



## Introduction

### What's a DAO ?

A DAO (Data Access Object) is a design pattern largely used in java. It's purpose is to create an interface with a RDBMS (Relational DataBase Management System), so that any code that is related to accessing the database is contained entirely inside a DAO. In Java this means that any coding related to jdbc will be contained only inside the DAO, the benefits of this pattern are several.

### So, what's the problems with DAOs ?

There is no real problem with the pattern, except that programmer's expend a lot of time creating these objects and debugging their code, up to the point when you realize that every DAO share some common characteristics, and that you're programming the same things over and over. The real purpose of a DAO is to execute queries and retrieve data from the database, so why don't we concentrate only on the queries and the data we need instead of worrying about the jdbc mechanism ?

Another problem with DAOs is that unless you are using "stored procedures" for every single query, all the coding for the queries will be inside a java class. This is a problem because to change and optimize these queries you'll need a java programmer to do it. But sometimes in your team you may have a SQL specialist that doesn't understand a thing about java and can literally "break" the application if he is to try to change the code to optimize a query.

### There comes Efreet

The Efreet package is a tool with the intention of solving these problems. Instead of creating a DAO class, you write a XML with the queries that you need to be executed and then simply call an object created by the package to execute them.

## Configuring Efreet to work with Tomcat

Efreet is currently designed for web applications, it can be used for standalone applications as well, but it really shines when you use it for a web application.

To configure efreet to work with a web application, first you must create a JDBC resource within your application server. I believe that this can be done with most application servers, but I'll use the tomcat for this example.

In recent versions of tomcat, the configuration for a single application can be configured on a XML file specific for that application. These file must be located inside the `<TOMCAT_HOME>/conf/Catalina<host>/` directory. To configure a JDBC resource you must include the following :

```
<Resource name="jdbc/oracle"
auth="Container"
type="javax.sql.DataSource"
factory="org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory"
driverClassName="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@127.0.0.1:1521:SID"
username="scott"
password="tiger"
maxActive = "20"
maxIdle = "10"
maxWait = "-1"
>
</Resource>
```

Depending on the version of the tomcat you're using, this may not work as you must add additional configuration on the same file , as follows :

```
<ResourceParams name="jdbc/oracle">
<parameter><name>url</name><value>jdbc:oracle:thin:@127.0.0.1:1521:SI
D</value></parameter>
<parameter><name>driverClassName</name><value>oracle.jdbc.driver.Orac
leDriver</value></parameter>
<parameter><name>username</name><value>scott</value></parameter>
<parameter><name>password</name><value>tiger</value></parameter>
</ResourceParams>
```

This will configure the resource for the connection used by the efreet package. Write down the name of the resource as we will need it later.

## Including efreet on your application

To use efreet in your application is very simple. You just have to add the efreet.jar to your lib directory. In the case of web applications, its WEB-INF/lib.

Then you can start creating XML query files for the queries and add a few lines of code to your application and you're ready to go.

### Creating a query file

For each distinct DAO that you want to create you will write a query file. A query file is a XML document where you can describe several queries for your DAO to execute. The name of the file must be same as the DAO you want to create. And you must put this file in your class directory (WEB-INF/classes)

### XML File Example : example.xml

```
<?xml version="1.0"?>
<!DOCTYPE dao PUBLIC "DTD for Efreet DAO Configuration File 1.0//EN"
"efreet-dao_1_0.dtd">

<DAO NAME="example" datasource="oracle">

  <QUERY NAME="nextUser">
<![CDATA[
  select USER_ID.NEXTVAL as ID from DUAL
]]>
  <RESULT INDEX="1">USER_ID</RESULT>
</QUERY>

  <QUERY NAME="search">
<![CDATA[
  select USER_ID, LOGIN, NAME
  from USER
  where USER_ID <> 0
  and USER_ID = decode(?, 0, USER_ID, '', USER_ID, ?)
  and LOGIN LIKE '%'||?||'%'
  order by LOGIN
]]>
  <PARAMETER INDEX="1" TYPE="NUMBER">USER_ID</PARAMETER>
  <PARAMETER INDEX="2" TYPE="NUMBER">USER_ID</PARAMETER>
  <PARAMETER INDEX="3" TYPE="CHAR">LOGIN</PARAMETER>
  <RESULT INDEX="1" TYPE="NUMBER">S_USER_ID</RESULT>
  <RESULT INDEX="2" TYPE="CHAR">S_LOGIN</RESULT>
  <RESULT INDEX="3" TYPE="CHAR">S_NOME</RESULT>
</QUERY>

</DAO>
```

