Apache-Empire-db tutorial

Introduction to Apache-Empire-db using the **empire-db-example-basic** project

Introduction

This tutorial demonstrates in eight easy steps how to work with Apache-Empire-db and perform the most common tasks of data persistence: inserting, updating and deleting records as well as performing queries with a simple and ready to run example application. For more advanced topics like complex queries, bulk operations and dynamic data model changes there is a second example application provided with the release.

The sample application used for this tutorial is provided with the Apache-Empire-db distribution and can be found in the ***empire-db-examples/empire-db-example-basic*** directory. In order to open the project in your favourite IDE Maven may be able to generate a project file for you (e.g. for Eclipse run “mvn eclipse:eclipse”). Otherwise you have to create a new project and import all files and dependencies manually. In your IDE, set a breakpoint on the first line of the main method in SampleApp.java and debug the sample.

The sample works with a HSQL database that is provided with the distribution files, and allows the application to run "off the shelf". However it can easily be reconfigured to work with a Microsoft SQL-Server or Oracle database instead by altering the ***config.xml*** file.

In detail the tutorial shows the following:

- How to declare your data model using Java classes and objects
- How to generate a DDL-Script from the data model definition and create all database objects on the server
- How to insert, update and delete individual records
- How to perform a database query including joins and constraints
- How to access the query results either sequentially row by row, as a bean list or as an XML document

Structure of the Sample Application

The sample application is made up of four files:

1. **SampleDB.java**: This is the database definition file which contains the database layout with tables, indices, relations, views, etc.
2. **SampleApp.java**: This class includes the applications main method which calls other methods to perform the individual tasks. Set a breakpoint here to step though the application using your debugger.
3. **SampleConfig.java**: This class provides access to configuration settings, which are read from the configuration file config.xml.
4. **SampleBean.java**: This class is used to store query results in a simple Plain Old Java Object (POJO). The properties of this class only hold the fields of the query and not all the fields of a full database entity.

Note: In order to run the sample on a database system other than the HSQLDB provided, first the corresponding JDBC driver has to be added to the classpath. Afterwards the settings for the database provider and the JDBC connection have to be adjusted accordingly in the file config.xml.

```
<properties>
<!-- provider name must match the property-section containing the connection data -->
       <databaseProvider>hsqldb</databaseProvider>
</properties>

<properties-hsqldb>
       <!-- jdbc properties -->
       <jdbcClass>org.hsqldb.jdbcDriver</jdbcClass>
       <jdbcURL>jdbc:hsqldb:file:hsqldb/sample;shutdown=true</jdbcURL>
       <jdbcUser>sa</jdbcUser>
       <jdbcPwd></jdbcPwd>
       <schemaName>DBSAMPLE</schemaName>
</properties-hsqldb>
```

Within the project you will also find a file called SampleOutput.txt which contains the console output of the sample application. This shows the SQL as well as the query results for all individual steps.

Data model definition (SampleDB.java)

For the data model definition we first create a «database» class called SampleDB, which inherits from org.apache.empire.db.DBDatabase. All tables (and possibly views) of the data model are declared as ***public final*** fields which are assigned to the corresponding table or view object. Additionally in the constructor a foreign key relation from the employees table to departments table is added.

```
// Table members
public final Departments  DEPARTMENTS = new Departments(this);
public final Employees    EMPLOYEES   = new Employees(this);

/**
 * Constructor SampleDB
 */
public SampleDB()
{
       // Define foreign key relations
       addRelation(EMPLOYEES.DEPARTMENT_ID
                     .referenceOn(DEPARTMENTS.DEPARTMENT_ID ));
}
```

In our example we create an individual class for each of the two tables. Instead of normal classes however, we use nested classes here. While not necessary, this is a convenient way of keeping the whole data model in one file. This is sensible since it contains no or only little logic.

```
public final class EMPLOYEES extends org.apache.empire.db.DBTable
{
    public DBTableColumn EMPLOYEE_ID;
    public DBTableColumn LASTNAME;
    public DBTableColumn GENDER;
     ...
     // Constructor for the table
    public Employees(DBDatabase db)
    {
     super("EMPLOYEES", db);
        // ID
     EMPLOYEE_ID = addColumn("EMPLOYEE_ID",   DT_AUTOINC,  0, true,
```

```
      "EMPLOYEE_ID_SEQUENCE");
      LASTNAME  = addColumn("LASTNAME",        DT_TEXT,    40, true);

      GENDER    = addColumn("GENDER",          DT_TEXT,     1, false);
         ...
      // Primary key
      setPrimaryKey(EMPLOYEE_ID);
      // Set other indeces
      addIndex("EMPLOYEE_NAME_IDX", true,
               new DBColumn[] { FIRSTNAME, LASTNAME, DATE_OF_BIRTH });
      // Set timestamp column to save updates
      setTimestampColumn(UPDATE_TIMESTAMP);

       // Create Options for GENDER column
       Options genders = new Options();
       genders.set("M", "Male");
       genders.set("F", "Female");
       GENDER.setOptions(genders);
      }
}
```

