Getting Started

by Brian McCallister

Table of contents

This document will guide you through the very first steps of setting up a project with OJB. To make this easier, OJB comes with a blank project template called ojb-blank which you're encouraged to use. You can download it here.

For the purpose of this guide, we'll be showing you how to setup the project for a simple application that handles products and uses MySQL. This is continued later on in the next tutorial parts.

1. Acquiring ojb-blank

First off, OJB uses Ant to build, so please install it prior to using OJB. In addition, please make sure that the environment variables `ANT_HOME` and `JAVA_HOME` are correctly set to the top-level folders of your Ant distribution and your JDK installation, respectively.

Next download the latest ojb-blank and OJB binary distributions. You can also start with the source distribution rather than the binary as the unit tests provide excellent sample code and you can build the ojb-blank project on your own with it.

The ojb-blank project contains all libraries necessary to get running. However, there may be additional libraries required when you venture deeper into OJB's APIs. See here for a list of additional libraries.
Most notably, you'll probably want to add the jdbc driver for you database unless you plan to use the embedded Hsqldb database for which the ojb-blank project is pre-configured (including all necessary jars).

2. Contents of ojb-blank

Copy the `ojb-blank.jar` file to your project directory and unpack it via the command

```
jar xvf ojb-blank.jar
```

This will unpack it into the `ojb-blank` directory under wherever you unpacked it from. You can move things out of that directory into your project directory, or, more simply, rename the `ojb-blank` directory to be whatever you want your project directory to be named.
After you unpacked the jar, you'll get the following directory layout:

```
\ojb-blank
    .classpath
    .project
    build.properties
    build.xml
    \lib
    \src
        \java
        \resources
        \schema
        \test
```

Here's a quick rundown on what the individual directories and files are:

.classpath, .project
An Eclipse project for your convenience. You can simply import it into Eclipse via **File -> Import... -> Existing Project into Workspace**.
build.xml, build.properties
The Ant build script and the build properties. These are described in more detail below.
lib
Contains the libraries necessary to compile and run your project. If you want to use a different database than Hsqldb, then put the jars of your jdbc driver in here.
src/java

Put your java source code here.
src/resources
Contains the runtime configuration files for OJB. For more detail see below.
src/schema
Here you will find a schema containing tables that are required by certain components of OJB such as clustered locking and OJB managed sequences. More information on these tables is available in the platform documentation. The schema is in a database-independent format that can be used by Torque or commons-sql to create the database.
The ojb-blank project contains the runtime files of Torque 3.0.2, and provides a build target that can be invoked on your schema (see below for details). Therefore, this directory also contains the build script of Torque, but you won't need to invoke it directly.
src/java
Place your unit tests in here.

2.1. Sample project

For our sample project, we should rename the directory to something more fitting, like `productmanager`.

Also, since we're using MySQL, we put the MySQL jar of the jdbc driver, which is called something like `mysql-connector-java-[version]-stable-bin.jar`, into the `lib` subdirectory.

The only other thing missing is the source code, but since that's what the other tutorials are dealing with, we will silently assume that it is already present in the `src/java` subdirectory.
If you don't want to write the code yourself, you can use the code from one of the tutorials which you can download here.

Warning:
Note that if you do not intent to use JDO, then you should delete the files in the `ojb.apache.ojb.tutorial5`, otherwise you'll get compilation errors.

3. The build files

3.1. Configuration via build.properties

The next step is to adapt the build files, especially the `build.properties` file to your environment. It basically contains two sets of information, the database settings and the build configuration. While you shouldn't have to change the latter, the database settings probably need to be adapted to suit your needs:

Property	Purpose
jcdAlias	The name of the connection. You should leave the default value, which is `default`.
databaseName	This is the name of the database, per default `ojb_blank`.
databaseUser	The user name for accessing the database (default: `sa`). If you're using Torque to create the database, then this user also requires sufficient rights to create databases and tables.
databasePassword	Password for the user, per default empty.

