

Desenvolvendo uma aplicação utilizando ClientDataSet com DBExpress e Firebird (modelo Cliente/Servidor)



Por Eduardo Rocha
eduardo@edudelphipage.com.br
EduDelphiPage - <http://www.edudelphipage.com.br>

Índice

Criando o Banco de Dados	3
Criando a Aplicação	12
Criando o DataModule Principal	13
Criando o DataModule para o Cadastro de Pedidos	18
Criando o Formulário de Cadastro de Pedidos	22
Parametrizando a Query	27
Joins	32
Joins - Buscando o Cliente	36
Ajustando UpdateMode do Provider - Performance na Atualização e Controle de Concorrência	43
Trabalhando com Mestre/Detalhe	47
Mestre/Detalhe - Utilizando MasterSource/MasterFields (trafegando todos os registros detalhes)	48
Mestre/Detalhe - Utilizando MasterSource/MasterFields (filtrando os registros detalhes)	54
Mestre/Detalhe - Utilizando NestedDataSet	56
Buscando Informações do Produto	62
Trabalhando com Clones	64
Implementando mais o Projeto - Calculando Valor Total do Item	67
Campos Aggregate	69
Campos InternalCalc	74
Generators – Trabalhando com Campos Auto-Incremento	77
Criando o Cadastro de Tipos de Pedidos	85
Criando a Pesquisa de Pedidos	91
Criando a Pesquisa de Clientes	101
Ordenando Registros	107
Ordenando os Registros – Utilizando Índices Temporários	108
Ordenando os Registros – Utilizando Índices Persistentes	109
Filtrando Registros em Memória	113
Filtrando os Registros em Memória – Utilizando a Propriedade Filter	114
Filtrando os Registros em Memória – Utilizando Range	115
Pesquisando os Registros em Memória	118
Trabalhando com Refresh	120
Trabalhando com Refresh – Utilizando o Método Refresh	121
Trabalhando com Refresh – Utilizando o Método RefreshRecord	122
Desfazendo Alterações – CancelUpdates, UndoLastChange, RevertRecord e SavePoint	124
Lendo e Gravando os Dados Localmente em Arquivos	126
Trabalhando com Tabelas Somente em Memória	129
Transações	136
Transações – Utilizando SQLQuery	137
Transações – Utilizando ClientDataSet	142
Eventos BeforeUpdateRecord e AfterUpdateRecord do Provider	146
Trabalhando com Múltiplas Tabelas – Definindo qual será Atualizada	150
ReconcileError - Tratamento de Erros	157
Monitorando mensagens do Banco de Dados	161
Distribuindo a aplicação	163

Criando o Banco de Dados

Em Paradox tínhamos diversos arquivos que formavam nosso banco de dados, onde cada arquivo correspondia a uma tabela e outros eram referentes a índices e campos blob's.

No caso do Firebird, temos apenas um arquivo, chamado de DataBase, onde conterà todas as tabelas, índices, integridades, generators entre outros.

Existem diversas formas para criarmos o banco de dados, o Firebird disponibiliza um utilitário chamado ISQL que é utilizado para criação e manipulação do banco de dados, porém não é tão elegante, pois é executado no prompt do dos, portanto não temos um visual agradável.

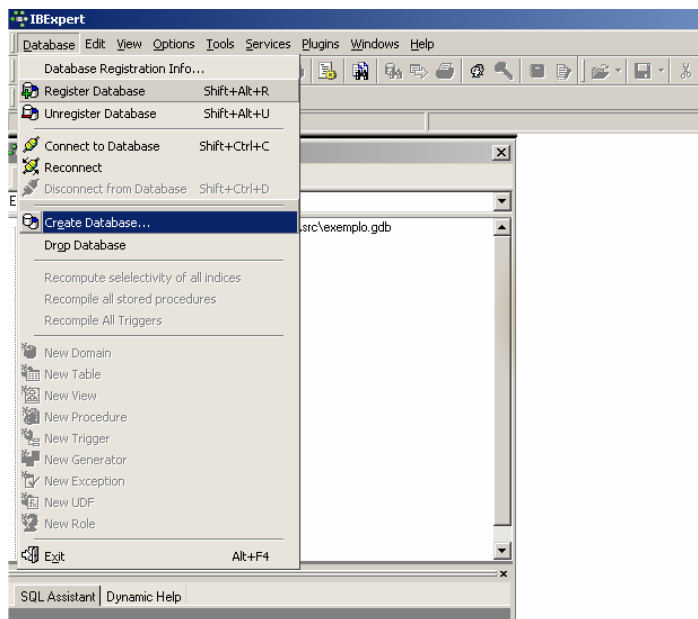
Por este motivo utilizaremos uma ferramenta gráfica chamada **IBExpert**, que é um utilitário para manipulação de banco de dados Interbase e Firebird. O IBExpert possui duas versões, Personal (gratuita) e Full (paga), pode ser baixado diretamente do site: <http://www.ibexpert.com>

Criando o diretório

Antes de criarmos o banco de dados, criaremos a seguinte estrutura de diretório para seu destino: C:\CursoClientDataSet\projeto\db

Criando o banco

Abra o **IBExpert**, vá ao menu **DataBase** e clique no item **CreateDataBase**.



Acessando menu CreateDataBase para criação do banco de dados

Será exibida a seguinte tela:

Tela de criação do banco de dados

Segue abaixo uma breve descrição dos campos:

Server:

Indica se o banco será criado localmente (Local) ou remotamente (Remote)

Server name:

Só estará disponível se optarmos por **Remote** na opção **Server**. Neste campo informamos o Nome ou IP do Servidor.

Protocol:

Só estará disponível se optarmos por **Remote** na opção **Server**. Neste campo informamos o tipo de protocolo de comunicação com o servidor, na maioria dos casos TCP/IP.

Database:

Caminho onde será criado o arquivo do banco de dados.

Username:

Nome do usuário do banco de dados. O usuário **SYSDBA** é o padrão do servidor.

Password:

Senha do usuário do banco de dados. A senha **masterkey** é a senha padrão do usuário **SYSDBA**.

Page Size:

Normalmente deixamos 1024 que é o tamanho de página padrão. Mais detalhes a respeito poderão ser encontrados na documentação do Interbase/Firebird.

Charset:

O mais recomendando é o WIN1252 para evitar problemas com acentos.

SQL Dialect:

O dialeto é útil para usarmos novos recursos que surgem no servidor de banco de dados que não são compatíveis com as versões anteriores.

- Dialeto 1 garante compatibilidade com as versões anteriores.
- Dialeto 2 usado muito em testes, diagnósticos.
- Dialeto 3 é o mais utilizado permitindo fazer uso dos novos recursos.

A grande diferença do Dialeto 3 para o Dialeto 1 está nos seguintes aspectos:

- Campos Data/Hora
- Campos numéricos
- Objetos delimitados por aspas duplas ("")

Outros detalhes das diferenças técnicas poderão ser obtidos na documentação do Interbase/Firebird.

Register Database After Creating:

Opção específica do IBExpert. Ativando esta opção, logo após clicarmos em OK, será aberta a tela de Registro do Banco de Dados, caso contrário, teríamos que fazer isso manualmente. O Registro será explicado mais abaixo.

Em nosso caso, deixaremos os campos preenchidos da seguinte forma:

Server: Local

Database: C:\CursoClientDataSet\projeto\db\exemplo.fdb

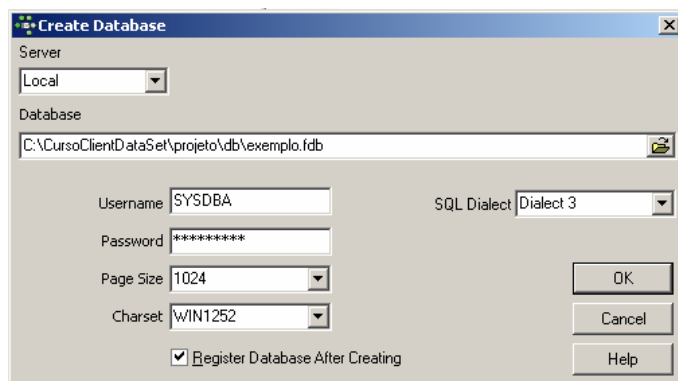
Username: SYSDBA

Password: masterkey

Page Size: 1024

Charset: WIN1252

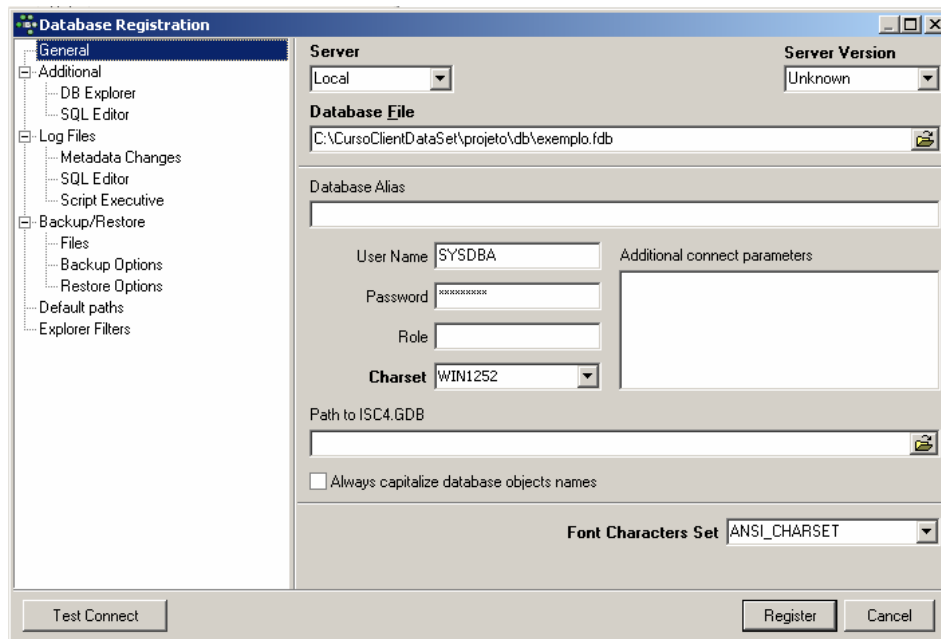
SQL Dialect: Dialect 3



Criando o banco de dados

Clique no botão **OK** para confirmar.

Registrando o banco no IBExpert



Registrando o banco

Agora já temos o banco de dados criado. Nesta tela criaremos o velho e antigo “alias” que estávamos acostumados a criar no SQLEditor por exemplo, porém, este “alias” é utilizado apenas no IBExpert, não estará disponível no Delphi.

Podemos perceber que temos muitos campos preenchidos, pois já informamos na tela anterior.

Basta então fazermos comentários de apenas alguns deles:

Server Version:

Aqui informamos qual banco/versão estaremos utilizando, pois o IBExpert permite-nos trabalhar com diversas versões do Interbase e Firebird.

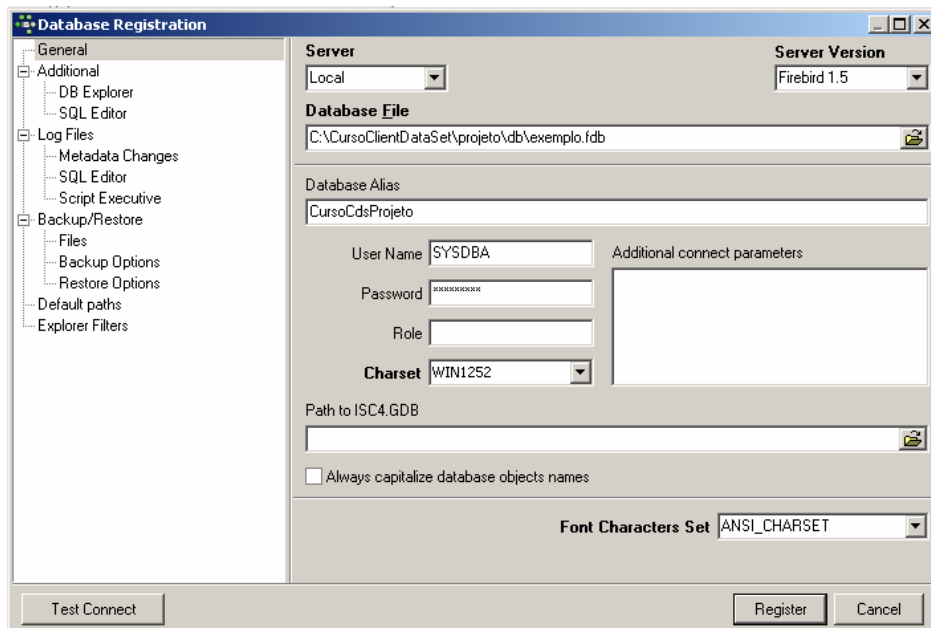
Database Alias:

Nome do alias para conexão que estamos criando.

A configuração da conexão deverá ficar da seguinte forma:

Server Version: Firebird 1.5

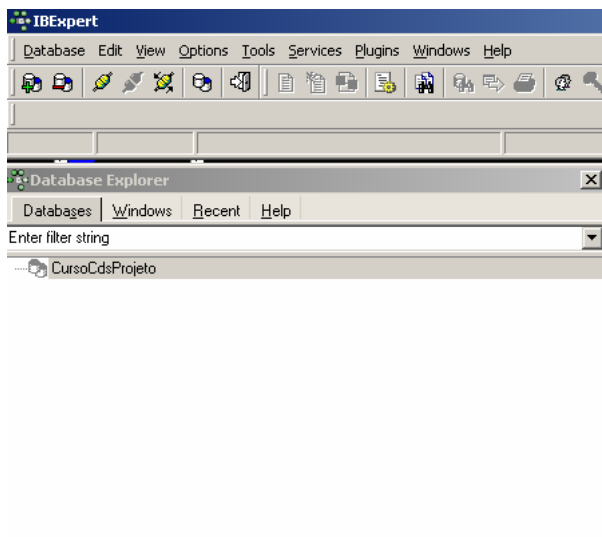
Database Alias: CursoCdsProjeto



Criando Conexão

Clique no botão **Register** para que a conexão seja registrada no IBExpert.

Podemos perceber a conexão criada sendo exibida do lado esquerdo na lista de conexões, como mostra a seguir:



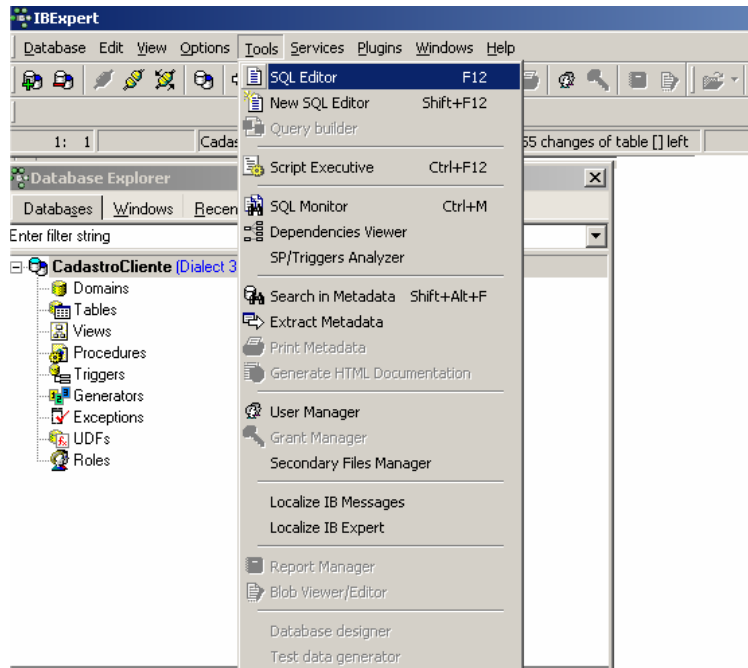
Conexão criada

Abra a conexão dando um duplo clique no item criado (CursoCdsProjeto) e agora já poderemos manipular nosso banco de dados.

Criando as tabelas

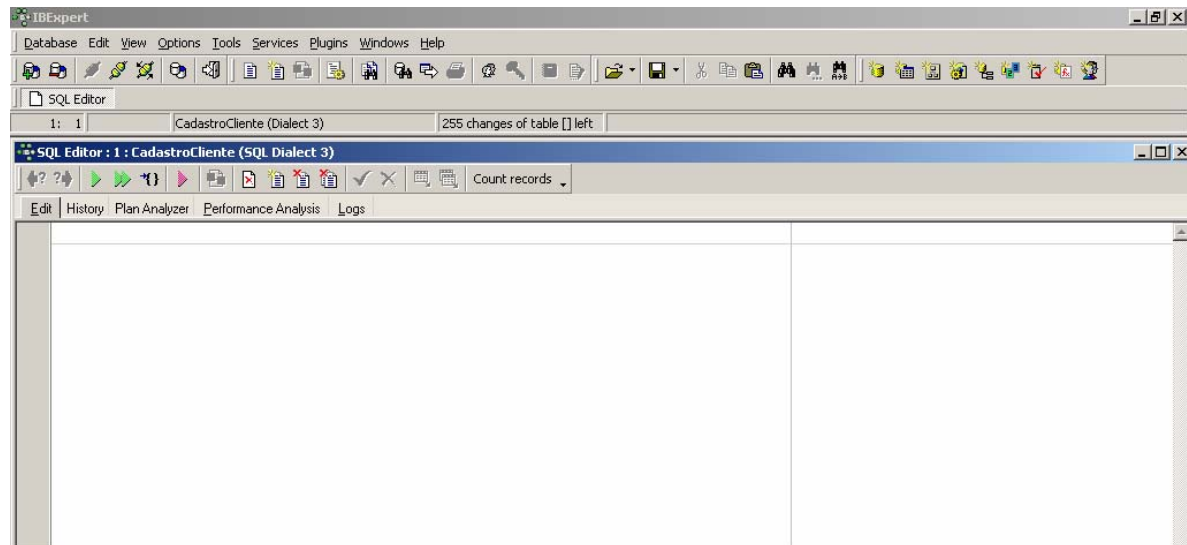
Podemos criar as tabelas do banco de dados facilmente através de cliques, porém utilizaremos scripts SQL para visualizarmos como manipulamos instruções SQL nesta ferramenta.

O **Editor SQL** do IBExpert pode ser acessado através da tecla de atalho **F12** ou no menu **Tools->SQL Editor**.



Acessando o Editor SQL

Clicando nesta opção, o Editor SQL será exibido:

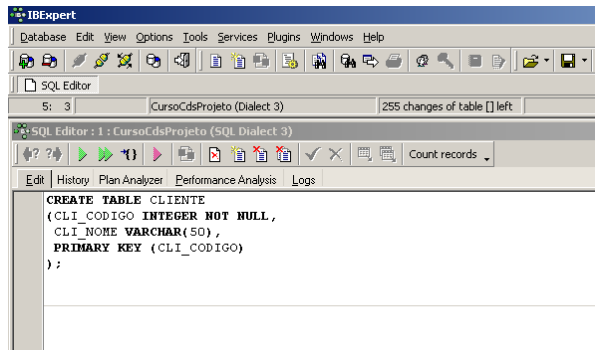


Editor SQL

Neste quadro é que devemos inserir nossas instruções SQL para manipulação do banco.

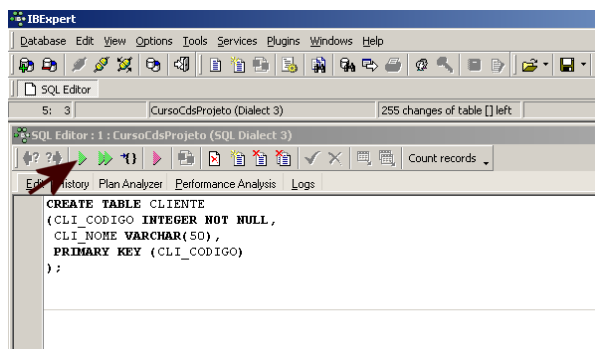
Insira o seguinte script para criação da tabela **CLIENTE**.

```
CREATE TABLE CLIENTE  
(CLI_CODIGO INTEGER NOT NULL,  
CLI_NOME VARCHAR(50),  
PRIMARY KEY (CLI_CODIGO)  
)
```



Inserindo script SQL

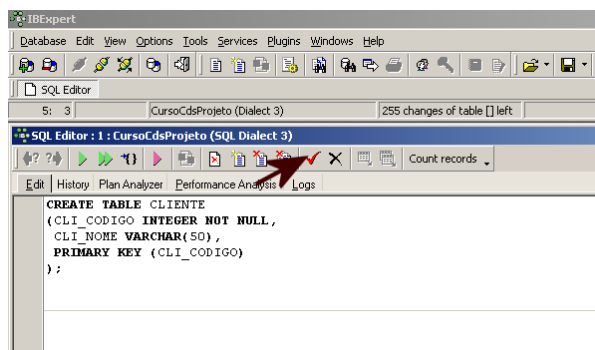
Em seguida, executamos o SQL através da tecla de atalho **F9** ou clicando no botão **Play** conforme demonstrado na figura abaixo:



Executando o script SQL

Devemos executar um COMMIT para que as alterações possam ser confirmadas no servidor de banco de dados.

Para isso, podemos utilizar a tecla de atalho **CTRL + ALT + C** ou utilizar o botão **Check**:



Executando um Commit

Repita os mesmos passos com as instruções SQL a seguir para termos a estrutura do banco pronta para utilizarmos em nosso projeto:

Criando a tabela de Produtos

```
CREATE TABLE PRODUTO  
(PROD_CODIGO INTEGER NOT NULL,  
  PROD_DESCRICAO VARCHAR(50),  
  PROD_VALOR NUMERIC(9,2),  
  PRIMARY KEY(PROD_CODIGO)  
)
```

Criando a tabela de Tipos de Pedido

```
CREATE TABLE TIPOPEDIDO  
(TP_CODIGO INTEGER NOT NULL PRIMARY KEY,  
  TP_DESCRICAO VARCHAR(40))
```

Criando a tabela de Pedidos

```
CREATE TABLE PEDIDO  
(PED_NUMERO INTEGER NOT NULL,  
  PED_DATA DATE,  
  PED_VALOR NUMERIC(9,2),  
  PED_DATAEXCLUSAO DATE,  
  PED_DATAMODIFICACAO DATE,  
  CLI_CODIGO INTEGER,  
  TP_CODIGO INTEGER,  
  PRIMARY KEY(PED_NUMERO),  
  FOREIGN KEY (CLI_CODIGO) REFERENCES CLIENTE,  
  FOREIGN KEY (TP_CODIGO) REFERENCES TIPOPEDIDO)
```

Criando a tabela de Itens do Pedido

```
CREATE TABLE PEDITEM  
(PED_NUMERO INTEGER NOT NULL,  
  PROD_CODIGO INTEGER NOT NULL,  
  PI_DESCRICAO VARCHAR(50),  
  PI_QTDE INTEGER,  
  PI_VALUNIT NUMERIC (9,2),  
  PI_VALTOTAL NUMERIC(9,2),  
  PRIMARY KEY(PED_NUMERO, PROD_CODIGO),  
  FOREIGN KEY (PED_NUMERO) REFERENCES PEDIDO,  
  FOREIGN KEY (PROD_CODIGO) REFERENCES PRODUTO  
)
```

Inserindo registros na tabela de Clientes

```
INSERT INTO CLIENTE (CLI_CODIGO, CLI_NOME) VALUES (1, 'CLIENTE 1')  
INSERT INTO CLIENTE (CLI_CODIGO, CLI_NOME) VALUES (2, 'CLIENTE 2')  
INSERT INTO CLIENTE (CLI_CODIGO, CLI_NOME) VALUES (3, 'CLIENTE 3')
```

Inserindo registros na tabela de Produtos

```
INSERT INTO PRODUTO (PROD_CODIGO, PROD_DESCRICAO, PROD_VALOR) VALUES  
(1, 'PRODUTO 1', 100)  
INSERT INTO PRODUTO (PROD_CODIGO, PROD_DESCRICAO, PROD_VALOR) VALUES  
(2, 'PRODUTO 2', 200)  
INSERT INTO PRODUTO (PROD_CODIGO, PROD_DESCRICAO, PROD_VALOR) VALUES  
(3, 'PRODUTO 3', 300)
```

Inserindo registros na tabela de Tipos de Pedido

```
INSERT INTO TIOPEDIDO (TP_CODIGO, TP_DESCRICAO) VALUES (1, 'TIPO 1')
```

```
INSERT INTO TIOPEDIDO (TP_CODIGO, TP_DESCRICAO) VALUES (2, 'TIPO 2')
```

```
INSERT INTO TIOPEDIDO (TP_CODIGO, TP_DESCRICAO) VALUES (3, 'TIPO 3')
```

Criando a Aplicação

Antes de iniciarmos com a criação do projeto, criaremos o diretório onde o mesmo será gravado:

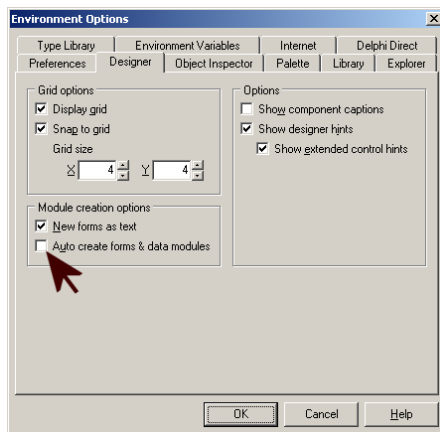
C:\CursoClientDataSet\projeto\src

Ajustando o Delphi

Faremos um ajuste para que todo e qualquer **Formulário/Data module** criado não seja definido como **AutoCreate**, pois sempre instanciaremos em tempo de execução, evitando assim consumo de memória desnecessário e sobrecarga do sistema em sua inicialização.

Vá ao menu **Tools** e clique em **Environment Options**.

Na tela que será exibida, clique na paleta **Designer** e **desabilite** a opção **Auto create forms & data modules**.



Desabilitando a opção Auto create forms & data modules em Environment Options

Confirme clicando no botão **OK**.

Iniciando o projeto

Crie um novo projeto, ajuste o nome do formulário principal para **frmPrincipal** e salve-o como **ufrmPrincipal.pas**. Em seguida salve o projeto com o nome **CursoCdsProjeto.dpr**.

Criando o DataModule Principal

Nosso próximo passo será criar o DataModule principal, mas antes vamos fazer um breve comentário sobre DataModules e como utilizaremos em nossa aplicação:

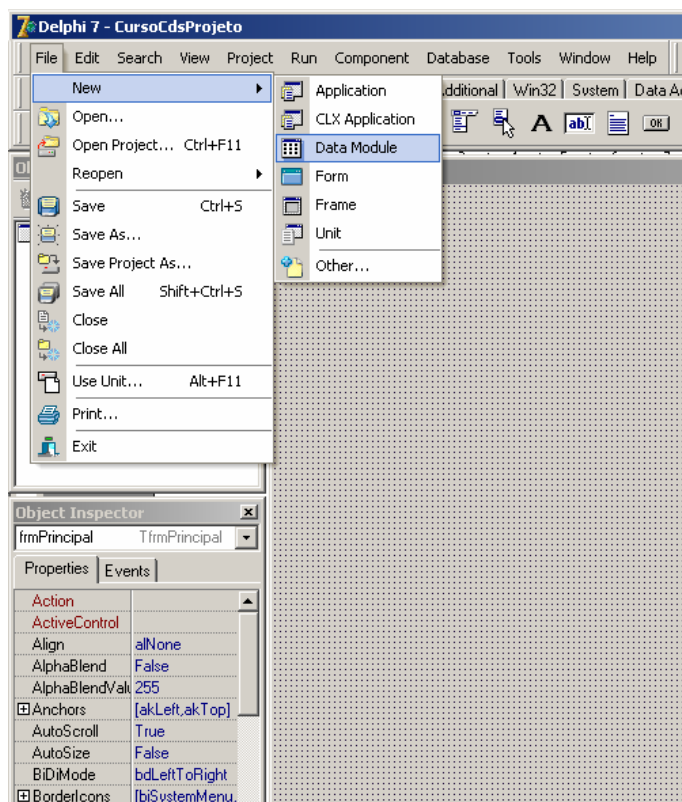
Os DataModules servem como um centralizador de componentes não visuais, são muito utilizados para componentes de acesso a dados, regras de negócios, etc..

Ao contrário do que muitos pensam, eles são super leves, não sobrecarregam a aplicação, pois são derivados diretamente da classe TComponent.

Em nossa aplicação teremos **um** DataModule para **cada** 'módulo' do sistema, onde colocaremos os componentes de acesso do respectivo módulo e suas regras de negócio. No formulário do respectivo módulo deixaremos apenas os componentes visuais e regras de interface. Este modelo é interessante, pois não sobrecarregamos o formulário com componentes de acesso a dados, regras de negócio e facilitamos também futuras manutenções.

Deixamos destacado que teremos **um** DataModule para **cada** módulo do sistema, não que isso seja obrigatório, mas quanto mais pudermos separar os módulos, mais fácil ficarão as manutenções, e assim também evitamos ter diversos componentes instanciados desnecessariamente num único DataModule.

Adicionaremos nosso primeiro DataModule ao projeto, para isso, clique no menu **File->New>Data Module**.



Criando um DataModule

Ajuste o nome para **dmPrincipal** e salve-o como **udmPrincipal.pas**.

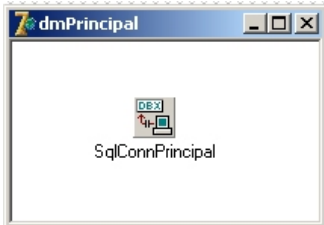
Este é o nosso DataModule principal, nele adicionaremos nosso **componente de conexão** que será utilizado pelos outros componentes de acesso a dados do nosso sistema.



TSQLConnection (dbExpress)

Name: SqlConnPrincipal

LoginPrompt: False



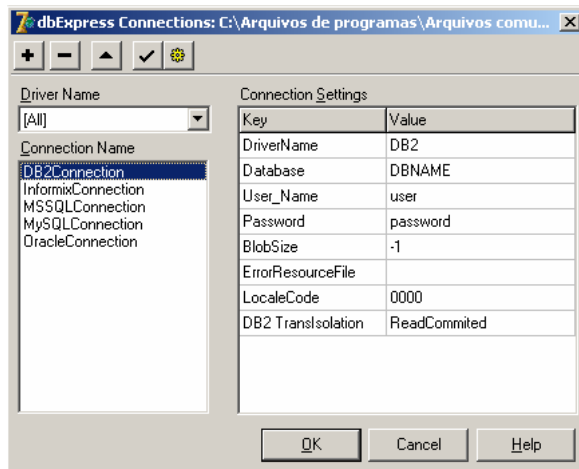
DataModule com o componente de Conexão

Desligamos a propriedade **LoginPrompt** para evitar que sejam requisitados os dados de usuário e senha quando formos conectar com o banco.


Neste momento temos apenas o componente de conexão no DataModule, precisamos agora configurá-lo para conectar-se ao banco de dados. Podemos fazer isso utilizando a propriedade **Params** ou utilizando uma conexão definida na DBExpress através da propriedade **ConnectionName**. Utilizaremos esta opção para fins didáticos e pela vantagem de podermos reutilizá-la em outros projetos caso precisemos conectar com o mesmo banco, portanto, nosso próximo passo será criar a respectiva conexão.

Criando uma Conexão

Agora criaremos nossa conexão, para isso precisamos abrir o **Editor de Conexões** dando um duplo clique no componente **SQLConnection**.



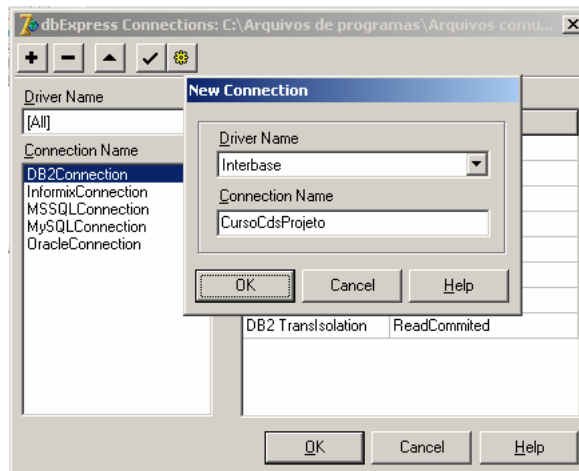
Editor de Conexões

Clicando no botão  , será exibida uma janela para informarmos o **Driver** e o **Nome da conexão**.

Defina da seguinte forma:

Driver Name: Interbase

Connection Name: CursoCdsProjeto



Definindo o Driver e o nome da conexão

Apesar de estarmos utilizando o banco Firebird, nosso acesso será feito com o driver para Interbase, pois já é nativo do Delphi. Podemos utilizá-lo com Firebird sem problemas. Existem drivers dbExpress no mercado específicos para Firebird, no qual poderão ser encontrados nos sites:

<http://www.upscene.com>

<http://www.progdigy.com/UIB/>

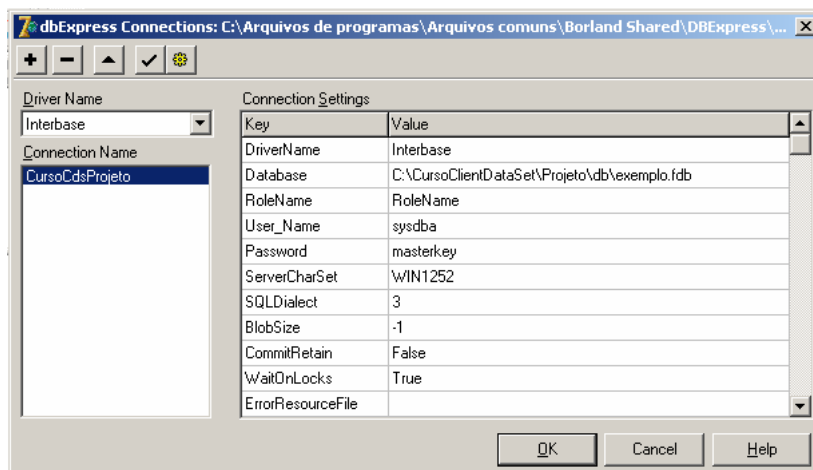
Clique no botão **OK** para confirmar.

Feito isso voltaremos à tela onde ajustaremos os parâmetros da conexão:

DataBase: C:\CursoClientDataSet\Projeto\db\exemplo.fdb

ServerCharSet: WIN1252

SQLDialect: 3



Ajustando os parâmetros da conexão

Entendendo o parâmetro DataBase

Em nosso caso, informamos o path do arquivo local, porém, em uma aplicação Cliente/Servidor, o banco de dados estaria no servidor, portanto usamos de forma diferente, temos que especificar o servidor + o caminho do banco de dados. Utilizando protocolo TCP/IP, fazemos isso utilizando a seguinte regra:

SERVIDOR:CAMINHO_DO_ARQUIVO

Supondo que o IP do servidor fosse 192.168.0.10 e o nome da máquina SERVBANCO, nosso path ficaria da seguinte forma:

Servidor Windows (utilizando protocolo TCP/IP)

192.168.0.10:C:\CursoClientDataSet\Projeto\db\exemplo.fdb

ou

SERVBANCO:C:\CursoClientDataSet\Projeto\db\exemplo.fdb

Servidor Linux (utilizando protocolo TCP/IP)

192.168.0.10:/CursoClientDataSet/Projeto/db/exemplo.fdb

ou

SERVBANCO:/CursoClientDataSe/Projeto/db/exemplo.fdb

Para os demais protocolos, a estrutura é diferente:

Utilizando protocolo NetBEUI

Estrutura: \\SERVIDOR\CAMINHO_DO_ARQUIVO


Exemplo: \\SERVBANCO\C:\CursoClientDataSet\Projeto\db\exemplo.fdb

Utilizando protocolo IPX/SPX (rede Novell)

Estrutura: SERVIDOR@VOLUME:/CAMINHO_DO_ARQUIVO

Exemplo: SERVBANCO@vol1:/CursoClientDataSet/Projeto/db/exemplo.fdb

Vale lembrar que o suporte para IPX/SPX está descontinuado no Firebird 1.5.

Vamos testar se a conexão está funcionando, para isso basta clicar no botão 

Estando tudo correto, confirme clicando no botão **OK**.

Podemos notar que o componente SqlConnection teve sua propriedade ConnectionName ajustada automaticamente para conexão que acabamos de criar. Se depois precisarmos desenvolver outro projeto utilizando o mesmo banco, não precisaremos mais criar a conexão, basta definir a propriedade ConnectionName do componente SqlConnection apontando para *CursoCdsProjeto*.

Criando o DataModule para o Cadastro de Pedidos

Agora criaremos um novo DataModule, ele será utilizado pelo formulário de Cadastro de Pedidos e armazenará regras de negócios e componentes de acesso a dados relativos a este módulo.

Siga o mesmo caminho utilizado na criação do primeiro DataModule, clique no menu **File->New->Data Module**.

Em seguida nomeie-o para **dmCadPedido** e salve-o como **udmCadPedido.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...  
implementation  
  
uses  
    udmPrincipal;  
...
```

Adicione os seguintes componentes no DataModule:



TClientDataSet (Data Access)

Name: cdsPedido

ProviderName: dspPedido



TDataSetProvider (Data Access)

Name: dspPedido

DataSet: qryPedido



TSQLQuery (dbExpress)

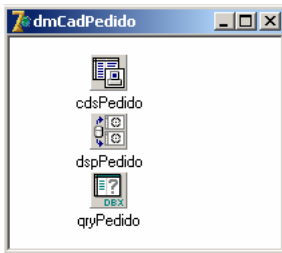
Name: qryPedido

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT  
*  
  
FROM  
    PEDIDO
```

*Poderá acontecer de não estar disponível na propriedade SQLConnection do SQLQuery o componente de conexão que está no DataModule principal, neste caso é necessário que você abra e mantenha aberto o respectivo DataModule (**dmPrincipal**).*



DataModule de Cadastro de Pedido

A primeira coisa que precisamos ajustar no ClientDataSet é a propriedade **ProviderName**. Esta é a propriedade que indica quem é o provedor de dados do ClientDataSet conforme já vimos nas explicações do seu funcionamento.

*Devemos tomar cuidado com esta propriedade, pois ela é do tipo **string**, portanto podemos estar digitando qualquer valor que o Delphi não irá fazer nenhum tipo de verificação em tempo de projeto e se alterarmos o nome do componente **dspPedido** a propriedade não será notificada, conseqüentemente não será ajustada automaticamente.*

No **Provider** ajustamos a propriedade **DataSet** apontando para o componente **SQLQuery**, é para este DataSet que o Provider fará a requisição dos Dados.

Como vimos no funcionamento do ClientDataSet, poderíamos ter qualquer DataSet associado ao Provider, optamos em utilizar o componente SQLQuery pela sua simplicidade e finalidade, já que neste caso queremos manipular o SQL para extração dos dados, mas trabalhando com DBExpress podemos utilizar os demais componentes como SQLTable, no qual apenas informamos o nome da tabela ao invés de manipularmos o SQL, SQLDataSet, etc. Para as outras tabelas, utilizaremos estes componentes para fins didáticos.

No **SQLQuery** indicamos qual é o seu componente de conexão através da propriedade **SQLConneciton** e a query utilizada para extração dos dados na propriedade **SQL**.

Com o SQL que utilizamos, todos os registros com todos os campos serão enviados ao ClientDataSet, porém isto não é o ideal, mais adiante ajustaremos para trabalharmos com parâmetros especificando também os campos, pois desta forma ganhamos performance e reduzimos o tráfego na rede.

Adicionando tabela de Tipo de Pedido utilizando SQLTable

Seguindo o mesmo conceito, agora incluiremos mais um conjunto de componentes para tabela de **Tipo de Pedido** que será utilizada como Lookup. Neste caso, ao invés de utilizarmos o componente **SQLQuery**, utilizaremos o **SQLTable**, pois é uma tabela simples, não haverá necessidade de definirmos o SQL, pois sempre retornaremos todos os registros.



TClientDataSet (Data Access)

Name: cdsTipoPedido

ProviderName: dspTipoPedido



TDataSetProvider (Data Access)

Name: dspTipoPedido

DataSet: tblTipoPedido



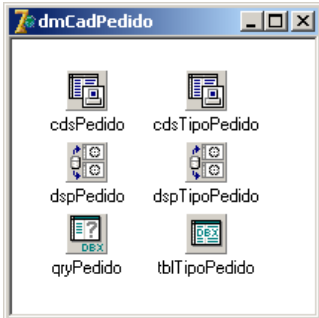
TSQLTable (dbExpress)

Name: tblTipoPedido

SQLConnection: dmPrincipal.SqlConnPrincipal

TableName: TIOPEDIDO

Podemos perceber que no componente SQLTable não informamos o SQL, definimos a tabela na propriedade **TableName**, desta forma, quando forem requisitados os dados, internamente será gerada uma query do tipo: `SELECT * FROM TIOPEDIDO`.



DataModule de Cadastro de Pedidos com tabela de Tipos de Pedido

Adicionando a tabela de Cliente utilizando SQLDataSet

Incluiremos também o conjunto de componentes para tabela de Clientes, na qual a princípio será utilizada como Lookup, porém mais adiante veremos que neste caso não é recomendado.

Utilizaremos o componente **SQLDataSet** apenas para fins didáticos, mas não seria necessário em nosso caso, vimos que o diferencial deste componente é que ele nos permite trabalharmos de 3 formas: Queries, Tables e Stored Procedures.

Adicione os seguintes componentes no DataModule:



TClientDataSet (Data Access)

Name: cdsCliente

ProviderName: dspCliente



TDataSetProvider (Data Access)

Name: dspCliente

DataSet: sdsCliente




TSQLDataSet (dbExpress)

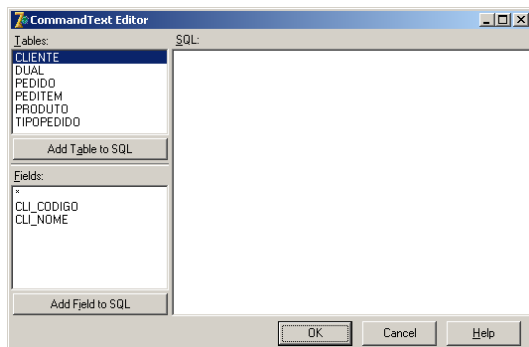
Name: sdsCliente

SQLConnection: dmPrincipal.SqlConnPrincipal

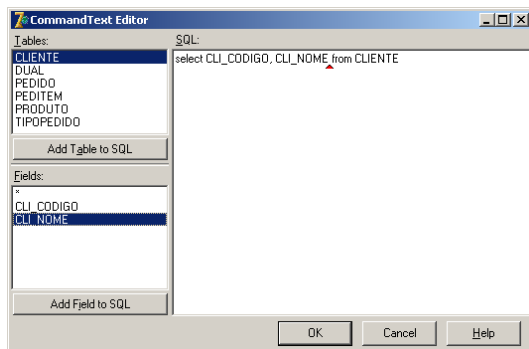
CommandType: ctQuery

CommandText: `SELECT CLI_CODIGO, CLI_NOME FROM CLIENTE`

A propriedade **CommandText** varia de acordo com a propriedade **CommandType**, definindo como **ctTable**, será disponibilizada uma lista de tabelas existentes no banco, optando por **ctStoredProc**, visualizaremos a lista de Stored Procedures e optando por **ctQuery**, teremos disponível um **EditorSql** que é acessado clicando no botão  da propriedade **CommandText**.



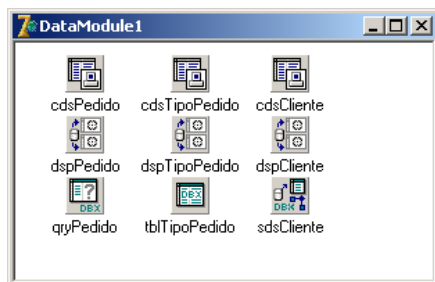
CommandText Editor – Editor SQL do componente SQLDataSet



CommandText Editor – Montando a Query

Para montarmos o SQL, podemos digitá-lo diretamente no quadro, clicar nos botões **Add Table To SQL** e **Add Field to SQL** ou dar um duplo clique nas tabelas e colunas até formar o SQL que desejamos.

Depois de montado o SQL e confirmado, o **Datamodule** deverá estar da seguinte forma:



DataModule de Cadastro de Pedidos com todas as tabelas

Criando o Formulário de Cadastro de Pedidos

Crie um novo formulário, ajuste o nome para **frmCadPedido** e salve-o como **ufrmCadPedido.pas**.

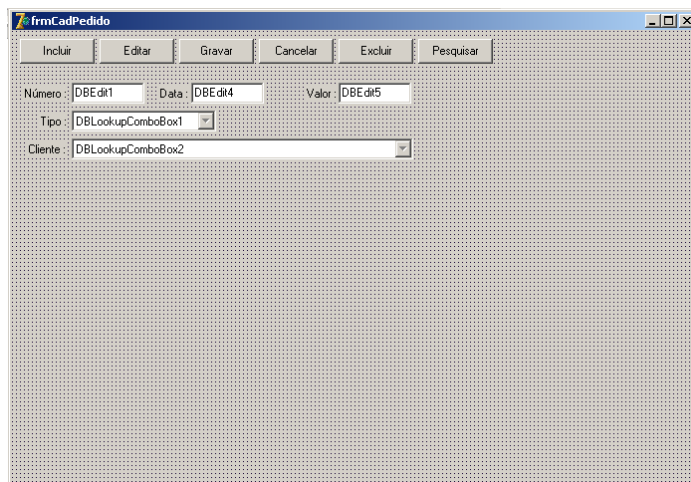
Adicione na cláusula **uses** a unit **udmCadPedido**.

```
...  
implementation
```

```
uses  
    udmCadPedido;
```

```
...
```

Monte o formulário da seguinte forma:



Tela de Cadastro de Pedidos

Nomeie os botões na respectiva ordem:

btnIncluir, **btnEditar**, **btnGravar**, **btnCancelar**, **btnExcluir** e **btnPesquisar**.

Adicione 3 DataSources:



TDataSource (Data Access)

Name: dtsPedido

DataSet: dmCadPedido.cdsPedido



TDataSource (Data Access)

Name: dtsTipoPedido

DataSet: dmCadPedido.cdsTipoPedido



TDataSource (Data Access)

Name: dtsCliente

DataSet: dmCadPedido.cdsCliente

Conforme já explicamos, sempre manipulamos os dados no ClientDataSet e não no DataSet ao qual o Provider está ligado, por este motivo ligamos os DataSources nos ClientDataSets.

Selecione todos DBEdits e ajuste a propriedade **DataSource** apontando para **dtsPedido**, em seguida defina o **DataField** de cada um.

Selecione o **DBLookupComboBox de Tipo de Pedido** e ajuste as seguintes propriedades:

DataSource: dtsPedido
DataField: TP_CODIGO
ListSource: dtsTipoPedido
ListField: TP_DESCRICAO
KeyField: TP_CODIGO

Faça também o ajuste no **DBLookupComboBox de Cliente**:

DataSource: dtsPedido
DataField: CLI_CODIGO
ListSource: dtsCliente
ListField: CLI_NOME
KeyField: CLI_CODIGO

Codificando o formulário

No evento **OnCreate** codificaremos para que seja instanciado o DataModule de Cadastro de Pedidos e em seguida aberto os ClientDataSets:

```
procedure TfrmCadPedido.FormCreate(Sender: TObject);
begin
  dmCadPedido := TdmCadPedido.Create(Self);
  dmCadPedido.cdsPedido.Open;
  dmCadPedido.cdsTipoPedido.Open;
  dmCadPedido.cdsCliente.Open;
end;
```

Já que estamos criando o DataModule e abrindo os ClientDataSets no evento OnCreate, faremos o contrário no evento **OnDestroy**:

```
procedure TfrmCadPedido.FormDestroy(Sender: TObject);
begin
  dmCadPedido.cdsPedido.Close;
  dmCadPedido.cdsTipoPedido.Close;
  dmCadPedido.cdsCliente.Close;
  dmCadPedido.Free;
  dmCadPedido := nil;
end;
```

No evento **OnClose**, liberamos o formulário:

```
procedure TfrmCadPedido.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
  frmCadPedido := nil;
end;
```

Codificando os botões

Botão Incluir:

```
procedure TfrmCadPedido.btnIncluirClick(Sender: TObject);
begin
    dmCadPedido.cdsPedido.Append;
end;
```

Botão Editar:

```
procedure TfrmCadPedido.btnEditarClick(Sender: TObject);
begin
    dmCadPedido.cdsPedido.Edit;
end;
```

Botão Gravar:

```
procedure TfrmCadPedido.btnGravarClick(Sender: TObject);
begin
    dmCadPedido.cdsPedido.Post;
    if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
        dmCadPedido.cdsPedido.CancelUpdates;
end;
```

Botão Cancelar:

```
procedure TfrmCadPedido.btnCancelClick(Sender: TObject);
begin
    dmCadPedido.cdsPedido.Cancel;
    dmCadPedido.cdsPedido.CancelUpdates;
end;
```

Botão Excluir:

```
procedure TfrmCadPedido.btnExcluirClick(Sender: TObject);
begin
    dmCadPedido.cdsPedido.Delete;
    if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
        dmCadPedido.cdsPedido.CancelUpdates;
end;
```

Entendendo os códigos

Os métodos que estamos executando são os mesmos que utilizamos nas TTable's por exemplo. A grande diferença está após a chamada dos métodos Post e Delete, pois chamamos o método **ApplyUpdates**. Isto é necessário, pois ele é o método responsável em aplicar, gravar fisicamente as mudanças feitas ao banco de dados.

O método **ApplyUpdates** é uma função que nos retorna o número de erros ocorrido na atualização. Esta função exige um parâmetro, no qual indicamos o número de erros permitidos. Este parâmetro poderá assumir os seguintes valores:

0: Indica que não permitimos nenhum tipo de erro, ou grava tudo com sucesso ou não grava nada.

-1: Indica que queremos gravar tudo que for possível, o que der erro, será descartado, mas não afetará aqueles que foram gravados com sucesso.

> 0: Qualquer número maior que zero estará indicando o número de erros permitidos

Na maioria dos casos, estaremos usando o valor **zero** como parâmetro.

Podemos notar que ao chamarmos o método ApplyUpdates, sempre verificamos se o retorno é diferente de zero.

```
if ApplyUpdates(0) <> 0 then CancelUpdates
```

Implementamos desta forma, pois caso algum erro ocorra na atualização, precisamos limpar as **pendências**, caso contrário, em uma próxima chamada ao método ApplyUpdates, o ClientDataSet tentará gravar não somente as novas alterações, mas também as que ficaram pendentes.

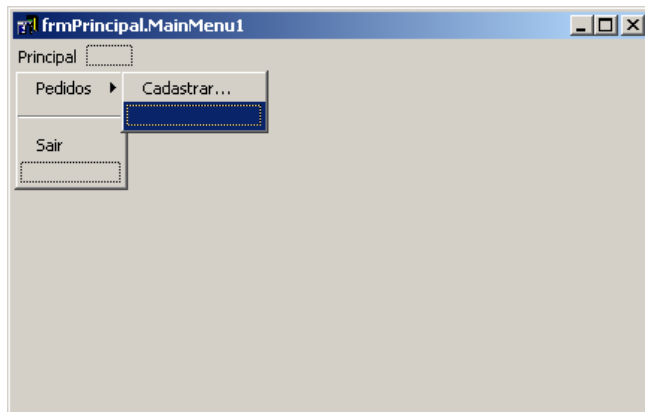
Outra diferença que notamos comparando com TTables é a chamada do método CancelUpdates após o Cancel, com a explicação anterior já podemos entender o motivo.

Dica: O uso dos métodos Post e Cancel são opcionais, os mesmos já são executados automaticamente pelos métodos ApplyUpdates e CancelUpdates respectivamente.

Ajustando a tela principal do sistema

Precisamos ajustar o formulário principal e criar uma chamada ao formulário de Cadastro de Pedidos.

Abra o **frmPrincipal** e insira um componente **MainMenu** ajustando-o para que fique com a seguinte estrutura:



Ajustando Menus

Na cláusula **uses** adicione a unit **ufrmCadPedido**.

```
...  
implementation  
  
uses  
    ufrmCadPedido;  
...
```

Codifique o evento **OnClick** do item de menu **Cadastrar** da seguinte forma:

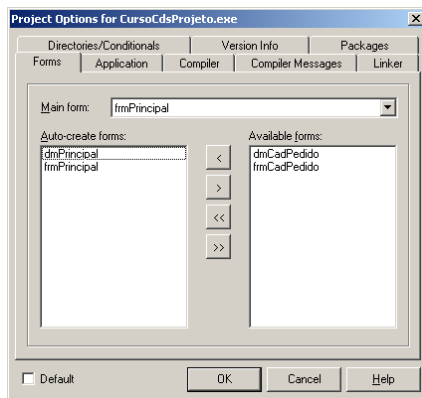
```
procedure TfrmPrincipal.Cadastrar1Click(Sender: TObject);
begin
    if not Assigned(frmCadPedido) then
        frmCadPedido := TfrmCadPedido.Create(Self);
    frmCadPedido.Show;
end;
```

Ajustando a ordem de criação do formulário e data module

No início configuramos o Delphi para que todos novos Formulários e DataModules não fossem instanciados automaticamente, porém nosso DataModule principal é uma exceção, precisamos instanciá-lo automaticamente e antes do formulário principal, já que poderíamos precisar utilizar antes da instância deste, por exemplo, em uma tela de login.

Portanto, faremos este ajuste nas opções do Projeto. Acesse o menu **Project->Options** e selecione a aba Forms.

Ajuste-a colocando o DataModule Principal do lado esquerdo e acima do Formulário Principal, ficando da seguinte forma:



Ajustando Auto-create forms

Feito os ajustes, basta confirmar clicando no botão **OK** e a aplicação já está pronta para ser testada.

Testando a aplicação

Execute a aplicação, inclua, edite e exclua Pedidos para verificar se os mesmos estão sendo aplicados fisicamente. ao banco de dados.

O importante é percebermos que o registro só é gravado fisicamente após a chamada do método **ApplyUpdates**, enquanto não é executado, ele é mantido apenas em memória. Podemos ter esta certeza fazendo uma pausa no código do botão **Gravar** por exemplo, colocando um break (através do debugador) no método Post, antes da chamada do método ApplyUpdates, assim checamos que após o Post, o registro ainda não está fisicamente no banco de dados, somente depois da execução do método ApplyUpdates.

Parametrizando a Query

Nossa query de Pedidos não possui uma cláusula WHERE que restrinja os registros, portanto todos serão enviados ao ClientDataSet, gerando assim um tráfego na rede, diferente do que ocorre no BDE quando trabalhamos com TQuery por exemplo, já que mesmo não colocando um WHERE, os registros são trafegados conforme vão sendo requisitados.

Um exemplo prático

Se tivermos uma tabela com 500.000 registros e abrirmos com uma query do tipo `SELECT * FROM TABELA` trabalhando com BDE/TQuery, certamente a abertura será instantânea, pois neste momento os registros não foram trafegados pela rede, eles serão trafegados somente quando requisitados através de códigos ou DataControls (dbgrids, dbnavigator, etc.). Já no caso do ClientDataSet, pelo fato de trabalhar tudo em memória, demandará mais tempo, pois ao abri-lo, ele requisitará os registros ao Provider e este requisitará a Query tudo que ela retorna, enviando em seguida para o ClientDataSet, portanto todos registros serão trafegados pela rede na abertura do ClientDataSet, gerando assim muito tráfego e lentidão.

Por este motivo, ao trabalharmos com este componente, devemos adotar alguns conceitos.

Novos conceitos

Um dos principais conceitos é trafegar apenas o necessário. Não mais abriremos um Grid com todos os registros para o usuário navegar, teremos sempre uma tela de pesquisa parametrizada para restringi-los.

Nesta tela de pesquisa até podemos permitir que o usuário edite o registro, porém pode ser mais interessante termos uma tela somente para manutenção, na qual passamos como parâmetro a chave do registro selecionado e esta tela faria um `SELECT` apenas neste registro.

Este modelo é interessante, pois existem casos onde temos tabelas com muitos campos, inclusive campos blobs, registros detalhes, e isso poderia gerar um tráfego na rede conforme o número de registros retornados pela pesquisa fosse aumentando. Portanto, na tela de pesquisa fazemos o `SELECT` apenas nos campos necessários para visualização, e na tela de edição podemos fazer o `SELECT` em todos os campos, não haverá problemas, pois estaremos trafegando apenas um registro.

Com base neste conceito, ajustaremos nossa Query para trafegar apenas um Pedido com base na chave primária, na qual será passada como parâmetro.

Colocando em prática

Abra o DataModule **dmCadPedido** e ajuste o **SQL** do componente **qryPedido** da seguinte forma:

```
SELECT
  PED_NUMERO,
  PED_DATA,
  PED_VALOR,
  TP_CODIGO,
  CLI_CODIGO
FROM
  PEDIDO
WHERE
  PED_NUMERO = :PED_NUMERO
```

Nesta Query tivemos 2 grandes mudanças:

Especificação dos campos

Não estamos mais utilizando o * (asterisco) para seleção dos campos, indicamos individualmente e apenas o necessário. Fazendo desta forma evitamos que todos os campos sejam trafegados pela rede e ganhamos um pouco de performance na execução da query, pois quando utilizamos * (asterisco), o servidor de banco de dados precisa consultar quais os campos disponíveis na tabela para poder retorná-los, então podemos poupá-lo deste trabalho especificando os campos individualmente.

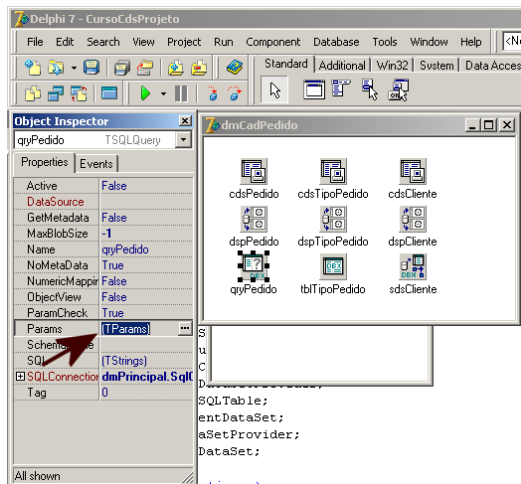
Restrição para apenas um registro

Estamos restringindo para que somente um registro seja aberto, pois a condição (WHERE) está sendo feita sobre o campo chave.

Quando definimos o valor **:PED_NUMERO**, estamos na realidade criando um parâmetro com o nome **PED_NUMERO**, sempre que precisamos criar um parâmetro na query utilizamos esta estrutura **:NOME**. Alimentaremos este parâmetro em tempo de execução, mas antes precisamos ajustar seu tipo de dado e parâmetro.

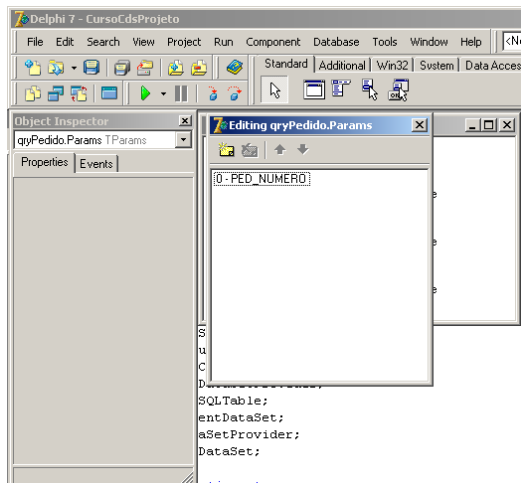
Ajustando o parâmetro criado

Os parâmetros sempre estarão armazenados na propriedade **Params** do DataSet, portanto clique nesta propriedade do componente **qryPedido**.



Acessando a propriedade Params do componente qryPedido

Em seguida será exibido o Editor de Parâmetros:



Editor de Parâmetros

Temos disponível o parâmetro **PED_NUMERO**. Selecione-o e ajuste as seguintes propriedades:

DataType: ftInteger

ParamType: ptInput

Em **DataType** indicamos o **Tipo de Dado**, no caso **ftInteger** (Integer).

Em **ParamType** indicamos o **Tipo de Parâmetro**, no caso **ptInput** (Entrada). O tipo **ptOutput** é utilizado para parâmetros de saída, por exemplo, quando trabalhamos com Stored Procedure.

Agora já temos o parâmetro criado e ajustado, precisamos alimentá-lo em 2 locais:

Alimentando o parâmetro na criação do formulário de cadastro

Ao criar o form, alimentaremos o valor deste parâmetro com **NULL**, desta forma o ClientDataSet será aberto e nenhum registro será trafegado, já que não existem Pedidos com chave nula, assim ele estará vazio, pronto para receber um novo registro a ser cadastrado.

Abra o formulário **frmCadPedido** e ajuste o evento **OnCreate** da seguinte forma:

```
procedure TfrmCadPedido.FormCreate(Sender: TObject);
begin
  dmCadPedido := TdmCadPedido.Create(Self);
  dmCadPedido.cdsPedido.FetchParams;
  dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').Value := NULL;
  dmCadPedido.cdsPedido.Open;
  dmCadPedido.cdsTipoPedido.Open;
  dmCadPedido.cdsCliente.Open;
end;
```

Entendendo o código

Adicionamos apenas 2 linhas:

```
dmCadPedido.cdsPedido.FetchParams;
```

Esta linha pede para o Provider trazer ao ClientDataSet os parâmetros disponíveis no DataSet ao qual ele (Provider) está ligado, ou seja, o parâmetro PED_NUMERO que criamos na Query estará disponível agora na propriedade Params do ClientDataSet.

Poderíamos acessar a propriedade Params diretamente na Query, porém teríamos problemas no modelo multicamadas, já que a Query não estaria no mesmo local do ClientDataSet, portanto, utilizamos desta forma para se adequar a qualquer modelo.

```
dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').Value := NULL;
```

Neste código apenas alimentamos o parâmetro PED_NUMERO com valor NULL, justamente para termos o resultado que comentamos, de não trafegar nenhum registro pela rede, deixando o ClientDataSet vazio para receber um novo cadastro.

Alimentando o parâmetro no botão de pesquisa

Implementaremos a princípio uma busca simples, onde o usuário apenas informará o Número do Pedido e alimentaremos o parâmetro com este valor para que assim possamos abrir apenas o Pedido pesquisado.

Codifique o evento **OnClick** do botão **Pesquisar** da seguinte forma:

```
procedure TfrmCadCliente.btnPesquisarClick(Sender: TObject);
var
  sNumero: string;
begin
  if InputQuery('Pesquisa de Pedido', 'Informe o Número', sNumero) then
  begin
    dmCadPedido.cdsPedido.Close;
    dmCadPedido.cdsPedido.FetchParams;
    dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').AsInteger :=
    StrToInt(sNumero);
    dmCadPedido.cdsPedido.Open;
    if dmCadPedido.cdsPedido.IsEmpty then
      ShowMessage('Pedido não encontrado !');
  end;
end;
```

Entendendo o código

Seguimos a seguinte sequência:

- Chamamos o método InputQuery para pegarmos o Número do Pedido a ser pesquisado.
- Fechamos o ClientDataSet.
- Pegamos os parâmetros disponíveis na Query.
- Alimentamos o parâmetro com o Número informado pelo usuário.
- Abrimos o ClientDataSet.
- Verificamos se o registro foi encontrado.

Testando a aplicação

Ao abrirmos o formulário de cadastro, o ClientDataSet estará vazio, justamente pelo fato de termos atribuído o valor NULL ao parâmetro PED_NUMERO. Cadastre um Pedido em seguida grave-o. Logo após, clique no botão Pesquisar informando um número de Pedido inexistente, veremos que uma mensagem será exibida informando que o Pedido não foi encontrado, em seguida, pesquise pelo número de Pedido cadastrado para checar se o mesmo é aberto com sucesso no ClientDataSet.

Joins

Implementaremos agora um recurso muito importante para reduzirmos tráfego na rede e ganharmos performance.

Nos campos **Cliente e Tipo de Pedido** utilizamos Lookups para podermos visualizar os registros e poder selecioná-los nos componente DBLookupComboBox.

Este recurso é muito utilizado em BDE/TTable/TQuery, facilita bastante, porém com o ClientDataSet temos que tomar cuidado, pois como dissemos, em sua abertura, ele trafega todos os registros retornados pela Query, portanto, todos os registros das tabelas Cliente e Tipo de Pedido serão trafegados pela rede para estarem disponíveis nos DBLookupComboBox, diferente do que ocorreria se fosse utilizado BDE, pois os registros seriam trafegados de acordo com a demanda, conforme navegasse pelos itens.

No caso do **Tipo de Pedido** não haveria tanto problema, pois será uma tabela pequena, com poucos registros, o maior problema é com tabelas grandes, por exemplo, a tabela de Clientes.

Neste caso, adotamos um outro método, o que chamamos de **JOINS**. Trazemos o Nome do Cliente não através de um Lookup, mas sim na própria Query de Pedido, e para o usuário selecionar um Cliente, devemos ter uma tela de Pesquisa de Cliente Parametrizada, reduzindo assim o tráfego na rede, onde ele pesquisa e seleciona apenas o Cliente que deseja associar ao Pedido.

Colocando em prática

A primeira coisa que precisamos fazer é ajustar o SQL do Pedido para que faça um JOIN com a tabela de CLIENTE trazendo o campo NOME do respectivo Cliente pertencente ao Pedido.

Abra o DataModule **dmCadPedido** e ajuste o **SQL** do componente **qryPedido** para que fique da seguinte forma:

```
SELECT
  PED.PED_NUMERO,
  PED.PED_DATA,
  PED.PED_VALOR,
  PED.CLI_CODIGO,
  PED.TP_CODIGO,
  CLI.CLI_NOME
FROM
  PEDIDO PED
  INNER JOIN CLIENTE CLI ON CLI.CLI_CODIGO = PED.CLI_CODIGO
WHERE
  PED.PED_NUMERO = :PED_NUMERO
```

Fizemos 2 mudanças na Query:

Nome dos campos prefixados com o alias

Ajustamos os campos para que sejam prefixados com o alias que definimos as tabelas, pois como estaremos trabalhando com mais de uma tabela, é importante fazermos isso para não haver conflitos de nome de campos iguais entre elas.

Inclusão da cláusula INNER JOIN

Para obter o resultado que precisamos, acrescentamos a cláusula **INNER JOIN** para unirmos o registro da tabela de CLIENTE ao registro tabela de PEDIDO e assim poder extrair o nome do Cliente do respectivo Pedido retornado pela Query.

A junção é feita por um ou mais campos, os registros são unidos quando os campos tiverem o mesmo valor, ou seja, para cada registro de Pedido, será localizado um Cliente cujo Código (CLI.CLI_CODIGO) seja o mesmo associado ao Pedido (PED.CLI_CODIGO), encontrando, teremos uma única linha contendo os dados do Pedido e do Cliente, assim podemos obter o Nome do Cliente acessando o campo CLI.CLI_NOME.

O INNER JOIN exige que o registro da outra tabela obrigatoriamente seja encontrado, ou seja, no nosso caso, o CLIENTE deve existir, se não, o registro do PEDIDO não será exibido. Nos casos onde temos situações que o registro possa não existir, ou até mesmo quando o campo pode ser NULO, devemos utilizar o **LEFT JOIN** ao invés de **INNER JOIN**, que possui a mesma sintaxe, diferenciando apenas que ele não exige a existência do registro da tabela que estamos unindo.

Inclusão do campo CLI_NOME

Adicionamos o campo CLI.CLI_NOME na seleção de campos, pois desta forma podemos ligá-lo a um DataControl.

Ajustando o formulário de cadastro

Abra o formulário **frmCadPedido**, remova o componente **DBLookupComboBox** e o **DataSource** referente à tabela de Cliente, logo após adicione um **TDBEdit**:



TDBEdit (Data Controls)

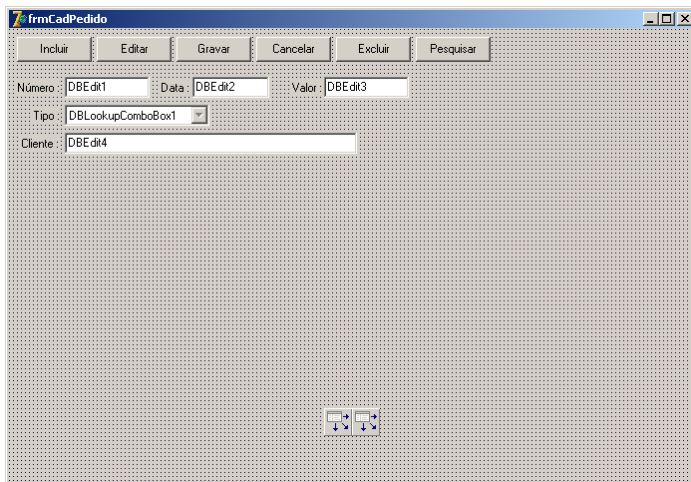
DataSource: dtsPedido

DataField: CLI_NOME

ReadOnly: True

Caso o campo CLI_NOME não esteja disponível, basta abrir e fechar o ClientDataSet de Pedidos através da propriedade Active para refletir o campo que adicionamos e assim disponibilizá-lo para utilização.

