EasyORM library – TUTORIAL

Audience

This tutorial is designed for Java developers who are writing their applications using either plain JDBC or a more complex O/R mapping library, and wish to switch to a lightweight ORM library.

Prerequisites

Developers are assumed to have at least basic knowledge of Java and SQL. The knowledge of JDBC can also help (especially when debugging the EasyORM's source code) but is not really important to use the library.

Copyright

Disclaimer

Some of the content in this tutorial may contain errors or inaccuracies. I, the author, shall not be held liable for any damage that might result from using this tutorial or the library.

1. The EasyORM library overview

This library targets primarily smaller to mid-sized projects. It does not need a web container (e.g. Tomcat, Jetty), though it can certainly be used with one. It is designed to be very much independent of any third party components so even logging capabilities have been integrated. Virtually no configuration is necessary and everything is done in code.
The library is based on the active record pattern and is thus best suited for applications that already have an object model that well matches the database object model it maps. Also, applications with a relatively simple database model will certainly benefit from using EasyORM because using a more complex library or framework often means that more time will be spent on resolving configuration and deployement issues.

Main features:

a) wraps the JDBC API
b) implementation is based on the active record pattern (enhanced to an active range through delegated persistance)
c) supports only DML (read, update, insert, delete) operations
d) supports tables/views as well as custom views (those that do not physically exist in a database)
e) uses its own connection pooling
f) implements transactions
g) integrated logging

2. The API

This will be added later. However, the API is very small and you should be fairly comfortable with using the library after going through the examples given below.

3. Environment/installation

In order to be able to use the library in your own projects, you basically have two options

a) Download and add EasyORM.jar to your project build path

b) Download EasyORM.zip and extract it to your workspace as a Java project. Now you can reference the EasyORM project from other projects (you'll want to do this if debugging into the EasyORM source code)

Note that you will also have to add a JDBC driver to your project build path.

4. Connection pool

EasyORM uses its own connection pool . Typically, you'll create a connection pool (or get a reference to it) before you begin executing queries against a database. Connections are reused by the pool so they don't have to be recreated but this only works if you use transactions (see below).

5. Database transactions

We usually define a database transaction as a set of related database operations which are treated as a single unit of work. This means that either all operations will succeed entirely or fail.
As an example, let's examine our Employee table with a column named address_id, which is a foreign key to the Address table. Now if we insert a record into the Address table and fail to insert a corresponding employee in the Employee table, our address table entry will have no associated employee (this is known as an orphaned record), so we basically corrupted the data.

Database transactions are a means of keeping the database consistent. While it's clear that transactions are important when we have multiple updates, they are important with reads as well, because , while reads don't change the state of the database, they can still hold locks on some data (or ranges of data), depending on the isolation level applied.
In addition to all this, connection reusing in EasyORM is implemented through transactions so it's best to use transactions even when you don't do updates on related tables.

6. Persistance object/class

Normally, any Java class that extends DBObject is considered persistent. This assumes that a DBObject implementing class has an associated database table or view (to which an instance of the class can be persisted). In case of a custom view (one that has no physical table or view), only reading is possible. Note that a persistent class must override the getPackageName method. It also has to use the @TableInfo annotation (if it maps a database table or view), which specify the name of the table / view as well as the identifier column (usually a primary key column). The @AttributeInfo annotation is optional but could be used when the persistant class contains another class/object that extends DBObject (see the example class below)

