Capítulo 1 – ADO.NET – Visão Geral

1. Introdução

- ADO.NET;
- Data Providers;
- DataTable;
- BindingSource;

Chamamos de ADO.NET o conjunto de classes usado com C# e .NET Framework que tem como finalidade viabilizar o acesso a dados em um formato relacional orientado em tabela, como é o caso do Microsoft Access, do Microsoft SQL Server e de outros bancos de dados. No entanto, seu uso não se restringe apenas a fontes relacionais. Por estar integrado em .NET Framework, o ADO.NET pode ser usado com qualquer linguagem .NET, dentre as quais a linguagem C# tem destaque.

2. ADO.NET

O ADO.NET recebeu esse nome devido a um conjunto de classes que foi muito usado em gerações anteriores das tecnologias Microsoft, que recebia o nome de ActiveX Data Objects, ou ADO. A escolha pelo nome ADO.NET ocorreu porque essa foi uma forma que a Microsoft encontrou para demonstrar que era uma substituição de ADO e que era a interface de acesso a dados preferida quando o assunto era o ambiente de programação .NET.

Nota: Devemos atentar para o fato de que, embora a finalidade do ADO.NET e do ADO seja a mesma, as classes, propriedades e métodos de cada um deles são diferentes.

Com os objetos **DataSet** e **DataTable**, oferecidos e otimizados pelo .ADO.NET, podemos armazenar dados na memória, mesmo estando desconectados do banco

de dados. Um objeto **DataTable** pode armazenar na memória, o resultado de um SELECT. Este resultado em memória pode ser alterado e depois salvo novamente na tabela do banco de dados. Já o **DataSet** nada mais é do que um array de DataTables com alguns recursos adicionais que veremos mais adiante.

No ADO.NET, estão o namespace **System.Data**, bem como seus namespaces aninhados, e classes relacionadas a dados provenientes do namespace **System.Xml**. Ao analisarmos as classes de ADO.NET, podemos dizer que elas estão no assembly **System.Data.dll** e assemblies relacionados a **System.Data.xxx.dll** (há exceções, como XML).

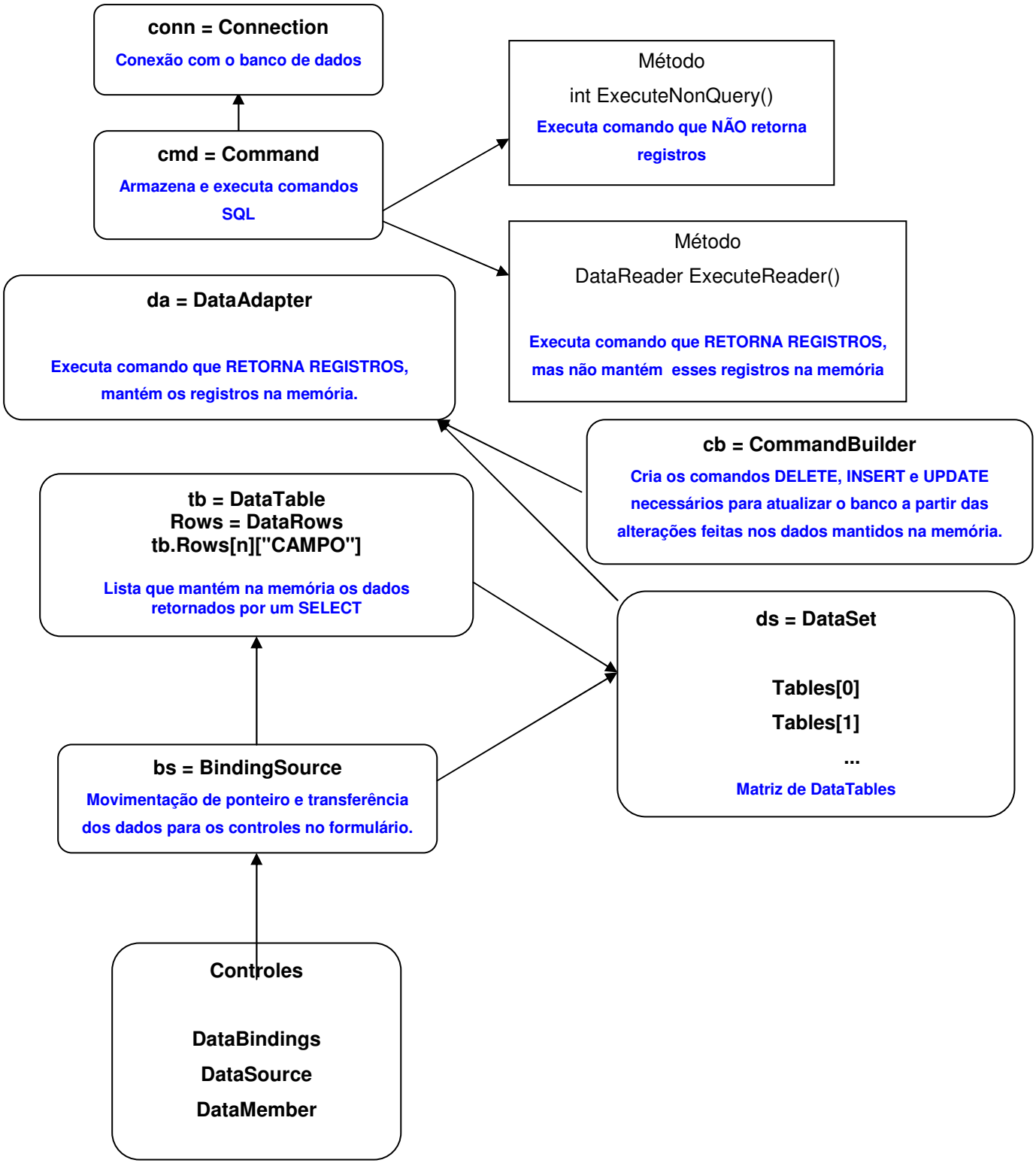
É importante termos em mente que o ADO.NET é constituído de um provedor de dados (data provider) e de um conjunto de dados (dataset). A seguir, esses dois elementos serão abordados com mais detalhes.

3. Data Provider

Existem classes no ADO.NET que atuam juntas para desempenhar o papel de provedor acesso a banco de dados. No entanto, devemos estar conscientes de que bancos de dados diferentes usam maneiras também diferentes de acessar informações, não importa qual seja o tipo de dado, desde o mais simples até o mais complexo. O ADO.Net possui várias famílias de classes para acesso a banco de dados. Todas as famílias possuem classes equivalentes que são:

- Connection: Define os parâmetros de conexão e estabelece a conexão com o banco de dados.
- Command: Define qual comando SQL será executado e o executa.
- Transaction: Controla processos de transação pelo lado da aplicação.
- DataReader: Recebe um leitor de dados gerado por Command e efetua a leitura de um SELECT gerado pela conexão atual e que ainda está armazenado no servidor.

- DataAdapter: Facilita a execução de consultas (SELECT). Permite também atualizar a tabela do banco de dados com base em alterações feitas na memória nos resultados de uma consulta anterior.
- CommandBuilder: Gera comandos DELETE, INSERT e UPDATE com base na estrutura de campos de um SELECT.


conn = Connection
Conexão com o banco de dados
cmd = Command
Armazena e executa comandos SQL
Método
int ExecuteNonQuery()
Executa comando que NÃO retorna registros
Método
DataReader ExecuteReader()
Executa comando que RETORNA REGISTROS, mas não mantém esses registros na memória
da = DataAdapter
Executa comando que RETORNA REGISTROS, mantém os registros na memória.
cb = CommandBuilder
Cria os comandos DELETE, INSERT e UPDATE necessários para atualizar o banco a partir das alterações feitas nos dados mantidos na memória.
tb = DataTable
Rows = DataRows
tb.Rows[n]["CAMPO"]
Lista que mantém na memória os dados retornados por um SELECT
ds = DataSet
Tables[0]
Tables[1]
...
Matriz de DataTables
bs = BindingSource
Movimentação de ponteiro e transferência dos dados para os controles no formulário.
Controles
DataBindings
DataSource
DataMember

Família Sql: Acesso a bancos de dados Microsoft Sql Server

System.Data.SqlClient.**SqlConnection**

SqlCommand:

SqlTransaction:.

SqlDataReader:

SqlDataAdapter:

SqlCommandBuilder:

Família SqlCe: Acesso a bancos de dados Microsoft Sql Server para dispositivos móveis.

System.Data.SqlServerCe.**SqlCeConnection**

SqlCeCommand

SqlCeTransaction

SqlCeDataReader

SqlCeDataAdapter

SqlCeCommandBuilder

Família Oracle: Acesso a bancos de dados Oracle

System.Data.OracleClient.**OracleConnection**

OracleCommand

OracleTransaction

OracleDataReader

OracleDataAdapter

OracleCommandBuilder

Família Odbc: Acesso a qualquer banco de dados desde que tenhamos um driver ODBC instalado para acessar um banco de dados específico.

System.Data.Odbc.**OdbcConnection**

OdbcCommand
OdbcTransaction
OdbcDataReader
OdbcDataAdapter
OdbcCommandBuilder

Família OleDb: Acesso a qualquer banco de dados desde que tenhamos um driver OleDb instalado para acessar um banco de dados específico.

System.Data.OleDb.**OleDbConnection**

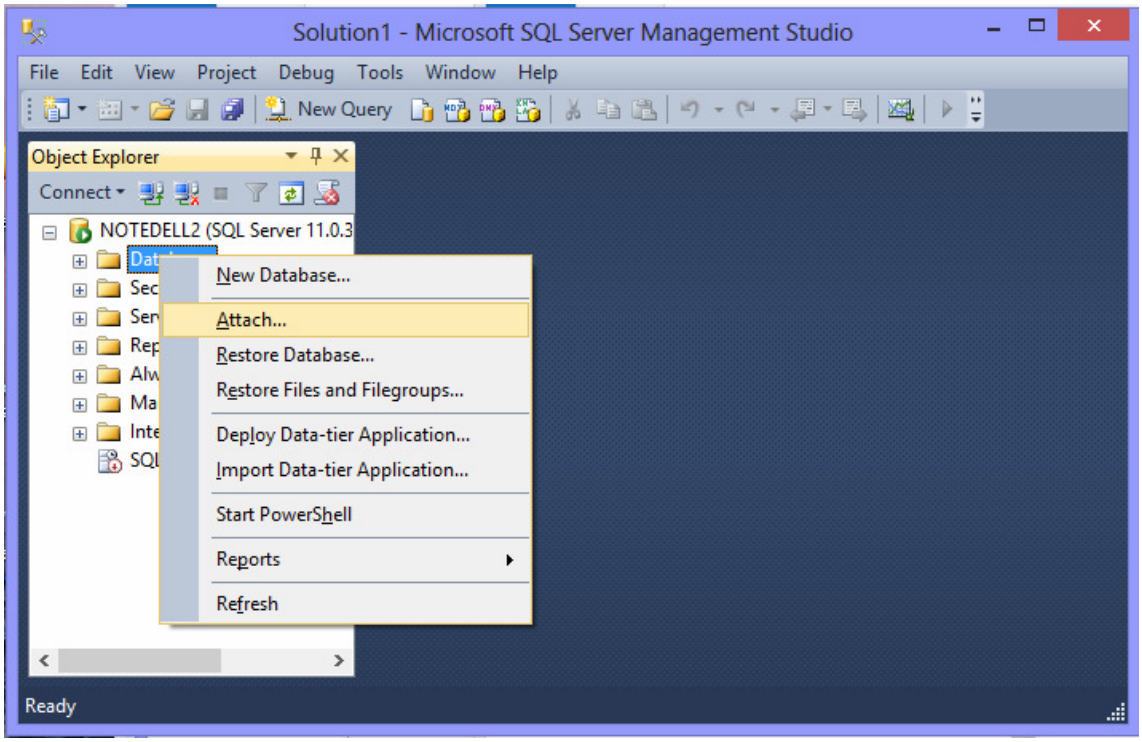
OleDbCommand
OleDbTransaction
OleDbDataReader
OleDbDataAdapter
OleDbCommandBuilder

4. Anexando um banco para ser usado no treinamento

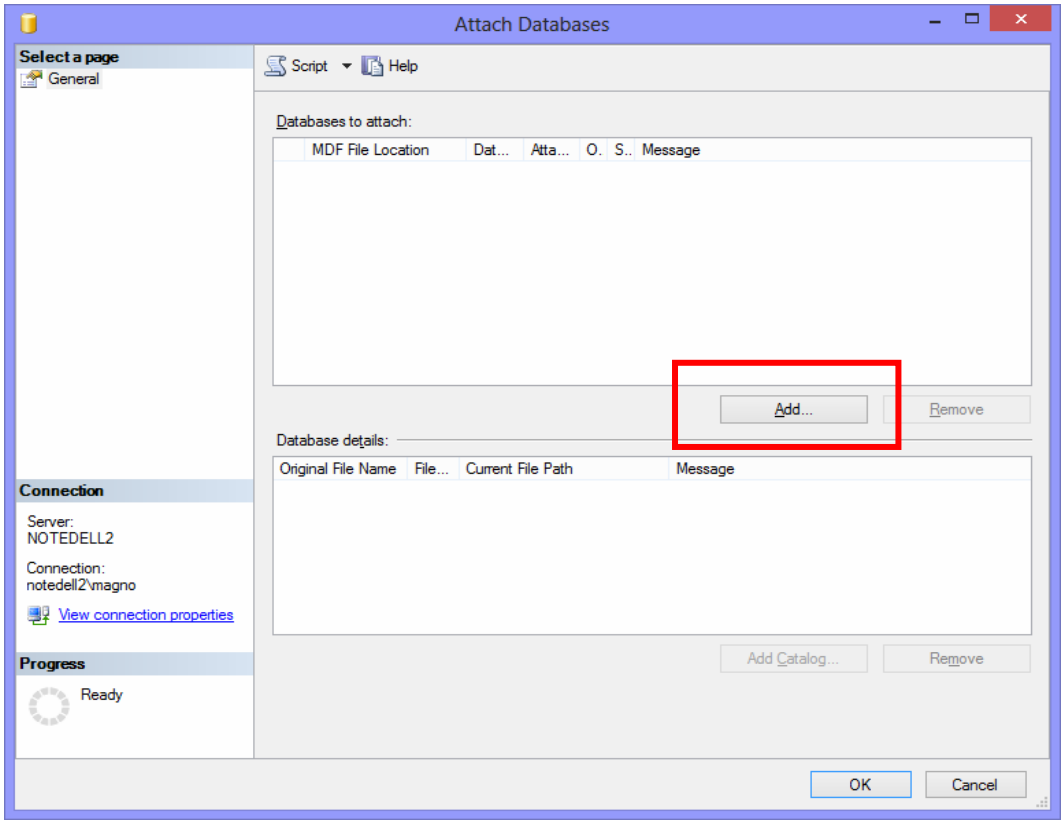
No kit do treinamento, existe uma pasta chamada Dados que contém os arquivos que compõem um banco de dados de PEDIDOS com milhares de registros já cadastrados. Para que possamos usar este banco de dados precisamos registrá-lo no SQL-Server. Siga os passos.

1. Abra o Sql Server Management Studio

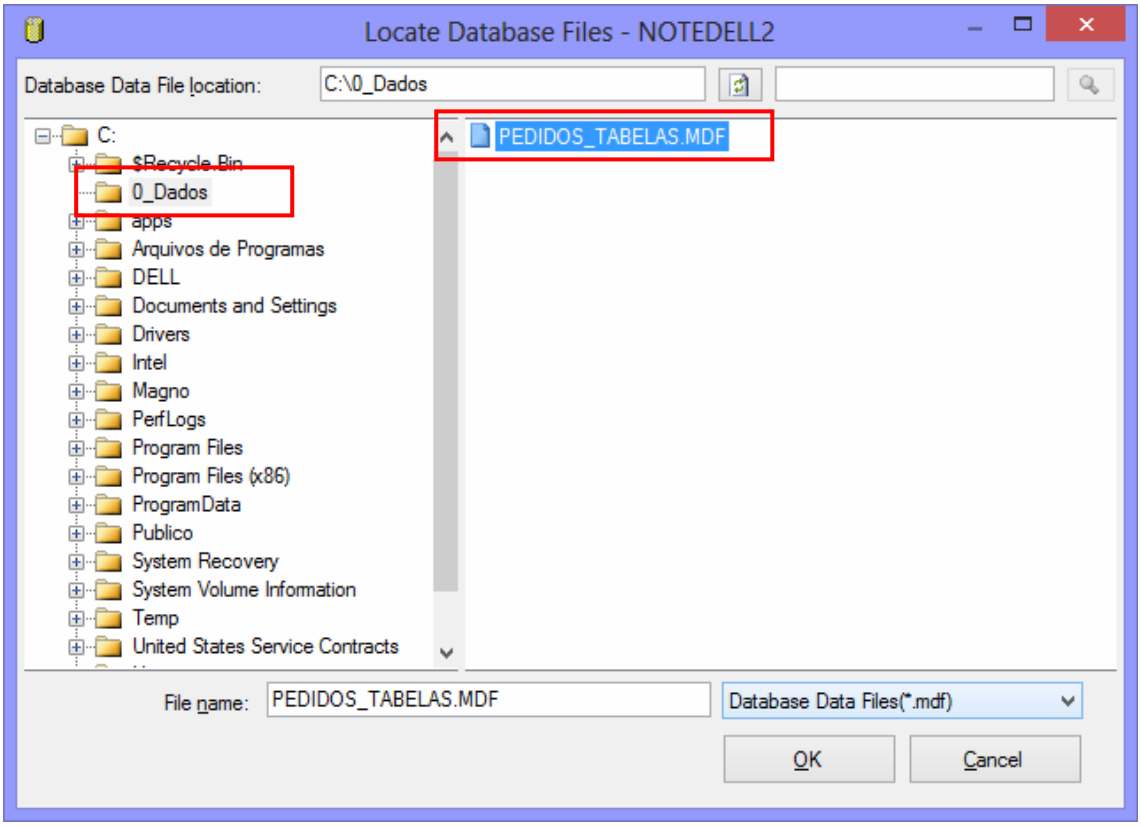
2. No Object Explorer dê um click direito sobre o item Databases e selecione a opção Attach.



2. Na próxima tela, clique no botão Add:

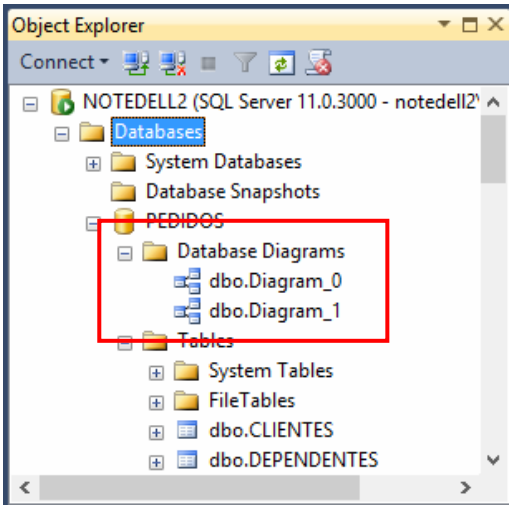


3. Em seguida, procure a pasta Dados e selecione o arquivo PEDIDOS_TABELAS.MDF.



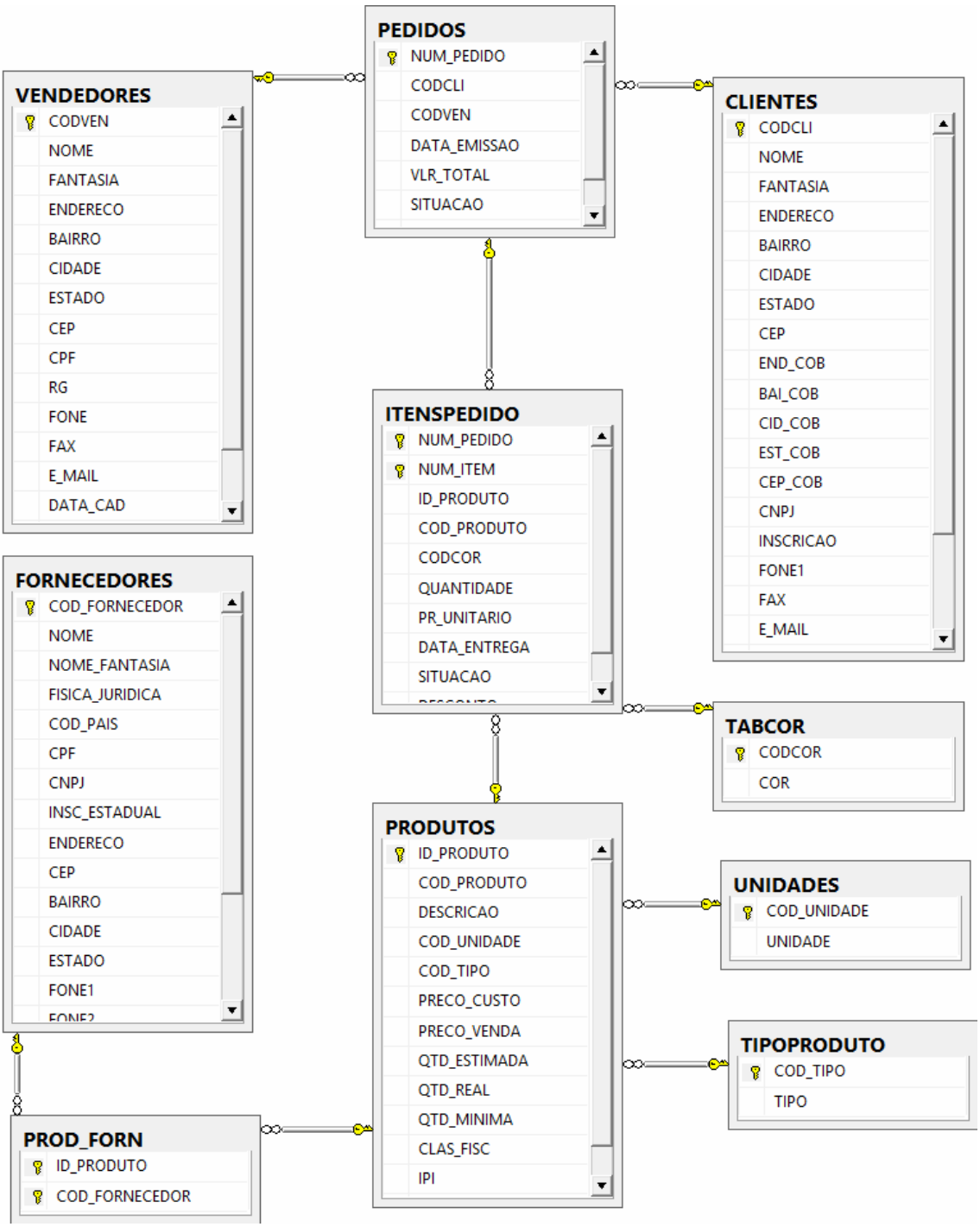
4. Confirme a operação clicando no botão Ok.

O banco de dados aparecerá no Object Explorer. Neste banco foram criados dois diagramas que mostram as tabelas existentes nele. Você conseguirá visualizar o diagrama executando um duplo-clique sobre o nome dele.



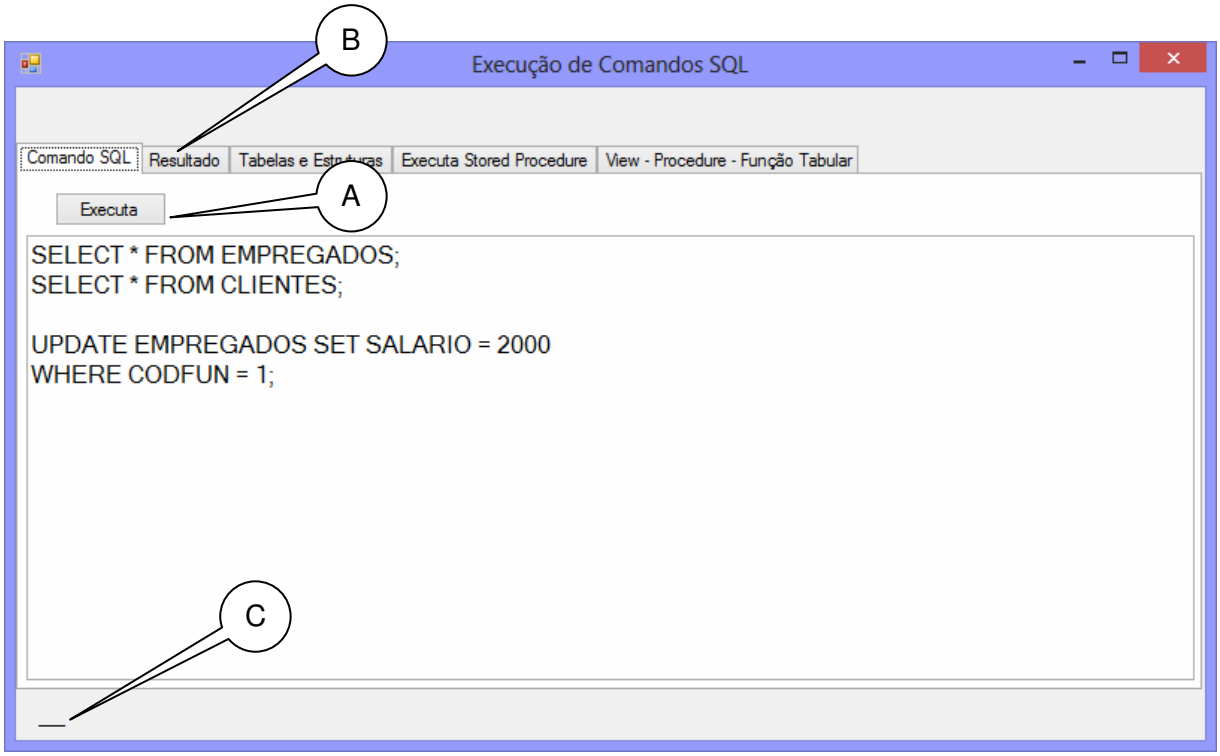
Obs.: Mais a diante veremos como criar um diagrama.

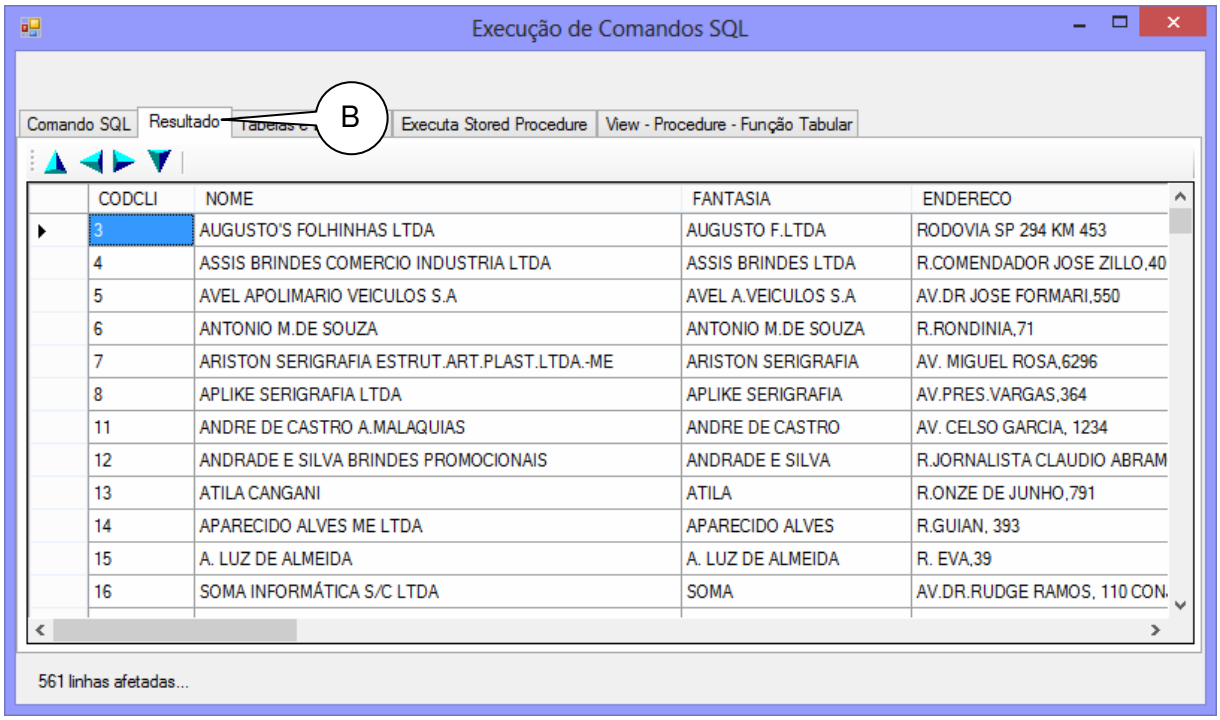
DIAGRAMA DE PEDIDOS



5. Exemplo – Projeto ExecutaSql

Objetivos: Mostrar a maioria dos recursos de acesso à banco de dados existentes no ADO.NET.





CODCLI	NOME	FANTASIA	ENDERECO
3	AUGUSTO'S FOLHINHAS LTDA	AUGUSTO F.LTDA	RODOVIA SP 294 KM 453
4	ASSIS BRINDES COMERCIO INDUSTRIA LTDA	ASSIS BRINDES LTDA	R.COMENDADOR JOSE ZILLO,40
5	AVEL APOLIMARIO VEICULOS S.A	AVEL A.VEICULOS S.A	AV.DR JOSE FORMARI,550
6	ANTONIO M.DE SOUZA	ANTONIO M.DE SOUZA	R.RONDINIA,71
7	ARISTON SERIGRAFIA ESTRUT.ART.PLAST.LTDA.-ME	ARISTON SERIGRAFIA	AV. MIGUEL ROSA,6296
8	APLIKE SERIGRAFIA LTDA	APLIKE SERIGRAFIA	AV.PRES.VARGAS,364
11	ANDRE DE CASTRO A.MALAQUIAS	ANDRE DE CASTRO	AV. CELSO GARCIA, 1234
12	ANDRADE E SILVA BRINDES PROMOCIONAIS	ANDRADE E SILVA	R.JORNALISTA CLAUDIO ABRAM
13	ATILA CANGANI	ATILA	R.ONZE DE JUNHO,791
14	APARECIDO ALVES ME LTDA	APARECIDO ALVES	R.GUIAN, 393
15	A. LUZ DE ALMEIDA	A. LUZ DE ALMEIDA	R. EVA,39
16	SOMA INFORMÁTICA S/C LTDA	SOMA	AV.DR.RUDGE RAMOS, 110 CON

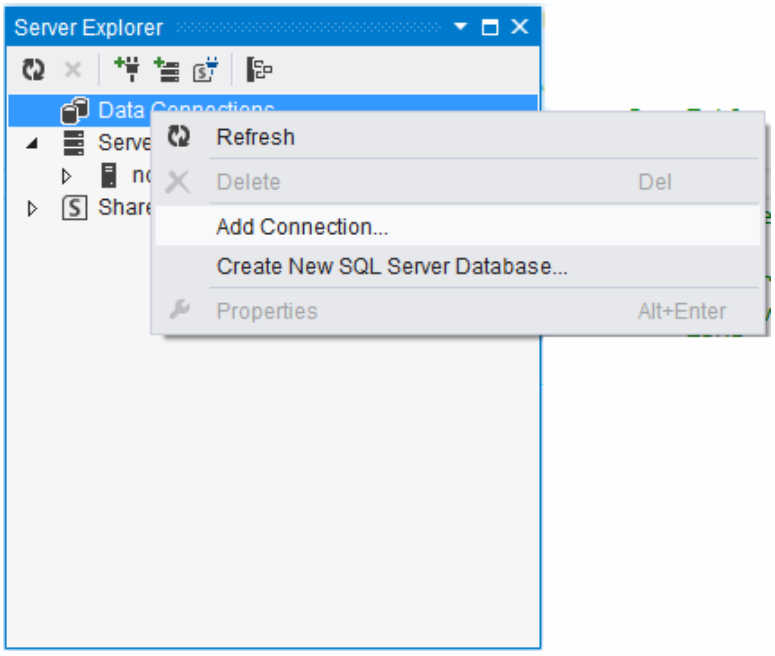
A. Escrevemos o comando SQL no TextBox, selecionamos o comando e clicamos no botão Executa.

B. Se for um comando SELECT mostra os dados na aba Resultado.

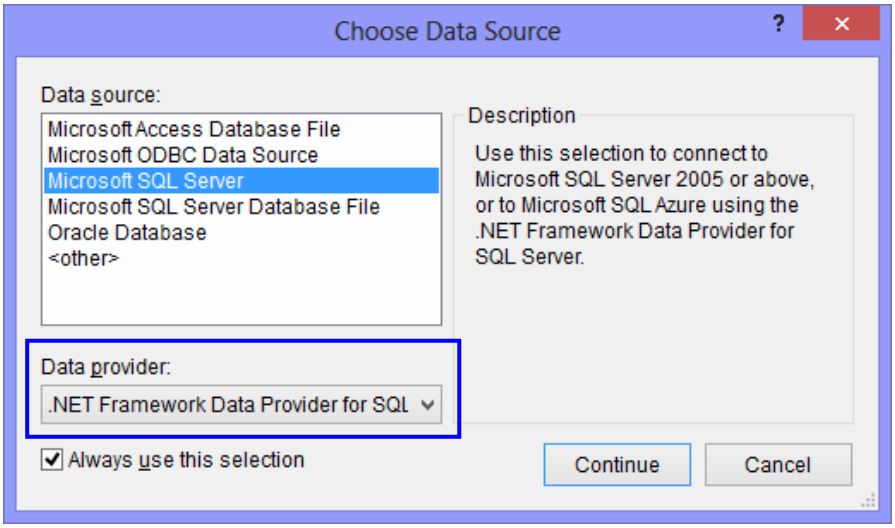
C. Qualquer que seja o comando, a quantidade de linhas afetadas aparecerá no label lblStatus

5.1. Criando o string de conexão

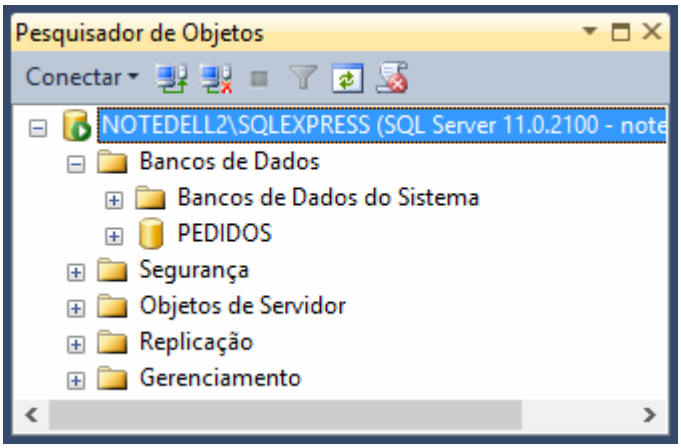
O primeiro passo será a criação do string de conexão. O string de conexão define todos os parâmetros necessários para conectarmos nossa aplicação com o banco de dados. Para facilitar esta tarefa podemos utilizar o Server Explorer do Visual Studio 2012. Caso o Server Explorer não esteja visível, clique no menu VIEW e selecione SERVER EXPLOORER.



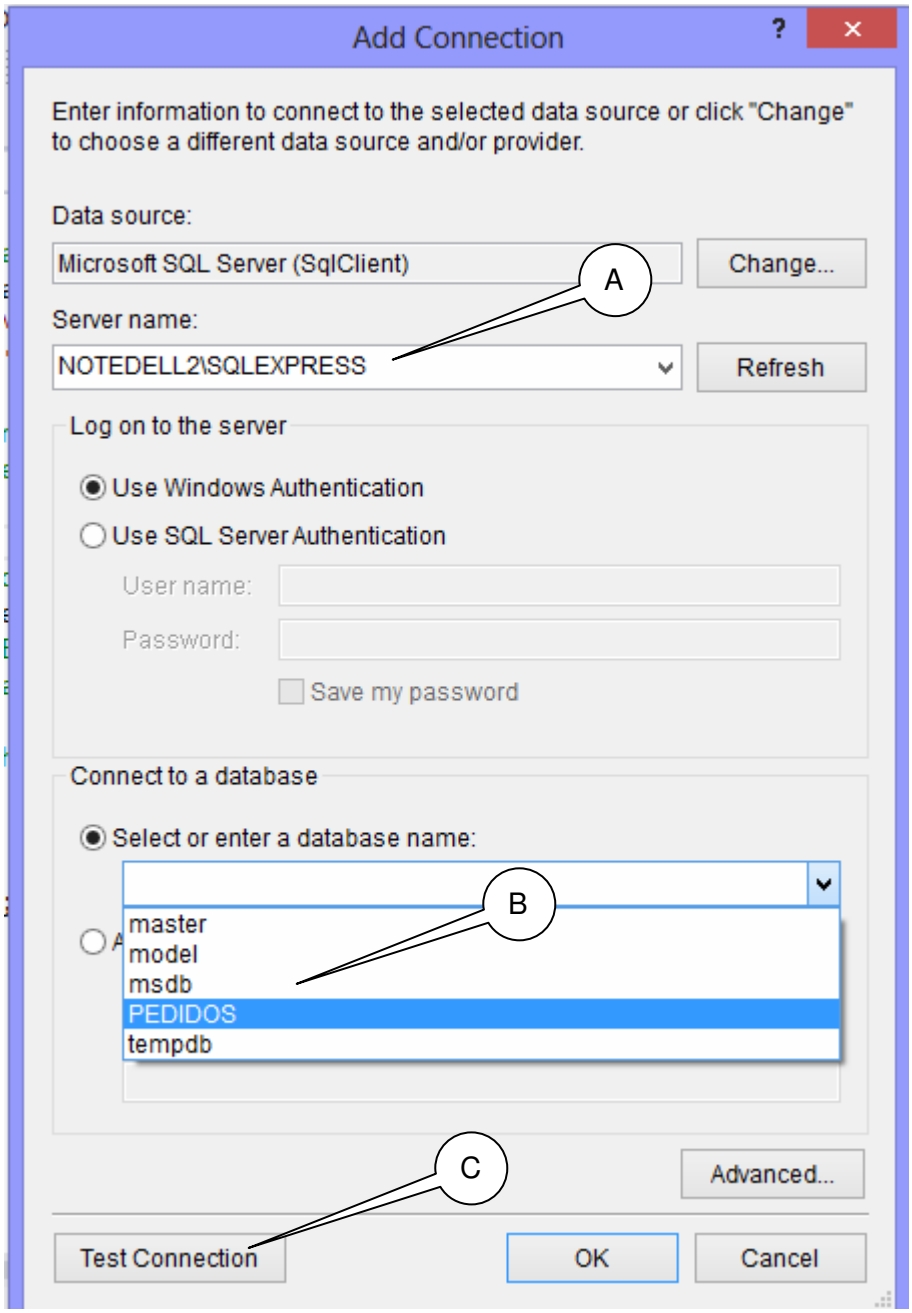
Neste caso usaremos o banco de dados PEDIDOS do Microsoft SQL Server com conexão feita pelo .NET Data Provider for SQL Server.



Observe no SQL Server Management Studio o nome da instância do SQL Server instalada no seu servidor:

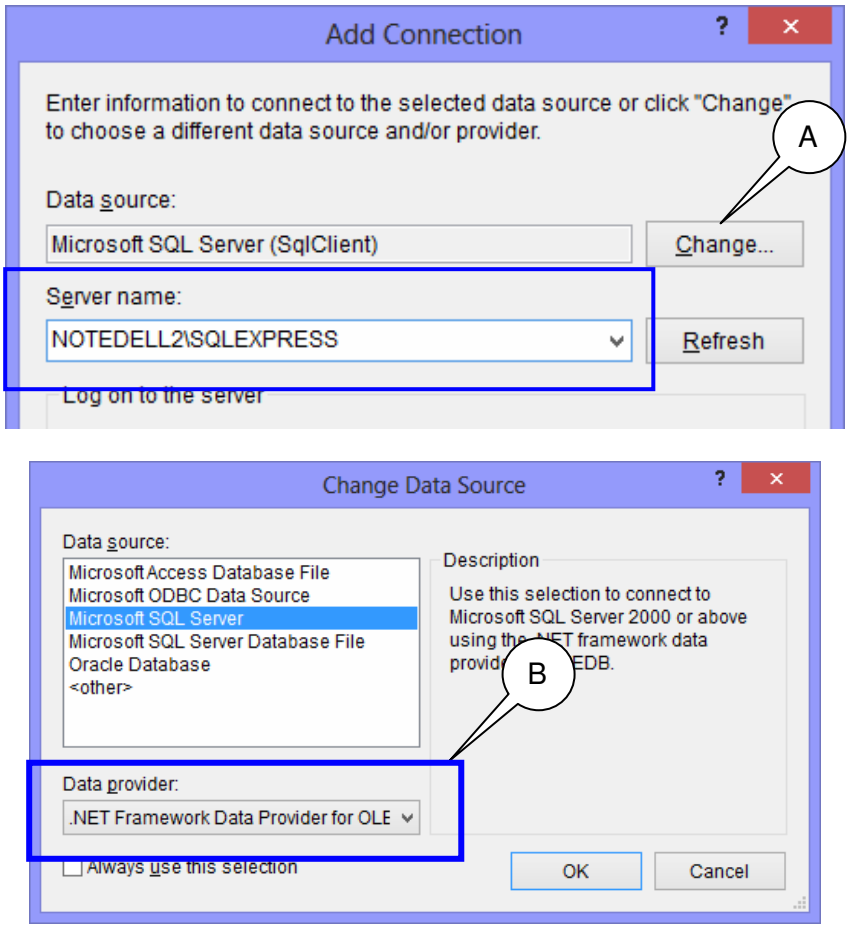


Utilize este mesmo nome na próxima tela do Server Explorer



A. Nome do servidor
B. Selecione o banco de dados que iremos conectar
C. Teste para ver se está tudo OK

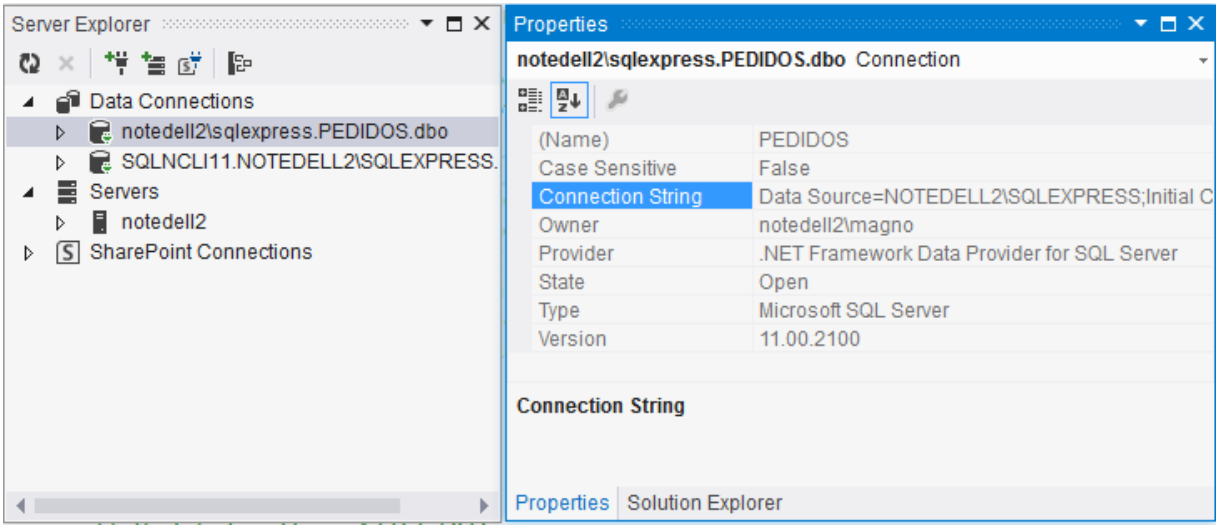
Repita o procedimento, mas agora vamos mudar o Provider para OleDb:



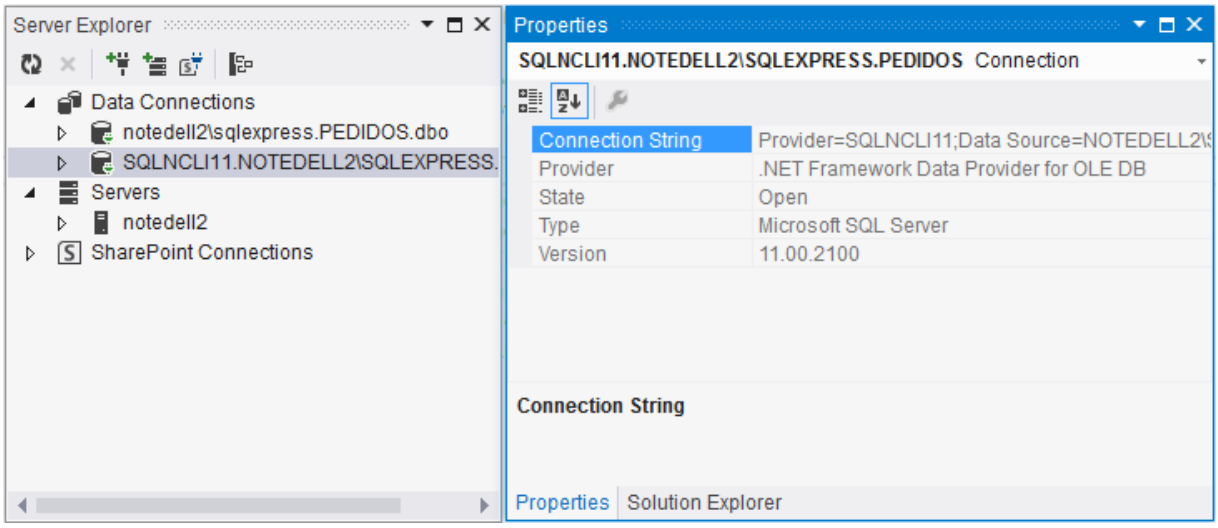
A. Clique no botão Change para alterar o Data Provider
B. Selecione OLE DB.

Depois disso teremos duas conexões disponíveis no Server Explorer e para cada uma delas foi criado um string de conexão:

Conexão com Data Provider for SQL Server



Conexão com Data Provider for OLE DB



5.2. Declarando os objetos de conexão

Agora precisaremos declarar algumas variáveis para uso de todos os métodos do formulário. Como este é um exemplo didático, usaremos no mesmo projeto duas conexões, uma usando Data Provider SQL Server e outra usando OLE DB.

```
namespace ExecutaSQL
{
    public partial class Form1 : Form
    {
        // objeto para conexão com o banco de dados usando driver OleDb
        OleDbConnection connOle = new OleDbConnection(@"Provider=SQLOLEDB;Data
Source=NOTEDELL2\SQLEXPRESS;Integrated Security=SSPI;Initial Catalog=PEDIDOS");
        // objeto para conexão com o banco de dados usando Data Provider for SQL
        SqlConnection connSql = new SqlConnection(@"Data
Source=NOTEDELL2\SQLEXPRESS;Initial Catalog=PEDIDOS;Integrated Security=True");
        // objeto para armazenar resultado de comando SELECT
        DataTable tbSelect = new DataTable();
        // objeto para controle de navegação e posição do ponteiro de registro
        BindingSource bsSelect = new BindingSource();
```

5.3. Urilizando o objeto Command

- Abra um método associado ao evento Click de btnExecSQL.

- O primeiro passo é criar o objeto Command para definirmos o comando que será executado. O objeto Command tem que estar ligado ao Connection:

```
private void btnExecSQL_Click(object sender, EventArgs e)
{
    // para executar qualquer instrução Sql precisamos
    // de um objeto da classe Command
    OleDbCommand cmd = connOle.CreateCommand();
    /*
     * ou
     *
     * OleDbCommand cmd = new OleDbCommand();
     * cmd.Connection = connOle;
     */
```

- Caso não exista nada selecionado no TextBox, o comando a ser executado será todo o texto do TextBox, caso contrário será apenas o texto selecionado. A propriedade **SelectionLength (int)** do TextBox devolve a quantidade de caracteres selecionados. A propriedade **CommandText (string)** do objeto Command armazena o comando que será executado.

```
    // se não tiver texto selecionado
    if (tbxSQL.SelectionLength == 0)
    {
        // define que o comando a ser executado corresponde a
        // todo o texto do TextBox
        cmd.CommandText = tbxSQL.Text;
    }
```

```
else
{
    // define que o comando a ser executado corresponde ao
    // texto selecionado no TextBox
    cmd.CommandText = tbxSQL.SelectedText;
}
```

- Mesmo que o comando esteja sintaticamente correto, existe a possibilidade da ocorrência de erro devido a concorrência ou constraints, então devemos sempre colocar o comando dentro de uma estrutura de tratamento de erro.

```
try
{


}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
finally
{
    connOle.Close();
}
```

- No bloco Try, devemos primeiro verificar se o comando selecionado é um SELECT, porque a forma de executar comando que retorna registros (DataSet) é diferente de executar comandos que não retornam registros.

```
try
{
    int linhas;
    // se o comando começar com a palavra SELECT
    if (cmd.CommandText.ToUpper().Trim().StartsWith("SELECT"))
    {
        // facilita a execução de comando SELECT
        OleDbDataAdapter da = new OleDbDataAdapter(cmd);
        // limpar linhas e colunas do DataTable
        tbSelect.Rows.Clear();
        tbSelect.Columns.Clear();
        // executar o SELECT e preencher o DataTable
        linhas = da.Fill(tbSelect);
        // mostrar os dados na tela
        // BindingSource recebe os dados do DataTable
        bsSelect.DataSource = tbSelect;
        // Bindingsource fornece os dados para a tela
        dgvSelect.DataSource = bsSelect;
        // mudar para a segunda aba do TabControl
        tabControl1.SelectedIndex = 1;
    }
    else // não é SELECT
    {
        connOle.Open();
        // executa comando que não retorna registros
        linhas = cmd.ExecuteNonQuery();
    }
    lblStatus.Text = linhas + " linhas afetadas";

}
catch (Exception ex)
```

```
        {
            MessageBox.Show(ex.Message);
        }
        finally
        {
            connOle.Close();
        }
```

- Na verdade, quem executa a instrução SELECT é o método Command.ExecuteReader() que retorna um objeto OleDbDataReader(), porém o trabalho de programação de executar o SELECT e trazer os dados para a memória é grande, para isso foi criada a classe **DataAdapter**, que já tem este código embutido. **A principal finalidade da classe DataAdapter é executar SELECT**.

  O método **Fill()** da classe **DataAdapter** executa as seguintes tarefas:

  - Testa se a conexão está aberta, se estiver fechada, abre a conexão
  - Executa o método ExecuteReader() da classee Command e recebe um objeto DataReader.
  - Cria um loop para percorrer as linhas do DataReder e armazena cada linha lida em uma nova linha do DataTable que recebeu como parâmetro.
  - Fecha o DataReader
  - Se inicialmente a conexão estava fechada, fecha a conexão.

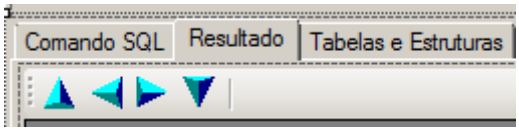
  TESTAR ATÉ ESTE PONTO

- Para podermos executar o comando com F5 ou Ctrl + E, crie um evento KeyDown para o TextBox tbxSQL.

```
private void tbxSQL_KeyDown(object sender, KeyEventArgs e)
{
    // se foi pressionada a tecla F5 OU
    if (e.KeyCode == Keys.F5 ||
        // foi pressionado Ctrl+E
        e.Control && e.KeyCode == Keys.E)
    {
        // executar o botão
        btnExecSQL.PerformClick();
    }
}
```

  TESTAR ATÉ ESTE PONTO

- Botões de navegação da aba "Resultado". Para aplicações Windows Forms existe a classe **BindingSource** que sincroniza o DataTable com os controles visuais que exibem os dados. Esta classe possui métodos para controle e movimentação do ponteiro de registro.

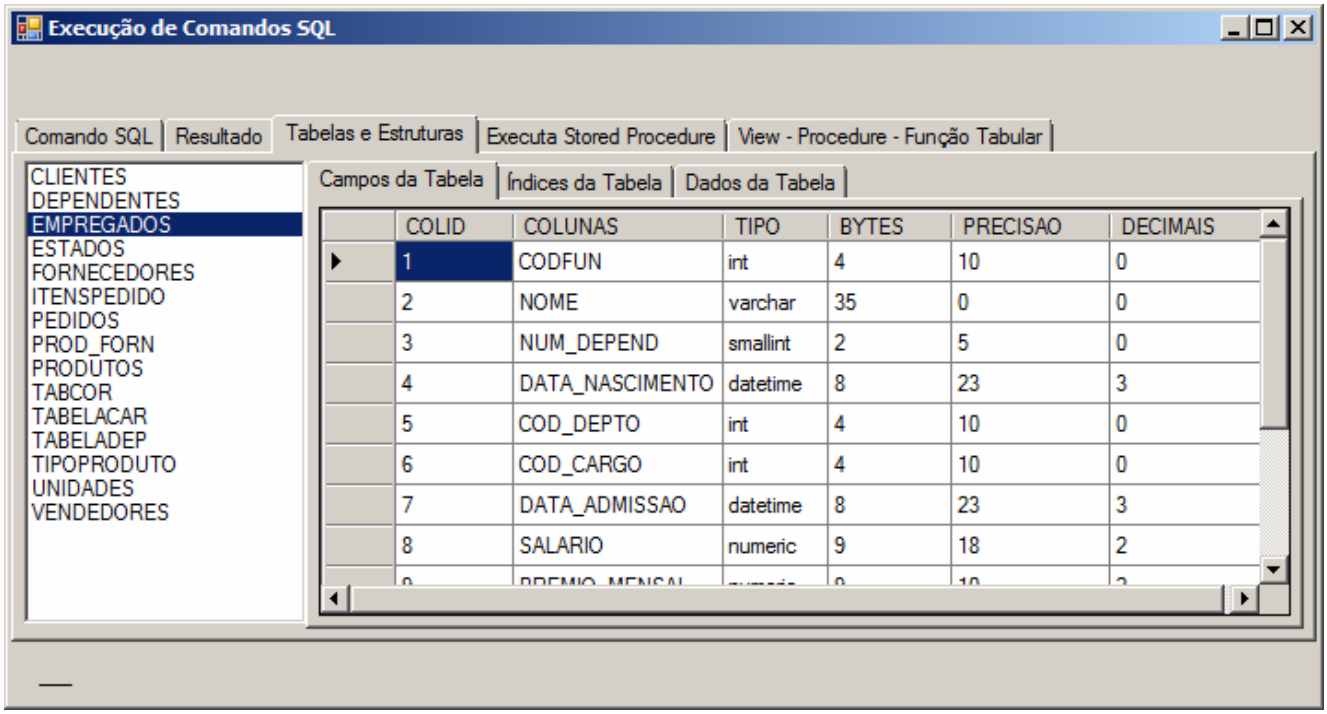



```
private void tsbPrimeiro_Click(object sender, EventArgs e)
{
    bsSelect.MoveFirst();
}

private void tsbAnterior_Click(object sender, EventArgs e)
{
    bsSelect.MovePrevious();
}

private void tsbProximo_Click(object sender, EventArgs e)
{
    bsSelect.MoveNext();
}

private void tsbUltimo_Click(object sender, EventArgs e)
{
    bsSelect.MoveLast();
}
```

TESTAR ATÉ ESTE PONTO

- A seguir, faremos aparecer no ListBox lbxTabelas os nomes de todas as tabelas existentes no banco de dados Pedidos. Para isso executaremos um SELECT que consulta tabelas de sistema do banco de dados. Este comando está disponível no script ExecutaSQL_SELECTS.sql contido no kit do treinamento.

```
--
SELECT NAME  FROM SYSOBJECTS
WHERE XTYPE = 'U' AND name NOT LIKE 'SYS%'
ORDER BY NAME
```



Todo banco de dados MS-SQL possui uma tabela de sistema chamada SYSOBJECTS, esta tabela armazena os nomes de todos os objetos existentes no banco de dados.

```
SELECT NAME, ID, XTYPE  FROM SYSOBJECTS
```

Results | Messages

	NAME	ID	XTYPE
75	STP_MAIOR_PEDIDO	242099903	P
76	PK_ITENSPEDIDO	245575913	PK
77	STP_TOTAL_VENDIDO	274100017	P
78	FN_TOTAL_VENDIDO	290100074	IF
79	DEPENDENTES	293576084	U
80	PK_Dependentes	309576141	PK
81	PEDIDOS	325576198	U
82	PK_PEDIDOS	341576255	PK
83	PRODUTOS	357576312	U
84	PK_PRODUTOS	373576369	PK
85	filestream_tombstone_1205579333	1205579333	IT

Stored Procedure

Chave Primária

Tabela de Usuário

Então se fizermos:

```
--
SELECT NAME  FROM SYSOBJECTS
WHERE XTYPE = 'U' AND name NOT LIKE 'SYS%'
ORDER BY NAME
```

Teremos os nomes de todas as tabelas existentes no banco de dados. Isso deve ser feito no momento em que o formulário for carregado na memória, ou seja, no evento Load do formulário.

```
private void Form1_Load(object sender, EventArgs e)
{
    cmbOrdem.SelectedIndex = 0;
    // definir o comando que será executado
    OleDbCommand cmd = conOle.CreateCommand();
    cmd.CommandText = @"SELECT NAME  FROM SYSOBJECTS
                        WHERE XTYPE = 'U' AND name NOT LIKE 'SYS%'
                        ORDER BY NAME";
    // neste caso, como o SELECT tem apenas uma coluna, vamos
    // executá-lo sem o uso do DataAdapter e DataTable, usaremos
    // DataReader e ExecuteReader()
    // 1. abrir conexão
    conOle.Open();
    // 2. executar o SELECT e o resultado ficará armazenado no banco
    OleDbDataReader dr = cmd.ExecuteReader();
    // 3. ler cada uma das linhas do SELECT e adicionar no listBox
    //    DataReader.Read(): Lê uma linha do SELECT, avança para
    //                       a próxima e retorna TRUE se existir
    //                       próxima linha ou FALSE caso contrário
    while (dr.Read())
    {
        lbxTabelas.Items.Add(dr["name"]);
        // ou
        // lbxTabelas.Items.Add(dr[0]);
    }
    // fechar o DataReader
    dr.Close();
    // fechar a conexão
    conOle.Close();
}
```

- Toda vez que mudarmos de item no ListBox, a estrutura (campos) da tabela devem ser mostrada no grid dgvCampos, ao lado do ListBox.

  Para consultar a estrutura de uma tabela do MS-SQL podemos utilizar o comando (disponível no script ExecutaSQL_SELECTS.sql):

```
-- ESTRUTURA DA TABELA ------------------------------------
SELECT C.COLID, C.NAME AS COLUNAS,
       DT.NAME AS TIPO, C.length AS BYTES, C.xprec AS PRECISAO,
       C.xscale AS DECIMAIS
FROM SYSOBJECTS T JOIN SYSCOLUMNS C ON T.id = C.id
     JOIN SYSTYPES DT ON C.xtype = DT.xtype
WHERE T.XTYPE = 'U' AND T.name = 'PEDIDOS'
      ORDER BY C.COLID;
```

	COLID	COLUNAS	TIPO	BYTES	PRECISAO	DECIMAIS
1	1	NUM_PEDIDO	int	4	10	0
2	2	CODCLI	int	4	10	0
3	3	CODVEN	smallint	2	5	0
4	4	DATA_EMISSAO	datetime	8	23	3
5	5	VLR_TOTAL	numeric	9	18	2
6	6	SITUACAO	varchar	1	0	0
7	7	OBSERVACOES	text	16	0	0

  Porem na condição de filtro o nome da tabela deverá ser uma variável.

```
WHERE T.XTYPE = 'U' AND T.name = 'PEDIDOS'
```

  O evento **SelectedIndexChanged** do ListBox é executado toda vez que mudarmos de item.

```
// executado sempre que mudarmos de item no ListBox
private void lbxTabelas_SelectedIndexChanged(object sender, EventArgs e)
{
    string tabela = lbxTabelas.SelectedItem.ToString();

    mostraCampos(tabela);
    mostraIndices(tabela);
    mostraDados(tabela);
}
```

- Método mostraCampos()

```
void mostraCampos(string tabela)
{
    // 1. definir o comando SELECT
    OleDbCommand cmd = conOle.CreateCommand();
    cmd.CommandText =
        @"SELECT C.COLID, C.NAME AS COLUNAS,
                 DT.NAME AS TIPO, C.length AS BYTES, C.xprec AS PRECISAO,
                 C.xscale AS DECIMAIS
          FROM SYSOBJECTS T JOIN SYSCOLUMNS C ON T.id = C.id
               JOIN SYSTYPES DT ON C.xtype = DT.xtype
          WHERE T.XTYPE = 'U' AND T.name = '" + tabela +
          "' ORDER BY C.COLID;";

    // 2. criar DataAdapter para executar o SELECT
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    // 3. criar DataTable para receber os dados do SELECT
    DataTable tb = new DataTable();
    // 4. executar o SELECT e preencher o DataTable
    da.Fill(tb);
    // 5. mostrar o resultado no grid
    dgvCampos.DataSource = tb;
}
```

Observe que o nome da tabela que queremos ver precisa estar entre apóstrofos na cláusula WHERE:

```
...WHERE T.XTYPE = 'U' AND T.name = '" + tabela +"' ORDER BY C.COLID;"
```

Esta é uma sequência padrão para execução de instrução SELECT, lembrando que podemos ter algumas pequenas variações:

- O DataTable pode ter sido declarado fora do método, neste caso precisaremos apagar suas linhas.
- Se quisermos controlar posição de ponteiro precisaremos de um BindingSource, que receberá os dados do DataTable e fornecerá os dados para a tela.

