TDK Maven

Starting your project and database with TDK 2.2 and Maven

```
From: Wordman <lward@tagaudit.com>
To: "Turbine User Mailing List (E-mail)" <turbine-user@jakarta.apache.org>
Date: Wed, 15 Jan 2003 18:26:54 -0500
```

Introduction

The Turbine Developers Kit (TDK 2.2) contains a sample application. Instructions for getting started with this application exist but have some missing elements. Some of these are discussed in a mailing list thread.

To these points, I would add the following: Do not overlook setting turbine.app.name in step three of the howto. The setting you provide here is used to provide the package name for a bunch of source files and directories during the process.

Additionally, none of this documentation really speaks to the best way to use and alter the database created in the TDK sample. The TDK's build process will create the default Turbine tables for you. This is a neat process, but at the end of it, you have a bunch of generated code that, it turns out, you no longer need. This is very confusing, because it gives the impression that you need to keep this code around and base future database work on it.

This post suggests a typical way of adding tables and altering your new database. Some of this information is buried in the instructions on expanding the user table.

WARNING: I am not a Turbine expert. What you see here is just the results of my tinkering with the system and muddling my way through. It contains what I wish I would have known to start with. People who know better, please post corrections.

Note, this works with Turbine 2.2, but may not work with other versions.

This post assumes that you have followed the TDK's how to, and have successfully built your initial database and got the sample application up and running. It also assumes that you will have changed the project and package name to your own. In the examples that follow, the application being created is called "Killer", written by DivNull Software. Thus, the TDK's build.properties file contained the following (note the addition of tdk.home, which is missing from the version supplied with the TDK):

```
tdk.turbineVersion = 2
tdk.project = killer
tdk.home = .
target.package = com.divnull.killer
target.directory = com/divnull/killer
```

The example presented here uses MySQL, so in step 4 of the howto, the following items were added or altered in the <tdk>/webapps/killer/WEB-INF/build/build.properties file:

```
tdk.home = d:/programming/src/tdk-2.2 (or wherever your TDK is. Need to add this.)
database = mysql
databaseDriver = org.gjt.mm.mysql.Driver
databaseHost = 127.0.0.1
databasePassword = password
databaseUrl = jdbc:mysql://${databaseHost}/${database.name}
createDatabaseUrl = ${databaseUrl}
buildDatabaseUrl = ${databaseUrl}
databaseUser = root
```

[Note, you probably discovered that between step 4 and 5, you had to create the database by hand by going into mysql and running "create database killer".]

[NOTE (by rbaynes): if you use createDatabaseUrl = jdbc:mysql://${databaseHost}/
the 'ant init' command will automatically create the database for you. The reason the one listed above fails is the mysql jdbc driver can't connect to that url because the specified database does not exist yet.]

[Also make sure that your MySQL does accept TCP connections. Specifically, comment out skip-networking in your my.cnf. Debian has this issue.]

OK, so you've finished the howto and got the sample application running. Now what?

Moving The Sample

Well, if you are anything like me, the first thing you probably want to do is move this sample application somewhere else. Most people want to get the code into their own directory structure so they can put it under source control. Few people will want to continue developing within the TDK directory.

The Killer application uses the default Maven directory layout. So, to setup Killer as a Maven application using the sample Turbine app you just generated, do the following (where <root> is the root directory for your application development, such as D:\Projects\killer or \home\user\me\killer or whatever):

1) Create <root> and the shell of a minimal Maven layout, like so:

```
<root>
   maven.xml (optional)
   project.xml
   build
   src
      application.xml
      announcements
      dtd
      iutest
      java
      sql
      test
      webapp
   xdocs
      images
      target
```

2) Copy the contents of <tdk>/webapps/killer/ to <root>/src/webapp

3) Copy the contents of <root>/src/webapp/WEB-INF/src/java to
<root>/src/java

4) Copy the contents of <root>/src/webapp/WEB-INF/src/sql to <root>/src/sql

5) Copy the contents of <root>/src/webapp/WEB-INF/build to <root>/build

6) DELETE the <root>/src/webapp/WEB-INF/build directory

7) DELETE the <root>/src/webapp/WEB-INF/classes directory

8) DELETE the <root>/src/webapp/WEB-INF/lib directory. (The Maven process will take care of the libs for you, assuming you [http://maven.apache.org/reference/user-guide.html configure your project.xml file]. Remember to include your database's driver.)

