

Unit Testing with TestNG and jmockit

TestNG is a testing framework for unit test development. jmockit is a framework for mock objects that provides mock object functionality using the `java.lang.instrument` package of jdk 1.5. Together, these frameworks can provide the tools to create very robust test cases without design limitations of other testing frameworks currently available. In part one of this two part tutorial, we will cover the creation of a test case and the implementation of the related class to be tested. We will reuse the same scenario as in my tutorial: Unit testing with JUnit and EasyMock. If you are new to unit testing in general, I suggest you check out the JUnit tutorial as well for it's section on unit testing in general. In part two, we will cover scenarios that favor the TestNG and jmockit frameworks including testing using mocks without injection and organizing tests into groups.

To begin, complete my Unit testing with JUnit and EasyMock tutorial. Then run TestNG's JUnit converter. That's it. Tutorial over. Just kidding (although TestNG does have a converter that you can run to convert existing JUnit tests to TestNG tests).

Setup

Although TestNG does provide a JUnit conversion tool, that is not the approach we will take here. So let's get started. I will be using Eclipse 3.3 Europa to do this tutorial. To begin, create a new java project and call it TestNGTutorial. Right click on your new project and select New --> Folder. Name it lib and click Finish. Usually you don't want to package your test code with your regular code, so let's make an additional source directory, test. To do that, right click on your new project and select Source Folder. Next we need to add TestNG to our build path. To make our life easier and since we are serious about unit testing ;) we will use the TestNG plugin for Eclipse. It includes a test runner and other functionality that will make things easier for test development. To install the plugin, in Eclipse, go to Help --> Software Updates --> Find and Install... From there, select "Search for new feature to install" and click "Next>". In the upper right hand corner, click on "New Remote Site". Enter "TestNG Plugin" as the Name and `http://beust.com/eclipse` as the URL and click OK. Make sure TestNG is the only thing with a check next to it and click Finish. In the new window that is displayed, check the box next to testng and click Next>. On the following screen, accept the license agreement and click Next>. Click Finish. Once the download is complete, Eclipse will ask you what to install. Click Install All. Once the install is complete, restart Eclipse. Once Eclipse is back, right click on your project and click Properties. Select Java Build Path from the left and click on the Libraries tab. On the right, click Add Variable and select TESTNG_HOME. Click the Extend... button to the right and select `/lib/testng-jdk15.jar` and click Ok. Click OK to exit the Properties window.

We now need to get jmockit. This is the mock framework we will be using for this tutorial. To download jmockit, go to [here](#) and download the JMockit 0.94c release. Once the download is complete, extract the jmockit.jar file from the archive and place it in the lib folder we created previously. To add the new jar to your classpath, right click on your project and select Properties. Go to Java Build Path and select the Libraries tab. From there, click on Add JARs.... Select the jmockit jar we just copied into the lib directory and click Ok. Click Ok to exit the properties window. You are now ready to code!

The scenario

In test driven design, we develop the unit test before the functionality. We write a test that verifies that the method should do X after our call. We prove that the test fails, we then create the component to make the test pass. In this case, we are going to create a service with a method that authenticates a user. Below is a class diagram of the scenario.

The interfaces

As stated before, we will be testing the same scenario as we did in the previous tutorial. To review, we will start our coding by defining two interfaces, LoginService and UserDao We will implement LoginService, however since in this tutorial UserDao will be mocked, we won't bother implementing it right now. For LoginService, we have a single method

that takes a String userName and String password and returns a boolean (true if the user was found, false if it was not). The interface looks like this:

```
/**
 * Provides authenticated related processing.
 */
public interface LoginService {

    /**
     * Handles a request to login. Passwords are stored as an MD5 Hash in
     * this system. The login service creates a hash based on the paramters
     * received and looks up the user. If a user with the same userName and
     * password hash are found, true is returned, else false is returned.
     *
     * @parameter userName
     * @parameter password
     * @return boolean
     */
    boolean login(String userName, String password);
}
```

The UserDao interface will look very similar to the LoginService. It will have a single method that takes a userName and hash. The hash is an MD5 hashed version of the password, provided by the above service.

```
/**
 * Provides database access for login related functions
 */
public interface UserDao {

    /**
     * Loads a User object for the record that
     * is returned with the same userName and password.
     *
     * @parameter userName
     * @parameter password
     * @return User
     */
    User loadByUsernameAndPassword(String userName, String password);
}
```

TestNG

Before we begin development, we will develop our test. Tests are structured by grouping methods that perform a test together in a test case. TestNG subscribes well to POJO development styles by not requiring you extend any class or implement any interface. The test lifecycle is directed via annotations (we will be covering standard JDK 1.5+ annotations in this tutorial, TestNG does support JDK 1.4 as well. It requires an additional JAR file and uses XDoclet style annotations). The annotations we will cover in this tutorial are:

- @Before*/@After*

The before and after series of annotations tell the test runner to execute the annotated method either before or after the specified test, group, method, suite or class.

- @Test

The @Test annotation indicates to the runner that a method is a test method.

The method we will be testing is the login method on the LoginService. Because this method has a dependency on the UserDao, we will have to create a mock for it. In this tutorial we will be using the jmockit mocking framework. Before we start writing any test code, let's cover some important points about the jmockit framework.

jmockit

Most mocking frameworks (JMock, EasyMock, etc) are based on the java.lang.Proxy object. They create a proxy object based on the interface you provide to be injected into the object to be tested. The developer tells the proxy what to expect and what to return when the expectation is met. There are two issues with this:

- It requires an interface. Don't get me wrong, programming to interfaces is a good thing, but not everything requires an interface. EasyMock does have an extension that handles classes without an interface, but it can be a pain to have two different EasyMock objects included in your test class to do basically the same thing.

