Unit testing database code

Richard Dallaway <richard@dallaway.com>
May 2001

Introduction

The problem is this: you have a SQL database, some stored procedures, and a layer of code sitting between your application and the database. How can you put tests in place to make sure your code really is reading and writing the right data from the database?

These are my notes on how I've gone about unit testing database functionality. The examples are in Java, but I think the ideas are applicable to a variety of programming environments. I'm always looking for better solutions.

Why bother?

I'm guessing some, if not a lot, of database development goes like this: set up database, write code to access database, run code, do a SELECT to see if the records showed up in the database. They did? Good, then we're done.

The problem with **visual inspection** is this: you don't do it often, and you don't check everything every time. It's possible that when you make changes to a system, maybe months later, you break something and some data will go missing. As a coder you may not spend much time checking the data itself, so it may take a while for this mistake to surface. I've worked on a web project where a mandatory field on a registration form was not being inserted into a database for the best part of a year. Although marketing had protested that they *needed* this information, the problem wasn't spotted because the data was never ever looked at it (but don't get me started on *that*).

Automated tests — painless tests that run often and test lots — reduce the chances of your data is going missing. I find they make it easier for me to sleep at night. (Tests have other positive features: they're good examples of how to use code, they act as documentation, they make other people's code less scary when you need to change it, they reduce debugging time).

What kinds of tests are we talking about?

Consider a simple user database, with a email address and a flag indicating if mail to the address has bounced or not. Your database layer might consist of methods for `insert`, `update`, `delete`, and `find`.

The `insert` method would call a stored procedure to write the address and field to the database. With much simplification and omission the code might look like this:

```
public class UserDatabase
{
  ...
  public void insert(User user)
  {
    PreparedStatement ps = connection.prepareCall("{ call User_insert(?,?) }");
    ps.setString(1, user.getEmail());
    ps.setString(2, user.isBad());  // In real life, this would be a boolean.
    ps.executeUpdate();
    ps.close();
  }
  ...
}
```

The kind of testing code I'm thinking of would look something like this:

```
public class TestUserDatabase extends TestCase
{
  ...
  public void testInsert()
  {
    // Insert a test user:
    User user = new User("some@email.address");
    UserDatabase database = new UserDatabase();
    database.insert(user);

    // Make sure the data really got there:
    User db_user = database.find("some@email.address");
    assertTrue("Expected non-null result", db_user != null);
    assertEquals("Wrong email", "some@email.address", db_user.getEmail());
    assertEquals("Wrong bad flag", false, db_user.isBad());
  }
  ...
}
```

... only you'd have more tests, probably. (And take care with some tests, like tests on dates).

The `assertTrue` and `assertEquals` methods test that a condition is true, and if not the test fails in some way giving a diagnostic message. The idea is that the test is automatically run via a test framework, and a clear indication of success or failure is flagged. This is based on JUnit (see resources, below), a testing framework for Java. The framework is available for other languages, including C, C++, Perl, Python, .NET (all languages), PL/SQL, Eiffel, Delphi, VB... (see resources, below).

The next question becomes: we have tests, but how do we manage the testing data in the database so that it doesn't "mess up" live data?

Approaches that don't work

Before I start, I should point out that I expect you to have a development database. You wouldn't want to do anything I've noted in here on a production database.

The first approach I tried was to manually insert some testing data in a copy of the production database. These would be records with known values, such as "testuser01@test.testing". If you were testing some searching functionality, you'd know that there were, say, five users in the database "@test.testing".

For inserted test records, as in the example above, the test itself would have to maintain the state of the database. I.e., the test would have to be sure to clean up after itself, be careful not to deleted required records, so the database was in a good state once the test had finished.

This approach troubles me for the following reasons:

- You have to synchronize your test records with other developers — assuming they also have their own test database.

- Having "magic values" in the database (special email addresses, reserved id prefixes) doesn't seem right.
- You're stuck in the case where you can't reserve magic values to mark data as test data, such as a record made up of integers and all values are plausible.
- Your testing is limited to records within the values you have reserved — at best this means you have to select your magic values very carefully.
- If the data is time sensitive it becomes hard to maintain the test cases. An example: a database holds product offers which are valid only between specific dates.

Some fixes I've tried: add an "is_test" field to records, as a way to flag test records. This avoids the "magic values" problem. The down side is that your test code needs to operate only on records where the is_test field is set, whereas your production code needs to work with records where is_test is false. If you have differences at that level, you're not really testing the same code.

You need four databases

Some thoughts: A good test set is self sufficient and creates all the data it needs. Testing can be simplified if you can get the database in a known state before a test is run. One way to do this is to have a separate unit test database which is under the control of the test cases: the test cases clean out the database before starting any tests.

In code, you can do this by having a dbSetUp method which might look like this:

```
...
public void dbSetUp()
{
  // Put the database in a known state:
  // (stored procedures would probably be better here)
  helper.exec("DELETE FROM SomeSideTable");
  helper.exec("DELETE FROM User");

  // Insert some commonly-used test cases:
  ...
}
```

Any database test would call dbSetUp() before anything else, which would put the database in a known state (mostly empty). This gives you the following advantages:

- All test data is communicated to other developers in the code: there's no problems syncronizing external test data.
- No magic values.
- Simple, easy to understand approach.
- Deleting and inserting data for every test may seem like a big time over head, but as tests use relatively little data, I find this approach to be quick enough (especailly if you're running against a local test database).

