# Database Testing

## Outside DB Testing

### ADO.NET “External Data Source” and “Nuke and Pave”

Observamos que las pruebas de filtros realizan sus asertos esperando que se encuentren determinada información en la BD.

Podríamos asumir que la información correcta ya se va a encontrar en la bd al momento de realizar la prueba, pero esto trae 2 problemas:

* es muy muy probable que en ese momento la información no se encuentre en la bd o no sea la correcta como para ejecutar la prueba de manera repetitiva al menos que cada vez que estés ejecutando la prueba revises la BD e ingreses los datos manualmente, lo que haría que la prueba no se para nada repetible.
* si en caso esta información se encontrase en la BD, el test es poco legible ya que no se puede determinar a simple vista cuál es el contexto sobre el cuál se está creando la prueba.

Por estos motivos las propias pruebas deben ser capaces de inicializar sus datos. En esta oportunidad utilizaremos DBUnit como una herramienta q nos va a permitir cargar datos desde archivos XML.

* Agregamos la referencia a NDBUnit.SqlClient
* De acuerdo a los nombres de los archivos podemos observar que estamos creando un datasource por test fixture. Entonces como estos archivo serán utilizados por todo el tests fixture, vamos a configurarlos una sola vez en todo el test fixture.

private static SqlDbUnitTest database;  
  
[ClassInitialize]  
public static void FixtureSetup(TestContext context)  
{  
    database = new SqlDbUnitTest(ConfigurationManager.ConnectionStrings["DB"].ConnectionString);  
    database.ReadXmlSchema(@"EmployeeADODatabase.xsd");  
    database.ReadXml(@"EmployeeADOTestData.xml");  
}

* DbUnit necesita 2 archivos para su funcionamiento:
  + Un archivo XML con información de la tabla(s) de BD a las cuales vamos a insertar datos. No es necesario que estos archivos contengan toda la información de las tablas y columnas, solo lo necesario para poder insertar los datos.

Este archivo no es otra cosa que el famoso dataset que se utiliza para realizar operaciones en la BD en entorno conectado, solo que ahora se necesita únicamente para metadata de la BD. La forma más rápida de crear este archivo es crear un nuevo dataset.

* + El segundo archivo, es un xml con los datos q vamos a insertar en la BD.
* Ahora necesitamos indicar a dbunit que necesitamos insertar la información contenida en la los archivos, pero esta información la necesitamos insertar antes que se realiza cada prueba, ya que la prueba anterior ha podido alterar o eliminar la información que hemos insertado.

[TestInitialize]  
public void Setup()  
{  
    this.employeeADO = new EmployeeADO();  
    database.PerformDbOperation(DbOperationFlag.Insert);  
}

* Ahora que ya lo tenemos configurado, vamos a ejecutar únicamente la primera prueba, vemos que la prueba falla xq no se están encontrando los archivo, seguramente ya hemos observado que cada vez que se ejecuta una prueba con MSTest se crea una carpeta TestsResults que es donde se copian las entradas y las salidas de las pruebas. Por lo tanto necesitamos decirle a MStests que copie esos archivos a esa carpeta.
  + Copy Always en las propiedades del archivo.
  + Agregar los atributos DeploymentItem

    [DeploymentItem("EmployeeADODatabase.xsd")]  
    [DeploymentItem("EmployeeADOTestData.xml")]

* Abrimos el profiler para observar cuales son los queries generados.
* Primero ejecutamos únicamente la primera prueba y vemos que pasa sin problemas.
* Luego ejecutamos todos los tests y observamos que varios fallan, si vemos cuales son los datos de la BD, vamos a observar que tenemos más datos de los esperados. El problema es que necesitamos restablecer el estado de la BD para que la siguiente prueba se pueda ejecutar de manera consistente.

En esta oportunidad para restablecer el estado de la BD vamos a utilizar la técnica de Nuke and Pave, que es una de las técnicas más fáciles de implementar ya que elimina absolutamente todos los datos de las tablas involucradas sin importar si estos no son manejados por el tests.

Una delas características de DBUnit es que permite aplicar esta estratégia fácilmente. Para esto solo tenemos que utilizar el flag DbOperationFlag.CleanInsert, que hará que se realice un delete de toda la tabla antes de insertar los datos, observamos esto en el profiler.

