Paul Grenyer February 2010

Integration Testing A Java Enterprise Application Data Access Layer

Enterprise Application Development in Java

Integration Testing

Finding the best design approach is only part of the solution to writing an Enterprise Application. As with all software, the application must be tested. Unit tests are the first line of defence, but they only the units of the application individually. You also need to test how the units interact with each other and other parts of the application, such as the database. These sorts of tests are called integration tests.

Soon after I discovered unit testing I accidental discovered integration testing. I was writing a system that interacted with a database at just about every stage. So each of my tests would start by creating and populating the database. Then the tests would be run and afterwards the database would be torn down. Creating and destroying the database for each test ensured that it was always in a known state. It also meant that every test took a very long time to run. When I only had a handful of tests this was not too much of a problem. Once I had in-excess of a hundred tests it was a big problem! A general rule is that tests should take close to zero time to run, or people stop running them.

I started to think about how I could improve the performance of my integration tests. So I created a data access layer as is described in my article Data Access Layer Design for Java Enterprise Applications [DAL]. This enabled my integration tests, which were only masquerading as unit tests, to become real unit tests as they could use a mocked-out data access layer, rather than a real database, there by reducing their run time to almost zero. However, I still needed to write true integration tests to test the data access layer itself. These tests still needed a real database.

Most of the time taken up in setting up and tearing down a database is in its creation and deletion. The actual insertion, update, reading and deletion of data is relativity quick. Most databases support transactions. Transactions prevent a unit of work from being committed to the database until it is completed successfully. If an error occurs part way through the unit of work the transaction can be rolled back and none of the work carried out is committed. If the unit of work is successful it is committed to the database in its entirety once complete.

This can also be applied to testing. At the beginning of an integration test a transaction is started. As necessary, data is inserted into the database and the tests run, which themselves may insert or remove data. Once the test is complete the transaction is rolled back and the database is returned to its original state before the tests. Even if the test fails, the transaction is rolled back and the database returned to its original state.

Once I discovered this I moved the creation of the database structure (the database itself, tables, views, stored procedures, reference data, etc) to my build script and started running my tests in transactions. This greatly improved the performance of my integration tests.

As I am sure you can imagine, there is a lot of boiler place code necessary to be able to run tests in a transaction. Luckily the Spring framework [Spring] provides a helper class that significantly reduces the amount of boiler plate that has to be written. In this article I will look at how to use `AbstractTransactionalJUnit4SpringContextTests` to write integration tests for a hibernate based data access layer. I will use the data access layer developed in my Data Access Layer Design for Java Enterprise Applications article which can be downloaded [DALDnld] from my website.

Setting Up The Project

Once you have downloaded the source code from Data Access Layer Design for Java Enterprise Applications, create a fresh eclipse project and copy the `uk.co.marauder.dataaccesslayer.model` package into it. In addition to the JARs supplied with the project you will need to add:

```
org.springframework.core-3.0.0.RELEASE.jar
org.springframework.beans-3.0.0.RELEASE.jar
org.springframework.web-3.0.0.RELEASE.jar
org.springframework.context-3.0.0.RELEASE.jar
org.springframework.asm-3.0.0.RELEASE.jar
org.springframework.expression-3.0.0.RELEASE
org.springframework.orm-3.0.0.RELEASE.jar
org.springframework.transaction-3.0.0.RELEASE.jar
org.springframework.jdbc-3.0.0.RELEASE.jar
```

which are all included in the Spring 3 release, which can be downloaded from the Spring website. Also remove `spring-2.5.6.jar` as Spring 3 has come out since I wrote the original article and it is always sensible to use the latest version of libraries where possible.

Creating The Test Fixture

If you review the `UserManager` class you will see that it has two very simple operations: `load` and `save`. The integration test only needs two tests methods:

```
public class UserManagerTest
{

	@Test
	public void testSave()
	{
		// Add save tests
	}

	@Test
	public void testLoad()
	{
		// Add load tests
	}
}
```