The down-side is that you need more than one database — but remember, they can all run on one server if necessary. The way I'm testing now needs four databases (well, two at a pinch):

1. The **production database**. Live data. No testing on this database.
2. Your **local development database**, which is where most of the testing is carried out.

3. A **populated development database**, possibly shared by all developers so you can run your application and see it work with realistic amounts of data, rather than the hand full of records you have in your test database. You may not strictly need this, but it's reassuring to see your app work with lots of data (i.e., a copy of the production database's data).
4. A **deployment database**, or integration database, where the tests are run prior to deployment to make sure any local database changes have been applied. If you're working alone, you may be able to live without this one, but you'll have to be sure any database structure or stored procedure changes have been made to the production database before you go live with your code.

With multiple database you have to make sure you keep the structure of the databases in sync: if you change a table definition or a stored procedure on your test machine, you'll have to remember to make those changes on the live server. The deployment database should act as a reminder to make those changes. Also I find source control systems help here if the commit comments are emailed to all developers automatically. CVS (see resources, below) does this, and I expect others do too.

Test against the right database

In this environment you have to be sure you're connecting to the right database. Running the test set against a production database will delete all your data. This scares the hell out of me.

There are ways to protect against this. For example, it's not uncommon to have database connection settings stored in a initialization file and you can use this to state which database is the test database. You might use one initialization file for testing (pointing to a local database), and another for production work (pointing to a live database).

In Java, an initialization file might look like this:

```
myapp.db.url=jdbc:mysql://127.0.0.1/mydatabase
```

This is the connection string for connecting to a database. You can also add a second connection string to identify the test database:

```
myapp.db.url=jdbc:mysql://127.0.0.1/mydatabase
myapp.db.testurl=jdbc:mysql://127.0.0.1/my_test_database
```

In test code you can add a check to make sure you'll only run when you're connecting to a test database:

```
public void dbSetUp()
{
   String test_db = InitProperties.get("myapp.db.testurl");
   String db = InitProperties.get("myapp.db.url");

   if (test_db == null)
     abort("No test database configured");

   if (test_db.equals(db))
   {
      // All is well: the database we're connecting to is the
      // same as the database identified as "for testing"
   }
   else
   {
      abort("Will not run tests against a non-test database");
   }
}
```

Another trick: if you have a local testing database, the tests can check the IP or hostname you've been asked to run against. If it's not localhost / 127.0.0.1 there's a risk you're running against a live database.

Conclusions

In these notes I've tried to say:

- unit testing of database code is worth doing;
- you can do the testing without too much pain if you're willing to give your test suite a database to itself.

There are other ways to approach this problem. I'm not yet confident enough to trust myself with mock objects (see resources, below). As I understand mock objects, you simulate a layer of a system (in this case, the RDBMS), so that your mock database always returns just what you expect. I like the sounds of that: it encourages you to layer your testing, perhaps by having a SQL-level set of tests, and then having a Java-level set which work on mock ResultSet objects.

My concern is only that some actions can lead to changes in a two or more tables, and at this point the mock objects/simulation may become a pain to maintain and implement. And of course, I'll need to find a good way to test the SQL-level of the database: remember, I want to be sure data really is making it into the database properly.

Comments

I'm always looking for better ways to test database code. If you have any ideas, or any comments, please do email me (richard@dallaway.com) or use the discussion board linked from http://www.dallaway.com/acad/dbunit.html

A note on testing dates

If you're storing date information you probably want to make sure that you're storing the right dates. Be aware of some issues.

First, ask yourself who is creating the date? If it's your code, that's fine because you should be able to compare the date you created to the date you get back when you go looking into the database. If the database is creating the date, perhaps as a default column value, then you may have some problems. For example, are you sure the timezone for your code is the same as the timezone for the database? It's not unheard of to have databases running in GMT/UTC. Are you sure the clock on the machine you're running on will be set to the same time as the the clock on the database? If not, you're going to have some margin of error when comparing times.

If you run into these situations there are a few things you can do. If you know the timezone that will be used, cast all dates into a single timezone before testing. This will clear up any "offset by hours" problems. Also, allow a margin of error in comparing dates: say a few minutes, hours or months either side. Sounds lame, but at least it'll catch errors where you're picking up null or 1970-01-01 dates.

Resources

- Related threads from the JUnit mailing list: http://www.dallaway.com/acad/junit-yahoo-db.html
  The JUnit mailing list at Yahoo! (http://groups.yahoo.com/group/junit/) occasionally carries discussions on database unit testing. I've collected together some of the threads that I've found useful.
- Testing Resources for Extreme Programming: http://www.junit.org/
  The home of the Java JUnit software.
- Testing Framework software for download: http://www.xprogramming.com/software.htm
  Covering many programming languages.
- A Change in the Way We Program: http://www.extremeprogramming.org/change.html
  Unit testing is one part of "extreme programming". You can find lots about XP, starting with the Change in the Way We Program article, at extremeprogramming.org and also at http://xprogramming.com/
- Cactus: http://jakarta.apache.org/cactus/
  An Apache Jakarta project, extending JUnit to test a MVC architecture using servlets.
- Working backwards (PDF): http://www.objectmentor.com/publications/Working%20Backwards.pdf
  James Newkirk, Robert C. Martin. A draft of chapter 10 (pp. 91-106) from Extreme Programming in Practice. Discusses the use of mock objects (spoofing) to test certain database functionality.
- CVS: Concurrent Versions System: http://www.cvshome.org/
  Source control.
- Developing JDBC applications test-first: http://www.mockobjects.com/papers/jdbc_testfirst.html
  How to test JDBC-based applications without a database.