Definimos a propriedade **ReadOnly** como **True** pois não poderíamos permitir o usuário alterar o Nome do Cliente nesta tela.

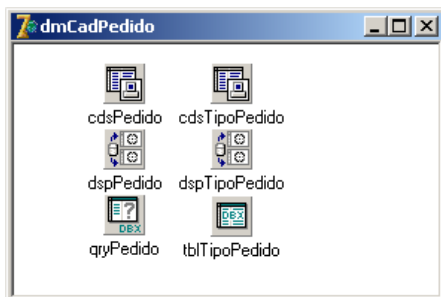


Tela de Cadastro de Pedidos – Trocando DBLookupComboBox de Cliente pelo DBEdit e removendo o DataSource

Removendo o ClientDataSet de Clientes

Já que estamos utilizando Joins, não precisaremos mais do ClientDataSet de Clientes, portanto, vamos remover os seguintes componentes do DataModule:

- cdsCliente
- dspCliente
- sdsCliente



DataModule de Pedidos com os componentes de Cliente removidos

Removendo as chamadas ao ClientDataSet

Nos eventos **OnCreate** e **OnDestroy** do formulário, abrimos e fechamos o ClientDataSet de Cliente, portanto, devemos removê-los também ficando assim:

```
procedure TfrmCadPedido.FormCreate(Sender: TObject);
begin
    dmCadPedido := TdmCadPedido.Create(Self);
    dmCadPedido.cdsPedido.FetchParams;
    dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').Value := NULL;
    dmCadPedido.cdsPedido.Open;
    dmCadPedido.cdsTipoPedido.Open;
end;
```

Testando a aplicação

Execute a aplicação, pesquise por um Pedido e ao abrí-lo, veremos que o Nome do Cliente será exibido no DBEdit.

Joins - Buscando o Cliente

Precisamos agora implementar uma forma de poder informar o Cliente no Pedido, já que não temos mais o Lookup.

Nesta primeira etapa teremos um campo onde o usuário informará o código do Cliente e depois de digitado, buscamos o respectivo nome. Depois implementaremos uma tela de pesquisa de Cliente parametrizada, onde o usuário poderá pesquisar pelo Cliente e depois de confirmado, alimentamos o campo no Pedido com o respectivo Código do Cliente selecionado.

Colocando em prática

Implementaremos uma forma de buscar o Nome após o campo Código do Cliente ter sido informado, portanto, a primeira coisa a ser feita é montar uma Query que busque o Nome do Cliente através de um valor passado por parâmetro, no qual este valor será alimentado com o Código do Cliente informado no cadastro do Pedido.

No DataModule **dmCadPedido** adicione um componente **SQLQuery**:



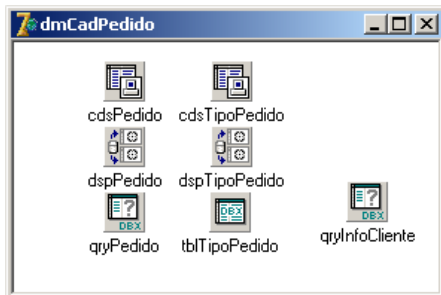
TSQLQuery

Name: qryInfoCliente

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  CLI_NOME
FROM
  CLIENTE
WHERE
  CLI_CODIGO = :CLI_CODIGO
```



DataModule de Cadastro de Pedidos com a Query qryInfoCliente

Seguindo o que já vimos, ajuste o parâmetro **CLI_CODIGO** que definimos na query da seguinte forma:

DataType: ftInteger

ParamType: ptInput

Determinando o evento a ser codificado

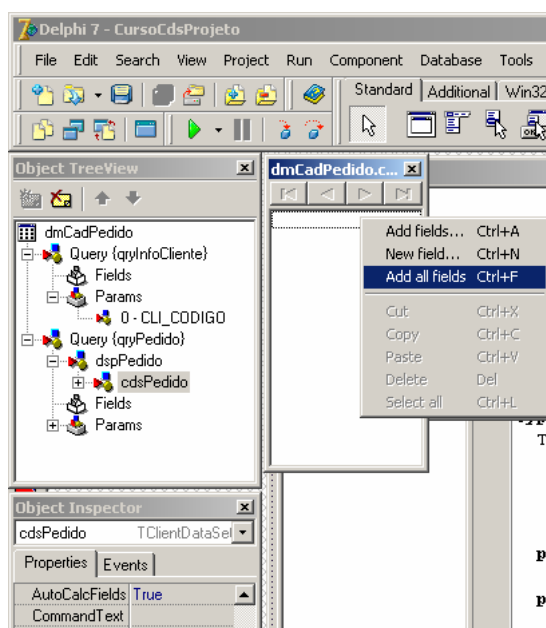
Precisamos determinar qual evento implementaremos para poder identificar a modificação do Código do Cliente e assim fazer uso da Query.

É muito comum fazer esses tipos de implementações no evento OnExit dos DBEdits. Não é um erro, depende muito do caso, por se tratar de uma regra de negócio, o local mais adequado seria no DataModule, no evento **OnValidate** do campo **CLI_CODIGO**.

Outro motivo muito importante também que nos leva a não codificar no evento OnExit é que, o mesmo será disparado somente se um novo componente receber o foco, por exemplo, quando saímos do campo. Se clicarmos no botão **Gravar** por exemplo, teremos o mesmo efeito, pois estamos utilizando um **TButton** e esta classe recebe foco, porém se utilizássemos um **TSpeedButton**, teríamos um problema, esta classe não recebe foco, portanto, o evento OnExit não seria disparado neste caso.

Implementando o Evento

Para implementarmos o evento **OnValidate** do campo, precisamos dele disponível no **FieldsEditor**, portanto, de um duplo clique no componente **cdsPedido** para que seja aberto o **FieldsEditor** e em seguida, clique com o botão direito do mouse e clique na opção **Add all fields**.



Adicionando os campos ao FieldsEditor

Feito isto, todos os campos estarão disponíveis no FieldsEditor. Selecione o campo **CLI_CODIGO**, acesse o evento **OnValidate** e adicione o seguinte código:

```
procedure TdmCadPedido.cdsPedidoCLI_CODIGOValidate(Sender: TField);
begin
  if not Sender.IsNull then
  begin
    qryInfoCliente.ParamByName('CLI_CODIGO').Value := Sender.Value;
    qryInfoCliente.Open;
    try
      if not qryInfoCliente.IsEmpty then
        cdsPedido.FieldByName('CLI_NOME').Value :=
          qryInfoCliente.FieldByName('CLI_NOME').Value
      else
        raise Exception.Create('Cliente não encontrado');
    finally
      qryInfoCliente.Close;
    end;
  end;
end;
```

Entendendo o código

Este evento nos fornece o parâmetro **Sender** que é do tipo TField e representa o campo que está sendo validado, no caso será o campo CLI_CODIGO. Utilizamos o objeto Sender para acessarmos o valor do campo e assim poder buscar o nome do cliente com base neste valor.

Executamos nossa sequência de código da seguinte forma:

Verificamos se o valor do campo não é nulo, não sendo, prosseguimos ajustando o valor do parâmetro **CLI_CODIGO** na **qryInfoCliente** com o mesmo valor do campo **CLI_CODIGO**, digitado no DBEdit. Em seguida abrimos nossa Query, verificamos se ela não está vazia, não estando, pegamos o valor do campo **CLI_NOME** obtido na **Query** e atribuímos ao campo **CLI_NOME** do **ClientDataSet de Pedido** (cdsPedido), assim teremos o nome do Cliente alimentado, conseqüentemente estando disponível no DBEdit. Caso a Query esteja vazia, geramos uma Exception para que o registro não seja gravado de forma alguma, barrando o campo até que informe um código válido ou limpe-o.

Ajustando o Formulário de Cadastro

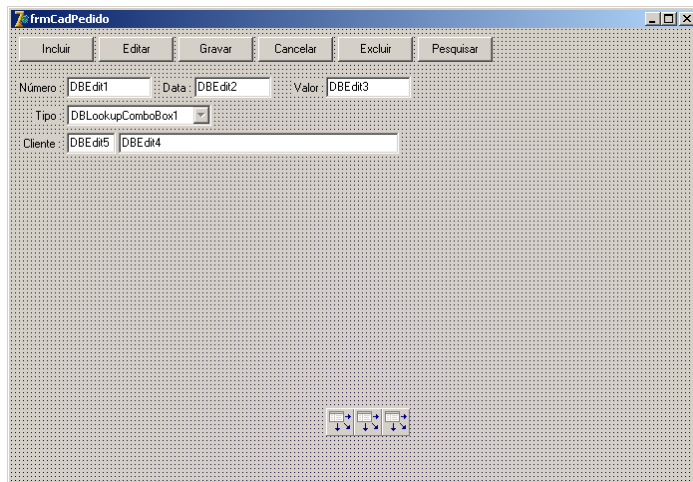
No Formulário de Cadastro adicione um DBEdit para o usuário poder informar o código do Cliente.



TDBEdit (Data Controls)

DataSource: dtsPedido

DataField: CLI_CODIGO



Cadastro de Pedidos – Incluindo campo DBEdit para o Código do Cliente

Testando a aplicação

Ao cadastrarmos um Pedido e informarmos o Código do Cliente, perceberemos que o Nome está sendo alimentado perfeitamente, porém, não conseguiremos gravar o registro, ao tentarmos fazer isto, o registro ficará em branco após a gravação.

Isso ocorre porque um erro está sendo gerado na gravação (no servidor de banco de dados) e nossa aplicação não está nos informando.

Para visualizarmos este erro, abra o DataModule **dmCadPedido** e implemente o evento **OnReconcileError** do **cdsPedido** com o seguinte código:

```
procedure TdmCadPedido.cdsPedidoReconcileError(  
  DataSet: TCustomClientDataSet; E: EReconcileError;  
  UpdateKind: TUpdateKind; var Action: TReconcileAction);  
begin  
  MessageDlg(E.Message, mtError, [mbOk], 0);  
end;
```

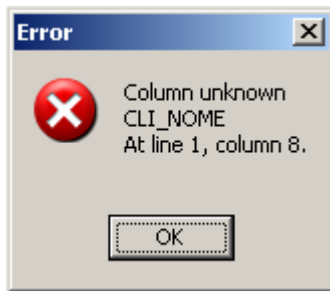
Este evento é gerado para cada registro que não pôde ser aplicado ao banco devido algum erro ocorrido, entraremos em detalhes mais adiante.

Adicione na cláusula **uses** a unit **Dialogs** (precisamos para o método MessageDlg).

```
...  
implementation
```

```
uses  
  udmPrincipal,  
  Dialogs;  
...
```

Podemos neste momento executar a aplicação, tentar incluir algum Pedido informando o Código do Cliente, conseqüentemente o Nome do Cliente será atualizado e ao tentarmos gravar, veremos o erro que está sendo gerado:



Erro ao gravar o Pedido

Ajustando o ProviderFlags

O erro está acontecendo porque faltou um passo a ser realizado, o ajuste dos **ProviderFlags**, mas antes de fazermos isso, vamos entender o motivo do erro.

O Provider está gerando uma query de inclusão utilizando todos os campos que fizemos o SELECT, portanto está sendo executada uma query da seguinte forma:

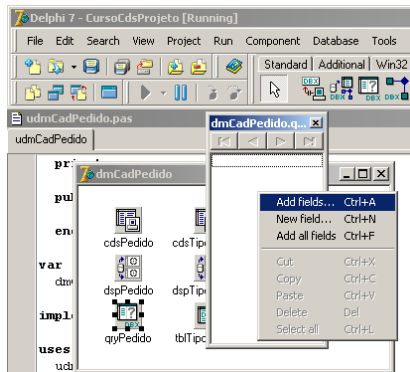
```
INSERT INTO PEDIDO (PED_NUMERO, PED_DATA, PED_VALOR, CLI_CODIGO, CLI_NOME) VALUES (...)
```

Perceba que o campo **CLI_NOME** está incluso na query, e isso não pode ocorrer, pois ele não pertence à tabela de PEDIDO e sim à tabela de CLIENTE.

O que precisamos fazer é informar ao Provider para não incluir este campo na atualização.

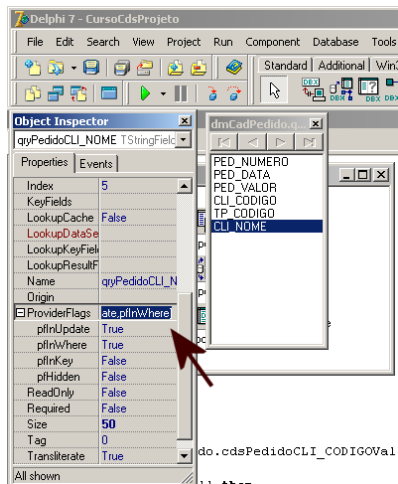
Para isso, precisamos ajustar a propriedade **ProviderFlags** do campo **CLI_NOME** na **qryPedido**. **Sempre fazemos este ajuste na Query e não no ClientDataSet.**

Abra o **FieldsEditor** do componente **qryPedido**, em seguida adicione todos os campos clicando com o botão direito do mouse e depois em **Add all fields**.



Adicionando todos os campos no componente qryPedido

Selecione o campo **CLI_NOME** e vamos analisar a propriedade **ProviderFlags**.



Acessando propriedade ProviderFlags do campo CLI_NOME

Esta propriedade nos disponibiliza as seguintes opções:

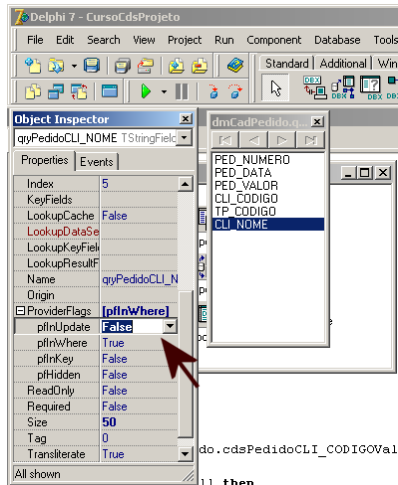
pflnUpdate: Indica se o campo será incluído na cláusula INSERT/UPDATE

pflnWhere: Indica se o campo será incluído na cláusula WHERE

pflnKey: Indica se o campo pertence à chave primária. Usado em conjunto com a propriedade UpdateMode do Provider na qual veremos mais adiante.

pfHidden: Indica que a coluna será incluída no DataPacket com um único objetivo de localização correta do registro.

Por padrão temos as opções **pflnUpdate** e **pflnWhere** ligadas nesta propriedade. O que faremos é **desligar** a opção **pflnUpdate** para que o campo não seja incluído na cláusula INSERT/UPDATE da query gerada pelo Provider, para isso, basta defini-la para **False**.



Desligando a opção pflnUpdate da propriedade ProviderFlags do campo CLI_NOME

Testando a aplicação

Na inclusão de um Pedido, perceberemos que o erro realmente desapareceu, porém se tentarmos editar e modificar o Código do Cliente, conseqüentemente o Nome do Cliente será atualizado e ao tentarmos gravar o Pedido, o erro aparecerá novamente.

Neste momento surge a grande questão:

- Se na inclusão o problema foi resolvido e na alteração o erro continua, significa que o ajuste que fizemos funciona apenas para tirar o campo da cláusula INSERT e não da cláusula UPDATE?

O erro até é pelo fato de estarmos alterando ao invés de inserindo, porém não está ligado diretamente à cláusula UPDATE, mas sim à cláusula WHERE.

Quando modificamos nosso registro, o Provider está gerando uma query da seguinte forma:

```
UPDATE PEDIDO SET CAMPO=VALOR... WHERE PED_NUMERO=? AND PED_DATA=?  
AND PED_VALOR=? AND CLI_CODIGO=? AND CLI_NOME=?
```

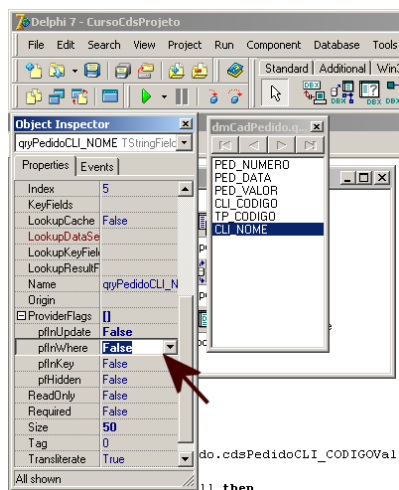
Na cláusula SET terá apenas os campos que foram modificados e que estejam com a opção **pflnUpdate** ligada na propriedade **ProviderFlags**. Em nosso caso, mesmo que o campo **CLI_NOME** tenha sido modificado, ele não será incluso, pois está com a opção **pflnUpdate** desligada.

Já na cláusula WHERE estarão todos os campos, onde os caracteres ? serão substituídos pelos valores originais dos campos para que o registro possa ser localizado e atualizado.

Podemos perceber que o campo **CLI_NOME** está incluso na cláusula WHERE, e é isto que está causando o erro. Precisamos informar ao Provider que este campo não poderá pertencer também a esta cláusula.

Para isto, vamos seguir os mesmos passos que fizemos com a opção **pflnUpdate**, porém agora desligaremos a opção **pflnWhere**.

Dê um duplo clique no componente **qryPedido**, selecione o campo **CLI_NOME** e em seguida defina **False** para opção **pflnWhere** da propriedade **ProviderFlags**.



*Desligando a opção **pflnWhere** da propriedade **ProviderFlags** do campo **CLI_NOME***

Testando a aplicação

Podemos executar e fazer os testes de inclusão, alteração e exclusão que o erro não será mais exibido.

Ajustando UpdateMode do Provider - Performance na Atualização e Controle de Concorrência

Neste momento, nossa aplicação está funcionando perfeitamente, porém, temos que nos atentar a uma propriedade importante existente no Provider que influenciará no controle de concorrência de usuários atualizando o mesmo registro e ganho de performance nas atualizações.

Como já explicamos, quando atualizamos nosso registro, internamente o Provider gera uma Query de atualização onde usará todos os campos (que estiverem com a opção pflnWhere do Provider ligada) na cláusula WHERE para poder localizar o registro.

Isso pode gerar uma queda de performance quando atualizarmos uma tabela com muitos registros, pois é pouco provável que tenhamos um índice na tabela composto por todos os campos utilizados na cláusula WHERE a fim de ganhar performance na atualização.

Além da questão de performance, temos também o lado da 'concorrência' de atualização do registro, ou seja, vários usuários atualizando o mesmo registro.

Vamos citar um exemplo para entendermos melhor:

Dois usuários entraram no Pedido cuja situação atual é:

Número: 120
Data: 01/01/1900
Valor: 200,00
Código do Cliente: 5

O primeiro usuário faz a alteração no Pedido modificando apenas a data para 01/01/1920 e logo em seguida grava, portanto o Provider irá gerar a seguinte query de atualização:

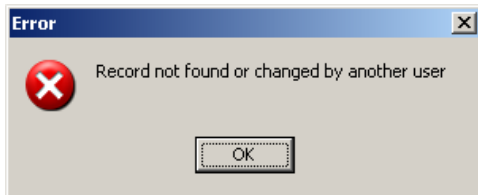
```
UPDATE PEDIDO SET PED_DATA = 01/01/1920 WHERE PED_NUMERO = 120 AND  
PED_DATA = 01/01/1900 AND PED_VALOR = 200 AND CLI_CODIGO = 5
```

Após a execução desta query, o registro será localizado e atualizado.

Enquanto isso, o segundo usuário estava com o registro aberto, ele ainda não sabe que houve modificações no registro, somente saberá caso abra-o novamente ou execute um refresh. Porém nada disto foi feito e o usuário alterou o campo Código do Cliente de 5 para 6 e gravou, portanto a seguinte query será gerada:

```
UPDATE PEDIDO SET CLI_CODIGO = 6 WHERE PED_NUMERO = 120 AND PED_DATA =  
01/01/1900 AND PED_VALOR = 200 AND CLI_CODIGO = 5
```

Analisando esta query, veremos que a condição que está sendo montada é exatamente igual à condição montada para o primeiro usuário, pois ambos abriram o registro na mesma situação. Só que neste momento não existe mais o registro nesta condição **WHERE ... AND PED_DATA = 01/01/1900** ... o mesmo já foi alterado pelo primeiro usuário, a data foi modificada, portanto, ao tentarmos gravar o registro, teremos a seguinte mensagem de erro:



Erro ao tentarmos gravar um registro modificado por outro usuário

Agora vem a grande questão:

Como controlarmos isso?

A primeira idéia que surgiria seria de 'travar' o registro na edição. Quando trabalhamos com TTable/Paradox, isso é feito de forma automática, porém o mesmo não ocorre em bancos de dados Cliente/Servidor. Dependendo do banco, até podemos fazer isso, porém é um tanto trabalhoso e não tão recomendado, pois teríamos que manter a transação aberta até que o usuário finalize-a, particularmente recomendaria somente em casos de extrema necessidade.

Da forma que esta o projeto, o segundo usuário não poderia gravar o registro, até poderíamos fazer o tratamento do erro, exibindo uma tela com os dados do registro atual, informando ao usuário de que o mesmo já foi modificado e ali, permitir que se faça alguns ajustes para poder atualizar o registro.

Uma alternativa interessante para contornarmos este problema é ajustar o Provider de modo que ele utilize apenas o campo chave na cláusula WHERE, pois desta forma sempre o registro será localizado, já que o campo chave jamais será modificado. O registro só não será encontrado caso o mesmo tenha sido excluído.

Com esta alternativa, além de resolvermos esta questão, ganhamos performance, pois será feito um WHERE somente no campo chave e já existe um índice para este campo, devido ao fato de ser chave primária.

Outra questão importante é que, utilizando esta forma, sempre prevalecerá às alterações feitas pelo último usuário, substituindo as já realizadas por outros usuários.

Então para fazermos este ajuste, temos dois caminhos:

- Selecionar todos os campos da Query (qryPedido) e ajustar a propriedade ProviderFlags desligando a opção pfInWhere e apenas deixar ativado no campo chave (PED_NUMERO).

Desta forma estamos indicando que apenas o campo chave irá para a cláusula WHERE, resolvendo assim nosso problema. Esta opção tem um inconveniente, todo novo campo que acrescentarmos no SELECT, seremos obrigados a ajustar a propriedade ProviderFlags para que não seja incluso na cláusula WHERE. A opção a seguir é a melhor alternativa, pois resolve esta questão.

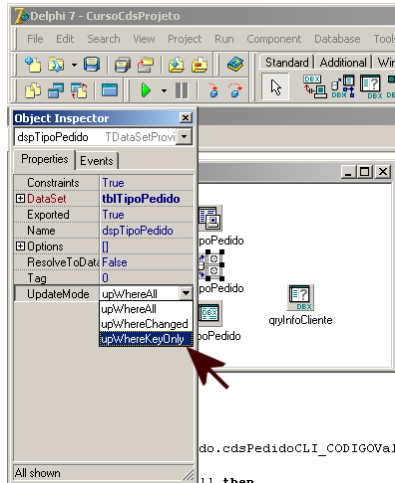
- Ajustar a propriedade UpdateMode do Provider para upWhereKeyOnly e em seguida, ajustar a propriedade ProviderFlags do campo chave PED_NUMERO ligando a opção pfInKey.

Quando ajustamos a propriedade UpdateMode do Provider para upWhereKeyOnly, estamos indicando para o Provider incluir somente o campo chave na cláusula WHERE, portanto temos que indicar qual seria nosso campo chave, fazemos isso ligando a opção **pfInKey** na propriedade **ProviderFlags** do campo chave. Desta forma podemos incluir novos campos sem nos preocuparmos em ajustar a propriedade ProviderFlags, pois como vimos, este ajuste é feito apenas no campo chave.

Optaremos então pela segunda opção.

Colocando em prática

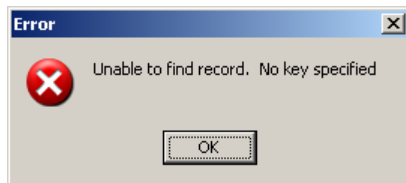
No DataModule **dmCadPedido** selecione o componente **dspPedido** e ajuste a propriedade **UpdateMode** para **upWhereKeyOnly**.



Ajustando propriedade UpdateMode do Provider para upWhereKeyOnly

Testando a aplicação

Pesquisa por um Pedido, faça alguma alteração e ao gravar, veremos a seguinte mensagem de erro:

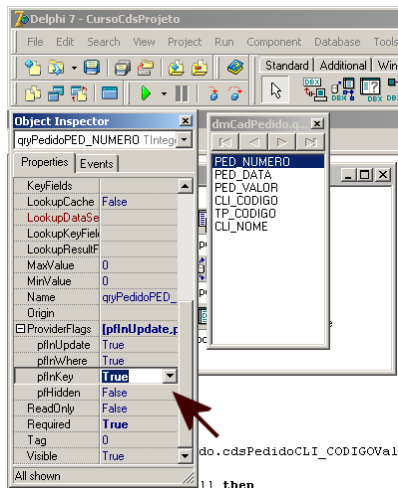


Erro ao tentar gravar uma modificação feita no Cadastro de Pedidos

Isto ocorre pois como havíamos comentando, precisamos informar ao Provider qual é a chave para que ele possa utilizar na cláusula WHERE.

Definindo a chave

Abra o **FieldsEditor** do componente **qryPedido** e selecione o campo **PED_NUMERO**. Na propriedade **ProviderFlags**, ligue a opção **pflnKey**.



Ligando opção *pfInKey* na propriedade *ProviderFlags* do campo *PED_NUMERO*

Testando a aplicação

Podemos testar a aplicação fazendo os testes de concorrência e perceberemos que o erro não mais ocorrerá e conseqüentemente temos um ganho de performance na atualização pelo fato de estarmos utilizando apenas o campo chave como meio de localização do registro.

Trabalhando com Mestre/Detalhe

O ClientDataSet nos possibilita trabalhar com relacionamento mestre/detalhe de três formas:

- Carregar todos os registros detalhes e utilizar as propriedades MasterSource e MasterFields para definir o relacionamento e assim filtrar os registros localmente.

Este modelo é semelhante ao que estamos acostumados a trabalhar com TTable, onde utilizamos as propriedades MasterSource e MasterFields para definir o relacionamento, porém devemos tomar cuidado pois no caso de tabelas detalhes muito grandes, teremos um tráfego de registros intenso, já que todos serão enviados e o filtro será aplicado no cliente.

O inconveniente deste modelo é que precisamos manualmente abrir o Detail após a abertura do Master e chamar o método ApplyUpdates após termos executado no Master, o mesmo vale para o método CancelUpdates.

- Carregar somente os registros detalhes relativos ao registro master e utilizar as propriedades MasterSource e MasterFields para definir o relacionamento.

Este modelo é um pouco semelhante ao anterior, porém evitamos que todos os registros detalhes sejam trafegados pela rede, limitamos isso na cláusula SQL da Query Detail, onde fazemos uma condição que carregue somente os registros detalhes relativos ao master.

Este modelo possui também o mesmo inconveniente descrito na primeira opção. Além disto, temos sempre que ficar atentos aos filtros (cláusula where do SQL), pois dependendo do filtro aplicado na Query Master e Detail, qualquer mudança na Master poderá implicar em modificar o filtro da Query Detail para que sempre trafegue somente os registros relativos ao Master.

- NestedDataSet

Este é a forma mais utilizada. Neste modelo definimos o relacionamento diretamente no DataSet ao qual o Provider está ligado, ou seja, no servidor de aplicação quando trabalhamos no modelo multicamadas, diferente do que ocorre nos outros modelos, onde o relacionamento é definido no cliente (ClientDataSet).

Trabalhando desta forma, definimos no SQL detail uma condição para que somente os registros relativos ao master sejam trafegados.

Quando abrimos o Master, o Detail é aberto automaticamente e o mais importante, os registros Details são inclusos no mesmo pacote do Master. Com isso, temos uma outra vantagem, as atualizações são feitas na mesma transação, chamamos o método ApplyUpdates somente no Master, e os registros Details são atualizados automaticamente, o mesmo vale para o método CancelUpdates, executamos apenas no Master.

Aplicaremos as 3 formas a fim de visualizarmos na prática as diferenças.

Para facilitar criaremos 2 cópias do projeto atual, onde na primeira cópia simularemos as 2 formas de relacionamento utilizando mastersource/masterfields e na segunda cópia, utilizaremos o NestedDataset.

Então salve todo o projeto atual e faça uma cópia para as seguintes pastas:

- C:\CursoClientDataSet\projeto\src_md_mastersource
- C:\CursoClientDataSet\projeto\src_md_nesteddataset

Feito isso, teremos 3 versões do projeto, a primeira não será mais utilizada, continuará na pasta **src** e trabalharemos nas outras cópias criadas.

Mestre/Detalhe - Utilizando MasterSource/MasterFields (trafegando todos os registros detalhes)

Primeiramente abra o projeto da pasta **src_md_mastersource**.

No datamodule **dmCadPedido**, adicione mais um conjunto dos 3 componentes:



TClientDataSet (Data Access)

Name: cdsPedItem

ProviderName: dspPedItem



TDataSetProvider (Data Access)

Name: dspPedItem

DataSet: qryPedItem



TSQLQuery (dbExpress)

Name: qryPedItem

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  PED_NUMERO,
  PROD_CODIGO,
  PI_DESCRICAO,
  PI_QTDE,
  PI_VALUNIT,
  PI_VALTOTAL
FROM
  PEDITEM
```

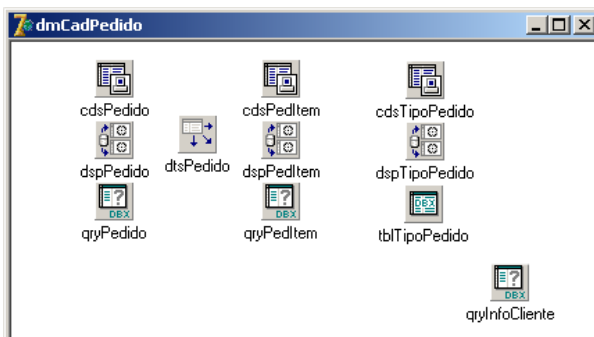
Por enquanto não utilizaremos nenhuma condição, mas adiante faremos o ajuste para trazermos somente os registros relativos ao master para reduzirmos o tráfego de informações.



TDataSource (Data Access)

Name: dtsPedido

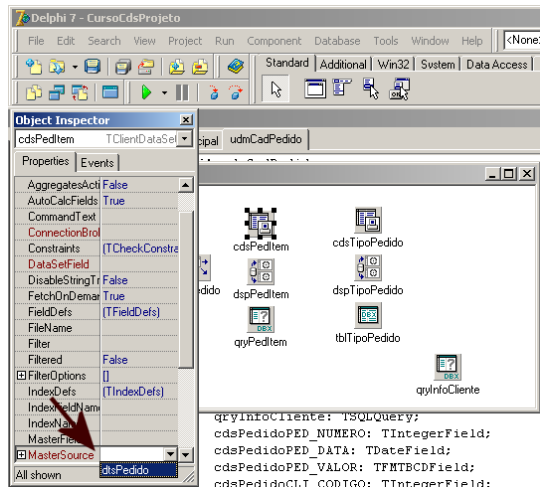
DataSet: cdsPedido



Datamodule de Pedido com a tabela detalhe

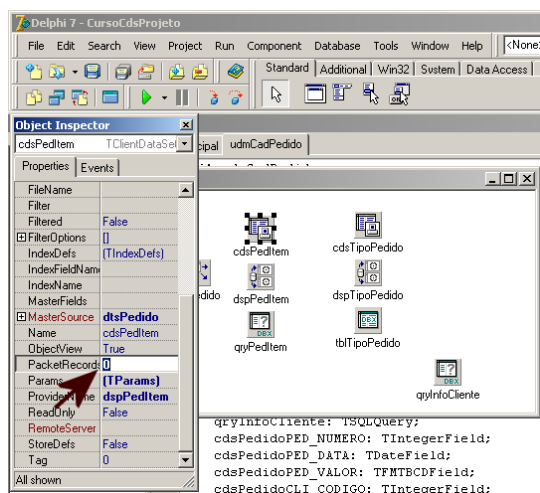
Desenvolvendo uma aplicação utilizando ClientDataSet com DBExpress e Firebird

Agora podemos trabalhar com as propriedades **MasterSource** e **MasterFields** do componente **cdsPedItem**, da mesma forma que estamos acostumados a fazer com a TTable. Selecione o componente **cdsPedItem** e ajuste a propriedade **MasterSource** para **dtsPedido**.



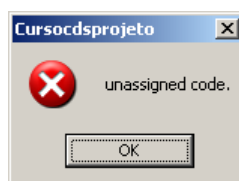
Definindo o MasterSource do cdsPedItem

Depois de associado o DataSource na propriedade MasterSource, perceberemos que a propriedade **PacketRecords** do **cdsPedItem** foi alterada para **0 (zero)**.



Propriedade PacketRecords definida como Zero automaticamente

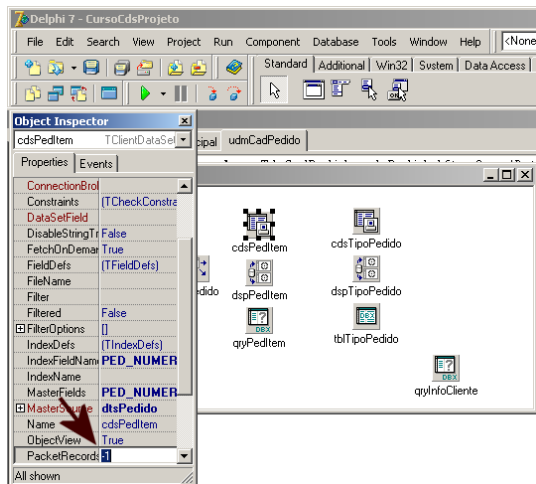
Isto significa que nenhum registro será enviado, apenas o metadado, neste caso se tentássemos abrir o **cdsPedItem**, seria exibida a seguinte mensagem de erro:



Erro ao abrir o cdsPedItem com a propriedade PacketRecords igual a Zero

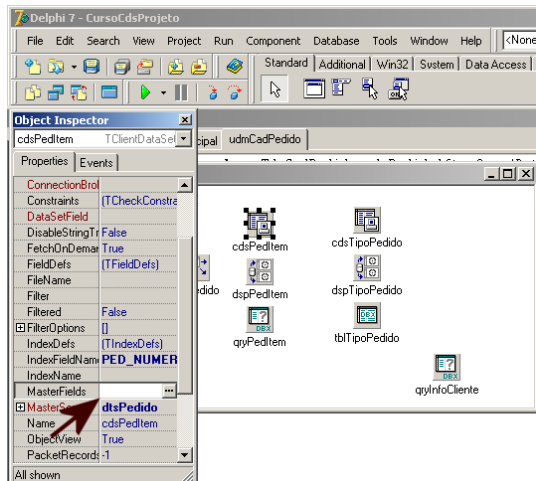
Para corrigirmos isto, basta ajustarmos a propriedade para seu valor original (**-1**), isto indica que todos registros detalhes serão enviados.