a) Example of a persistence class

```
@TableInfo(tableName = "employee",tableIdColumnName="id")
public class EmployeeDB extends DBObject implements Serializable {

	public EmployeeDB(ResultSet rs) throws  EasyORMException {
		super(rs);
	}
	public EmployeeDB(DBTransaction dbTrx) throws  EasyORMException {
		super(dbTrx,EmployeeDB.class);
	}
	public EmployeeDB(){}
	//Employee table columns
	public static String COLUMN_ID= "id";
	public static String COLUMN_NAME= "first_name";
	public static String COLUMN_SURNAME = "last_name";
	public static String COLUMN_YEARS_WORK = "years_at_job";
	public static String COLUMN_DEPARTMENT = "department";
	public static String COLUMN_DATE_ADDED = "date_added";
	public static String COLUMN_DATE_UPDATED = "date_updated";
	public static String COLUMN_ADDRESS_ID = "address_id";

	//custom view column name - corresponds to Certificate.cert_name
	public String COLUMN_CERT_NAME = "cert_name";

	//holds address information but does not correspond to any column name
	public AddressDB empAddress;

	@Override
	public String getPackageName(){
		return getClass().getPackage().getName();
	}
	protected String getIdentifierColumnName() {
		return COLUMN_ID;
	}
	public AddressDB getAddress(){
		return empAddress;
	}
	@AttributeInfo(attributeType="AddressDB")
	public void setAddress(AddressDB address){
		empAddress=address;
	}
	public Integer getId () {
		return (Integer)getValue(COLUMN_ID);
	}

	public void setId (java.lang.Integer id) {
		setValue(COLUMN_ID, id);
	}
	public Integer getAddressId () {
		return (Integer)getValue(COLUMN_ADDRESS_ID);
	}
	public void setAddressId (java.lang.Integer id) {
		setValue(COLUMN_ADDRESS_ID, id);
	}
	public Date getDateAdded () {
		return (Date)getValue(COLUMN_DATE_ADDED);
	}
	public void setDateAdded (Date dateAdded) {
```

```
            setValue(COLUMN_DATE_ADDED, dateAdded);
      }
      public Date getDateUpdated () {
            return (Date)getValue(COLUMN_DATE_UPDATED);
      }
      public void setDateUpdated (Date dateUpdated) {
            setValue(COLUMN_DATE_UPDATED, dateUpdated);
      }
      public String getName () {
            return (String)getValue(COLUMN_NAME);
      }
      public void setName (String name) {
            setValue(COLUMN_NAME, name);
      }
      public String getSurname () {
            return (String)getValue(COLUMN_SURNAME);
      }
      public void setSurame (String surname) {
            setValue(COLUMN_SURNAME, surname);
      }
      public Integer getYearsAtWork () {
            return (Integer)getValue(COLUMN_YEARS_WORK);
      }
      public void setYearsAtWork (int years) {
            setValue(COLUMN_YEARS_WORK, years);
      }
      public String getDepartment () {
            return (String)getValue(COLUMN_DEPARTMENT);
      }
      public void setDepartment (String department) {
            setValue(COLUMN_DEPARTMENT, department);
      }
      public String getCertificateName () {
            return (String)getValue(COLUMN_CERT_NAME);
      }
      }
```

b) Example of persisting employee and address objects to the database

```
public class SimpleEasyORMApp {

      public static void main(String[] args) {

            final String jdbcDriver = "org.postgresql.Driver";
            final String jdbcURL    =
            "jdbc:postgresql://localhost:5432/SimpleWebTest";
            final String user       = "postgres";
            final String password   = "1111";
            AddressDB address=null;
            EmployeeDB emp = null;
            DBTransaction dbTrx=null;
            try{
                  //-obtain a reference to the Connection pool
                  ConnectionPool connPool =
                  ConnectionPool.getInstance(jdbcDriver, jdbcURL, user,
                  password);
                  //create a transaction object that will be used for
                  commits/rollbacks but it also reuses connections from the pool
                  dbTrx=new DBTransaction(connPool);
```

```
//now create address and emp objects and populate them with
data
java.sql.Date sqlDate=java.sql.Date.valueOf("2015-06-19");
System.out.println("Creating Address and Employee objects");
address = new AddressDB(dbTrx);
address.setCity("London");
address.setStreetName("Morgan avenue");
address.setStreetNumber("2");
address.setDateAdded(sqlDate);
address.setDateUpdated(sqlDate);

System.out.println("Inserting record for Address");
int addId=address.insert();
System.out.println("Record for Address inserted with id:
"+addId);

emp = new EmployeeDB(dbTrx);
emp.setName("Peter");
emp.setSurame("Mason");
emp.setDepartment("Finances");
emp.setYearsAtWork(3);
emp.setAddressId(addId);//or address.getId()
emp.setDateAdded(sqlDate);
emp.setDateUpdated(sqlDate);

System.out.println("Inserting record for Employee");
int empId=emp.insert();
System.out.println("Record for Employee inserted with id:
"+empId);

System.out.println("Commiting to Address/Employee tables");
dbTrx.commit();
catch(EasyORMException e){
System.out.println("Exception: "+e);
}}}
```

In the code above we first get a reference to the connection pool by using

```
ConnectionPool connPool =
ConnectionPool.getInstance(jdbcDriver, jdbcURL, user,
password);
```

Alternatively, we could have used a different overloaded getInstance method e.g `ConnectionPool.getInstance`(String propertyFile, boolean useJndiName) if the connection properties were put in a property file. The property file (e.g. Connection.properties) may look like below(properties in bold should keep their names)

dbUrl=jdbc:postgresql://localhost:5432/SimpleWebTest
dbDriverName=org.postgresql.Driver
dbUserName=postgres
dbPassword=1111
dbDataSource=java:/comp/env/jdbc/SimpleWebTest

The useJndiName argument (if set to true) says that a data source will be used to create a connection (the data source corresponds to the dbDataSource) . You should set useJndiName to true only if you're using EasyORM with a web container like Tomcat (typically, context.xml within Tomcat's conf directory will have a data source entry)

Once we have a reference to the connection pool, we can create a transaction object thus

```
dbTrx=new DBTransaction(connPool);
```

and use this object when creating Address and Employee objects respectively

```
address = new AddressDB(dbTrx);
....
emp = new EmployeeDB(dbTrx);
```

The transaction object (dbTrx) is not absolutely necessary there and we could have used a no-args constructors to create these two objects, but as I said earlier, transaction context guarantees that either both inserts will succeed or both will fail, thus maintaining database consistency.

