Utilizando os componentes da paleta DBExpress para acesso ao banco Firebird

Esta apostila destina-se aqueles que tem algum interesse em aprender a conectar-se com um banco Firebird. Caso você seja um iniciante no acesso à banco de dados ou no Delphi/Kylix em geral, não se preocupe, pois mostrarei nesta apostila o básico de acesso à dados.
Qualquer dúvida mande-me um e-mail: obubiba@ig.com.br

Caso você esteja utilizando uma versão Open Edition(OE) do Kylix, e não tiver os componentes da paleta DBExpress instalado, dê uma olhada no link: http://www.clubekylix.com.br/artigos_ver.php?art=1 de autoria do colega **Avalle**.

No Delphi 6 e 7, e no Kylix 2 e 3 Enterprise os componentes DBExpress já vem originalmente instalados. Com a paleta já devidamente instalada, nós temos os sequintes componentes(iguais tanto no Delphi, quanto no Kylix):

SQLConnection;
SQLDataSet;
SQLQuery;
SQLStoredProc;
SQLTable;
SQLMonitor e
SQLClienteDataSet.

Temos uma exceção em relação ao Delphi 7, pois neste, o componente SQLClientDataSet foi substituído pelo **SimpleDataSet**, mas nada que não possa ser contornado. No final da apostila, estarei explicando sobre o SimpleDataSet.

Mostrarei nesta apostila, os componentes DBExpress e em seguida montaremos um exemplo de acesso.

A seguir explicarei detalhadamente cada um dos componentes DBExpress.

1 – O SQLConnection

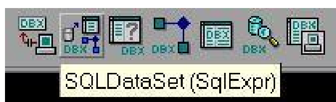
Este é o componente responsável pela conexão com o arquivo físico do banco de dados (*.gdb). É através dele, que obtemos o acesso aos dados, mas não acesso direto ao conteúdo das tabelas, pois isto é feito utilizando-se os outros componentes.
Vou listar aqui, as principais propriedades do SQLConnection:

Propriedade	Significado
Connected	Define se o componente está conectado ao banco. Atenção: para que a conexão ocorra corretamente, é necessário que o servidor Firebird ou Interbase esteja rodando, ou que o banco cliente esteja instalado, caso contrário, na tentativa de conexão, o componete retornará a seguinte exceção: "unavailable database".
Connection Name	Define o nome da conexão a ser utilizada, no nosso caso **IBConnection**.
DriverName	Define qual será o driver utlizado para acessar o banco, no nosso caso **Interbase**.
LoginPrompt	Define se o componente vai requerer o nome do usuário e a senha no nomento da conexão.
Params*	Essa propriedade nos mostra uma lista de subpropriedades do componente, e dente elas, destacamos: **Params.Database**: Define o caminho(path) do arquivo de banco de dados. Informa-se aqui o IP do host de onde estiver o banco, caso a aplicação seja em rede. **Params.SQLDialect**: Define qual dialeto será utilizado no acesso ao banco. Dialeto é o conjunto de caracteres aceito pelo banco. Utilizaremos sempre o Dialeto 3, pois aceita acentos. **Params.User_Name**: Define qual será o nome do usuário. **Params.Password**: Define a senha do usuário. Estes parâmetros podem ser também acessados, dando-se 2 cliques rápidos em cima do componente SQLConnection.

* Para se definir os valores dos parâmetros em modo de execução, utlize a sintaxe:

SQLConnection1.Params.NomeDoParametro := Valor;

2 – O SQLDataSet



Um DataSet é uma estrutura onde são armazenadas listas de registros do banco. O SQLDataSet nos permite mostrar o resultado de uma consulta ou de uma StoredProcedure, executar StoredProcedures que não retornem dados, ou obter os parâmetros disponíveis no banco(tabelas, StoredProcedures, campos de uma tabela). O SQLDataSet é um DataSet unidirecional, ou seja, ele não armazena em memória muitos registros simultâneos, e por causa disto, você só pode navegar usando os métodos First e Next. Ele não suporta atualização de dados na memória, atualizações devem ser feitas utilizando-se a sintaxze SQL diretas no servidor.

