

Olá,

Criei estas apostilas a mais de 5 anos e atualizei uma série delas com alguns dados adicionais. Muitas partes desta apostila está desatualizada, mas servirá para quem quer tirar uma dúvida ou aprender sobre .Net e as outras tecnologias.

Perfil Microsoft: <https://www.mcpvirtualbusinesscard.com/VBCServer/msincic/profile>

Marcelo Sincic trabalha com informática desde 1988. Durante anos trabalhou com desenvolvimento (iniciando com Dbase III e Clipper S'87) e com redes (Novell 2.0 e Lantastic).

Hoje atua como consultor e instrutor para diversos parceiros e clientes Microsoft.

Recebeu em abril de 2009 o prêmio **Latin American MCT Awards** no MCT Summit 2009, um prêmio entregue a apenas 5 instrutores de toda a América Latina (<http://www.marcelosincic.eti.br/Blog/post/Microsoft-MCT-Awards-America-Latina.aspx>).

Recebeu em setembro de 2009 o prêmio **IT HERO** da equipe Microsoft Technet Brasil em reconhecimento a projeto desenvolvido (<http://www.marcelosincic.eti.br/Blog/post/IT-Hero-Microsoft-TechNet.aspx>). Em Novembro de 2009 recebeu novamente um premio do programa IT Hero agora na categoria de especialistas (<http://www.marcelosincic.eti.br/Blog/post/TechNet-IT-Hero-Especialista-Selecionado-o-nosso-projeto-de-OCS-2007.aspx>).

Acumula por 5 vezes certificações com o título **Charter Member**, indicando estar entre os primeiros do mundo a se certificarem profissionalmente em Windows 2008 e Windows 7.

Possui diversas certificações oficiais de TI:

- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2008
- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2005
- MCITP - Microsoft Certified IT Professional Windows Server 2008 Admin
- MCITP - Microsoft Certified IT Professional Enterprise Administrator Windows 7 Charter Member
- MCITP - Microsoft Certified IT Professional Enterprise Support Technical
- MCPD - Microsoft Certified Professional Developer: Web Applications
- MCTS - Microsoft Certified Technology Specialist: Windows 7 Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows Mobile 6. Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Active Directory Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Networking Charter Member
- MCTS - Microsoft Certified Technology Specialist: System Center Configuration Manager
- MCTS - Microsoft Certified Technology Specialist: System Center Operations Manager
- MCTS - Microsoft Certified Technology Specialist: Exchange 2007
- MCTS - Microsoft Certified Technology Specialist: Windows Sharepoint Services 3.0
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2008
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 3.5, ASP.NET Applications
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2005
- MCTS - Microsoft Certified Technology Specialist: Windows Vista
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 2.0
- MCDBA – Microsoft Certified Database Administrator (SQL Server 2000/OLAP/BI)
- MCAD – Microsoft Certified Application Developer .NET
- MCSA 2000 – Microsoft Certified System Administrator Windows 2000
- MCSA 2003 – Microsoft Certified System Administrator Windows 2003
- Microsoft Small and Medium Business Specialist
- MCP – Visual Basic e ASP
- MCT – Microsoft Certified Trainer
- SUN Java Trainer – Java Core Trainer Approved
- IBM Certified System Administrator – Lotus Domino 6.0/6.5

1	Visão Geral do ADO.NET	3
1.1	Características do ADO.NET	3
1.2	Objetos do ADO.NET	3
2	Conexão com Bancos de Dados	4
2.1	Utilizando o Modo Gráfico	4
2.2	Utilizando o Modo Programático	5
2.3	Utilizando Transações	5
2.3.1	TransactionScope	6
3	Executando Comandos	7
3.1	Utilizando o Modo Gráfico	7
3.2	Utilizando o Modo Programático	7
3.3	Utilizando Transações	8
3.4	Executando Stored Procedure com Parâmetros	9
4	Modelo Conectado	10
4.1	Retornando Dados	10
4.2	Atualizando dados	11
5	Modelo Desconectado	12
5.1	Utilizando o Modo Gráfico	13
5.2	Criação e Manipulação de DataSet Manual	20
5.2.1	Criação de Tabelas	20
5.2.2	Atualizações em Linhas	20
5.2.3	Criação de Relacionamentos	21
6	Utilizando Documentos XML	22
6.1	Gravando Documentos XML	22
6.2	Lendo Documentos XML	22

1 Visão Geral do ADO.NET

Tecnologias de acesso a dados sempre foram essenciais para o rápido desenvolvimento de soluções. Cada diferente ambiente de desenvolvimento utiliza um sistema para acesso a dados próprio, como o BDE do Delphi, o JET do Access, NetLibrary do Oracle e outros.

A biblioteca de acesso a dados da Microsoft evoluiu do JET com DAO em 1992, RDO em 1995, ADO em 1997 e finalmente o ADO.NET em 2001.

A grande vantagem do ADO (Active Data Objects) é sua facilidade em desenvolvimento para múltiplos tipos de banco de dados. Independente do uso de ODBC (genérico para DBMS) ou OLEDB (proprietário para cada banco) os métodos de desenvolvimento são únicos. Por exemplo, com o DAO e o RDO utilizávamos diferentes métodos para mover registros quando o banco de dados era MSSQL ou Oracle, enquanto com o ADO isto já era convertido para o padrão pelo próprio engine de dados.

1.1 Características do ADO.NET

O motivo da evolução do ADO para o ADO.NET ocorreu por necessidade de melhor integração com XML e um novo modo de acesso.

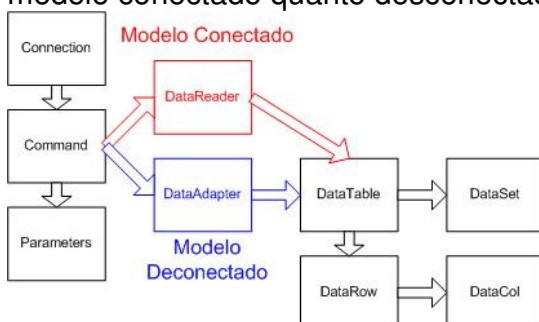
Este novo modo de acessos aos dados explora as capacidades de memória do cliente, pois até os modelos anteriores utilizamos sempre dados no servidor e consultas a todo o momento, linha por linha. Já no modelo ADO.NET os dados podem ser retornados completos ao cliente, não a tabela inteira e sim apenas a seleção, que armazena na memória no formato XML e atualiza o servidor ao final das operações com a tabela.

A estes modelos chamamos de conectados e desconectados, o ADO.NET fornece suporte aos dois modos de operação, sendo tratados individualmente nos módulos 4 e 5 desta apostila.

Quanto a utilização de XML no ADO.NET se torna muito simples, pois ele pode ler e gravar XML sem a necessidade de utilização do XMLDOM (Document Object Model), um completo conjunto de instruções que ligam com o XML a nível de elementos e não como dados relacionais.

1.2 Objetos do ADO.NET

No diagrama abaixo é possível conhecer os principais objetos que formam o ADO.NET, tanto no modelo conectado quanto desconectado:



- Os objetos `Connection` e `Command` funcionam tanto para o modelo conectado quanto o desconectado
- O `DataReader` é utilizado apenas no modelo conectado
- O `TableAdapter` é utilizado apenas no modelo desconectado
- O `DataSet` contém um conjunto de `DataTables`, e os `DataTables` podem ser tanto conectados quanto desconectados

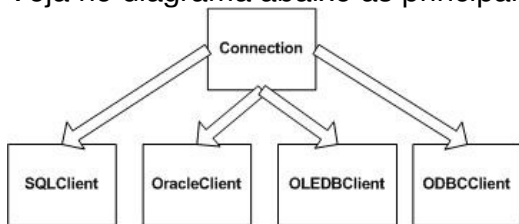
2 Conexão com Bancos de Dados

O primeiro e essencial objeto a ser utilizado no ADO.NET é o *connection*, responsável pelo acesso físico ao banco de dados a ser consultado.