As its name suggests, `AbstractTransactionalJUnit4SpringContextTests` relies on JUnit 4 [Junit], so make sure:

```
junit-4.8.1.jar
```

is added to the project. To take advantage of `AbstractTransactionalJUnit4SpringContextTests`'s transaction support you need to inherit from it. However, as you will soon see it needs to be customised to work easily with Hibernate and those customisations will be useful in all Hibernate integration tests, so it makes sense to create a Hibernate specific version:

```
public class AbstractTransactionalJUnit4SpringContextHibernateTests
      extends AbstractTransactionalJUnit4SpringContextTests
{
      // Add hibernate specific customisations
}
```

and inherit from that:

```
public class UserManagerTest
      extends AbstractTransactionalJUnit4SpringContextHibernateTests
{
      //...
}
```

If you try to run the tests now they will fail with an error stating that `ApplicationContext` cannot be found. `AbstractTransactionalJUnit4SpringContextTests` looks for a Spring application context on start-up. I explain the Spring Application Context in my Integrating Spring with the Google Web Toolkit - Part I [ISGWT] article. For our purposes it is best to think of the Spring Application Context as a factory for objects required by `AbstractTransactionalJUnit4SpringContextTests`. This instructions for how to create these objects are written in XML and stored in a file. The file can be named anything, but something starting with `applicationContext` is the convention. Let's call our file `applicationContext-dataAccess.xml`:

```
<beans      xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd" >
</beans>
```

The file can go anywhere in classpath. In a Spring based Enterprise Application the chances are you will have an application context file already which can be used. If not and you are only using Spring for integration testing Hibernate, put it in with your test files. To enable `AbstractTransactionalJUnit4SpringContextTests` to find `applicationContext-dataAccess.xml` add the following annotation to `UserManagerTest`:

```
@ContextConfiguration(locations={"applicationContext-dataAccess.xml"})
public class UserManagerTest
```

```
        extends AbstractTransactionalJUnit4SpringContextHibernateTests
{
        //...
}
```

The locations attribute specifies the classpath to one or more xml files from which the application context will be created. If you try and run the tests again you will find that `AbstractTransactionalJUnit4SpringContextTests` is still not happy. It requires a java.sqlDataSource to be injected from the application context. In Spring Application Context in my Integrating Spring with the Google Web Toolkit – Part I I describe how to declare a connection pool data source and a Hibernate session factory:

```
<beans        xmlns="http://www.springframework.org/schema/beans"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-3.0.xsd" >

        <bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
              <property name="location"
                     value="classpath:dataaccesslayertest.properties"/>
        </bean>

        <bean id="dataSource"
              class="org.apache.commons.dbcp.BasicDataSource"
              destroy-method="close">
              <property   name="driverClassName"
                          value ="${database.driverClassName}"/>
              <property name="url" value ="${database.url}"/>
              <property name="username" value="${database.username}"/>
              <property name="password" value="${database.password}"/>
              <property name="initialSize" value ="${database.initialSize}"/>
              <property name="maxActive" value ="${database.maxActive}"/>
        </bean>

        <bean id="sessionFactory"
              class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
              <property name="dataSource" ref="dataSource" />
              <property name="mappingResources">
              <list>
                     <value>
                           uk/co/marauder/dataaccesslayertest/model/User.hbm.xml
                     </value>
              </list>
              </property>
              <property name="hibernateProperties">
                     <props>
                           <prop key="hibernate.dialect">
                                  ${hibernate.dialect}
                           </prop>
                           <prop key="hibernate.show_sql">
                                  ${hibernate.show_sql}
                           </prop>
                     </props>
              </property>
        </bean>
</beans>
```

You will also need to add the following JARs:

```
commons-dbcp-1.2.2.jar
commons-pool-1.5.4.jar
```

Running the tests again, however, still does not work. `AbstractTransactionalJUnit4SpringContextTests` requires one more Spring bean from the application context:

```
<beans      xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd" >

      ...

      <bean id="transactionManager"
         class="org.springframework.orm.hibernate3.HibernateTransactionManager">
            <property name="sessionFactory" ref="sessionFactory" />
      </bean>
</beans>
```

In order to create, commit and rollback trasnactions `AbstractTransactionalJUnit4SpringContextTests` needs a transaction manager for the data source being used. As we are using Hibernate we need a `HibernateTransactionManager`, which takes a reference to the Hibernate session factory. Now if you run the tests you will get the green bar of success, but of course the tests are not testing anything yet.

