

Unit Testing with TestNG and jmockit Part 2

This tutorial is part two in a two part series on TestNG and jmockit. The previous tutorial (found [here](#)) covered the classic JUnit and EasyMock scenario, only with TestNG and jmockit. Although you can do that type of testing with these two technologies, that is not their strong point. In this tutorial, we will cover some more advanced features of TestNG and use jmockit's ability to "remap" a class in your JVM to handle a more robust suite of testing scenarios.

Setup

Since this is part two of the series, we will reuse the same Eclipse project for this tutorial that we did in part one (I go into detail on setting it up [here](#)). There is no further setup required.

The scenario

This scenario is the same as the previous tutorial. However, we will be implementing the solution slightly different. Instead of using a Spring like dependency injection of our UserDao, our service will be constructing it by itself. Just for reference, the class diagram for the scenario is below.

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 and implementing interfaces, LoginService and UserDao. 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);
}

```

Mocking without injection

Mockito is based on the concept of JVM class redefinition. In JDK 1.5, the `java.lang.instrument.Instrumentation` class was created. It allows you to "remap" a class definition within a JVM programmatically. An example would be if I defined `ClassA` and `ClassB`. I can tell the JVM "If an instance of `ClassA` is requested, give them `ClassB` instead". This is a very high level description. Please refer to the javadocs here for more detail.

Below is the implementation of the `LoginServiceImpl` we will be starting with. It is currently implemented to accept the `UserDAO` to be injected via some form of dependency injection (Spring, etc).

```

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;
    }
}

```

The test for the above method is below. These items together constitute our starting point.

```

import mockit.Expectations;

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

```

```

public class LoginServiceTest extends Expectations {

    private LoginServiceImpl service;
    private UserDao mockDao;

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

    /**
     * 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 =
            "ýö & 17ýýNi=";

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

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

```

To begin with our code changes, we first want to refactor our test to expect the "remapping" of our UserDaoImpl to our mock. To do this, let's start with what we don't need anymore. We don't need to extend Expectations anymore. Since we will be defining a mock manually, this isn't used. We can also remove the declaration of UserDao as a field and all references to it, including the creation of the mock and injection in the setupMocks method as well as the invokeReturning call and endRecording call in testRosyScenario. What you should be left with is below:

```

import mockit.Mockit;

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

public class LoginServiceTest {

    private LoginServiceImpl service;

    @BeforeMethod
    public void setupMocks() {
        service = new LoginServiceImpl();
    }

    /**
     * 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() {
    final String userName = "testUserName";

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

```

Now let's create our mock and it's expectation. To do that, we are going to use `mockit.Mockit.redefineMethods()` method. This static method takes two parameters, a class that you want to remap and the class you want to remap it to. In our case, we want to remap the `UserDaoImpl` class to a mock we will define inline (for ease of this tutorial). You can see the updated test method below:

```

...
/**
 * 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() {
    final String userName = "testUserName";

    Mockit.redefineMethods( UserDaoImpl.class, new Object() {
    } );

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

```

Now let's define the method that our service will call. In this case, we will assert that the parameters we expect were passed in and return an empty `User` object (since we only care if the `User` object was returned or not).

```

...
/**
 * 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() {
    final String userName = "testUserName";

    Mockit.redefineMethods( UserDaoImpl.class, new Object() {
        public User loadByUsernameAndPassword( String un, String password ) {
            assert un.equals( userName ) : "Username did not match";
            assert "b0¶ & 17€€³Ni=".equals( password ) :
                "Password hash did not match";
        }
    } );
}
}

```

```

        return new User();
    }
});

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

```

If you execute the test at this point, you should get a `NullPointerException`. This is because we haven't refactored the service to instantiate the DAO by itself (It's still looking for it to be injected). To refactor the service, we'll make a slight code change:

```

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

public class LoginServiceImpl implements LoginService {

    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());

            UserDao userDao = new UserDaoImpl();

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

        return valid;
    }
}

```

Above, you'll notice that we removed the instance reference to the `UserDAO`. We also added the instantiation of the `UserDAO` in our login method. Now if you run the test, it should pass.

Test grouping in TestNG

One of the more powerful features of TestNG is the ability to group test methods together and run only the ones tagged as a member of a specific group. For example, you could have a collection of tests you run before checking in, a different set of tests for your continuous integration environment and yet another set of tests to be run as only an integration suite. This is a very powerful tool that is very easy to implement. Let's start by creating another test method so we have two that we can differentiate thru grouping. In the second test method, we will test the scenario where the user is not found.

