Implementando Aplicaciones .NET con ODP.NET & Entity Framework

Por Francisco Riccio

Introducción

Este material está dedicado a detallar algunos conceptos básicos de la arquitectura ADO .NET de Microsoft llevados hacia las aplicaciones que interactúan con la base de datos Oracle.

ADO.NET es la capa dentro de .NET Framework que permite interactuar a las aplicaciones con un motor relacional, tiene especificado una serie de interfaces que todo proveedor de base de datos debe implementar si desea que su tecnología de base de datos pueda integrarse a las aplicaciones de Microsoft .NET.

Se adjunta la arquitectura de ADO.NET:

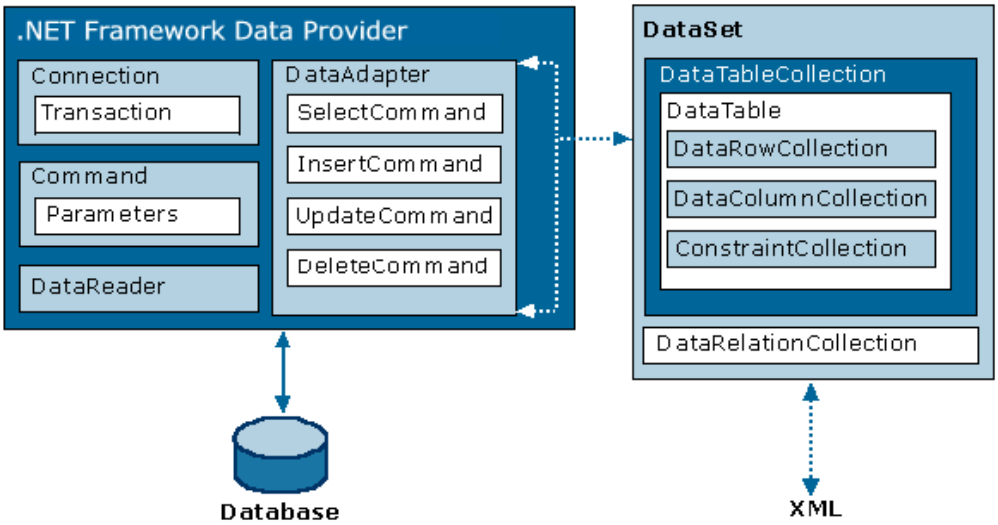

Figura 1

En la arquitectura de ADO.NET se puede trabajar con los datos de manera conectada y desconectada con la base de datos. La clase de que nos permite trabajar de manera conectada es el DataReader el cual será el que se usará en este documento y de manera desconectada se realiza a través de las clases: DataAdapter y DataSet. La clase DataSet almacena información obtenida de la base de datos extraída mediante la clase DataAdapter.

Recomiendo la lectura a la siguiente página web para obtener mayor detalle de ADO.NET:

http://msdn.microsoft.com/en-us/library/e80y5yhx.aspx

Microsoft cuenta con una librería llamada Microsoft Oracle Client como parte de .NET Framework. Esta librería implementa la interfaz de ADO.NET para conectarse a la base de datos Oracle. Oracle Corporation también liberó una librería conocida como ODP.NET entregando mejores resultados de performance y ventajas sobre Microsoft Oracle Client, aprovechando todos la gran mayoría de features de la base de datos directamente.

En la siguiente dirección web se detallan las ventajas entre ODP.NET vs Microsoft Oracle Client:

http://www.oracle.com/technetwork/topics/dotnet/index-154765.html

Cabe mencionar que Microsoft Oracle Client permite implementar aplicaciones con conexión de base de datos Oracle versión 8.1.7 hacia adelante y viene a través del namespace System.Data.OracleClient.

ODP.NET está disponible su descarga gratuitamente en la página oficial de Oracle que se adjunta:

http://www.oracle.com/technetwork/developer-tools/visual-studio/downloads/index.html

Una parte del material estará enfocado a desarrollar un ejemplo de cómo trabajar con ODP.NET para realizar un mantenimiento a una tabla de base de datos Oracle y asimismo la segunda parte del documento implementará el mismo mantenimiento utilizando Entity Framework y LINQ de manera práctica.

Entity Framework es un conjunto de tecnologías diseñadas para desarrollar aplicaciones que accedan a la información en un modelo de aplicaciones conceptual basado en un modelo de objetos y no enfocado a tablas de un modelo relacional. Un similar framework a esté en el mercado es NHibernate.

La manera de como se manipula la información en este tipo de modelo se realiza no mediante sentencias SQL sino a través del lenguaje LINQ. LINQ se encarga de traducir nuestro requerimiento a la sintaxis propia del motor de base de datos donde se conecta.

Ambas implementaciones se han realizado en una configuración con Oracle Database 11g (11.2.0.3) en plataforma Oracle Linux 32 bits y .NET Framework 4.0 con ODP.NET 4. El IDE de desarrollo para las implementaciones es sobre Visual Studio 2010 basado en el lenguaje C#.

Se asume que el lector ha tenido experiencia previa con Visual Studio en la construcción de aplicaciones .NET con acceso a datos; si no fuera el caso recomiendo la revisión de la siguiente página web como referencia:

http://www.oracle.com/technetwork/es/articles/dotnet/cook-dotnet-083575-esa.html

Implementación ODP.NET

El objetivo del siguiente programa a implementar es construir una aplicación .NET que pueda dar mantenimiento a la tabla PRODUCTO la cual reside en la base de datos Oracle.

Se presenta la estructura de la tabla PRODUCTO acompañado de su script de creación.

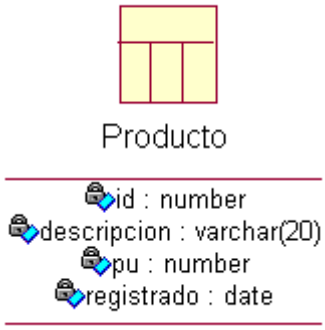

Figura 2

SQL> create table PRODUCTO (id number, descripcion varchar(20), pu number, registrado date);

La arquitectura que tendrá la aplicación está basado en un modelo de tres capas las cuales son: Capa de Acceso a datos encargada de acceder a la fuente de datos provista en este caso con ODP.NET, Lógica de Negocio la cual lleva la implementación de los escenarios y reglas del negocio y la capa de Presentación que podría ser un programa de escritorio, web o móvil. En nuestra implementación se realizará una aplicación de escritorio. La manera de como se lleva la información entre las capas presentadas se realiza mediante la librería Entidades. La librería Entidades lleva la información almacenada en la base de datos mediante objetos entre las diferentes capas mencionadas.

Este tipo de arquitectura nos permite tener una aplicación flexible a cambios y de fácil mantenimiento y soporte.

Se adjunta un gráfico de los componentes mencionados.