In the constructor of each table class we add the columns and assign the column object to a ***public final*** field. This will allow us to browse and access the column objects directly from our source code.

Afterwards we set the primary key, add other indices and set a timestamp field which is internally used for optimistic locking.

Finally additional column metadata is added here, which could however also be added elsewhere.

Naming convention note: Since we declare all table and column objects as ***public final*** fields we write them in all upper-case letters. In order to simplify browsing of these properties using code completion and get them all well grouped together you might as well additionally add a prefix like T_ for tables and C_ for columns, which we recommend but have not done in this example. It's up to you whether you want to stick to these conventions or not.

Empire-db SampleApp – Step by step

When running the sample application the entry point is the main method found in SampleApp.java. This will perform the following steps:

Step 1 – Step 3: Set up a database connection and open the database

First the application needs to open a database connection to the database server. For this a jdbcClass, jdbcURL, jdbcUser and jdbcPwd must be provided with the configuration in config.xml. The configuration file is parsed and read by calling `config.init()`. To use a different configuration file, this filename can be passed to the main method as an argument. Afterwards a JDBC connection is created.

```
// Init Configuration
config.init((args.length > 0 ? args[0] : "config.xml" ));
// STEP 1: Get a JDBC Connection
Connection conn = getJDBCConnection();
```

In step two the sample creates and initializes a database driver object for the target DBMS. This is HSQLDB by default.

```
// STEP 2: Choose a driver
DBDatabaseDriver driver = getDatabaseDriver(config.getDatabaseProvider());
```

Then in step three the database object is opened using the driver. Only when opened, other methods of the database object may be used. Finally we check whether or not our database objects exist.

```
// STEP 3: Open Database and check if tables exist
db.open(driver, conn);
databaseExists(conn);
```

In order to check existence of the database the sample simply performs a query on the Departments table (`"select count(*) from DEPARTMENTS"`) using the following code:

```
DBCommand cmd = db.createCommand();
cmd.select(db.DEPARTMENTS.count());
db.querySingleInt(cmd.getSelect(), -1, conn);
```

If the select succeeds then the database is assumed to exist and step 4 is skipped. Otherwise step 4 will create all database objects.

Step 4 – Create a DDL script and the database objects

Based on the database definition in the class SampleDB.java a DDL script for the whole database is generated for the creation of all tables, sequences, indices and relations.

```
// create DLL for Database Definition
String ddlScript = db.getCreateDDLScript(driver);
```

Now the individual DDL commands are extracted and executed line by line using the driver's `executeSQL()` method.

Note: If you want to create or delete individual database objects such as tables, views, columns and relations you can obtain the corresponding SQL by calling the driver's `driver.getDDLCommand()` method.

Step 5 – Delete data records

This step empties the two tables of the database by deleting all data records:

```
DBCommand cmd = db.createCommand();
// Delete all Employees (no constraints)
db.executeSQL(cmd.getDelete(db.EMPLOYEES), conn);
// Delete all Departments (no constraints)
db.executeSQL(cmd.getDelete(db.DEPARTMENTS), conn);
```

Step 6 – Insert data records

In this step we add sample records for both the Departments and Employees table.

```
// Insert a Department
DBRecord rec = new DBRecord();
rec.create(db.DEPARTMENTS);
rec.setValue(db.DEPARTMENTS.NAME, "Development");
rec.setValue(db.DEPARTMENTS.BUSINESS_UNIT, "ITTK");
rec.update(conn);

// Insert an Employee
DBRecord rec = new DBRecord();
rec.create(db.EMPLOYEES);
rec.setValue(db.EMPLOYEES.FIRSTNAME, "Peter");
rec.setValue(db.EMPLOYEES.LASTNAME, "Sharp");
rec.setValue(db.EMPLOYEES.GENDER, "M");
rec.setValue(db.EMPLOYEES.DEPARTMENT_ID, 1);
rec.update(conn);
```