- Seguindo a mesma sequência crie o método mostraIndices() que deve executar o SELECT a seguir, onde o nome da tabela (CLIENTES) deve ser variável. O resultado deve ser mostrado no grid dgvIndices **(faça você)**:

```
-- ÍNDICES DA TABELA ----------------------------------------
SELECT IX.NAME AS NOME_INDICE, C.name AS CAMPO, IX.type_desc
FROM SYS.INDEXES IX
       JOIN SYSOBJECTS TABELA ON IX.OBJECT_ID = TABELA.ID
       JOIN sys.index_columns IC
         ON IC.object_id = IX.object_id AND IC.index_id = IX.index_id
       JOIN SYSCOLUMNS C
         ON C.id = IC.object_id AND IC.column_id = C.colid
      WHERE TABELA.name = 'CLIENTES'       ON C.id = IC.object_id AND
      IC.column_id = C.colid
      WHERE TABELA.name = 'CLIENTES'
```

- Seguindo a mesma sequência crie o método mostraDados() que deve executar o SELECT a seguir, onde o nome da tabela (CLIENTES) deve ser variável **(faça você)**:

  SELECT * FROM nomeTabela

  Onde nomeTabela é variável, neste caso o SQL, não aceita parâmetro em nomeTabela, precisaremos fazer uma concatenação:

  cmd.CommandText = "SELECT * FROM " + nomeTabela

- O script a seguir (disponível no arquivo ExecutaSQL_CriaStoredProcs.sql) cria alguns objetos no banco de dados pedidos:

  STP_COPIA_PEDIDO: Gera cópia de um pedido já existente e retorna um SELECT contendo o número do pedido gerado.

  STP_COPIA_PEDIDO_P: Gera cópia de um pedido já existente e retorna parâmetros de OUTPUT com o número do pedido gerado.

  VIE_TOTAL_VENDIDO: VIEW que retorna o total vendido em cada um dos meses de 2012.

  STP_TOTAL_VENDIDO: Stored Procedure que retorna o total vendido em cada um dos meses do ano recebido como parâmetro.

  FN_TOTAL_VENDIDO: Função tabular que retorna o total vendido em cada um dos meses do ano recebido como parâmetro.

```
USE PEDIDOS
GO
-- Faz cópia de um pedido e retorna um DataSet de 1 linha contendo
-- o número do pedido gerado e uma mensagem
CREATE PROCEDURE STP_COPIA_PEDIDO @NUM_PEDIDO_FONTE INT
AS BEGIN
-- Declarar variável para armazenar o número do novo pedido
DECLARE @NUM_PEDIDO_NOVO INT;

BEGIN TRAN
-- Abrir bloco de comandos protegidos de erro
BEGIN TRY
      IF NOT EXISTS(SELECT * FROM PEDIDOS WHERE NUM_PEDIDO =
@NUM_PEDIDO_FONTE)
         RAISERROR('PEDIDO NÃO EXISTE',16,1)
      -- Copiar registro de PEDIDOS
      INSERT INTO PEDIDOS (CODCLI, CODVEN, DATA_EMISSAO, VLR_TOTAL,
SITUACAO, OBSERVACOES)
      SELECT CODCLI, CODVEN, GETDATE(), VLR_TOTAL, 'P', OBSERVACOES
      FROM PEDIDOS
      WHERE NUM_PEDIDO = @NUM_PEDIDO_FONTE
      -- Descobrir o NUM_PEDIDO que foi gerado
      SET @NUM_PEDIDO_NOVO = SCOPE_IDENTITY()
      -- Copiar os itens do pedido (ITENSPEDIDO)
```

```
      INSERT INTO ITENSPEDIDO (NUM_PEDIDO, NUM_ITEM, ID_PRODUTO,
COD_PRODUTO, CODCOR, QUANTIDADE, PR_UNITARIO, DATA_ENTREGA, SITUACAO,
DESCONTO)
      SELECT @NUM_PEDIDO_NOVO, NUM_ITEM, ID_PRODUTO, COD_PRODUTO, CODCOR,
QUANTIDADE, PR_UNITARIO, GETDATE()+10, SITUACAO, DESCONTO
      FROM ITENSPEDIDO
      WHERE NUM_PEDIDO = @NUM_PEDIDO_FONTE;
      -- Retornar SELECT com o número do novo pedido
      COMMIT
      SELECT @NUM_PEDIDO_NOVO AS NUM_PEDIDO_NOVO,
             'SUCESSO' AS MSG;
END TRY
-- abrir bloco de tratamento de erro
BEGIN CATCH
    ROLLBACK
    SELECT -1 AS NUM_PEDIDO_NOVO,
           ERROR_MESSAGE() AS MSG
END CATCH
END
GO

-- TESTANDO
EXEC STP_COPIA_PEDIDO 5136
GO
--
================================================================================
-- Faz cópia de um pedido e retorna 2 parâmetros de OUTPUT contendo
-- o número do pedido gerado e uma mensagem
CREATE PROCEDURE STP_COPIA_PEDIDO_P @NUM_PEDIDO_FONTE INT,
                                    @NUM_PEDIDO_NOVO INT OUTPUT,
                                    @MSG VARCHAR(1000) OUTPUT
AS BEGIN
BEGIN TRAN
-- Abrir bloco de comandos protegidos de erro
BEGIN TRY
      IF NOT EXISTS(SELECT * FROM PEDIDOS WHERE NUM_PEDIDO =
@NUM_PEDIDO_FONTE)
         RAISERROR('PEDIDO NÃO EXISTE',16,1)

      -- Copiar registro de PEDIDOS
      INSERT INTO PEDIDOS (CODCLI, CODVEN, DATA_EMISSAO, VLR_TOTAL,
SITUACAO, OBSERVACOES)
      SELECT CODCLI, CODVEN, GETDATE(), VLR_TOTAL, 'P', OBSERVACOES
      FROM PEDIDOS
      WHERE NUM_PEDIDO = @NUM_PEDIDO_FONTE
      -- Descobrir o NUM_PEDIDO que foi gerado
      SET @NUM_PEDIDO_NOVO = SCOPE_IDENTITY()
      -- Copiar os itens do pedido (ITENSPEDIDO)
      INSERT INTO ITENSPEDIDO (NUM_PEDIDO, NUM_ITEM, ID_PRODUTO,
COD_PRODUTO, CODCOR, QUANTIDADE, PR_UNITARIO, DATA_ENTREGA, SITUACAO,
DESCONTO)
      SELECT @NUM_PEDIDO_NOVO, NUM_ITEM, ID_PRODUTO, COD_PRODUTO, CODCOR,
QUANTIDADE, PR_UNITARIO, GETDATE()+10, SITUACAO, DESCONTO
      FROM ITENSPEDIDO
      WHERE NUM_PEDIDO = @NUM_PEDIDO_FONTE;
      -- Retornar mesnagem de sucesso
      SET @MSG = 'SUCESSO';
```

```
      COMMIT
      RETURN 0;
END TRY
-- abrir bloco de tratamento de erro
BEGIN CATCH
    ROLLBACK
    SET @NUM_PEDIDO_NOVO = -1;
    SET @MSG = ERROR_MESSAGE();
    RETURN -1;
END CATCH
END
GO

-- Cria constraint que impete o campo QUANTIDADE de ser menor ou
-- igual a zero. Isto provocará erro ao tentarmos copiar o pedido
-- de número 999
ALTER TABLE ITENSPEDIDO WITH NOCHECK
ADD CONSTRAINT CK_ITENSPEDIDO_QUANTIDADE CHECK(QUANTIDADE > 0)
GO


--
================================================================================
-- Cria uma VIEW para consultar o total vendido em cada
-- um dos meses de 2012
CREATE VIEW VIE_TOTAL_VENDIDO AS
SELECT TOP 12 MONTH( DATA_EMISSAO ) AS MES,
       YEAR( DATA_EMISSAO ) AS ANO,
       SUM( VLR_TOTAL ) AS TOTAL_VENDIDO
FROM PEDIDOS
-- NÃO ACEITA PARÂMETRO. Trabalha somente com constantes
WHERE YEAR(DATA_EMISSAO) = 2012
GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
ORDER BY MES
GO
------------------------------------------------------
-- LIMITAÇÃO: Uma VIEW não pode receber parâmetros
------------------------------------------------------
-- executa com SELECT
SELECT * FROM VIE_TOTAL_VENDIDO
-- Pode fazer ORDER BY
SELECT * FROM VIE_TOTAL_VENDIDO
ORDER BY TOTAL_VENDIDO DESC
GO

--
================================================================================
-- Cria uma STORED PROCEDURE para consultar o total vendido em cada
-- um dos meses do ano que foi passado como parâmetro
CREATE PROCEDURE STP_TOTAL_VENDIDO @ANO INT
AS BEGIN
SELECT MONTH( DATA_EMISSAO ) AS MES,
       YEAR( DATA_EMISSAO ) AS ANO,
       SUM( VLR_TOTAL ) AS TOTAL_VENDIDO
FROM PEDIDOS
-- Aceita parâmetro. Trabalha com dados variáveis
WHERE YEAR(DATA_EMISSAO) = @ANO
```

```
GROUP BY MONTH(DATA_EMISSAO), YEAR(DATA_EMISSAO)
ORDER BY MES
END
GO
-----------------------------------------------------
-- LIMITAÇÃO: É executada com EXEC, não tem JOIN,
-- ORDER BY, WHERE etc...
-----------------------------------------------------
EXEC STP_TOTAL_VENDIDO 2010
EXEC STP_TOTAL_VENDIDO 2011
EXEC STP_TOTAL_VENDIDO 2012
GO

--
=========================================================================
-- Cria uma FUNÇÃO TABULAR para consultar o total vendido em cada
-- um dos meses do ano que foi passado como parâmetro
CREATE FUNCTION FN_TOTAL_VENDIDO( @ANO INT )
RETURNS TABLE
AS
RETURN ( SELECT MONTH( DATA_EMISSAO ) AS MES,
                YEAR( DATA_EMISSAO ) AS ANO,
                SUM( VLR_TOTAL ) AS TOTAL_VENDIDO
                FROM PEDIDOS
                -- Aceita parâmetros. Trabalha com variáveis
                WHERE YEAR(DATA_EMISSAO) = @ANO
                GROUP BY MONTH( DATA_EMISSAO ),
                         YEAR( DATA_EMISSAO ) )
GO
-----------------------------------------------------
-- Executa com SELECT
SELECT * FROM FN_TOTAL_VENDIDO( 2011)
ORDER BY ANO, MES
-- Aceita filtro
SELECT * FROM FN_TOTAL_VENDIDO( 2012)
WHERE MES > 6
      ORDER BY ANO, MES
```

- Botão btnExecReader vai executar a procedure STP_COPIA_PEDIDO, que retorna SELECT. O programador C# não precisa saber criar a stored procedure no SQL, mas ele precisa das seguintes informações.
    - Quais parâmetros temos que passar para a stored procedure
    - Como ele me devolve os resultados (se devolver algo)? Com SELECT ou parâmetros de OUTPUT.
    - Quais são os nomes retornados pelo SELECT ou quais são os nomes dos parâmetros de OUTPUT.