9) DELETE the <root>/src/webapp/WEB-INF/src directory.

Adapting to the Move

You now have a standard Maven directory structure, but you need to take care of some fallout from making these moves. This fallout is related to the <root>/build/build.xml file, which is the Ant script you used to generate the database in the TDK. You will be doing two things: a) you'll be changing your build properties to adapt to the new directory structure and b) you'll be changing the build script to allow you to easily add additional tables to your database.

Adjust Properties

Edit the <root>/build/build.properties file first:

1) Adjust (or add) the tdk.home property. If you previously set this to an absolute path, this should not change. (I'd prefer to remove all dependencies on the TDK from my project, for for a couple of Byzantine reasons, that is beyond the scope of this document.)

2) Move the src.dir property towards the top of the file (e.g. right after tdk.home) and set it to <root>/src.

3) Create three new entries which (evidently) need to be overridden from how they worked when building the sample application:

```

torque.sql.dir=${src.dir}/sql
torque.schema.dir=${src.dir}/webapp/WEB-INF/conf
torque.java.dir=${src.dir}/java
```

4) Change the following properties to these values:

```
app.root = ${src.dir}/webapp
build.webappRoot = ${src.dir}/webapp
build.dest = ${src.dir}/../target/classes
conf.dir = ${torque.schema.dir}
master.conf.dir = ${tdk.home}/tdk/ancillary/${tdk.turbineVersion}/src/conf
lib.dir = ${tdk.home}/webapps/${tdk.project}/WEB-INF/lib
```

The lib.dir setting takes a bit of explaining, because the way it is setup in this document is a bit hacky. The lib.dir property is intended to tell the database build process where the jar files are that it needs to work. This is a bit of an issue for a Maven application, because of the way Maven works. One of the ideas of Maven is that you should not keep the jars upon which you are dependant under source control. Instead, you configure your project.xml file to tell Maven what jar files you use, and the build process of Maven will assemble these into your .war file when the time comes.

This is actually a cool feature of Maven, but it causes a bit of a snag here, because it means that your Maven project does not have a directory of convenient jar files at which lib.dir can be pointed. You do, however, still have the sample application you generated in the TDK directory. The above example setting will use that sample app's lib directory. You can use some other directory if you happen to have one, just be sure that it has the jars the build process needs (see the <tdk>/webapps/killer/WEB-INF/lib for what these are).

One note about build.dest as well: by default, a Maven project will create a
<root>/target/classes directory in which to build java classes, so it is used for build.dest. If you have not yet run Maven to build your application, this directory may not yet exist. The database build process will not create it for you, so you may have to create it by hand.

5) There are a few bugs to correct in <root>/build/build.xml (note, these may be corrected by the time you read this).

5a) In the path tag of the script, the dir attribute should be ${lib.dir}.

5b) In the check-webinf-exists target, the file attribute should be set to use the correct property. It should be ${app.root}/WEB-INF (I think).

6) You can test your settings by moving to <root>/build and running "ant init" again. WARNING: this will completely DESTROY and recreate your database. Since it is assumed that you just created this database when you built the sample application, this should not be a huge problem.

Prepare for Database Expansion

The <root>/build/build.xml Ant script does a good job of generating a default database. Unfortunately, it also has some issues that you need to be wary of:

- The act of creating the sample database generated a lot of code that got used once, but that you no longer need.
- The script, as written, now contains some targets that can be dangerous if you use them on a database you care about. The init target, for example, will destroy the database and rebuild it from scratch without warning. Depending on what you are doing, this may or may not be desirable.
- If adding tables to the system (particularly if they have foreign keys to turbine tables), the script could be made more useful.
- The Ant script has a few bugs (see above).

This section details changes to make to your environment to clean it up and allow you to more easily add new tables. (Again, this example uses the settings for the Killer application mentioned above. Your paths will differ.)