For the above code Empire-db generates and executes the following insert statements:

```
INSERT INTO DEPARTMENTS( DEPARTMENT_ID, NAME, BUSINESS_UNIT, UPDATE_TIMESTAMP)
VALUES ( 2, 'Development', 'ITTK', '2008-01-08 07:31:11.120')

INSERT INTO EMPLOYEES( EMPLOYEE_ID, FIRSTNAME, LASTNAME, DEPARTMENT_ID, GENDER, RETIRED,
UPDATE_TIMESTAMP)
VALUES ( 1, 'Peter', 'Sharp', 1, 'M', 0, '2008-01-08 07:31:11.151')
```

The database driver creates the ID values automatically using either sequences provided by the target DBMS or by using a special internal table for sequence number generation. The decision how sequence numbers are generated is the responsibility of the DBDatabaseDriver used.

If an update timestamp field has been declared for the table, it is automatically managed by Empire-db and used for optimistic locking.

Step 7 – Update data records

Step seven demonstrates how to update records. The following code gives an example of how to update an employee's phone number.

```
// Update an Employee
DBRecord rec = new DBRecord();
rec.read(db.EMPLOYEES, idPers, conn);
rec.setValue(db.EMPLOYEES.PHONE_NUMBER, '+49-7531-457160');
rec.update(conn);
```

For this code Empire-db generates the following update statement:

```
UPDATE EMPLOYEES
SET PHONE_NUMBER='+49-7531-457160',
             UPDATE_TIMESTAMP='2008-01-08 07:31:11.183'
WHERE EMPLOYEE_ID=1 AND UPDATE_TIMESTAMP='2008-01-08 07:31:11.150'
```

Important issues to notice:

1. Only changed fields are written in the database. You may check the modification status of the record or an individual field at any time and react on it if necessary.
2. The update timestamp is generated automatically. By using a constraint on the timestamp column, Empire-db checks whether the record has been concurrently changed by another user.
3. You may extend the DBRecord by creating special entity record classes for your database entities. This will not only provide further type-safety but also allows you to add new methods or override existing ones for custom behaviour.

Step 8 – Perform a query and access the results

Finally, this step shows how to perform a database query and how to access the query results.

For our example we query some employee fields from the database. The LASTNAME and FIRSTNAME fields are concatenated to provide us with the full name of the employee. Additionally we join the Departments table in order to retrieve the name of the department an employee belongs to. Finally we add constraints and the order in which we want the data to be supplied from the DBMS.

In order to create this statement we need a DBCommand object. This can be obtained from our database object. The command object offers methods whose names match those of the SQL keywords select, where, group by, having and order by. There is no need to specify which table we want to access our data from. However joins must be manually declared using the join() method.

The final code looks as follows:

```
// Define the query
DBCommand cmd = db.createCommand();
DBColumnExpr EMPLOYEE_FULLNAME= db.EMPLOYEES.LASTNAME.append(", ")
                          .append(db.EMPLOYEES.FIRSTNAME).as("FULL_NAME");
// Select required columns
cmd.select(db.EMPLOYEES.EMPLOYEE_ID, EMPLOYEE_FULLNAME);
cmd.select(db.EMPLOYEES.GENDER, db.EMPLOYEES.PHONE_NUMBER);
cmd.select(db.DEPARTMENTS.NAME.as("DEPARTMENT"));
cmd.select(db.DEPARTMENTS.BUSINESS_UNIT);
// Set Joins
cmd.join(db.EMPLOYEES.DEPARTMENT_ID, db.DEPARTMENTS.DEPARTMENT_ID);
// Set contraints and order
cmd.where(EMP.LASTNAME.length().isGreaterThan(0));
cmd.orderBy(EMP.LASTNAME);
```

This will then generate the following SQL select statement:

```
SELECT t2.EMPLOYEE_ID, t2.LASTNAME + ', ' + t2.FIRSTNAME AS FULL_NAME, t2.GENDER,
t2.PHONE_NUMBER, t1.NAME AS DEPARTMENT, t1.BUSINESS_UNIT
FROM EMPLOYEES t2 INNER JOIN DEPARTMENTS t1 ON t1.DEPARTMENT_ID = t2.DEPARTMENT_ID
WHERE len(t2.LASTNAME)>0
ORDER t2.LASTNAME
```