MySQL Transactions

I have tested transactions on both Microsoft SQL Server and MySQL. They work out of the box for SQL Server, but not for MySQL. In order to get transactions to work with MySQL, tables need to be of type `INNODB`. For example:

```
CREATE TABLE Users
(
      username varchar(50) NOT NULL UNIQUE,
      password varchar(50) ,
      enabled bit NOT NULL,

    CONSTRAINT Pk_Users PRIMARY KEY CLUSTERED (username ASC)
) TYPE = INNODB;


CREATE TABLE Authorities
(
      username varchar(50) NOT NULL,
      authority varchar(50) NOT NULL,

      CONSTRAINT Pk_Authorities PRIMARY KEY CLUSTERED (username,authority ASC),
      CONSTRAINT Fk_Authorities_User FOREIGN KEY (username) REFERENCES Users
(username)
```

```
) TYPE = INNODB;
```

This is because transactions have a performance overhead which MySQL tries to avoid.

Something you will probably notice when you run the tests is that you get the following error message at the console:

```
log4j:WARN No appenders could be found for logger
(org.springframework.test.context.junit4.SpringJUnit4ClassRunner).
log4j:WARN Please initialize the log4j system properly.
```

This is because Spring uses log4j and it has been unable to find a log4j properties file. This is easily fixed. Simply drop one into the root of the classpath. A suitable example can be found in appendix A.

Writing Integration Tests

Let's just have a quick recap of what we have so far. We have a single test fixture that when instantiated creates a Spring application context from a specified `applicationContext-dataAccess.xml` file and is injected with a data source and a transaction manager which are used, in theory at this stage, to run each of its test in a transaction. Now we need some test code.

If you review the `UserManager` class you will see that it takes a single constructor parameter of type `DMRegistry`. There is only one implementation of this interface, `HibernateTemplateDMRegistry` and it takes a single constructor parameter of type `HibernateTemplate`. A `HibernateTemplate` instance is something that is likely to be used a lot in Hibernate integration tests, so it is sensible to have it served up by `AbstractTransactionalJUnit4SpringContextHibernateTests`. `HibernateTemplate` takes a single constructor parameter of type `SessionFactory` and although the is a `sessionFactory` bean in the application context it is not currently being injected. This is how you inject it:

```
public class AbstractTransactionalJUnit4SpringContextHibernateTests
      extends AbstractTransactionalJUnit4SpringContextTests
{
      private SessionFactory sessionFactory;

      @Required
      @Resource
      public void setSessionFactory(SessionFactory sessionFactory)
      {
            this.sessionFactory = sessionFactory;
      }

      protected SessionFactory getSessionFactory()
      {
            return sessionFactory;
      }
}
```

The `@Required` annotation tells the application context that it must call the `setFactory` method straight after instantiating any bean of type `AbstractTransactionalJUnit4SpringContextHibernateTests`. The `@Resource` annotation tells the application context to look for a Spring bean called `sessionFactory` of type `SessionFactory` and use that as the parameter when `setSessionFactory` is called. Although the argument name matches the name of the `SessionFactory` bean, the name of the bean that is looked for is actually derived from the method name after removing the '`set`' part and lower casing the first letter. The argument could be named anything. The `getSessionFactory` method simple provides access to the `SessionFactory`. The `setSessionFactory` method is also a convenient place to create a `HibernateTemplate`:

```
public class AbstractTransactionalJUnit4SpringContextHibernateTests
	extends AbstractTransactionalJUnit4SpringContextTests
{
	private SessionFactory sessionFactory;
	private HibernateTemplate hibernateTemplate;

	@Required
	@Resource
	public void setSessionFactory(SessionFactory sessionFactory)
	{
		this.sessionFactory = sessionFactory;
		this.hibernateTemplate = new HibernateTemplate(sessionFactory);
	}

	// ...

	protected HibernateTemplate getHibernateTemplate()
	{
		return hibernateTemplate;
	}
}
```

The `getHibernateTemplate` method gives access to the `HibernateTemplate`.

