P7- Pruebas de integración

Pruebas de integración con base de datos

El objetivo de esta práctica es automatizar pruebas de integración con una base de datos, implementando los drivers correspondientes. Para controlar el estado de la base de datos durante los tests utilizaremos la librería DbUnit.

Bitbucket

El trabajo de esta sesión también debes subirlo a Bitbucket. Todo el trabajo de esta práctica, tanto los proyectos maven que vamos a crear, como cualquier otro documento con vuestras notas de trabajo, deberán estar en la carpeta **P7** de vuestro repositorio**.**

Para esta práctica proporcionamos código adicional en la carpeta PlantillasP7

Base de datos MySQL

En la máquina virtual tenéis instalado un servidor de bases de datos MySQL, al que podéis acceder con el usuario "root" y password "ppss" a través del siguiente comando, desde el terminal:

```
> mysql –u root –p
```

Dicho servidor se pone en marcha automáticamente cuando arrancamos la máquina virtual

Ejecución de scripts sql con Maven: plugin sql-maven-plugin

Hemos visto en clase que podemos utilizar un plugin de maven (*sql-maven-plugin*) para ejecutar scripts sql sobre una base de datos mysql.

El plugin *sql-maven-plugin* puede tener configuradas diferentes ejecuciones (varias secciones <execution>, cada una con un valor de <id> diferente). Para ejecutarlas podemos hacerlo de diferentes formas:

- Si hay varias <execution> asociadas a la misma fase y ejecutamos dicha fase maven, se ejecutarán todas ellas
- Si queremos ejecutar solamente una de las <executions>, podemos hacerlo con mvn sql:execute@execution-id, siendo execution-id el identificador de la etiqueta <id> de la <execution> correspondiente
- Si ejecutamos simplemente mvn sql:execute, sin indicar ninguna ejecución en concreto, por defecto se ejecutará aquella que tenga identificada con <id>default-cli</id>. Si no hay ninguna <execution> con ese nombre, no se ejecutará nada.

Ejemplo: Dado el siguiente fragmento de código con las <executions> del plugin,

- La ejecución *mvn pre-integration-test* ejecutará los scripts script1.sql y script1.sql
- La ejecución *mvn sql:execute@create-customer-table2* solamente ejecutará el script2.sql
- La ejecución *mvn sql:execute* no ejecutará nada

```
...
<executions>
  <execution>
     <id>create-customer-table</id>
     <phase>pre-integration-test</phase>
     <goals>
        <goal>execute</goal>
     </goals>
     <configuration>
        <srcFiles>
            <srcFile>src/test/resources/sql/script1.sql</srcFile>
        </srcFiles>
    </configuration>
  </execution>
  <execution>
    <id>create-customer-table2</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>execute</goal>
    </goals>
    <configuration>
      <srcFiles>
        <srcFile>src/test/resources/sql/script2.sql</srcFile>
      </srcFiles>
    </configuration>
  </execution>
</executions>
...
```

Ejercicios

1. Crea un nuevo proyecto Maven "**dbunitexample**" (recuerda que debes crearlo en el directorio **P7**), con valores para ArtifactId y GroupId **dbunitexample** y **ppss**, respectivamente. En el campo de texto Package que nos muestra Netbeans pondremos como valor **ppss**.

   En el directorio Plantillas-P7/ejercicio1 encontrarás:

   - la implementación de las clases sobre las que realizaremos las pruebas: *ppss.Customer* y *ppss.CustomerFactory*,
   - la implementación de dos tests de integración (*CustomerFactoryIT.java*),
   - ficheros xml con los datos iniciales de la BD (*customer-init.xml*), y resultado esperado de uno de los tests (*customer-expected.xml*),
   - fichero con el script sql para restaurar el esquema y las tablas de la base de datos (*create-table-customer.xml*)

   Se pide:

   A) Organiza el código proporcionado en el proyecto Maven que has creado según hemos visto en clase, y modifica el pom.xml convenientemente para poder utilizar DbUnit y el plugin *sql-maven-plugin.* Añade, además, las siguientes dependencias, se trata de librerías de *logging.* Solamente las incluimos para que no nos aparezcan advertencias (*warnings)* al construir el proyecto con Maven.

```
<!-- Logging -->
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.13</version>
</dependency>
```

B) En el código de pruebas, explica qué representan las variables `_customerFactory` y `databaseTester` indicando para qué y dónde se utilizan.