* Aún el Delete no pasa, el problema es que está intentando eliminar un registro con un ID fijo, pero debido a que nuestras columnas son identity, este campo siempre es autoincrementando. Para evitar estos problemas podemos Utilizar el DbOperationFlag.CleanInsertIdentity que desabilitará esas columnas y permitirá insertar un valor fijo en la columna Id que podamos referenciar desde el test.

  <Employee>   
    <Id>1</Id>  
    <LastName>Pacheco</LastName>  
    <HireDate>2010-12-30</HireDate>  
  </Employee>

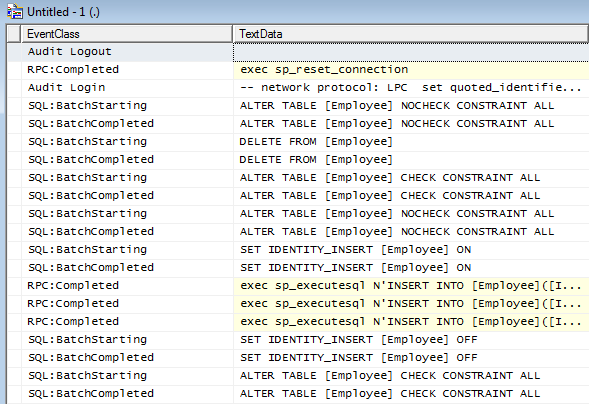
* Luego de finalizadas todas las pruebas y que estas pasan correctamente, podemos observar que aún permanecen algunos registros en la BD, ya que estamos restableciendo el estado antes de comenzar la siguiente prueba y no al terminar la anterior. Esto no necesariamente es malo ya que estamos asegurando que cada prueba se asegura de tener la BD en el estado adecuado.

En esta oportunidad vamos a eliminarlos para poder comenzar el siguiente ejercicio desde 0.

* Para eliminarlos podemos escribir una rutina que eliminé todos os registros y que únicamente se ejecute al finalizar todos los tests.

[ClassCleanup]  
public static void CleanUpDatabae()  
{  
    database.PerformDbOperation(DbOperationFlag.DeleteAll);  
}

* Examinemos el profiler.



### ADO.NET “External Data Source” and “Nuke and Pave”

### Recomendaciones Entity Framework

Ahora vamos a ver el último enfoque “Pruebas autosuficientes” para inicializar el estado de la BD y vamos a utilizarlo junto con “Transacion-Rollback”.

Si recordamos el enfoque de pruebas autosuficientes hace que cada prueba se encargue de inicializar sus datos y únicamente aquellos datos q necesita.

[TestMethod]  
public void Find\_WithLastNameFilter\_ReturnTheEmployeesWithTheExactLastName()  
{  
    Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee1);  
    Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
    LoadData(employee2);  
  
    String lastName = "Pacheco";  
  
    List<Employee> employees = this.employeeADO.Find(lastName, null, null);  
  
    Assert.AreEqual(1, employees.Count);  
}

private void LoadData(Employee entity)  
{  
    this.employeeADO.Create(entity);  
}

Ejecutamos únicamente esta prueba pero eliminamos los datos de la BD que quedaron del ejercicio anterior.

Ejecutamos la misma prueba nuevamente y observamos que falla. El problema es que los datos del ejercicio anterior se mantiene.

La estrategia Transaction-Rollback que como dijimos consiste en encerrar el test en una transacción y hacer rollback de esta transacción al finalizar el test de tal manera que cualquier cambio que haya realizado el test va a ser revertido por el rollback.

private TransactionScope transactionScope;   
  
[TestInitialize]  
public void Setup()  
{  
    this.employeeADO = new EmployeeADO();  
    transactionScope = new TransactionScope();  
}  
  
[TestCleanup]  
public void TearDown()  
{  
    transactionScope.Dispose();  
}

Al no realizar ningún commmit, el dispose de la transacción realizará un rollback automático. Cabe destacar que el transactionScope no necesariamente involucra una transacción distribuida, sino este se mantiene lightweigh mientras no se abran conexiones diferentes o conexiones embebidas dentro del ámbito de la transacción.

Completamos el siguiente test. Un consejo, es utilizar mismo set de datos en todas las pruebas de tal manera que luego podamos refactorizar esa inserción de datos y encapsularla en un único método.

[TestMethod]  
public void Find\_WithHireDateFilters\_ReturnTheEmployeesBetweenHireDates()  
{  
    Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee1);  
    Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
    LoadData(employee2);  
    Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28));  
    LoadData(employee3);  
  
    DateTime startHireDate = new DateTime(2010, 12, 29);  
    DateTime endHireDate = new DateTime(2010, 12, 30);  
  
    List<Employee> employees = this.employeeADO.Find(null, startHireDate, endHireDate);  
  
    Assert.AreEqual(2, employees.Count);  
}

Miremos el Insert y el Delete. Debido a que en esta oportunidad el mismo créate es usado para la inserción de la data de prueba, debemos evaluar la necesidad de hacerle una prueba directa, esto depende mucho del método de inserción que estemos utilizando, que no necesariamente es el mismo.

En el caso del delete insertamos un registro de prueba.

[TestMethod]  
public void Delete\_TheEmployeeIsDeleted()  
{  
    Employee employee = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee);  
  
    this.employeeADO.Delete(employeeLoaded.Id);  
  
    Employee employeeDeleted = this.employeeADO.Get(employeeToDelete.Id);  
    Assert.IsNull(employeeDeleted);  
}

Refactorizamos la inserción de datos. Esto tiene la ventaja de mantenimiento pero la desventaja a nivel de velocidad de la prueba.

[TestInitialize]  
public void Setup()  
{  
    this.employeeADO = new EmployeeADO();  
    transactionScope = new TransactionScope();  
  
    Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee1);  
    Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
    LoadData(employee2);  
    Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28));  
    LoadData(employee3);  
  
    employeeToDelete = employee1;  
}

### Examinando Entity Framework

En esta oportunidad vamos a realizar nuestras pruebas a métodos de acceso implementado con Entity Framework.

*¿Quienes no conocen nada de EF Code First, o nisiquiera han hecho una aplicación de ejemplo? Explicar brevemente EF en caso sea necesario(modelo,mappings,tablas).*

Examinemos los métodos de acceso a datos.

* Ahora existen 3 clases: AppDbContext,Employee, EmployeeEF
* AppDbContext es la clase más importante para el funcionamiento y la configuración de EF. *(Explicar un poco de EF en caso sea necesario)*
* *Employee* que es nuestra entidad.
* *EmployeeEF* que es nuestra clase de acceso a datos. *Examinamos que hace esta clase.*
* *EmployeeEFTests* contiene las mismas pruebas que vimos anteriormente.

Examinemos primero los tests, hablemos sobre el insert, , usualmente no se prueba el créate xq sería prácticamente probar el comportamiento del framework, pero si se puede probar direcamente el add para verificar los mappings. De la misma manera el delete se podría usar para verificar que ciertas cascadas están funcionando. En el caso del Get si realmente no tiene sentido.

Ejecutamos la primera prueba. Observaremos que los datos no se insertan.

Cualquier alteración de información que realicemos a través de un ORM, primero se realiza en memoria dentro del ámbito del ORM y necesitamos indicarle explícitamente al ORM que todos esos cambios se transformen en sentencias SQL. Por lo tanto cuando nosotros realicemos una llamada al Delete o Create no significa que se han producido los cambios en la BD.

[TestInitialize]  
public void Setup()  
{  
    this.context = new AppDbContext();  
    this.employeeEF = new EmployeeEF(context);  
    transactionScope = new TransactionScope();  
  
    Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee1);  
    Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
    LoadData(employee2);  
    Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28));  
    LoadData(employee3);  
    FlushChanges();  
  
    employeeLoaded = employee1;  
}  
  
  
[TestMethod]  
public void Create\_TheEmployeeIsPersisted()  
{  
    Employee employee = new Employee("Luis", "Carranza", new DateTime(2010, 12, 15));  
  
    employeeEF.Create(employee);  
    FlushChanges();  
  
    Employee employeePersisted = employeeEF.Get(employee.Id);  
    Assert.IsNotNull(employeePersisted);  
}  
  
[TestMethod]  
public void Delete\_TheEmployeeIsDeleted()  
{  
    employeeEF.Delete(employeeLoaded.Id);  
    FlushChanges();  
  
    Employee employeeDeleted = employeeEF.Get(1);  
    Assert.IsNull(employeeDeleted);  
}

### Entity Framework: “External DataSource” and “Transaction”

Vamos a seguir realizando Nuke and Pave pero ya no vamos a utilizar NBUnit sino la característica innata que tienen todos los ORMs como entity framework de poder generar automáticamente la BD. Si recordamos el Nuke and Pave elimina es borrar todo y volver a comenzar, en la anterior oportunidad eliminábamos todos los datos y los volvíamos a insertar, pero ahora vamos a ver como funciona el Nuke and Pave a otro nivel, eliminando también las tablas, prácticamente vamos a destruir toda la BD y vamos a volverla a crear.

¿Cómo funciona esto? Apenas abrimos una conexión con EF, este verifica si la BD de nuestra cadena de conexión existe, sino existe, crea toda la BD incluyendo sus tablas, pero además nos ofrece la posibilidad de poder insertar datos, estos datos se deben encontrar en otra clase y en forma de objetos.

Incluimos la clase EmployeeEFTestData dentro del proyecto.

public class EmployeeEFInitializer : DropCreateDatabaseAlways<AppDbContext>  
{  
    protected override void Seed(AppDbContext context)  
    {  
        Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
        Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
        Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28));  
  
        context.Employees.Add(employee1);  
        context.Employees.Add(employee2);  
        context.Employees.Add(employee3);  
    }  
}

EF tiene el concepto de initializers que son clases que le indican a EF como es que se va a recrear toda la BD. Para el caso de las pruebas el initializers más adecuado es el DropCreateAlways, ya que nosotros queremos generar toda la BD siempre que ejecutemos el test.

En la anterior oportunidad creábamos y destruíamos todos los datos en cada test, ahora lo que vamos a hacer es crear toda la BD pero únicamente al inicio de los tests y vamos a utilizar la estrategia de *Transaction-Rollback* para mantener la consistencia entre test y test.

[ClassInitialize]  
public static void InitializeDatabase(TestContext context)  
{  
    Database.SetInitializer(new EmployeeEFInitializer());  
    using (var dbcontext = new AppDbContext())  
    {  
        dbcontext.Database.Initialize(force: true);  
    }  
}

Vamos a ejecutar únicamente la primera prueba, como esta prueba no altera la información de la BD, no nos vamos a preocupar por el momento por como vamos a mantener la consistencia entre los tests.

Ejecutamos todas las pruebas de consulta y abrimos el profiler. Observamos que se está abriendo una transacción y ningún commit.

Ahora vamos a descomentar las otras pruebas, lo primero que tenemos que hacer para poder ejecutar estas pruebas es ver como vamos a mantener la BD en su estado conocido, para esto la estrategia Transaction-Rollback que como dijimos consiste en encerrar el test en una transacción y hacer rollback de esta transacción al finalizar el test de tal manera que cualquier cambio que haya realizado el test va a ser revertido por el rollback.

private TransactionScope transactionScope;

[TestInitialize]  
public void Setup()  
{  
    this.context = new AppDbContext();  
    this.employeeEF = new EmployeeEF(context);  
    transactionScope = new TransactionScope();  
}  
  
[TestCleanup]  
public void TearDown()  
{  
    transactionScope.Dispose();  
}

Al no realizar ningún commmit, el dispose de la transacción realizará un rollback automático. Cabe destacar que el transactionScope no necesariamente involucra una transacción distribuida, sino este se mantiene lightweigh mientras no se abran conexiones diferentes o conexiones embebidas dentro del ámbito de la transacción.

Ahora necesitamos resolver un siguiente problema, cualquier alteración de información que realicemos a través de un ORM, primero se realiza en memoria dentro del ámbito del ORM y necesitamos indicarle explícitamente al ORM que todos esos cambios se transformen en sentencias SQL. Por lo tanto cuando nosotros realicemos una llamada al Delete o Create no significa que se han producido los cambios en la BD.

public void Create\_TheEmployeeIsPersisted()  
{  
    Employee employee = new Employee("Luis", "Carranza", new DateTime(2010, 12, 15));  
  
    employeeEF.Create(employee);  
    FlushChanges();  
  
    Employee employeePersisted = employeeEF.Get(employee.Id);  
    Assert.IsNotNull(employeePersisted);  
}  
  
[TestMethod]  
public void Delete\_TheEmployeeIsDeleted()  
{  
    employeeEF.Delete(1);  
    FlushChanges();  
  
    Employee employeeDeleted = employeeEF.Get(1);  
    Assert.IsNull(employeeDeleted);  
}  
  
private void FlushChanges()  
{  
    context.SaveChanges();  
}

Seguramente nos habremos dado cuenta de la gran cantidad de tiempo que toman los enfoques Nuke and Pave, especialmente este que reconstruyen absolutamente toda la BD. Una característica de EF que puede ayudar a aplicar este enfoque reduciendo un poco el tiempo de ejecución de la prueba, es que podemos crear nuestros propios initializers. El initializers que estamos utilizando por defecto elimina y crea toda la BD, pero podemos crear un initializers que lo único que haga alterar los datos de las tablas sin eliminar estas tablas.

public class EmployeeEFInitializer : IDatabaseInitializer<AppDbContext>  
{  
    public void InitializeDatabase(AppDbContext context)  
    {  
        DeleteData<Employee>(context);  
        ResetIdentitySeed(context);  
        LoadData(context);  
    }  
  
    private void LoadData(AppDbContext context)  
    {  
        Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30)) { Id = 1 };  
        Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29)) { Id = 2 };  
        Employee employee3 = new Employee("Luis", "Tovar", new DateTime(2010, 12, 28)) { Id = 3 };  
  
        context.Employees.Add(employee1);  
        context.Employees.Add(employee2);  
        context.Employees.Add(employee3);  
  
        context.SaveChanges();  
    }  
  
    private void DeleteData<T>(AppDbContext context) where T : class  
    {  
        var tableName = TableName<T>(context);  
        context.Database.ExecuteSqlCommand("Delete from " + tableName);  
    }  
  
    private static void ResetIdentitySeed(AppDbContext context)  
    {  
        var tableName = TableName<Employee>(context);  
        context.Database.ExecuteSqlCommand("DBCC CHECKIDENT ('" + tableName + "', RESEED, 0)");  
    }  
  
    private static string TableName<T>(AppDbContext context) where T : class  
    {  
        return typeof(T).Name;  
        // Sí se utiliza el pluralize convention  
        /\* var objectContext = ((IObjectContextAdapter)context).ObjectContext;  
           return objectContext.CreateObjectSet<T>().EntitySet.Name;  
        \*/  
    }  
}

Dense cuenta que podemos utilizar un enfoque similar utilizando DBUnit y Transactions.

### Entity Framework: “Self-Contained Tests” and “Transactions.”

Ahora vamos a ver el último enfoque “Pruebas autosuficientes” para inicializar el estado de la BD y vamos a utilizarlo junto con “Transacion-Rollback”.

Si recordamos el enfoque de pruebas autosuficientes hace que cada prueba se encargue de inicializar sus datos y únicamente aquellos datos q necesita.

Nos dirigimos a la primera prueba y dentro de esta creamos los datos que necesita, agregamos estos datos al contexto y luego hacemos flush a los datos.

public void Find\_WithLastNameFilter\_ReturnTheEmployeesWithTheExactLastName()  
{  
    Employee employee1 = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee1);  
    Employee employee2 = new Employee("Luis", "Quispe", new DateTime(2010, 12, 29));  
    LoadData(employee2);  
    FlushChanges();  
  
    string lastName = "Pacheco";  
  
    IList<Employee> employees = employeeEF.Find(lastName, null, null);  
  
    Assert.AreEqual(1, employees.Count);  
}

private void LoadData(Employee entity)  
{  
    context.Employees.Add(entity);  
}

Ejecutamos únicamente esta prueba pero eliminamos los datos de la BD que quedaron del ejercicio anterior.

Ya no completaremos el resto de las pruebas de los filtros ya que son similares. La prueba del créate queda de la misma manera ya que no necesita datos previos para funcionar pero el delete si necesita por lo menos un dato en la BD.

[TestMethod]  
public void Delete\_TheEmployeeIsDeleted()  
{  
    Employee employee = new Employee("Luis", "Pacheco", new DateTime(2010, 12, 30));  
    LoadData(employee);  
    FlushChanges();  
  
    employeeEF.Delete(employee.Id);  
    FlushChanges();  
  
    Employee employeeDeleted = employeeEF.Get(1);  
    Assert.IsNull(employeeDeleted);  
}

Como se darán cuenta, este es el enfoque que menos tiempo toma y es el recomendado.

## Inside DB Testing

Podemos observar que en la BD tenemos 2 procedures con una lógica muy similar a la que vimos en los ejercicios anteriores. El objetivo de este ejercicio es realizar pruebas para ambos de estos procedures.

tSQLt se instala a nivel de BD por eso necesitamos ejecutar los scripts que vienen en el directorio que descargado desde internet.

* Debido a que usa cierta funcionalidad el CLR por eso necesitamos habilitarlo. Ejecutar el script SetClrEnabled.sql
* Ejecutar el script tSQLt.class.sql. este creará números objetos dentro del esquema tsqlt(procedures, tablas, ensamblados) dentro de la BD.

Ahora que tenemos los scripts instalados. Vamos a comenzar creando un Test Fixture. De manera similar a nuestras pruebas anteriores vamos a utilizar un TestFixture por Store Procedure.

Creamos el TestFixture.

EXEC tSQLt.NewTestClass 'EmployeeFind';

GO

Creamos el Primer Test.

Debemos seguir la siguiente convención:

* El nombre completo del test o procedure esta compuesto por [Esquema].[Nombre Procedure]: el Esquema debe tener el nombre del test fixture al cuál pertenece, el Nombre del Procedure debe comenzar con la palabra “Test”.

Entonces creamos el primer Test y le agregamos un aserto para que este falle. Gran parte de las utilidades que nos ofrece tsqlt está en forma de store procedures es por eso que constamente utilizamores el **EXEC** para llamar a estos procedures.

Podemos observar que dentro de los procedures creados dentro del esquema tsqlt hay otros procedures que nos permiten escribir asertos.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName] AS

BEGIN

EXEC tSQLt.Fail 'Test Incomplete'

END

GO

Para ejecutar todos los Test

EXEC tSQLt.RunAll

Podemos observar que dentro de los procedures creados dentro del esquema tsqlt hay otros procedures que nos permiten ejecutar los tests, a nivel de fixture, por nombre, etc.

Una característica muy importante de tsqlt, en general de las herramientas que trabajan de manera interna, es que deben se crean utilizando el patrón del “self-contained tests”.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

END

GO

Ejecutamos el procedure y contamos el número de filas devueltas y escribimos el assert.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

-- ACT

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

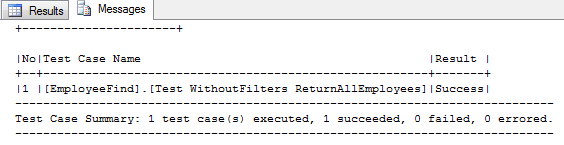
EXEC tSQLt.assertEquals 1, @RowCount;

END

GO

Lo que nos falta ahora, es como mantener en un estado conocido la BD luego del tests. Una característica muy importante de tsqlt es que ya implementa internamente el patrón “transaction-rollback”. Con lo cuál todos nuestros tests se ejecutan dentro de una transacción que es revertida al finalizar los mismos.

Si ejecutamos los tests veremos que estos pasan.



Pero también observamos que hay un comportamiento extraño, ya que la ventana de resultados muestra un result set y a la vez la salida del output. Esto se debe a que nuestro procedure devuelve un resultset y el ssms lo muestra automáticamente.

Esto no es tan malo, pero tenemos un problema mayor, que pasaría si nosotros queremos verificar que efectivamene el result set devuelto es la fila que contiene al empleado Perez. Entonces si queremos explorar el result set devuelto para luego examinarlo, tenemos que insertarlo en una tabla, para esto podemos utilizar una tabla temporal.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

END

GO

Ahora necesitamos verificar que el contenido de esta tabla tiene al empleado “Perez”. Tsqlt tiene para esto un aserto que nos permite comparar que el contenido de 2 tablas sea exactamente igual, el asserto falla si es que por lo menos el valor de alguna celda es diferente. Este aserto recibo como parámetro el nombre de las tablas.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO

Ahora necesitamos crear la tabla #expected e insertar los datos que esperamos que se encuentre en la tabla #actual para que estos sean comparados.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Expected (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

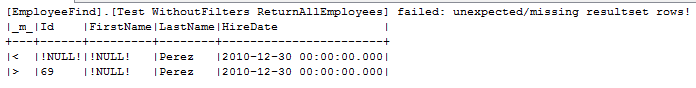
GO

Como ahora ambas tablas van a ser comparadas por completo no es necesario realizar otro aserto sobre el número de filas. Por lo tanto este código ya no es necesario.

