DAO Examples

Tables

- Create the table in the database using SQL:

```
create table account (
  id serial not null,
  current_balance integer not null,
  name varchar(100) not null,
  created_on date not null,
  primary key (id),
  constraint account_name_uc unique (name)
);
```

- Declare the table in the configuration file as shown below and generate the DAOs:

```
<table name="account">
   <auto-generated-column name="id" />
</table>
```

The DAOs will provide the following methods:

Select by Primary Key "Get the account with **PK** #1004".

```
AccountDAO a = AccountDAO.select(1004);
```

Returns null if the row is not found.

This method is only available if the table has a PK (simple or composite).

Select by Unique Index "Get the account with **Name** 'CHK1004'".

```
AccountDAO a = AccountDAO.selectByUIName("CHK1004");
```

Returns null if the row is not found.

There is one method like this one for each unique constraint or unique index the table has.

Select by Example "Get all the accounts **Created On** a specific day".

```
AccountDAO filter = new AccountDAO();
filter.setCreatedOn(creationDate);
List<AccountDAO> accounts = filter.select();
```

An empty list object is returned if there are no matches.

If more properties of the filter are set, the condition will match them all using an "AND" operator. If no properties are set this method returns all the rows of the table.

Select by Example with order "Get all the accounts **Created On** a specific day, ordered by **Current Balance** (descending), then by **Name** (ascending)".

```
AccountDAO filter = new AccountDAO();
filter.setCreatedOn(creationDate);
List<AccountDAO> accounts = filter.select(
    AccountOrderBy.CURRENT_BALANCE$DESC , AccountOrderBy.NAME);
```

An empty list object is returned if there are no matches.

If more properties of the filter are set, the condition will match them all using an "AND" operator. If no properties are set this method returns all the rows of the table.

Select by Foreign Keys "Get all **Transactions** related to this account (**a**) by **Account Id**.

```
AccountDAO a = ...
List<TransactionDAO> txs = a.selectChildrenTransaction().byAccountId();
```

The list is empty when there are no matches.

There is one method like this one for each Foreing Key (imported and exported) to allow easy access to related DAOs using parent, children and reflexive database relationships.

Select by Foreign Keys with order "Get all **Transactions** related to this account (**a**) by **Account Id**, ordered by **Federal Branch Id** (ascending), then by **Amount** (descending)".

```
AccountDAO a = ...
List<TransactionDAO> txs = a.selectChildrenTransaction().byAccountId(
    TransactionOrderBy.FED_BRANCH_ID , TransactionOrderBy.AMOUNT$DESC);
```

Insert "Create an account with **Id** 'CHK4010', with zero **Balance**".

```
AccountDAO a = new AccountDAO();
a.setName("CHK4010");
a.setCreatedOn(creationDate);
a.setCurrentBalance(0);
a.insert();
System.out.println("The new account id is: " + a.getId());
```

When the primary key is autogenerated (by sequences or identity columns) its value is not specified in the DAO, but returned by the database after inserting.

Update by Primary Key "Update the balance to $250 on the account #1004".

```
AccountDAO a = AccountDAO.select(1004);
a.setCurrentBalance(250);
int rows = a.update();
```

This method is only available if the table has a PK, since it uses it to update the row.

Update by Example "Set the **Current Balance** to 100 for all accounts **Created On** a specific date".

```
AccountDAO filter = new AccountDAO();
filter.setCreatedOn(creationDate);
AccountDAO updateValues = new AccountDAO();
updateValues.setCurrentBalance(100);
int rows = filter.updateByExample(updateValues);
```

Delete by Primary Key "Delete the account #1007".

```
AccountDAO a = new AccountDAO();
a.setId(1007);
int rows = a.delete();
```

This method is only available if the table has a PK, since it uses it to delete the row.

Delete by Example "Delete all accounts **Created On** a specific date".

```
AccountDAO filter = new AccountDAO();
filter.setCreatedOn(creationDate);
int rows = filter.deleteByExample();
```

Custom MyBatis Mappers

If you write a custom MyBatis mapper you can use it by manually calling it.

```
    SqlSession sqlSession = null;
    try {
      sqlSession = AccountDAO.getSqlSessionFactory().openSession();

      List<AccountDAO> activeAccounts = sqlSession
          .selectList("com.company.daos.complexaccountsearch");
      for (AccountDAO a : activeAccounts) {
        System.out.println("Active account #: " + a.getId());
      }

    } finally {
      if (sqlSession != null) {
        sqlSession.close();
      }
    }
```

The DAO generator will pick up and register custom MyBatis mappers located in the "custom" directory inside the mappers directory. Those mappers will be available to use as shown above for select, insert, delete, or update.