1) DELETE all .java files in <root>/src/java/com/divnull/killer/om and the map directory inside it. (Note, if you have already followed the howto on expanding the user table, do not delete your *Adapter files. These instructions assume you have not followed that howto. In fact, these instructions duplicate some of that howto.) These files were used to generate the default schema and, now that it is generated, you don't need them.

2) Rename <root>/src/webapp/WEB-INF/conf/id-table-schema.xml to id-table-schema-nogenerate.xml. This will prevent the sql for the id table from being rebuilt all of the time.

3) Rename your <root>/src/webapp/WEB-INF/conf/turbine-schema.xml file to turbine-schema-nogenerate.xml. This will prevent the build script from generating these tables and their associated code again. NOTE: you are now in dangerland, because if you call the "ant init" process now, the default turbine tables will .not. be created. So, we fix this in the next step.

4) With the turbine schema now taken out of the generator, the ability of the init target to destroy and create the database must be stopped. Fortunately, the author provided a simple way to do this. Add the following properties to <root>/build/build.properties:

```
database.manual.creation=true
noSecuritySQL=true
```

5) At this point the "ant init" target:

- will no longer destroy and create the database.
- will no longer generate sql for the turbine tables.
- will no longer generate om classes for the turbine tables.
- continues to generate sql and code for the killer-schema.xml file.
- continues to rebuild the TurbineResources.properties file by copying the TurbineResources.template file and substituting in the database information. (If you use the init target and make changes to your properties file, therefore, you should make changes to the .template version.)
- continues to execute the sql scripts in <root>/src/sql. Note that most of these scripts will DROP AND RECREATE the tables they define. This INCLUDES the Turbine table scripts.

6) (Optional) If you want to prevent the dropping and creation of the turbine tables by the ant script, you have a couple of options:

- Stop using "ant init". If you call "ant project-om", your Peer-based object model classes will be created in whatever you have specified for the targetPackage property in your <root>/build/build.properties file. Calling "ant schema-sql", the sql for your schema will be generated to <root>/src/sql. From there, you can manually execute it into your database (or, write a Maven/Ant task to execute specific files).
- Move the turbine sql files out of <root>/src/sql. The "insert-sql-files" target in the <root>/build/build.xml file will execute any *schema.sql file directly in that directory (I think). You could, for example, create <root>/src/sql/turbine and move the following files into it:

```
create-db.sql
sqldb.map
turbine-schema.sql
turbine-schema-idtable-init.sql
turbine-security.sql
```

(Note that this solution will NOT work with any .sql files that continue to be generated by the "schema-sql" target, since that target will regenerate the files if they are moved. Adding the "-nogenerate" to the turbine schema definitions prevents this.)

7) (Optional) Instead of (or in addition to) step six, you might want to just comment out the init target as being dangerous. To prevent dependency problems, the easiest thing is to add a fail statement before the first
<antcall> tag in the init target, like this one:

```
<fail message="The init task has been intentionally disabled to prevent accidental database problems."/>
```

8) (Optional) You may want to prevent the generation of the TurbineResources.properties from the .template file. Again, the solution is to avoid "init". Specifically, you want to skip the "update-tr-props" target.

My personal preference for generating the database is to move the turbine .sql into a "turbine" directory, and define a new target in the build.xml file (very similar to init, but used only for generating the database):

```
<target
  name="init-db"
  depends="setup-webinf"
  description="--> generates the database">

  <antcall target="create-database"/>
  <antcall target="schema-sql"/>
  <antcall target="idtable-init-sql"/>
  <antcall target="security-sql"/>
  <antcall target="insert-sql-files"/>
  <antcall target="project-om"/>
</target>
```

This can be called from the maven.xml file with the following code:

```
<ant dir="build" inheritAll="false" target="init-db"/>
```

Defining New Tables

After the all of this rigamarole, we come to the payoff. You can now just define new tables in the killer-schema.xml file and, when you run the "init-db" target (or something similar), Ant will build your SQL definitions for that table as well as java code defining the objects that read from and write to it. Even better, these objects automatically handle huge amounts of the drudgery of moving data between database and object. You can examine the contents of the turbine-schema-nogenerate.xml file to see how the syntax of the schema file works. It is fairly straightforward.