The datasource parameter in the first line must correspond to the same name you use in the configuration of the jdbc resource on tomcat. In this case “**oracle**”.

## ***Writing code to use the efreet package on your code.***

Next step is to write code using the efreet package so that you can access the queries defined in the query file.

Here's a sample code that uses both queries defined in our example.

### **Code Sample**

At anytime your objects can access the DAOs created by the factory with minimum coding :

```
import org.utopia.efreet.DAOFactory;
import org.utopia.efreet.DataAccessObject;
import org.utopia.efreet.QueryParameter;
import org.utopia.efreet.QueryResult;

...

// Example 1
// Retrieving a single line of results using a query

DataAccessObject dao = DAOFactory.createDAO("example");

QueryResult qr = dao.executeQuerySingle("nextUser");

int i = qr.getInt("USER_ID");

...

// Example 2
// Retrieving a collection of lines using a query
// First we create a QueryParameter object and populate it,
// this are the parameters to be passed to the query
QueryParameter qp = new QueryParameter();
qp.put("USER_ID", i);
qp.put ("LOGIN", "login");

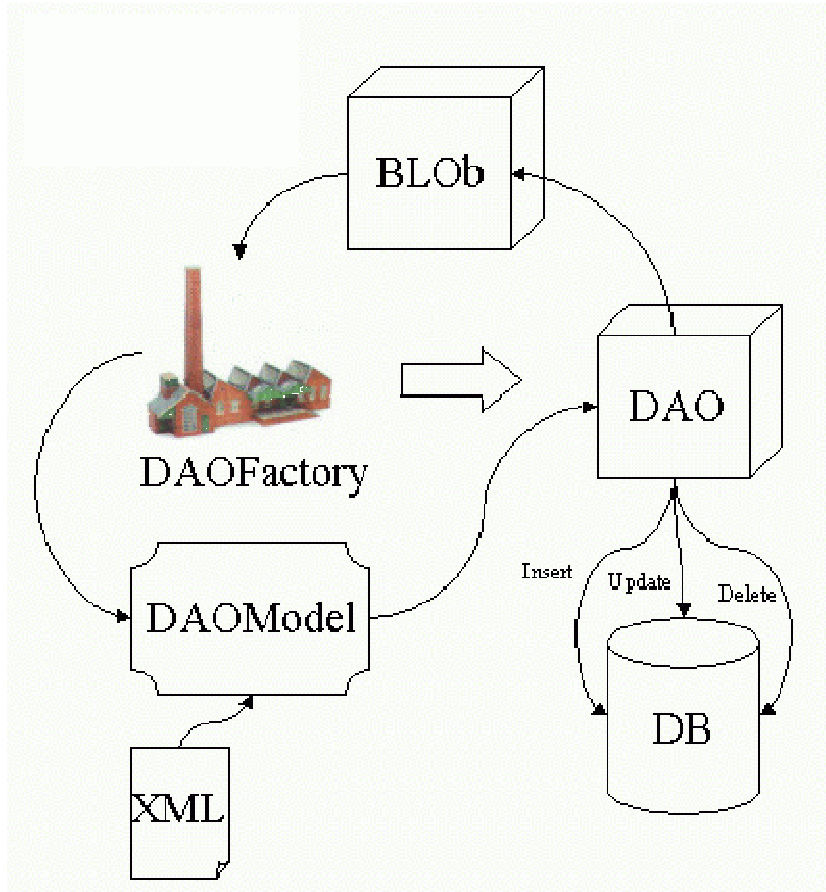
Collection col = dao.executeQuery("search", qp);

// This is a Collection of objects of type QueryResult
// To retrieve the value you must iterate over it
Iterator iter = col.iterator();
while (it.hasNext()) {
    QueryResult qr = (QueryResult) it.next();
    int userId = qr.getInt("S_USER_ID");
    String login = qr.getString("S_LOGIN");
    String name = qr.getString("S_NOME");
}

...
```