Now we have everything we need to create a `UserManager` instance for testing:

```
@ContextConfiguration(locations={"applicationContext-dataAccess.xml"})
public class UserManagerTest
	extends AbstractTransactionalJUnit4SpringContextHibernateTests
{
	private UserManager userMgr;

	@Before
	public void setUp() throws Exception
	{
		final DMRegistry registry
			= new HibernateTemplateDMRegistry(getHibernateTemplate());

		userMgr = new UserManager(registry);
	}

	// ...
```

```
}
```

The most straight forward way to test `UserManager.save` is to create and save some users and then check that they are in the database. Let's look at this in two sections. First saving users:

```
@Test
public void testSave()
{
	final String auth1 = "ROLE_LOGIN";
	final String auth2 = "ROLE_ADMIN";
	final String auth3 = "ROLE_USER";
	final String auth4 = "ROLE_LOG";

	final User user1 = new User("Paul","password1",true);
	user1.addAuth(auth1);
	user1.addAuth(auth2);
	user1.addAuth(auth3);

	final User user2 = new User("James","password2",false);
	user2.addAuth(auth1);
	user2.addAuth(auth3);

	final User user3 = new User("John","password3",true);
	user3.addAuth(auth1);
	user3.addAuth(auth2);
	user3.addAuth(auth4);

	userMgr.save(user1);
	userMgr.save(user2);
	userMgr.save(user3);

	// Interrogate database
}
```

The `testSave` method creates three different users, Paul, James and John with varying properties and roles and then saves each. If you execute the code at this stage you will get the green bar of success. If you check the database manually you will not see new users, which is expected. However, you cannot prove that saving the users worked as even if it did, the transaction was rolled back and the database returned to its original state. Therefore you need to query the database as part of the test:

```
@Test
public void testSave()
{
	// ..

	flushAndClear();

	final List<String> result =
		getJdbcTemplate().query(
			"select u.username,u.password,u.enabled,a.authority from Users
			 u, Authorities a where u.username = a.username order by
			 u.username, a.authority", new RowMapper<String>()
	{
		@Override
```

```
            public String mapRow(ResultSet rs, int arg1) throws SQLException
            {
                  return      rs.getString(1) + " " +
                              rs.getString(2) + " " +
                              rs.getString(3) + " " +
                              rs.getString(4);
            }
      });

      final String[] expectedResults =
      {
            "James password2 0 ROLE_LOGIN",
            "James password2 0 ROLE_USER",
            "John password3 1 ROLE_ADMIN",
            "John password3 1 ROLE_LOG",
            "John password3 1 ROLE_LOGIN",
            "Paul password1 1 ROLE_ADMIN",
            "Paul password1 1 ROLE_LOGIN",
            "Paul password1 1 ROLE_USER"
      };

      assertArrayEquals(expectedResults, result.toArray(new String[]{}));
}
```

The first thing you should notice are two methods that have not been implemented yet, `flushAndClear` and `getJdbcTemplate`. Spring 3's `JdbcTemplate` encapsulates a lot of the boiler plate code that is required to make calls to a database using JDBC. It takes a DataSource as a constructor parameter and should also be created in `AbstractTransactionalJUnit4SpringContextHibernateTests`:

```
public class AbstractTransactionalJUnit4SpringContextHibernateTests
      extends AbstractTransactionalJUnit4SpringContextTests
{
      // ...

      private DataSource dataSource;
      private JdbcTemplate jdbcTemplate;

      @Required
      @Resource
      @Override
      public void setDataSource(DataSource dataSource)
      {
            this.dataSource = dataSource;
            this.jdbcTemplate = new JdbcTemplate(dataSource);
            super.setDataSource(dataSource);
      }

      protected DataSource getDataSource()
      {
            return dataSource;
      }

      protected JdbcTemplate getJdbcTemplate()
      {
            return jdbcTemplate;
      }
```

```
    // ...
}
```

AbstractTransactionalJUnit4SpringContextTests, the super class of AbstractTransactionalJUnit4SpringContextHibernateTests already has a getDataSource method, but no method for accessing the DataSource. Therefore to get the DataSource the setDataSource method must be overriden, the DataSource stored and then passed to the super class's setDataSource method. Unfortunatley the @Required and @Resource annotations are not inherited and must be respecified. The getDataSource method is also a convenient place to create the JdbcTemplate and getJdbcTemplate and getDataSource methods provide access to it and the DataSource respectively.