Para utiliza o *connection* é necessário entender que não existe apenas uma classe de acesso a dados no ADO.NET como era no ADO anterior.

No ADO.NET temos quatro principais bibliotecas de acesso a dados e cada uma delas com seus objetos proprietários. A vantagem da utilização de objetos específicos para cada tipo de banco é a performance por não exigir compatibilidade com outro fabricante. Por outro lado, trabalhar no modelo específico tem a desvantagem da vinculação ao produto.

Veja no diagrama abaixo as principais classes de acesso a dados no ADO.NET:



- SQLClient – Microsoft SQL Server 7 e 2000
- OracleClient – Oracle 8.0 ou superior
- OLEDBClient – Access, Excel, flat-file, DB2, Sybase e outros que trabalhem com o padrão OLEDB
- ODBCClient – Informix, Progress, Paradox e outros compatíveis com ODBC

2.1 Utilizando o Modo Gráfico

O VS 2008 possui os objetos *connection* prontos para serem utilizados com drag-drop.

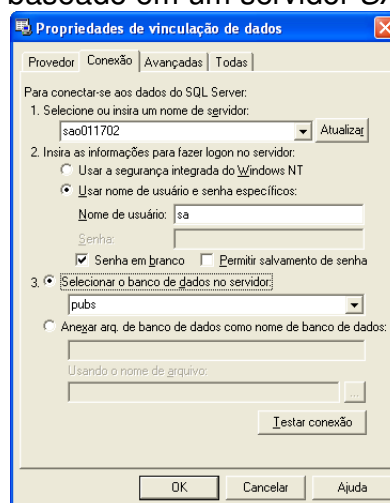
Vá para a barra de ferramentas e na aba *Data* encontrará os diferentes objetos para cada diferente classe de dados, bem como o resultado na página ao arrastarmos um objeto conexão para SQL Server no formulário.

Logo após colocar a conexão no formulário será necessário configura-lo, e para isto tenha em mãos os seguintes dados:

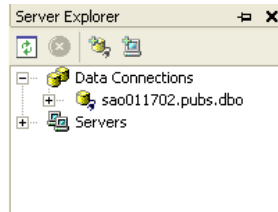
1. Nome do servidor que será consultado
2. Nome e senha do usuário com acesso ao banco de dados
3. Banco de dados a ser acessado

Alem dos dados obrigatórios também podemos configurar tempo de timeout, código de página e outras configurações avançadas.

Veja abaixo as tela de configuração baseado em um servidor *SAO011702*:



Para chegar a tela de configuração, clique com o botão direito no objeto *sqlconnection1* e escolha propriedades. Na janela de propriedades escolha *ConnectionString* e depois *New Connection*. Isso só será necessário a primeira vez, pois nas subseqüentes pode-se utilizar as conexões já definidas que ficam visíveis no “Server Explorer”, como abaixo:



Ao arrastar uma tabela a partir desta janela automaticamente será criado um objeto de acesso a dados e também um grid, facilitando manipulação de dados.

2.2 Utilizando o Modo Programático

Para fazer o uso do objeto *connection* sem o modo gráfico precisamos instanciar a classe e definir a sua principal propriedade, a *connectionstring* que define todos os dados a respeito da conexão, como o exemplo abaixo:

```
try
{
    System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
    Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
    Conexao.Open();
    MessageBox.Show("Conexão aberta com sucesso");
}
catch(System.Data.SqlClient.SqlException Erro)
{
    MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
}
```

Quando em modo programático definimos na *conectionstring* os mesmos dados definidos anteriormente no modo gráfico, mas para acessar recursos da conexão, seja ela criada no modo gráfico ou programático, precisamos conhecer alguns de seus principais métodos e propriedades listadas abaixo:

Propriedade ou Método	Funcionalidade
P-ConnectionTimeout	Permite definir em segundos o tempo de resposta para a conexão incorrer em erro, o padrão são 30 segundos.
P-ServerVersion	Retorna a versão do RDBMS que está sendo consultado.
P-State	Permite saber se a conexão está aberta, fechada, abrindo ou fechando.
M-BeginTransaction	Abre uma transação no servidor para esta conexão.
M-Close	Libera fisicamente a conexão com o servidor.
M-Open	Ativa a conexão fisicamente com o servidor.

2.3 Utilizando Transações

Para utilizarmos transações no ADO.NET precisamos primeiro abrir uma transação e com o método *BeginTransaction* criar um objeto para referenciar a transação, chamado *transaction*.

No exemplo a seguir adicionamos a transação ao objeto criado anteriormente:

```
try
{
    System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
    Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
    Conexao.Open();
    MessageBox.Show("Conexão aberta com sucesso");
}
```

```
        System.Data.SqlClient.SqlTransaction Transacao;  
        Transacao = Conexao.BeginTransaction();  
        Transacao.Commit();           //Grava a transação  
        Transacao.Rollback();        //Descarta a transação  
    }  
    catch(System.Data.SqlClient.SqlException Erro)  
    {  
        MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);  
    }  
}
```

Um mesmo objeto *connection* pode conter diversas transações vinculadas a ele, sendo cada uma das transações independentes uma das outras.

2.3.1 TransactionScope

Um novo modo de lidar com transações agora é utilizando o objeto *TransactScope* que cria uma transação hierárquica, podendo juntar diversas transações, conexões com fácil aninhamento e controle de processo. Veja o exemplo abaixo:

```
using (Transacao as new  
    TransactionScope(TransactionScopeOption.RequiresNew))  
{  
    using (Conexao as new SqlConnection(sConexao))  
    {  
        Conexao.Open()  
        SqlCommand Comando = new SqlCommand(sSQL, Conexao)  
        Comando.ExecuteNonQuery()  
        Transacao.Complete  
    }  
}
```

A transação fica dentro de um escopo de comandos que deverão estar sincronizados. A vantagem do *TransactionScope* sobre o *Transaction* é o fato do objeto não precisar ser criado e vinculado a conexão, o que gera problemas quando nem todos os comandos precisam ser transacionados. Além disso, o *TransactionScope* pode incluir as mensagens recebidas e enviadas pelo MessageQueue (MSMQ).

3 Executando Comandos

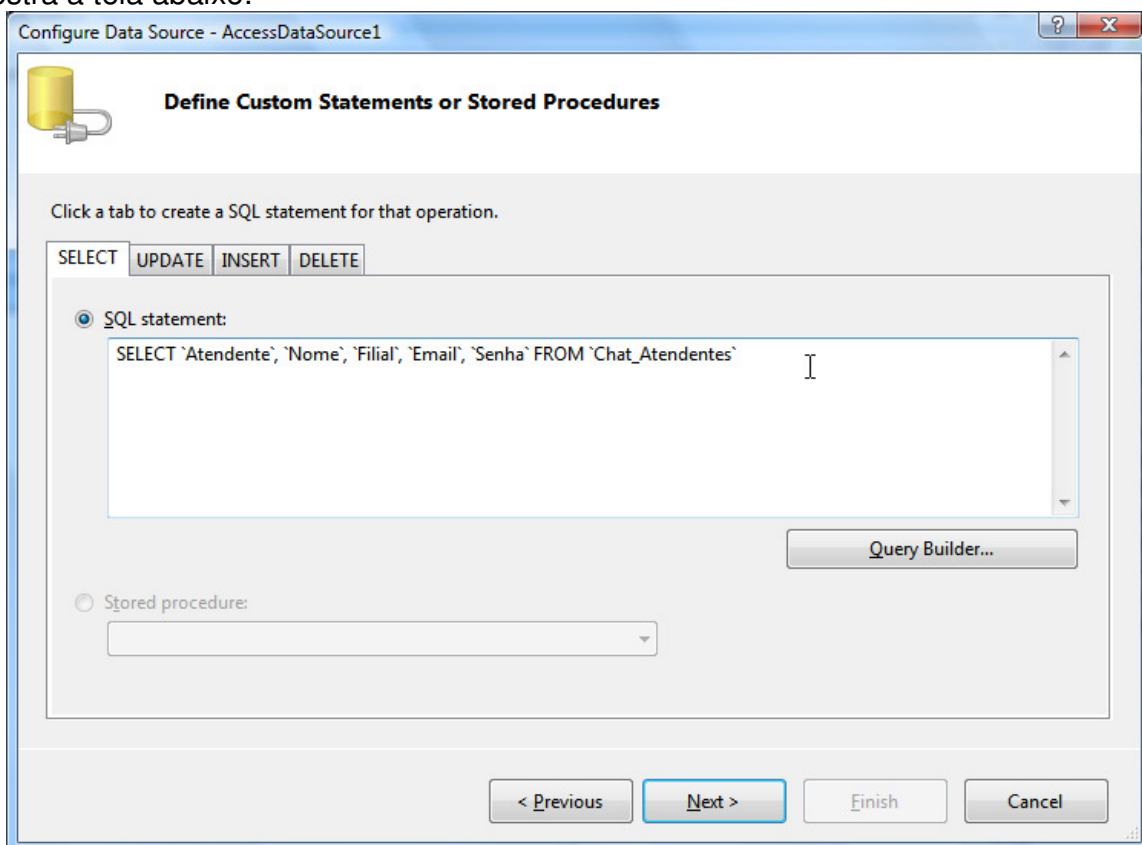
O segundo objeto na hierarquia do ADO.NET é o *command*, utilizado para executar comandos no banco de dados.

O *command* é obrigatório como ponte entre a interface com o usuário e o objeto *connection*, sendo o único com capacidade de executar DML, DCL ou DDL.

Como o objeto *command* precisa se comunicar com o banco de dados, antes de utilizá-lo estabelecemos a conexão e informamos ao *command* a conexão que está ativa.

3.1 Utilizando o Modo Gráfico

Ao arrastar um objeto a partir do “Server Explorer” do tipo “Stored Procedure” ou mesmo uma tabela ou outro controle, o comando pode ser configurado nas propriedades do “Data Source” utilizado, como mostra a tela abaixo:



3.2 Utilizando o Modo Programático

Utilizar o modo programático do *command* não difere do modo gráfico na construção, mas consideraremos propriedades e métodos específicos.

O exemplo anterior da conexão agora com o comando de *update* nos dados pode ser escrito da seguinte forma:

```
try  
{
```

```

System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
Conexao.Open();
MessageBox.Show("Conexão aberta com sucesso");
System.Data.SqlClient.SqlCommand Comando = new System.Data.SqlClient.SqlCommand();
Comando.CommandText="Update Authors Set Au_Lname='Silva' Where Au_Fname='Michael'";
Comando.CommandType = System.Data.CommandType.Text;
Comando.Connection = Conexao;
int Afetados = Comando.ExecuteNonQuery();
MessageBox.Show("Comando afetou " + Afetados.ToString() + " linhas.");
}
catch(System.Data.SqlClient.SqlException Erro)
{
    MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
}

```

Ao executar um objeto comando podemos fazê-lo de três diferentes formas:

Método	Funcionalidade	Retorno
ExecuteScalar	Utilizado para funções de agregação, retorna apenas uma coluna com os dados agregados.	Object com um item
ExecuteNonQuery	Envia o comando e retorna quantos registros o comando afetou, muito utilizado para <i>insert</i> , <i>update</i> e <i>delete</i> , DCL e DDL.	Um número inteiro
ExecuteReader	Executa um <i>select</i> e alimenta um objeto <i>datareader</i> , considerado no próximo módulo.	Objeto <i>DataReader</i>

No exemplo de código acima foi utilizado o tipo *ExecuteNonQuery* pois um update apenas nos interessa o número de linhas alteradas, onde caso retornasse zero saberíamos que não foi encontrado o registro desejado, e então trataríamos conforme especificação.

3.3 Utilizando Transações

No módulo anterior vimos como abrir uma transação, mas para esta transação ser ou não utilizada no comando, precisamos definir explicitamente.

Isto pode ser feito por definir no objeto *command* a propriedade *transaction* o nome do objeto transação criado anteriormente.

O exemplo abaixo atualizado com transação demonstra como utilizar:

```

try
{
    System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
    Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
    Conexao.Open();
    System.Data.SqlClient.SqlTransaction Transacao;
    Transacao = Conexao.BeginTransaction();
    MessageBox.Show("Conexão aberta com sucesso");
    System.Data.SqlClient.SqlCommand Comando = new System.Data.SqlClient.SqlCommand();
    Comando.CommandText="Update Authors Set Au_Lname='Silva' Where Au_Fname='Michael'";
    Comando.CommandType = System.Data.CommandType.Text;
    Comando.Connection = Conexao;
    Comando.Transaction = Transacao;
    int Afetados = Comando.ExecuteNonQuery();
    MessageBox.Show("Comando afetou " + Afetados.ToString() + " linhas.");
    Transacao.Rollback();
}
catch(System.Data.SqlClient.SqlException Erro)
{
    MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
}

```

3.4 Executando Stored Procedure com Parâmetros

Algumas SP contêm parâmetros para informação de dados de entrada e parâmetros de saída de informação. Nestes casos não basta apenas colocar no nome do comando a stored procedure, pois precisamos conseguir resgatar o retorno.

Neste caso utilizamos uma coleção de objetos específicos de nome *parameters*.

Abaixo está o exemplo de uma SP criada que devemos informar o código do autor e a SP nos retorna o nome e a cidade do autor desejado:

```
--SP QUE RECEBE O CODIGO E RETORNA O NOME E A CIDADE
CREATE PROCEDURE DADOSAUTOR
  (@CODIGO CHAR(11),
   @NOME CHAR(60) OUTPUT,
   @CIDADE CHAR(20) OUTPUT)
AS
SELECT @NOME = AU_FNAME+' '+AU_LNAME,
       @CIDADE = CITY
FROM AUTHORS
WHERE AU_ID = @CODIGO

--TESTA A STORED PROCEDUDRE
DECLARE @NOME CHAR(60), @CIDADE CHAR(20)
EXECUTE DADOSAUTOR '427-17-2319', @NOME OUT, @CIDADE OUT
PRINT @NOME
PRINT @CIDADE
```

Para conseguir executar a SP e retornar os valores devidos para as variáveis *@Nome* e *@Cidade* o seguinte código deve ser utilizado:

```
try
{
    System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
    Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
    Conexao.Open();
    System.Data.SqlClient.SqlCommand Comando = new System.Data.SqlClient.SqlCommand();
    Comando.CommandText="DADOSAUTOR";
    Comando.CommandType = System.Data.CommandType.StoredProcedure;
    Comando.Connection = Conexao;
    Comando.Parameters.Add("@Codigo", "427-17-2319");
    Comando.Parameters.Add("@Nome", System.Data.SqlDbType.Char, 60);
    Comando.Parameters["@Nome"].Direction=System.Data.ParameterDirection.Output;
    Comando.Parameters.Add("@Cidade", System.Data.SqlDbType.Char, 20);
    Comando.Parameters["@Cidade"].Direction=System.Data.ParameterDirection.Output;
    int Afetados = Comando.ExecuteNonQuery();
    MessageBox.Show(Comando.Parameters["@Nome"].Value.ToString());
    MessageBox.Show(Comando.Parameters["@Cidade"].Value.ToString());
}
catch(System.Data.SqlClient.SqlException Erro)
{
    MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
}
```

Note que o parâmetro para código não foi necessário definir o tipo de dados nem o tamanho da string, uma vez que é um parâmetro de entrada.

Nos parâmetros de saída é obrigatório definir além do nome o tipo e tamanho, bem como a direção, ou seja, *output*.

Para ler o retorno das variáveis após a execução da SP utilizamos o nome do objeto comando mais o nome do parâmetro e a propriedade *value* que deve ser transformada em string mesmo quando o retorno já é texto.

