

# Hibernate DataViewObjects

2010-01-17 20:03:54

When working with Hibernate and nested domain object models, it can be a performance problem when trying to display several (including nested) properties in a view. Usually for such cases the use of view objects is practical.

This is the first scetch of my idea to automate population of view objects with hibernate.

Meanwhile I worked out some extended example with the use of Annotations. However, the current form is not usefull enough, as using criterions for restrictions is quite difficult. But more details on the next part of this. Here the first one:

Assume we have some nested domain model consisting of a person, an address and a country as following:

```
@Entity public class Person{ private Long id; private String firstname; private String lastName; private Date birthdate; private String comment; private Address address; @Id @GeneratedValue(strategy=GenerationType.AUTO) public Long getId(){ return id; } // appropriate getters and setter for id, // firstname, lastname, birthdate and comment @ManyToOne public Address getAddress(){ return address; }} @Entity public class Address{ private Long id; private String street; private Country country; // appropriate getters and setter for // id, street and country @ManyToOne public Country getCountry(){ return country; }} @Entity public class Country{ private String iso3166; // iso 3166 2 letter code // (uppercase), like DE for Germany private String name; @Id @Column(length=2) public String getIso3166(){ return iso3166; } // appropriate getters and setter for name and iso3166 }
```

Now let's assume we want to build a list displaying only the lastname and the countryname of a person. In such a case it's unnecessary to do a query like "SELECT p FROM Person p", since even if lazy loading is enabled, hibernate will load all properties leading to lot's of unneeded data transported from database to application.

In such a case we would prefer to build a view object, that only will hold the information we need to display:

```
public class PersonLastNameCountryViewObject{ private String lastName; private String countryName; // appropriate getters and setter }
```

So we now have a lightweight view object, but how do we get the data into it?

One possibility would be to build a custom query for this, like:

```
SELECT p.lastName, c.name FROM Person p LEFT JOIN p.address AS a LEFT JOIN a.country AS c
```

This would give us the needed data and we only would have to map the results to our view object.

Having several views may lead to lots of custom queries and code doing the retrieval and mapping stuff. So why not automating it?

I created some classes to handle this work, but there are few informations that have to be given to it, especially about which view object properties to map to which domain model properties.

In the above view object example, I decided to use a map of view object properties and define a domain model path for each, just like in commons-beanutils.

The mapping for the above example would be:

```
lastName -> lastName
countryName -> address.country.name
```

The path here starts with the initial domain class, in this case the Person class. Since Hibernate does not support nested object paths like in

```
SELECT p.name, p.address.country.name FROM Person p
```

I built some code to split the path and join the appropriate domain models into the query just to have the working query denoted previously.

So my view object mapping helpers contain of following classes:

```
// Just an interface that will create a // view object for us for each resultset row public interface
ViewObjectFactory{ public Object createNewViewObject(); } // a simple class to hold all meta data
needed for building our view objects public class ViewObjectMetaData{ // This will hold the mappings like
// lastName -> lastName // countryName -> address.country.name private final HashMap<String,
String> propertiesToObjectPaths = new HashMap<String, String>(); // this will hold our initial domain
class, like Person.class. // we need this to create a criteria in hibernate later on private Class
domainClass; // the factory we will use to create the view objects private ViewObjectFactory
voFactory; // as default, if no object paths are given, we assume that each view object property // will
be mapped non-nestingly to the domain class properties with the same name. // so the mapping will be
something like: // propertyA -> propertyA // propertyB -> propertyB // will need BeanUtils to work, or
make some own code with reflection public ViewObjectMetaData(ViewObjectFactory voFactory, Class
domainClass){ this.voFactory = voFactory; this.domainClass = domainClass; try { Map desc =
BeanUtils.describe(dtoFactory.createDto()); desc.remove("class"); final
Collection<String> properties = (Collection<String>) desc.keySet(); for (final String
property:properties){ propertiesToObjectPaths.put(property, property); } } catch (Exception
e) { // just make a runtime exception from it... if (e instanceof RuntimeException){ throw
(RuntimeException) e; } else{ final RuntimeException r = new RuntimeException(e);
r.setStackTrace(e.getStackTrace()); throw r; } } } public
ViewObjectMetaData(ViewObjectFactory voFactory, Class domainClass, Map<String, String>
propertiesToObjectPaths){ this.voFactory = voFactory; this.domainClass = domainClass;
this.propertiesToObjectPaths.putAll(propertiesToObjectPaths); } // appropriate getters }
```

Now we have all information needed to build up queries and populate the view objects. The main work will consist of building up the query from the object path and mapping the resultset to the view objects.