Hibernate is very clever and will avoid writing to the database until the very last moment. Under normal circumstances in test cases like this Hibernate would not write to the database until the transaction is successfully committed, but of course the test case transaction is never committed. This means that when JdbcTemplate is used to query the database it will not see the saved users. However, Hibernate can be coaxed into writing to the database early. This is where the flushAndClear method comes in:

```
public class AbstractTransactionalJUnit4SpringContextHibernateTests
      extends AbstractTransactionalJUnit4SpringContextTests
{
      // ...

      protected void flushAndClear()
      {
            final Session session
                  = SessionFactoryUtils.getSession(
                              getHibernateTemplate().getSessionFactory(),
                              false);
          session.flush();
          session.clear();
      }
}
```

Hibernate's SessionFactoryUtils.getSession method can be used to get the Hibernate Session bound to the current thread from the SessionFactory passed to the HibernateTemplate. The Session's flush method forces Hibernate to write to the database and the clear method completely clears the session and evicts all loaded instances and cancels all pending saves, updates and deletes, just to be on the safe side.

Test code should be as simple as possible. All we need the testSave method to test is that each user was saved with the right user name, password, enabled statues and roles. The following SQL query will return all of this information:

```
SELECT u.username,u.password,u.enabled,a.authority
FROM Users u, Authorities a
WHERE u.username = a.username
```

```
ORDER BY u.username, a.authority
```

The result set is ordered to make comparing it to the expected result easier. You could write two queries. One to list all the user names, passwords and enabled status and another to list the authorities for each user, but the code required call these two queries and interpret the results would be unnecessarily verbose and complicated. The `JdbcTemplate.query` method is used to turn the results of the query into an array of strings:

```
final List<String> result =
      getJdbcTemplate().query("...", new RowMapper<String>()
      {
            @Override
            public String mapRow(ResultSet rs, int arg1) throws SQLException
            {
                  return      rs.getString(1) + " " +
                              rs.getString(2) + " " +
                              rs.getString(3) + " " +
                              rs.getString(4);
            }
      });
```

The `RowMapper.mapRow` method is called for each row in the result set. The user name, password, enabled status and role are pulled out (remember that the first JDBC index is 1, not 0) and appended together into a single string and returned. `JdbcTemplate` adds each string to a list which is passed back by the `query` method.

The expected results are specified in a static string array and `assertArrayEquals` used to compare them to the actual results:

```
final String[] expectedResults =
{
      "James password2 0 ROLE_LOGIN",
      "James password2 0 ROLE_USER",
      "John password3 1 ROLE_ADMIN",
      "John password3 1 ROLE_LOG",
      "John password3 1 ROLE_LOGIN",
      "Paul password1 1 ROLE_ADMIN",
      "Paul password1 1 ROLE_LOGIN",
      "Paul password1 1 ROLE_USER"
};

assertArrayEquals(expectedResults, result.toArray(new String[]{}));
```

You can demonstrate that the test is working as expected by commenting out `flushAndClear` or modifying `expectedResults` and seeing the test fail.

Testing `UserManager.load` is quite different. Ideally the database needs to already be populated with some user details that can be loaded and verified. This data could be inserted as reference data by the build script, but would effect the results of `testSave` and as we all know each test should be independent and not effect the other tests. There is an opportunity here to take advantage of the transaction and the

AbstractTransactionalJUnit4SpringContextTests.executeSqlScript method and insert the required data only for the life of the test and the transaction:

```
@Test
public void testLoad()
{
	executeSqlScript(
	"classpath:/uk/co/marauder/dataaccesslayer/model/dao/UserManagerTest.sql",
	false);

	// ..
}
```

The `executeSqlScript` method loads a file, in this case from the classpath, and executes it within the transaction. It is best to keep SQL scripts that are used by tests in the package with the tests themselves and name them appropriately. `UserManagerTest.sql` looks like this:

```
-- Users --
INSERT INTO Users (username,password,enabled) VALUES('paul','password1',true);
INSERT INTO Users (username,password,enabled) VALUES('james','password2',false);
INSERT INTO Users (username,password,enabled) VALUES('john','password3',true);

-- Roles --
INSERT INTO Authorities (username,authority) VALUES ('paul','ROLE_ADMIN');
INSERT INTO Authorities (username,authority) VALUES ('paul','ROLE_LOGIN');
INSERT INTO Authorities (username,authority) VALUES ('paul','ROLE_USER');

INSERT INTO Authorities (username,authority) VALUES ('james','ROLE_LOGIN');
INSERT INTO Authorities (username,authority) VALUES ('james','ROLE_USER');

INSERT INTO Authorities (username,authority) VALUES ('john','ROLE_LOGIN');
INSERT INTO Authorities (username,authority) VALUES ('john','ROLE_ADMIN');
INSERT INTO Authorities (username,authority) VALUES ('john','ROLE_LOG')
```