We save the two objects to the database by calling insert()

```
int addId=address.insert();
...
int empId=emp.insert();
```

Note that the insert method (and update()/delete() for that matter) when used within a transaction context doesn't commit the objects to the database. In fact, permanent persistance will take place only after a call to commit() has been made on the transaction object.

```
dbTrx.commit(); //this actually persists the objects to the db
```

Now, let's update this inserted record. Since we already have the object (emp) , all we have to do is set the attributes (columns) to update and call update() on the object. For instance

```
emp.setDepartment("Human Resources");
emp.update();
```

To delete this record we would just call

```
emp.delete();
```

If we hadn't had this object, we would first have had to obtain the object by calling one of the DBSelect class methods, e.g `getRecordsForParamQuery` or g `getRecordsForCustomQuery`

```
HashMap<String,Object> hMap=new HashMap<String,Object>();
hMap.put("firstName","Peter");
hMap.put("lastName","Mason");
List<EmployeeDB> empList=dbSelect.getRecordsForParamQuery("select * from employee
where first_name=:firstName and last_name=:lastName",hMap,EmployeeDB.class,0,0);
```

Assuming there is only one employee by the name of Peter Mason you would do the following to update the object's state

```
EmployeeDB emp=empList.get(0);
emp.setDepartment("Human Resources");
emp.update();
```

EasyORM supports a concept of delegated persistence. This means that an active record can be used to persist not only its own state but also the states of other objects. This pattern makes sense only for update and delete operations where the where-clause is the same. For instance, if we were to find all records having 'Financies' in the department column and update them to 'Financial', we could do something like

```
query="SELECT id FROM employee WHERE department='Financies'";
List<EmployeeDB> employeeList = dbSelect.getRecordsForCustomQuery(query,
EmployeeDB.class, 0, 0);
Integer [] ids=new Integer[employeeList.size()];
int i=0;
for(EmployeeDB emp:employeeList){
      emp.setDateUpdated(sqlDate);
      emp.setDepartment("Financial");
      //emp.update();this would be inefficient
      ids[i++]=emp.getId();//fill the array
                        }
EmployeeDB emp=employeeList.get(0);//we could've used any object from the list
emp.updateRange(ids);
```

The updateRange method takes an array as its argument and affects a number of rows in the Employee table (depending on the where clause). This approach is certainly much more efficient than having to call the update method on every single object that needs to be updated.

A class that extends DBObject should normally provide getter and setter methods for every column that exists in the table it maps (see EmployeeDB above). However, it could also add additional methods (getter/setter) to support queries that return data that are not found in the mapped table. For instance, this query

```
query=" select emp.*, cer.cert_name from employee emp, certificate cer, "+
"employee_certificate ec where emp.id=ec.emp_id and ec.cert_id=cer.id ";
```

returns all columns from the employee table plus cert_name from the certificate table. In this case we can simply add one field (cert_name) to the EmployeeDB class as well as the corresponding getter and setter methods.
This approach is fine some queries but may not be practical if a query returns multiple objects. For example, the following query

```
query=="select employee.*, address.* from  employee left outer join address on
employee.address_id=address.id ";
```

returns all columns from both the employee and address tables. In this case, it makes more sense to add an Address object to the EmployeeDB class and have an AttributeInfo annotation on the setAddress method.

```
@AttributeInfo(attributeType="AddressDB")
      public void setAddress(AddressDB address){
            empAddress=address;
      }
```

EasyORM will use this annotation to create and populate the Address object automatically. Alternativelly , you don't have to use the @AttributeInfo annotation, but you will have to call the createChildObject method manually.

```
for(EmployeeDB emp:employeeList){
      emp.setAddress((AddressDB)emp.createChildObject(AddressDB.class));
                        }
```

Note that the AddressDB class must provide a constructor that has Object as its argument, if it is to be used within another class/object (as in the example above).

```
...
public <T>AddressDB(Object enclosingCls)  {
            super(AddressDB.class,enclosingCls);
      }
...
```

7. Integrated logging

Integrated logging just means that EasyORM doesn't need a third party logger (e.g. log4j) to do the logging. Instead, programmers can create an instance of the Logger class and use the debug, info and error methods on the Logger object. Still, you will need to specify a log file.

```
Logger.setLogFileName("C:\\test\\logger.log");//set file only once
Logger logger = new Logger(LogTester.class);
logger.enableAllLogMsgTypes();//enable every log type
logger.disbleLogMsgType(LogMsgType.MSG_LOG_ERROR);//disable error
message
logger.info("This is the first info line");
logger.info("This is the second info line");
logger.debug("This is the first debug line");
logger.info("This is the third info line");
logger.error("This is the first error line");//this will not end up
in the log file because it has been disabled
logger.flush();//this writes to file everything that has not been
written and closes the file stream
```