  No caso da stored procedure STP_COPIA_PEDIDO, as respostas são:

    - Precisamos passar o número do pedido já existente, aquele que será copiado. O nome do parâmetro é @NUM_PEDIDO_FONTE INT.
    - Devolve os resultados com SELECT:

```
SELECT @NUM_PEDIDO_NOVO AS NUM_PEDIDO_NOVO,
               'SUCESSO' AS MSG;
```

Onde NUM_PEDIDO_NOVO é o número do novo pedido que foi criado e MSG é a mensagem de erro.

Caso ocorra erro no processo a procedure retornará -1 em NUM_PEDIDO_NOVO e a mensagem de erro correspondente em MSG.

```
private void btnExecReader_Click(object sender, EventArgs e)
{
    // definir o comando que será executado
    OleDbCommand cmd = conOle.CreateCommand();
    cmd.CommandText = "EXEC STP_COPIA_PEDIDO " + updNumPed.Value;
    // como esta procedure retorna um SELECT com apenas 1 linha
    // e 2 colunas não vamos criar DataTable e nem DataAdapter
    conOle.Open();
    // executa comando que retorna SELECT
    OleDbDataReader dr = cmd.ExecuteReader();
    // lê a única linha do SELECT
    dr.Read();

    // ler as colunas desta linha
    int numPed = (int)dr[0]; // (int)dr["NUM_PEDIDO_NOVO"];
    string msg = dr[1].ToString(); // dr["MSG"].ToString();

    dr.Close();
    conOle.Close();

    if (numPed < 0) lblResposta.Text = "Erro: " + msg;
    else
        lblResposta.Text = "Gerado Pedido Numero " + numPed;
}
```

- Botão btnExecParam vai executar a procedure STP_COPIA_PEDIDO_P que devolve parâmetros de OUTPUT e tem "return value"
    - A finalidade desta stored procedure é a mesma da anteriior.
    - Parâmetros de INPUT: @NUM_PEDIDO_FONTE (INT) com o número do pedido que será copiado.
    - Parâmetros de OUTPUT:
        - @NUM_PEDIDO_NOVO INT: Número do novo pedido gerado ou -1 se der erro.
        - @MSG VARCHAR(1000): Mensagem de erro ou sucesso.
    - RETURN_VALUE: -1 se der erro ou zero caso contrário.

Executando a procedure com Data Provider Sql-Server.

```
private void btnExecParam_Click(object sender, EventArgs e)
{
    // criar o objeto Command
    SqlCommand cmd = conSql.CreateCommand();
    // quando a stored procedure devolve parâmetros de OUTPUT não
    // é possível montar o comando EXEC procedure...
    // 1. Definir que o Command vai executar stored procedure
    cmd.CommandType = CommandType.StoredProcedure;
    // 2. CommandText vai receber apenas o nome da stored procedure
    cmd.CommandText = "STP_COPIA_PEDIDO_P";
    // 3. Definir cada um dos parâmetros
    //    parâmetros de INPUT já podemos criar e passar o valor
    cmd.Parameters.AddWithValue("@NUM_PEDIDO_FONTE", updNumPed.Value);
    // parâmetros de OUTPUT somente terão valor após a execução
    cmd.Parameters.Add("@NUM_PEDIDO_NOVO", SqlDbType.Int);
    cmd.Parameters["@NUM_PEDIDO_NOVO"].Direction = ParameterDirection.Output;

    cmd.Parameters.Add("@MSG", SqlDbType.VarChar, 1000);
    cmd.Parameters["@MSG"].Direction = ParameterDirection.Output;

    // se fosse com OleDbCommand, este teria que ser o primeiro
    // parâmetro a ser passado, antes mesmo do param de INPUT
    cmd.Parameters.Add("@RETURN_VALUE", SqlDbType.Int);
    cmd.Parameters["@RETURN_VALUE"].Direction = ParameterDirection.ReturnValue;
    // executar a procedure
    conSql.Open();
    cmd.ExecuteNonQuery();
    conSql.Close();

    // ler os parâmetros de OUTPUT que agora já têm valor
    int numPed = (int)cmd.Parameters["@NUM_PEDIDO_NOVO"].Value;
    string msg = cmd.Parameters["@MSG"].Value.ToString();
    int ret = (int)cmd.Parameters["@RETURN_VALUE"].Value;

    if (ret < 0) lblResposta.Text = "Erro: " + msg;
    else lblResposta.Text = "Gerado pedido numero " + numPed;
}
```

Executando a procedure com Data Provider OleDb.

```
private void btnExecParamOleDb_Click(object sender, EventArgs e)
{
    // criar o objeto Command
    OleDbCommand cmd = conOle.CreateCommand();
    // quando a stored procedure devolve parâmetros de OUTPUT não
    // é possível montar o comando EXEC procedure...
    // 1. Definir que o Command vai executar stored procedure
    cmd.CommandType = CommandType.StoredProcedure;
    // 2. CommandText vai receber apenas o nome da stored procedure
    cmd.CommandText = "STP_COPIA_PEDIDO_P";
    // 3. Com OleDbCommand, este tem que ser o primeiro
    // parâmetro a ser passado, antes mesmo do param de INPUT
```

```
        cmd.Parameters.Add("@RETURN_VALUE", OleDbType.Integer);
        cmd.Parameters["@RETURN_VALUE"].Direction = ParameterDirection.ReturnValue;
        // 3. Definir cada um dos parâmetros
        //    parâmetros de INPUT já podemos criar e passar o valor
        cmd.Parameters.AddWithValue("@NUM_PEDIDO_FONTE", updNumPed.Value);
        // parâmetros de OUTPUT somente terão valor após a execução
        cmd.Parameters.Add("@NUM_PEDIDO_NOVO", OleDbType.Integer);
        cmd.Parameters["@NUM_PEDIDO_NOVO"].Direction = ParameterDirection.Output;

        cmd.Parameters.Add("@MSG", OleDbType.VarChar, 1000);
        cmd.Parameters["@MSG"].Direction = ParameterDirection.Output;

        // executar a procedure
        conOle.Open();
        cmd.ExecuteNonQuery();
        conOle.Close();

        // ler os parâmetros de OUTPUT que agora já têm valor
        int numPed = (int)cmd.Parameters["@NUM_PEDIDO_NOVO"].Value;
        string msg = cmd.Parameters["@MSG"].Value.ToString();
        int ret = (int)cmd.Parameters["@RETURN_VALUE"].Value;

        if (ret < 0) lblResposta.Text = "Erro: " + msg;
        else lblResposta.Text = "Gerado pedido numero " + numPed;
    }
```

- O botão btnExecView vai executar a view VIE_TOTAL_VENDIDO. A forma de executar uma VIEW é com SELECT, então já conhecemos o procedimento, DataAdapter e DataTable.

```
    private void btnExecView_Click(object sender, EventArgs e)
    {
        // definir comando que será executado
        OleDbCommand cmd = conOle.CreateCommand();
        cmd.CommandText = @"SELECT * FROM VIE_TOTAL_VENDIDO
                            ORDER BY " + cmbOrdem.Text;
        // como é SELECT, criar DataAdapter para executar
        OleDbDataAdapter da = new OleDbDataAdapter(cmd);
        // criar DataTable para receber os dados do SELECT
        DataTable tb = new DataTable();
        // executar o SELECT e colocar os dados no DataTable
        da.Fill(tb);
        // mostrar os dados no grid
        dgvProc.DataSource = tb;
    }
```

Observe que como a view não pode receber parâmetro, esta não é uma boa solução para este caso, porque esta view sempre retorna com os dados de 2012.

- O botão btnExecProc executa a procedure STP_TOTAL_VENDIDO que devolve o total vendido em cada um dos meses do ano. Como esta procedure retorna com SELECT, a forma de executar é a mesma usada para comando SELECT.

```
private void btnExecProc_Click(object sender, EventArgs e)
{
    // definir comando que será executado
    OleDbCommand cmd = conOle.CreateCommand();
    cmd.CommandText = @"EXEC STP_TOTAL_VENDIDO " + updAno.Value;
    // como é SELECT, criar DataAdapter para executar
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    // criar DataTable para receber os dados do SELECT
    DataTable tb = new DataTable();
    // executar o SELECT e colocar os dados no DataTable
    da.Fill(tb);
    // mostrar os dados no grid
    dgvProc.DataSource = tb;
}
```

Esta solução é melhor que a anterior (VIEW) porque podemos passar o ano como parâmetro, mas como a procedure é executada com EXEC, e não SELECT, não podemos modificar o seu retorno, ele sempre será aquele definido pelo SELECT contido na procedure.

- O botão btnFuncTabular vai executar a função FN_TOTAL_VENDIDO. Funções tabulares são executadas com SELECT, então já conhecemos a solução.

```
private void btnFuncTabular_Click(object sender, EventArgs e)
{
    // definir comando que será executado
    OleDbCommand cmd = conOle.CreateCommand();
    cmd.CommandText = @"SELECT * FROM FN_TOTAL_VENDIDO( " +
                      updAno.Value  + " ) " + "ORDER BY " + cmbOrdem.Text;

    // como é SELECT, criar DataAdapter para executar
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    // criar DataTable para receber os dados do SELECT
    DataTable tb = new DataTable();
    // executar o SELECT e colocar os dados no DataTable
    da.Fill(tb);
    // mostrar os dados no grid
    dgvProc.DataSource = tb;
}
```

Observe que como a função tabular é executada com SELECT podemos alterar seu resultado incluindo ORDER BY, WHERE, JOIN etc...

Tópicos para revisão do capítulo

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo:

- ADO.NET é o conjunto de classes que tem como finalidade viabilizar o acesso a dados em um formato relacional orientado em tabela. No entanto, seu uso não se restringe apenas a fontes relacionais;

- A classe Command é utilizada para definir e executar uma instrução Sql;

- A classe DataAdapter facilita a execução de instrução SELECT;

- Os objetos das classes **Dataset** e DataTable podem ser usadas para receber o retorno de uma instrução SELECT;

- O método Fill() da classe DataAdapter executa uma instrução SELECT e coloca o seu resultado em um objeto DataTable ou DataSet.