Views

Database views can be used to retrieve data from the complex joins they represent. They can also be reused for multiple different queries since a filter can be specified at runtime. Therefore, they are more flexible than Selects (explained below). However, they don't give access to all exotic database-specific possibilities that Selects can.

- Create the view in the database using SQL:

```
create view tx_branch
    (account_id, branch_id, branch_name, amount) as
  select t.account_id, b.id, b.name, t.amount
    from transaction t, federal_branch b
    where t.fed_branch_id = b.id
    order by t.amount;
```

- Declare the table in the configuration file as:

```
<view name="tx_branch" />
```

Select by Example Get all the accounts transactions for **Branch** 681, ordered by **Account Id**, then by **Amount**.

```
TxBranchDAO tb = new TxBranchDAO();
tb.setBranchId(681);
List<TxBranchDAO> txs = tb.select(TxBranchOrderBy.ACCOUNT_ID ,
    TxBranchOrderBy.AMOUNT );
```

Views are considered read-only, so this is the only persistence method available in the DAO.

Selects

Selects give you access to full database-specific SQL, and allows you to define the specific parameters you want to use for a select SQL. Selects criteria and structure is fixed at development time.

- No need to create any database table or view.
- Can use zero, one, or more parameters.
- Declare the select in the configuration file as:

```
<select name="unusual_tx">
  <![CDATA[
  select a.* , t.*
  from account a, transaction t
  where t.account_id = a.id
    {*
    and a.current_balance <=
      #{maxBalance,javaType=java.lang.Integer,jdbcType=NUMERIC}
    and t.amount >=
      #{minAmount,javaType=java.lang.Integer,jdbcType=NUMERIC}
    order by a.id, t.seq_id
    limit
      #{maxRows,javaType=java.lang.Integer,jdbcType=NUMERIC}
    offset
      #{offset,javaType=java.lang.Integer,jdbcType=NUMERIC}
    *}
  ]]>
</select>
```

Parameterized Select "Select transactions with a maximum balance of $50, and with transaction's amounts of at least $300. Skip the first 5000 rows, then limit the result to 1000 rows".

```
List<UnusualTxDAO> utxs = UnusualTxDAO.select(50, 300, 1000, 5000);
```

Transactions

All DAO operations work in auto-commit mode by default.

Simple Transaction

```
    TxManager txManager = null;
    try {

      txManager = AccountDAO.getTxManager();
      txManager.begin();

      AccountDAO a1 = AccountDAO.select(1404);
      a1.setCurrentBalance(350);
      a1.update();

      AccountDAO a2 = AccountDAO.select(2780);
      a2.setCurrentBalance(100);
      a2.update();

      txManager.commit();

    } finally {
      if (txManager != null) {
        txManager.close();
      }
    }
```

If the second update fails, the first update is rolled back. A **`finally`** block is used to always release the resources allocated by the transaction manager. If the commit() is not executed the whole transaction is rolled back when the resources are released; this is the expected behavior when exceptions are thrown during the database changes, or for any other reason. All DAOs provide a method to retrieve the transaction manager.

Custom Transaction

You can also initiate custom transactions. In the case above, you could get a SERIALIZABLE transaction by changing the line:

```
      txManager.begin();
```

to:

```
      SqlSession sqlSession = txManager.getSqlSessionFactory().openSession(
          TransactionIsolationLevel.SERIALIZABLE);
      txManager.begin(sqlSession);
```

Interleaving Transactions

Interleaving transactions are multiple parallel transactions performed in the same database connection and same processing thread. Though rare, they are supported if the DBMS and the JDBC driver support it.

MyBatis Transactions

You can also use the underlying MyBatis transaction manager.

```
    SqlSession sqlSession = null;
    try {
      sqlSession = AccountDAO.getSqlSessionFactory().openSession();

      Details details = new Details(1404, 350);
      sqlSession.update("com.company.daos.updateBalance", details);

      details = new Details(2780, 100);
      sqlSession.update("com.company.daos.updateBalance", details);

    } finally {
      if (sqlSession != null) {
        sqlSession.close();
      }
    }
```

In this case you'll need to create the class Details to send the parameters, and also will need to write the exact name of the SQL mapper to use. Also, unlike the DAOs transaction manager, the MyBatis transaction manager commits by default, unless you explicitly execute a rollback().