## Architecture Details

Efreet was build considering the following design in mind :



Whereas :

- **BLOb** - Business Logic Object - This represents any object in your system that will use the DAOs.
- **DAOFactory** - The DAOFactory class is responsible for managing the models and generate DAOs to be used by the BLObs.
- **DAOModel** - A model based on an XML file to represent the DAO.
- **DAO** - The Data Access Object - An object with the capability to access RDBMS.
- **DB** - The Database.
- **XML** - The xml file containing the description for the DAO.

## Creating a DAO

To create a DAO, you must invoke the DAOFactory method *createDAO*, passing the name of the DAO to be created as a parameter. The name of the query file must match the name of the same DAO exactly. In our example, the query file was called “example.xml” and the DAO was called “example”. The efreet package searches for these files in the classpath. The DTD for the xml file must also be located on the classpath, otherwise the xml parser will complain. If you wish to put the query file inside a directory, you do so, but when you make a call to the *createDAO* method you must pass the directory name plus the name of the dao.

Examples :

```
DAOFactory.createDAO("myDAO");
```

will search for a “myDAO.xml” file in the base classpath (in web applications this means WEB-INF/classes)

```
DAOFactory.createDAO("efreet/myDAO");
```

will search for a “myDAO.xml” file in the base classpath + “/efreet” directory (in web applications this means WEB-INF/classes/efreet)

## Writing the Query File

The query file represents a model for the DAO to be created. This is a XML file, with a syntax defined in the DTD (provided with the efreet package).

The query file always starts with a <dao> tag. There can be only one dao definition per file. This tag receives two parameters “name” and “datasource”. The name parameter represents the name of the DAO, and the datasource must correspond to the name of the jdbc global resource. Future implementations will expand this parameters so that you may use a non-web resource.

Inside the <dao> tag, you may have several <query> tags. Each query is a piece of SQL code that may be executed by itself. Each query must have at least a “name” attribute that is used to identify itself to the DAO. Optionally you may write a “type” attribute that describes the type of query (query, update, conditional).

The <query> tag has three basic components :

- the SQL code itself, I recommend you to put the SQL code inside “<![CDATA[“ delimiters, because the symbols “<” and “>” are sometimes interpreted by the xml parser and may generate errors.
- Parameters , most of the time queries are not static, they depend on parameters passed from your application. To use a parameter, you must use the “?” symbol on the SQL code, and use a <parameter> tag inside the <query>. The parameter is used to define the type of parameter are to be passed. There are currently four supported types “CHAR” for all Strings and characters, “NUMBER” for all numeric values (including double, long, and BigDecimal), “DATE” for date types, and “TIMESTAMP” when you need to use date and time together. The name of the parameter is passed as plain text inside the parameter tag.

- Results, whenever you are using a query that returns something, you must declare a name for this value, so that you can recover it inside your application. The `<result>` tag defines this parameter, it receives a single attribute “index” that defines the position in the result of the query, and you must provide a name for the value to be used inside your application. The type parameter in the result is optional; you only need to define it if you are uncertain about the automatic conversion.

Example of a query inside the XML file :

```
<query name="myquery" type="query">
  <![CDATA[
    select products.price
      from products
     where product.name = ?
  ]]>

  <parameter index="1" type="CHAR">name</parameter>
  <result index="1">price</result>
</query>
```

In the previous example, we are passing one “name” parameter of type “CHAR” to the query and we are suppose to retrieve only one value that will be treated as “price” inside our application.