DECLARE @RowCount int = @@ROWCOUNT;

EXEC tSQLt.assertEquals 1, @RowCount;

Si ejecutamos el tests observaremos que ha fallado, debido a que nuestra tablas #actual contiene los valores identity que se encuentran en la tabla Employee y son devueltos por el Procedures, asimismo no existe forma de predecir consistemente cuál serál el valor de la columna identity al momento de ejecutar la prueba.



Dentro de nuestros tests necesitamos concentrarnos únicamente en los datos que son relevantes para el test, pero las constraints y los campos autogenerados hacen esto muy difícil. Tsqlt nos ofrece una funcionalidad para crear una nueva tabla exactamente igual pero sin ningún tipo dee constraints. La tabla original será restaurada al final del tests debido al rollback.

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

EXEC tSQLt.FakeTable 'Employee'

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Expected (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

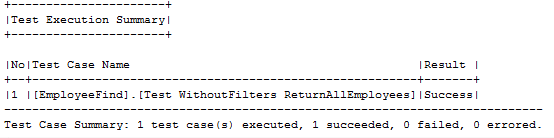
EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO

Luego de esto podremos observar que el tests ya está pasando.



Lo que acabamos de hacer es uno otro de los patrones más comunes al crear pruebas de BD “Compare Table Assert” . En el cuál construimos 2 conjuntos de datos(en este caso tablas): actual y esperado, para compararlos entre así dentro a través de un aserto.

Asimismo hemos podido observar la gran cantidad de código que se necesita para crear una prueba y lo poco mantenibles que són ya que cualquier cambio en el procedure o en alguna de las tablas originales, ocasionará que tengamos que modificar varios lugares en varias pruebas.

Para solucionar lo primero, tsqlt también nos ofrece procedimientos Setup que podemos agregar al Test Fixture para que sean ejecutados siempre antes de las pruebas. Vamos a refactorizar la prueba para incluir el Setup. La convención para crear un setup es que el nombre del procedure debe ser [Nombre Test Fixture].[Setup]

CREATE PROCEDURE EmployeeFind.SetUP

AS

BEGIN

EXEC tSQLt.FakeTable 'Employee'

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Perez','2010-12-30')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO [dbo].[Employee](LastName,HireDate) VALUES('Tovar','2010-12-28')

CREATE TABLE EmployeeFind.Expected (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

CREATE TABLE EmployeeFind.Actual (

Id int,

FirstName nvarchar(max),

LastName nvarchar(max),

HireDate DateTime

);

END

GO

CREATE PROCEDURE EmployeeFind.[Test WithLastNameFilter ReturnTheEmployeesWithTheExactLastName]

AS

BEGIN

-- ARRANGE

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

EXEC EmployeeFind 'Perez',null,null

DECLARE @RowCount int = @@ROWCOUNT;

-- ASSERT

EXEC tSQLt.assertEquals 1, @RowCount;

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO

Podemos crear otro test y verificar que este pase.

CREATE PROCEDURE EmployeeFind.[Test WithHireDateFilters ReturnTheEmployeesBetweenHireDates]

AS

BEGIN

-- ARRANGE

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Garcia','2010-12-29')

INSERT INTO EmployeeFind.Expected(LastName,HireDate) VALUES('Perez','2010-12-30')

-- ACT

INSERT INTO EmployeeFind.Actual

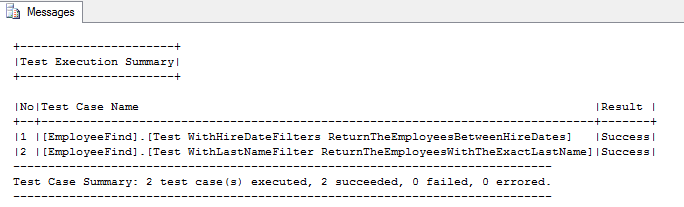
EXEC EmployeeFind null,'2010-12-29','2010-12-30'

-- ASSERT

EXEC tSQLt.AssertEqualsTable 'EmployeeFind.Expected', 'EmployeeFind.Actual';

END

GO



Ahora vamos a crear pruebas para el procedure EmployeeCreate. Si examinamos este procedure veremos que tiene una pequeña lógica condicional en la cuál si no se ingresa la fecha, esta será la fecha actual del sistema.