As principais propriedades deste componente são:

Propriedade	Significado
Active	Define se o componente está ativado, e executando o comando passado em CommandText, visto logo abaixo.
CommandText	Define o comando em linguagem SQL a ser passado. Podem ser passadas consultas, inclusões e etc..
CommandType	Define o tipo do comando que foi passado em CommandText. Seus valores são ctQuery, ctStoredProc e ctTable.
DataSource	Define qual será o objeto TDataSource que será conectado ao SQLDataSet.
SQLConnetion	Define qual será o componete do tipo TSQLConnection que proverá acesso ao banco.

3 – O SQLQuery

O SQLQuery executa comandos SQL no banco de dados, retornando resultados de consultas, inclusões, exclusões e etc.. O SQLQuery também é unidirecional, só suportando a execução de comandos SQL.

Suas principais propriedades são:

Propriedade	Significado
Active	Define se o SQLQuery está ativado.
SQL	É onde devemos informar qual comando SQL deverá ser executado pela Query.
SQLConnetion	Define qual será o componete do tipo TSQLConnection que proverá acesso ao banco.

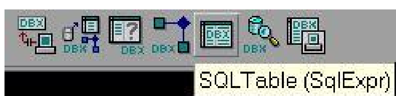
4 – O SQLStoredProc

O SQLStoredProc é um componte específico para a execução de stored procedures existentes no banco. Pode armazenar o resultado de uma stored procedure que retorne um cursor(posição). Também é um componente unidirecional. As execuções das stored procedures são em sintaxe SQL.

Principais propriedades:

Propriedade	Significado
Active	Define se o SQLStoredProc está ativado.
SQLConnetion	Define qual será o componete do tipo TSQLConnection que proverá acesso ao banco.
StoredProcName	Define o nome da stored procedure a ser executada e seus parâmetros, se existirem.

5 – O SQLTable



O SQLTable representa uma tabela do banco de dados. O TSQLTable traz todas as colunas e linhas da tabela especificada, mas também é um componente unidirecional, não permitindo a movimentação entre os registros. Trabalha com uma única tabela de cada vez, podendo realizar alterações, inclusões etc..

Principais propriedades:

Propriedade	Significado
Active	Define se o SQLTable está conectado à tabela especificada.
SQLConnetion	Define qual será o componete do tipo TSQLConnection que proverá acesso ao banco.
TableName	É onde definimos qual o nome da tabela a ser acessada.

6 – O SQLMonitor

O SQLMonitor é um componente utilizado para fazer a depuração da comunicação entre a aplicação e o servidor de banco de dados. Ele grava em log os comandos SQL de uma conexão, adicionando à uma string list.

Principais propriedades do componente:

Propriedade	Significado
Active	Define se o SQLMonitor está ativo e monitorando as mensagens passadas ao banco.
AutoSave	Define se os eventos do banco logados serão automaticamente salvos em um arquivo no disco.
FileName	Define qual será o arquivo no disco que receberá os logs da conexão.
SQLConnetion	Define qual será o componete do tipo TSQLConnection que proverá acesso ao banco.
TraceList	É a propriedade utilizada para se acessar a lista de comandos logados. É do tipo string list. A lista é automaticamente atualizada quando o componente de conexão passa alguma mensagem ao banco.

7 – O SQLClientDataSet

O SQLClientDataSet é um conjunto dos componentes TSQLDataSet e TDataSetProvider(provedor de acesso ao banco). Ele combina o acesso de um dataset unidirecional com a possibilidade de habilitar edições e navegação entre os dados. O componente armazena todo o conteúdo em memória, permitindo salvar as mudaças feitas pela aplicação. Pelo motivo de ter um dataset provider interno, pode armazenar as alterações, e gravá-las mais tarde no banco de dados. É o típico componente utilizado para conexão com o componente TDBGrid.

Principais propriedades do SQLClienteDataSet:

Propriedade	Significado
Active	Define se o componente está conectado ao SQLConnection, e portanto com as informações atualizadas. Ele pode trabalhar desconectado, pois faz cache dos dados em memória. Porém, os dados armazenados não serão os mais atuais.
CommandText	Define o comando SQL a ser passado.
CommandType	Define o tipo do comando que foi passado em CommandText. Seus valores são ctQuery, ctStoredProc e ctTable.
DBConnetion	Define qual será o componete que proverá acesso ao banco de dados.

8 – O SimpleDataSet(Delphi 7)

O SimpleDataSet foi introduzido no lugar do SQLClientDataSet no Delphi 7. Ele tem a mesma função básica do SQLClientDataSet, ou seja, fazer cache de dados e permitir alterações fora do servidor.