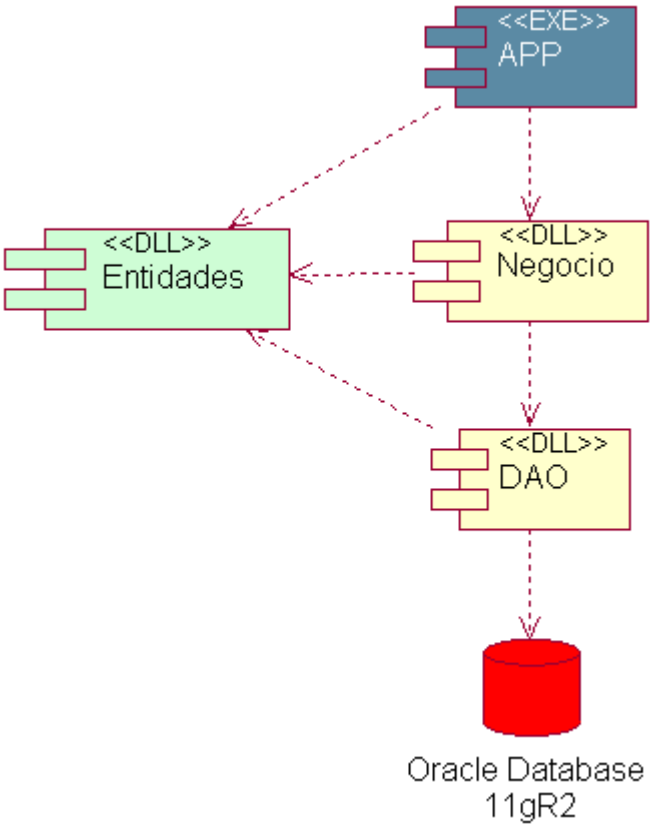

Figura 3

Donde en el Visual Studio se representará en 4 proyectos como se muestra:

La interfaz de la pantalla de mantenimiento de la tabla Producto tendrá la siguiente forma:

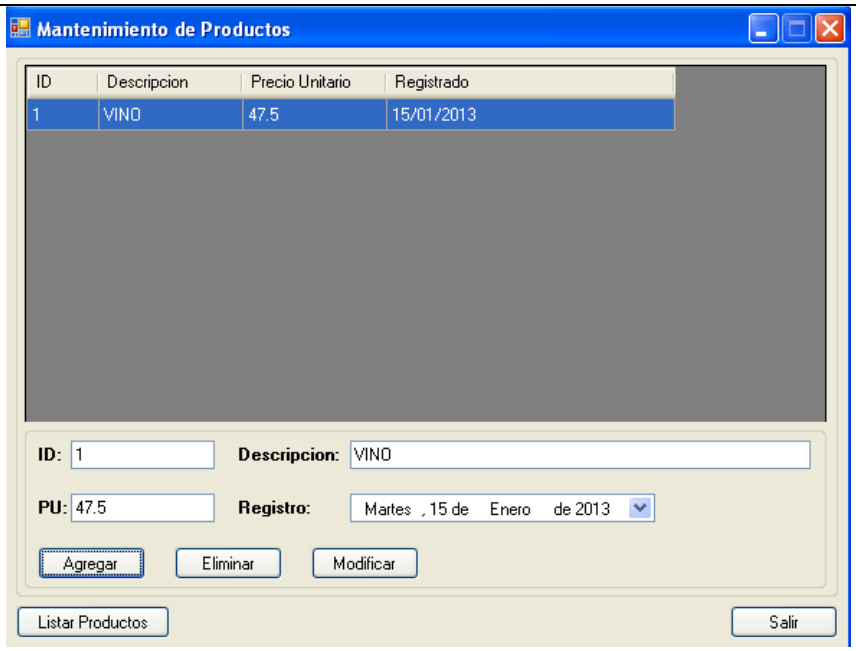

Figura 4

Se detallará el desarrollo de la capa de Acceso a Datos (DAO) mediante la implementación de ADO.NET con ODP.NET.

Capa DAO

Para su implementación se realizó las siguientes actividades:

a) Debemos crear un nuevo proyecto de tipo Class Library, ejemplo:

Figura 5

b) Al nuevo Class Library agregado le añadimos la librería de ODP.NET (Oracle.DataAccess).

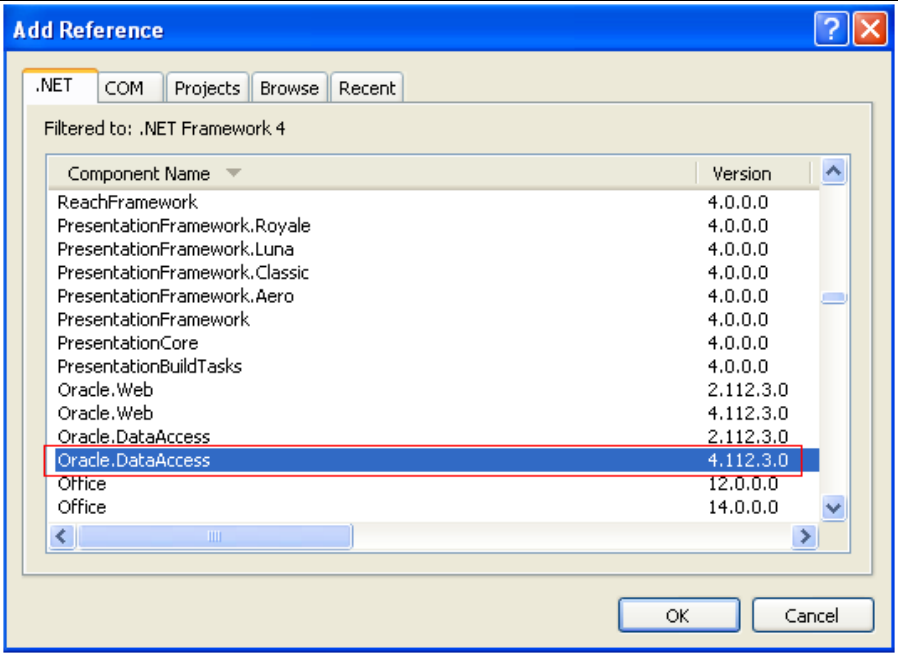

Figura 6

Nota: Esta librería aparecerá en nuestro Visual Studio cuando instalemos ODP.NET sobre la estación de desarrollo.

c) La librería DAO tendrá dos clases: Conexion y ProductoDAO.

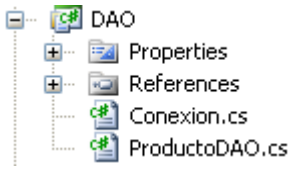

Figura 7

La clase Conexion se encarga de generar la conexión hacia la base de datos y la clase ProductoDAO implementa las operaciones de mantenimiento a la tabla Producto.

d) Se detalla la implementación de la clase Conexión.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Oracle.DataAccess.Client;
using System.Configuration;

namespace DAO
{
    public class Conexion
    {

        private OracleConnection cn { get; set; }

        public OracleConnection getConexion()
        {
            if (cn == null)
            {
                string conexion = System.Configuration.ConfigurationManager.AppSettings["CONEXION"].ToString();
                cn = new OracleConnection(conexion);
            }
            return cn;
        }

    }
}
```

Donde se observa lo siguiente:

- La clase conexión de ODP.NET que nos permite conectarnos a la base de datos se llama OracleConnection el cual está en el namespace Oracle.DataAccessClient.
- La clase OracleConnection recibe la cadena de conexión cuya especificación está almacenada en la configuración de la aplicación (App.config).

Figura 8

El contenido del archivo es:

```
<?xml version="1.0" encoding="utf-8" ?>

<configuration>

  <appSettings>

    <add key="CONEXION" value="Data Source=(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=142.68.1.20)(PORT=1521)))(CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=orcl)));User
```

```
Id=FRICCIO;Password=oracle;Enlist=false;Pooling=true"/>

  </appSettings>

</configuration>
```

El atributo **Pooling** de la conexión de la base de datos permite que la aplicación no esté creando una conexión física a la base de datos en cada solicitud que se necesite una conexión, por tal motivo la conexión se crea una vez y se mantiene establecida en la base de datos con mínimos recursos utilizados en la base de datos. Por default su valor es true y se recomienda dejarlo en ese valor.

El atributo **Enlist** permite que las operaciones DML que se ejecutan mediante la conexión establecida se trabajen como parte de una transacción distribuida. Si no fuera nuestro caso, se debe colocar el valor de false ya que por default tiene el valor de true.

Nota: La siguiente página web contiene una serie de ejemplos de como generar nuestra cadena de conexión con diferentes opciones:

http://www.connectionstrings.com/oracle

e) Se detalla la implementación de la clase ProductoDAO.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Entidades;
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;
using System.Reflection;

namespace DAO
{
    public class ProductoDAO
    {

        public List<EProducto> getProductos()...

        public void insertar(EProducto pProducto)...

        public void eliminar(EProducto pProducto)...

        public void modificar(EProducto pProducto)...

    }

}
```

Se puede observar que la clase implementa una función para obtener la lista de Productos almacenados en la base de datos y sus tres operaciones de mantenimiento (insertar/eliminar/modificar).

La clase utiliza los namespaces: Oracle.DataAccess.Client y Oracle.DataAccess.Types.

Implementación de la función que obtiene la lista de Productos:

La función getProductos devuelve un arreglo que contiene los productos almacenados en la tabla Producto. Cada elemento del arreglo es un objeto de la clase EProducto de la librería Entidades.

Se adjunta la implementación de la clase EProducto de la librería Entidades:

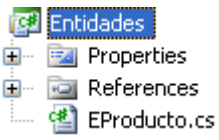

Figura 9

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Entidades
{
    public class EProducto
    {

        public int id { get; set; }
        public string descripcion { get; set; }
        public double pu { get; set; }
        public DateTime registrado { get; set; }

    }

}
```

La función utiliza el stored procedure spu_listarProductos para obtener la lista de productos.

Se adjunta la implementación del Stored Procedure spu_listarProductos:

```
create or replace procedure spu_listarProductos (pcursor out sys_refcursor)

is

begin

 open pcursor for

          select id,descripcion,pu,registrado

           from producto;

end;

/
```

El stored procedure recibe un parámetro de salida (out) de tipo sys_refcursor.

El curso es abierto dentro del stored procedure y las operaciones de fetch serán ejecutadas en la capa de aplicación y cuya responsabilidad será cerrarlo.

Es recomendable que las consultas y las operaciones DML sean implementadas mediante stored procedures ya que entrega mejor performance.

En la siguiente página se detalla la implementación de la clase y como podríamos consumir el stored procedure presentado.

```
public List<EProducto> getProductos()
{
    Conexion objConexion = new Conexion();
    OracleConnection cn = objConexion.getConexion();
    cn.Open();
    OracleCommand cmd = cn.CreateCommand();
    cmd.CommandType = System.Data.CommandType.StoredProcedure;
    cmd.CommandText = "spu_listarProductos";
    OracleParameter par1 = new OracleParameter();
    par1.OracleDbType = OracleDbType.RefCursor;
    par1.Direction = System.Data.ParameterDirection.Output;
    cmd.Parameters.Add(par1);
    cmd.ExecuteNonQuery();
    OracleRefCursor cursor = (OracleRefCursor)par1.Value;
    OracleDataReader dr = cursor.GetDataReader();
    FieldInfo fi = dr.GetType().GetField("m_rowSize", BindingFlags.Instance | BindingFlags.NonPublic);
    int rowsize = Convert.ToInt32(fi.GetValue(dr));
    dr.FetchSize = rowsize * 100;
    List<EProducto> listaProductos = new List<EProducto>();
    while (dr.Read())
    {
        EProducto objProducto = new EProducto();
        objProducto.id = Convert.ToInt32(dr["id"]);
        objProducto.descripcion = dr["descripcion"].ToString();
        objProducto.pu = Convert.ToDouble(dr["pu"]);
        objProducto.registrado = Convert.ToDateTime(dr["registrado"]);
        listaProductos.Add(objProducto);
    }
    cn.Close();
    par1.Dispose();
    cmd.Dispose();
    cn.Dispose();
    objConexion = null;
    return listaProductos;
}
```

Donde:

- La función obtiene una conexión mediante la clase Conexion y con dicho objeto se creará el objeto command (OracleCommand).
- La clase OracleCommand permite ejecutar código SQL & PLSQL directamente desde la aplicación como stored procedures. Para esta acción se define que ejecutaremos un stored procedure e indicamos el procedimiento almacenado a ejecutar.
- Debido a que el stored procedure recibe un parámetro de salida creamos un objeto de tipo OracleParameter y lo definimos que será de salida y de tipo RefCursor.

- El método executeNonQuery ejecuta el stored procedure y éste nos devolverá el cursor ya abierto el cual tiene identificado el conjunto de filas a recuperar.

- Se crea el objeto DataReader mediante el valor obtenido por el cursor (OracleParameter) para recorrer la información y cargarla a nuestro arreglo de Productos.

- Un **fetch** es un conjunto de filas que recoge la capa de aplicación de la base de datos mientras recorre un cursor. Por default este valor es 64 KB, es decir en bloques de 64 KB se va obteniendo todas las filas de un cursor.

  En muchas situaciones es recomendable aumentar el tamaño en ventaja de recoger la información más rápido siendo más eficientes. Esto se realiza con la siguiente línea de código:

  dr.FetchSize = cmd.RowSize * 100;

  En este caso esperamos recibir la información en bloques de 100 filas, por lo cual si una tabla tiene 1000 filas se harán 10 viajes de la capa de base de datos a la capa cliente para obtener la información completa de la tabla.

  La vista V$SQL tiene los campos: executions, fetches y rows_processed los cuales nos pueden ayudar a definir la cantidad de filas hacer retornadas en un fetch de manera eficiente. Por ejemplo: Si obtenemos el ratio de **rows_processed/executions** nos daría la cantidad de filas promedio obtenidas en una ejecución del query. El ratio de **fetches/executions** nos entrega la cantidad de fetchs en cada ejecución. Obteniendo ambos ratios tenemos la cantidad de filas y fetchs de cada ejecución del query los cuales podrían ser reducidos ampliando la cantidad de filas a traer en cada operación de fetch.

  Existe un bug con el atributo RowSize del objeto Command ya que devuelve siempre el valor de 0. Para evitar el bug y conseguir el tamaño en bytes de una fila se ha implementado las siguientes líneas de código.

```
OracleDataReader dr = cursor.GetDataReader();
FieldInfo fi = dr.GetType().GetField("m_rowSize", BindingFlags.Instance | BindingFlags.NonPublic);
int rowsize = Convert.ToInt32(fi.GetValue(dr));
```

Nota 1: En caso siempre una consulta retorne una sola fila con un solo campo es recomendable usar el método ExecuteScalar del objeto OracleCommand, ya que está diseñado a este tipo de escenario entregando mejor performance.

Nota 2: La forma de trabajar con los datos fueron de manera conectada en la función getProductos. La manera desconectada se recomienda cuando queramos crear reportes que realizan operaciones de join entre múltiples tablas, donde la información se almacenará en objetos DataSet.

Implementación del procedimiento insertar:

Se ha diseñado este procedimiento para que ejecute la operación insert desde la capa de aplicación y no mediante un stored procedure. El objetivo es demostrar como ejecutar sentencias desde la capa de aplicación, más no sería lo más óptimo.

```
public void insertar(EProducto pProducto){
    string SQL = "INSERT INTO PRODUCTO(ID,DESCRIPCION,PU,REGISTRADO) VALUES (:PID,:PDESCRIPCION,:PPU,:PREGISTRO)";
    Conexion objConexion = new Conexion();
    OracleConnection cn = objConexion.getConexion();
    cn.Open();

    OracleCommand cmd = cn.CreateCommand();
    cmd.CommandType = System.Data.CommandType.Text;
    cmd.CommandText = SQL;
    cmd.Parameters.Add("PID", OracleDbType.Int32).Value = pProducto.id;
    cmd.Parameters.Add("PDESCRIPCION", OracleDbType.Varchar2).Value = pProducto.descripcion;
    cmd.Parameters.Add("PPU", OracleDbType.Double).Value = pProducto.pu;
    cmd.Parameters.Add("REGISTRO", OracleDbType.Date).Value = pProducto.registrado;

    cmd.ExecuteNonQuery();

    cn.Close();
    cmd.Dispose();
    cn.Dispose();
    objConexion = null;
}
```

A diferencia del caso anterior, aquí el objeto OracleCommand se configura para recibir una sentencia (System.Data.CommandType.Text). La sentencia recibe los parámetros :PID,:PDESCRIPCION,:PPU,:PDESCRIPCION que más adelante son especificados con sus tipos de datos y valores.

Nota: Es recomendable que la sentencia SQL utilice parámetros para que diferentes valores que adopten los parámetros, Oracle Database no cree un plan de ejecución por cada uno de ellos, haciendo que nuestra memoria del Shared Pool de la base de datos sea ineficiente. Usando parámetros evitamos **hard parsing** en la base de datos.

Si deseamos evitar **soft parsing**, debemos mantener cursores en cache en la capa de aplicación evitando repetido procesamiento de metadata. Para realizar esta labor debemos habilitar el cache de sentencias en la cadena de conexión.

En nuestro ejemplo deberíamos agregar la cláusula "Statement Cache Size" a la cadena de conexión como se adjunta en el siguiente ejemplo:

```
Data Source=(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=142.68.1.20)(PORT=1521)))(CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=orcl)));User Id=FRICCIO;Password=oracle;Enlist=false;Pooling=true; Statement Cache Size=50;
```

El valor por default es 10.

Implementación del procedimiento eliminar:

Este procedimiento se ejecuta el stored procedure spu_eliminarProducto el cual se detalla a continuación:

```
create or replace procedure spu_eliminarProducto (pid number)
is
begin
 delete
 from producto
 where id=pid;
end;
/
```

Podemos observar que la confirmación de la transacción no se ha especificado en el código del Stored Procedure porque como buena práctica dicha confirmación debe venir de la capa de aplicación. Por default la conexión establece la propiedad autocommit; es decir terminado una ejecución de una operación DML automáticamente se confirma.

Nota: Una buena práctica es que la confirmación de la operación DML sea declarativo y no esté configurado en autocommit.

En el ejemplo de eliminar un producto se confirma la transacción mediante la propiedad autocommit.

Se adjunta la implementación del procedimiento eliminar de la clase ProductoDAO:

```
public void eliminar(EProducto pProducto)
{
    Conexion objConexion = new Conexion();
    OracleConnection cn = objConexion.getConexion();
    cn.Open();

    OracleCommand cmd = cn.CreateCommand();
    cmd.CommandType = System.Data.CommandType.StoredProcedure;
    cmd.CommandText = "spu_eliminarProducto";
    cmd.Parameters.Add("PID", OracleDbType.Int32).Value = pProducto.id;

    cmd.ExecuteNonQuery();

    cn.Close();
    cmd.Dispose();
    cn.Dispose();
    objConexion = null;
}
```

Implementación del procedimiento modificar:

Este procedimiento se ejecuta el stored procedure spu_modificarProducto el cual se detalla a continuación:

```
create or replace procedure spu_modificarProducto (pid number, pdescripcion varchar, ppu number, pregistrado date)

is

begin

 update producto

 set descripcion=pdescripcion, pu=ppu, registrado=pregistrado

 where id=pid;

end;

/
```

En esta ocasión la confirmación de la operación UPDATE no se ejecutará con la propiedad autocommit que es lo más recomendable.

```
public void modificar(EProducto pProducto)
{
    Conexion objConexion = new Conexion();
    OracleConnection cn = objConexion.getConexion();
    cn.Open();

    OracleCommand cmd = cn.CreateCommand();
    cmd.CommandType = System.Data.CommandType.StoredProcedure;
    cmd.CommandText = "spu_modificarProducto";
    cmd.Parameters.Add("PID", OracleDbType.Int32).Value = pProducto.id;
    cmd.Parameters.Add("PDESCRIPCION", OracleDbType.Varchar2).Value = pProducto.descripcion;
    cmd.Parameters.Add("PPU", OracleDbType.Double).Value = pProducto.pu;
    cmd.Parameters.Add("PREGISTRADO", OracleDbType.Date).Value = pProducto.registrado;

    OracleTransaction tx = cn.BeginTransaction();
    cmd.ExecuteNonQuery();
    tx.Commit();

    cn.Close();
    cmd.Dispose();
    cn.Dispose();
    objConexion = null;
}
```

Para lograr con el objetivo de no utilizar la propiedad autocommit debemos crear un objeto de la clase OracleTransaction y puntualmente debemos ejecutar el procedimiento commit si deseamos confirmar la transacción de un conjunto de operaciones DML que pudiera ejecutar el OracleCommand o el procedimiento rollback si deseamos cancelar.

Nota: ODP.NET nos permite configurar algunos aspectos de globalización de las conexiones que apertura la aplicación hacia la base de datos. Esto lo realizamos a través de la clase OracleGlobalization. En el ejemplo se modifica el formato de fecha y el calendario para las conexiones de base de datos que ha realizado la aplicación.

```
OracleGlobalization objGlobal = OracleGlobalization.GetClientInfo();

objGlobal.DateFormat="DD/MON/YYYY";

objGlobal.Calendar="Persian";

OracleGlobalization.SetThreadInfo(objGlobal);
```

Implementación Entity Framework & LINQ

Para implementar Entity Framework sobre el manejo de una base de datos Oracle debemos contar con Oracle Developer Tools versión 11.2.0.2.40 o superior para Visual Studio.

La implementación a realizar mantiene la misma arquitectura de software presentada en la anterior implementación únicamente la librería de acceso a datos será modificada para ajustarla a Entity Framework.

A continuación se detallará el desarrollo de la implementación.

a) El primer requisito para trabajar con Entity Framework es que nuestras tablas cuenten con primary key en su diseño.

b) Sobre la librería de acceso a datos agregamos un elemento ADO.NET Entity Data Model como se muestra:

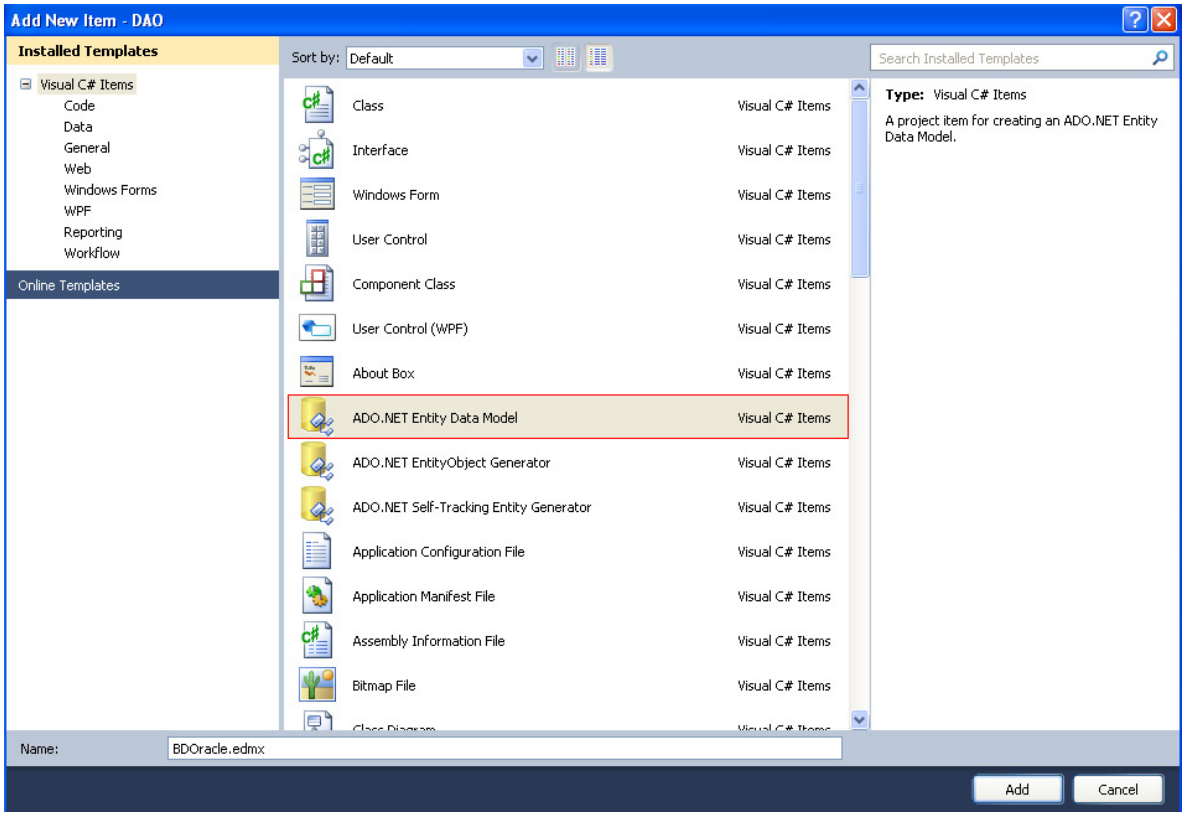

Figura 10

c) Luego traeremos de la base de datos el modelo lógico.

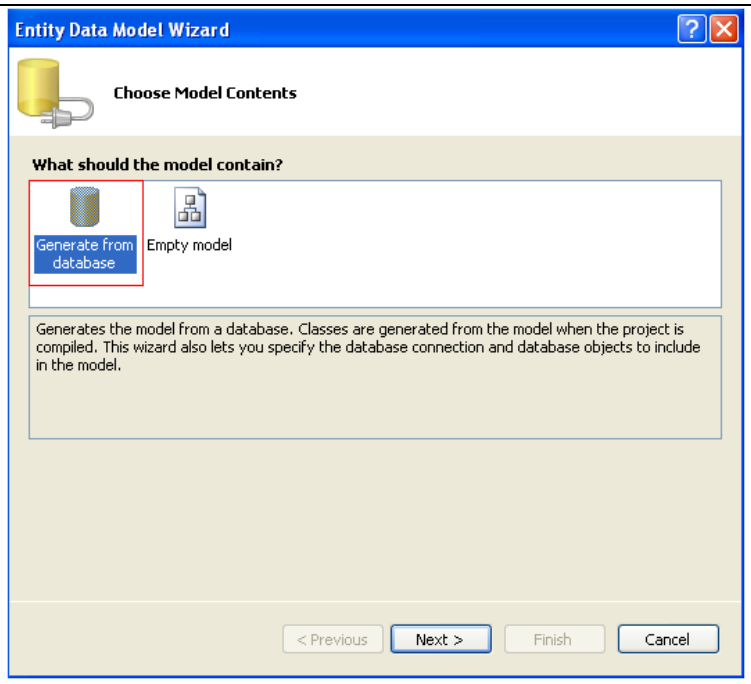

Figura 11

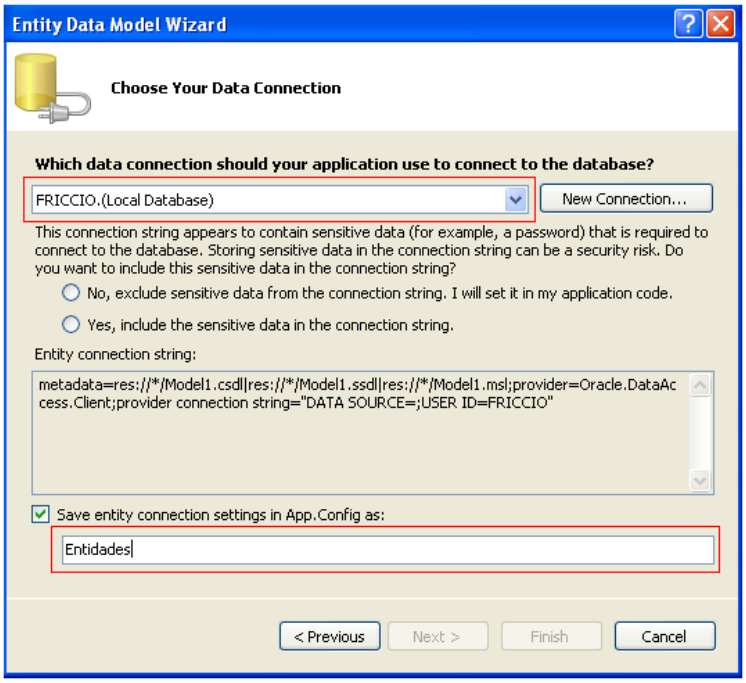

Figura 12

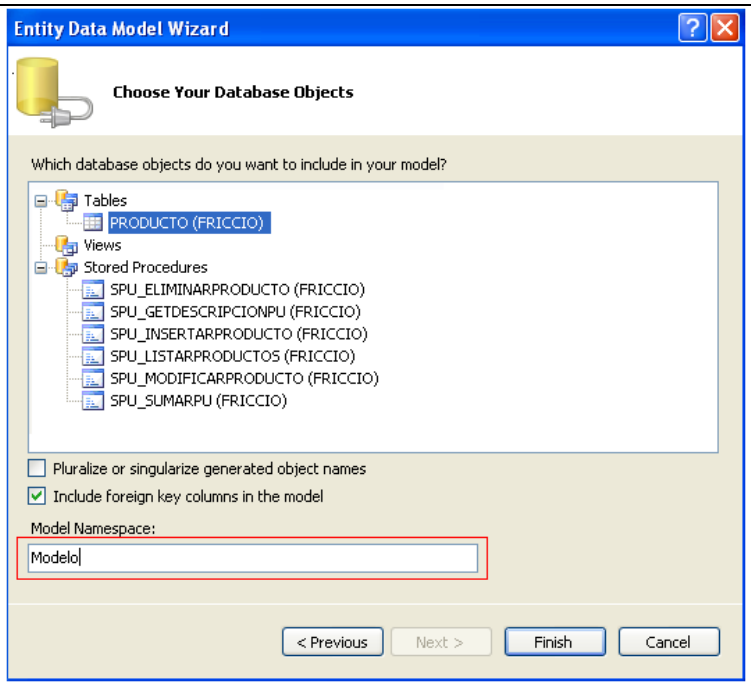

Figura 13

Nota: La clase controladora que nos permitirá trabajar con todo el esquema de Entity Framework acorde a la configuración realizada será Entidades.

Al finalizar la configuración de Entity Framework veremos el diseño de la clase PRODUCTO que representa la tabla Producto de nuestra base de datos.

Figura 14

Nuestra librería de Acceso a Datos debería visualizarse de la siguiente manera:

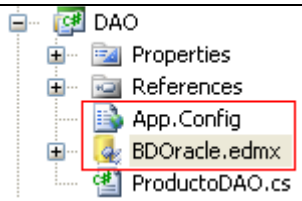

Figura 15

Podemos apreciar que Entity Framework nos ha creado el archivo App.config, el cual lleva la información de la cadena de conexión hacia la base de datos.

Debemos posteriormente agregar la librería System.Data.Entity en el componente de Acceso a Datos.

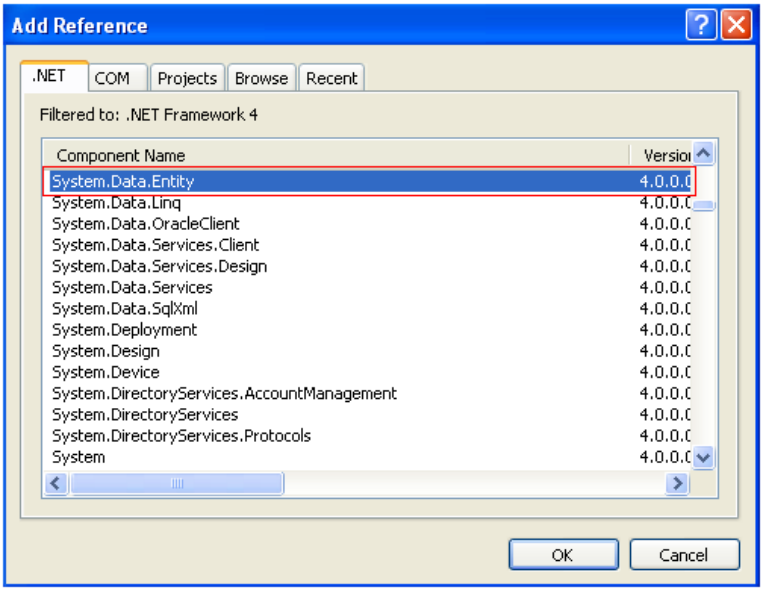

Figura 16

c) Se configurará las propiedades para realizar los mantenimientos a la tabla Producto.

Entity Framework nos provee todo un código ya implementado para realizar el mantenimiento (insert/delete/update) a cada entidad, pero podría ser reemplazado por stored procedures que realicen está labor en pro de conseguir mejor performance.

Para esta labor se reutilizará los stored procedures existentes en la anterior implementación. La única operación a implementar será el INSERT que no estaba realizado hasta este punto.

```
create or replace procedure spu_insertarProducto (ppid number, pdescripcion varchar,
ppu number, pregistrado date)
```

```
is

begin

 insert into producto(id,descripcion,pu,registrado) values
(ppid,pdescripcion,ppu,pregistrado);

end;

/
```

Sobre las propiedades de la clase PRODUCTO de Entity Framework seleccionamos la opción Stored Procedure Mapping.

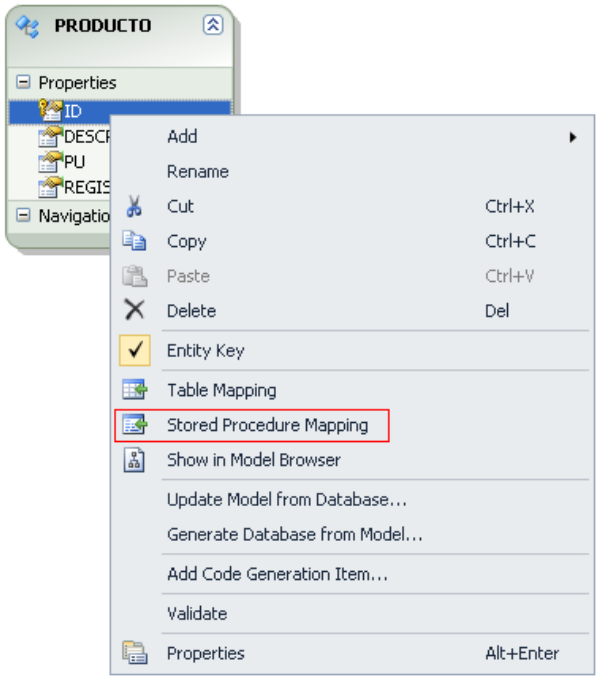

Figura 17

Y en la parte inferior relacionaremos cada operación DML a su stored procedure que ejecutará la acción.

Ejemplo:

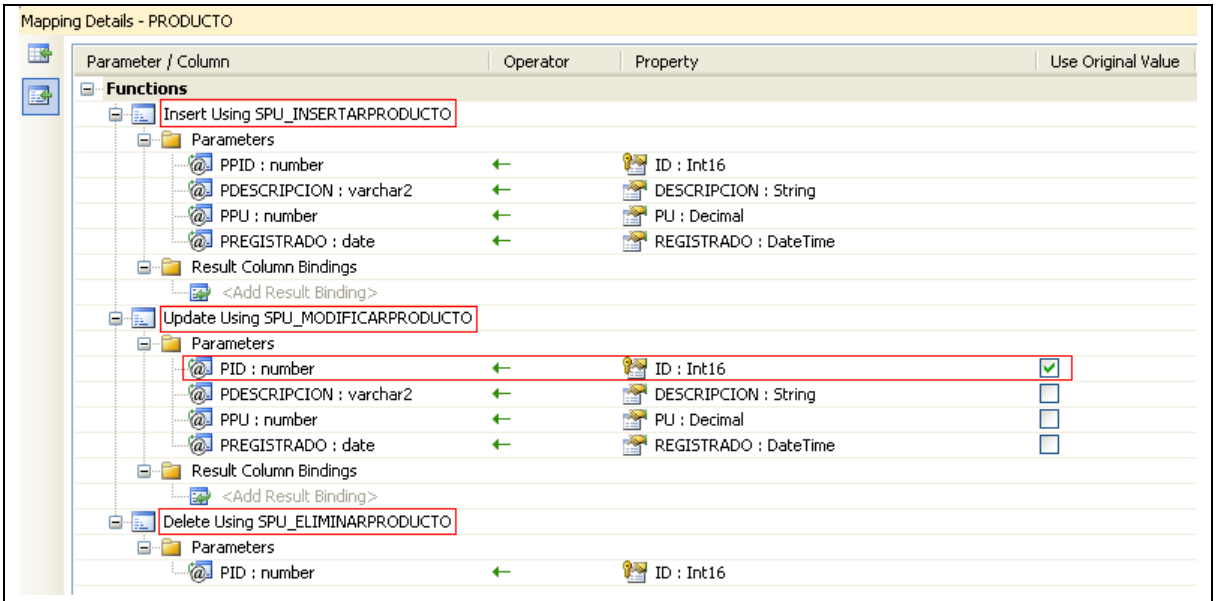

Figura 18

Es importante la operación de UPDATE tenga marcado la propiedad "Use Original Value" para los campos que conforman el primary key de esa entidad.

Nota: Si optamos utilizar stored procedures, las tres operaciones (insert/delete/update) deben implementarse con procedimientos almacenados obligatoriamente.

d) Se presentará las operaciones DML desde la clase ProductoDAO como se realizó en la anterior implementación pero utilizando Entity Framework.

Operación INSERT:

```
public static void insertar(EProducto pProducto)
{
    Entidades ctx = new Entidades();
    PRODUCTO objProducto = new PRODUCTO();
    objProducto.ID = Convert.ToInt16(pProducto.id);
    objProducto.DESCRIPCION = pProducto.descripcion;
    objProducto.PU = pProducto.pu;
    objProducto.REGISTRADO = pProducto.registrado;
    ctx.AddToPRODUCTO(objProducto);
    ctx.SaveChanges();
    ctx.Dispose();
}
```

Donde en el código presentado se crea un objeto CTX de la clase Entidades cuya función será ser el coordinador con las entidades que podamos haber traído de la base de datos para ejecutar operaciones.

En este caso el objeto CTX solicita almacenar un objeto PRODUCTO en la base de datos

mediante las operaciones AddToPRODUCTO y SaveChanges.

Operación DELETE y UPDATE:

```
public static void eliminar(EProducto pProducto)
{
    Entidades ctx = new Entidades();
    PRODUCTO objProducto = (from PROD in ctx.PRODUCTO where PROD.ID == pProducto.id select PROD).FirstOrDefault();
    ctx.DeleteObject(objProducto);
    ctx.SaveChanges();
}

public static void modificar(EProducto pProducto)
{
    Entidades ctx = new Entidades();
    PRODUCTO objProducto = (from PROD in ctx.PRODUCTO where PROD.ID == pProducto.id select PROD).FirstOrDefault();
    objProducto.DESCRIPCION = pProducto.descripcion;
    objProducto.PU = pProducto.pu;
    objProducto.REGISTRADO = pProducto.registrado;
    ctx.SaveChanges();
}
```

En ambos casos solicitamos al objeto coordinador (CTX) que nos devuelva el objeto PRODUCTO a eliminar o modificar. La forma de como se lo solicitamos es mediante el lenguaje LINQ. LINQ es transparente sobre qué motor de base de datos se ejecutará.

La operación delete solicitará la eliminación del objeto mediante el método DeleteObject y SaveChanges y en el caso de la operación Update cualquier cambio en los atributos del objeto Producto serán almacenados en la base datos mediante el método SaveChanges.

En las tres operaciones básicas el objeto coordinador CTX solicitará la ejecución de los stored procedures relacionados a cada operación.

e) Ejecutando otras operaciones.

A continuación se detallará diferentes ejemplos de cómo utilizar EntityFramework con LINQ:

e.1) Obtener la relación completa de productos.

