PLANTLORE

Database Layer API

Charles University in Prague
Faculty of Mathematics and Physics

Author: Tomáš Kovařík, tkovarik@gmail.com

Last updated: 30. 3. 2006

Introduction

This document describes how the database layer used for executing DB queries is implemented and should be used in Plantlore. The purpose of this document is to provide detailed explanation of the implementation used and should be consulted before posting questions to Plantlore devlopement listservs.

Terminology

- *Projection* – selection of columns we want to have in the result of a query.
- *Restriction* – selection of rows we want to have in the result of a query.

Implementation and source files

Database layer in Plantlore is implemented in several classes in the source tree. This document describes only API for database access and does not describe issues connected with RMI. The client – server distinction is not used here, for the purpose of the document Plantlore is a single application.

- `net.sf.plantlore.middleware.DBLayer` – Interface defining API for working with database.
- `net.sf.plantlore.server.HibernateDBLayer` – Implementation of the DBLayer interface using Hibernate. Currently only this implementation is provided, however different implementations (e.g. for direct database access using JDBC) are possible.
- `net.sf.plantlore.middleware.SelectQuery` – Interface defining API for buidling SELECT queries.
- `net.sf.plantlore.server.SelectQueryImplementation` – Implementation of SelectQuery interface using Hibernate criteria queries. Currently only this implementation is available, however different implementations are possible.

Executing INSERT queries

Inserting data into database is done using `executeInsert()` method defined in the DBLayer interface. The argument of this method is holder object with the data we wish to save. Any holder object representing database table can be used.

```
/*************************************************************
Insert new author into the database
*************************************************************/
Author author = new Author();           // Create new author
author.setWholeName(name);              // Set parameters
author.setOrganization(organization);
author.setRole(role);
author.setAddress(address);
author.setPhoneNumber(phoneNumber);
author.setEmail(email);
author.setUrl(url);
author.setNote(note);
// Execute INSERT query
rowId = database.executeInsert(author);
```

If the query is successfully executed, `executeInsert()` method returns unique identifier of the newly inserted record.

Executing DELETE queries

Deleting data from the database is possible using `executeDelete()` method defined in the DBLayer interface. This method requires as a parameter object representing a record we wish to delete.

No value is returned by `executeDelete()` method

TODO: Add delete by id

TODO: Add example.

Executing UPDATE queries

Updating data in the database is possible using `executeUpdate()` method defined in the DBLayer interface. Again, the only parameter is an object representing updated record. Record to be updated is identified using unique identifier (primary key of the table) in the holder object.

No value is returned by `executeUpdate()` method.

TODO: Add example

Executing SELECT queries

Selecting data from the database is a little bit more complicated. In order to be able to construct general SELECT queries and modify them on the server side (in order to filter certain data), API for query construction was implemented.

Constructing a query

Construction of a SELECT query is started by calling `createQuery()` method defined by the DBLayer interface. An argument of this method is a Class object representing one of the holder objects. This holder object is a primary record we want to select from the database. Of course records of other types can be associated and joined into the query.

`CreateQuery()` method returns an instance of SelectQuery object which is later used for query construction and finally for the execution of Query.

Projections

Projections allow us to select columns we want to have in the result after query execution. When no projections are set, all the columns of the selected table(s) are returned. Note that some users do not see all the available columns (these are filtered on the server side)

Projections can be added to the query by means of `addProjection()` method defined in the SelectQuery interface. Parameters of this method are type of projection and name of the projected column. Names of the columns are available as constants stored in the respective holder objects.

Sometimes we do not use only plain columns in the SELECT clause of the query but we use various functions as well (e.g. COUNT, MAX, AVG etc.). These functions can be used by providing correct value of the first parameter of `addProjection()` method. Available "types of projection" defined in `net.sf.plantlore.common.PlantloreConstants` are:

- PROJ_AVG – Average value, SQL function AVG()
- PROJ_COUNT – Count of records, SQL function COUNT()
- PROJ_COUNT_DISTINCT – Count of distinct records
- PROJ_GROUP - ?
- PROJ_MAX – Maximum value, SQL function MAX()
- PROJ_MIN – Minimum value, SQL function MIN()
- PROJ_PROPERTY – Simple projection as described above, selecting columns we want to have in the result.
- PROJ_ROW_COUNT - ?
- PROJ_SUM – Sum of the values, SQL function SUM()

Restrictions

Restrictions give us a way how to limit the selection of records to those satisfying certain constraints. In SQL, restrictions are expressed as the WHERE clause of the SELECT query.