It inserts the same data that is save in the `testSave` test. The code to test the `load` method is then very straight forward:

```
User user = userMgr.load("Paul");
assertEquals("Paul",user.getUsername());
assertEquals("password1",user.getPassword());
assertTrue(user.isEnabled());
assertEquals(	"['ROLE_ADMIN','ROLE_LOGIN','ROLE_USER']",
		user.getAuths().toString());

user = userMgr.load("James");
assertEquals("James",user.getUsername());
assertEquals("password2",user.getPassword());
assertFalse(user.isEnabled());
assertEquals("['ROLE_LOGIN','ROLE_USER']",user.getAuths().toString());

user = userMgr.load("John");
assertEquals("John",user.getUsername());
assertEquals("password3",user.getPassword());
assertTrue(user.isEnabled());
assertEquals(	"['ROLE_LOGIN','ROLE_USER','ROLE_LOG']",
```

```
                    user.getAuths().toString());

assertNull(userMgr.load("Charlotte"));
```

Each user is loaded in turn and its user name, password, enabled status and list of roles checked. Finally a check is made to load a user that does not exist.

Both tests, `testSave` and `testLoad`, will now run successfully, neither interferes with the other and both leave the database in its original state.

Finally

In this article I have demonstrated the need for integration tests when developing an Enterprise Application and how tests that use a database can be written to run as quickly as possible. This is made possible with the use of database transactions and the amount of code needed is reduced by using a helper class from the Spring Framework.

As always I have only scratched the surface of what can be achieved, but there is enough here provide a firm foundation.

Appendix A: log4j.properties

```
# Set root category priority to WARN and its only appender to CONSOLE.
log4j.rootCategory=WARN, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{dd-MMM-yyyy HH:mm:ss.SSS} %-
5p [%-8.8X{Request.User.Name}] %-30c{1} - %m%n

# Print only messages of level WARN or above in the package org.hibernate
log4j.logger.org.hibernate=WARN

# log the the hibernate-generated SQL used to create, update and delete entities
log4j.logger.org.hibernate.persister.entity=WARN

# Print only messages of level WARN or above in the package org.springframework
log4j.logger.org.springframework=WARN
```

Appendix B: applicationContext-dataAccess.xml

```
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=    "http://www.springframework.org/schema/beans
                                http://www.springframework.org/schema/beans/spring-beans-3.0.xsd" >

        <bean   id="propertyConfigurer"
                class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
                <property name="location" value="classpath:dataaccesslayertest.properties"/>
        </bean>

        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
                <property name="driverClassName" value ="${database.driverClassName}"/>
                <property name="url" value ="${database.url}"/>
```

```
            <property name="username" value="${database.username}"/>
            <property name="password" value="${database.password}"/>
            <property name="initialSize" value ="${database.initialSize}"/>
            <property name="maxActive" value ="${database.maxActive}"/>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
            <property name="dataSource" ref="dataSource" />
            <property name="mappingResources">
                    <list>
                            <value>uk/co/marauder/dataaccesslayertest/model/User.hbm.xml</value>
                    </list>
            </property>
            <property name="hibernateProperties">
                    <props>
                            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
                            <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
                    </props>
            </property>
    </bean>

    <bean  id="transactionManager"
            class="org.springframework.orm.hibernate3.HibernateTransactionManager">
            <property name="sessionFactory" ref="sessionFactory" />
    </bean>

</beans>
```

References

[DAL] Data Access Layer Design for Java Enterprise Applications: http://www.paulgrenyer.net/Data_Access_Layer_Design_for_Java_Enterprise_Applications.pdf

[Spring] The Spring Framework: http://www.springsource.org/

[DALDnld] Download Data Access Layer project: http://paulgrenyer.net/dnld/dataaccesslayer.zip

[Junit] JUnit: http://www.junit.org/

[ISGWT] Integrating Spring with the Google Web Toolkit - Part I: http://www.paulgrenyer.net/Integrating_Spring_with_the_Google_Web_Toolkit_-_Part_I.pdf

Acknowledgements

Thank you to Richard Walker at Validus-IVC Ltd. for demonstrating how to use `AbstractTransactionalJUnit4SpringContextTests`.