Principais propriedades:

Propriedade	Significado
Active	Define se o componente está conectado ao SQLConnection, e portanto com as informações atualizadas. Ele pode trabalhar desconectado, pois faz cache dos dados em memória. Porém, os dados armazenados não serão os mais atuais.
Connection	Define qual será o componete que proverá acesso ao banco de dados.
DataSet	Esta propriedade refere-se a classe DataSet interna do SimpleDataSet. Suas subpropriedades são: **DataSet.Active**: define se o dataset interno está ativo. **DataSet.CommandText**: define qual será o comando SQL passado ao dataset. **DataSet.CommantType**: define qual é o comando passado em CommandText. Seus valores são ctQuery, ctStoredProc e ctTable. Atente que são os mesmos conceitos do já vistos no SQLClientDataSet.

Um exemplo prático

Agora que já conhecemos quais são os componentes da tecnologia DBExpress e suas funções, montarei um exemplo utilizando o Delphi 6 e o Firebird versão 1(compilação 1.0.2.908). O exemplo poderá ser criado no Kylix ou Delphi 7, pois são os mesmos componentes, e a mesma tecnologia de acesso.

Para este exemplo estarei utilizando um banco de dados com uma única tabela chamada Comandos(onde guardaremos os comandos e funções pascal). Para criar o banco com esta tabela, primeiro crie um novo arquivo do tipo texto(ex.: Banco.txt), contendo os seguintes comandos SQL:

```
create database 'Testedb.gdb';
connect 'Testedb.gdb';

create table Comandos(Sintaxe varchar(30) not null,
                      Funcao varchar(50),
                      constraint Sint_Chave primary key(Sintaxe));
```

Salve o arquivo, e com o Firebird/Interbase já instalado, execute o programa Isql.exe, que se encontra na pasta \bin da instalação do servidor em um promtp do MS-Dos(lembre-se que o servidor tem de estar rodando – c:\Firebird\bin\ ibguard.exe):

```
C:\Firebird\Bin\>isql –u SYSDBA – p masterkey
```

Quando aparecer o prompt do Isql, digite:

```
SQL> input 'c:\pasta\Banco.txt'; [Enter]
```

Responta Y à pergunta que aparecer, e o banco será criado.
Dê `exit;` para sair do Isql, e mova o banco recém criado(**Testedb.gdb**) para a pasta onde você for criar nosso exemplo no Delphi, para ficar mais simples e feche o prompt do MS-Dos – nosso trabalho com ele já terminou. Não tenho a intenção de me aprofundar no assunto da criação do banco, seus componentes e na sintaxe SQL, pois isto está além do escopo desta apostila. Para tanto, obtenha minha apostila sobre o servidor Firebird para mais detalhes(http://www.delphix.com.br/Apostilas.php > Apostila Básica de Firebird).

Agora que nosso banco de dados já está criado, podemos inicar a criação do exemplo pelo Delphi.

Menu **File** – **New** – **CLX Application**.

Porquê CLX Application? Para que não ocorram problemas na abertura deste exemplo pelo Kylix.
Monte um formulário como o da imagem:

No componente **SQLConnection**, defina os seguintes valores para as propriedades:

ConnectionName: IBConnection
DriverName: Interbase
LoginPrompt: False

Os valores das outras propriedades automaticamente se preencherão ao se informar as duas primeiras.
Clique no botão de reticências da propriedade **Params**, e no item **DataBase** informe o caminho do nosso arquivo **Testedb.gdb**.

Aponte a propriedade **SQLConnection** dos componentes **SQLDataSet1** e **SQLQuery** para o SQLConnetion1.
Utilizaremos o SQLDataSet para realizar as inclusões, alterações e exclusões e o SQLQuery para realizar as consultas.

É oportuno neste momento, falarmos um pouco sobre as transações do Firebird/Interbase.