Quando utilizado o modo gráfico é muito mais simples ler os parâmetros, bastando clicar nas propriedades do objeto *command* e escolher a propriedade *parameters* e utilizando o botão *add* informar os parâmetros, tipos e direções.

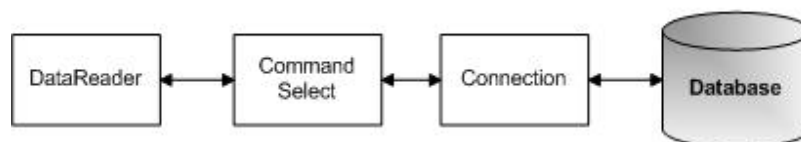
4 Modelo Conectado

O modelo conectado é mais rápido que o modelo desconectado, mas não oferece as vantagens do modo gráfico, apenas programático.

Outra desvantagem do modelo conectado é sua capacidade *forward-only* e *read-only*, ou seja, ele navega apenas para a frente, não permitindo alterações.

Por outro lado, a grande vantagem de um *datareader* é a sua performance que pode chegar a três vezes mais rápido que o *dataadapter* e também ganha-se com tráfego em rede, uma vez que ele retornar registros conforme a solicitação, enquanto o *dataadapter* traz toda a tabela de uma só vez para a memória do cliente.

Para criar um objeto *datareader* precisamos dos dois objetos obrigatórios que se comunicam com o DBMS, o objeto *command* e o objeto *connection*.



DICA: O modelo conectado na verdade gera um cursor estático no servidor e a cada novo readline utilizado no datareader é enviado um fetch ao servidor, que por sua vez retorna uma linha completa. Por isso o modelo é forward-only e read-only.

4.1 Retornando Dados

Retornar dados com o *datareader* é relativamente simples desde que tenha sempre em mente que as linhas retornam uma a uma. Lembre-se que não existe métodos para voltar.

Como o *datareader* é baseado em cursor no servidor, a cada registro que deseja retornar executa-se o método *read* que envia um *fetch* ao servidor de banco de dados e o *datareader* fica alimentado com a linha atual.

O código abaixo abre a conexão, executa o *command* e o resultado é alimenta o *datareader* que por meio de um loop retornar os registros e popula um *listbox*:

```
try
{
    System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
    Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
    Conexao.Open();
    System.Data.SqlClient.SqlCommand Comando = new System.Data.SqlClient.SqlCommand();
    Comando.CommandText="Select * From Authors";
    Comando.CommandType = System.Data.CommandType.Text;
    Comando.Connection = Conexao;
    System.Data.SqlClient.SqlDataReader Retorno = Comando.ExecuteReader();
    while(Retorno.Read())
    {
        listBox1.Items.Add(Retorno["Au_Lname"].ToString());
    }
}
catch(System.Data.SqlClient.SqlException Erro)
{
    MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
}
```

Note que para retornar o conteúdo de cada coluna utilizamos o nome do objeto *datareader* seguido do índice ou nome da coluna.

4.2 Atualizando dados

Em uma aplicação onde foi escolhido trabalhar com o *datareader* temos a limitação das atualizações. Para que aplicações baseadas em *datareader* possam alterar dados é necessário que programaticamente montemos uma instrução DML e a enviemos pelo objeto *command*. Abaixo o código demonstra como uma aplicação com dois *textbox* e um botão resolveriam este problema:

```
//No carregamento do formulario alimenta os textbox com o select filtrado para um autor
private void Form1_Load(object sender, System.EventArgs e)
{
    try
    {
        System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
        Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
        Conexao.Open();
        System.Data.SqlClient.SqlCommand Comando = new System.Data.SqlClient.SqlCommand();
        Comando.CommandText="Select * From Authors Where Au_ID=427-17-2319";
        Comando.CommandType = System.Data.CommandType.Text;
        Comando.Connection = Conexao;
        System.Data.SqlClient.SqlDataReader Retorno = Comando.ExecuteReader();
        while(Retorno.Read())
        {
            textBox1.Text = Retorno["Au_Lname"].ToString();
            textBox2.Text = Retorno["Au_Fname"].ToString();
        }
    }
    catch(System.Data.SqlClient.SqlException Erro)
    {
        MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
    }
}

//Botão de gravação. Monta uma string com o DML de update e envia ao servidor
//Utiliza o retorno do command para verificar se houve ou não a alteração
private void btnGravar_Click(object sender, System.EventArgs e)
{
    try
    {
        System.Data.SqlClient.SqlConnection Conexao = new System.Data.SqlClient.SqlConnection();
        Conexao.ConnectionString="Data Source=SAO011702; User ID=sa; Password=; Initial Catalog=Pubs";
        Conexao.Open();
        string Atualiza = "Update Authors Set Au_Lname=" + textBox1.Text + ", Au_Fname=" + textBox2.Text + "
        Where Au_ID='427-17-2319'";
        System.Data.SqlClient.SqlCommand Comando = new System.Data.SqlClient.SqlCommand();
        Comando.CommandText = Atualiza;
        Comando.CommandType = System.Data.CommandType.Text;
        Comando.Connection = Conexao;
        int Retorno = Comando.ExecuteNonQuery();
        if(Retorno != 1)
            MessageBox.Show("Autor não foi alterado..");
        else
            MessageBox.Show("Autor alterado com sucesso." + (char)13 + Atualiza);
    }
    catch(System.Data.SqlClient.SqlException Erro)
    {
        MessageBox.Show("Ocorreu um erro. Mensagem: " + Erro.Message);
    }
}
```

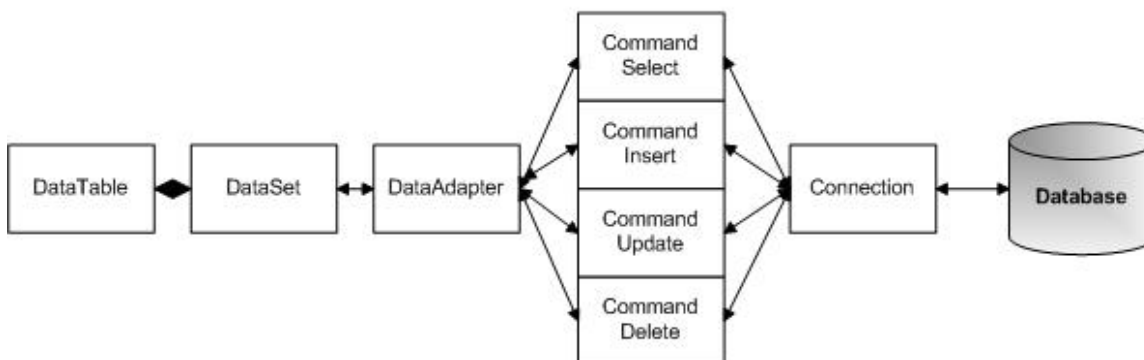
No carregamento do formulário foi alimentado os *textbox* com o nome e sobrenome do autor, e no botão gravar criamos manualmente um comando de *update* para atualizar o autor que foi carregado.

5 Modelo Desconectado

O modelo de dados desconectado é útil por permitir trabalho online ou offline.

Por exemplo, podemos alimentar a memória do cliente, que pode ser um notebook ou handheld, que trabalhará normalmente fazendo atualizações, inserções e exclusões todas nestes dados em memória, podendo gravar em disco no formato XML. Ao se reconectar a um servidor os dados e alterações são reenviados a base de dados que atualiza os dados fisicamente.

Para trabalhar com dados desconectados utilizamos um objeto de nome *TableAdapter* que interage com o banco de dados fisicamente e um objeto de nome *DataSet* que armazena e manipula dados na memória. Ou seja, com o *TableAdapter* retornamos os dados, enquanto o *dataset* é o objeto memória para fazer o cache local.



A tabela abaixo compara o *datareader* com o *dataadapter*.