Important issues to notice:

1. As you can see, except for column renaming, no string literals are necessary to create the query. This ensures maximum of compile-time safety. The code is also portable and not tied to a particular DBMS.
2. Constraints for filtering can easily be added using the command's `where()` method, which may be called any number of times. This allows to easily add constraints conditionally without affecting code readability as with string operations.
3. When creating your command all required table and column objects as well as SQL functions can easily be added using the IDE's code completion, which always gives you a selection of available objects. This prevents you from making typing mistakes and improves your productivity.

Finally we need to execute the query and print our query results. For the latter we show three different options for doing this:

1. Iterating through the results row by row.
2. Fetching a list of JavaBean / POJO objects each containing the data of one result row. For this we will use the SampleBean class whose properties match the query result (see SampleBean.java).
3. Obtaining an XML document element that contains the query results and even includes column metadata.

Option 1: This code shows how to iterate and print the results row by row:

```
// Open the reader using command object
DBReader reader = new DBReader();
reader.open(cmd, conn);
// Text-Output by iterating through all records.
while (reader.moveNext())
{
	System.out.println(reader.getString(EMP.EMPLOYEE_ID)
	+ "\t" + reader.getString(EMPLOYEE_FULLNAME)
	+ "\t" + EMP.GENDER.getOptions() .get(reader.getString(EMP.GENDER))
	+ "\t" + reader.getString(DEP.NAME));
}
```

Option 2: This code shows how to fetch a list of sample beans

```
// Open the reader using command object
DBReader reader = new DBReader();
reader.open(cmd, conn);
// Text-Output using a list of Java Beans supplied by the DBReader
List<SampleBean> beanList = reader.getBeanList(SampleBean.class);
System.out.println(String.valueOf(beanList.size())
	+ " SampleBeans returned from Query.");
for (SampleBean b : beanList)
{
	System.out.println(b.toString());
}
```

Option 3: To obtain the result as an XML document the following code is required:

```
// Open reader
DBReader reader = new DBReader();
reader.open(cmd, conn);
// XML output
Document doc = reader.getXmlDocument();
// Print XML document to System.out
XMLWriter.debug(doc);
```

The XML document obtained by the above code looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<rowset>
	<column key="1" mandatory="1" name="EMPLOYEE_ID"></column>
	<column mandatory="1" name="FULL_NAME" size="40"></column>
	<column name="GENDER" size="1">
		<option value="M">Male</option>
		<option value="F">Female</option>
	</column>
	<column name="PHONE_NUMBER" size="40"></column>
	<column mandatory="1" name="DEPARTMENT" size="80"></column>
	<column mandatory="1" name="BUSINESS_UNIT" size="4"></column>
	<row>
		<EMPLOYEE_ID>41</EMPLOYEE_ID>
		<FULL_NAME>Bloggs, Fred</FULL_NAME>
		<GENDER>M</GENDER>
		<PHONE_NUMBER>+49-5555-505050</PHONE_NUMBER>
		<DEPARTMENT>Development</DEPARTMENT>
		<BUSINESS_UNIT>ITTK</BUSINESS_UNIT>
	</row>
	<row>
		<EMPLOYEE_ID>40</EMPLOYEE_ID>
		<FULL_NAME>Sharp, Peter</FULL_NAME>
		<GENDER>M</GENDER>
		<PHONE_NUMBER>+49-7531-457160</PHONE_NUMBER>
		<DEPARTMENT>Development</DEPARTMENT>
		<BUSINESS_UNIT>ITTK</BUSINESS_UNIT>
	</row>
	<row>
		<EMPLOYEE_ID>42</EMPLOYEE_ID>
		<FULL_NAME>White, Emma</FULL_NAME>
		<GENDER>F</GENDER>
		<PHONE_NUMBER>+49-040-125486</PHONE_NUMBER>
		<DEPARTMENT>Sale</DEPARTMENT>
		<BUSINESS_UNIT>ITTK</BUSINESS_UNIT>
	</row>
</rowset>
```

This XML can easily be transformed to another syntax by applying XSLT transformations. The metadata supplied here can be crucial for these transformations. Besides the metadata provided here even more and custom metadata may easily be added through column attributes.