If you need to create a table that has a foreign key to the user table (which is likely), there is a howto that covers this (http://jakarta.apache.org/turbine/turbine/turbine-2.3.1/howto/extend-user-howto.html). Some of what you will be asked to do in the howto you have already done by following these instructions. There are a few spots of this howto that can use additional clarification, however.

1) Where the document mentions the project-schema.xml file, what is meant is the schema file specific to your application. In the case of the example in this document, that would be the killer-schema.xml file.

2) After running all of this sample app code and init target stuff, you might think that a few things are missing from the howto. For example, it talks about adding a Title to the turbine_user table, but the examples don't seem to actually use this field anywhere. How does the field get auto-generated? The answer is that it doesn't't; you have to add your extra fields to the turbine_user table manually. The howto does say this, but it is easy to miss.

3) The howto has you create three new classes (TurbineUser'***Adapter, TurbineUser***'PeerAdapter, TurbineMap'*_Builder), but before it does this, it has you add the following lines to the "project-schema.xml" file:

```
    <table name="EXTENDED_USER" alias="TurbineUser"
            baseClass="org.mycompany.sampleapp.om.TurbineUserAdapter"
            basePeer="org.mycompany.sampleapp.om.TurbineUserPeerAdapter">
        <column name="USER_ID" required="true" primaryKey="true" type="INTEGER"/>
    </table>
```

How this interacts with the build process is not explained very well. What ends up happening is that the name EXTENDED_USER ends up generating a new set of class files: ExtendedUser.java, ExtendedUser_**'Peer.java, BaseExtended'***User.java and BaseExtended*'**UserPeer.java. If you change the name to be "FOO_BAR", the classes will instead be called FooBar.java, FooBar'***Peer.java, BaseFoo*'**Bar.java and BaseFoo'**_BarPeer.java, so choose your name parameter such that it creates classes with names you like.

When these classes are created, the Base*.java classes are setup to inherit from the classes given in the baseClass and basePeer attributes. If you follow the example, these will be the adapter classes you defined following the howto. The Base* classes should not be edited, as they get generated from the "project-schema.xml" file. You should instead edit the "non-base" classes (e.g. ExtendedUser.java, ExtendedUser_*'Peer.java) as these only get generated if they are do not already exist.

In the Killer application, I used "KILLER_USER" as the name of my extension, and the resulting class hierarchy looked like this:

```
org.apache.turbine.om.security.TurbineUser*
 |
 +-com.divnull.killer.om.TurbineUser*Adapter
    |
    +-com.divnull.killer.om.BaseKillerUser*
       |
       +-com.divnull.killer.om.KillerUser*
```

Additional Maven Integration

The <root>/build/build.xml contains a number of targets to build your sample application. You can continue to use these if you like, but you may want to use Maven to provide those services instead. Presumably, that is why we set up a Maven directory structure in the first place. If this is the case, the easiest thing to do is to delete some of the targets from
<root>/build/build.xml. The following table describes the
<root>/build/build.xml target and lists the Maven goal that should replace it. (The Maven goals should just work for free. This is a good example of the whole point of Maven: it takes 100 lines of Ant code and turns it into zero lines of Ant code.)

```
TARGET                          MAVEN GOAL
assemble-webapp-in-container        -none- (This target assembles
                                files in the TDK directory for
                                some unfathomable reason)
deploy-war                      war:war
deploy-container                ear:ear (maybe? again, the purpose
                                of this target is unclear)
clean                           clean
```

Suggestions for the TDK Team

During the writing of this document, several issues came up that could go away if the TDK was changed a little. They are:

1) Have the sample app be built into a Maven structure. This should include a good sample project.xml file. It would also be interesting to see a maven.xml file that integrated the execution of the "init" tag at some point (after the jar process, perhaps).

2) Move the "build" directory out of WEB-INF in the sample app. Don't put it in WEB-INF's parent directory either. This director gets .war'd up for deployment, and the build scripts should not be deployed inside a .war.

3) Same goes for the src directory.