Desenvolvendo uma aplicação utilizando ClientDataSet com DBExpress e Firebird



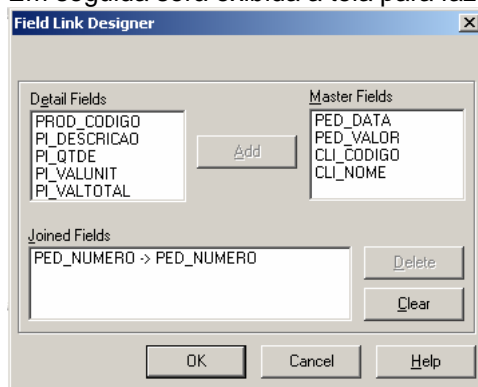
Voltando o valor original da propriedade PacketRecord (-1)

Clique na propriedade **MasterFields** para definirmos o relacionamento:



Acessando propriedade MasterFields

Em seguida será exibida a tela para fazermos o relacionamento, defina-o da seguinte forma:



Definindo os campos do relacionamento Mestre/Detalhe

Ajustando o Formulário de Cadastro

Abra o formulário **frmCadPedido** e adicione os seguintes componentes:



TDataSource

Name: dtsPedItem

DataSet: dmCadPedido.cdsPedItem



TDBGrid (Data Controls)

Name: dbgrdItens

DataSource: dtsPedItem

Formulário de Cadastro de Pedidos com Itens

Abrindo a tabela Detalhe

Precisamos abrir a tabela de Itens após a abertura da tabela de Pedidos, já que isso não é feito automaticamente neste modelo, faremos isso no evento **OnCreate** do formulário e no botão **Pesquisar**:

Evento OnCreate do Formulário:

```
procedure TfrmCadPedido.FormCreate(Sender: TObject);
begin
    dmCadPedido := TdmCadPedido.Create(Self);
    dmCadPedido.cdsPedido.FetchParams;
    dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').Value := NULL;
    dmCadPedido.cdsPedido.Open;
    dmCadPedido.cdsPedItem.Open;
    dmCadPedido.cdsTipoPedido.Open;
end;
```

Evento OnClick do Botão Pesquisar:

```
procedure TfrmCadPedido.btnPesquisarClick(Sender: TObject);
var
  sNumero: string;
begin
  if InputQuery('Pesquisa de Pedido', 'Informe o Número', sNumero) then
  begin
    dmCadPedido.cdsPedido.Close;
    dmCadPedido.cdsPedido.Close;
    dmCadPedido.cdsPedido.FetchParams;
    dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').AsInteger :=
      StrToInt(sNumero);
    dmCadPedido.cdsPedido.Open;
    dmCadPedido.cdsPedido.Open;
    if dmCadPedido.cdsPedido.IsEmpty then
      ShowMessage('Pedido não encontrado !');
    end;
  end;
end;
```

Aplicando as atualizações nos registros detalhes

Da mesma forma que fizemos manualmente a abertura dos dados na tabela de Itens, precisamos também executar manualmente o método **ApplyUpdates** após gravar ou excluir o Pedido, portanto faremos este ajuste nos botões **Gravar** e **Excluir**:

Botão Gravar:

```
procedure TfrmCadPedido.btnGravarClick(Sender: TObject);
begin
  dmCadPedido.cdsPedido.Post;
  if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
    dmCadPedido.cdsPedido.CancelUpdates;

  if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
    dmCadPedido.cdsPedido.CancelUpdates;
end;
```

Botão Excluir:

```
procedure TfrmCadPedido.btnExcluirClick(Sender: TObject);
begin
  while not dmCadPedido.cdsPedido.IsEmpty do
    dmCadPedido.cdsPedido.Delete;

  dmCadPedido.cdsPedido.Delete;

  if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
    dmCadPedido.cdsPedido.CancelUpdates;

  if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
    dmCadPedido.cdsPedido.CancelUpdates;
end;
```

Entendendo o código

Na primeira linha estamos excluindo todos registros detalhes do respectivo pedido. Em seguida, excluimos o Pedido. Feito isto, aplicamos tudo ao banco de dados, porém seguindo uma ordem lógica: Primeiro os registros detalhes e depois o master.

Fazemos nesta ordem para não ocorrer erro de integridade, pois se aplicássemos primeiro o registro master, o mesmo tentaria ser excluído do banco, e temos registros detalhes dependentes, portanto, um erro de integridade seria gerado pelo servidor de banco de dados.

Cancelando as pendências nos registros detalhes

Da mesma forma que chamamos o método ApplyUpdates no ClientDataSet de Itens, temos também que executar o método CancelUpdates ao cancelar o Pedido, portanto ajuste o código do botão **Cancelar** da seguinte forma:

```
procedure TfrmCadPedido.btnCancelarClick(Sender: TObject);  
begin  
    dmCadPedido.cdsPedido.Cancel;  
    dmCadPedido.cdsPedido.CancelUpdates;  
    dmCadPedido.cdsPedItem.CancelUpdates;  
end;
```

Testando a aplicação

Execute a aplicação e cadastre um Pedido com os respectivos itens, em seguida grave-os e pesquise o Pedido para checar se os dados foram gravados com sucesso.

Mestre/Detalhe - Utilizando MasterSource/MasterFields (filtrando os registros detalhes)

Com o mesmo projeto, o que faremos agora é um ajuste na query dos registros detalhes para que não sejam trafegados todos os registros, mas sim somente aqueles pertencentes ao master.

Ajuste o script SQL do componente **qryPedItem** para que fique da seguinte forma:

```
SELECT
  PED_NUMERO,
  PROD_CODIGO,
  PI_DESCRICAO,
  PI_QTDE,
  PI_VALUNIT,
  PI_VALTOTAL
FROM
  PEDITEM
WHERE
  PED_NUMERO = :PED_NUMERO
```

A diferença agora é que incluímos o parâmetro **PED_NUMERO**, alimentaremos da mesma forma que alimentamos o parâmetro no Pedido, pois desta forma, teremos apenas os itens do respectivo Pedido sendo trafegado.

Depois de criado, ajuste este parâmetro da seguinte forma:

DataType: dtInteger

ParamType: ptInput

Agora nos eventos onde alimentamos o parâmetro PED_NUMERO do Pedido, alimentaremos também este parâmetro que criamos.

No evento OnCreate do Formulário:

```
procedure TfrmCadPedido.FormCreate(Sender: TObject);
begin
  dmCadPedido := TdmCadPedido.Create(Self);
  dmCadPedido.cdsPedido.FetchParams;
  dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').Value := NULL;
  dmCadPedido.cdsPedItem.FetchParams;
  dmCadPedido.cdsPedItem.Params.ParamByName('PED_NUMERO').Value := NULL;
  dmCadPedido.cdsPedido.Open;
  dmCadPedido.cdsPedItem.Open;
  dmCadPedido.cdsTipoPedido.Open;
end;
```

No evento Botão Pesquisar:

```
procedure TfrmCadPedido.btnPesquisarClick(Sender: TObject);
var
  sNumero: string;
begin
  if InputQuery('Pesquisa de Pedido', 'Informe o Número', sNumero) then
  begin
    dmCadPedido.cdsPedido.Close;
    dmCadPedido.cdsPedItem.Close;
    dmCadPedido.cdsPedido.FetchParams;
    dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').AsInteger :=
StrToInt(sNumero);
    dmCadPedido.cdsPedItem.FetchParams;
    dmCadPedido.cdsPedItem.Params.ParamByName('PED_NUMERO').AsInteger :=
StrToInt(sNumero);
    dmCadPedido.cdsPedido.Open;
    dmCadPedido.cdsPedItem.Open;
    if dmCadPedido.cdsPedido.IsEmpty then
      ShowMessage('Pedido não encontrado !');
  end;
end;
```

Testando a aplicação

De modo geral não veremos diferença na aplicação, porém vale lembrar que agora estamos restringindo o os registros detalhes, antes carregávamos todos os registros e aplicávamos um filtro localmente, agora somente os registros relativos ao Pedido estão sendo trafegados, ou seja, o filtro (WHERE) é aplicado no servidor. Se tivéssemos uma tabela de Itens grande, perceberíamos a diferença no tempo de abertura desta tabela.

Mestre/Detalhe - Utilizando NestedDataSet

Abra o projeto da pasta **src_md_nestdeddataset** conforme já havíamos feito a cópia.

No DataModule **dmCadPedido** adicione os seguintes componentes:



TClientDataSet (Data Access)

Name: cdsPedItem



TSQLQuery (dbExpress)

Name: qryPedItem

DataSource: dtsPedido

SQLConnecton: dmPrincipal.SqlConnPrincipal

SQL:

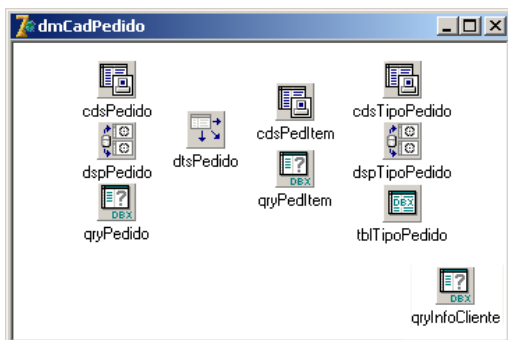
```
SELECT
  PED_NUMERO,
  PROD_CODIGO,
  PI_DESCRICAO,
  PI_QTDE,
  PI_VALUNIT,
  PI_VALTOTAL
FROM
  PEDITEM
WHERE
  PED_NUMERO = :PED_NUMERO
```



TDataSource (Data Access)

Name: dtsPedido

DataSet: qryPedido



DataModule de Pedido com a tabela detalhe

Neste momento já percebemos uma diferença, **não temos o Provider para o ClientDataSet de Itens**, isto ocorre mesmo, o Provider da tabela de Pedido será responsável por tudo.

Notamos outra grande diferença na propriedade **DataSet** do **DataSource**, ligamos com a Query e não com o ClientDataSet, e também utilizamos a propriedade **DataSource da Query Detalhe**. Definimos isso pois faremos relacionamento mestre/detalhe entre queries, logo adiante veremos a regras e detalhes sobre isso.

Analisando o SQL da Query detalhe, percebemos que não há diferenças no que vimos até agora, porém o parâmetro PED_NUMERO não será alimentado por nós, mas sim automaticamente pela Query mestre (qryPedido), pois como dissemos, estamos fazendo um relacionamento mestre/detalhe entre elas.

Relacionamento Mestre/Detalhe entre as Querys

Para fazermos um relacionamento mestre/detalhe entre Querys, precisamos:

- Definir o DataSource da Query Detalhe

Semelhante a propriedade MasterSource de uma TTable, temos a propriedade DataSource na Query, na qual devemos ligar com o DataSource que está ligado com a Query mestre.

- Seguir algumas regras na montagem do SQL

Na Query detalhe, temos que montar uma condição que retorne somente os registros relativos ao registro master, por isso usamos a condição:

```
WHERE PED_NUMERO = :PED_NUMERO
```

Comentamos que o parâmetro é alimentando automaticamente, para que isso aconteça, precisamos definir seu nome exatamente igual ao nome do campo chave da tabela master. Em nosso caso temos o campo chave chamado PED_NUMERO, por este motivo criamos o nome do parâmetro com este mesmo nome.

Seguindo as regras descritas acima, quando a query detalhe for aberta (isto é feito automaticamente pela query master), ela perceberá que está trabalhando como detalhe, pois a propriedade DataSource está alimentada, então verificará todos os parâmetros existentes e para cada um, tentará procurar o campo com mesmo nome na query Master, achando, alimentará o parâmetro com o valor do campo encontrado.

Agora precisamos ajustar o parâmetro **PED_NUMERO** da Query Detail, portanto clique na propriedade **Params**, selecione o parâmetro e ajuste as seguintes propriedades:

DataType: ftInteger

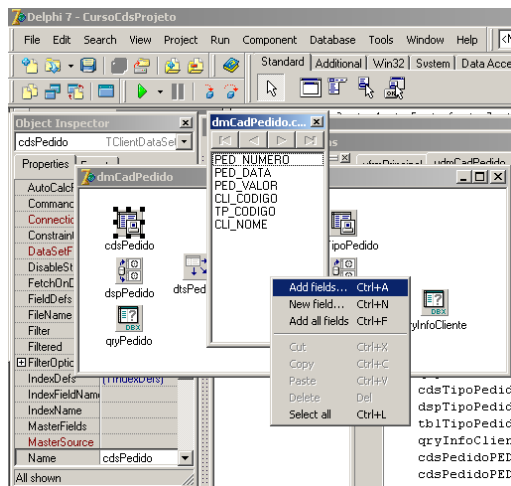
ParamType: ptInput

Um novo conceito, TDataSetField

Pode parecer estranho, mas todos os registros detalhes serão enviados para o ClientDataSet mestre em forma de um campo, é um campo do tipo **TDataSetField**, ou seja, é um DataSet, para cada registro mestre, teremos este campo preenchido com todos os registros detalhes.

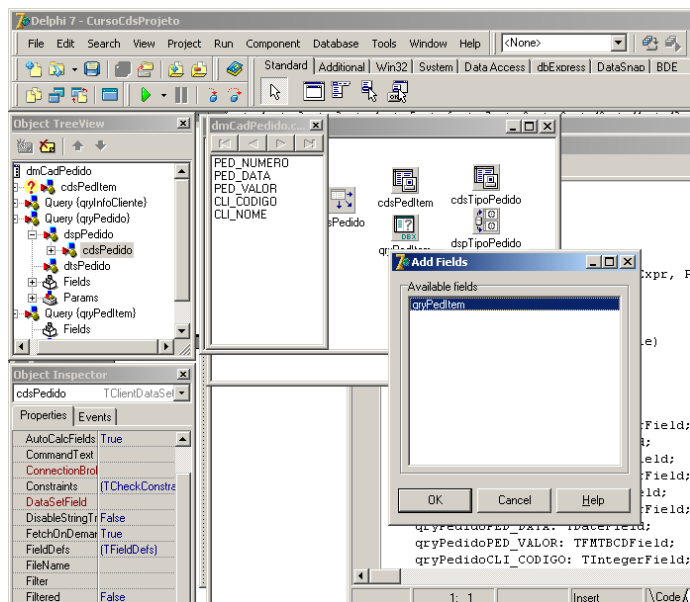
Depois de feito este relacionamento, teremos este campo disponível, para isso precisamos atualizar o FieldsEditor do ClientDataSet Master.

Abra o **FieldsEditor** do **cdsPedido** e clique com o botão direito na janela, em seguida clique em **Add Fields...**



Atualizando FieldsEditor do cdsPedido com o novo campo do tipo TDataSetField

O novo campo estará disponível na tela seguinte.



Adicionando o campo que representará os registros detalhes

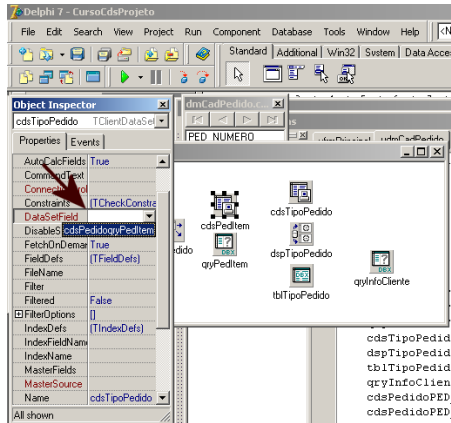
O nome do campo possui o mesmo nome da query detalhe. Confirmando, ele estará disponível na lista de campos, pronto para ser usado.

Utilizando o TDataSetField

Como este campo representa um DataSet, precisamos de algum componente que nos permita manipular os dados contidos nele, então utilizaremos o próprio ClientDataSet, por isso temos nosso **cdsPedItem**, que até o momento não havíamos utilizado, agora utilizaremos para esta finalidade.

Para definirmos que o **cdsPedItem** manipulará este campo, precisamos associá-lo ao campo, fazemos isso através da sua propriedade **DataSetField**. Acessando esta propriedade teremos disponível um objeto chamado **cdsPedidoqryPedItem** que é o campo que acabamos de inserir no **FieldsEditor**.

Desenvolvendo uma aplicação utilizando ClientDataSet com DBExpress e Firebird



Acessando propriedade DataSetField do componente cdsPedItem

Defina o valor desta propriedade para **cdsPedidoqryPedItem** e o ClientDataSet detalhe está pronto para ser utilizado.

Ajustando o Formulário de Cadastro

No formulário **frmCadPedido** adicione os seguintes componentes:



TDataSource

Name: dtsPedItem

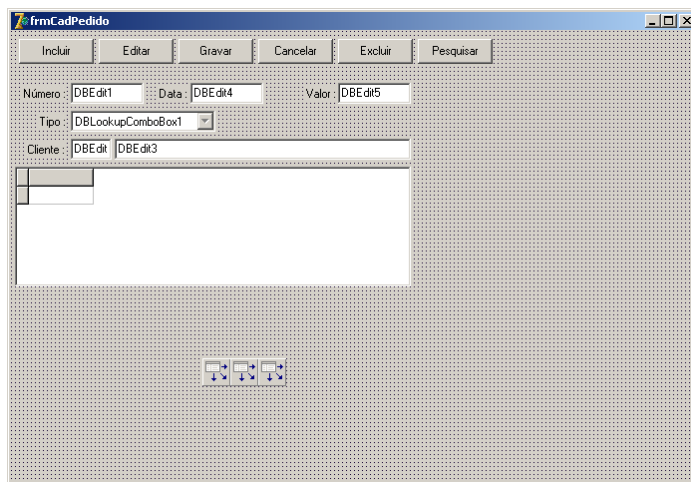
DataSet: dmCadPedido.cdsPedItem



TDBGrid

Name: dbgrdItens

DataSource: dtsPedItem



Formulário de Cadastro de Pedidos com Itens

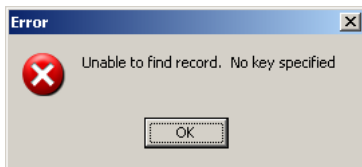
Desenvolvendo uma aplicação utilizando ClientDataSet com DBExpress e Firebird

No modelo anterior precisamos codificar diversos pontos do sistema devido à inclusão do ClientDataSet Detalhe, como havíamos comentando, neste modelo tudo é controlado pelo mestre, apenas precisamos codificar o botão Excluir para que os registros detalhes sejam excluídos também.

```
procedure TfrmCadPedido.btnExcluirClick(Sender: TObject);
begin
    while not dmCadPedido.cdsPedItem.IsEmpty do
        dmCadPedido.cdsPedItem.Delete;
    dmCadPedido.cdsPedido.Delete;
    if dmCadPedido.cdsPedido.ApplyUpdates(0) <> 0 then
        dmCadPedido.cdsPedido.CancelUpdates;
end;
```

Testando a aplicação

Faça o cadastro de um Pedido e grave, perceba que o mesmo foi aplicado com sucesso, porém ao tentarmos alterar um registro detalhe e gravar o pedido, veremos a seguinte mensagem de erro:

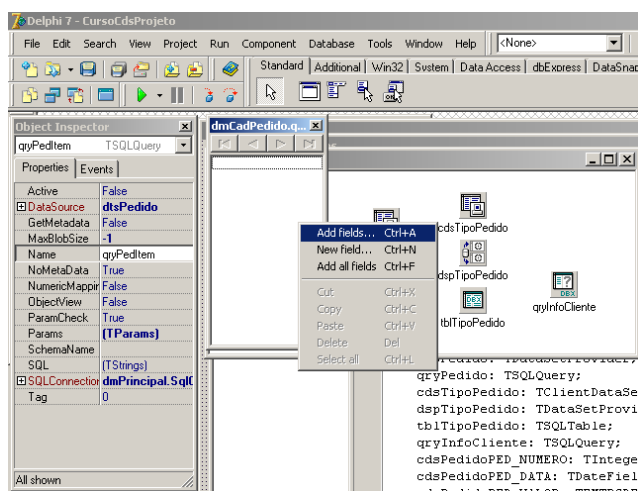


Mensagem de erro ao tentar gravar o pedido depois de modificado algum item

Isto ocorre por que o Provider está com a propriedade **UpdateMode** ajustada para **upWhereKeyOnly**, como vimos, isto determina que somente os campos chaves irão para cláusula WHERE. No componente **qryPedido** já havíamos definido quais são os campos chaves através do **ProviderFlags**, agora precisamos fazer o mesmo para a query detalhe (**qryPedItem**).

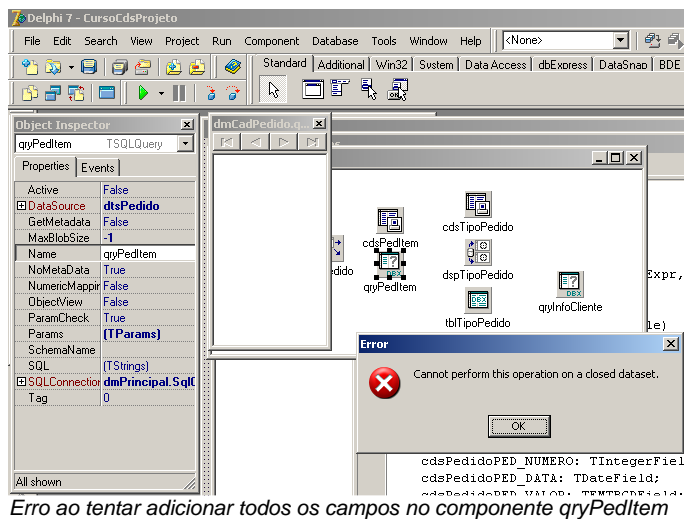
Ajustando o ProviderFlags da Query Detalhe

Dê um duplo clique no componente **qryPedItem** para acessar o **FieldsEditor** e em seguida, clique com o botão direito do mouse acessando o item de menu **Add all fields**.



Adicionando os campos a qryPedItem

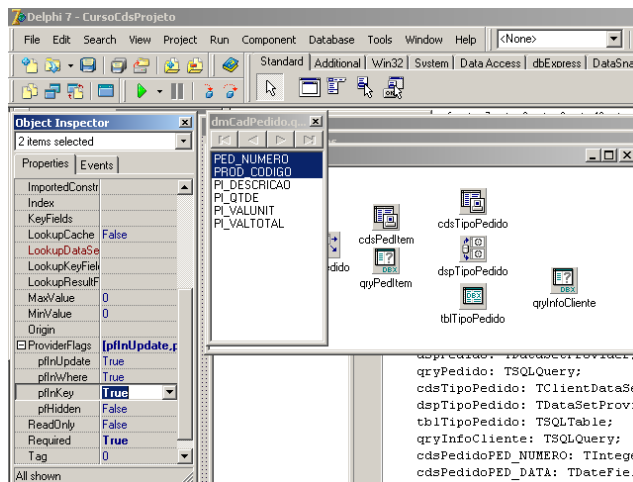
Ao clicarmos no item **Add Fields** veremos a seguinte mensagem de erro:



Erro ao tentar adicionar todos os campos no componente qryPedItem

Isto ocorre pois quando pedimos para adicionar os campos no FieldsEditor, a query está sendo aberta internamente, e por ela estar em um relacionamento mestre/detalhe, exige-se que a query master esteja ativa, portanto, ative-a (**qryPedido**) definindo **True** para a propriedade **Active**, siga os passos descritos anteriormente e em seguida, fecha-a, definindo **False** para a propriedade **Active**.

Com o **FieldsEditor** aberto, selecione os **dois campos chaves**, **PED_NUMERO** e **PROD_CODIGO** e ajuste a propriedade **ProviderFlags** ligando a opção **pInKey**.



Ligando a opção pInKey na propriedade ProviderFlags dos campos PED_NUMERO e PROD_CODIGO

Testando a aplicação

Podemos agora cadastrar e modificar os itens normalmente que o erro não mais será gerado, pois agora o Provider já sabe quem são os campos chaves da tabela detalhe para poder montar a query de atualização corretamente.

Buscando Informações do Produto

Quando digitamos o Código do Cliente validamos e buscamos seu nome, utilizaremos o mesmo conceito no Código do Produto, ou seja, depois de digitado, validaremos e buscaremos a descrição e seu valor unitário.

As técnicas que utilizaremos, são as mesmas já vistas na validação do Código do Cliente, portanto, não entraremos em detalhes, apenas implementaremos este recurso passo a passo.

No DataModule **dmCadPedido** adicione um componente **TSQLQuery**:



TSQLQuery (dbExpress)

Name: qryInfoProduto

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  PROD_DESCRICAO,
  PROD_VALOR
FROM
  PRODUTO
WHERE
  PROD_CODIGO = :PROD_CODIGO
```

Neste SQL apenas estamos buscando a descrição e o valor de acordo com o Código do Produto.

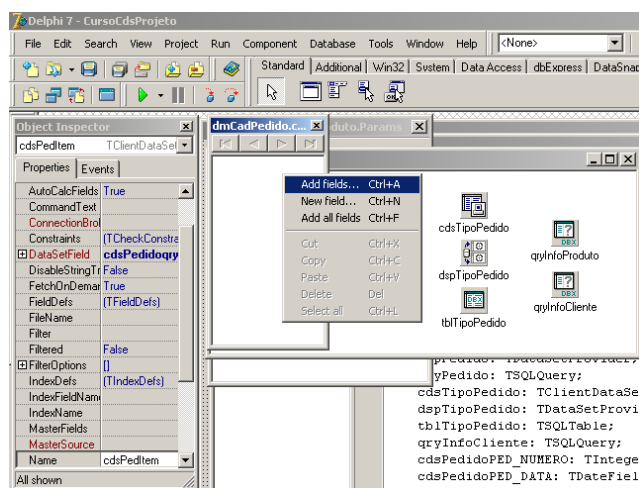
Em seguida ajuste o parâmetro que criamos, PROD_CODIGO da seguinte forma:

DataType: ftInteger

ParamType: ptInput

Implementaremos a validação do Código do Produto,

Abra o **FieldsEditor** do **cdsPedItem**, adicione todos os campos clicando com o botão direito do mouse e acessando o item de menu **Add all Fields**.



Adicionando os campos no cdsPedItem

Em seguida, selecione o campo **PROD_CODIGO** e no evento **OnValidate** insira o seguinte código:

```
procedure TdmCadPedido.cdsPedItemPROD_CODIGOValidate(Sender: TField);
begin
  if not Sender.IsNull then
  begin
    qryInfoProduto.ParamByName('PROD_CODIGO').Value := Sender.Value;
    qryInfoProduto.Open;
    try
      if not qryInfoProduto.IsEmpty then
      begin
        cdsPedItem.FieldByName('PI_DESCRICAO').Value :=
          qryInfoProduto.FieldByName('PROD_DESCRICAO').Value;
        cdsPedItem.FieldByName('PI_VALUNIT').Value :=
          qryInfoProduto.FieldByName('PROD_VALOR').Value
      end
      else
        raise Exception.Create('Produto não encontrado');
    finally
      qryInfoProduto.Close;
    end;
  end;
end;
```

Testando a aplicação

Podemos testar a aplicação e veremos que, ao digitar o código do produto nos itens, teremos a descrição e o valor ajustado automaticamente.

Trabalhando com Clones

Clonar um ClientDataSet é fazer uma cópia do mesmo para outro ClientDataSet, ambos terão os mesmos registros, apontando para o mesmo ponteiro, qualquer inclusão, alteração ou exclusão feita em um ClientDataSet afetará o outro.

Um ClientDataSet clonado possui o controle de estado e o posicionamento do cursor independente, ou seja, podemos nos movimentar ou colocar em edição um ClientDataSet e o outro não será afetado, continuará no mesmo registro e no estado que estava.

Podemos utilizar Clones em diversas situações, em nosso projeto temos um exemplo de utilização na qual nos trará muitos benefícios.

Um exemplo de utilização

Em nosso banco de dados definimos que a chave primária da tabela de Itens é o Número de Pedido + Código do Produto, portanto, não poderíamos permitir a inclusão de Produtos iguais no mesmo Pedido. Poderíamos validar isso no momento da digitação do Código do Produto, verificamos se o mesmo já existe nos Itens, se sim, barramos a inclusão, caso contrário, deixamos prosseguir.

Seguindo este conceito, teríamos um problema, no momento em que estamos digitando o Código do Produto, a tabela de Itens está em modo de edição e para checarmos se o mesmo já existe, teríamos que fazer uma busca na tabela, e sabemos que não podemos varrer a tabela em modo de edição.

É possível contornar esse problema de várias formas, mas daria muito trabalho. Neste caso um ClientDataSet Clonado resolve por completo nosso problema, pois clonariamos a tabela de Itens e após digitar o Código do Produto no ClientDataSet original, faríamos uma busca no ClientDataSet clonado, no qual não estará em modo de edição, portanto não haverá problemas ao fazer a busca.

Colocando em prática

No DataModule **dmCadPedido** adicione um **TClientDataSet**:



TclientDataSet (Data Access)

Name: cdsPedItemClone

Neste ClientDataSet não ajustaremos nada, pois será um Clone do **cdsPedItem**.

Para clonarmos um ClientDataSet devemos executar o método **CloneCursor** no ClientDataSet que representará o Clone, porém somente depois de o ClientDataSet original já ter sido aberto.

Este método possui a seguinte declaração:

```
procedure TCustomClientDataSet.CloneCursor(Source: TCustomClientDataSet;  
Reset, KeepSettings: Boolean);
```

Source: Indica o ClientDataSet a ser clonado

Reset e KeepSettings: Estes dois parâmetros nos indicam como serão tratadas as propriedades Filter, Filtered, FilterOptions, OnFilterRecord, IndexName, MasterSource, MasterFields, ReadOnly, RemoteServer e ProviderName.

Se **Reset** e **KeepSettings** forem falsos, todas essas propriedades serão compartilhadas e emparelhadas no novo DataSet.

Se **Reset** for **True**, todas as propriedades serão limpas.

Se **Reset** for **False** e **KeepSettings** for **True**, essas propriedades não serão mudadas.

Já que podemos Clonar somente após a abertura do ClientDataSet original, faremos o Clone no evento **AfterOpen** do **cdsPedItem**:

```
procedure TdmCadPedido.cdsPedItemAfterOpen(DataSet: TDataSet);
begin
    cdsPedItemClone.CloneCursor(cdsPedItem, True, False);
end;
```

Desta forma, após a abertura do **cdsPedItem**, teremos os dados disponíveis no **cdsPedItemClone**, portanto, podemos codificar a validação do Produto.

Ajuste o código do evento **OnValidate** do campo **PROD_CODIGO** do **cdsPedItem** da seguinte forma:

```
procedure TdmCadPedido.cdsPedItemPROD_CODIGOValidate(Sender: TField);
var
    BookOriginal,
    BookClone: TBookmark;
begin
    if not Sender.IsNull then
    begin
        if cdsPedItemClone.Locate('PROD_CODIGO', Sender.Value, []) then
        begin
            BookOriginal := cdsPedItem.GetBookmark;
            try
                BookClone := cdsPedItemClone.GetBookmark;
            try
                if cdsPedItem.CompareBookmarks(BookOriginal, BookClone) <> 0 then
                    raise Exception.Create('Produto já existente no Pedido')
                finally
                    cdsPedItemClone.FreeBookmark(BookClone);
                end;
            finally
                cdsPedItem.FreeBookmark(BookOriginal);
            end;
        end;
    end;

    qryInfoProduto.ParamByName('PROD_CODIGO').Value := Sender.Value;
    qryInfoProduto.Open;
    try
        if not qryInfoProduto.IsEmpty then
        begin
            cdsPedItem.FieldByName('PI_DESCRICAO').Value :=
                qryInfoProduto.FieldByName('PROD_DESCRICAO').Value;
            cdsPedItem.FieldByName('PI_VALUNIT').Value :=
                qryInfoProduto.FieldByName('PROD_VALOR').Value
            end
        else
            raise Exception.Create('Produto não encontrado');
        finally
            qryInfoProduto.Close;
        end;
    end;
end;
```

Entendendo o código

A primeira coisa que fazemos é verificar se o código do produto existe no clone (cdsPedItemClone) utilizando o método Locate, poderíamos utilizar outros métodos, esta é a forma mais simples, não teremos diferenças de performance já que temos pouca quantidade de registros. Caso o produto seja encontrado, fazemos uma comparação de Bookmarks (ponteiros dos registros) para checar se o registro localizado não é o mesmo que estamos editando, não sendo, barramos gerando uma exceção informando que o produto já está cadastrado. No final liberamos os Bookmarks que havíamos alocado e seguimos a sequência do código para obter os dados do produto.

Testando a aplicação

Ao tentarmos cadastrar itens com Códigos de Produtos iguais, seremos barrados com a mensagem de erro, e notaremos que em nenhum momento a posição do registro atual é modificada, pois a busca está sendo feita diretamente no Clone, não afetando o cursor original.

Implementando mais o Projeto - Calculando Valor Total do Item

Para que nosso projeto fique mais completo, iremos agora implementar um código para que o valor total do item seja calculado automaticamente.

Poderíamos codificar no evento BeforePost, porém não teríamos o resultado instantâneo ao modificarmos a quantidade do item por exemplo, somente após gravarmos.

Utilizaremos o evento **OnValidate** dos campos **Valor Unitário** e **Quantidade** para atualizarmos o valor total do item.

No DataModule **dmCadPedido** crie um método na seção **private** chamado **AtualizaValorTotalItem**:

```
...
private
  procedure AtualizaValorTotalItem;
public
  { Public declarations }
end;
...
```

Implemente-o da seguinte forma:

```
procedure TdmCadPedido.AtualizaValorTotalItem;
begin
  cdsPedItem.FieldByName('PI_VALTOTAL').AsFloat :=
    cdsPedItem.FieldByName('PI_VALUNIT').AsFloat *
    cdsPedItem.FieldByName('PI_QTDE').AsInteger;
end;
```

Entendendo o código

Neste código, nosso único objetivo é atualizar o valor total com a multiplicação entre os campos valor unitário e quantidade.

Executando o método

Chamaremos o método no evento **OnValidate** dos campos **PI_VALUNIT** e **PI_QTDE**, portanto, no **FieldsEditor** do componente **cdsPedItem** selecione o campo **PI_VALUNIT** e insira o seguinte código no evento **OnValidate**:

```
procedure TdmCadPedido.cdsPedItemPI_VALUNITValidate(Sender: TField);
begin
  AtualizaValorTotalItem;
end;
```

Repita o mesmo procedimento para o campo **PI_QTDE**:

```
procedure TdmCadPedido.cdsPedItemPI_QTDEValidate(Sender: TField);
begin
  AtualizaValorTotalItem;
end;
```

Testando a aplicação

Cadastre um Pedido incluindo itens e perceberemos que o Valor total do Item é atualizado automaticamente após informarmos a quantidade e o valor unitário.

Campos Aggregate

É muito comum nas aplicações precisarmos totalizar colunas, obter médias, valor mínimo, valor máximo, etc. Em instruções SQL's podemos utilizar funções agregadas para obter este tipo de resultado.

Podemos também fazer isso utilizando um recurso que o ClientDataSet nos disponibiliza, chamado de **AggregateField**, ou seja, podemos criar um campo que representara um valor agregado com base nos dados existentes no ClientDataSet.

Neste momento pode-se surgir à dúvida de qual opção é a mais recomendada. Para responder esta questão, precisamos analisar a situação, podemos resumir da seguinte forma:

Se as informações já estão disponíveis no ClientDataSet e elas são suficientes para se obter o resultado, certamente utilizar o AggregateField será a melhor opção, pois não estaremos incomodando o servidor de banco de dados desnecessariamente, obteremos o resultado de forma instantânea, já que será calculado em memória.

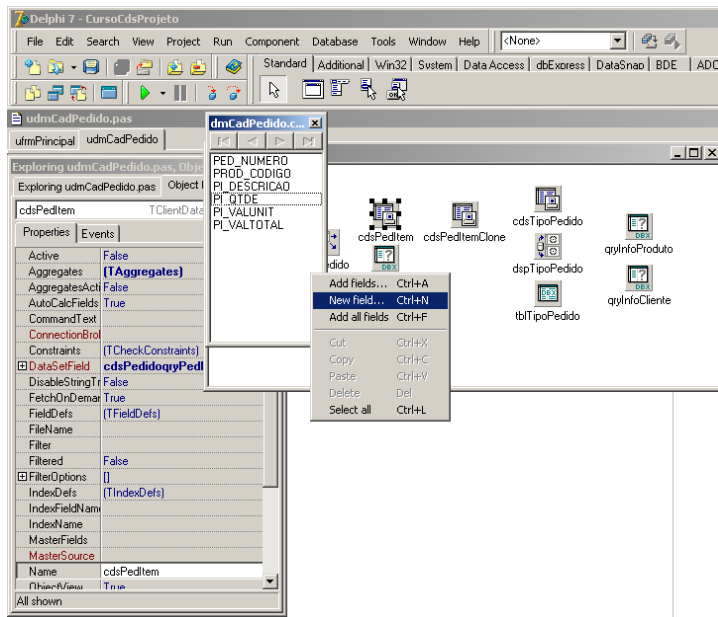
Por outro lado, se não temos as informações disponíveis no ClientDataSet, não valeria a pena abri-lo com os dados e utilizar o AggregateField, estaríamos trafegando registros desnecessariamente, neste caso, é muito mais recomendável executarmos uma instrução SQL no servidor.

Em nosso projeto utilizaremos este recurso do ClientDataSet para podermos visualizar o valor total de todos os itens do pedido aberto. O mais interessante é que o valor total será atualizado automaticamente a cada inserção, modificação ou exclusão de algum item, sem precisarmos codificar nenhuma linha de código.

Colocando em prática

Abra o DataModule **dmCadPedido** e em seguida o **FieldsEditor** do ClientDataSet onde será criado o campo Aggregate, no nosso caso será o **cdsPedItem**.

Com o **FieldsEditor** aberto, clique com o botão direito do mouse e clique no item **New field**.



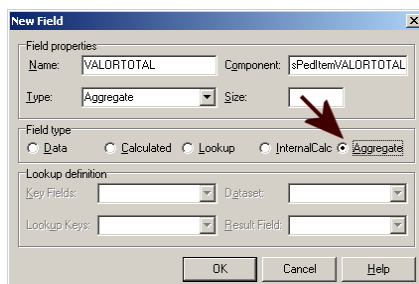
Acessando item New field do FieldsEditor

Será exibida a janela de criação de campos que já estamos acostumados a utilizar.

Preencha os dados da seguinte forma:

Name: VALORTOTAL

Field Type: Aggregate



Criando o campo Aggregate

O mais importante é observarmos o tipo de campo que estamos definindo: **Aggregate**.

Utilizaremos este campo como um qualquer, ligando em um DBText por exemplo para visualizarmos o resultado, porém, antes precisamos fazer ajustes em algumas propriedades.

Depois de criado o campo, selecione-o no FieldsEditor e ajuste as seguintes propriedades:

Active: True

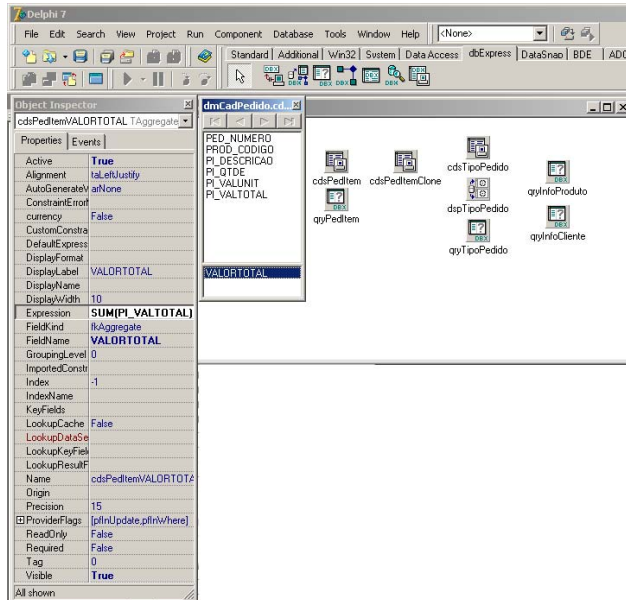
Na propriedade **Active** definimos se o campo está ou não ativo, pois em determinados casos podemos desativá-lo para evitar cálculos desnecessários.

Expression: SUM(PI_VALTOTAL)

Na propriedade **Expression** definimos a expressão de acordo com o tipo de resultado que desejamos obter, utilizamos o SUM, pois precisamos da soma total dos itens. Poderíamos usar: SUM, AVG, MAX e MIN. É possível também fazermos cálculos entre campos e depois sumarmos, exemplo: SUM(CAMPO1 + CAMPO2).

Visible: True

Nesta propriedade determinamos que o campo será visível para ser utilizado nos DataControls.

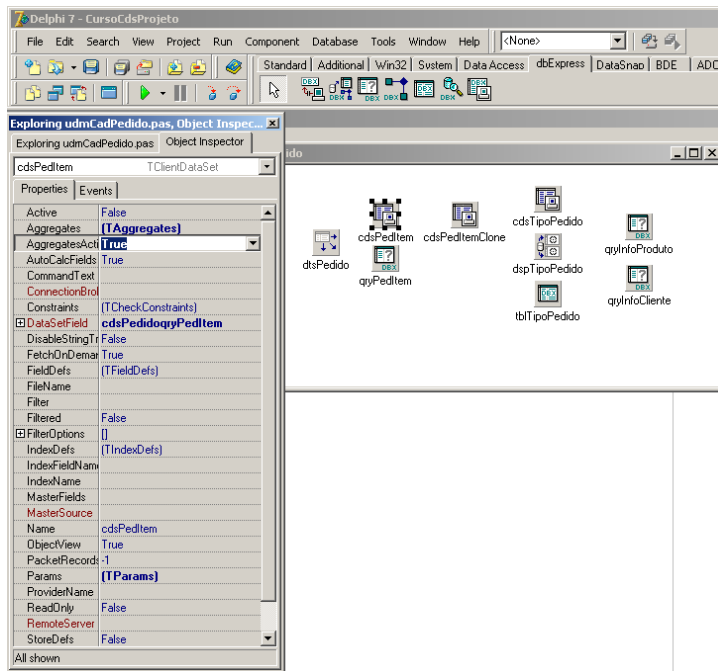


Ajustando as propriedades do campo Aggregate

Podemos notar que os campos agregados ficam em um quadro separado dos demais campos, isto facilita em muito na manutenção.

Da mesma forma que ativamos um determinado campo agregado, devemos também ativar este recurso no ClientDataSet.

Selecione o componente **cdsPedItem** e ajuste propriedade **AggregatesActive** para **True**.



Ajustando a propriedade `AggregatesActive` do `cdsPedItem`

Ajustando o Formulário de Cadastro

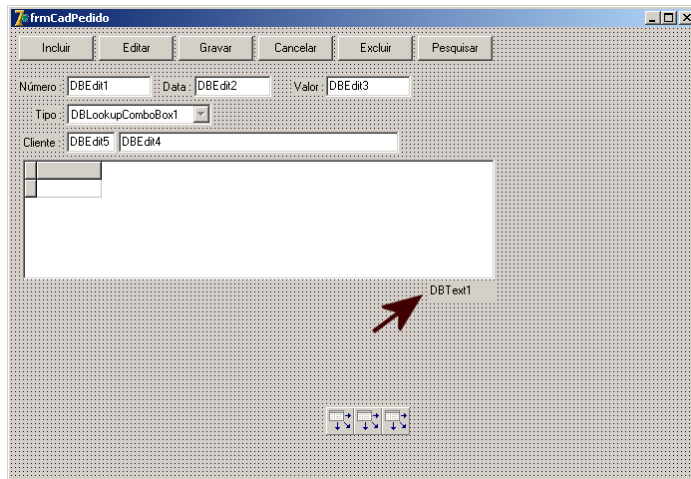
Abra o formulário `frmCadPedido` e adicione um componente para visualizar o resultado:



TDBText (Data Controls)

DataSource: `dtsPedItem`

DataField: `VALORTOTAL`



Cadastro de Pedido com o `DBText` adicionado para visualização do campo agregado

Testando a aplicação

É interessante observarmos que o valor é atualizado a todo o momento, logo que gravamos o item mais precisamente, porém isso não gera sobrecarga, pois internamente o ClientDataSet recalcula apenas o que foi alterado.

Campos InternalCalc

Trabalhar com campos calculados não é novidade, porém, o diferencial no ClientDataSet é que, além deste, existe outro tipo de campo calculado chamado **InternalCalc**.

Ele é muito semelhante ao campo **Calculated**, porém possui algumas diferenças:

Editável a qualquer momento

Campos **Calculateds** podem ser alterados somente no evento **OnCalcFields** do DataSet, já o **InternalCalc**, por ser armazenado no ClientDataSet, pode ser alterado a qualquer momento, no evento **OnCalcFields** e até mesmo em uma edição, como um campo qualquer do DataSet. Isto é muito interessante, pois podemos criá-lo como um campo “extra” no ClientDataSet podendo assim editá-lo, associá-lo a um DataControl, etc., a diferença é que não será enviado para o servidor na atualização do banco de dados.

Ganho de performance

Utilizando o evento **OnCalcFields**, podemos evitar que o campo **InternalCalc** seja calculado a todo o momento, basta verificarmos o *State* do ClientDataSet, se estiver como *dsInternalCalc*, significa que ele está processando esses campos, este é o momento ideal para ajustarmos seu valor. Fazendo testes, percebemos que este estado ocorre no momento em que gravamos o registro. Com base nisso, temos um número de recálculo inferior aos campos **Calculateds**, no qual são recalculados a todo o momento, pois não podemos fazer a mesma comparação, sendo assim temos uma perda de performance em casos onde o processo que o alimenta é muito “pesado”.

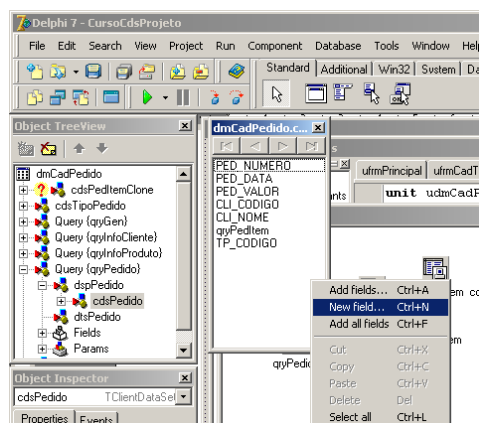
Indexação

Pelo fato de estarem armazenados no ClientDataSet, podemos indexar ou definir um índice que utilizam campos **InternalCalc**, diferente do que ocorre com os campos **Calculated**, que ao tentarmos fazer isto, uma mensagem de erro é exibida alertando de que isto não é possível.

Colocando em prática

Utilizaremos este campo de forma bem simples, criaremos um campo do tipo **InternalCalc** para representar a comissão do vendedor no Cadastro de Pedido.

Abra o DataModule **dmCadPedido**, em seguida, acesse o **FieldsEditor** do ClientDataSet **cdsPedido**. Com o botão direito do mouse, clique em **New Field...**



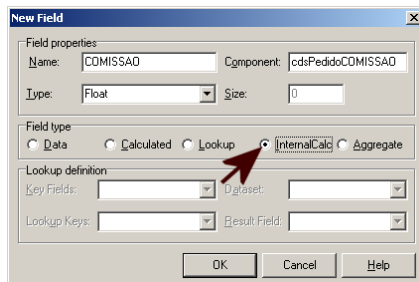
Criando novo campo no cdsPedido

Na tela de criação do campo, informe os dados da seguinte forma:

Name: COMISSAO

Type: Float

Field type: InternalCalc



Criando campo InternalCalc

Perceba que ajustamos o **Field type** para **InternalCalc**, é assim que determinamos este tipo de campo.

Confirme clicando no botão **OK**.

Para calcularmos este campo, utilizaremos o mesmo evento que já estamos acostumados, portanto, insira o seguinte código no evento **OnCalcFields** do **cdsPedido**:

```
procedure TdmCadPedido.cdsPedidoCalcFields(DataSet: TDataSet);
begin
    cdsPedido.FieldByName('COMISSAO').AsFloat :=
        cdsPedido.FieldByName('PED_VALOR').AsFloat * 0.05;
end;
```

Ajustando o formulário de Cadastro

Abra o formulário **frmCadPedido** e adicione os componentes relativos à **Comissão**.

A

TLabel

Caption: Comissão



TDBEdit

DataSource: dtsPedido

DataField: COMISSAO

Cadastro de Pedidos com campo InternalCalc para Comissão

Testando a aplicação

Ao entrarmos no Cadastro de Pedidos e editarmos os dados, veremos que o campo está sendo calculado a todo o momento, por exemplo, basta modificarmos o valor do pedido e a comissão já é calculada. Como havíamos comentado, podemos evitar isso para não sobrecarregar a aplicação quando o cálculo é demorado, portanto, faremos o devido ajuste.

Adicionando condição para evitar o recálculo desnecessário

Ajuste o evento **OnCalcFields** da seguinte forma:

```
procedure TdmCadPedido.cdsPedidoCalcFields(DataSet: TDataSet);
begin
    if cdsPedido.State = dsInternalCalc then
        cdsPedido.FieldByName('COMISSAO').AsFloat :=
            cdsPedido.FieldByName('PED_VALOR').AsFloat * 0.05;
end;
```

Entendendo o código

Apenas adicionamos a condição para verificar se o ClientDataSet está processando os campos InternalCalc, fazendo a comparação do *State* com *dsInternalCalc*.

Testando a aplicação

Se realizarmos o teste agora, verificaremos que o campo será calculado somente após a gravação, claro que, o mesmo também é calculado quando é aberto pela primeira vez, que é o caso de fazermos a pesquisa e abrirmos o registro.

É interessante notarmos também que podemos alterar o valor deste campo livremente, porém de nada adiantará, pois seu valor será sobreposto pelo cálculo que fazemos no evento OnCalcFields.

Generators – Trabalhando com Campos Auto-Incremento

Até o momento estamos cadastrando o Pedido informando o número manualmente. Implementaremos agora os famosos campos auto-incremento, faremos com que o Número do Pedido seja alimentado seqüencialmente.

Em tabelas Paradox definíamos um campo auto-incremento na sua própria estrutura, não precisávamos codificar nada na aplicação, a própria tabela já tinha este controle interno.

Muitos bancos relacionais possuem este tipo de campo, porém, o Firebird até a versão 1.5 não possui, neste caso utilizamos um novo conceito, chamado **Generators**.

Generators como o próprio nome já diz, são geradores, mais precisamente, geradores de valor, não são ligados a nenhuma tabela e a nenhum campo, é apenas um objeto que criamos no banco de dados responsável em armazenar um único valor. Incrementamos e retornamos seu valor com uso de uma função que o servidor de banco de dados nos disponibiliza.

Para entendermos melhor, exemplificaremos nosso caso.

Temos a tabela de Pedido e precisamos que o campo PED_NUMERO seja seqüencial. Se estivéssemos trabalhando com Paradox, definiríamos este campo como AutoInc e tudo estaria resolvido. No caso do Firebird, criaremos um GENERATOR com um nome qualquer, por exemplo, GEN_PED_NUMERO, e antes de inserirmos o Pedido no banco de dados, utilizaremos a função GEN_ID do Firebird (passando o nome do nosso Generator como parâmetro) para que ela possa incrementar o Generator e nos retornar o novo valor, com isso teremos o valor para ser definido no campo PED_NUMERO do Pedido.

Colocando em prática

Criando o Generator

Abra o IBExpert e execute o seguinte script SQL:

```
CREATE GENERATOR GEN_PED_NUMERO
```

Com este código, apenas definimos um novo generator no banco de dados com o nome GEN_PED_NUMERO.

Definindo um valor ao Generator

Quando criamos um Generator, seu valor padrão é zero. Para ajustarmos, podemos utilizar o seguinte script:

```
SET GENERATOR GEN_PED_NUMERO TO 1
```

Obtendo e Incrementando o valor de um Generator

Para obtermos e incrementarmos o valor de um generator, utilizaremos função GEN_ID que o Firebird nos disponibiliza, esta função possui 2 parâmetros:

```
GEN_ID(nome_do_generator, valor_incrementar)
```

No primeiro parâmetro informamos o nome do generator, em nosso caso, GEN_PED_NUMERO. Já no segundo, informamos o valor a ser incrementado, na maioria dos casos, informamos 1, ou seja, será incrementado de um em um, poderíamos definir um valor negativo, assim decrementaríamos o valor do generator.

```
GEN_ID(GEN_PED_NUMERO, 1)
```

Exemplificando na inserção de um registro, utilizamos assim:

```
INSERT INTO PEDIDO (PED_NUMERO, PED_VALOR) VALUES  
(GEN_ID(GEN_PED_NUMERO,1), 200)
```

Perceba que no lugar do valor do Número do Pedido, utilizamos a função GEN_ID para que o generator seja incrementado e o resultado seja passado para o campo.

Podemos executar esta instrução diversas vezes e os pedidos serão inseridos com a chave incrementada.

Em nosso projeto não poderemos utilizar desta forma, pois não manipulamos a instrução INSERT, como vimos, ela é gerada automaticamente pelo Provider. O que teremos que fazer é, obter o valor do generator (através de um select) incrementado e em seguida, atribuir este valor ao campo PED_NUMERO do ClientDataSet para que seja gerado o INSERT já com o Número do Pedido definido.

Obtendo o valor de um Generator através do SELECT

Retornamos o valor de um Generator através da instrução SELECT da seguinte forma:

```
SELECT  
  GEN_ID(GEN_PED_NUMERO,1) AS PED_NUMERO_NOVO  
FROM  
  TABELA
```

O que estamos fazendo é criando um campo fictício chamado PED_NUMERO_NOVO cujo valor será alimentado com o retorno da função GEN_ID.

Esta técnica é muito interessante, podemos criar campos fictícios e atribuir um valor retornado de funções, cálculo ou até mesmo um valor fixo. Experimente executar o seguinte script:

```
SELECT  
  10 AS TESTE  
FROM  
  TABELA
```

Será exibido um campo chamado TESTE cujo valor sempre será 10. No script anterior, trocamos esse valor por uma função.

O mais importante é atentarmos a TABELA que estamos fazendo o SELECT. Se escutarmos este script utilizando uma tabela que possui 20 registros por exemplo, veremos o resultado 20 vezes.

Voltando ao script anterior, onde utilizamos a função GEN_ID, se executarmos nesta tabela com 20 registros, a função será executada 20 vezes, conseqüentemente incrementará o Generator 20 vezes, perceberemos este resultado nos registros que serão exibidos, onde cada linha estará incrementada com um valor.

Podemos simular este teste com nossa tabela TIOPEDIDO que contém 3 registros:

```
SELECT  
  GEN_ID(GEN_PED_NUMERO,1) AS PED_NUMERO_NOVO  
FROM  
  TIOPEDIDO
```

Ao executarmos este script, veremos sempre 3 linhas, já que a tabela possui 3 registros e cada linha possui um valor diferente, pois como dissemos, a função GEN_ID está sendo executada em cada linha, portanto retornará um valor diferente para cada uma.

Por este motivo, precisamos de uma tabela que contenha **exatamente um registro**.

No banco Oracle, por exemplo, existe uma tabela para este caso, chamada de DUAL. Já no Firebird não temos, muitos utilizam a tabela RDB\$DATABASE, que é uma tabela interna que possui outros propósitos mas que contém apenas um registro. Particularmente acho mais interessante ter uma tabela específica para isso, portanto, vamos criá-la:

```
CREATE TABLE DUAL  
(DUAL_ID INTEGER NOT NULL PRIMARY KEY)
```

Definimos o nome DUAL somente para seguir o modelo do Oracle, mas poderíamos utilizar qualquer nome.

Precisamos inserir um registro nesta tabela, porém, antes execute um Commit no IBExpert já que acabamos de criá-la. Em seguida execute o script abaixo:

```
INSERT INTO DUAL (DUAL_ID) VALUES (1)
```

O valor que atribuímos ao campo DUAL_ID não é importante, o que devemos nos atentar é na quantidade de registros existentes nesta tabela, que sempre deverá ser **um**.

Agora podemos utilizar o SELECT para retornar o valor do generator:

```
SELECT  
  GEN_ID(GEN_PED_NUMERO,1) PED_NUMERO_NOVO  
FROM  
  DUAL
```

Cada vez que executarmos este SELECT, teremos um novo valor sendo exibido, ou seja, a função GEN_ID está incrementando o valor e retornando ao SELECT. É isso que precisamos, faremos uso disto para atribuímos ao nosso campo chave no projeto.

Aplicando o conceito ao projeto

Determinando o evento adequado

A primeira grande dúvida que surge é em que local iremos obter o valor do Generator e aplicar ao campo. Poderíamos utilizar o evento `OnNewRecord` do `ClientDataSet`, porém não é muito recomendado, pois o usuário pode começar a inserir um registro e cancelar, neste caso teríamos 'gasto' um Generator e não utilizado no banco de dados.

Outro evento seria o `BeforePost`, mas mesmo assim não é o mais adequado, poderíamos ter algum erro na gravação do registro (ainda em memória) antes de ser enviado ao banco de dados.

O evento mais adequado é **`BeforeUpdateRecord`** do **Provider**, ele é chamado no momento que o registro está prestes a ser enviado ao banco de dados, portanto, podemos fazer os ajustes e deixar o **Provider** prosseguir com a atualização, refletindo as novas mudanças.

Definindo um valor fictício para chave primária

Já que alimentaremos o campo somente no evento `BeforeUpdateRecord`, o Pedido será inserido com o Número em branco. Deixá-lo como NULO pode nos gerar problemas, além de termos que desligar o `Required` do campo, ao trabalhar com diversos registros, teríamos que 'gravar fisicamente' cada Pedido antes de iniciar um novo, caso contrário teríamos uma violação de chave primária no `ClientDataSet`, pois mais de um registro estaria sendo incluso com o mesmo valor, no caso NULO.

Portanto o que fazemos é preencher o campo chave com um valor fictício, no caso, um número negativo, já que não teremos Pedidos com números deste tipo.

Declare na seção **private** do DataModule **dmCadPedido** uma variável global chamada **FSeqTmp** do tipo **Integer** para utilizarmos como seqüenciador negativo:

```
...
private
  FSeqTmp: Integer;
  procedure AtualizaValorTotalItem;
public
  { Public declarations }
end;
...
```

Devemos inicializar com valor **Zero** na criação do DataModule:

```
procedure TdmCadPedido.DataModuleCreate(Sender: TObject);
begin
  FSeqTmp := 0;
end;
```

Utilizaremos a variável no evento **OnNewRecord** do **cdsPedido**:

```
procedure TdmCadPedido.cdsPedidoNewRecord(DataSet: TDataSet);
begin
  Dec(FSeqTmp);
  cdsPedido.FieldByName('PED_NUMERO').AsInteger := FSeqTmp;
end;
```


Entendendo o código

A cada novo registro a variável será decrementada e enviada ao campo Número do Pedido. Neste caso não há problemas de o usuário cancelar a inclusão, pois neste momento apenas utilizamos o seqüenciador negativo que é utilizado temporariamente.

Antes de codificarmos o próximo evento, prepararemos nossa query que buscará o valor do generator incrementado, portanto inclua no DataModule uma SQLQuery:



TSQLQuery (dbExpress)

Name: qryGen

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  GEN_ID(GEN_PED_NUMERO,1) PED_NUMERO_NOVO
FROM
  DUAL
```

Implementando o evento BeforeUpdateRecord

Antes de implementarmos o evento, precisamos declarar mais uma variável global na seção **private** chamada **FNumPedido** do tipo **Integer** que será utilizada e explicada a seguir.

```
private
  FNumPedido: Integer;
  FSeqTmp: Integer;
  procedure AtualizaValorTotalItem;
public
  { Public declarations }
end;
```

Implementaremos o evento **BeforeUpdateRecord** do **dspPedido** da seguinte forma:

```
procedure TdmCadPedido.dspPedidoBeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind = ukInsert then
  begin
    if SourceDs = qryPedido then
    begin
      qryGen.Open;
      try
        FNumPedido := qryGen.FieldByName('PED_NUMERO_NOVO').AsInteger;
      finally
        qryGen.Close;
      end;
      DeltaDs.FieldByName('PED_NUMERO').NewValue := FNumPedido;
    end else
    if (SourceDs = qryPedItem) then
    begin
      if cdsPedItem.FieldByName('PED_NUMERO').AsInteger < 0 then
        DeltaDs.FieldByName('PED_NUMERO').NewValue := FNumPedido;
    end;
  end;
```

Entendendo o código

O código parece ser complexo, mas explicando passo a passo entenderemos facilmente.

Primeiramente é importante sabermos que este evento será executado diversas vezes de acordo com o número de operações realizadas no ClientDataSet. Por exemplo, se incluímos um Pedido e um Item, este evento será executado 2 vezes, na ordem que o ClientDataSet organiza para não haver erro de integridade no banco, portanto na primeira vez será para inserção do Pedido e na segunda para inclusão do Item.

Analisando o código, verificamos se está sendo feito uma inclusão comparando UpdateMode com ukInsert, em seguida, checamos se refere a Pedido comparando o SourceDs com qryPedido (sempre devemos comparar com o DataSet do Provider e não o ClientDataSet), em caso positivo, temos então que utilizar o generator, fazemos isso abrindo a query e guardando o valor retornado na variável global. Depois atribuímos o valor obtido ao **Delta** na propriedade **NewValue** do campo. O Delta representa os dados que serão enviados ao banco, portanto, é nele que devemos fazer as modificações e sempre na propriedade **NewValue** do campo.

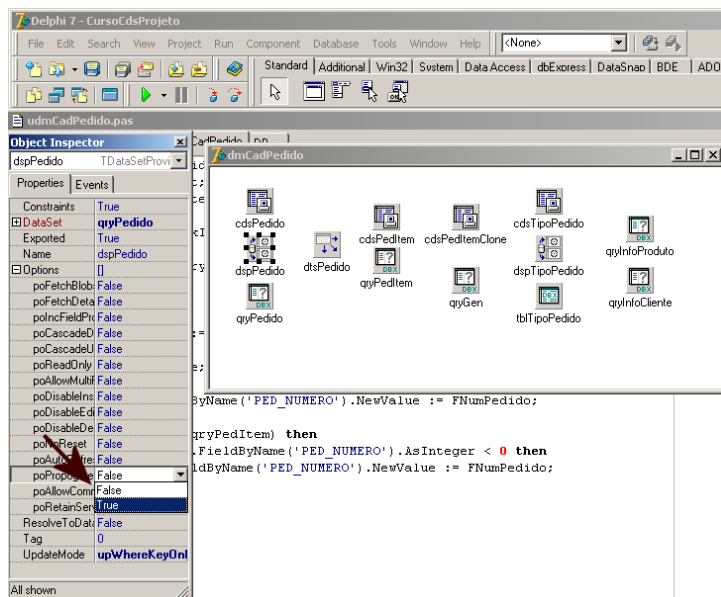
Nas linhas seguintes comparamos se estamos incluindo um item (em nosso exemplo será verdadeira na segunda chamada deste evento), sendo verdadeiro então verificamos se o Número do Pedido do item é negativo para saber se o mesmo pertence a um Pedido novo, neste caso então atribuímos ao campo chave o valor da variável global que obtivemos na primeira execução do evento.

Refletindo no ClientDataSet as modificações feitas no Delta

As alterações que fizemos no Delta foram aplicadas ao banco de dados, porém o ClientDataSet não visualiza essas mudanças, portanto, continuaremos com o Número Pedido negativo no DBEdit, somente após fecharmos e abrímos o ClientDataSet veremos as mudanças.

Para que as mudanças no Delta sejam refletidas, precisamos pedir ao Provider que propague-as ao ClientDataSet, fazemos isso ajustando a propriedade **Options** do **Provider** ligando a opção **poPropagateChanges**.

Portanto, selecione o componente **dspPedido** e na propriedade **ProviderOptions** defina **True** na opção **poPropagateChanges**.



Ligando opção *poPropagateChanges* no Provider

Testando a aplicação

Antes de testarmos, vamos limpar todos registros de pedidos e itens para não haver conflitos de Números já existentes, portanto executaremos 2 scripts no banco de dados:

DELETE FROM PEDITEM

DELETE FROM PEDIDO

Neste momento podemos executar a aplicação e simular a inserção de um Pedido. Perceberemos que o Número estará negativo na inclusão, porém ao gravar, será ajustado com o valor do Generator.

Ocultando o número negativo do DBEdit

Talvez possa ser um incomodo ficar visualizando o Número do Pedido negativo no DBEdit, podemos ajustá-lo visualmente, deixando em branco nestes casos, assim ele será exibido com o valor real (positivo) após a gravação no banco de dados.

Abra o **FieldsEditor** do **cdsPedido** e selecione o campo **PED_NUMERO**. No evento **OnGetText** insira o seguinte código:

```
procedure TdmCadPedido.cdsPedidoPED_NUMEROGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if Sender.AsInteger < 0 then
    Text := ''
  else
    Text := Sender.AsString;
end;
```

Entendendo o código

Apenas verificamos se o valor é menor que zero, sendo, atribuímos um valor em branco à variável Text, caso contrário, alimentamos com o valor real.

Testando a aplicação

Fazendo a inserção do Pedido, veremos que o Número do Pedido começa em branco, e logo após a gravação, temos o campo sendo exibido com o valor real.

Criando o Cadastro de Tipos de Pedidos

Crie um novo DataModule, ajuste o nome para **dmCadTipoPedido** e salve-o como **udmCadTipoPedido.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...  
implementation  
  
uses  
    udmPrincipal;  
...
```

Adicione os seguintes componentes no DataModule:



TClientDataSet (Data Access)

Name: cdsTipoPedido

ProviderName: dspTipoPedido



TDataSetProvider (Data Access)

Name: dspTipoPedido

DataSet: qryTipoPedido



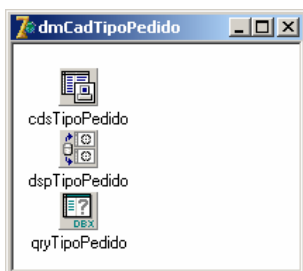
TSQLQuery (dbExpress)

Name: qryTipoPedido

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT  
    TP_CODIGO,  
    TP_DESCRICAO  
FROM  
    TIPOPEDIDO
```



DataModule de Cadastro de Tipos de Pedido

Implementando o Generator

Implementaremos agora o Generator, para que possamos ter a chave primária sendo alimentada automaticamente, da mesma forma que fizemos no cadastro de Pedidos.

Antes, devemos criá-lo no banco de dados:

```
CREATE GENERATOR GEN_TP_CODIGO
```

Devemos ajustar o valor inicial, pois já temos 3 registros na tabela, portanto, para não haver conflitos de chave, defina-o com o valor 3:

```
SET GENERATOR GEN_TP_CODIGO TO 3
```

Adicione um SQLQuery no DataModule, utilizaremos para incrementar e retornar o valor do Generator:



TSQLQuery (dbExpress)

Name: qryGen

SQLConnection: dmPrincipal.SqlConnPrincipal

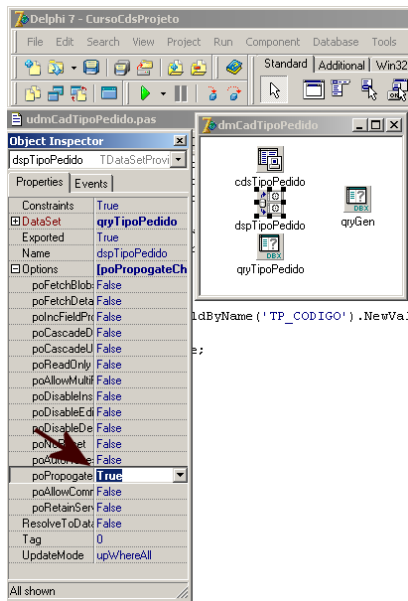
SQL:

```
SELECT
  GEN_ID(GEN_TP_CODIGO,1) TP_CODIGO_NOVO
FROM
  DUAL
```

Da mesma forma que fizemos no Cadastro de Pedidos, faremos aqui codificando o evento **BeforeUpdateRecord** do Provider, porém perceberemos que o código é mais simples, pois não temos tabela detalhe neste caso.

```
procedure TdmCadTipoPedido.dspTipoPedidoBeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind = ukInsert then
    if SourceDS = qryTipoPedido then
      begin
        qryGen.Open;
        try
          DeltaDS.FieldByName('TP_CODIGO').NewValue :=
            qryGen.FieldByName('TP_CODIGO_NOVO').Value;
        finally
          qryGen.Close;
        end;
      end;
    end;
end;
```

Como havíamos comentando, precisamos ajustar a propriedade **Options** do **Provider** ligando a opção **poPropagateChanges** para refletir as mudanças no ClientDataSet, portanto, defina **True** para esta opção no componente **dspTipoPedido**.