```

...
/**
 * This method will test the negative of the "rosy"

```

```

* scenario of passing a valid username and password and
* retrieving the user. Once the user is returned
* to the service, the service will return true to
* the caller.
*/
@Test
public void testNotFoundScenario() {
    final String userName = "notFoundUser";

    Mockito.redefineMethods( UserDaoImpl.class, new Object() {
        public User loadByUsernameAndPassword( String un, String password ) {
            assert un.equals( userName ) : "Username did not match";
            assert "p0q & 17€€³Ni=".equals( password ) :
                "Password hash did not match";
            return null;
        }
    });

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

```

If you run your test class with the above method added, it should pass. So for our example, we are going to group the positive scenarios together and the negative ones together and have a group that runs them all as well. To do this, we will add a groups parameter to each `@Test` annotation as well as the `@BeforeMethod` annotation we have on the `setupMocks` method. We will have one group named `positive` and one group named `negative`. We will have a third group named `all` that will run all tests. Below is the updated code:

```

...
@BeforeMethod(groups = {"positive", "all", "negative"})
public void setupMocks() {
    service = new LoginServiceImpl();
}

/**
* This method will test the "rosy" scenario of
* passing a valid username and password and
* retrieving the user. Once the user is returned
* to the service, the service will return true to
* the caller.
*/
@Test(groups = {"positive", "all"})
public void testRosyScenario() {
    final String userName = "testUserName";

    Mockito.redefineMethods( UserDaoImpl.class, new Object() {
        public User loadByUsernameAndPassword( String un, String password ) {
            assert un.equals( userName ) : "Username did not match";
            assert "p0q & 17€€³Ni=".equals( password ) :
                "Password hash did not match";
            return new User();
        }
    });

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

```

```

/**
 * This method will test the negative of the "rosy"
 * scenario of passing a valid username and password and
 * retrieving the user. Once the user is returned
 * to the service, the service will return true to
 * the caller.
 */
@Test(groups = {"negative", "all"})
public void testNotFoundScenario() {
    final String userName = "notFoundUser";

    Mockito.redefineMethods( UserDaoImpl.class, new Object() {
        public User loadByUsernameAndPassword( String un, String password ) {
            assert un.equals( userName ) : "Username did not match";
            assert "b0f & 17€€³Ni=".equals( password ) :
                "Password hash did not match";
            return null;
        }
    });

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

```

Now we can run our test four different ways, we can run it with the group all, the group negative, the group positive or the default (which is all of the tests). Let's start by running only the positive ones. To run just the group positive, right click on your test class and select Run As --> Open Run Dialog.... On the Test tab, you have the option of selecting a class, group or suite. In this case, we are going to select the group radio button and click the Browse button to the right of that row. You'll notice that Eclipse offers you three choices: negative, positive and all, the three groups we have defined in our test. Select the positive option and click OK. Click Run to execute the test. You'll notice that it succeeds and that only one test was executed, our positive one. If you do the above steps again, you can select either the negative or all groups to be executed.

You'll notice that we had to include the groups parameter on the @BeforeMethod annotation. This is very powerful in that you can specify setup methods for each group. Groups are also not limited to a single class file. Groups can span multiple test classes. By doing this, you can define groups for an entire project (like check in tests, continuous integration tests, integration tests, etc) that can be run based on the situation. Groups also have a hierarchy that you can extend when you implement groups at the class level (instead of at the method level like this tutorial).

Conclusion

The features of TestNG and jmockit provide a very robust feature set that allow the testing of just about every scenario possible. They allow you the flexibility to design your system in the best way possible instead of making sacrifices for testability. What we have covered in this tutorial and the previous one are just the tip of the iceberg. I hope it will give you the incentive to take a look at all TestNG and jmockit have to offer.