## ***Executing the Queries***

Once you have an instance of a DAO in your hands, then you can execute any of the queries defined in the query file. There are several methods defined in the DAO for executing the queries :

**executeQuery** (queryName);

This is the most common type of query. This method will execute the query passing no parameters at all, and retrieving as many rows of data as possible. This method returns a Collection of QueryResult objects, each QueryResult object is an abstract representation of a row, from which you can retrieve the values retrieved by the query. The QueryResult object has several methods for implicit conversion of objects.

**executeQuerySingle** (queryName);

This query will return a single QueryResult object representing the first row retrieved by the Database, this method is useful for queries where you are sure to retrieve only a single row of information.

**executeQuery** (queryName, QueryParameter);

A QueryParameter object is a container to pass parameters to the query. You may create an instance of this object normally, and it has methods to store different types of objects and primitive values. It will respect the order on which the objects are stored to pass to the query. The *executeQuerySingle* method also accepts a QueryParameter as a second parameter.

**executeQuery** (queryName, QueryParameter, startRow, size);

Use this method when you need to retrieve a block of rows instead of bring all of the rows at once (as when implementing scrolling or paging of information). It will receive two extra parameters, the first is an int representing the number of the row where to start, and the second is the size of the block to be retrieved.

**executeUpdate**(queryName, QueryParameter);

This method is specific for updating actions such as “insert”, “update” and “delete” statements on the SQL.

## Setting Parameters and Retrieving values

Two objects are defined in the efreet package for dealing with values to and from the database. These objects are directly related to the tags <parameter> and <result> defined in the query file.

**QueryParameter** – this object acts like a Collection, and it’s used to set parameters for the queries. The values are processed by the query in the same order they are declared in the XML file. This object has several variations of the “put” method so it can receive different primitive types and also String, Date and BigDecimal objects. In previous versions of efreet there was also “add” methods, this are now deprecated as their use depends on the order of the parameters.

**QueryResult** – this object is returned by the query methods of the DAO, it’s an extension of a HashMap containing the values retrieved by the database mapped to the same names defined for each result in the query file. As his counterpart for parameters, the Query Result has several methods for retrieving specific primitive types and String, Date, and BigDecimal as well.

These objects are also very lenient in respect to null values. If a null value is passed it will treat it accordingly to pass it to the database as a null value.

**ParameterAdaptor** – this object is an adaptor created to simplify the job of creating the QueryParameter object, if you have well formed bean objects with accessor methods, you can use this adaptor to simply pass your bean as a parameter.

**ResultAdaptor** – Similar to the ParameterAdaptor, the job of the ResultAdaptor is to transform a QueryResult object into a bean of your own creation, it only depends on accessor methods of your bean to populate it.

## Transactions

By default, in the case of *executeUpdate* the DAO will do auto-commit for each update, in this way we don't have to worry about commit or even closing the connection with the database (a recurrent problem in many applications).

But, what if I do want to process a whole bunch of updates and only commit if they all succeed ? Then you can set the DAO to Transaction Mode, but in this case you are responsible for doing the commit (and or rollback) steps by yourself in the code. The following example shows how to use the DAO in transaction mode.

Example :

```
...
    DataAccessObject dao = DAOFactory.createDAO("tm_example");

    dao.setTransactionMode(true);

    try {
        QueryParameter qp = new QueryParameter();
        qp.put("param1", "Parameter 1");
        qp.put("param2", "Parameter 2");
        qp.put("param3", 10);

        dao.executeUpdate("firstUpdate", qp);

        dao.executeUpdate("secondUpdate", qp);

        qp.put("param4", "Parameter 4");
        qp.put("param5", true);

        dao.executeUpdate("thirdUpdate", qp);

    } catch DAOException {
        dao.rollback();
    } finally {
        dao.close();
    }
}
...

```

## Conditional Queries

Sometimes a single query with parameters is not enough, as an example, suppose we want to implement some kind of search that gives the user different options to filter this search in our application. If we have 4 options, so that each option may be turned on or off, we would end with 16 different combinations for the search, that means we would have to write 16 different queries. If we were writing the query in our code directly, we could use the programming logic to add pieces of the SQL code to our query if a particular option is turned on. Using the efreet package there's also a way to achieve this result.