Restrictions can be added to a query using `addRestriction()` method defined in the SelectQuery interface. This method takes up to 5 parameters:

1. *Type of restriction* – defines which SQL operator should be used. Available types are defined in `net.sf.plantlore.common.PlantloreConstants`. Please consult this file for more uinformation.
2. *First property name* – name of the column used for operators which accept one or two columns as arguments.
3. *Second property name* – name of the column used for operators which accept two columns as arguments. If the restriction works with only one column, this parameter can be null.
4. *Value* – Any value we want to use as an argument to the selected operator.
5. *Values* – Certain operators (e.g. operator IN) work with collection of values. For these operators, tha last argument is a collection of values we want to use with the given operator. If you use operator which doesn't accept collection of values, this parameter can be null.

In the current version, restrictions cannot be grouped using AND and OR operators, all the defined restrictions are applied (as if they were connected using AND).

In case no restrictions are defined for a query, all the records from the selected table are returned.

Aliases and joining

In order to select data from more tables, SelectQuery API makes it possible to define aliases. Alias is just another name for the associated table which can be used to access columns of this table. To assign an alias, use `createAlias()` method. The first parameter of this method is the column of the main table which associates it with another table. The second is the name of the alias, which

can be any reasonable string (such that it can be used as an alias in the SQL query – e.g. cannot contain spaces). When the alias is defined, we can access the fields of associated table using the dot notation (<alias_name>.<field_name>).

```
/*************************************************************
We want to select a history record for a given occurrence.
Table tHistory is associated with table tHistoryChange and
this table is associated with tOccurrences table (via foreign
keys). The primary table we want to select from is tHistory,
we need to create aliases for tables tHistoryChange and
tOccurences.
*************************************************************/
// Select data from tHistory table
query = database.createQuery(HistoryRecord.class);
// Create aliases for other tables.
// See how the dot notation is used
query.createAlias("historyChange", "hc");
query.createAlias("hc.occurrence", "occ");
// Add restriction to CUNITVALUE column of tOccurence table
query.addRestriction(PlantloreConstants.RESTR_EQ,
                     "occ.unitValue", null, "unitValue", null);
```

Ordering the results

Ordering results is possible using `addOrder()` method defined in the SelectQuery interface. `AddOrder()` method takes two arguments, the direction of ordering and the column to be used for ordering the results. Direction can be either ascending or descending (see constants DIRECT_ASC and DIRECT_DESC defined in `net.sf.plantlore.common.PlantloreConstants`).

```
/*************************************************************
Select data from the DB. Find authors according to the name.
*************************************************************/
// Create new SelectQuery - select from tAuthors
SelectQuery query = database.createQuery(Author.class);
// Add restriction on the author name.
query.addRestriction(PlantloreConstants.RESTR_LIKE,
          Author.WHOLENAME, null, "%tom%", null);
// Order by the whole name of the author
query.addOrder(PlantloreConstants.DIRECT_ASC, Author.WHOLENAME);
// Execute query, get id of the result
resultId = database.executeQuery(query);
```

Fetch mode

TODO

Running a query

When the query is constructed, we can proceed to the execution. Query is executed by calling `executeQuery()` method defined in the DBLayer interface which takes an instance of SelectQuery we want to execute as an argument.

`ExecuteQuery()` method returns unique id of result which is used for reading the results. This is necessary because more query results can be opened at the same time.

Working with the query results

Executing query doesn't fetch any data from the database. To get the selected data, either `next()` or `more()` method has to be used. These methods are defined in the DBLayer interface.

Method `next()` fetches next record from the result of the query. The only parameter of this method is the id of the result we want to read. `Next()` returns an array of records read from the database. Although we are selecting a single row (record), very often associated records are fetched. For example, when selecting User record from TUSER table, associated Right record (table TRIGHT) is fetched. When retrieving results of a query, you have to know which records were fetched and cast the items in the returned array accordingly.

Method `more()` is used for retrieving an array of records from the result. (equivalent to multiple `next()` calls). Parameters of this method are the indexes of the first and last record to retrieve and of cource an identifier of the result we want to read. The return value is a two dimensional array containg records in the same way as with the `next()` method.

To get the number of records returned by the query, `getNumRows()` method is available. The only parameter is the identifier of the result we are interested in. If the result is empty, value zero is returned.

Initializing and closing DB connection

TODO:

Conclusion

TODO: