mpdb-test

Anthony Waters

February 6, 2008

Contents

0.1 Introduction

This part of the documentation explains the process of create test for the server side part of the application. The test take advantage of the JUnit framework [1] and also use DBUnit [2].

0.2 Why Perform Unit/Integration Test?

Well to start off lets define some keywords **unit testing** is a procedure used to validate that individual units of source code are working properly [3]. While **integration testing** is the phase of software testing in which individual software modules are combined and tested as a group [4]. In this project we primarily use integration testing and most of the test are centered around database interactions. The main reason to perform testing is so that end users do not encounter software bugs that could have been avoided by performing proper testing. It is also useful to verify whether a certain piece of code that works as intended i.e. a count query returns the right number of objects.

0.3 How Do We Perform Testing?

As mentioned before we use JUnit and DBUnit to aid us in testing these frameworks provide a lot of tools that make it easier to write and execute test. In the case of the server side test (mpdb-test-server) all it takes to execute the test is using the Eclipse IDE right click on the test file and go to Run As-¿JUnit Test. This will automatically run the test and give a status report of the success and failures. In the case of the client side test (mpdb-test-client) in order to execute the JUnit test you need to right click on the test.xml file and do Run As-¿Ant Build. The Ant script is necessary for client side testing because it is responsible for setting up the database with test data.

0.4 Writing Test

0.4.1 Server Side Testing

When doing server side testing all you need to do is write a class that extends from `DatabaseTestCase.java`. This is necessary because it initializes the database with test data for the database. An example of a test is located in the package `edu .rpi .metpetdb .server .dao` and within the class `SampleDaoTest.java`. In order to explain creating server side testing I will use the `SampleDaoTest.java` test as an example. When viewing the file the

[1] Unit testing framework http://www.junit.org/
[2] Initializes a test database http://www.dbunit.org/
[3] http://en.wikipedia.org/wiki/Unit_Testing
[4] http://en.wikipedia.org/wiki/Integration_testing

first thing you will notice is the constructor `public SampleDaoTest()` which is responsible for telling `DatabaseTestCase` which input file to use to get data from for the database. In this case we get data from the file `test-data/test-sample-data.xml`. The rest of the source file is just testing methods. To mark a method to be tested you need to add the `@Test` annotation above the method. There are plenty more annotations for JUnit that can be found at their home page. An example test is `testSampleById()`, what it basically does is load a sample by a certain id, in this case 1, and verifies that the loaded sample has the same id. This is how JUnit test usually work, you specify something you want to do, do the action, and then test that the action as successful. Testing that a test succeeds is done with using assertEquals, assertTrue and assertFalse [5] . While the method may be very short (2 lines) a lot more action is behind the helper method `super.byId` which is contained in `DatabaseTestCase.java`. There are also a lot more methods in `DatabaseTestCase.java` that are responsible for handling interactions with the database.

0.4.2 Client Side Testing

Client side testing works similar to server side testing there are a few major differences though. The first difference is that all of the test need to extend `GWTTestCase`. By extending this class a few methods need to be implemented `public String getModuleName()`, which in our case has to return `edu .rpi .metpetdb .MetPetDBApplication`. There is also another difference in the way the `GWTTestCase` handles `AsyncCallback` [6] . An example test method `testLoadSample()` shows how to do asynchonous calls within a test. Basically making calls to the server in client side code is how you do it in a test. There are two differences though, the first one is you need to call `delayTestFinish(x)`, where x is the length in miliseconds, to delay the test from finishing while the asynchonous calls complete. Then you need to call `finishTest` to mark that the test is done.

[5] There are more assert* methods that can be found at JUnit's homepage

[6] see http://code.google.com/webtoolkit/documentation/com.google.gwt.doc .DeveloperGuide.JUnitIntegration.html