HOWTO: “JOIN”

foobar.

There are two entities, say, Room and Building, where Room belongs to a Building and Building has a name. In order to display a list of Rooms and the name of the Building they belong to, one would simply do a JOIN in ordinary SQL:

```
SELECT Room.name, Building.number FROM Room JOIN Building ON Room.building = Building.id
```

However, using the weave/bakery ORM it is not possibly to issue a custom query to the database engine. You can, however, do the following to obtain Rooms and the names of the Buildings they belong to in two steps:

1. Select the rooms, and
2. select the Buildings which id is in one of the Rooms building-references.

This can be written like this:

```
List<Room> $rooms = manager().getRoom(...FILTERS...);
$eq = new Function<Room,Filter>() {
	public Filter call(Room $r) {
		return eq(Building.id, $r.getBuilding());
	}
}
List<Building> $buildings = manager().getBuilding(any(map($eq, $rooms)));
```

This will be translated to exactly two queries, no JOIN is done at all. The idea of how to deal with this data in a Template is the following:

1. Pass both lists to the template
2. Refer to the entities via their id

This may look like this (Java):

```
put(“room”, $rooms);
put(“building”, $buildings);
```

and the template like this (XSL-T):

```
<xsl:for-each select=”//item:room”>
	<xsl:value-of select=”@number” />
	located in
	<xsl:value-of select=”//item:building[@id = current()/@building]” />
</xsl:for-each>
```

Explanation

map() takes the anonymous `Function` created above which extracts the id form a room and returns a comparing `Filter` (which states “*I am true if Person.id equals the number I got in my second parameter*”). Since that function is mapped on the $rooms we obtain a list of filters which are than connected by any() - which means, that all buildings that do have any one of the ids mentioned in the list constructed by *map()* are included in the result.

This is illustrated quite more beautiful in the following piece of pseudo-code:

```
rooms <- { any rooms where FILTER applies }
buildings <- { any building which id is in map(func(r) -> get-building-id(r), rooms) }
```

Imports

You will need to import the following functions:

```
import static de.fu.bakery.orm.java.filters.Filters.*;
import static de.fu.junction.functional.F.*;
```

The problem

The problem regarding mapping objects from a JOIN-query is, that the standards-mechanism puts a result into an object. The result of a JOIN-query would be an object which is not defined in our OR-Schema (one containing properties from two entities – I'd like to call such an object a virtual object). In many languages this wouldn't be a problem (PHP, JavaScript, Lisp), since one can extend or dynamically create an object to her likings. However, in Java this is not possible. Since for such a dynamically created object one would need a class definition (which needs to be their statically) we are left with declaring more than we should have to either way.

Another approach would be to create two objects from one result – however, this is a tedious work which would need to be done in a generic way to be included in Code-generation. This is rather mind-breaking stuff, which, in principle, might be done in a sunny sunday afternoon – however, it has not beed done yet.

The problem to tackle here is, that two different entities may have same named attributes. Of course, one can rename things, access them qualified or even adjust the automatically generated schema to prefix attribute-names in such a way that they are unique across all entities (for example, by prefixing the name of the entities they belong to).

The salvation

I believe that the workaround is nearly as fast as a JOIN, since only two queries are done and in the second query a rather rigid filter is defined. The only overhead added is one query which is constructed using some higher-order functions (which are used for the sole purpose of downsizing the amount of code needed to express things).

I even do think that this approach in some cases may be better than joins. In a join the full cross product would be built – resulting in more data transferred from the database, etc. This way only two, for large data sets much smaller, results are to be transferred. The operation itself is also less complex, since simply two tables are read independently.