dbmsName	The type of database, which is one of the following: **Db2**, **Firebird**, **Hsqldb**, **Informix**, **MaxDB**, **MsAccess**, **MsSQL**, **MySQL**,**Oracle** (pre-9i versions), **Oracle9i**, **WLOracle9i** (Oracle 9i or above used from WebSphere), **PostgreSQL**, **Sapdb**, **Sybase** (generic), **SybaseASA**, **SybaseASE**. Please note that this setting is case-sensitive. Per default, Hsqldb is used, which is an embedded database. All files required for this database come with the ojb-blank project.
jdbcRuntimeDriver	The fully-qualified classname of the jdbc driver. For Hsqldb this is `org.hsqldb.jdbcDriver`.
jdbcLevel	The jdbc level that the driver conforms to. Please check the documentation of your jdbc driver for this value, though most jdbc drivers conform to version 2.0 at least. For the Hsqldb jdbc driver this is 2.0.
urlProtocol	The protocol of the database url (see below), usually `jdbc`.
urlSubprotocol	The sub-protocol of the database url which is database- and driver-specific. For Hsqldb, you're using `hsqldb`.
urlDbalias	This is the address that points the jdbc driver to the database. For Hsqldb this is per default the database name.
torque.database	If you're using Torque to create the database, then you have to set the database here (again). Unfortunately, this value is different from the `dbmsName` which defines the database for OJB. Currently, these values are defined: **axion**, **cloudscape**, **db2**, **db2400**, **hypersonic** (which is Hsqldb), **interbase** (use for Firebird), **mssql**, **mysql**, **oracle**, **postgresql**, **sapdb**, and **sybase**. Default value is `hypersonic` for use with Hsqldb.
torque.database.createUrl	This specifies the url that Torque will use in order to create the database. Depending on the database, this may be the same as the normal access url (the default value), but for some database this is different. Please check the manual of your database for this url.

If you know how the jdbc url for connecting to your database looks like, then you can derive the settings `databaseName`, `databaseName`, `databaseName` and `databaseName` easily:
Assume this url is given as:

```
jdbc:mysql://localhost:3306/myDatabase
```

then these properties are

Property	Value
databaseName	`myDatabase`
urlProtocol	`jdbc`

urlSubprotocol	`mysql`
urlDbalias	`//localhost/myDatabase`

3.2. Building via build.xml

After setting up the build you're probably eager to actually build the project. Here's the actions that you can perform using the Ant build file `build.xml`:

Action (target in the build.xml file)	What it does
clean	Cleans up all files from the previous build.
compile	Compiles your java source files to `build/classes`. Usually, you don't run this target, but rather the next one which includes the compilation step.
build	Compiles your java sources files (using the compile action), and prepares the runtime configuration files using the settings that you specified in the `build.properties` file, most notably the `repository_database.xml` which will be located in the `build/resources` directory after the build. After you run this action, your application is ready to go (if the action ran successfully, of course).
jar	A convenience action that packs your successfully build application into a jar.
xdoclet	Creates the runtime configuration files that describe the repository, from javadoc comments embedded in your java source files. Details on how to this are given in the tutorials and in the documentation of the XDoclet OJB module.
setup-db	Creates the database and tables from a database-independent schema using Torque. You'll find more info on this schema in the documentation of the XDoclet OJB module and on the Torque homepage.
enhance-jdori	This is a sample target that shows how a class meant to be persistent with JDO, is processed by the JDO bytecode enhancer from the JDO reference implementation. It uses the `Product` class from the JDO tutorial (tutorial 5).

So, a typical build would be achieved with this Ant call:

```
ant build
```

If you want to create the database as well, and you have javadoc comments in your source code that describe the repository, then you would call Ant this way:

```
ant build setup-db
```

This will perform in that order the actions `build`, `xdoclet` (invoked automatically from the next action) and `setup-db`.
Of course, you do not need to use Torque to setup your database, but it is a convenient way to do so.

3.3. Sample project

First we change the database properties to these values (assuming that Torque will be used to setup the database):

Property	Value
jcdAlias	We leave the default value of `default`.
databaseName	Since the application manages products, we call the database `productmanager`.
databaseUser	This depends on your setup. For the purposes of this guide, let's call him `steve`.
databasePassword	Again depending on your setup. How about `secret` (you know that you should not use this password in reality ?!).
dbmsName	`MySQL`
jdbcRuntimeDriver	Its called `com.mysql.jdbc.Driver`.
jdbcLevel	For the newer Mysql drivers this is 3.0.
urlProtocol	The default of `jdbc` will do.
urlSubprotocol	For MySQL, we're using `mysql`.
urlDbalias	Assuming that the database runs locally on the default port, we have `//localhost/${databaseName}`.
torque.database	We want to use Torque, so we put `mysql` here.
torque.database.createUrl	MySQL allows to create a database via jdbc. The url that we should use to do so, is the normal url used to access the database minus the database name. So the value here is: `${urlProtocol}:${urlSubProtocol}://localhost/`. Please note that the trailing slash is important.