```
public static List<EProducto> getProductos()
{
    Entidades ctx = new Entidades();
    List<EProducto> lista = new List<EProducto>();
    foreach (PRODUCTO prod in ctx.PRODUCTO.ToList()){
        EProducto objProducto = new EProducto();
        objProducto.id = prod.ID;
        objProducto.descripcion = prod.DESCRIPCION;
        objProducto.pu = Convert.ToDecimal(prod.PU);
        objProducto.registrado = Convert.ToDateTime(prod.REGISTRADO);
        lista.Add(objProducto);

    }
    return lista;

}
```

El objeto coordinador CTX con la función PRODUCTO.ToList() nos devuelve un arreglo de PRODUCTOS. Por un tema de diseño estoy transformando la lista de objetos PRODUCTOS a una lista de EPRODUCTOS con la intención de no modificar la capa de Negocio y Presentación de nuestro sistema.

e.2) Obtener un Producto mediante su ID.

```
public static EProducto getProducto(EProducto pProducto)
{
    Entidades ctx = new Entidades();
    int id = pProducto.id;
    PRODUCTO prod = (from PROD in ctx.PRODUCTO.ToList() where PROD.ID == id select PROD).FirstOrDefault();
    EProducto objProducto = new EProducto();
    pProducto.descripcion = prod.DESCRIPCION;
    pProducto.pu = Convert.ToDecimal(prod.PU);
    pProducto.registrado = Convert.ToDateTime(prod.REGISTRADO);
    return pProducto;
}
```

Al objeto coordinador CTX se le solicita la ejecución del siguiente query de LINQ:

```
(from PROD in ctx.PRODUCTO.ToList() where PROD.ID == id select PROD).FirstOrDefault()
```

Donde filtramos de la lista de Productos (ctx.PRODUCTO.ToList()) aquel objeto que tenga en su atributo ID el valor que estamos buscando. El query espera devolver un arreglo como respuesta a la consulta, pero la operación FirstOrDefault() hará que se devuelva el primer objeto del arreglo por lo cual se devuelve un solo objeto PRODUCTO.

e.3) Utilizando LINQ con las operaciones ORDER BY y GROUP BY.

```
public static DataTable getReporteTotalProdFecha()
{
    Entidades ctx = new Entidades();
    List<PRODUCTO> lista;

    lista = (from PROD in ctx.PRODUCTO.ToList() where PROD.PU > 0 && PROD.DESCRIPCION.Contains("V")
             orderby PROD.ID select PROD).ToList();

    var listaAgrupada = (from p in lista group p by p.REGISTRADO into g
                         select new {Fecha = g.Key, Total = g.Count()});

    DataTable dt = new DataTable("reporte");
    dt.Columns.Add("Fecha");
    dt.Columns.Add("Total");
    foreach (var fila in listaAgrupada)
    {
        dt.Rows.Add(fila.Fecha,fila.Total);
    }
    return dt;
}
```

Al objeto coordinador CTX se le solicita la lista de productos cuyo precio unitario es mayor a 0 y su descripción contenga la letra V, el resultado de esta información será ordenada por el campo ID, el resultado se le asigna a la variable lista.

La lista obtenida se le aplicará nuevamente una consulta LINQ solicitando que la lista sea agrupada por el campo registrado obteniendo en cada fecha de registro, la cantidad de productos registrados en ese mismo momento de tiempo, similar a un query como este: SQL> select registrado, count(*) from producto group by registro;

El resultado de esta consulta LINQ sobre la lista es transferida a la variable listaAgrupada definida como var. Las variables definidas con la cláusula var indica que en tiempo de ejecución, la variable obtendrá el tipo de dato que corresponde, en este caso la variable listaAgrupada será un arreglo donde cada elemento se conformará de dos campos.

Posterior la información listaAgrupada es copiada a un DataTable con la finalidad de enviarlo a la capa de Negocio. Por temas de diseño copio el resultado devuelto por LINQ a un DataTable al no contar con una entidad de negocio que represente el reporte que hemos realizado en caso contrario tendría que estar creando Entidades de Negocio por cada reporte que pudiera tener la aplicación, no sería mantenible en el tiempo.

Nota 1: Se puede apreciar que el lenguaje LINQ no solo lo podemos utilizar en Entity Framework sino en cualquier escenario donde manejos arreglos.

Nota 2: Ejecutar operaciones ORDER BY o GROUP BY sobre la base de datos muchas veces tiene un alto costo, podemos evaluar utilizar LINQ para realizar estas labores en la capa de aplicación, liberando recursos en la base de datos. Por experiencia realizar estas labores en la aplicación ha entregado mejores tiempos que realizarlo en la misma base de datos

e.4) Utilización de Stored Procedures

Entity Framework nos permite ejecutar stored procedures que pudieran tener acciones específicas. **No está soportado el uso de funciones PLSQL**.

Ejemplos:

e.4.1) Se desea obtener el la suma de los precios unitarios de todos los productos registrados en la base de datos.

Al no poder utilizar funciones PLSQL en Entity Framework creamos un stored procedure que devuelva mediante un parámetro de tipo out el cálculo solicitado.

Ejemplo:

```
create or replace procedure spu_sumarPU(ptotal out number)
is
 v_total number;
begin
 select sum(pu) into v_total from producto;
 ptotal:=v_total;
end;
/
```

Para registrar el stored procedure en la clase PRODUCTO de Entity Framework ejecutamos los siguientes pasos:

- Botón derecho sobre el modelo de Entity Framework y seleccionamos Add->Function Import.

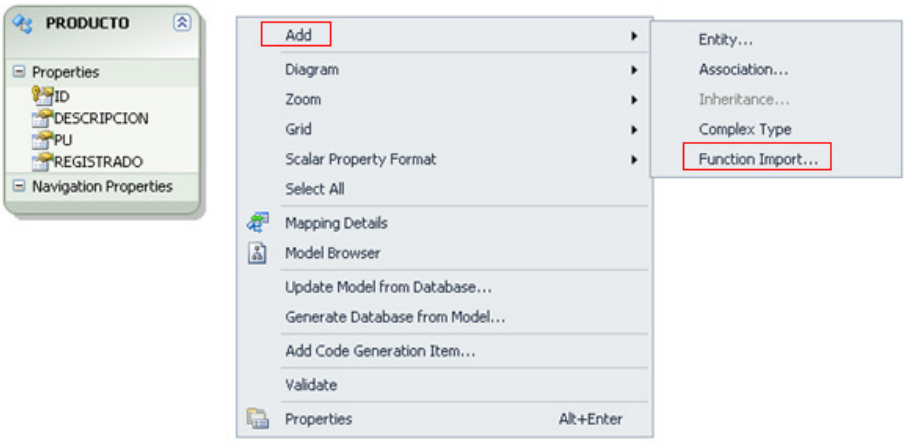

Figura 19

- Luego visualizaremos la siguiente pantalla escogiendo el stored procedure creado y adicional debemos darle un nombre el cual servirá para llamarlo desde el objeto coordinador CTX. Seleccionamos la opción None en Returns a Collection Of porque no se devuelve ninguna colección. simplemente el stored procedure retorna un tipo de dato number.

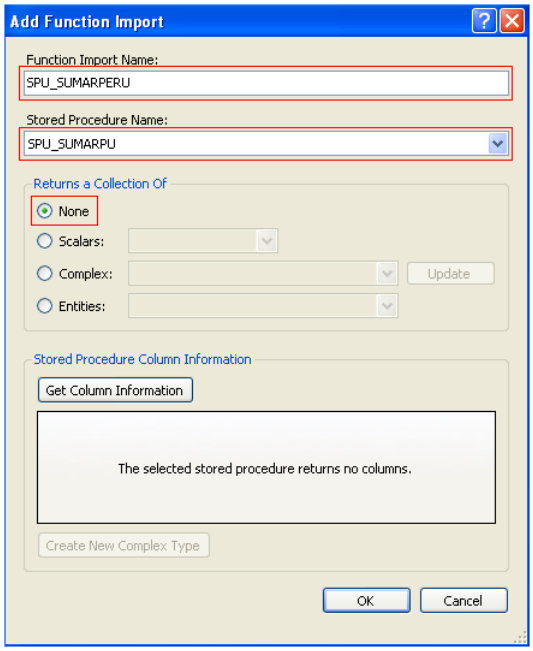

Figura 20

- Invocando el stored procedure registrado.