Característica	DataReader	TableAdapter
Armazenar dados na memória local do cliente		✓
Navegação livre nas linha da tabela		✓
Alta performance	✓	
Manipulação de XML		✓
Menor utilização de rede	✓	
Criação de objetos graficamente		✓
Vinculação direta nos controle databound		✓

Uma grande diferença entre o *datareader* e o *TableAdapter* é que o *adapter* não é um objeto de acesso a dados, mas sim um repositório, como pode ser visto na figura anterior.

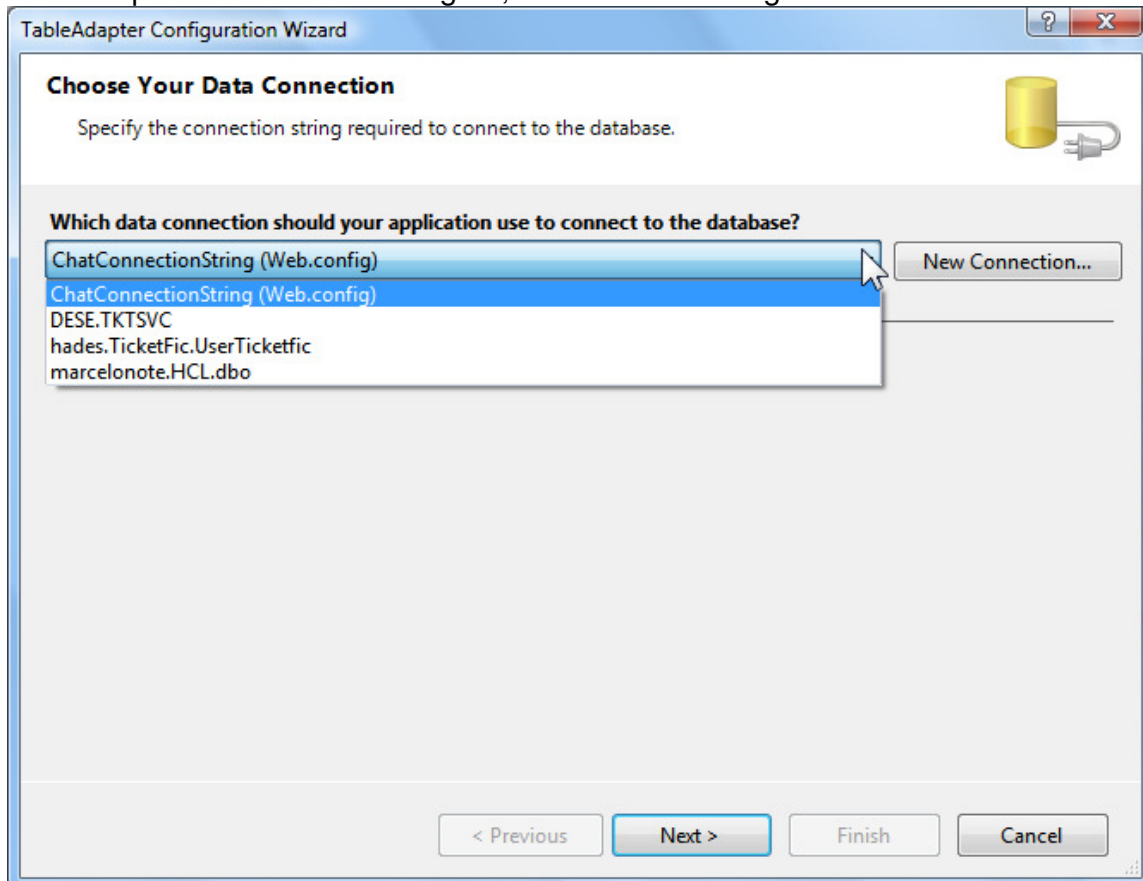
Dentro de um *TableAdapter* existem vários objetos para seleção e manipulação de dados, chamados de *Query*. Cada um destes objetos é configurado utilizando uma coleção de parâmetros e tem o comando específico para a função que desempenha.

O funcionamento de um *adapter* é preencher os dados em memória utilizando por padrão o método *fill*, com estes dados em memória, no *dataset*, o usuário trabalha normalmente e no final de uma operação invocamos o método criado para *update* do *dataset* que por sua vez verifica quais as tabelas sofreram alterações, e para cada tabela existe um *adapter* com o comando específico para a operação que deve ser feita, no caso, um *update*, *insert* ou *delete*.

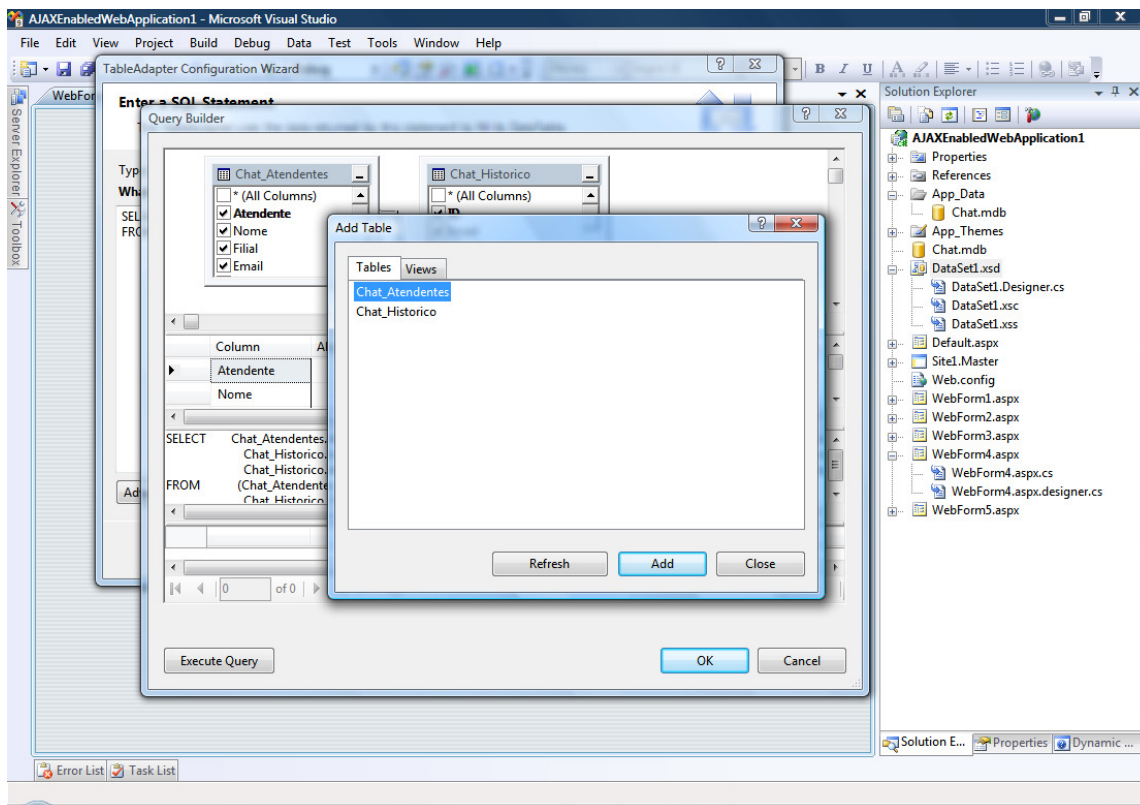
5.1 Utilizando o Modo Gráfico

Ao inserir no projeto um item do tipo “Dataset” podemos utilizar a interface gráfica anterior do ASP.NET, porem com uma grande mudança nas diferentes *query* que podem ser geradas, uma vez que no modelo anterior apenas uma *query* era possível.

Para incluir uma tabela arraste o objeto “TableAdapter” da caixa de ferramentas e automaticamente o *wizard* pedirá os dados da origem, como na tela a seguir:



Logo após definir a origem dos dados será possível escolher se iremos utilizar uma *query Select* ou uma *Stored procedure*. No caso de *SQL* o VS2008 abre um *wizard* como o a seguir com a possibilidade da construção gráfica da *query*:



É na tela a seguir que notamos a principal mudança no *Dataset*, quando podemos escolher o nome do comando criado. Isso é importante pois podemos criar diferentes comandos, cada um com um filtro diferente e escolher o filtro por colocar o sinal "?" em uma cláusula *where* da consulta.