For building the query, I used the criteria API of hibernate, which allows to add further restrictions on the query later on. The algorithm for transforming the object path to a criteria is not that complicated. We have to think about two different cases in the object path:

1. object path consists of one part, like "lastName". This has to be mapped to a projection, like:

```
SELECT p.lastName FROM Person p
```

2. the object consists of multiple parts, like "address.street" or "address.country.name". In this case we have to create a hibernate alias (which internally maps to a join) for each part except the first (which is the root domain class) and the last, which has to be added as a projection as it is the property we want to have. So we have for the two part object path example:

```
SELECT a.street FROM Person p LEFT JOIN p.address AS a
```

and the three part object path example:

```
SELECT c.name FROM Person p LEFT JOIN p.address AS a LEFT JOIN a.country AS c
```

Also we'll keep track of the indices of the projected properties in the resultset for the mapping between resultset and viewobject

```
public class ViewObjectCriteriaBuilder{ // just a helper class to transport // criteria and
columnmappings public static class ViewObjectCriteria{ private Criteria criteria; private
HashMap<String, Integer> propertyToColumnMappings = new HashMap<String, Integer>(); //
appropriate getters/setters } private SessionFactory sessionFactory; // appropriate setter for the
sessionFactory // or just pass a current session to the next method public ViewObjectCriteria
buildCriteria(ViewObjectMetaData voMetaData){ final ViewObjectCriteria voCriteria = new
ViewObjectCriteria(); final Criteria criteria = sessionFactory.getCurrentSession().
createCriteria("&quot;d&quot;", voMetaData.getDomainClass()); voCriteria.setCriteria(criteria); final
Map<String, String> propertiesToObjectPaths = voMetaData.getPropertiesToObjectPaths(); // our
projections final ProjectionList projections = Projections.projectionList(); // max. amount of
aggregations in object path int maxAggregationDepth = 0; // index of property in resultset int
propertyColumnIndex = 0; String objectPath; // objectpath of current vo property // amount of
aggregations in current object path int aggregationDepth; final ArrayList<String> aggregationParts =
new ArrayList<String>(5); // parts of the object path, splitted // in the first step we only collect
information // about the needed aggregations, // the joins will be build later on for (final String
property:propertiesToObjectPaths.keySet()){ objectPath = propertiesToObjectPaths.get(property);
// this helper just counts the // occurrences of '.' in the objectPath aggregationDepth =
StringUtil.occurrences(objectPath, '.'); if (aggregationDepth<1){ // no aggregation, like //
lastName -> d.lastName projections.add(Projections.property("&quot;d.&quot;"+objectPath)); }
else{ maxAggregationDepth = Math.max(maxAggregationDepth, aggregationDepth);
// aggregations exist // first split the object path aggregationParts.clear(); // this helper
splits the objectPath // on &quot;\.&quot; and adds all parts to the // aggregationParts-list.
Works the same // way as String.split CollectionSplitter.toList(aggregationParts, objectPath,
&quot;\.&quot;); // last aggregations alias String lastAggregationName = null; String
currentAggregationPart, currentAggregationName; // when we're here, there are at least two parts in
the object path // first part is an property of the // root domain class, e. g. address -> d.address
for (int aggregationIndex=0; aggregationIndex<=aggregationDepth; aggregationIndex++){
currentAggregationPart = aggregationParts.get(aggregationIndex); if (aggregationIndex==0){
// equals to &quot;LEFT JOIN d.address AS ag_address&quot;; lastAggregationName =
&quot;ag_&quot;+currentAggregationPart; aggregations.put(
&quot;d.&quot;+currentAggregationPart, lastAggregationName); } else if
(aggregationIndex==aggregationDepth){ // last part in object path, which // denotes a
property we want to have projected // equals to &quot;SELECT ... ag_address_country.name
FROM ...&quot;; projections.add( Projections.property(
lastAggregationName+&quot;.&quot;+currentAggregationPart)); } else{ // somewhere
in the middle // equals to // &quot;LEFT JOIN ag_address.country AS
ag_address_country&quot;; currentAggregationName =
lastAggregationName+&quot;.&quot;+currentAggregationPart; aggregations.put(
lastAggregationName+&quot;.&quot;+currentAggregationPart, currentAggregationName);
lastAggregationName = currentAggregationName; } } } // update index of property in
resultset voCriteria.getPropertyToColumnMappings(). put(property, propertyColumnIndex++); }
```

```

// if there are any aggregations, we have // to add an alias for each to the criteria if
(maxAggregationDepth>0){ // sort aggregatios (joins) by their alias length // this is to enshure that
ag_address is // added before ag_address_country final List<String> aggregationsSizeSorted =
new ArrayList<String>(aggregations.keySet()); Collections.sort(aggregationsSizeSorted, //
the string length comparator just compares // the length of two strings new
StringLengthComparator()); for (final String fromAggregation:aggregationsSizeSorted){ // here we
also may use LEFT JOIN or some other join type criteria.createAlias( fromAggregation,
aggregations.get(fromAggregation)); } } // add projections to criteria
criteria.setProjection(projections); return voCriteria; } }