```
public static double getSumarPU()
{
    Entidades ctx = new Entidades();
    ObjectParameter total = new ObjectParameter("ptotal", typeof(double));
    ctx.SPU_SUMARPERU(total);
    return Convert.ToDouble(total.Value);
}
```

e.4.2) Se desea obtener una lista que contenga la siguiente información: descripción, precio unitario y precio con descuento de cada producto.

Se adjunta el stored procedure que resuelve la solicitud:

```
create or replace procedure spu_getDescripcionPU(pcursor out sys_refcursor)

is

begin

 open pcursor for

        select descripcion, pu, pu*0.5 as pu_dsct

        from producto;

end;

/
```

Para registrar este stored procedure en la clase PRODUCTO de Entity Framework ejecutamos los siguientes pasos:

- El stored procedure al no devolver un tipo de dato simple para Entity Framework (Entero/Cadena/Fecha) debemos indicar los campos que devolverá el cursor con sus tipos de datos en el archivo App.config.

  En nuestro caso, el cursor devuelve tres campos: descripción, precio unitario y precio con descuento.

  Por lo cual el app.config tendría el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>

  <connectionStrings>

    <add name="Entidades"
connectionString="metadata=res://*/BDOracle.csdl|res://*/BDOracle.ssdl|res://*/BDOracle.msl;provider=Ora
```

```
cle.DataAccess.Client;provider connection string=&amp;quot;DATA SOURCE=;PASSWORD=oracle;USER
ID=FRICCIO&amp;quot;" providerName="System.Data.EntityClient" />

  </connectionStrings>

  <oracle.dataaccess.client>

    <settings>

      <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursor.PCURSOR"
value="implicitRefCursor bindinfo='mode=Output'" />

      <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursorMetaData.PCURSOR.Column.0"
value="implicitRefCursor
metadata='ColumnName=descripcion;BaseColumnName=DESCRIPCION;BaseSchemaName=FRICCI
O;BaseTableName=PRODUCTO;NATIVEDATATYPE=Varchar2;ProviderType=Varchar2'" />

      <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursorMetaData.PCURSOR.Column.1"
value="implicitRefCursor
metadata='ColumnName=pu;BaseColumnName=PU;BaseSchemaName=FRICCIO;BaseTableName=
PRODUCTO;NATIVEDATATYPE=Number;ProviderType=Decimal'" />

      <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursorMetaData.PCURSOR.Column.2"
value="implicitRefCursor
metadata='ColumnName=pu_dsct;NATIVEDATATYPE=Number;ProviderType=Int32'" />

    </settings>

  </oracle.dataaccess.client>

</configuration>
```

En este caso, podemos apreciar que primero definimos el parámetro de retorno de tipo RefCursor y luego entramos en detalle sobre las columnas que devolverá el cursor.

En el caso de las dos primeras columnas (descripción y precio unitario) directamente se le relaciona con los campos originales de la tabla Producto pero en el caso del campo precio descuento al ser un campo calculado no se le relaciona a ningún campo de ninguna tabla.

```
<oracle.dataaccess.client>
  <settings>
    <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursor.PCURSOR" value="implicitRefCursor bindinfo='mode=Output'" />

    <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursorMetaData.PCURSOR.Column.0" value="implicitRefCursor
    metadata='ColumnName=descripcion;BaseColumnName=DESCRIPCION;BaseSchemaName=FRICCIO;BaseTableName=PRODUCTO;
    NATIVEDATATYPE=Varchar2;ProviderType=Varchar2'" />

    <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursorMetaData.PCURSOR.Column.1" value="implicitRefCursor
    metadata='ColumnName=pu;BaseColumnName=PU;BaseSchemaName=FRICCIO;BaseTableName=PRODUCTO;
    NATIVEDATATYPE=Number;ProviderType=Decimal'" />

    <add name="FRICCIO.SPU_GETDESCRIPCIONPU.RefCursorMetaData.PCURSOR.Column.2" value="implicitRefCursor
    metadata='ColumnName=pu_dsct;
    NATIVEDATATYPE=Number;ProviderType=Int32'" />

  </settings>
</oracle.dataaccess.client>
```

Ejecutamos los mismos pasos como el caso anterior para registrar el stored

procedure con la diferencia que damos clic en el botón Get Column Information para que recupere la información que devolverá el cursor, asimismo marcamos la opción Complex ya que estamos devolviendo un tipo de dato compuesto en un arreglo.

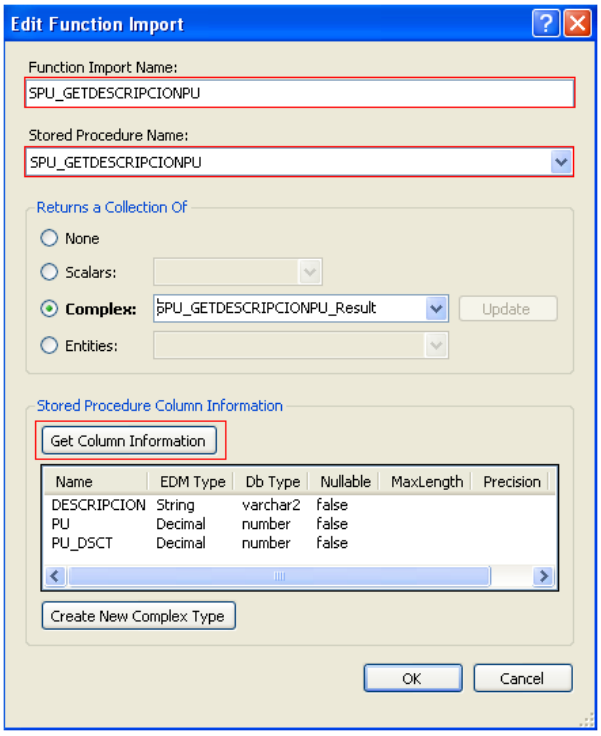

Figura 21

- Invocando el stored procedure registrado.

```
public static DataTable spu_getDescripcionPU()
{
    Entidades ctx = new Entidades();
    var resultado = ctx.SPU_GETDESCRIPCIONPU();
    DataTable dt = new DataTable("reporte");
    dt.Columns.Add("Descripcion");
    dt.Columns.Add("PU");
    dt.Columns.Add("PU_DESCUETO");
    foreach (var fila in resultado)
    {
        dt.Rows.Add(fila.DESCRIPCION, fila.PU, fila.PU_DSCT);
    }
    return dt;
}
```

Nota:Entity Framework solo trabajará con el primer cursor out declarado en un stored procedure, si hubieran más cursores out serán ignorados.

Conclusión

Durante todos los ejemplos revisados se ha visto una gran compatibilidad entre la base de datos Oracle y Microsoft .NET Framework gracias a la librería ODP.NET que comunica ambas tecnologías.

Se pudo apreciar que ODP.NET nos permite crear aplicaciones escalables aprovechando la gran mayoría de features que ofrece la base de datos Oracle además de ofrecer el soporte completo al diseño de desarrollo de software orientado al mapeo de objetos relacionales (ORM) y su persistencia tal como lo hace Entity Framework e implementar lenguajes de consulta como LINQ como complemento a esta tecnología.

Publicado por Ing. Francisco Riccio. Es un IT Specialist en IBM Perú e instructor de cursos oficiales de certificación Oracle. Está reconocido por Oracle como un Oracle ACE y certificado en productos de Oracle Application & Base de Datos.

e-mail: francisco@friccio.com

web: www.friccio.com