Ok, now we have everything configured for building. The `build.properties` file now looks like this (the comments have been removed for brevity):

```
jcdAlias=default
databaseName=productmanager
databaseUser=steve
databasePassword=secret

dbmsName=MySQL
jdbcLevel=3.0
jdbcRuntimeDriver=com.mysql.jdbc.Driver
urlProtocol=jdbc
urlSubprotocol=mysql
urlDbalias=//localhost/${databaseName}

torque.database=mysql
torque.database.createUrl=${urlProtocol}:${urlSubprotocol}://localhost/

jar.name=projectmanager.jar

source.dir=src
source.java.dir=${source.dir}/java
source.resource.dir=${source.dir}/resources
source.test.dir=${source.dir}/test
source.schema.dir=${source.dir}/schema

build.dir=build
build.lib.dir=lib
build.classes.dir=${build.dir}/classes/
build.resource.dir=${build.dir}/resources/

target.dir=target
```

Looks like we're ready for building. Again, we're assuming that the source code is already present. So we're invoking Ant now in the top-level folder `productmanager`:

```
ant build setup-db
```

which should (assuming five java classes) produce an output like this

```
Buildfile: build.xml

compile:
    [mkdir] Created dir: /home/steve/projects/productmanager/build
    [mkdir] Created dir: /home/steve/projects/productmanager/build/classes
    [javac] Compiling 5 source files to
/home/steve/projects/productmanager/build/classes

build:
     [copy] Copying 10 files to
/home/steve/projects/productmanager/build/resources

xdoclet:
[ojbdoclet] (XDocletMain.start                   47  ) Running <ojbrepository/>
[ojbdoclet] Generating ojb repository descriptor
(build/resources//repository_user.xml)
[ojbdoclet] Type test.Project
[ojbdoclet] Processed 5 types
[ojbdoclet] Processed 5 types
[ojbdoclet] (XDocletMain.start                   47  ) Running <torqueschema/>
[ojbdoclet] Generating torque schema (build/resources//project-schema.xml)
[ojbdoclet] Processed 5 types

setup-db:

check-use-classpath:

check-run-only-on-schema-change:

sql-check:

sql:
     [echo] +------------------------------------------+
     [echo] |                                          |
     [echo] | Generating SQL for YOUR Torque project!  |
     [echo] | Woo hoo!                                 |
     [echo] |                                          |
     [echo] +------------------------------------------+

sql-classpath:
[torque-sql] Using contextProperties file:
             /home/steve/projects/productmanager/build.properties
[torque-sql] Using classpath
[torque-sql] Generating to file
/home/steve/projects/productmanager/build/resources/report.productmanager.sql.generation
[torque-sql] Parsing file: 'ojbcore-schema.xml'
[torque-sql] (transform.DTDResolver              128 ) Resolver: used
database.dtd from
             org.apache.torque.engine.database.transform package
[torque-sql] Parsing file: 'project-schema.xml'
[torque-sql] (transform.DTDResolver              140 ) Resolver: used
             http://jakarta.apache.org/turbine/dtd/database.dtd

sql-template:
```

```
create-db-check:

create-db:
[torque-data-model] Using classpath
[torque-data-model] Generating to file
/home/steve/projects/productmanager/build/resources/create-db.sql
[torque-data-model] Parsing file: 'ojbcore-schema.xml'
[torque-data-model] (transform.DTDResolver               128 ) Resolver: used
database.dtd from
                    org.apache.torque.engine.database.transform package
[torque-data-model] Parsing file: 'project-schema.xml'
[torque-data-model] (transform.DTDResolver               140 ) Resolver: used
                    http://jakarta.apache.org/turbine/dtd/database.dtd
     [echo]
     [echo]       Executing the create-db.sql script ...
     [echo]
      [sql] Executing file:
            /home/steve/projects/productmanager/build/resources/create-db.sql
      [sql] 2 of 2 SQL statements executed successfully

insert-sql:
[torque-sql-exec] Our new url -> jdbc:mysql://localhost/productmanager
[torque-sql-exec] Executing file:
/home/steve/projects/productmanager/build/resources/project-schema.sql
[torque-sql-exec] Executing file:
/home/steve/projects/productmanager/build/resources/ojbcore-schema.sql
[torque-sql-exec] 50 of 50 SQL statements executed successfully

BUILD SUCCESSFUL
```

That was it. You now have your database setup properly. Go on, have a look:

```
mysql -u steve productmanager

mysql> show tables;
```

There, all tables for your project, as well as the tables required for some OJB functionality which we also used in the above process (you can recognize them by their names which start with `ojb_`).

4. The runtime configuration files

The last thing missing for actually running your project is to adapt the runtime configuration files used by OJB. There are basically three sets of configuration that need to be provided: configuration of the OJB runtime, description of the database connection, and description of the repository.