Transações são métodos adotados nos bancos de dados Cliente/Servidor para garantir a integridade de todos os dados alterados no banco. Vou exemplificar:

Você precisa corrigir uma folha de pagamento de uma empresa, onde os registros precisam passar por muitas validações e devem ser confirmados ou abandonados todos de uma vez, fazendo sucessivas alterações em duas ou mais tabelas. E se no meio do processo ocorre algum erro? Algumas tabelas terão seus dados gravados, e outras não, tornando os resultados da operação não confiáveis, com informações incorretas. É neste ponto que entram as transações.
Quando se inicia uma transação no banco, o servidor faz cache em memória de todos os registros das tabelas que estão sendo atualizados/apagados. Os dados só são permanentemente gravados no banco, quando a transação se completa com sucesso(com o comando Commit). Caso algum erro venha a ocorrer no meio da transação, todos os dados são abandonados, retornando as tabelas ao modo como estavam antes, sem perda de dados, corrupção, ou erros nas tabelas.

O recurso de transação não existe em bancos Desktop(Paradox por exemplo), apenas em bancos Cliente/Servidor, e tem a função principal de manter a integridade e seguraça dos dados, principalmente onde existem muitos clientes acessando as mesmas informações ao mesmo tempo.

Neste meu exemplo, estarei utilizando o conceito de transação não apenas para incluir, alterar ou excluir dados da tabela, mas também para realizar as consultas. Quando você abre uma transação para realizar uma consulta, você sempre tem certeza que os dados retornados são os mais atuais possíveis, independente se outros clientes estão alterando ou excluindo o registro que você está pesquisando.
Para finalizar uma transação gravando os dados no banco, utlizamos o comando **Commit**, e para desfazer as alterações realizadas pela transação, sem gravar as informações, uilizamos o comando **Rollback**.

Para podermos utilizar corretamente as transações em nosso exemplo, precisamos declarar uma variável, no início da nossa unit:

```
var
  Form1: TForm1;
  Transacao: TTransactionDesc;

implementation

{$R *.xfm}
```

A TtransactionDesc é uma estrutura(record) que define a transação. Um ponteiro dessa estrutura é passado cada vez que se inicia, finaliza ou se cancela uma transação. Utilizaremos transações em todas as nossas ações no banco de dados. E é essa exatamente a intenção nos bancos Cliente/Servidor.

Dê um duplo clique no botão **Incluir Comando**, e no editor de códigos, digite:

```
procedure TForm1.BtIncluirClick(Sender: TObject);
begin
   try
      Transacao.TransactionID := 1;
      Transacao.IsolationLevel := xilREPEATABLEREAD;
      SQLConnection1.StartTransaction(Transacao);
      SQLDataSet1.Close;
      SQLDataSet1.CommandType := ctQuery;
      SQLDataSet1.CommandText := 'insert into
Comandos(Sintaxe, Funcao) values(:Sint, :Func)';
      SQLDataSet1.ParamByName('Sint').AsString :=
EdSintaxe.Text;
      SQLDataSet1.ParamByName('Func').AsString :=
EdFuncao.Text;
      SQLDataSet1.ExecSQL;
      SQLConnection1.Commit(Transacao);
   except
      on Exc:Exception do
         begin
            ShowMessage('Ocorreu um erro na tentativa de
inclusão do registro: ' + Exc.Message);
            SQLConnection1.Rollback(Transacao);
         end;
   end;
end;
```

Definimos um número para a transação que for ser iniciada(TransactionID := 1). Este número de transação deve ser único, não tendo outras transações utilizando o mesmo. Isto é necessário para que cada transação possa ser executada em uma determinada ordem pelo banco, sem que ocorram conflitos.

Logo abaixo, definimos o IsolationLevel para a transação. O IsolationLevel determina como a transação interagirá com outras transações simultâneas que estejam trabalhando com as mesmas tabelas. Para informações mais detalhadas sobre os níveis de isolação, consulte o help do delphi.

Iniciamos a transação com o comando SQLConnetion1.StartTransaction(Transacao);

Enviamos o comando SQL para o SQLDataSet1. As palavras iniciadas com : (dois pontos) são parâmetros, onde passamos seu valor pela propriedade ParamByName(nome).AsTipo := Valor;

Por final executamos o comando com SQLDataSet1.ExecSQL e finalizamos a transação com SQLConnetion.Commit(Transacao).

Caso algum erro ocorra(comando SQL inválido ou conexão com o banco problemática), o código vai para a parte except, e encerramos a transação com SQLConnetion.Rollback(Transacao), sem gravar os dados no banco.