Ligando opção *poPropagateChanges* no Provider

Feito isto, vamos agora codificar para termos o seqüenciador negativo, da mesma forma que fizemos no Cadastro de Pedidos.

Declare uma variável do tipo **Integer** na seção **private** do DataModule chamada **FSeqTmp**.

```
...
private
    FSeqTmp: Integer;
public
    { Public declarations }
end;
...
```

No evento **OnCreate** do DataModule, inicialize-a com o valor zero:

```
procedure TdmCadTipoPedido.DataModuleCreate(Sender: TObject);
begin
    FSeqTmp := 0;
end;
```

No evento **OnNewRecord** do **ClientDataSet** utilizamos a variável:

```
procedure TdmCadTipoPedido.cdsTipoPedidoNewRecord(DataSet: TDataSet);
begin
    Dec(FSeqTmp);
    cdsTipoPedido.FieldName('TP_CODIGO').AsInteger := FSeqTmp;
end;
```

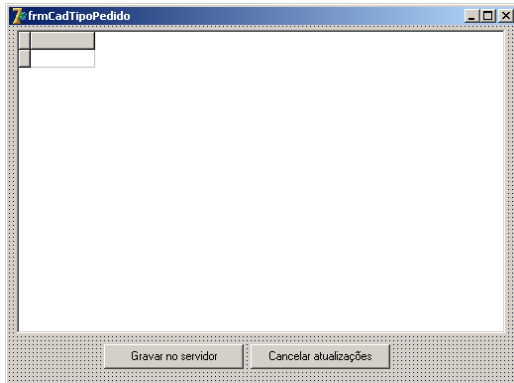
Criando o Formulário

Crie um novo formulário, ajuste o nome para **frmCadTipoPedido** e salve-o como **ufmCadTipoPedido.pas**.

Adicione na cláusula **uses** a unit **udmCadTipoPedido**.

```
...  
implementation  
  
uses  
    udmCadTipoPedido;  
...
```

Em seguida, monte o formulário da seguinte forma:



Formulário de Cadastro de Tipos de Pedido

Ajuste o nome dos botões para:
btnGravarServidor e **btnCancelarAtualizacoes**

Em seguida adicione um **DataSource**:



TDataSource (Data Access)

Name: dtsTipoPedido

DataSet: dmCadTipoPedido.cdsTipoPedido

No **DBGrid**, ajuste a propriedade **DataSource** apontando para **dtsTipoPedido**.

No evento **OnCreate** insira o seguinte código:

```
procedure TfrmCadTipoPedido.FormCreate(Sender: TObject);  
begin  
    dmCadTipoPedido := TdmCadTipoPedido.Create(Self);  
    dmCadTipoPedido.cdsTipoPedido.Open;  
end;
```

E no evento **OnDestroy**:

```
procedure TfrmCadTipoPedido.FormDestroy(Sender: TObject);  
begin  
    dmCadTipoPedido.cdsTipoPedido.Close;  
    dmCadTipoPedido.Free;  
    dmCadTipoPedido := nil;  
end;
```


Codificaremos os botões da seguinte forma:

Gravar no Servidor:

```
procedure TfrmCadTipoPedido.btnGravarServidorClick(Sender: TObject);
begin
    if dmCadTipoPedido.cdsTipoPedido.ApplyUpdates(0) <> 0 then
        dmCadTipoPedido.cdsTipoPedido.CancelUpdates;
end;
```

Cancelar Atualizações:

```
procedure TfrmCadTipoPedido.btnCancelaAtualizacoesClick(Sender: TObject);
begin
    dmCadTipoPedido.cdsTipoPedido.CancelUpdates;
end;
```

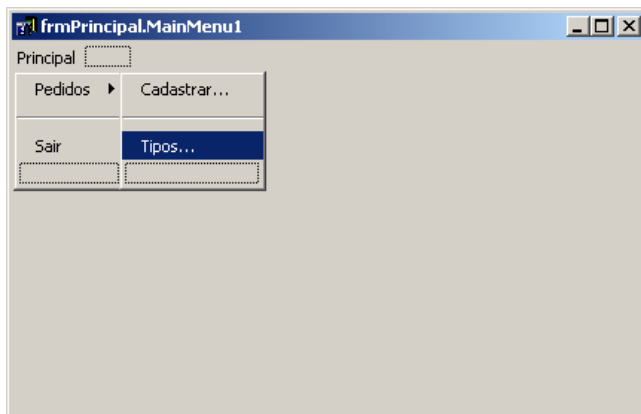
Ajustando o Formulário Principal

Abra o formulário **frmPrincipal** e na cláusula **uses** adicione a unit **ufrmCadTipoPedido**.

```
...
implementation

uses
    ufrmCadPedido,
    ufrmCadTipoPedido;
...
```

No **MainMenu**, inclua um item chamado **Tipos** dentro do menu **Pedidos**, deixando da seguinte forma:



Criando Item de Menu 'Tipos'

No evento **OnClick** do Item, insira o seguinte código:

```
procedure TfrmPrincipal.ipos1Click(Sender: TObject);
begin
    if not Assigned(frmCadTipoPedido) then
        frmCadTipoPedido := TfrmCadTipoPedido.Create(Application);
    frmCadTipoPedido.Show;
end;
```

Testando a aplicação

Neste modelo perceba que temos o botão **Gravar no Servidor**, isso significa que podemos fazer as manutenções neste cadastro e ao final, aplicá-las para que possam ser gravadas no banco de dados.

Criando a Pesquisa de Pedidos

Neste momento nosso projeto contém apenas uma busca simples, onde pedimos o Número do Pedido ao usuário através da chamada do método InputQuery. Implementaremos agora uma busca avançada, teremos uma tela onde o usuário poderá informar o Número do Pedido ou Período de Data de Cadastro dos Pedidos.

É interessante atentarmos a este tela, pois como dissemos, temos que seguir um novo conceito quando trabalhamos com Cliente/Servidor, não abrimos todos os registros num DBGrid para o usuário selecionar qual deseja editar por exemplo, teremos sempre uma busca parametrizada, assim o usuário selecionará o registro e o abriremos especificamente para edição.

Colocando em prática

Crie um novo DataModule, ajuste o nome para **dmPesqPedido** e salve-o como **udmPesqPedido.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...  
implementation  
  
uses  
    udmPrincipal;  
...
```

Em seguida, adicione os seguintes componentes:



TClientDataSet (Data Access)

Name: cdsPedido

ProviderName: dspPedido



TDataSetProvider (Data Access)

Name: dspPedido

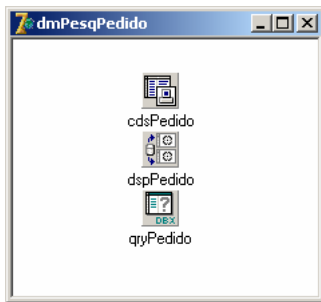
DataSet: qryPedido



TSQLQuery (dbExpress)

Name: qryPedido

SQLConnection: dmPrincipal.SqlConnPrincipal



DataModule de Pesquisa de Pedido

O próximo passo seria ajustarmos o SQL da Query, porém, aqui já temos um diferencial. Nossa pesquisa será dinâmica, o usuário poderá pesquisar pelo Número do Pedido ou pela Data, portanto, teremos um script SQL variável, teremos a cláusula WHERE diferenciada de acordo com a pesquisa.

Por este motivo, deixaremos em branco e definiremos o SQL em tempo de execução.

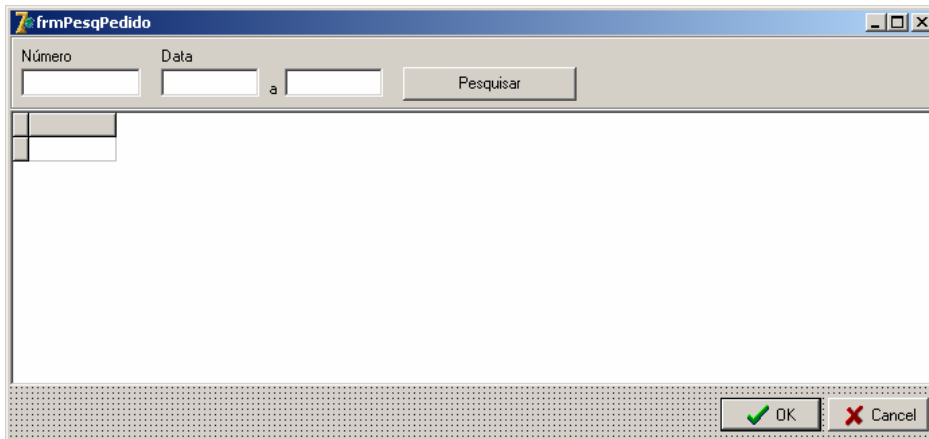
Criando o Formulário

Crie um novo formulário, ajuste o nome para **frmPesqPedido** e salve-o como **ufrmPesqPedido.pas**.

Adicione cláusula **uses** a unit **udmPesqPedido**.

```
...  
implementation  
  
uses  
    udmPesqPedido;  
...
```

Em seguida, monte o formulário da seguinte forma:



Tela de Pesquisa de Pedido

Nomeie os componentes na respectiva ordem para facilitar nosso acesso mas adiante:

Edits: edtNumero, edtDataIni, edtDataFim

Buttons: btnPesquisar, btnOk, btnCancel

DBGrid: dbgrdPedido

Ajuste a propriedade **BorderStyle** do formulário para **bsDialog**, pois chamaremos como Modal.

Vale lembrar que os botões Ok e Cancelar devem estar com a propriedade **Kind** configurada:

Botão Ok: bkOk

Botão Cancelar: bkCancel

Em seguida adicione um **DataSource**:



TDataSource

Name: dtsPedido

DataSet: dmPesqPedido.cdsPedido

Ajuste as seguintes propriedades do **DBGrid**:

DataSource: dtsPedido

ReadOnly: True

Ligamos a propriedade **ReadOnly** do **DBGrid** para não permitirmos qualquer tipo de alteração nos dados.

Codificaremos os eventos **OnCreate** e **OnDestroy** do formulário para instanciarmos e destruímos o DataModule:

OnCreate:

```
procedure TfrmPesqPedido.FormCreate(Sender: TObject);
begin
    dmPesqPedido := TdmPesqPedido.Create(Self);
end;
```

OnDestroy:

```
procedure TfrmPesqPedido.FormDestroy(Sender: TObject);
begin
    dmPesqPedido.Free;
    dmPesqPedido := nil;
end;
```

Implementando o botão Pesquisar

Implementaremos agora o código do **botão Pesquisar** para realizarmos a consulta do Pedido de acordo com o que foi informado nos campos, portanto, codifique o evento **OnClick** do botão da seguinte forma:

```
procedure TfrmPesqPedido.btnPesquisarClick(Sender: TObject);
var
  sWhere: string;
begin
  if edtNumero.Text <> '' then
    sWhere := 'ped_numero = ' + edtNumero.Text
  else
    sWhere := 'ped_data between ' +
      QuotedStr(FormatDateTime('mm/dd/yy',
        StrToDate(edtDataIni.Text))) +
      ' and ' +
      QuotedStr(FormatDateTime('mm/dd/yy',
        StrToDate(edtDataFim.Text)));

  dmPesqPedido.cdsPedido.Close;
  dmPesqPedido.cdsPedido.CommandText :=
    'select ped_numero, ped_data, ped_valor from pedido where ' + sWhere;
  dmPesqPedido.cdsPedido.Open;
end;
```

Entendo o código

Analisando o código, temos a montagem da cláusula WHERE sendo feito de acordo com os campos informados e ao final atribuímos à propriedade **CommandText** do **ClientDataSet** o SQL montado.

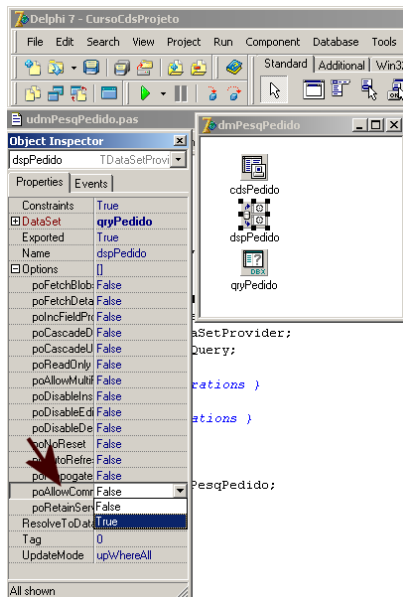
Entendendo a propriedade CommandText do ClientDataSet

Por que estamos informando o SQL na propriedade **CommandText** do ClientDataSet e não no SQLQuery?

Esta propriedade permite que o cliente (ClientDataSet) substitua o SQL existente no DataSet ao qual o Provider está ligado, isto é interessante, pois no modelo multicamadas o SQLQuery não estaria no mesmo local que o ClientDataSet, não teríamos acesso ao componente da mesma forma que temos ao ClientDataSet, portanto enviamos o SQL através do ClientDataSet.

Por padrão, se tentarmos utilizar esta propriedade, não teremos resultado, pois precisamos configurar o Provider para aceitar o comando vindo através da propriedade CommandText do ClientDataSet. Esta permissão é feita ligando a opção **poAllowCommandText** na propriedade Options do **Provider**.

Portanto, abra o DataModule **dmPesqPedido**, selecione o componente **dspPedido** e defina **True** na opção **poAllowCommandText** da propriedade **ProviderOptions**.



Ligando poAllowCommandText no Provider

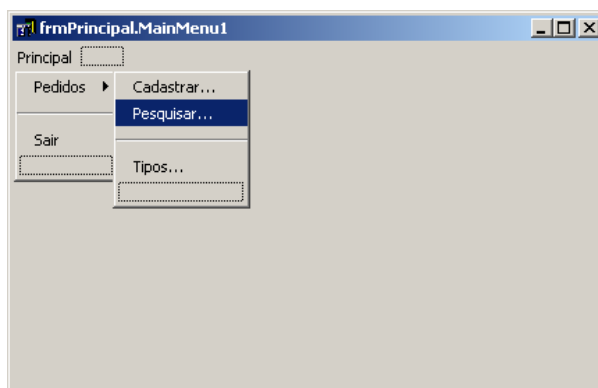
Ajustando Formulário Principal

Abra o formulário **frmPrincipal** e adicione na cláusula **uses** a unit **ufrmPesqPedido**.

```
...  
implementation
```

```
uses  
    ufrmCadPedido,  
    ufrmCadTipoPedido,  
    ufrmPesqPedido;  
...
```

No **MainMenu**, crie um item chamado **Pesquisar** dentro do menu **Pedidos**.



Criando item de menu Pesquisar

Codifique o evento **OnClick** do Item da seguinte forma:

```
procedure TfrmPrincipal.Pesquisar1Click(Sender: TObject);
begin
    frmPesqPedido := TfrmPesqPedido.Create(Self);
    frmPesqPedido.ShowModal;
    frmPesqPedido.Free;
    frmPesqPedido := nil;
end;
```

Testando a aplicação

Execute a aplicação e abra a tela de Pesquisa, informe o Número do Pedido ou as Datas e clique no botão **Pesquisar**, os registros serão exibidos no DBGrid de acordo com a busca informada.

Abrindo a tela de cadastro com o registro pesquisado.

O nosso próximo passo agora é fazer com que, ao dar um duplo clique no registro, seja aberta a tela de cadastro com o respectivo registro.

O que faremos é criar um método no formulário de cadastro para que possamos instanciá-lo passando como parâmetro a chave do Pedido, assim ele repassará para a query e conseqüentemente teremos o respectivo Pedido aberto.

Abra o formulário **frmCadPedido** e na seção **public**, crie um **método de classe** chamado **AbreForm**.

```
...
private
{ Private declarations }
public
    class procedure AbreForm(NumPedido: Variant);
end;
...
```

Implemente o método da seguinte forma:

```
class procedure TfrmCadPedido.AbreForm(NumPedido: Variant);
begin
    if not Assigned(frmCadPedido) then
        frmCadPedido := TfrmCadPedido.Create(Application);

    dmCadPedido.cdsPedido.Close;
    dmCadPedido.cdsPedido.FetchParams;
    dmCadPedido.cdsPedido.Params.ParamByName('PED_NUMERO').Value :=
        NumPedido;
    dmCadPedido.cdsPedido.Open;

    frmCadPedido.Show;
end;
```

Entendendo o código

A maioria das chamadas feitas no evento **OnCreate** do formulário está sendo executada agora neste método, a diferença é que também estamos **criando o formulário** e **alimentando o parâmetro PED_NUMERO** de acordo com o parâmetro **NumPedido** passado no método.

Precisamos agora ajustar o evento **OnCreate** do formulário, de modo que seja responsável apenas pela criação do DataModule e abertura da tabela de Tipos de Pedido.

```
procedure TfrmCadPedido.FormCreate(Sender: TObject);
begin
  dmCadPedido := TdmCadPedido.Create(Self);
  dmCadPedido.cdsTipoPedido.Open;
end;
```

Ajustando o formulário Principal

Abra o formulário **frmPrincipal** e modifique o código do evento **OnClick** do item **Cadastrar** da seguinte forma:

```
procedure TfrmPrincipal.Cadastrar1Click(Sender: TObject);
begin
  TfrmCadPedido.AbreForm(NULL);
end;
```

Entendendo o código

O que estamos fazendo é apenas chamando o método **AbreForm** da classe **TfrmCadPedido** passando como parâmetro o valor **NULL** para que o ClientDataSet seja aberto vazio, sem nenhum registro. Não precisaremos mais criar o formulário neste local, isto já está sendo feito pelo próprio método **AbreForm**.

Ajustando Tela de Pesquisa

Criaremos um método dentro do Formulário de Pesquisa que será responsável em instanciar o formulário como Modal e retornar a chave do Pedido pesquisado. Desta forma teremos este método disponível para ser utilizado em conjunto com a chamada do método **AbreForm** no Formulário Principal e também no botão **Pesquisar** da tela de Cadastro.

Abra o formulário **frmPesqPedido** e crie um **método de classe** na seção **Public** chamado **Pesquisa**:

```
...
private
{ Private declarations }
public
  class function Pesquisa(var NumPedido: Integer): Boolean;
end;
...
```

Implemente-o da seguinte forma:

```
class function TfrmPesqPedido.Pesquisa(var NumPedido: Integer): Boolean;
begin
  with TfrmPesqPedido.Create(Application) do
    try
      if ShowModal = mrOk then
        begin
          Result := True;
          NumPedido :=
dmPesqPedido.cdsPedido.FieldByName('ped_numero').AsInteger;
        end
      else
        Result := False;
      finally
        Release;
      end;
    end;
  end;
end;
```

Entendendo o código

Primeiro criamos o formulário, em seguida exibimos como Modal através da função **ShowModal**.

Nesse ponto o código somente prosseguirá após fecharmos o formulário, pois estamos exibindo-o como Modal.

Depois comparamos o resultado com a constante **mrOk**, pois sendo verdadeiro, significa que o usuário clicou no botão **OK**, neste caso retornamos **True** para a função e atribuímos ao parâmetro **NumPedido** o valor da chave (campo PED_NUMERO) do registro selecionado. Ao final destruimos o formulário com o método **Release**.

Impedindo que o usuário confirme uma pesquisa sem registros

No evento **OnCloseQuery** impediremos o fechamento do formulário caso o usuário tente confirmar a pesquisa com o ClientDataSet vazio.

```
procedure TfrmPesqPedido.FormCloseQuery(Sender: TObject;
var CanClose: Boolean);
begin
  if ModalResult = mrOk then
    if dmPesqPedido.cdsPedido.IsEmpty then
      begin
        CanClose := False;
        MessageDlg('Nenhum registro selecionado !', mtInformation, [mbOk], 0);
      end;
    end;
  end;
end;
```

Entendo o código

Verificamos se a pesquisa foi confirmada, se sim, então checamos se o ClientDataSet está vazio, neste caso impedimos o fechamento do formulário definindo **False** para a variável **CanClose** e em seguida exibimos uma mensagem ao usuário.

Confirmando a pesquisa quando der um duplo clique no DBGrid

Implementaremos o evento **OnDbClick** do DBGrid, para que ao dar um duplo clique no registro, tenha o mesmo efeito de confirmar a pesquisa:

```
procedure TfrmPesqPedido.dbgrdPedidoDblClick(Sender: TObject);
begin
    btnOk.Click;
end;
```

Ajustando formulário Principal

Abra o formulário **frmPrincipal** e modifique o evento **OnClick** do item **Pesquisar** da seguinte forma:

```
procedure TfrmPrincipal.Pesquisar1Click(Sender: TObject);
var
    iNumPedido: Integer;
begin
    if TfrmPesqPedido.Pesquisa(iNumPedido) then
        TfrmCadPedido.AbreForm(iNumPedido);
end;
```

Entendo o código

Podemos perceber o quanto ficou simples nosso código, tudo está centralizado nos métodos **Pesquisa** e **AbreForm** dos formulários.

Na primeira linha exibimos a tela de pesquisa passando como parâmetro nossa variável local para que seja alimentada com o número do Pedido pesquisado. Logo depois, caso a pesquisa tenha sido confirmada, chamamos o método **AbreForm** passando a variável local como parâmetro, pois assim abriremos o Formulário de Cadastro com o Número do Pedido pesquisado.

Ajustando Tela de Cadastro (Botão Pesquisar)

Abra o formulário **frmCadPedido** e adicione na cláusula **uses** a unit **ufrmPesqPedido**.

```
...
uses
    udmCadPedido,
    ufrmPesqPedido;
...
```

Em seguida ajuste o código do evento **OnClick** do botão **Pesquisar** da seguinte forma:

```
procedure TfrmCadPedido.btnPesquisarClick(Sender: TObject);
var
    iNumPedido: Integer;
begin
    if TfrmPesqPedido.Pesquisa(iNumPedido) then
        TfrmCadPedido.AbreForm(iNumPedido);
end;
```

Testando a aplicação

Executando a aplicação, temos disponível agora uma tela de Pesquisa de Pedidos parametrizada, que confirmando, abrimos o respectivo Pedido para edição.

Criando a Pesquisa de Clientes

Como havíamos comentado no início do projeto, precisamos de uma tela de Pesquisa de Clientes para não obrigar o usuário ter que lembrar do Código do Cliente no Cadastro de Pedidos, nesta pesquisa ele poderá buscar pelo Nome e confirmando, pegamos o Código do respectivo Cliente e repassamos para o Pedido, conseqüentemente o Nome do Cliente será exibida no campo onde fizemos o Join, pois já implementamos isso no evento OnValidate.

Os conceitos que utilizaremos serão semelhantes ao que vimos na tela de Pesquisa de Pedidos, portanto, não entraremos muito em detalhes.

Colocando em prática

Crie um novo DataModule, ajuste o nome para **dmPesqCliente** e salve-o como **udmPesqCliente.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...
implementation

uses
  udmPrincipal;
...
```

Em seguida, adicione os seguintes componentes:



TClientDataSet (Data Access)

Name: cdsCliente

ProviderName: dspCliente



TDataSetProvider (Data Access)

Name: dspCliente

DataSet: qryCliente



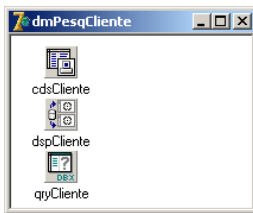
TSQLQuery (dbExpress)

Name: qryCliente

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  CLI_CODIGO,
  CLI_NOME
FROM
  CLIENTE
WHERE
  CLI_NOME CONTAINING :CLI_NOME
```



DataModule de Pesquisa de Cliente

No caso da Pesquisa de Pedidos, havíamos utilizado um SQL dinâmico, já que o usuário poderia pesquisar por diversos campos, já na pesquisa de Clientes, teremos apenas o campo Nome disponível para pesquisa, portanto utilizamos um SQL já fixo no componente recebendo como parâmetro o Nome a ser pesquisado.

O que temos de novidade no SQL é a cláusula **CONTAINING**. O resultado é semelhante a um Like %...%, porém não faz diferenciação entre maiúsculas e minúsculas.

Precisamos ajustar o parâmetro **CLI_NOME**, portanto, clique na propriedade **Params** da Query e ajuste-o da seguinte forma:

DataType: ftString
ParamType: ptInput

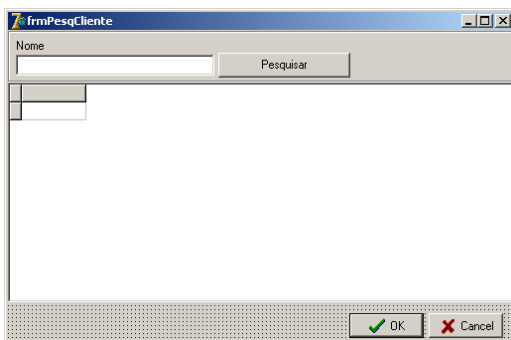
Criando o Formulário

Crie um novo formulário, ajuste o nome para **frmPesqCliente** e salve-o como **ufrmPesqCliente.pas**.

Adicione na cláusula **uses** a unit **udmPesqCliente**.

```
...  
implementation  
  
uses  
    udmPesqCliente;  
...
```

Em seguida, monte o formulário da seguinte forma:



Tela de Pesquisa de Clientes

Nomeie os componentes na respectiva ordem para facilitar nosso acesso mas adiante:

Edit: edtNome

Buttons: btnPesquisar, btnOk, btnCancel

DBGrid: dbgrdClientes

Ajuste a propriedade **BorderStyle** do formulário para **bsDialog**, pois chamaremos como Modal.

Vale lembrar que os botões Ok e Cancelar devem estar com a propriedade **Kind** configurada:

Botão Ok: bkOk

Botão Cancelar: bkCancel

Em seguida adicione um **DataSource**:



TDataSource

Name: dtsCliente

DataSet: dmPesqCliente.cdsCliente

Ajuste as seguintes propriedades do **DBGrid**:

DataSource: dtsCliente

ReadOnly: True

Codifique os eventos do formulário:

OnCreate:

```
procedure TfrmPesqCliente.FormCreate(Sender: TObject);
begin
    dmPesqCliente := TdmPesqCliente.Create(Self);
end;
```

OnDestroy:

```
procedure TfrmPesqCliente.FormDestroy(Sender: TObject);
begin
    dmPesqCliente.Free;
    dmPesqCliente := nil;
end;
```

OnCloseQuery:

```
procedure TfrmPesqCliente.FormCloseQuery(Sender: TObject;
    var CanClose: Boolean);
begin
    if ModalResult = mrOk then
    begin
        if dtsCliente.DataSet.IsEmpty then
        begin
            CanClose := False;
            MessageDlg('Cliente não selecionado', mtInformation, [mbOk], 0);
        end;
    end;
end;
```

Codifique o evento **OnClick** do botão **Pesquisar** da seguinte forma:

```
procedure TfrmPesqCliente.btnPesquisarClick(Sender: TObject);
begin
    dmPesqCliente.cdsCliente.Close;
    dmPesqCliente.cdsCliente.FetchParams;
    dmPesqCliente.cdsCliente.Params.ParamByName('CLI_NOME').AsString :=
edtNome.Text;
    dmPesqCliente.cdsCliente.Open;
end;
```

Precisamos agora codificar o evento **OnDbClick** do **DBGrid** para que a pesquisa também possa ser confirmada dando um duplo clique no registro:

```
procedure TfrmPesqCliente.dbgrdClientesDblClick(Sender: TObject);
begin
    btnOk.Click;
end;
```

Criando o método responsável pela pesquisa e instância do formulário

Criaremos agora nosso método que será responsável em abrir a tela de Pesquisa e retornar o Código do Cliente selecionado.

Na seção **public** do formulário, crie um **método de classe** chamado **Pesquisa**.

```
...
private
{ Private declarations }
public
    class function Pesquisa(varCodigo: Integer): Boolean;
end;
...
```

Implemente-o da seguinte forma:

```
class function TfrmPesqCliente.Pesquisa(varCodigo: Integer): Boolean;
begin
    with TfrmPesqCliente.Create(Application) do
        try
            Result := ShowModal = mrOk;
            if Result then
                Codigo :=
dmPesqCliente.cdsCliente.FieldByName('CLI_CODIGO').AsInteger;
            finally
                Release;
            end;
        end;
    end;
```

Entendendo o código

Utilizamos o mesmo conceito visto na Pesquisa de Pedidos, ou seja, este método criará o formulário de pesquisa e retornará o Código do Cliente selecionado.

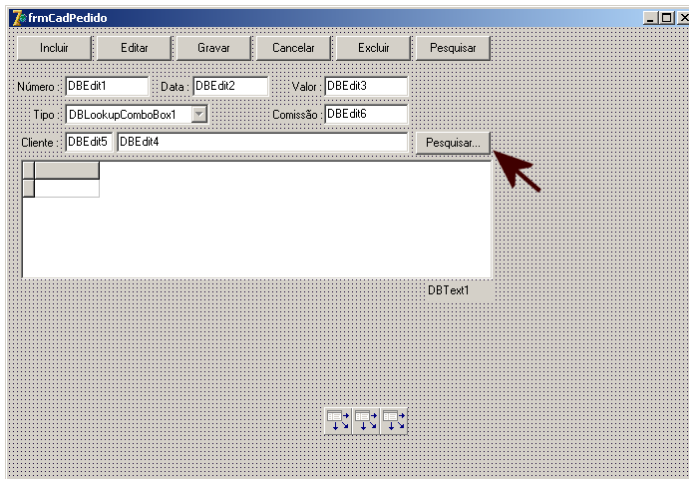
Ajustando Tela de Cadastro de Pedidos

Agora ajustaremos a tela de Pedidos para que possamos ter acesso a Pesquisa de Cliente.

Abra o formulário **frmCadPedido** e adicione na cláusula **uses** a unit **ufrmPesqCliente**.

