g. transparent navigation and transparent update detection), but they are more lightweight, they are built faster and they consume less memory.

SELECT DISTINCT

Queries that use projection may return duplicate results. For example, the following query may return the same currency more than once:

```
SELECT c.currency FROM Country AS c WHERE c.name LIKE 'I%'
```

Both Italy and Ireland (whose name starts with 'I') use Euro as their currency. Therefore, the query result list contains "Euro" more than once.

Duplicate results can be eliminated easily in JPQL by using the DISTINCT keyword:

```
SELECT DISTINCT c.currency FROM Country AS c WHERE c.name LIKE 'I%'
```

The only difference between SELECT and SELECT DISTINCT is that the later filters duplicate results. Filtering duplicate results might have some effect on performance, depending on the size of the query result list and other factors.

SELECT in Criteria Queries

The criteria query API provides several ways for setting the SELECT clause.

Single Selection

Setting a single expression SELECT clause is straightforward.

For example, the following JPQL query:

```
SELECT DISTINCT c.currency FROM Country c
```

can be built as a criteria query as follows:

```
  CriteriaQuery<Country> q = cb.createQuery(Country.class);
  Root<Country> c = q.from(Country.class);
  q.select(c.get("currency")).distinct(true);
```

The `select` method takes one argument of type `Selection` and sets it as the SELECT clause content (overriding previously set SELECT content if any). Every valid criteria API expression can be used as selection, because all the criteria API expressions are represented by a sub interface of `Selection` - `Expression` (and its descendant interfaces).

The distinct method can be used to eliminate duplicate results as demonstrated in the above code (using method chaining).

Multi Selection

The `Selection` interface is also a super interface of `CompoundSelection`, which represents multi selection (which is not a valid expression by its own and can be used only in the SELECT clause).

The `CriteriaBuilder` interface provides three factory methods for building `CompoundSelection`instances - `array`, `tuple` and `construct`.

CriteriaBuilder's array

The following JPQL query:

```
SELECT c.name, c.capital.name FROM Country c
```

can be defined using the criteria API as follows:

```
  CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);
  Root<Country> c = q.from(Country.class);
  q.select(cb.array(c.get("name"), c.get("capital").get("name")));
```

The `array` method builds a `CompoundSelection` instance, which represents results as arrays.

The following code demonstrates execution of the query and iteration over the results:

```
  List<Object[]> results = em.createQuery(q).getResultList();
  for (Object[] result : results) {
      System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
```

```
}
```

CriteriaBuilder's tuple

The Tuple interface can be used as a clean alternative to Object[]:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Country> c = q.from(Country.class);
q.select(cb.tuple(c.get("name"), c.get("capital").get("name")));
```

The tuple method builds a CompoundSelection instance, which represents Tuple results.

The following code demonstrates execution of the query and iteration over the results:

```
List<Tuple> results = em.createQuery(q).getResultList();
for (Tuple t : results) {
    System.out.println("Country: " + t.get(0)  + ", Capital: " + t.get(1));
}
```

The Tuple interface defines several other methods for accessing the result data.

CriteriaBuilder's construct

JPQL user defined result objects are also supported by the JPA criteria query API:

```
CriteriaQuery<CountryAndCapital> q = cb.createQuery(CountryAndCapital.class);
Root<Country> c = q.from(Country.class);
q.select(cb.construct(CountryAndCapital.class,
    c.get("name"), c.get("capital").get("name")));
```

The construct method builds a CompoundSelection instance, which represents results as instances of a user defined class (CountryAndCapital in the above example).

The following code demonstrates execution of the query:

```
List<CountryAndCapital> results = em.createQuery(q).getResultList();
```

As expected - the result objects are CountryAndCapital instances.

CriteriaQuery's multiselect

In the above examples, CompoundSelection instances were first built by a CriteriaBuilderfactory method and then passed to the CriteriaQuery's select method.

The CriteriaQuery interface provides a shortcut method - multiselect, which takes a variable number of arguments representing multiple selections, and builds a CompoundSelectioninstance based on the expected query results.

For example, the following invocation of multiselect:

```
q.multiselect(c.get("name"), c.get("capital").get("name"));
```

is equivalent to using select with one of the factory methods (array, tuple or construct) as demonstrated above.

The behavior of the multiselect method depends on the query result type (as set when CriteriaQuery is instantiated):

- For expected Object and Object[] result type - array is used.
- For expected Tuple result - tuple is used.
- For any other expected result type - construct is used.