- Dependency injection of some kind is required. Don't get me wrong, I love Spring and use it on my current project. But what if you don't want to have Spring manage everything. You cannot mock an object that is created in your method. The only way to use mock objects is if you control the instantiation of an object elsewhere (in a factory typically).

jmockit addresses these two issues using the redefinition ability. If you are unfamiliar with the new ability to redefine classes in JDK 1.5+, essentially what it does is it allows you to programatically tell the JVM to remap a class to a different class file. So when you create a mock with jmockit, you will be creating a new class that the JVM will replace the dependent class with.

The test case

So now that we know a tiny bit about both TestNG and jmockit, let's start to flush out our test case. To start, we'll create an empty TestNG testcase with the appropriate imports:

```
import org.testng.annotations.*;

public class LoginServiceTest {

}
```

Note that the class does not extend any class or implement any interface. This is because TestNG does not require it. We will dictate all of the lifecycle stages using annotations. As we noted before, the login method is dependent on the UserDao. Because of this, let's create a setupMocks method to be run before each test. The setupMocks method will create any required mocks we need to create and inject them accordingly.

```
import mockit.Expectations;

import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class LoginServiceTest extends Expectations {

    private LoginServiceImpl service;
    UserDao mockDao;

    @BeforeMethod
```

```

public void setupMocks() {
    service = new LoginServiceImpl();
    service.setUserDao( mockDao );
}
}

```

Let's go over what we have in the above class. We have the two required imports for this test case, `mockit.Mockit` (the main utility for `jmockit`) and `org.testng.annotations.*` to import the required TestNG annotations we will use. We have declared the two objects we care about: the service to be tested and the mock DAO to be injected. Note, `LoginService` is private but `UserDAO` is not. `jmockit` will create a mock object for the `mockDao` automatically for any non private fields.

Next we have our `setupMocks` method. This has TestNG's `@BeforeMethod` annotation. This tells TestNG to run this method once before each test. This will give us a fresh instance to test so that we don't have to be concerned with any clean-up that may be required. We then create the instance of our service to be tested and finally, inject the mock object into our implementation. If this looks alot like `EasyMock`, it should. The usage of `jmockit` and `EasyMock` are very similar when using interfaces. It's when you use classes that the usage differs.

The next step is to create our actual test method. In our test method, we want to test the following scenario:

Even with the very basic method we want to test above, there are still a number of different scenarios that require tests. We will start with the "rosy" scenario, passing in two values and getting a user object back. Below is the updated test with our new test method.

```

...
/**
 * This method will test the "rosy" scenario of passing a valid
 * username and password and retrieveing the user. Once the user
 * is returned to the service, the service will return true to
 * the caller.
 */
@Test
public void testRosyScenario() {
    User results = new User();
    String userName = "testUserName";
    String password = "testPassword";
    String passwordHash =
        "ÿö & I7ÿÿÿNi=";

    invokeReturning(
        mockDao.loadByUsernameAndPassword( userName,
                                           passwordHash ),
        results );
    endRecording();

    assert service.login( userName, password ) :
        "Expected true, but was false";
}
...

```

So let's go thru the code above. We start out with our annotation `@Test`. This tells TestNG that this is a test method. Once in the method, we create the expected result of our DAO call, `results`. In this case, our method will just check to see if an object was returned, so we don't need to populate our user object with anything, we just need an empty instance. Next we declare the values we will be passing into our service call. The password hash may catch you off guard. It's considered unsafe to store passwords as plain text so our service will generate an MD5 hash of the password and that value is the value that we will pass to our DAO.

Once declaring the various variables, the next call:

```
invokeReturning(
    mockDao.loadByUsernameAndPassword( userName, passwordHash ),
    results );
```

tells jmockit to expect the method `loadByUsernameAndPassword` to be called with the values previously declared in `userName` and `passwordHash`. When that happens, return the value `results`. The next line:

```
endRecording();
```

tells jmockit that we are done recording the expectations for this test case. jmockit automatically goes into replay mode at this point. Finally we perform the actual "test" by executing the method to be tested, `login` and asserting that it is true. If it is not, we present a message. TestNG considers a test passing if no exceptions are thrown. If an exception is thrown from a method that is declared a test, the test fails.

To run our test case, we have one more thing to configure. We need to tell the JVM that we will have a java agent running. To do that, right click on the `LoginServiceTest`, select `Run As --> Open Run Dialog`. On the left, select `TestNG`. At the top, click on the new icon. In the new area to the right, Enter `LoginServiceTest` in the `Class` box. Click the `Arguments` tab at the top. In the `VM Arguments` box, enter the following (where `<PATH_TO_LIB>` is the fully qualified path to the lib folder of your project):

```
-javaagent:<PATH_TO_LIB>/jmockit.jar
```

All that is left is to click `Run`. What? The test failed?! That's because we haven't implemented the method we are testing yet. The code for that is below.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
```

```
public class LoginServiceImpl implements LoginService {
    UserDAO userDao;
```

```
    public void setUserDao(UserDAO userDao) {
        this.userDao = userDao;
    }
```

```
    public boolean login(String userName, String password) {
```

```
boolean valid = false;
try {
    String passwordHash = null;
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    md5.update(password.getBytes());
    passwordHash = new String(md5.digest());

    User results =
        userDao.loadByUsernameAndPassword(userName, passwordHash);
    if(results != null) {
        valid = true;
    }
} catch (NoSuchAlgorithmException ignore) {}

return valid;
}
```

Conclusion

As you can see, TestNG provides a robust featureset that isn't foreign to someone who is used to JUnit styles of unit tests. In part two of this tutorial, we will refactor the LoginService to take better advantage of the jmockit framework and create additional tests that we can organize with TestNG. Until next time!