Dê um duplo clique no botão **Alterar Comando**, e no editor de códigos, digite:

```
procedure TForm1.BtAlterarClick(Sender: TObject);
begin
   try
      Transacao.TransactionID := 1;
      Transacao.IsolationLevel := xilREPEATABLEREAD;
      SQLConnection1.StartTransaction(Transacao);
      SQLDataSet1.Close;
      SQLDataSet1.CommandType := ctQuery;
      SQLDataSet1.CommandText := 'update Comandos set Sintaxe
= :Sint, Funcao = :Func where Sintaxe = :Sint';
      SQLDataSet1.ParamByName('Sint').AsString :=
EdSintaxe.Text;
      SQLDataSet1.ParamByName('Func').AsString :=
EdFuncao.Text;
      SQLDataSet1.ExecSQL;
      SQLConnection1.Commit(Transacao);
   except
      on Exc:Exception do
         begin
            ShowMessage('Ocorreu um erro na tentativa de
alteração do registro: ' + Exc.Message);
            SQLConnection1.Rollback(Transacao);
         end;
   end;
end;
```

Dê um duplo clique no botão **Excluir Comando**, e no editor de códigos, digite:

```
procedure TForm1.BtExcluirClick(Sender: TObject);
begin
   try
      Transacao.TransactionID := 1;
      Transacao.IsolationLevel := xilREPEATABLEREAD;
      SQLConnection1.StartTransaction(Transacao);
      SQLDataSet1.Close;
      SQLDataSet1.CommandType := ctQuery;
      SQLDataSet1.CommandText := 'delete from Comandos where
Sintaxe = :Sint';
      SQLDataSet1.ParamByName('Sint').AsString :=
EdSintaxe.Text;
      SQLDataSet1.ExecSQL;
      SQLConnection1.Commit(Transacao);
   except
```

```
      on Exc:Exception do
         begin
            ShowMessage('Ocorreu um erro na tentativa de
exclusão do registro: ' + Exc.Message);
            SQLConnection1.Rollback(Transacao);
         end;
   end;
end;
```

Dê um duplo clique no botão **Procurar**, e no editor de códigos, digite:

```
procedure TForm1.BtProcurarClick(Sender: TObject);
var
   Consulta: string;
begin
   if Trim(EdBusca.Text) <> '' then
      begin
         Consulta := 'select Sintaxe, Funcao from comandos
where ';
         //No campo:
         if RadioCampo.ItemIndex = 0 then
            Consulta := Consulta + 'Sintaxe '
         else
            Consulta := Consulta + 'Funcao ';

         //Que:
         if RadioTipo.ItemIndex = 0 then
            begin
               Consulta := Consulta + 'starting with ''' +
EdBusca.Text + '''';
            end
         else
            if RadioTipo.ItemIndex = 1 then
               Consulta := Consulta + 'like ''%' +
EdBusca.Text + '%'''
            else
               Consulta := Consulta + 'like ''%' +
EdBusca.Text + '''';

         try
            Transacao.TransactionID := 1;
            Transacao.IsolationLevel := xilREPEATABLEREAD;
            SQLConnection1.StartTransaction(Transacao);
            SQLQuery1.Close;
            SQLQuery1.SQL.Clear;
            SQLQuery1.SQL.Append(Consulta);
```

```
            SQLQuery1.Open;
            SQLConnection1.Commit(Transacao);
         except
            on Exc:Exception do
               begin
                  ShowMessage('Ocorreu um erro na consulta: '
+ Exc.Message);
                  SQLConnection1.Rollback(Transacao);
               end;
         end;
         CarregaLista;
      end;
end;
```

Como eu já havia dito, utilizo o SQLQuery para realizar as consultas.
A propriedade SQL do SQLQuery é uma stringlist, que contém os comandos a serem executados.
Limpamos o conteúdo da lista SQL do SQLQuery com SQL.Clear, adicionamos o comando referente à consulta, e a executamos com o comando Open.

Eu preencho a variável Consulta conforme o usuário escolhe os filtros nos RadioGroups. Para maiores informações sobre sintaxe SQL para filtro, leia minha apostila sobre Firebird.

