How to use JPA criteria

How to use JPA criteria

Introduction

The purpose of this document is to describe the usage of the JPA Criteria which are replacing the previously used Hibernate Criteria.

Development Guide

Equivalent between JPA criteria and Hibernate Criteria

JPA Criteria	Hibernate Criteria
CriteriaBuilder	NONE
CriteriaQuery	Criteria
Root	NONE
Selection	Projection
Predicate	Criterion
javax.persistence.criteria.JoinType	org.hibernate.sql.JoinType
javax.persistence.criteria.Order	org.hibernate.criterion.Order
Use the **.fetch()** method	Object[][] fetchesMatrix
FunctionmapperId	NONE
FunctionmapperType	NONE
Subquery	DetachedCriteria

Declaration and explanation of the different elements

CriteriaBuilder

The CriteriaBuilder is the base of the JPA Criteria, in order to use it you need to call it with :

```
CriteriaBuilder builder = sessionFactory.getCurrentSession().getCriteriaBuilder();
```

CriteriaQuery

The CriteriaQuery is the query representation that will be sent and executed by the server.

You can declare it in two ways that will change its usage in the next steps :

OBJECT TYPED QUERY

In the case you only want to get all the fields of an object, you can indicate to the query the class you need.

```
CriteriaQuery<Class> query = builder.createQuery(Class);
```

TUPLE TYPED QUERY

In the case that you want to select only a certain number of fields or you are using a special type like BidoEquipmentSearch, you need to create a tupleQuery that will use a mapper that you'll need to create or use if it already exist.

```
CriteriaQuery<Tuple> tupleQuery = builder.createTupleQuery();
```

Root

The root represents the query **from** clause.

The declaration is the same whether the query is an Object or a Tuple typed query.

The root is mandatory to the functionment of the query

```
//Both are the same
Root<Class> root = query.from(Class);
//OR
Root<Class> rootTuple = queryTuple.from(Class);
//The alias is also mandatory and must always be "root"
root/rootTuple.alias("root");
```

Selection

The selections are the query **select** clause and can be used in two ways :

SINGLE SELECTION

```
Selection<?> selection = root.get("attribute");
//distinct is optional but it adds the distinct clause in the select clause
Selection<?> selectionDistinct = root.get("attribute").distinct(true);
```

MULTIPLE SELECTIONS

```
List<Selection<?>> selections = new ArrayList<>();
//With Selection field
Selection<?> selection = root.get("attribute");
selection.alias("attributeName");
selections.add(selection);
//Directly in the selections.add
selections.add(root.get("attribute").alias("attributeName"));
```

The use of an alias is mandatory so that the tuple can get the information of the attribute.

> **Important**
>
> In order to get an attribute's ID you need to get the object's ID first

```
Selection<?> selectionId = root.get("id").get("idAttribute");
```

> **Example**
>
> ```
> Selection<?> selectionBidoEquipmentId = root.get("id").get("equipmentCode");
> ```

Predicate

The predicates are the query **where** clause and can be used in two ways :

SINGLE PREDICATE

```
Predicate predicate = builder.isNotNull(root.get("attribute"));
```

MULTIPLE PREDICATES

```
List<Predicate> predicates = new ArrayList<>();
predicates.add(builder.isNull(root.get("attribute")));
```

THE DIFFERENT CONDITIONS

```
//Check if the attribute is equal to the value
builder.equal(root.get("attribute"),"value");
//Check if the attribute is not equal to the value
builder.notEqual(root.get("attribute"),"value");
//Check if the attribute is in the same form of string as the value
builder.like(root.get("attribute"),"value");
//Check if the attribute is not in the same form of string as the value
builder.notLike(root.get("attribute"),"value");
//Check if the subquery result exists
builder.exists(subquery);
//Check if the subquery result doesn't exists
builder.not(builder.exists(subquery));
//Check if the attribute is present in the value
builder.in(root.get("attribute")).value("value");
//Check if the attribute is not present in the value
builder.not(builder.in(root.get("attribute")).value("value"));
//Check if the attribute is greater than or equal to the number
builder.greaterThanOrEqualTo(root.get("attribute"),number);
//Check if the attribute is lesser than or equal to the number
builder.lessThanOrEqualTo(root.get("attribute"),number);
//Check if the attribute is between the value1 and the value2
builder.between(root.get("attribute"),value1,value2);
//Check if the attribute is null
builder.isNull(root.get("attribute"));
//Check if the attribute is not null
builder.isNotNull(root.get("attribute"));
//Add an and clause to the where
builder.and(predicate1,predicate2);
//Add an or clause to the where
builder.or(predicate1,predicate2);
```

Joins

The join represents the query **join** clause.