First, we must declare our query in the query file :

```
<query name="mainQuery">
```

```
<![CDATA[
select  products.name, products.price
        from  products
        where 1 = 1
]]>
  <result index="1">name</result>
  <result index="2">price</result>
</query>
```

Then, we can start to write conditional queries in the query file, for each piece of SQL code we may want (or not) to add to the query, including parameters:

```
<query name="qfilter1" type="conditional">
  <![CDATA[
    and products.color = ?
  ]]>
  <parameter index="1" type="CHAR">color</parameter>
</query>

<query name="qfilter2" type="conditional">
  <![CDATA[
    and products.price between ? and ?
  ]]>
  <parameter index="1" type="NUMBER">min_price</parameter>
  <parameter index="2" type="NUMBER">max_price</parameter>
</query>

<query name="qfilter3" type="conditional">
  <![CDATA[
    and products.release = ?
  ]]>
  <parameter index="1" type="DATE">release_date</parameter>
</query>
```

You can write as many conditional expressions as you like, the only difference is that executing a conditional query by itself will generate a database error.

Then, in your application you may write a code that look like this :

```
...
QueryParameter qp = new QueryParameter();

if (filter1 != null && filter1.length() > 0) {
  dao.appendConditionalToQuery("mainQuery", "qfilter1");
  qp.put("color", filter1);
}
if (filter2 != null && filter2.max > 0.0) {
  dao.appendConditionalToQuery("mainQuery", "qfilter2");
  qp.put("min_price", filter2.min);
  qp.put("max_price", filter2.max);
}
if (filter3 != null) {
  dao.appendConditionalToQuery("mainQuery", "qfilter3");
  qp.put("release_date", filter3);
}

dao.executeQuery("mainQuery", qp);
...
```

The method *appendConditionalToQuery* will change the behaviour of the “mainQuery”, by appending the conditional queries to the end of the query. And all the parameters for the query are also added to the end of the list of parameters. That way we can have a query that may be different every time we execute it. For example, suppose we pass only the values for the “filter2” variables, our final query would end up looking like this :

```
select products.name, products.price
  from products
  where 1 = 1
 and products.price between ? and ?
```

and we are going to process only the “filter2.min” and “filter2.max” parameters for this query.

Note : the “where 1 = 1” clause is a typical strategy used in cases like this, it has no effect on the query except to allow us to write all the conditionals as “AND” clauses.

## Variables

Conditional queries are very flexible and should be used in most of the cases, because it allow us to keep all our SQL code inside the query file, which in turn would allow us to better organize our code. But sometimes, writing a conditional query may be not enough. For this cases, the efreet package allows us to declare variables in our queries.

A variable is a simple string that will be replaced in the query prior to execution. Variables always begins with the “\${” symbol and end with a “}” symbol. That way we can easily identify a variable inside our SQL code. As an example :

```
<query name="variableExample">
<![CDATA[
select * from numbers where numbers.value in ( ${values} )
]]>
<result index="1">id</result>
<result index="2">value</result>
<result index="3">set</result>
</query>
```

In our application code, we may then use a method from the QueryParameter object to set the value of our variable, like this :

```
...
QueryParameter qp = new QueryParameter();

qp.setVariable("values", "1, 3, 5, 7, 9");

Collection col = dao.executeQuery("variableExample", qp);

if (col != null) {
  Iterator iter = col.iterator();
  while (iter.hasNext()) {
    QueryResult qr = (QueryResult) iter.next();
    if (qr != null) {
      logger.debug(qr.getInt("id"));
    }
  }
}
```

Efreet package – <http://sourceforge.net/projects/efreet>

```
doSomething(qr.getInt("value"),
            qr.getString("set"));
    }
}
...
```

During the execution, our query will look like this :

```
select * from numbers where numbers.value in ( 1, 3, 5, 7, 9 )
```

**Beware, you should ALWAYS prefer to use parameters and conditional expressions instead of variables. Why ? because efreet doesn't validate variables as it does with parameters, and one of the main purposes of efreet is to remove SQL code from your own Java Code, and variables can break this if not used properly.**