TableAdapter Configuration Wizard

Enter a SQL Statement

The TableAdapter uses the data returned by this statement to fill its DataTable.

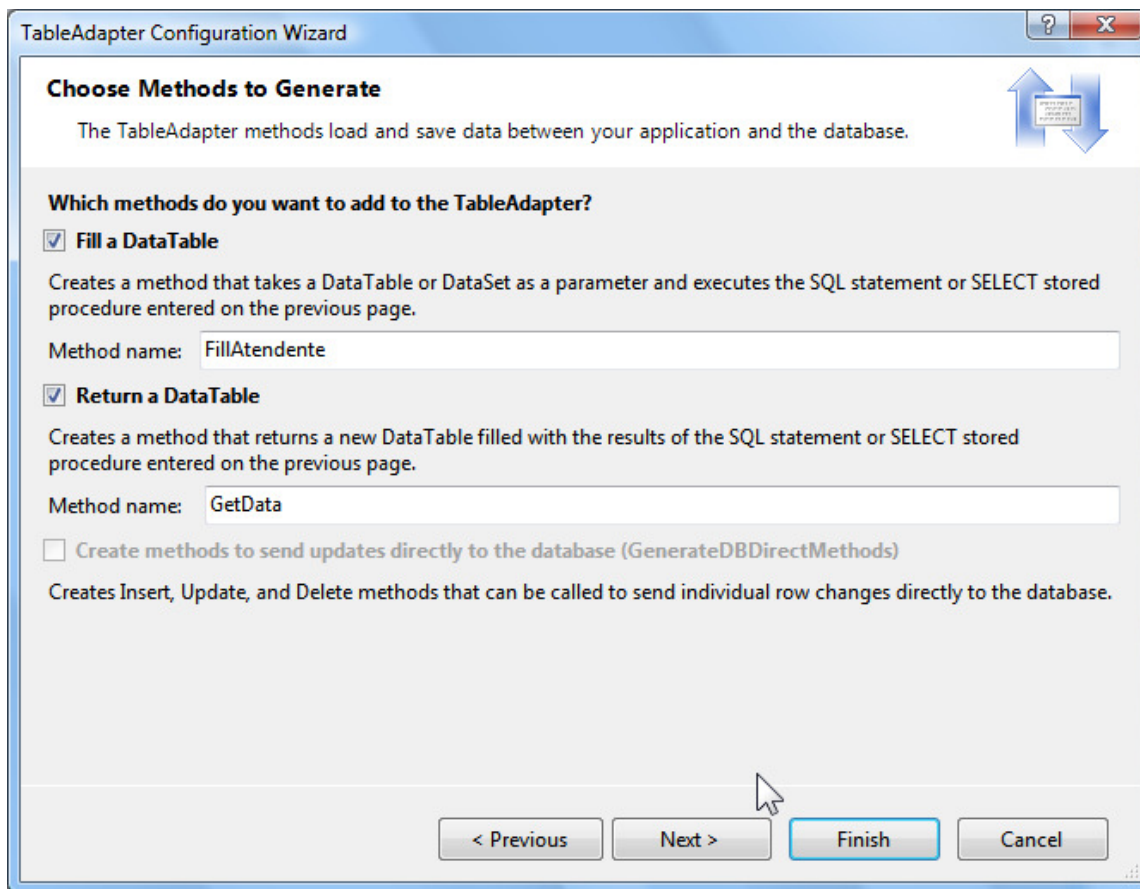
Type your SQL statement or use the Query Builder to construct it. What data should be loaded into the table?

What data should be loaded into the table?

```
SELECT Chat_Atendentes.Atendente, Chat_Atendentes.Nome, Chat_Atendentes.Filial, Chat_Atendentes.Email,
Chat_Atendentes.Senha, Chat_Historico.ID,
Chat_Historico.Email AS Expr1, Chat_Historico.HoraEntrada, Chat_Historico.HoraAtendimento,
Chat_Historico.HoraSaida, Chat_Historico.Conversacao,
Chat_Historico.Atendente AS Expr2, Chat_Historico.Filial AS Expr3, Chat_Historico.Nome AS Expr4,
Chat_Historico.Telefone
FROM (Chat_Atendentes INNER JOIN
Chat_Historico ON Chat_Atendentes.Atendente = Chat_Historico.Atendente)
WHERE Chat_Atendentes.Atendente = ?
```

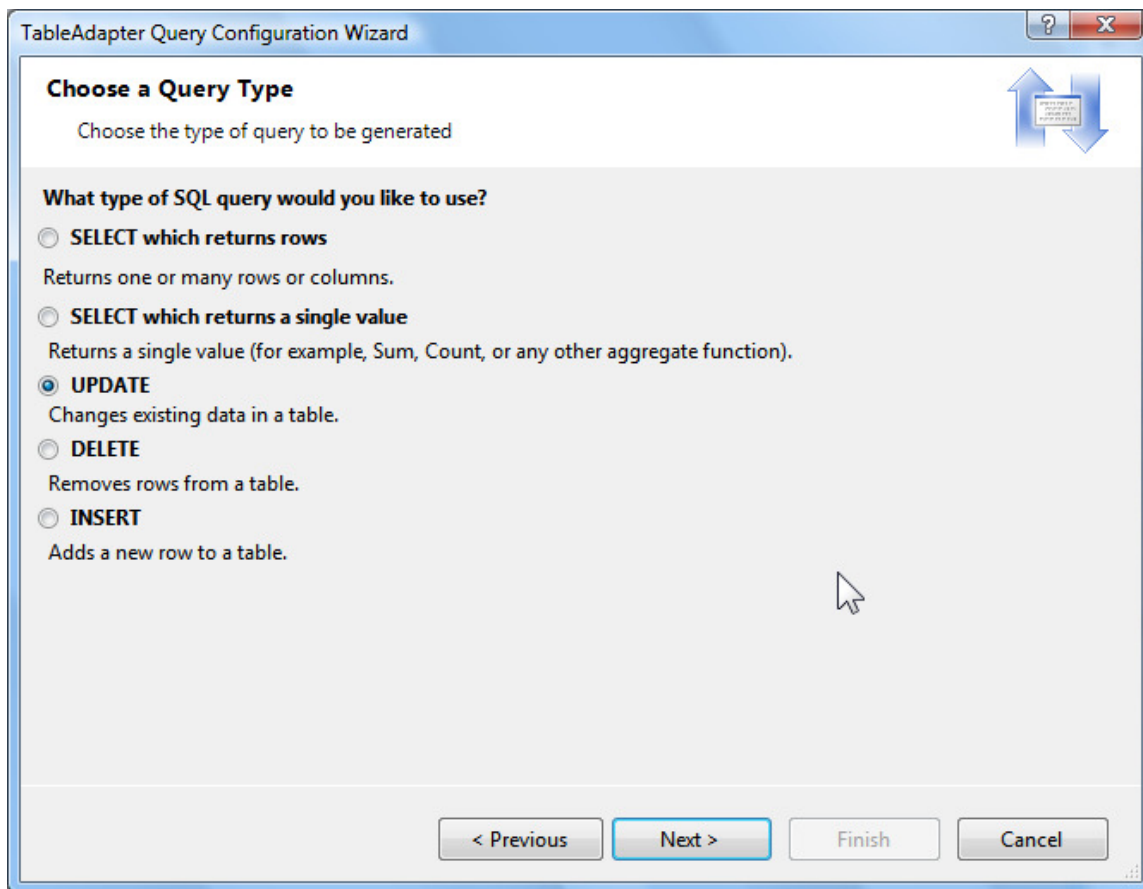
Advanced Options... Query Builder...