4.1. Configuring the OJB runtime

With the OJB.properties file and OJB-logging.properties (both located in `src/resources`), you configure and finetune the runtime aspects of OJB. For a simple application you'll probably won't have to change anything in them, though.

4.2. Configuring the database connection

For projects that use OJB, you configure the connections to the database via jdbc connection descriptors. These are usually defined in a file called `repository_database.xml` (located in `src/resources`). In the ojb-blank project, the build file will setup this file for you and place it in the `build/resources` directory.

4.3. Configuring the repository

Finally you need to configure the repository. It consists of descriptors that define which java classes are mapped in what way to which database tables, and it is typically contained in the

`repository_user.xml` file. This is the most complicated configuration part which will be explained in much more detail in the rest of the tutorials.
An convenient way of creating the repository metadata is to use the XDoclet OJB module. Basically, you put specific Javadoc comments into your source code, which are then processed by the build file (`xdoclet` and `setup-db` targets) and the repository metadata and the database schema are generated.

4.4. Sample project

Actually, there is not much to do here. For our simple sample application the default properties of OJB work just fine, so we leave `OJB.properties` and `OJB-logging.properties` untouched.

Also, the build file generated the connection descriptor for us, and we were using the XDoclet OJB module and Torque to generate the repository metadata and database for us. For instance, the processed connection descriptor (file `build/resources/repository_database.xml`) looks like this:

```
<jdbc-connection-descriptor
    jcd-alias="default"
    default-connection="true"
    platform="MySQL"
    jdbc-level="3.0"
    driver="com.mysql.jdbc.Driver"
    protocol="jdbc"
    subprotocol="mysql"
    dbalias="//localhost/productmanager"
    username="steve"
    password="secret"
    eager-release="false"
    batch-mode="false"
    useAutoCommit="1"
    ignoreAutoCommitExceptions="false"
>
    <object-cache class="org.apache.ojb.broker.cache.ObjectCacheDefaultImpl">
        <attribute attribute-name="timeout" attribute-value="900"/>
        <attribute attribute-name="autoSync" attribute-value="true"/>
    </object-cache>
    <connection-pool
        maxActive="21"
        validationQuery="" />
    <sequence-manager
className="org.apache.ojb.broker.util.sequence.SequenceManagerHighLowImpl">
        <attribute attribute-name="grabSize" attribute-value="20"/>
        <attribute attribute-name="autoNaming" attribute-value="true"/>
        <attribute attribute-name="globalSequenceId" attribute-value="false"/>
        <attribute attribute-name="globalSequenceStart"
attribute-value="10000"/>
    </sequence-manager>
</jdbc-connection-descriptor>
```

If you're curious as to what this stuff means, check this reference guide.

The repository metadata (file `build/resources/repository_user.xml`) starts like:

```
<class-descriptor
    class="productmanager.Product"
    table="Product"
>
    <field-descriptor
        name="name"
        column="name"
        jdbc-type="VARCHAR"
        length="32"
    >
    </field-descriptor>
    <field-descriptor
        name="price"
        column="price"
        jdbc-type="FLOAT"
    >
    </field-descriptor>
    <field-descriptor
        name="stock"
        column="stock"
        jdbc-type="INTEGER"
    >
    </field-descriptor>
    <field-descriptor
        name="id"
        column="id"
        jdbc-type="INTEGER"
        primarykey="true"
    >
    </field-descriptor>
</class-descriptor>
...
```

Now you should be able to run your application:

```
cd build/resources

java productmanager.Main
```

Of course, you'll need to setup the `CLASSPATH` before running your application. You'll should add all jars from the `lib` folder except the ones for Torque (`torque-[version].jar`, `velocity-[version].jar` and `commons-collections-[version].jar`) and for the XDoclet OJB module (`xdoclet-[version].jar`, `xjavadoc-[version].jar` and `xdoclet-ojb-module-[version].jar`).

It is important to note that OJB per default assumes the `OJB.properties` and `OJB-logging.properties` files in the directory where you're starting the application. Hence, we changed to the `build/resources` directory before running the application. This of course requires the compiled classes to be on the classpath, as well (directory `build/classes`).

Per default, the same applies to the other configuration files (`repository*.xml`) but you can change this in the `OJB.properties` file.

5. Learning More

After you've have learned about building and configuring projects that use OJB, you should check out the tutorials to learn how to specify your persistent classes and how to use OJB's APIs to perform database operations. The Mapping Tutorial in particular shows you how to map your classes to tables in an RDBMS.