Após realizarmos a consulta, executamos a procedure CarregaLista, que vai preencher o ListView com o ResultSet retornado pelo banco, cujo código é o dado a seguir:

```
procedure TForm1.CarregaLista;
begin
   SQLQuery1.First;
   ListView1.Items.Clear;
   while true do
      begin
         ListView1.Items.Add;
         ListView1.Items.Item[ ListView1.Items.Count - 1
].Caption := SQLQuery1.Fields.Fields[0].AsString;
         ListView1.Items.Item[ ListView1.Items.Count - 1
].SubItems.Add(SQLQuery1.Fields.Fields[1].AsString);
         try
            SQLQuery1.Next;
         except
            break;
         end;
      end;
end;
```

A procedure funciona da seguinte maneira:

Após realizarmos a consulta com o comando Open do SQLQuery, direcionamos o cursor do DataSet para o primeiro registro(SQLQuery1.First).
Enquanto a condição for verdadeira, o while continua, até pegar todos os registros do DataSet e passá-lo para o ListView, um a um, com SQLQuery.Next, passando os valores de Field[0] e Field[1].
Logo que o cursor do DataSet chegar no último registro, o comando SQLQuery.Next resultará em uma exceção, pois não existem mais registros, executando break e saindo do loop.

No evento **OnDblClick**(Duplo clique) do ListView1, digite:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
begin
   EdSintaxe.Text := ListView1.ItemFocused.Caption;
   EdFuncao.Text := ListView1.ItemFocused.SubItems.Text;
end;
```

Isso fará com que ao ser dado um duplo clique em algum dos valores mostrados, o conteúdo seja transferido para os edits de edição, para poderem ser alterados.

Dê um duplco clique no formulário e digite no editor de códigos:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   try
      SQLConnection1.Connected := true;
   except
      ShowMessage('Erro ao tentar conectar-se ao banco.');
   end;
end;
```

Assim, quando o form for criado, a aplicação tentará se conectar ao banco.
E no evento **OnClose** do form, digite:

```
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
   SQLConnection1.Connected := false;
end;
```

para que a conexão seja finalizada ao se fechar o form.

Agora, já temos um formulário básico para entrada de dados, e alterações das informações do banco. Só utilizamos 3 componentes de acesso à dados.

Criando uma DataSet navegável

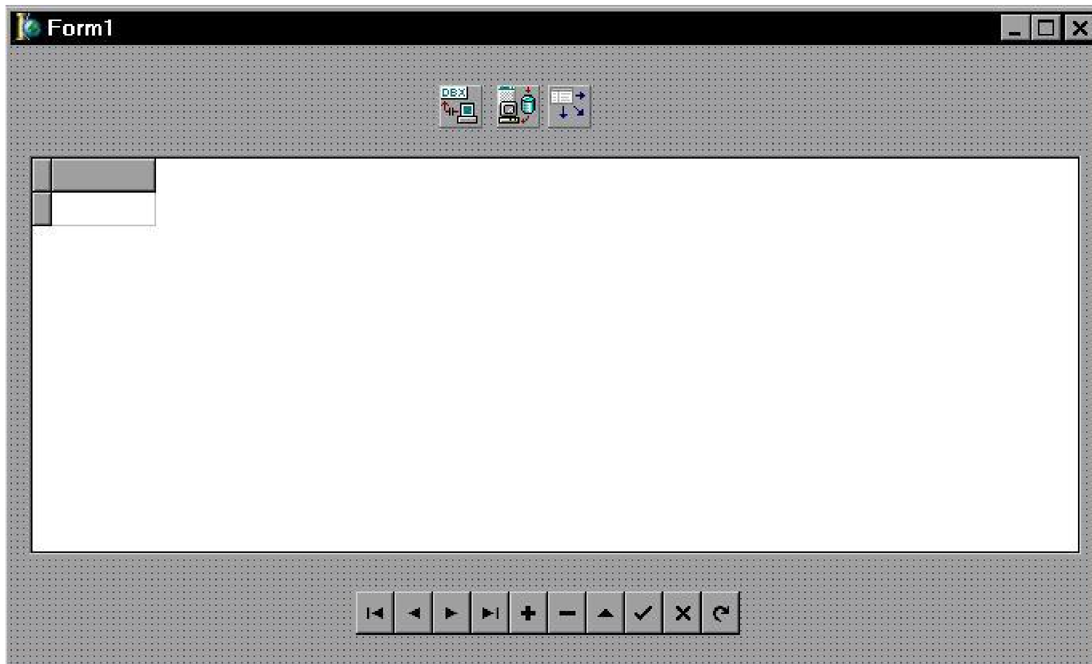
Neste nosso exemplo, nós não utilizamos nenhum DataSet navegável, visto que os componentes DBExpress são unidirecionais, e não permitem tal fato. Para tanto utilizamos o conjunto SQLQuery + ListView para poder trazer à tela a lista de registros.

