Unit Testing Database Access Methods

Testing database code in Java presents a difficult problem, the tests need to be repeatable, isolated, and automated. In addition the tests will require access to a "stable" database that the tester can control to ensure every time the tests run the test data is the same. DBUnit to the rescue.

There are a number of pre-test steps that are required to setup Unit testing of database access code:

- The Java Entity class must be compiled and ready to test
  - I will use the Customer Entity class during this discussion, please note I will need to run the Maven Install command to build the Entity jar file the DAO project will access to use the Entity class
- Test data will need to be created, DBUnit makes this easy.
  - I have created a DAOTestUtilities project in Eclipse, and there you will find BuildDBUnitData class which connects to the Database and builds an XML file holding the test data for our DBUnit tests this test data is named fullDB.xml, and goes into the src/test/resources folder in a sub-folder named data
- A Test database will need to create created during the tests, and discarded after the tests complete, the SQL create commands need to written
  - I have a Customer.sql file that contains the create commands to build the test database, this file is also in the src/test/resources folder in a sub-folder named data
- The DAO Interface and Implementation needs to written and compiled
  - I will use the CustomerDAO and CustomerDAOImpl classes in this discussion
- The JUnit and DBUnit dependencies need to be added to the Maven pom.xml file
  - I have a MavenBuildMaster project that contains these dependencies, this project is the parent of the DAO project.

- Although not strictly required, creating a “handler” to make the creation of the test database transparent is a good idea
  - I have created DBUnitJDBCUtility and DBUnitJPATestUtility classes in the DAOTestUtilities project to handle the test database creation, the JDBC test utility is for the Customer project, and the JPA test utility is for the World project
- Finally you will need to create new test data to test insert/update/delete methods, I copy and modify the original test data
  - I have provided the addCustomer.xml, the deleteCustomer.xml and the updateCustomer.xml files that hold the modified database records in the src/test/resources folder in a sub-folder named data

Planning DAO testing using DBUnit

My CustomerDAO/CustomerDAOImpl has this method:

Customer findCustomerById(UserCredentials credentials, int id)

I will need to write and run the following tests:
- testUnauthorizedFindCustomerById
- testFindCustomerByValidId
- testFindCustomerByInvalidId

I also have this method:

int addCustomer(UserCredentials credentials, Customer customer)

That will need the following tests:
- testUnauthorizedAddCustomer
- testNullAddCustomer
- testAddCustomer

For performance reasons, I divided the DAO tests into two classes:

- CustomerDAOSlowTests
  - The tests that change the database are here, these tests include:
    - testAddCustomer
    - testDeleteCustomer
    - testUpdateCustomer
- CustomerDAOFastTests
  - The tests that do not change the database are here, these include:
    - all the query tests
    - all the unauthorized tests
    - all the tests the throw an exception, like the null value passed tests

The biggest difference between the FastTests and the SlowTests is how I created the test database. In the FastTests the test database is created once in the @BeforeClass and used by all the tests. In the SlowTests the test database is recreated before each test in the @Before, this is to ensure each test is isolated and repeatable, meaning the record added during the testAddRecord is removed and the records changes in the testUpdateCustomer are restored.

Running DBUnit Tests

Steps:

1. Have the SQL to build the test database ready
2. Have the XML file to lod the test database ready
3. Have any additional XML files to hold the changed records to support add/update/delete tests ready
4. In the JUnit test @BeforeClass/@Before construct the DBUnitJDBCUtility/DBUnitJPAUtility object that will hold the test database, passing the SQL and XML files.
5. In the JUnit @AfterClass/@After methods call the shutdown method of the utility test database object
6. In each test:
   1. Construct a DAO Implementation object to test

2. Pass to the DAO the readOnly/readWrite connection as needed, getting that connection from the utility test database object
3. Create the Mock Credentials you need for testing
4. Call the method in the DAO to access the database

7. If the test involves a change to the test database, you will need to confirm the update happened. The test utility has methods to extract data from the test database, and to compare the expected values against an XML file.

```
try
{
    // Fetch database data after executing your code
    ITable actualTable = utility.getTableFromDatabase("CUSTOMER");
    ITable expectedTable = utility.getTableFromFile(CUSTOMER_ADD_FILE, "CUSTOMER");

   // Assert actual database table match expected table
   // This will check every row, and every column of the table
   Assertion.assertEquals(expectedTable, actualTable);
}
catch (Exception error)
{
     fail(error.getMessage());
}
```