< Previous Next > Finish Cancel

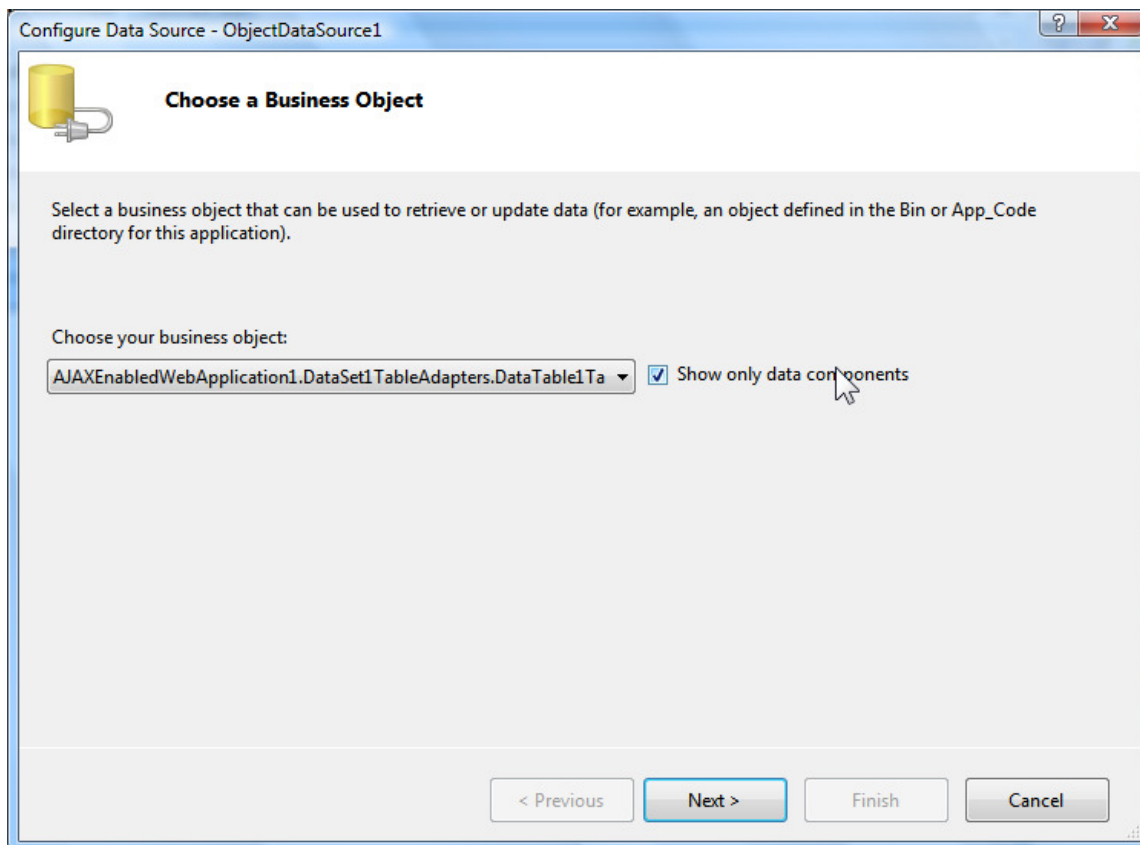


Com este modelo podemos criar diversas *queries*, cada uma com um parâmetro diferente, o que facilita em muito o preenchimento de *grids* e outros objetos sem a necessidade da recriação do comando inteiro.

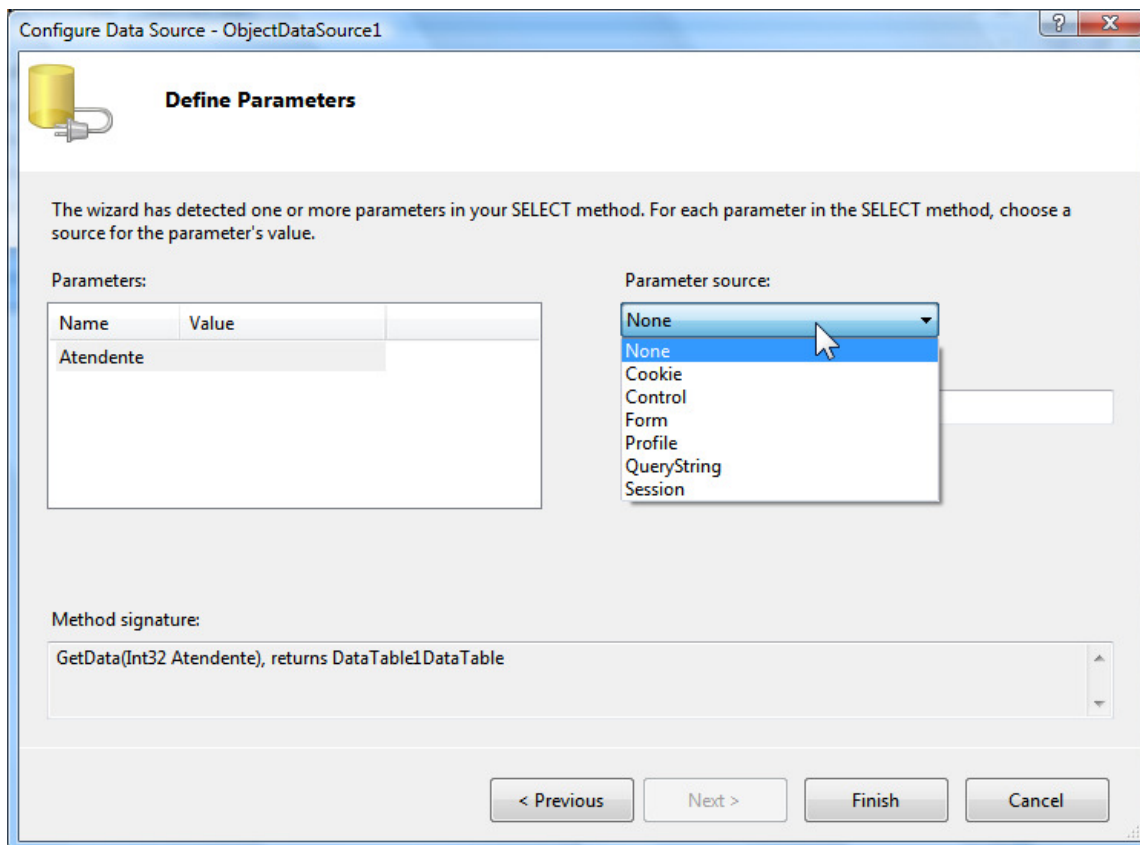
Para inserir as *queries* de manipulação de dados clique no objeto “TableAdapter” e utilize a opção “Add Query”, escolhendo primeiro o tipo de comando utilizado e na sequencia qual dos comandos SQL será gerado:



O *wizard* criará automaticamente todos os parâmetros para que o comando seja utilizado. Para utilizar agora em um formulário arraste o objeto "ObjectDataSource" e escolha o *TableAdapter* criado anteriormente, como a tela a seguir:



Na tela seguinte escolha o comando que será utilizado (um dos *GetData* anteriormente criados) e defina de onde o parâmetro será carregado, o que pode incluir um textbox do formulário, uma variável de memória ou manual como a seguir:



Neste exemplo, para utilizar programaticamente coloque o objeto "DataGrid" sem alterar qualquer propriedade nele e dinamicamente iremos ligá-lo ao objeto de dados:

```
protected void Page_Load(object sender, EventArgs e)
{
    ObjectDataSource1.SelectParameters[0].DefaultValue = "1";
    GridView1.DataSourceID = "ObjectDataSource1";
}
```

Com estes exemplos vemos a facilidade de utilizar e criar *Datasets* em aplicações de forma gráfica.

5.2 Criação e Manipulação de DataSet Manual

Em certas ocasiões precisamos criar os *dataset* manualmente, como por exemplo, uma aplicação desconectada.

Como exemplo podemos imaginar uma aplicação que irá rodar em um *handheld* na mão de um vendedor autônomo. As tabelas e o *dataset* irão estar na memória do cliente, sendo manipulados localmente.