It is composed of two things and is used on the root.

JOINTYPE

The JoinType is used on the root **join** clause to specify the query join type (INNER OR LEFT).

Example

```
JoinType joinType = JoinType.LEFT/INNER;
```

ATTRIBUTE

The attribute is the value we are going to join on, it is the @ManyToOne or @OneToMany association present in the class choosen for the root.

Example

```
"bidoEquipments"
```

USAGE

In our case we use the **joinWithFrom** method present in the `JoinCriteriaJPAUtil.java` file to avoid repeated joins.

```
JoinCriteriaJPAUtil.joinWithFrom(root,"attribute",joinType);
```

Fetches

Fetches are used to retrieve data in output. The fetches in ***JPA Criteria*** only works if no select are specified.

The default join type on fetch is INNER.

Example

```
//If JoinType is inner
root.fetch(BIDO_EQUIPMENT_ATTRIBUTES);
//If JoinType is left
root.fetch(BIDO_EQUIPMENT_ATTRIBUTES, JoinType.LEFT);
//Also works with join
bidtEquipmentJoin.fetch(BIDT_WAREHOUSE);
//Can be chained
root.fetch(BIDT_EQUIPMENTS).fetch(BIDT_WAREHOUSE,JoinType.LEFT);
```

Order

Order represents the **order by** clause in the SQL request.

```
Order order = builder.asc/desc(root.get("attribute"));
Order[] orders = {order1,order2,....,orderX};
```

Subquery

Subqueries are used in some situation, they represent the subqueries in the SQL request.

```
Subquery<Class> subquery = query.subquery(Class);
Root<Class> rootSubquery = subquery.from(Class);
```

MapperId and MapperType

The mappers are functions present in the `DOMAIN/src/main/java/com/twomoro/aerowebb/domain/mapper` package, then each mapper is located in the corresponding model package (ex : bido, bire, bidm, bidt, bsde).

They are used to map a Tuple entity to the class we need.

MAPPERID

The mapperId field is necessary in every function call to the GenericDaoImpl like **paginateForSearchByCriteriaJPA** so that the **addIdRestrictionToQuery** method works.

Example

```
public static final Function<Tuple, BidoEquipmentId> BIDO_EQUIPMENT_ID_MAPPER = tuple ->
(BidoEquipmentId) tuple.get("id");
```

MAPPERTYPE

The mapperType field is used in case selections are present in the query so that we can map the tupleQuery to our result class.

If we do not use a tuple for select, ***Hibernate*** will try to find the corresponding constructor and will return an error, mapping the tuple with the fields selected of our entity resolve this problem.

Example

```
public static final Function<Tuple, BidoEquipmentSearch> FULL_MAPPER = tuple -> {
    BidoEquipmentSearch bidoEquipmentSearch = new BidoEquipmentSearch();
    bidoEquipmentSearch.setEquipmentCode((String) tuple.get(HelperBidoEquipmentId.EQUIPMENT_CODE));
    bidoEquipmentSearch.setSn((String) tuple.get(HelperBidoEquipment.SN));
    bidoEquipmentSearch.setEquipmentDesignation((String)
tuple.get(HelperBidoEquipment.EQUIPMENT_DESIGNATION));
    bidoEquipmentSearch.setEquipmentFunction((String)
tuple.get(HelperBidoEquipment.EQUIPMENT_FUNCTION));
    bidoEquipmentSearch.setStatusState((String) tuple.get(HelperBidoEquipment.STATUS_STATE));
    bidoEquipmentSearch.setQuantity((BigDecimal) tuple.get(HelperBidoEquipment.QUANTITY));
    bidoEquipmentSearch.setBatchNumber((String) tuple.get(HelperBidoEquipment.BATCH_NUMBER));
    bidoEquipmentSearch.setQuantityUnit((String) tuple.get(HelperBidoEquipment.QUANTITY_UNIT));
    bidoEquipmentSearch.setLogbookTracked((Integer)
tuple.get(HelperBidoEquipment.LOGBOOK_TRACKED));
    bidoEquipmentSearch.setStatus((String) tuple.get(HelperBidoEquipment.STATUS));
    bidoEquipmentSearch.setContainerPnCode((String)
tuple.get(HelperBidoEquipment.CONTAINER_PN_CODE));
    bidoEquipmentSearch.setEquEquipmentCode((String) tuple.get("equEquipmentCode"));
    bidoEquipmentSearch.setPnCode((String) tuple.get(HelperBidoPnId.PN_CODE));
    bidoEquipmentSearch.setFamilyVariantCode((String)
tuple.get(HelperBidoFamilyVariantId.FAMILY_VARIANT_CODE));
    bidoEquipmentSearch.setSiteCode((String) tuple.get("siteCode"));
    bidoEquipmentSearch.setManufacturingBatch((String)
tuple.get(HelperBidoEquipment.MANUFACTURING_BATCH));
    bidoEquipmentSearch.setContainerSnManufacturer((String) tuple.get("containerSnManufacturer"));
    bidoEquipmentSearch.setSiteName((String) tuple.get("siteName"));
    bidoEquipmentSearch.setWhName((String) tuple.get("whName"));
    bidoEquipmentSearch.setWhCode((String) tuple.get("whCode"));
    return bidoEquipmentSearch;
};
```