C) En el código de pruebas, explica la diferencia entre un `dataset` y una `table`. Identifica dónde y para qué se utilizan en el código proporcionado.

D) Utiliza Maven para ejecutar los tests, para ello debes asegurarte previamente de que tanto la cadena de conexión, como el login y el password sean correctos, tanto en el pom.xml, como en el código de pruebas. Justifica claramente por qué debes utilizar dicho comando. Observa la consola que muestra la salida de la construcción del proyecto, anota todas las goals que se se ejecutan y verifica que el orden de ejecución de las mismas es el que hemos visto en clase. Anota los artefactos que se han generado.

E) Una vez ejecutados los tests, cambia los datos de prueba del método test_insert() y realiza las modificaciones necesarias adicionales para que los tests sigan en "verde". Puedes probar a realizar cambios en los datos de entrada del otro test para familiarizarte con el código.

2. En el directorio Plantillas-P7/ejercicio2 tienes dos proyectos Maven denominados *matriculacion-comun* y *matriculacion-dao*.

   Se pide:

   A) El proyecto ***matriculacion-comun*** contiene la definición de ciertas clases, de las que depende el proyecto *matriculacion-dao*. ¿Puedes indicar qué tienen en común todas las clases definidas en dicho proyecto? ¿Es necesario hacer pruebas unitarias sobre dicho proyecto? Justifica por qué.

   B) Ejecuta la fase de Maven ***package*** sobre el proyecto *matriculacion-comun*, y observa que en el directorio target se crea un fichero (artefacto) con extensión ***.jar.*** A continuación ejecuta la fase ***install***. En esta fase se copia el artefacto anterior en el repositorio local. Comprueba que efectivamente se almacena donde corresponde. Puedes ver el contenido del archivo empaquetado, desde el terminal, con el comando jar -tvf *archivo.jar*. También puedes ver su contenido desde la ventana *Files* de Netbeans. En clase hemos explicado por qué la fase *package* se realiza después de ejecutar los tests unitarios y antes de ejecutar los tests de integración. Viendo el contenido del archivo generado en la fase *package* te tiene que quedar claro lo que se explicó en clase.

   C) El proyecto ***matriculacion-dao*** contiene la implementación de los objetos que acceden a la base de datos, y sobre los que realizaremos las pruebas. Identifica la implementación del patrón DAO en el código de directorio src/main/java, que hemos explicado en clase.

D) Enumera y justifica todas las dependencias (etiqueta <dependency>) del fichero pom.xml del proyecto matriculacion-dao.

E) En la clase ***FuenteDatosJDBC*** se configura la conexión con la BD. Tendrás que asegurarte de que la cadena de conexión con la base de datos sea la correcta, tal y como hemos hecho en el ejercicio anterior. Para comprobar que hemos configurado correctamente dicha cadena de conexión proporcionamos la clase ***ConsultaBD***, que puedes ejecutar (con botón derecho sobre dicho fichero: opción *Run File.* Si esta opción no te aparece activada es porque Netbeans no ha reconocido dicho fichero java como fichero ejecutable. Si ejecutas alguna fase Maven, Netbeans marcará dicho fichero como ejecutable y ya te aparecerá activada la opción *Run File* del menú contextual de dicho fichero). Para que ConsultaBD funcione correctamente tendrás que ejecutar previamente los *scripts* matriculacion.sql, y datos.sql, en ese orden. El primero crea el esquema de la base de datos del proyecto y el segundo añade datos en la tabla *alumnos* de dicha base de datos.

F) Configura el pom para poder utilizar la librería dbunit tal y como se ha visto en clase de teoría.