5.2.1 Criação de Tabelas

Ao criar um *dataset* manualmente precisamos fazer toda a manipulação das tabelas e colunas, como o código abaixo gera:

```
private DataSet dsVendas = new DataSet("Vendas");
private void Form1_Load(object sender, System.EventArgs e)
{
    //Tabela de Clientes
    DataTable Clientes = new DataTable("Clientes");
    Clientes.Columns.Add("Nome", typeof(string));
    Clientes.Columns.Add("Endereco", typeof(string));
    Clientes.Columns.Add("Cidade", typeof(string));
    Clientes.Columns.Add("Estado", typeof(string));
    //Tabela de Vendas
    DataTable Vendas = new DataTable("Vendas");
    Vendas.Columns.Add("Nome", typeof(string));
    Vendas.Columns.Add("Produto", typeof(string));
    Vendas.Columns.Add("Quantidade", typeof(int));
    Vendas.Columns.Add("Preco", typeof(decimal));
    //Coloca as tabelas no Dataset
    dsVendas.Tables.Add(Clientes);
    dsVendas.Tables.Add(Vendas);
    //Faz o bind no datagrid e nos textbox
    dataGrid1.DataSource = Clientes;
    textBox1.DataBindings.Add("Text", Clientes, "Nome");
    textBox2.DataBindings.Add("Text", Clientes, "Cidade");
}
```

Note que a criação do *dataset* foi feita fora do método pois ele precisa ser lido em outras partes do código.

Veja que as tabelas foram criadas sem qualquer configuração e na seqüência foram definidos e criadas as colunas necessárias, com o nome da coluna e o tipo de dados.

Após a criação e definição da tabela vemos que ela precisou ser inserida no *dataset*, para depois podermos utilizar os *bind* no *grid* e nos *textbox*.

5.2.2 Atualizações em Linhas

Para alterar ou excluir linhas de um *dataset* utilizamos o número da linha.

Por exemplo, para excluir a linha 3 utilizamos:

```
dsVendas.Tables["Clientes"].Rows[3].Delete();
```

Para alterar a linha 3 da tabela seguimos um modelo similar:

```
dsVendas.Tables["Clientes"].Rows[3].BeginEdit();
dsVendas.Tables["Clientes"].Rows[3]["Nome"]="João";
dsVendas.Tables["Clientes"].Rows[3]["Cidade"]="Osasco";
dsVendas.Tables["Clientes"].Rows[3].EndEdit();
```

Na exclusão não é necessário abrir e fechar, mas no caso da alteração é recomendado abrir com o *beginedit* e fechar com o *endedit*.

No caso da inserção de novas linha é um pouco diferente, principalmente porque criamos um *array* do tipo *object* e inserimos este array dentro da tabela, como o código exemplificando a seguir:

```
object[] NovoReg = new object[4];  
NovoReg[0]="Joao";  
NovoReg[1]="Rua dos Abrolhos, 50";  
NovoReg[2]="Osasco";  
NovoReg[3]="SP";  
dsVendas.Tables["Clientes"].Rows.Add(NovoReg);
```

O modo para inserção é utilizar um *array* posicional e não nomeado.

5.2.3 Criação de Relacionamentos

Os relacionamentos entre as tabelas de um *dataset* são importantes para garantir a integridade dos dados, bem como criar um *datagrid* hierárquico onde os dados da venda de um cliente aparecem abaixo dos dados do cliente.

Para o nosso exemplo podemos relacionar a tabela de clientes com a tabela de vendas utilizando o nome do cliente, com o código a seguir:

```
dsVendas.Relations.Add("Clientes_Vendas", Clientes.Columns["Nome"], Vendas.Columns["Nome"], true);
```

Veja na figura abaixo como ficou o *datagrid* com este relacionamento ativado:

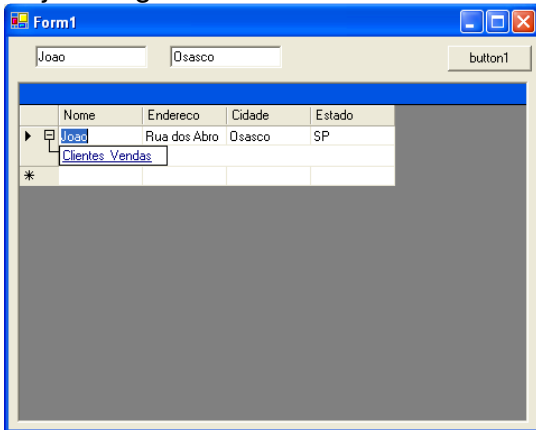


Figura 1 - Relacionamento ativo

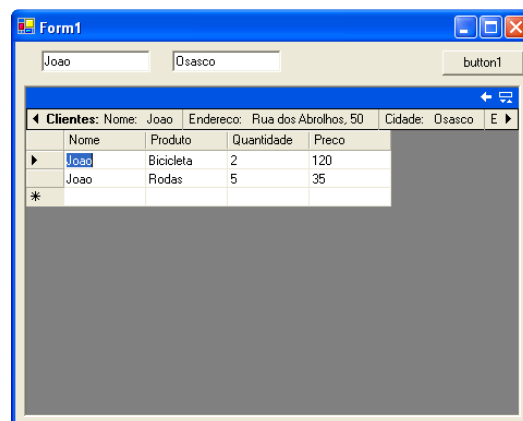


Figura 2 - Ao clicar no link de vendas

6 Utilizando Documentos XML

O ADO.NET surgiu como evolução do anterior ADO integrado com XML. Com isto, ler e utilizar XML no ADO.NET foi simplificado ao máximo.

6.1 Gravando Documentos XML

Utilizando como base o *dataset* criado no módulo anterior, para gravar um XML apenas precisamos utilizar o código abaixo:

```
dsVendas.WriteXml(AppDomain.CurrentDomain.BaseDirectory + @"\Vendas.xml");
```

O método *writexml* apenas precisa do caminho de rede onde o XML será salvo, e o arquivo gerado consta abaixo:

```
<?xml version="1.0" standalone="yes" ?>
- <Vendas>
- <Clientes>
  <Nome>Joao</Nome>
  <Endereco>Rua dos Abrolhos, 50</Endereco>
  <Cidade>Osasco</Cidade>
  <Estado>SP</Estado>
</Clientes>
- <Clientes>
  <Nome>Maria</Nome>
  <Endereco>Rua das Hortencias, 30</Endereco>
  <Cidade>Curitiba</Cidade>
  <Estado>PR</Estado>
</Clientes>
- <Vendas>
  <Nome>Joao</Nome>
  <Produto>Bicicleta</Produto>
  <Quantidade>2</Quantidade>
  <Preco>120</Preco>
</Vendas>
- <Vendas>
  <Nome>Joao</Nome>
  <Produto>Rodas</Produto>
  <Quantidade>5</Quantidade>
  <Preco>35</Preco>
</Vendas>
</Vendas>
```

6.2 Lendo Documentos XML

Para ler um XML é tão simples quanto a operação de gravação, bastando o seguinte código listado abaixo:

```
dsVendas.ReadXml(AppDomain.CurrentDomain.BaseDirectory + @"\Vendas.xml");
//Faz o bind no datagrid e nos textbox
dataGridView1.DataSource = dsVendas.Tables["Clientes"];
textBox1.DataBindings.Add("Text", dsVendas.Tables["Clientes"], "Nome");
textBox2.DataBindings.Add("Text", dsVendas.Tables["Clientes"], "Cidade");
dsVendas.Relations.Add("Clientes_Vendas", dsVendas.Tables["Clientes"].Columns["Nome"],
dsVendas.Tables["Vendas"].Columns["Nome"], true);
```

Não foi necessário criar estrutura nem qualquer outro objeto para ter os dados disponíveis, bastando apenas após a leitura do XML fazer o *bind* e recriar os relacionamentos, uma vez que esta opção não é incluída no XML.