Implementation and usage

Simple Query

```
CriteriaBuilder builder = sessionFactory.getCurrentSession().getCriteriaBuilder();
CriteriaQuery<BidoEquipment> query = builder.createQuery(BidoEquipment.class);
Root<BidoEquipment> bidoEquipmentRoot = query.from(BidoEquipment.class);
bidoEquipmentRoot.alias("root");
//To get a list of result
List<BidoEquipment> resultList =
sessionFactory.getCurrentSession().createQuery(query).getResultList();
//To get a unique result
BidoEquipment result = sessionFactory.getCurrentSession().createQuery(query).uniqueResult();
```

Query with select and options

```
CriteriaBuilder builder = sessionFactory.getCurrentSession().getCriteriaBuilder();
CriteriaQuery<Tuple> query = builder.createTupleQuery();
Root<BidoEquipment> bidoEquipmentRoot = query.from(BidoEquipment.class);
bidoEquipmentRoot.alias("root");
Selection<?> selectionEquipmentCode = root.get("id").get("equipmentCode");
selectionEquipmentCode.alias("equipmentCode");
Predicate predicateEquipmentCode = builder.equal(root.get("id").get("equipmentCode"),"value");
query.select(selectionEquipmentCode).where(predicateEquipmentCode);
List<BidoEquipment> resultList =
sessionFactory.getCurrentSession().createQuery(query).getResultList();
```

TupleQuery with multiselect and options

```
CriteriaBuilder builder = sessionFactory.getCurrentSession().getCriteriaBuilder();
CriteriaQuery<Tuple> queryTuple = builder.createTupleQuery();
Root<BidoEquipment> bidoEquipmentRoot = query.from(BidoEquipment.class);
bidoEquipmentRoot.alias("root");
List<Selection<?>> selections = new ArrayList<>();
//If joinType is LEFT
Join<BidoEquipment, BidtEquipment> joinLeft =
JoinCriteriaJPAUtil.joinWithFrom(root,"bidtEquipment",LEFT);
//If joinType is INNER, no need to precise it
Join<BidoEquipment, BidoEquipment> joinInner =
JoinCriteriaJPAUtil.joinWithFrom(root,"bidtEquipment");
//Join with another join
Join<BidtEquipment, BidtWarehouse> joinWithJoin =
JoinCriteriaJPAUtil.joinWithFrom(joinLeft/joinInner, "bidtWarehouse");
selections.add(root.get("id").get("equipmentCode").alias("equipmentCode"));
selections.add(joinLeft.get("equipmentCode").alias("equipmentCode");
selections.add(joinWithJoin.get("whCode").alias("whCode");
List<Predicate> predicates = new ArrayList<>();
predicates.add(builder.equal(join.get("whCode"),"value"));
Order orderEquipmentCode = builder.desc(root.get("id").get("equipmentCode"));
Order orderBidtEquipmentCode = builder.asc(join.get("equipmentCode"));
Order[] orders = {orderEquipmentCode, orderBidtEquipmentCode};
queryTuple.multiselect(selections).where(predicates.toArray(Predicate[]::new));
List<Tuple> resultTuple =
sessionFactory.getCurrentSession().createQuery(queryTuple).getResultList();
List<BidoEquipment> resultBidoEquipment = mapTupleToListEntity(resultTuple, mapper);
```