Mas você pode encontrar situações onde um DataSet navegável e de fácil manutenção seja necessário. Para podermos ter tal recurso, utilizamos o **SQLClientDataSet**, que pode armazenar um DataSet visualizável e navegável através de um DBGrid.

Então vamos aprender como fazer isso. Primeiro utilizando o SQLClientDataSet do Delphi 6 e Kylix, e depois utilizando o SimpleDataSet, disponível no Delphi 7.

Utilizando o SQLClientDataSet

Monte um formulário como o da figura (CLX Application):



Configure o **SQLConnection** da mesma maneira que no exemplo anterior.

No **SQLClientDataSet** aponte a propriedade **DBConnention** para o SQLConnetion1. Preencha a propriedade **CommandText** com 'select * from COMANDOS', ou clique no botão das reticências para preencher o campo. Escolha ctQuery para a propriedade **CommandType**.

Na propriedade **DataSet** do **DataSource** aponte para o SQLClientDataSet1.

Conecte o **DBGrid** e o **DBNavigator** ao DataSource, e ative as conexões. Caso existam registros na tabela Comandos, eles aparecerão no DBGrid.

Rode a aplicação e você pode utilizar os botões do DBNavigator para alterar os dados. Temos então um DataSet navegável e atualizável.

O SQLClientDataSet mantém em memória todos os registros recuperados da tabela ou da junção de tabelas à qual estiver conectado, permitindo a alteração. E ele não é um componente unidireional, permitindo que o cursor navegue entre os registros.

Com o SQLClientDataSet, você pode trabalhar em um ambiente Cliente/Servidor ou em um ambiente Desktop, pois você pode definir o filtro aplicado à tabela por parâmetros SQL, ou pela propriedade Filter.

Quando você filtra a tabela por meio de instruções SQL(cláusula Where), o banco apenas retorna o resultado da pesquisa, assim como acontece com os outros compoentes unidirecionais da tecnologia DBExpress. Mas quando você escolhe filtrar pela propriedade Filter, toda a tabela é trazida para a aplicação, e o filtro sendo aplicado sobre ela, consumindo mais banda da rede e memória. Por isso, em ambientes Cliente/Servidor, prefira realizar o filtro principal diretamente com instruções SQL, e refiltrar utilizando a propriedade Filter.

Utilizando o SimpleDataSet (Delphi 7)

Para utilizar o SimpleDataSet monte o mesmo form anterior, apenas substituindo o SQLClientDataSet pelo SimpleDataSet.

No **SimpleDataSet**, preencha a propriedade Connection com o **SQLConnetion1**. Abra o item **DataSet**, e na propriedade **CommandText** ponha 'select * from COMANDOS' ou clique no botão de reticências. Escolha ctQuery para **CommanType**, e na propriedade **DataSource** aponte para o **DataSource1**.

Ative o SimpleDataSet, e se o SQLConnetion estiver conectado, e a tabela Comandos contiver registros, eles aparecerão no DBGrid.
Rode a aplicação e você poderá trabalhar nos registros com o DBNavigator.

Nestes exemplos eu omiti muito código que seria referente à validação das entradas de dados, como verificar se os edits contém algum texto ou não. Mas este é um exemplo básico de acesso, porém não distante de como deve ser realmente uma aplicação do tipo Cliente/Servidor, por exemplo sem conter um DataSet navegável, que traria um consumo maior de recursos.
O uso da tecnologia DBExpress é baseada no fato de que o servidor é quem faz tudo na aplicação, ou seja, o cliente apenas mostra os resultados das sentenças SQL, ou no caso de inclusões/alterações não mostra nada, apenas que foi feito. As restrições impostas pelo fato de utilizar componentes unidirecionais, não permitindo a navegação entre registros é compensada pela velocidade no processamento principalmente.
Onde antes seria necessário a utilização de computadores mais potentes para os clientes, apenas se melhora a performance do servidor, trazendo muita economia tanto na manutenção quanto na implementação do banco de dados.

Espero com esta apostila que você tenha um ponto de partida para estudar e utilizar a tecnologia DBExpress de forma eficiente.