```

Now we have build a criteria which will load all neccessary data from the database. Meanwhile we could add some restrictions or limitations to the query to only retrieve a portion of all rows. Somewhere all along the code we will need to have a valid session/transaction, but I'll skip that, as this is just an example.

Next we build a class that will execute the criteria and map the columns from the resultset to the viewobjects. Here are several possibilites. For example using a special hibernate rowsetmapper, but the way I did it below also works fine.

```

public List buildViewObjects(ViewObjectCriteria voCriteria, ViewObjectMetaData voMetaData){
ScrollableResults result = null; try{ Criteria criteria = voCriteria.getCriteria(); Map<String, Integer>
propertyColumnMappings = voCriteria.getPropertyColumnMappings(); // some cache method or
scroll lock method here, if needed result = criteria.scroll(ScrollMode.FORWARD_ONLY); Object []
rowdata; Object resultObject; final ArrayList viewObjects = new ArrayList(); while (result.next()){
rowdata = result.get(); resultObject = voMetaData.getViewObjectFactory().
createNewViewObject(); for (final String property: propertyColumnMapping.keySet()){ try {
// also possible via reflection PropertyUtils.setProperty(resultObject, property,
rowdata[propertyColumnMappings.get(property)]); } catch (Exception e) { if
(LOG.isErrorEnabled()){ LOG.error("&quot;While setting property &quot;&quot;+
property+"&quot;&quot; from column &quot;&quot;+
propertyColumnMapping.get(property)+"&quot;&quot;&quot;, e); } resultObject = null; }
} if (resultObject!=null){ viewObjects.add(resultObject); } } return viewObjects; }
finally{ if (result!=null){ result.close(); } } }

```

That's it (for now).

Limitations:

- the datatype of the setter in the view object has to be the same as in the corresponding domain property
- restricting the query is quite difficult, as you don't know the aliases of the generated aggregations a priori

Further improvements could be:

- some annotations so that we just annotate our view object and rest is done in background
- make a full working open-source API out of it
- optimizations on the queries. if we only would like to select the id of a nested domain class, like "address.id", which already is known to the Person table (since this is the referencing foreign key), it would be great not to have to join the addresses

## **Narcanti**

Hibernate DataViewObjects

- automatic mapping between differing property types of view object and domain object (eg. long to string conversions)
- improve the possibility to restrict the generated query