G) Como ya hemos visto, DbUnit permite definir los datos de las tablas (tanto los iniciales como los que esperamos tener como salida) en ficheros XML. Vamos a exportar el estado actual de la tabla "alumnos" al formato XML de DbUnit para obtener un ejemplo del formato a utilizar. Para ello ejecuta la clase ***DatabaseExport***,(cuyo código tienes en el directorio Plantillas-P7/ejercicio2/DatabaseExport.java), asegurándote antes de que tanto la url de conexión con la BD, como el login y el password son correctos. Puedes comprobar que se han generado dos ficheros: *matriculación.dtd* y *dataset.xml* (en el directorio src/test/resources). El primero de ellos contiene la gramática de los ficheros XML para nuestro esquema de base de datos. El segundo contiene toda la información de nuestra base de datos, que es el que necesitamos para las pruebas

H) A continuación crea los ficheros *tabla2.xml*, *tabla3.xml*, y *tabla4.xml*, conteniendo los "datasets" necesarios para implementar los casos de prueba de la tabla 1 (los datasets se definen en las tablas 2, 3 y 4). Los ficheros xml se crearán en la carpeta src/test/resources, con la opción New→Other→XML→XML Document (desde el directorio /src/test/resources), indicando el nombre del fichero, y marcando como tipo de documento *"Well Formed Document"*. Asegúrate de referenciar el dtd en la cabecera de los documentos xml de las tablas:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dataset SYSTEM 'matriculacion.dtd'>

<dataset>
</dataset>
```

I) Implementa con *DbUnit* los casos de prueba de la tabla 1 en una clase `IAlumnoDAOIT`. Cada caso de prueba debe llevar el nombre indicado en la columna ID de la tabla 1. Ejecuta los *tests*. Evidentemente, para hacer las pruebas, tendremos que utilizar un objeto que implemente la interfaz IAlumnoDAO. Así, por ejemplo para cada test Ax de la tabla 1, utilizaremos una sentencia del tipo:

```
new FactoriaDAO().getAlumnoDAO().addAlumno(alumno);
```

Para especificar la fecha, debes utilizar la clase Calendar. Por ejemplo, para indicar la fecha de nacimiento “1985-02-22 00:00:00.0” de un objeto alumno, puede hacerse de la siguiente forma:

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.HOUR, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.MILLISECOND, 0);
cal.set(Calendar.YEAR, 1985);
cal.set(Calendar.MONTH, 1); //Nota: en la clase Calendar, el primer mes es 0
cal.set(Calendar.DATE, 22);
alumno.setFechaNacimiento(cal.getTime());
```

Tabla 1. Casos de prueba para IAlumnoDAO

ID	Método a probar	Entrada	Salida Esperada
testA1	void addAlumno (AlumnoTO p)	p.nif = "33333333C" p.nombre = "Elena Aguirre Juarez" p.fechaNac = 1985-02-22 00:00:00.0	Tabla 3
testA2	void addAlumno (AlumnoTO p)	p.nif = "11111111A" p.nombre = "Alfonso Ramirez Ruiz" p.fechaNac = 1982-02-22 00:00:00.0	DAOException
testA3	void addAlumno (AlumnoTO p)	p.nif = "44444444D" p.nombre = null p.fechaNac = 1982-02-22 00:00:00.0	DAOException
testA4	void addAlumno (AlumnoTO p)	p = null	DAOException
testA5	void addAlumno (AlumnoTO p)	p.nif = null p.nombre = "Pedro Garcia Lopez" p.fechaNac = 1982-02-22 00:00:00.0	DAOException
testB1	void delAlumno (String nif)	nif = "11111111A"	Tabla 4
testB2	void delAlumno (String nif)	nif = "33333333C"	DAOException

Tabla 2. Base de datos de entrada. Contenido tabla ALUMNOS

NIF	Nombre	Dirección	E-mail	
11111111A	Alfonso Ramirez Ruiz	Rambla, 22	alfonso@ppss.ua.es	1982-02-22 00:00:00.0
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22 00:00:00.0

Tabla 3. Base de datos esperada como salida en testA1

NIF	Nombre	Dirección	E-mail	
11111111A	Alfonso Ramirez Ruiz	Rambla, 22	alfonso@ppss.ua.es	1982-02-22 00:00:00.0
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22 00:00:00.0
33333333C	Elena Aguirre Juarez			1985-02-22 00:00:00.0

Tabla 4. Base de datos esperada como salida en testB1

NIF	Nombre	Dirección	E-mail	
22222222B	Laura Martinez Perez	Maisonnave, 5	laura@ppss.ua.es	1980-02-22 00:00:00.0