CREATE PROCEDURE [dbo].[EmployeeCreate]

@FirstName nvarchar(100),

@LastName nvarchar(100),

@HireDate DateTime

AS

BEGIN

IF @HireDate IS NULL

SET @HireDate=GetDate();

INSERT INTO [dbo].[Employee]

([FirstName]

,[LastName]

,[HireDate])

VALUES

(@FirstName

,@LastName

,@HireDate);

END

Creamos la primera prueba.

CREATE PROCEDURE EmployeeCreate.[Test WithExplicitHireDate TheEmployeeHasTheExplicitTheHireDate]

AS

BEGIN

-- ARRANGE

EXEC tSQLt.FakeTable 'Employee'

-- ACT

EXEC EmployeeCreate 'Juan','Marquez','2010-12-31'

DECLARE @Count int=(SELECT count(1) FROM Employee)

-- ASSERT

EXEC tSQLt.AssertEquals 1, @Count

END

Esta prueba no es suficiente ya que nuestro procedures tiene segmentos lógicos que necesitan se probados, en este caso necesitamos probar que al insertar el empleadado se debe usar la fecha ingresada si es que esta no es null. Necesitamos verificar que el valor del hiredate que se encontrará en la tabla es el misma fecha ingresada como parámetro del procedure. Para esto podemos volver a utilizar el patrón “Compare Table Assert”.

CREATE PROCEDURE EmployeeCreate.[Test WithExplicitHireDate TheEmployeeHasTheExplicitTheHireDate]

AS

BEGIN

-- ARRANGE

EXEC tSQLt.FakeTable 'Employee'

-- Store Expected Values

SELECT 'Juan' as FirstName

,'Marquez' as LastName

,'2010-12-31' as HireDate

INTO #Expected

-- ACT

EXEC EmployeeCreate 'Juan','Marquez','2010-12-31'

-- Collect Actual Data

SELECT FirstName,LastName,HireDate

INTO #Actual

FROM Employee

-- ASSERT

EXEC tSQLt.AssertEqualsTable '#Expected', '#Actual';

END

Podemos observar que en este caso podemos observar que estamos utilizando tablas temporales y por lo tanto no es necesario crear de manera anticipada las columnas de estas tablas sino que serán creadas apenas se estén ingresen valores la primera vez. Utilizar estas tablas pude ayudar a aumentar un poco la mantenibilidad de las pruebas aunque no siempre pueden ser utilizadas.

Ahora vamos a crear una nueva prueba para verificar la otra condición lógica de el procedure (que se utilize la fecha actual si el parámetro hiredate es nulo).

CREATE PROCEDURE EmployeeCreate.[Test WithoutHireDate TheEmployeeHasTheCurrentHireDate]

AS

BEGIN

EXEC tSQLt.FakeTable 'Employee'

EXEC EmployeeCreate 'Juan','Marquez',null

DECLARE @HireDate DateTime=(SELECT TOP 1 HireDate FROM Employee);

IF @HireDate IS NULL

EXEC tSQLt.Fail 'Expected: NOT NULL but was: NULL'

END

GO

En esta oportunidad ya no vamos a verificar el contenido de las otras columnas, únicamente vamos a verificar que el empleado no será insertado con un valor de nulo. Debido a que la implementación del Procedures asigna la fecha del sistema la HireDate es un muy dificil realizar una comparación con un valor fijo; como alternativa podríamos declarar 2 variables: una que guarde la fecha actual antes de ejecutar el procedure EmployeeCreate y otra que guarde la fecha al finalizar el procedure, como aserto podríamos verificar que el HireDate se encuentra entre esas 2 fechas.

## DB Projects

C:\Program Files (x86)\Microsoft Visual Studio 10.0\VSTSDB\Extensions\SqlServer\2008\DBSchemas\master.dbschema