```
...  
implementation  
  
uses  
    udmCadPedido,  
    ufrmPesqPedido,  
    ufrmPesqCliente;  
...
```

Em seguida inclua um **Botão** ao lado do Nome do Cliente para ser utilizado na chamada da Pesquisa de Clientes e nomeie-o como **btnPesqCliente**.



Tela Cadastro de Pedidos com o botão de Pesquisa para Clientes

Codifique o evento **OnClick** do botão da seguinte forma:

```
procedure TfrmCadPedido.btnPesqClienteClick(Sender: TObject);  
var  
    iCodigo: Integer;  
begin  
    if TfrmPesqCliente.Pesquisa(iCodigo) then  
        begin  
            if not (dmCadPedido.cdsPedido.State in dsEditModes) then  
                dmCadPedido.cdsPedido.Edit;  
            dmCadPedido.cdsPedido.FieldByName('CLI_CODIGO').AsInteger := iCodigo;  
        end;  
    end;  
end;
```

Entendendo o código

Utilizamos o mesmo conceito da Pesquisa de Pedido, apenas chamamos a Pesquisa de Cliente passando a variável local como parâmetro e concluindo a pesquisa, atribuímos ao campo Código do Cliente o valor da variável. Desta forma o evento OnValidate do campo CLI_CODIGO será executado e conseqüentemente teremos o mesmo efeito de termos digitado o Código do Cliente manualmente, ou seja, o Nome será alimentado automaticamente.

O único diferencial está na comparação que fizemos no início, pois o usuário poderia ter clicado no botão de pesquisa do Cliente antes de colocar o Pedido em edição, neste caso, forçamos a edição quando confirmado a pesquisa.

Testando a aplicação

Podemos cadastrar um Pedido informando o código do Cliente manualmente ou através da Pesquisa, veremos que o resultado será o mesmo.

Ordenando Registros

Quando trabalhamos com TTable's, utilizamos a propriedade IndexName ou IndexFieldNames para ordenarmos os registros, porém é necessário que haja um índice físico existente na tabela. Com TQuery fazemos a ordenação utilizando a própria cláusula Order By do SQL, porém se precisarmos ajustar a ordenação diversas vezes, a todo o momento precisaremos refazer e executar a query no servidor de banco de dados, o que poderia gerar lentidão e tráfego na rede.

Com o ClientDataSet resolvemos essas questões facilmente, fazemos a ordenação dos registros em memória, definindo no próprio ClientDataSet, desta forma não fazemos requisição ao servidor de banco de dados e não dependemos dos índices existentes na tabela, conseqüentemente temos menos tráfego de rede, melhor performance e a ordenação é feita em questão de milissegundos, em casos de muitos registros (100.000 por exemplo), alguns segundos.

Existem tem 3 tipos de índices que usamos no ClientDataSet:

Índices padrões

São índices padrões criados e nomeados automaticamente pelo ClientDataSet e representam ordenações específicas:

DEFAULT_ORDER: Representa a ordem original dos registros, da forma que foram abertos no Provider.

CHANGEINDEX: Representa a ordem de atualização dos dados, a mesma ordem que o Provider utilizará para aplicar as atualizações ao banco de dados.

PRIMARY_KEY: Representa a ordenação pela chave primária.

Índices temporários

São definidos através da propriedade **IndexFieldNames**, onde informamos por quais campos os registros serão ordenados. No caso de ser mais de um campo, separamos com ponto-e-vírgula. Estes índices possuem uma limitação, só podemos utilizá-los em ordem ascendente, para índices complexos, utilizamos os persistentes, onde podemos definir o tipo de ordenação, case-insensitive entre outras opções.

São chamados de índices temporários pelo fato de não serem reaproveitáveis, são criados e descartados automaticamente quando trocamos de campo. Exemplo: Definimos o campo PED_NUMERO na propriedade IndexFieldNames, um índice será gerado com este campo. Em seguida ajustamos para o campo PED_DATA, então o índice anterior é descartado e um novo índice é criado com base neste campo. Quando voltarmos a definir o campo PED_NUMERO novamente o índice será recriado, demandando praticamente o mesmo tempo de processo da primeira vez, não havendo reaproveitamento.

Índices Persistentes

São definidos através da propriedade **IndexDefs** no qual podemos criá-los informando outras características, tais como: Ascendente/Descendente, Primário, Case-Insensitive, etc.

Diferente dos índices temporários, eles são reaproveitáveis. Com base no exemplo citado anteriormente, utilizando índices persistentes, ao voltarmos a utilizar o índice com base no campo PED_NUMERO, que já foi processado uma vez, o tempo e os recursos gastos seriam praticamente zero, pois ele é gerado uma única vez e mantido.

Ordenando os Registros – Utilizando Índices Temporários

Implementaremos em nosso projeto um recurso muito interessante que é a ordenação dos registros de acordo com a coluna clicada no DBGrid. Nesta primeira etapa faremos da forma mais simples, utilizando os índices temporários, em seguida implementaremos utilizando os índices persistentes, respeitando a ordenação ascendente e descendente.

Abra o formulário **frmPesqPedido** e no evento **OnTitleClick** do **dbgrdPedido** insira o seguinte código:

```
procedure TfrmPesqPedido.dbgrdPedidoTitleClick(Column: TColumn);  
begin  
    dmPesqPedido.cdsPedido.IndexFieldNames := Column.FieldName;  
end;
```

Entendendo o código

O código é muito simples, apenas ajustamos a propriedade **IndexFieldNames** do **ClientDataSet** de acordo com o nome do campo clicado na coluna.

Vale lembrar que esta propriedade aceita informarmos mais de um campo, para isto basta utilizarmos o caractere ponto-e-vírgula entre eles.

Testando a aplicação

Abrindo a tela de Pesquisa de Pedidos e aplicando uma busca, podemos clicar na coluna na qual queremos indexar e a ordenação será feita instantaneamente, sem a necessidade de acessar o servidor de banco de dados.

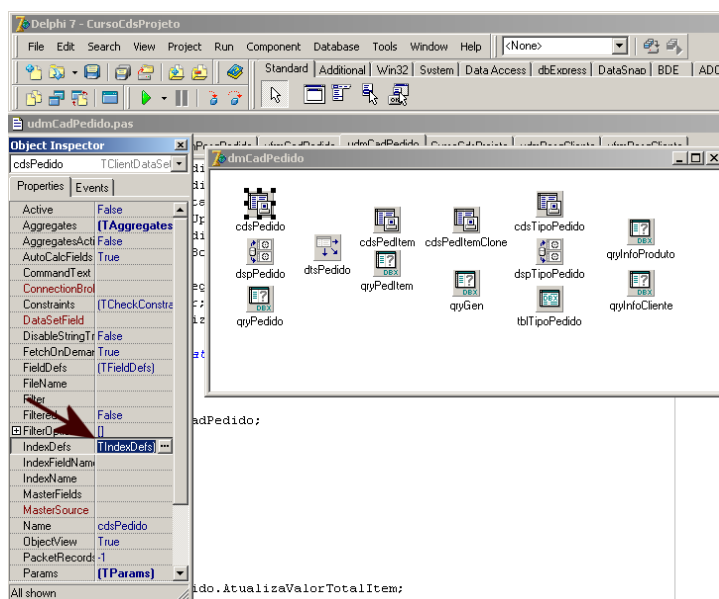
Ordenando os Registros – Utilizando Índices Persistentes

Utilizaremos agora os **Índices Persistentes** para ordenarmos os registros de forma mais complexa, podendo informar o tipo Ascendente ou Descendente.

Como havíamos comentado, eles são definidos através da propriedade **IndexDefs**, porém não iremos criá-los em tempo de projeto, apenas em tempo de execução, pois a ordenação é muito dinâmica, temos além da variação dos campos, o tipo de ordenação (ascendente quando clicado pela primeira vez na coluna e ascendente na segunda), portanto não valeria a pena criar diversos índices deixando-os pronto para serem utilizados.

Para fins didáticos, apenas demonstraremos como seria a criação do índice em tempo de projeto através da propriedade **IndexDefs**, e logo em seguida colocaremos em prática nossa codificação para criação em tempo de execução.

Abra o DataModule **dmPesqPedido** e clique na propriedade **IndexDefs** do componente **cdsPedido**.



Acessando a propriedade **IndexDefs** do **cdsPedido**

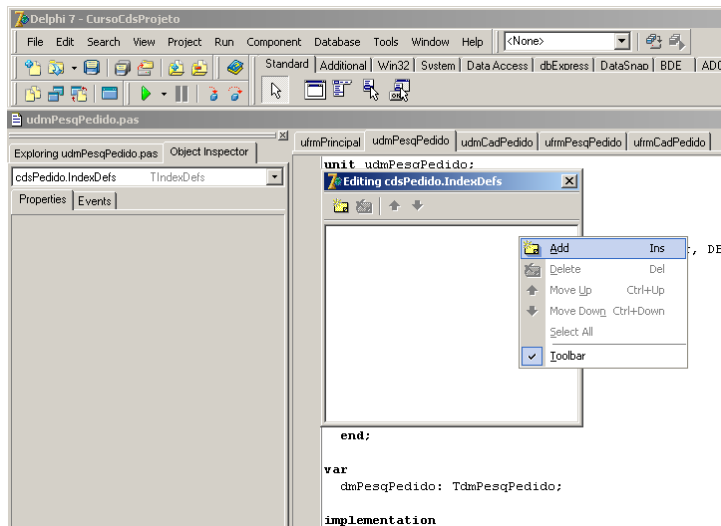
Será exibida a seguinte janela:



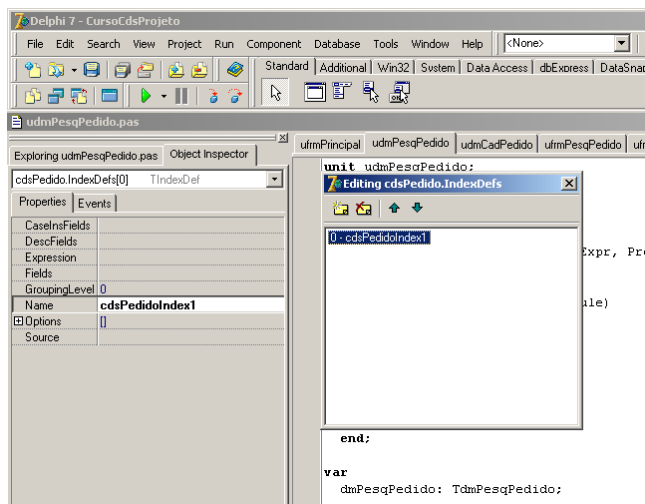
Janela exibida após clicar na propriedade **IndexDefs** do **ClientDataSet**

Clicando com o botão direito do mouse, aparecerá um submenu com a opção **Add**. Clique nesta opção e um novo item será adicionado neste quadro:

Desenvolvendo uma aplicação utilizando ClientDataSet com DBExpress e Firebird



Criando o índice na propriedade IndexDefs do ClientDataSet



Índice criado na propriedade IndexDefs do ClientDataSet

Para cada índice podemos definir as seguintes propriedades:

CaseInsFields: Especifica quais são os campos case-insensitive

DescFields: Especifica quais são os campos que serão ordenados em ordem decendente.

Expression: Utilizado somente para dBASE. Maiores detalhes poderão ser obtidos no Help do Delphi.

Fields: Especifica os campos pertencentes ao índice.

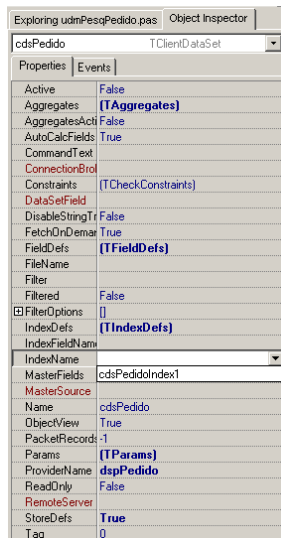
GroupingLevel: Nível de agrupamento, utilizado em conjunto com a propriedade GroupingLevel do ClientDataSet.

Name: Nome do índice.

Options: Determina alguns detalhes do índice (primário, único, case-insensitive, etc.)

Source: Utilizado somente para dBASE. Maiores detalhes poderão ser obtidos no Help do Delphi.

Depois de criado o índice, ele ficará disponível para utilizarmos na propriedade **IndexName** do ClientDataSet:



Índice criado disponível na propriedade *IndexName* do *ClientDataSet*

Como havíamos comentando, não criaremos em tempo de projeto, mas sim em tempo de execução, portanto, remova-o acessando novamente a propriedade *IndexDefs*.

No formulário **frmPesqPedido**, adicione o seguinte código no evento **OnTitleClick** do **dbgrdPedido**:

```
procedure TfrmPesqPedido.dbgrdPedidoTitleClick(Column: TColumn);
var
  sIndexName: string;
  Options: TIndexOptions;
begin
  //dmPesqPedido.cdsPedido.IndexFieldNames := Column.FieldName;

  dmPesqPedido.cdsPedido.IndexDefs.Update;
  if dmPesqPedido.cdsPedido.IndexName = Column.FieldName + '_ASC' then
  begin
    sIndexName := Column.FieldName + '_DESC';
    Options := [ixDescending];
  end
  else
  begin
    sIndexName := Column.FieldName + '_ASC';
    Options := [];
  end;
  if dmPesqPedido.cdsPedido.IndexDefs.IndexOf(sIndexName) < 0 then
    dmPesqPedido.cdsPedido.AddIndex(sIndexName, Column.FieldName,
Options);
  dmPesqPedido.cdsPedido.IndexName := sIndexName;
end;
```

Entendendo o código

Comentamos a primeira linha pois era a forma que utilizávamos para índices temporários. Após temos a seguinte seqüência: Executamos uma atualização na lista dos índices disponíveis, em seguida verificamos se o campo do índice atual é o mesmo que está sendo ordenado e em ordem Ascendente, neste caso ajustamos o nome e as opções do índice para Descendente, se não, utilizamos o nome como Ascendente e as opções em branco, já que a ordenação padrão é Ascendente.

No final verificamos se o índice já existe, ou seja, se já criamos, não encontrado, criamos em seguida, e logo definimos ao ClientDataSet para utilizar o nome do índice que obtivemos.

Testando a aplicação

Abrindo a tela de Pesquisa, após termos os registros disponíveis, podemos clicar a primeira vez na coluna e os registros serão ordenados em ordem Ascendente, clicando pela segunda vez, em ordem Descendente.

Filtrando Registros em Memória

Da mesma forma que aplicamos filtros nas TTables utilizando a propriedade Filter, evento OnFilterRecord, Ranges, etc., aplicamos também ao ClientDataSet, a grande diferença está que, nas TTables por exemplo, os filtros eram aplicados com base em todos os registros existentes fisicamente na tabela, enquanto que no ClientDataSet, será aplicado apenas aos registros que estão em memória.

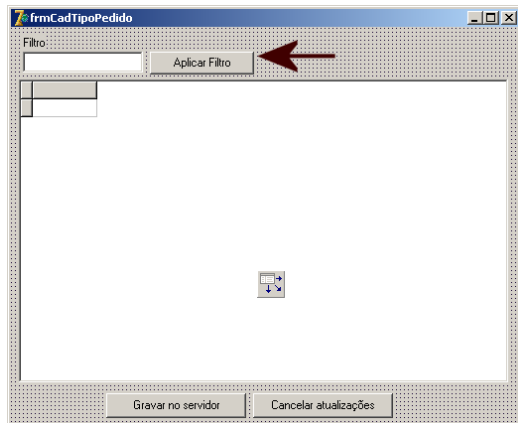
Outro diferencial é que utilizando Ranges em TTables por exemplo, por ele depender de um índice, somos obrigados a tê-lo fisicamente na tabela, com o ClientDataSet continuamos dependendo do índice, porém não fisicamente, criamos através da propriedade IndexDefs ou utilizamos a propriedade IndexFieldNames, como vimos anteriormente.

Com base nesses conceitos, concluímos que os registros podem ser restringidos em 2 etapas, primeiramente no próprio SQL da Query utilizando a cláusula WHERE para evitarmos tráfego na rede, e em seguida, localmente, no próprio ClientDataSet, depois de os registros já terem sido trafegados e estarem em memória.

Aplicaremos este recurso de duas formas, utilizando a propriedade **Filter** que como vimos, não dependemos de índice, e utilizando **Ranges**, que neste caso, teremos de criar o respectivo índice.

Filtrando os Registros em Memória – Utilizando a Propriedade Filter

Abra o formulário **frmCadTipoPedido** e ajuste-o deixando da seguinte forma:



Formulário de Cadastro de Tipos de Pedido com opção de Filtro

Ajuste o nome dos componentes adicionados da seguinte forma:

Edit: edtFiltro

Button: btnAplicarFiltro

Codifique o evento **OnClick** do botão **btnAplicarFiltro** da seguinte forma:

```
procedure TfrmCadTipoPedido.btnAplicarFiltroClick(Sender: TObject);
begin
    dmCadTipoPedido.cdsTipoPedido.Filter := edtFiltro.Text;
    dmCadTipoPedido.cdsTipoPedido.Filtered := True;
end;
```

Entendendo o código

O código é bem simples, como havíamos comentado, usamos da mesma forma que aplicamos nas TTables por exemplo, portanto, definimos a propriedade **Filter** com o conteúdo do **Edit** e logo ligamos o filtro ativando a propriedade **Filtered**.

Testando a aplicação

Execute a aplicação, abra a tela de Cadastro de Tipos de Pedido e no campo de Filtro, digite por exemplo:

TP_DESCRICAO = 'A*'

Ao clicarmos no botão **Aplicar Filtro**, os registros serão restringidos de acordo com o Filtro, neste caso, apenas os registros cuja descrição comece com a letra A serão exibidos.

Vale lembrar que o filtro está sendo aplicado localmente, em memória, nenhum acesso ao servidor está sendo feito.

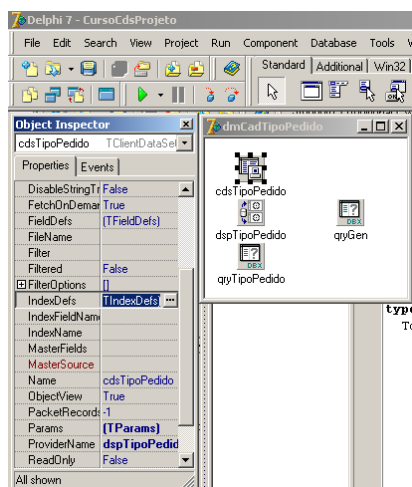
Filtrando os Registros em Memória – Utilizando Range

Como havíamos comentando, para utilizarmos este recurso, temos que ter um índice que contenha os campos que utilizaremos no range, portanto, podemos utilizar a propriedade **IndexFieldNames** ou **IndexName**.

Utilizaremos a segunda para fins didáticos e também por ter uma melhor performance quando o número de registros é muito grande, portanto, precisaremos criar o respectivo índice para ser utilizado na propriedade **IndexName**.

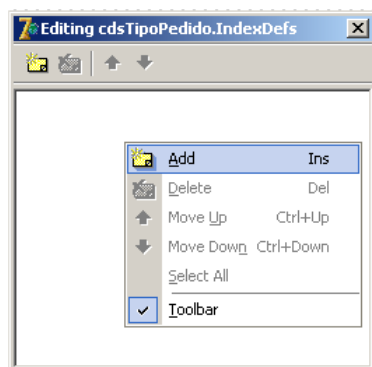
Colocando em prática

Abra o DataModule **dmCadTipoPedido**, selecione o componente **cdsTipoPedido** e clique na propriedade **IndexDefs**.



Acessando a propriedade *IndexDefs* do *cdsTipoPedido*

Em seguida será aberto o Editor de Índices, clique com o botão direito do mouse na janela e clique no item **Add**.

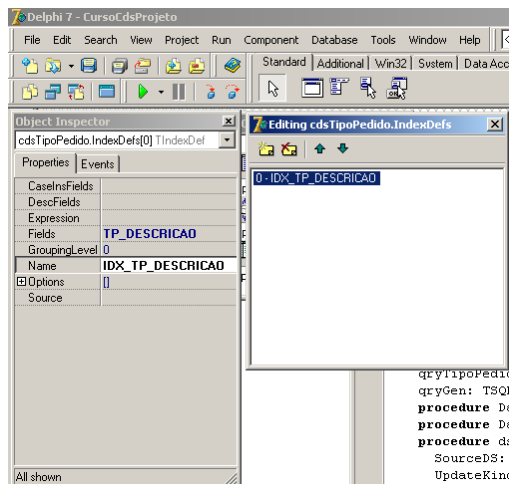


Criando o índice no *cdsTipoPedido*

Depois de adicionado, ajuste-o da seguinte forma:

Fields: TP_DESCRICA0

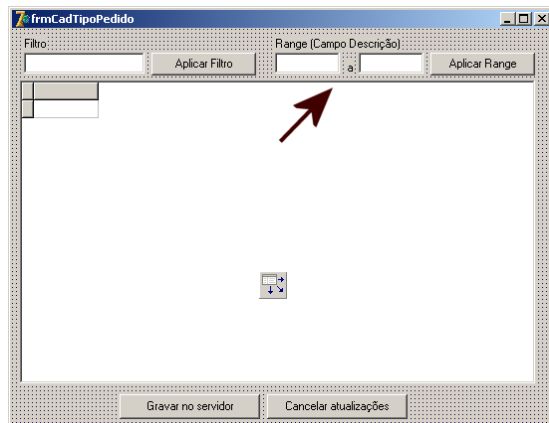
Name: IDX_TP_DESCRICA0



Ajustando o índice no cdsTipoPedido

Ajustando o formulário de Cadastro

Abra o formulário **frmCadTipoPedido** e adicione os componente para aplicarmos o range:



Formulário de Cadastro de Tipos de Pedidos com Range

Ajuste o nome dos componentes adicionados:

Edits: edtDescricaoInicial e edtDescricaoFinal

Button: btnAplicarRange

Codifique o evento **OnClick** do botão **Aplicar Range** da seguinte forma:

```
procedure TfrmCadTipoPedido.btnRangeClick(Sender: TObject);
begin
  dmCadTipoPedido.cdsTipoPedido.IndexName := 'IDX_TP_DESCRICAO';
  dmCadTipoPedido.cdsTipoPedido.SetRange([edtDescricaoInicial.Text],
[edtDescricaoFinal.Text]);
end;
```

Entendendo o código

Na primeira linha estamos informando o índice que deverá ser utilizado pelo Range, e logo em seguida aplicamos utilizando o método SetRange, onde no primeiro parâmetro informamos o valor inicial e no segundo, o valor final.

Vale lembrar que o método SetRange foi utilizado apenas para fins didático, mas poderíamos ter o mesmo resultado utilizando por exemplo, os métodos SetRangeStart, SetRangeEnd e ApplyRange.

Testando a Aplicação

Podemos testar por exemplo, informando a letra A no primeiro campo e a letra B no segundo, ao clicarmos no botão **Aplicar Range**, veremos que somente registros cuja descrição estiverem neste intervalo serão exibidos.

Pesquisando os Registros em Memória

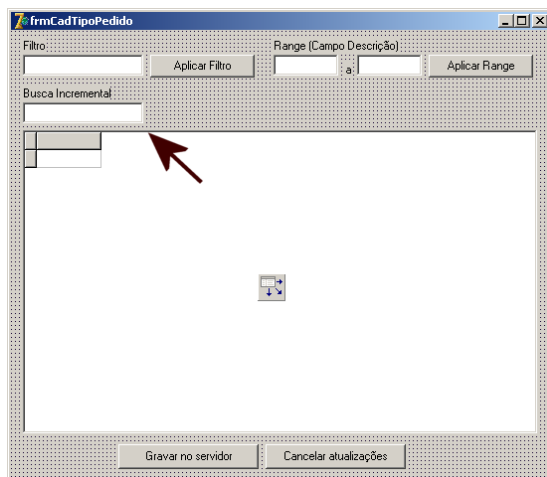
Da mesma forma que podemos filtrar os registros em memória, podemos também localizar utilizando os métodos que já conhecemos, tais como: Locate, FindKey, FindNearest entre outros.

Segue-se o mesmo conceito de Filtros, a pesquisa é feita nos registros em memória, ou seja, podemos dizer que também temos as 2 etapas de pesquisa, aplicamos o WHERE na Query para tráfegarmos somente os registros necessários, e depois de estarem em memória, podemos fazer uma busca entre eles. Um caso típico é aquele onde o usuário faz uma busca incremental por exemplo, conforme vai digitando, o cursor vai se posicionando no registro.

Simularemos este caso no Cadastro de Tipos Pedidos, teremos um campo para que o usuário possa digitar a descrição e buscaremos o registro utilizando o método FindNearest.

Colocando em prática

Abra o formulário **frmCadTipoPedido** e adicione os componentes para busca incremental:



Cadastro de Tipos de Pedidos com Busca Incremental

Ajuste o nome do Edit adicionado para **edtBuscaIncremental**.

No evento **OnKeyUp** do Edit, adicione o seguinte código:

```
procedure TfrmCadTipoPedido.edtBuscaIncrementalKeyUp(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  dmCadTipoPedido.cdsTipoPedido.IndexFieldNames := 'TP_DESCRICAO';
  dmCadTipoPedido.cdsTipoPedido.FindNearest([edtBuscaIncremental.Text]);
end;
```

Entendendo o código

No código não há diferença alguma comparando com a utilização em TTables, primeiro indexamos pelo campo a ser pesquisado e em seguida, realizamos a busca.

Vale lembrar que neste caso, estamos utilizando a propriedade `IndexFieldNames`, mas poderíamos também utilizar o índice que já temos criado para este campo, utilizando a propriedade `IndexName`.

Testando a aplicação

Abrindo a tela de Cadastro de Tipos de Pedidos, primeiramente os registros serão trafegados, depois de disponíveis no `ClientDataSet`, podemos digitar algo no campo de busca incremental e veremos o cursor se posicionando no registro que contém a descrição aproximada ao que foi digitado.

O importante saber é que nesta busca, nenhum acesso físico ao banco de dados está sendo feito, como dissemos, a busca é feita em memória, por isso temos o resultado de forma instantânea.

Trabalhando com Refresh

O método **Refresh** que utilizamos nas TTables, também está disponível no ClientDataSet, e além deste, temos um outro chamado **RefreshRecord**, utilizado para atualizar apenas o registro atual, diferente do **Refresh** que atualiza todos os registros.

Refresh

Quando utilizamos o método Refresh, internamente o Provider executa um **Fetch** no banco de dados para cada registro, ele verifica as mudanças ocorridas e logo atualiza o ClientDataSet com as novas informações.

Chamar o método Refresh é muito mais rápido do que fecharmos e abriremos o ClientDataSet, pois desta forma, será executado todo processo de requisição dos dados e assim os registros serão novamente trafegados pela rede, demandando um maior tempo.

A utilização deste método não permite que haja pendências no ClientDataSet, se tentar executar e alguma atualização não tiver sido aplicada ainda ao banco de dados, uma mensagem de erro será exibida e o método não será executado.

RefreshRecord

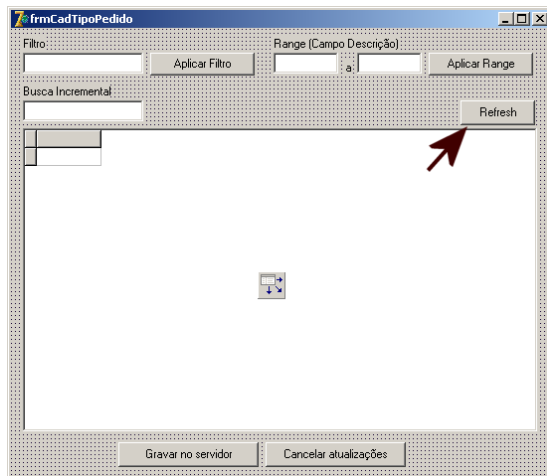
Este método atualiza apenas o registro atual diferente do Refresh que atualiza todos os registros. Outro grande diferencial é que, neste caso, não é realizado um Fetch no banco, mas sim um novo SELECT, porém baseado na chave primária para que apenas o registro atual possa ser trafegado e atualizado.

Para que este método possa ser utilizado, precisamos ajustar o ProviderFlags do campo que pertence à chave primária ligando a opção pflnKey, para que assim o Provider possa montar corretamente o SELECT e extrair o registro do banco de dados.

Utilizando este método não será gerada uma exceção caso haja pendências no registro, porém o mesmo não será atualizado, o log de alterações é mantido.

Trabalhando com Refresh – Utilizando o Método Refresh

Abra o formulário **frmCadTipoPedido** e apenas adicione o botão **Refresh**, nomeando-o como **btnRefresh**.



Cadastro de Tipos de Pedido com Refresh

Codifique o evento **OnClick** do botão da seguinte forma:

```
procedure TfrmCadTipoPedido.btnRefreshClick(Sender: TObject);  
begin  
    dmCadTipoPedido.cdsTipoPedido.Refresh;  
end;
```

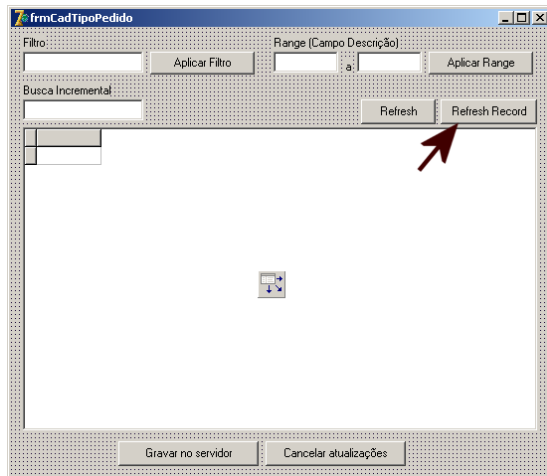
Testando a aplicação

Para verificarmos o resultado, simularemos 2 usuários visualizando a mesma tabela, o primeiro usuário fará as alterações e o segundo visualizará após chamar o Refresh.

Instancie duas vezes a aplicação abrindo a tela de Cadastro de Tipos de Pedido. Na primeira instância, faça inclusões, alterações e exclusões de registros, em seguida aplique clicando no botão **Gravar no servidor**. Ao voltarmos para segunda instância, veremos que os dados permanecem os mesmos, então clique no botão **Refresh** e veja que as informações serão atualizadas de acordo com as novas mudanças feitas no banco de dados, e melhor, instantaneamente, pois como dissemos, esta sendo executado apenas um Fetch nos registros.

Trabalhando com Refresh – Utilizando o Método RefreshRecord

Abra o formulário **frmCadTipoPedido** e apenas adicione o botão **RefreshRecord**, nomeando-o como **btnRefreshRecord**.



Cadastro de Tipos de Pedido com RefreshRecord

Codifique o evento **OnClick** do botão **RefreshRecord** da seguinte forma:

```
procedure TfrmCadTipoPedido.btnRefreshRecordClick(Sender: TObject);  
begin  
    dmCadTipoPedido.cdsTipoPedido.RefreshRecord;  
end;
```

Testando a aplicação

Ao clicarmos no botão **RefreshRecord**, será exibida a seguinte mensagem de erro:



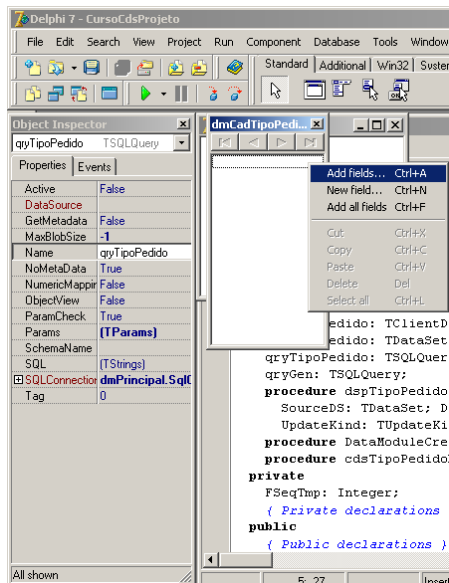
Mensagem de erro ao clicar no botão RefreshRecord

Como havíamos comentado, quando executamos este método, internamente o Provider precisa gerar um SELECT para extrair apenas o registro atual e assim poder atualizar o ClientDataSet, portanto, ele necessita saber qual é a chave primária para poder montar corretamente a Query. Fazemos isso ajustando a propriedade **ProviderFlags** do campo (da Query) chave ligando a opção **pfInKey**.

Ajustando o ProviderFlags – Definindo a chave

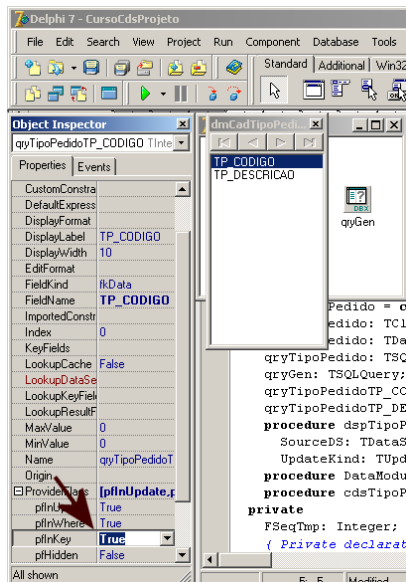
Abra o DataModule **dmCadTipoPedido** e em seguida de um duplo clique no componente **qryTipoPedido** para que seja aberto o **FieldsEditor**.

Adicione todos os campos clicando com o botão direito em seguida clicando no item **Add all fields**.



Adicionando os campos na qryTipoPedido

Em seguida, selecione o campo **TP_CODIGO** e na propriedade **ProviderFlags**, defina **True** na opção **pflnKey**.



Ligando a opção **pflnKey** no campo **TP_CODIGO** da Query

Testando a aplicação

Executando a aplicação e clicando no botão **RefreshRecord**, veremos que o erro não será mais exibido.

Para verificarmos a diferença entre o método **RefreshRecord** e **Refresh**, podemos simular a mesma situação que fizemos utilizando o **Refresh** e veremos que somente o registro atual será atualizado.

Desfazendo Alterações – CancelUpdates, UndoLastChange, RevertRecord e SavePoint

O ClientDataSet disponibiliza diversos métodos para desfazermos alterações, cada um com suas características próprias:

CancelUpdates

Como já vimos, utilizamos este método no botão Cancelar do Cadastro de Pedidos. Sua finalidade é de cancelar tudo que há pendente (não aplicado) no ClientDataSet.

UndoLastChange (FollowChange: Boolean): Boolean;

Desfaz a última inclusão, alteração ou exclusão ocorrida no ClientDataSet e retorna True caso tenha sido restaurado com sucesso. O parâmetro **FollowChange** Indica se o cursor deverá ser posicionado no registro restaurado.

RevertRecord

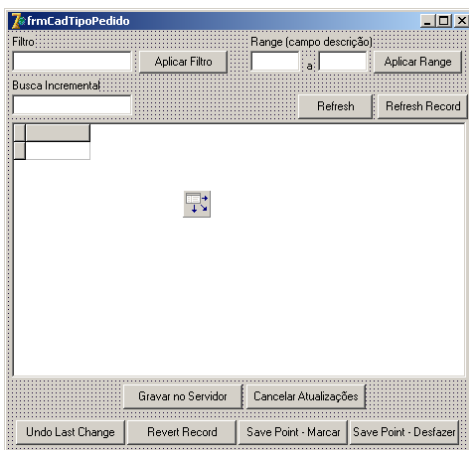
Desfaz as alterações ocorridas no registro atual, voltando ao seu estado original.

SavePoint

Não é um método e sim uma propriedade que nos permite desfazer as operações a partir de um determinado ponto.

Colocando em prática

Abra o formulário **frmCadTipoPedido** e adicione 4 botões:



Cadastro de Tipos de Pedido com os botões para desfazer alterações

Ajuste o nome dos botões na respectiva ordem:

btnUndoLastChange, btnRevertRecord, btnSavePointMarcaPonto, btnSavePointDesfaz

Codificaremos os eventos **OnClick** dos botões:

UndoLastChange:

```
procedure TfrmCadPedido.btnUndoLastChangeClick(Sender: TObject);  
begin  
    dmCadTipoPedido.cdsTipoPedido.UndoLastChange(True);  
end;
```

Estamos passando True no parâmetro, indicando que o cursor deverá ser posicionando no registro restaurado.

RevertRecord:

```
procedure TfrmCadPedido.btnRevertRecordClick(Sender: TObject);
begin
    dmCadTipoPedido.cdsTipoPedido.RevertRecord;
end;
```

Save Point - Marcar:

```
procedure TfrmCadPedido.btnSavePointMarcaPontoClick(Sender: TObject);
begin
    FSavePoint := dmCadTipoPedido.cdsTipoPedido.SavePoint;
end;
```

Como dissemos, SavePoint é uma propriedade do ClientDataSet. Ela é do tipo Integer e nos retorna a situação atual do ClientDataSet. Guardamos o valor em uma variável global, em seguida podemos fazer as alterações nos dados e ao final, se precisarmos, podemos voltar à situação em que estava no momento que obtivemos o valor de SavePoint, atribuindo a esta propriedade o valor da variável global. Vejamos sua utilização no botão abaixo.

SavePoint - Desfazer:

```
procedure TfrmCadPedido.btnSavePointDesfazClick(Sender: TObject);
begin
    dmCadTipoPedido.cdsTipoPedido.SavePoint := FSavePoint;
end;
```

Declare a variável global **FSavePoint** na seção **private** do formulário:

```
...
private
    { Private declarations }
    FSavePoint: Integer;
public

end;
...
```

Testando a aplicação

Podemos abrir a tela de Cadastro de Tipos de Pedido, alterar os registros e verificar o efeito de cada botão. O mais interessante é o **SavePoint**, pois podemos clicar para marcar o ponto, fazer as mudanças e depois clicamos no botão **Save Point – Desfazer** para voltar ao estado que marcamos.

Lendo e Gravando os Dados Localmente em Arquivos

Podemos estar gravando os dados do ClientDataSet não apenas no servidor de banco de dados, mas também em arquivos locais, no formato binário ou XML.

Um caso muito comum de utilização é para tabelas auxiliares, utilizadas como Lookup, que sofrem pouca manutenção e dependendo o caso, vale a pena além de mantê-la no servidor, salvá-la localmente nas máquinas dos usuários, pois assim, quando fossemos utilizá-la em um TDBLookupComboBox por exemplo, carregariamos localmente a partir do arquivo, evitando tráfego na rede.

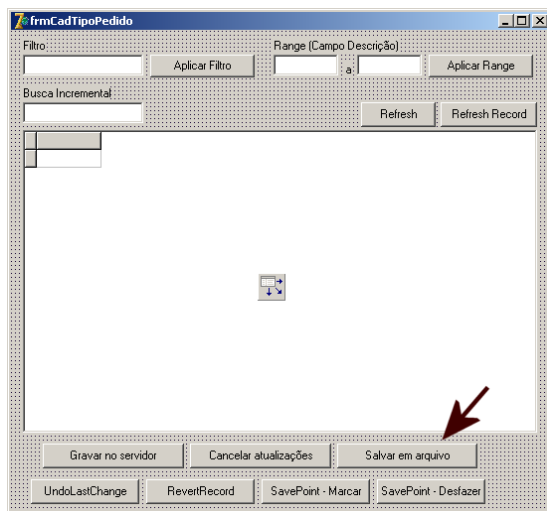
Claro que devemos pensar muito bem antes de adotarmos esta solução, pois isso envolve termos um controle de atualização das máquinas quando esta tabela fosse atualizada no servidor.

Em nosso projeto aplicaremos este recurso com base nesta idéia, mas é importante sabermos que existem muitos outros casos que podemos utilizar, por exemplo, quando trabalhamos com WebServices, onde precisamos salvar e carregar os dados em formato XML.

Nossa idéia será ajustar a tela de Cadastro de Tipos de Pedido, permitindo também gravar os dados localmente em arquivo, pois assim ajustaremos nosso DataModule de Pedidos para carregar o ClientDataSet de Tipo de Pedidos a partir deste arquivo e não mais diretamente do servidor.

Gravando o arquivo

Abra o formulário **frmCadTipoPedido** e adicione o botão **Salvar em Arquivo** e ajuste o nome para **btnSalvarEmArquivo**.



Cadastro de Tipos de Pedido com botão Salvar em arquivo

No evento **OnClick** do botão, insira o seguinte código:

```
procedure TfrmCadTipoPedido.btnSalvarEmArquivoClick(Sender: TObject);  
begin  
    dmCadTipoPedido.cdsTipoPedido.SaveToFile('c:\tipopedido.cds');  
end;
```

O método **SaveToFile** é o responsável por gravar os dados em arquivo. Ele recebe 2 parâmetros:

FileName: string = Caminho do arquivo.

Format: TDataPacketFormat = Formato do arquivo. O padrão é dfBinary, por isso não informamos esse parâmetro, pois é o formato que desejamos gravar. Outros possíveis formatos seriam para XML: dfXML, dfXMLUTF8.

Testando a aplicação

Executando a aplicação, ao clicarmos no botão **Salvar em arquivo**, podemos perceber que o arquivo será gerado no caminho que especificamos, c:\tipopedido.cds. Este arquivo contém a estrutura da tabela com os respectivos registros, faremos sua leitura no próprio ClientDataSet.

Lendo o Arquivo

O que precisamos fazer agora é carregá-lo em um ClientDataSet, como dissemos, faremos isso no Lookup de Tipo de Pedido no DataModule de Cadastro de Pedidos.

Podemos ler o arquivo de 2 formas:

- Utilizando o método LoadFromFile do ClientDataSet

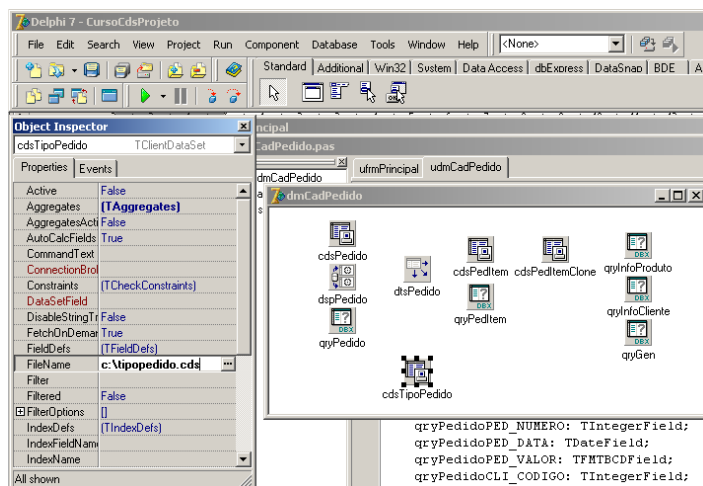
```
cdsTipoPedido.LoadFromFile('c:\tipopedido.cds')
```

Este método possui apenas um parâmetro, que indica o caminho do arquivo. Se deixarmos em branco, ele utilizará o caminho especificado na propriedade FileName do ClientDataSet.

- Ajustando a propriedade FileName do ClientDataSet

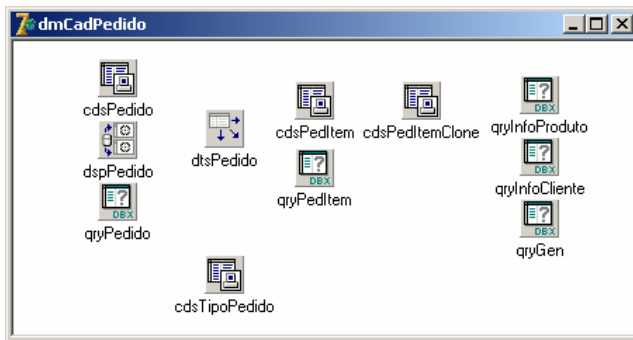
Esta é a forma mais simples, apenas ajustamos a propriedade FileName indicando o caminho do arquivo e ao abrirmos o ClientDataSet, os dados serão carregados diretamente do arquivo.

Utilizaremos a segunda opção, portanto abra o DataModule **dmCadPedido** e selecione o componente **cdsTipoPedido**. Na propriedade **FileName** informe o caminho **c:\tipopedido.cds**.



Ajustando propriedade FileName no cdsTipoPedido

Como estaremos lendo os dados a partir de um arquivo, não precisaremos mais acessar o servidor de banco de dados, portanto, podemos remover o Provider e a Query relativa ao Tipo de Pedido. Portanto, **limpe a propriedade ProviderName** do **cdsTipoPedido** e **remova os componentes dspTipoPedido e qryTipoPedido**.



DataModule de Cadastro de Pedidos sem o Provider/Query para o Tipo de Pedido

Testando a aplicação

Ao entrarmos no Cadastro de Pedidos, teremos o Lookup disponível como estava antes, porém, para termos a certeza de que está sendo utilizado o arquivo, entre na tela de Cadastro de Tipos de Pedidos e faça modificações na tabela gravando apenas no servidor. Ao entrarmos no Cadastro de Pedidos novamente, veremos que o Lookup permanece como estava, portanto, entre novamente no Cadastro de Tipos de Pedidos e clique no botão **Salvar em arquivo**, depois confira se o Lookup estará atualizado na tela de Cadastro de Pedidos.

Trabalhando com Tabelas Somente em Memória

O ClientDataSet nos disponibiliza uma forma de trabalharmos com tabelas somente em memória definindo a estrutura dos campos no próprio ClientDataSet, sem necessitarmos de uma tabela física, conexão com o servidor, etc.

Este recurso é muito bem utilizado naqueles casos onde temos relatórios complexos, no qual temos que unir informações de diversas tabelas e montar os dados em uma tabela temporária. Trabalhando-se com Paradox, precisaríamos criá-las fisicamente, já com o ClientDataSet não é necessário, definimos a estrutura da tabela no próprio ClientDataSet e alimentamos os dados em memória, podendo ser utilizando normalmente em um relatório.

Para colocarmos este recurso em prática, criaremos uma tela onde o usuário informará um período e alimentaremos o ClientDataSet com o valor total de vendas de cada dia dentro do período informado. Obviamente que conseguiríamos este resultado de maneira simples através de uma instrução SQL, porém faremos desta forma apenas para fins didáticos, pois assim veremos a criação da estrutura e como alimentamos o ClientDataSet para podermos visualizá-lo em um DBGrid por exemplo.

Colocando em prática

Crie um novo DataModule, ajuste o nome para **dmResumoDiario** e salve-o como **udmResumoDiario.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...  
implementation  
  
uses  
    udmPrincipal;  
...
```

Em seguida, adicione os seguintes componentes:



TClientDataSet (Data Access)

Name: cdsResumo



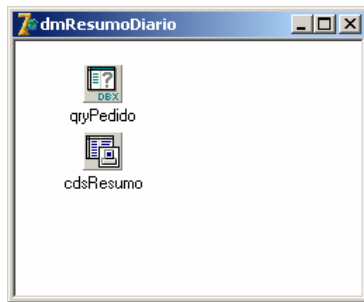
TSQLQuery (dbExpress)

Name: qryPedido

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT  
    PED_DATA,  
    PED_VALOR  
FROM  
    PEDIDO  
WHERE  
    PED_DATA BETWEEN :PED_DATAINI AND :PED_DATAFIM
```



DataModule do Resumo Diário

Nossa query será responsável apenas em obter a data e o valor dos pedidos dentro do período. Criamos para isso dois parâmetros, data inicial e data final, portanto precisamos ajustá-los. Clique na propriedade **Params** do componente **qryPedido** e ajuste-os da seguinte forma:

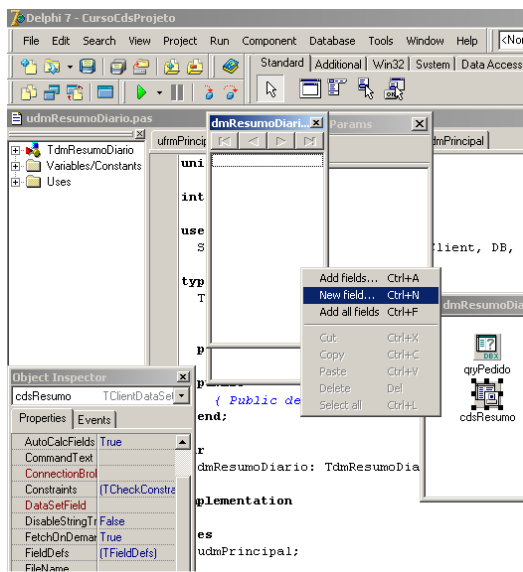
DataType: ftDate

ParamType: ptInput

Definindo a estrutura do ClientDataSet

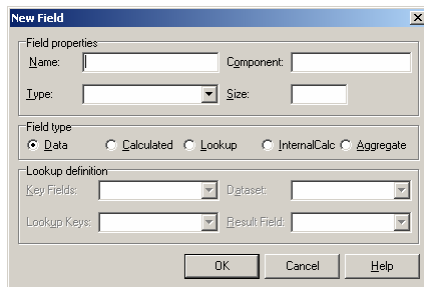
Definiremos agora os campos da nossa tabela 'temporária', para isso utilizaremos o **FieldsEditor** do ClientDataSet, portanto de um duplo clique no componente **cdsResumo**.

Com o **FieldsEditor** aberto, clique com o botão direito do mouse e acesse a opção **NewField**.



Criando campo no cdsResumo

A janela de criação de campos será aberta:



Criando campos no cdsResumo

Criaremos 2 campos com a seguinte estrutura:

Name: DATA
Type: Date
FieldType: Data

Name: VALOR
Type: Float
FieldType: Data

Feito isto, teremos o FieldsEditor com os 2 campos criados:



FieldsEditor com os 2 campos criados

Neste momento temos o ClientDataSet estruturado, precisamos agora implementar a rotina que o alimentará.

Alimentando os dados no ClientDataSet

Criaremos um método no DataModule responsável por isso, para chamarmos através do formulário.

Na seção **public** do DataModule, adicione um método chamado **ProcessaResumo**.

```
private
{ Private declarations }
public
  procedure ProcessaResumo(DataIni, DataFim: TDateTime);
end;
```

Implemente-o da seguinte forma:

```
procedure TdmResumoDiario.ProcessaResumo(DataIni, DataFim: TDateTime);
begin
  if cdsResumo.Active then
    while not cdsResumo.IsEmpty do
      cdsResumo.Delete
    else
      cdsResumo.CreateDataSet;

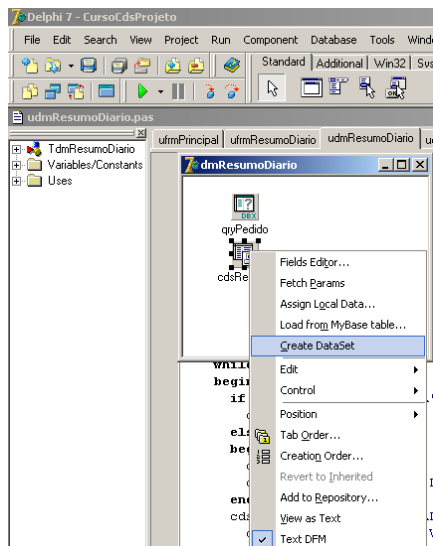
  qryPedido.ParamByName('PED_DATAINI').Value := DataIni;
  qryPedido.ParamByName('PED_DATAFIM').Value := DataFim;
  qryPedido.Open;
  try
    while not qryPedido.Eof do
      begin
        if cdsResumo.Locate('DATA', qryPedido.FieldByName('PED_DATA').Value, [])
then
          cdsResumo.Edit
        else
          begin
            cdsResumo.Append;
            cdsResumo.FieldByName('DATA').Value :=
qryPedido.FieldByName('PED_DATA').Value;
          end;
            cdsResumo.FieldByName('VALOR').AsFloat :=
            cdsResumo.FieldByName('VALOR').AsFloat +
            qryPedido.FieldByName('PED_VALOR').AsFloat;
            cdsResumo.Post;
            qryPedido.Next;
          end;
        finally
          qryPedido.Close;
        end;
      end;
end;
```

Entendendo o código

O código é um pouco extenso, mas não temos muito o que comentar, é pura programação do dia a dia., apenas fizemos um loop na query e alimentamos o ClientDataSet na sequência.

Perceba que não utilizamos o método Open ou a propriedade Active para abrir o ClientDataSet, utilizamos o método **CreateDataSet**, que é responsável em criar o DataSet com sua estrutura e em seguida abrí-lo.

Se precisarmos abrir o ClientDataSet em tempo de projeto, podemos executar este método clicando com o botão direito do mouse sobre o componente e acessando o item CreateDataSet.



Acessando o CreateDataSet em tempo de projeto

Um fato interessante é que não precisamos utilizar um ClientDataSet/Provider em conjunto com a Query de Pedidos para obter os dados, como dissemos, os componentes DBExpress são unidirecionais, somente vão a uma direção, para frente, por isso podemos usar o método Next sem problemas, porém jamais poderíamos chamar o Prior, pois DBExpress não faz cache.

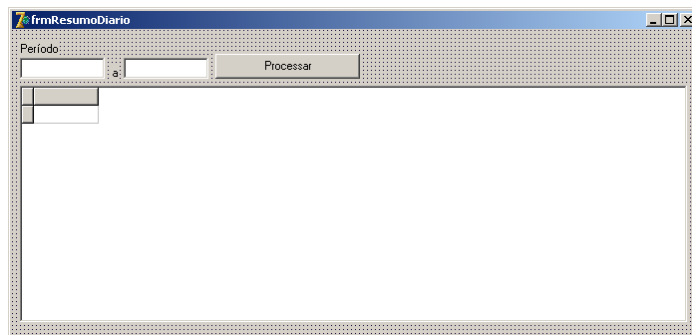
Criando o Formulário

Crie um novo formulário, ajuste o nome para **frmResumoDiario** e salve-o como **ufrmResumoDiario.pas**.

Adicione na cláusula **uses** a unit **udmResumoDiario**.

```
...  
implementation  
  
uses  
    udmResumoDiario;  
...
```

Em seguida, monte o formulário da seguinte forma:



Tela de Resumo Diário

Ajuste o nome dos componentes na respectiva ordem:

Edits: edtDataIni e edtDataFim

Button: btnProcessar

DBGrid: dbgrdResumo

Adicione um **DataSource**:



TDataSource (Data Access)

Name: dtsResumo

DataSet: dmResumoDiario.cdsResumo

Ajuste a propriedade **DataSource** do **DBGrid** para **dtsResumo**.

No evento **OnClick** do botão **Processar**, inclua o seguinte código:

```
procedure TfrmResumoDiario.btnProcessaClick(Sender: TObject);
begin
    dmResumoDiario.ProcessaResumo(StrToDate(edtDataIni.Text),
    StrToDate(edtDataFim.Text));
end;
```

Precisamos agora criar o **DataModule**, portanto, faremos isso no evento **OnCreate** do formulário:

```
procedure TfrmResumoDiario.FormCreate(Sender: TObject);
begin
    dmResumoDiario := TdmResumoDiario.Create(Self);
end;
```

No evento **OnDestroy**, liberamos:

```
procedure TfrmResumoDiario.FormDestroy(Sender: TObject);
begin
    dmResumoDiario.Free;
    dmResumoDiario := nil;
end;
```

No evento **OnClose**, liberamos o formulário:

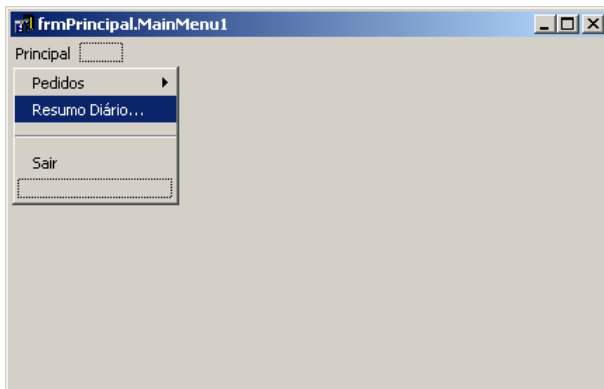
```
procedure TfrmResumoDiario.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    Action := caFree;
    frmResumoDiario := nil;
end;
```

Ajustando o Formulário Principal

Abra o formulário **frmPrincipal** e adicione na cláusula **uses** a unit **ufrmResumoDiario**.

```
...
implementation
uses
    ufrmCadPedido,
    ufrmCadTipoPedido,
    ufrmPesqPedido,
    ufrmResumoDiario;
...
```

Em seguida, no **MainMenu** crie um item chamado **Resumo Diário**.



Adicionando o item de menu *Resumo Diário*

No evento **OnClick** do item insira o seguinte código:

```
procedure TfrmPrincipal.ResumoDiarioClick(Sender: TObject);
begin
    if not Assigned(frmResumoDiario) then
        frmResumoDiario := TfrmResumoDiario.Create(Self);
    frmResumoDiario.Show;
end;
```

Testando a aplicação

Podemos testar a aplicação cadastrando primeiramente alguns Pedidos com datas comuns, em seguida, entrando na tela de Resumo Diário, veremos os dados processados, gravados no ClientDataSet que criamos somente em memória, podemos inclusive editá-los normalmente pelo DBGrid.

Vale lembrar que muito dos recursos que vimos até agora, poderiam ser aplicados a este ClientDataSet, como por exemplo, ordenação dos registros, filtros, busca, salvar em arquivos, etc.

Transações

Trabalhar com transações não é algo específico do ClientDataSet ou DBExpress, fazemos isso em qualquer Engine de acesso, inclusive no BDE utilizando Paradox, dependemos apenas do banco de dados suportar, a grande maioria suporta.

Para quem não conhece pode parecer um termo estranho, mas o conceito é bem interessante.

Supondo que temos de executar diversas rotinas de atualizações no banco de dados. Por exemplo, realizamos uma venda e precisamos executar 3 processos:

- Baixar Estoque
- Gerar Nota
- Gerar Duplicatas

Temos que gerar os 3 processos obrigatoriamente, se um deles falhar, precisamos desfazer tudo que foi feito com sucesso, não poderíamos baixar o estoque e deixar de gerar nota por exemplo.

Neste caso, ao invés de fazermos um trabalho 'braçal' de desfazer tudo o que foi feito caso algum erro tenha ocorrido, optamos em trabalhar com transações. Iniciamos uma transação, fazemos todas as modificações no banco de dados e ao final aplicamos (commit) ou descartamos (rollback) as alterações, tudo com base na transação iniciada.

Quando iniciamos uma transação e fazemos alterações no banco de dados, só serão gravadas efetivamente se confirmarmos com um Commit, caso contrário, tudo o que foi feito será descartado.

Como já comentamos o ClientDataSet trabalha com transações automaticamente, sempre que chamamos o método ApplyUpdates o Provider inicia uma transação, aplica as mudanças ao banco de dados e ao final, se tudo foi aplicado com sucesso, executa um Commit para confirmar, caso contrário, se houve algum erro, executa um Rollback para desfazer tudo que foi feito na transação. Claro que isso dependerá muito do parâmetro que passamos ao método ApplyUpdates indicando o número de erros permitidos, como já havíamos comentado.

Na maioria dos cadastros dificilmente precisaremos trabalhar manualmente com transações, será muito comum sua utilização em algum módulo específico para executar alguns processos no banco de dados, neste caso, podemos utilizar o ClientDataSet ou somente as Querys, veremos os dois casos.

Em nosso projeto vamos simular o seguinte caso:

Precisamos excluir todos os Pedidos de um determinado Cliente.

Este processo nos obriga a atualizar 2 tabelas: Itens de Pedido e Pedido. Temos que excluir primeiro os Itens e depois os Pedidos, e tudo deve ser executados com sucesso, não poderemos excluir os itens e deixar de excluir os Pedidos pelo fato de algum erro ter ocorrido, se isso acontecer, devemos desfazer tudo.

Então iniciaremos uma transação, executaremos duas querys no servidor e ao final confirmamos (commit) se tudo foi aplicado com sucesso ou desfazemos (rollback) se algum erro ocorrer.

Transações – Utilizando SQLQuery

Crie um novo DataModule, ajuste o nome para **dmTransacao** e salve-o como **udmTransacao.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...  
implementation  
  
uses  
    udmPrincipal;  
...
```

Em seguida, adicione os seguintes componentes:



TSQLQuery (dbExpress)

Name: qryExcluiPedido

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
DELETE  
FROM  
    PEDIDO  
WHERE  
    CLI_CODIGO = :CLI_CODIGO
```



TSQLQuery (dbExpress)

Name: qryExcluiItem

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
DELETE FROM  
    PEDITEM  
WHERE  
    EXISTS (SELECT PED_NUMERO FROM PEDIDO WHERE PED_NUMERO =  
        PEDITEM.PED_NUMERO AND CLI_CODIGO = :CLI_CODIGO)
```



DataModule de Transação

Precisamos ajustar o parâmetro **CLI_CODIGO** definido nas 2 Querys, portanto, ajuste-os da seguinte forma:

DataType: ftInteger

ParamType: ptInput

Criaremos um método no DataModule que será responsável em executar todo processo. Declare-o na seção **public** com o nome de **ExecutarProcessoComSQLQuery**.

```
...
private
{ Private declarations }
public
  procedure ExecutaProcessoComSQLQuery(Codigo: Integer);
end;
...
```

Implemente-o da seguinte forma:

```
procedure TdmTransacao.ExecutaProcessoComSQLQuery(Codigo: Integer);
var
  TD: TTransactionDesc; // requer unit Dbxpress
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  dmPrincipal.SQLConnPrincipal.StartTransaction(TD);
  try
    qryExcluiItem.ParamByName('CLI_CODIGO').Value := Codigo;
    qryExcluiItem.ExecSql;
    qryExcluiPedido.ParamByName('CLI_CODIGO').Value := Codigo;
    qryExcluiPedido.ExecSQL;
    dmPrincipal.SQLConnPrincipal.Commit(TD);
  except
    dmPrincipal.SQLConnPrincipal.Rollback(TD);
    raise;
  end;
end;
```

Precisamos declarar a unit **DBxpress** na cláusula **uses** do DataModule para termos disponível o tipo **TTransactionDesc** que utilizamos na declaração da variável.

Entendendo o código

Declaramos uma variável do tipo **TTransactionDesc** que será responsável em descrever a transação que iniciaremos, nesta variável podemos atribuir diversas informações, porém obrigatoriamente apenas precisamos informar:

TransactionID: Id da transação, deverá ser único para transações simultâneas, no caso, sempre utilizaremos o número 1, já que temos apenas uma transação sendo aberta na aplicação.

IsolationLevel: Nível de isolamento onde na maioria dos casos utilizaremos **xilREADCOMMITTED**, que indica para transação ler somente os dados comitados.

Depois de informado os dados da transação, chamamos o método **StartTransaction** no componente de conexão **SQLConnPrincipal** passando como parâmetro nossa variável **TD**. Este é o método responsável por iniciar a transação. A partir deste ponto, tudo que for feito no banco de dados nesta transação, ao final deverá ser confirmado com o método **Commit** ou desfeito com o método **Rollback**. Então executamos as 2 queries e em seguida chamamos o método **Commit**, que como dissemos, confirma as atualizações. No final utilizamos o **except** para que se, caso algum erro ocorra, executamos o método **RollBack** para tudo ser desfeito, e em seguida o método **raise** para levantar o erro.

É importante observarmos a sequência do código e a estrutura do bloco **try/except**, se algum erro ocorrer na segunda query por exemplo, o comando Commit **não** será executado, logo as alterações não serão confirmadas e o código seguirá para o bloco **except** executando o método **Rollback** para desfazê-las.

Criando o Formulário

Crie um novo formulário, ajuste o nome para **frmTransacao** e salve-o como **ufrmTransacao.pas**.

Adicione na cláusula **uses** a unit **udmTransacao**.

```
...  
implementation  
  
uses  
    udmTransacao;  
...
```

Em seguida, monte o formulário da seguinte forma:



Formulário de Transação

Ajuste o nome dos componentes:

Edit: edtCodigo

Button: btnExecutaComSQLQuery

Codifique os eventos do formulário da seguinte forma:

OnCreate:

```
procedure TfrmTransacao.FormCreate(Sender: TObject);  
begin  
    dmTransacao := TdmTransacao.Create(Self);  
end;
```

OnDestroy:

```
procedure TfrmTransacao.FormDestroy(Sender: TObject);  
begin  
    dmTransacao.Free;  
    dmTransacao := nil;  
end;
```

OnClose:

```
procedure TfrmTransacao.FormClose(Sender: TObject;  
    var Action: TCloseAction);  
begin  
    Action := caFree;  
    frmTransacao := nil;  
end;
```

No evento **OnClick** do botão **btnExecutaComSQLQuery** insira o seguinte código:

```
procedure TfrmTransacao.btnExecutaComSQLQueryClick(Sender: TObject);
begin
    dmTransacao.ExecutaProcessoComSQLQuery(StrToInt(edtCodigo.Text));
end;
```

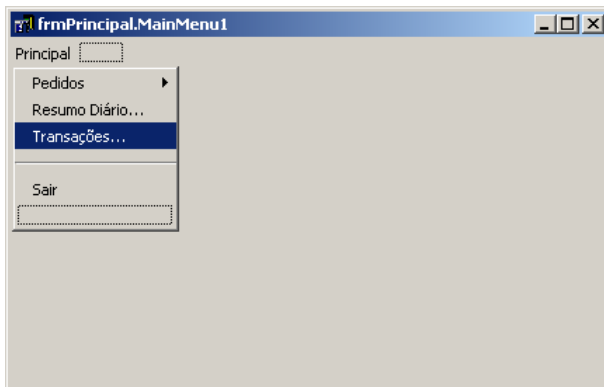
Ajustando o formulário principal

Abra o formulário **frmPrincipal** e adicione na cláusula **uses** a unit **ufrmTransacao**.

```
...
implementation

uses
    ufrmCadPedido,
    ufrmCadTipoPedido,
    ufrmPesqPedido,
    ufrmResumoDiario,
    ufrmTransacao;
...
```

Em seguida, no **MainMenu** crie um item chamado **Transações**.



Incluindo item de menu Transações no formulário principal

Codifique o evento **OnClick** do Item da seguinte forma:

```
procedure TfrmPrincipal.ransaes1Click(Sender: TObject);
begin
    if not Assigned(frmTransacao) then
        frmTransacao := TfrmTransacao.Create(Self);
    frmTransacao.Show;
end;
```

Testando a aplicação

Na tela de transação, após informarmos o código do cliente, podemos clicar no botão **ExecutarComSqlQuery** e veremos que os Pedidos do Cliente serão excluídos. Neste caso dificilmente um erro ocorrerá, mas vamos simular para que possamos ver a exclusão sendo desfeita.

Para simplificar, simularemos o erro no próprio código, gerando uma exceção com o método **raise**.

Abra o DataModule **dmTransacao** e ajuste a procedure **ExecutaProcessoComSQLQuery** da seguinte forma:

```
procedure TdmTransacao.ExecutaProcessoComSQLQuery(Codigo: Integer);
var
  TD: TTransactionDesc; // requer unit Dbxpress
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  dmPrincipal.SQLConnPrincipal.StartTransaction(TD);
  try
    qryExcluiItem.ParamByName('CLI_CODIGO').Value := Codigo;
    qryExcluiItem.ExecSql;

    raise Exception.Create('Erro qualquer...');

    qryExcluiPedido.ParamByName('CLI_CODIGO').Value := Codigo;
    qryExcluiPedido.ExecSQL;
    dmPrincipal.SQLConnPrincipal.Commit(TD);
  except
    dmPrincipal.SQLConnPrincipal.Rollback(TD);
    raise;
  end;
end;
```

Entendendo o código

Acrescentamos apenas uma linha:

```
raise Exception.Create('Erro qualquer...');
```

Na linha que adicionamos o **raise**, simulamos como se a primeira Query tivesse sido executada com sucesso e um erro ocorresse na execução da segunda query (qryPedido), portanto o método **Commit** não será executado, pois o código passará para o bloco **except** executando o **Rollback** e as atualizações serão desfeitas, neste caso a exclusão dos itens.

Testando a aplicação

Abra a tela de Transação e informe um Código de Cliente que contenha Pedidos e Itens, ao clicarmos no botão **ExecutarComSqlQuery**, o processo será executado e podemos depois checar que o Pedido e os Itens não foram excluídos, justamente pelo fato de termos gerado a exceção na transação.

Transações – Utilizando ClientDataSet

Em determinadas situações, pode ser que seja necessário fazer as atualizações no banco com o conjunto ClientDataSet/Provider/DataSet ao invés de utilizar somente Querys executando o SQL diretamente por exemplo.

Utilizando este modelo, pode surgir a seguinte dúvida:

Se o Provider já inicia uma transação automaticamente na atualização dos dados, porque precisaríamos controlar manualmente?

O Provider realmente já faz esse trabalho, porém em nosso caso teríamos 2 conjuntos dos componentes ClientDataSet/Provider/Query, um para excluir Pedidos e outro para Itens, portanto, teríamos 2 ClientDataSet. Neste caso chamaríamos o método ApplyUpdates em ambos, logo teríamos a seguinte ordem de transações iniciadas e encerradas automaticamente:

ApplyUpdates no 1.o ClientDataSet

- Inicia uma transação
- Atualiza o Banco
- Encerra a transação

ApplyUpdates no 2.o ClientDataSet

- Inicia uma transação
- Atualiza o Banco
- Encerra a transação

Perceba que para cada chamada ao método ApplyUpdates temos uma transação sendo iniciada e encerrada, portanto, se algum erro ocorrer na 2.a chamada, será desfeito apenas o que foi feito no 2.o ClientDataSet, mas o que foi feito no 1.o não seria possível, pois a transação já foi encerrada.

O que precisamos fazer nestes casos é iniciarmos a transação manualmente e em seguida executarmos os métodos ApplyUpdates dos ClientDataSets, pois desta forma, o Provider não iniciara a transação internamente, pois antes de iniciar ele verifica se alguma transação já está iniciada, se tiver, ele não fica mais responsável pelo controle das transações.

Neste caso, teríamos a seguinte ordem de execução:

- Inicia transação manualmente
- ApplyUpdates no 1.o ClientDataSet
- ApplyUpdates no 2.o ClientDataSet
- Encerra transação

Perceba que apenas uma transação foi aberta e encerrada manualmente, portanto, havendo algum erro no 2.o ClientDataSet, podemos cancelar a transação e os dados de ambos os ClientDataSets serão desfeitos.

Colocando em prática

No DataModule **dmTransacao** insira os seguintes componentes:

Componentes para exclusão de Pedidos



TClientDataSet (Data Access)

Name: cdsPedido

ProviderName: dspPedido



TDataSetProvider (Data Access)

Name: dspPedido

DataSet: qryPedido



TSQLQuery (dbExpress)

Name: qryPedido

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  PED_NUMERO
FROM
  PEDIDO
WHERE
  CLI_CODIGO = :CLI_CODIGO
```

Componentes para exclusão dos itens:



TClientDataSet (Data Access)

Name: cdsPedidoItem

ProviderName: dspPedidoItem



TDataSetProvider (Data Access)

Name: dspPedidoItem

DataSet: qryPedidoItem



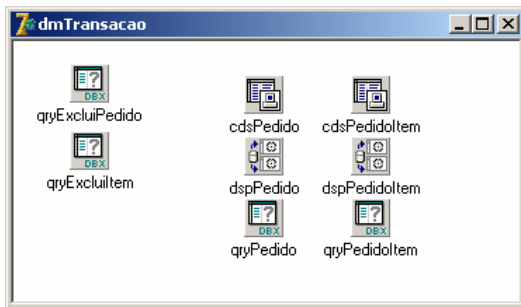
TSQLQuery (dbExpress)

Name: qryPedidoItem

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  PED_NUMERO,
  PROD_CODIGO
FROM
  PEDITEM
WHERE
  EXISTS (SELECT PED_NUMERO FROM PEDIDO WHERE PED_NUMERO =
  PEDITEM.PED_NUMERO AND CLI_CODIGO = :CLI_CODIGO)
```



DataModule de Transação utilizando ClientDataSet

Ajuste o parâmetro **CLI_CODIGO** definido nas 2 queries da seguinte forma:

DataType: ftInteger

ParamType: ptInput

Crie um método chamado **ExecutaProcessoComCDS** na seção **public** do DataModule:

```
private
{ Private declarations }
public
  procedure ExecutaProcessoComSQLQuery(Codigo: Integer);
  procedure ExecutaProcessoComCDS(Codigo: Integer);
end;
```

Implemente-o da seguinte forma:

```
procedure TdmTransacao.ExecutaProcessoComCDS(Codigo: Integer);
var
  TD: TTransactionDesc; // requer unit Dbxpress
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  dmPrincipal.SQLConnPrincipal.StartTransaction(TD);
  try
    if cdsPedidoItem.Active then cdsPedidoItem.Close;
    cdsPedidoItem.FetchParams;
    cdsPedidoItem.Params.ParamByName('CLI_CODIGO').Value := Codigo;
    cdsPedidoItem.Open;
    while not cdsPedidoItem.IsEmpty do
      cdsPedidoItem.Delete;

    if cdsPedido.Active then cdsPedido.Close;
    cdsPedido.FetchParams;
    cdsPedido.Params.ParamByName('CLI_CODIGO').Value := Codigo;
    cdsPedido.Open;
    while not cdsPedido.IsEmpty do
      cdsPedido.Delete;

    if (cdsPedidoItem.ApplyUpdates(0) = 0) and
       (cdsPedido.ApplyUpdates(0) = 0) then
      dmPrincipal.SQLConnPrincipal.Commit(TD)
    else
      dmPrincipal.SQLConnPrincipal.Rollback(TD);
  except
    dmPrincipal.SQLConnPrincipal.Rollback(TD);
    raise;
  end;
end;
```

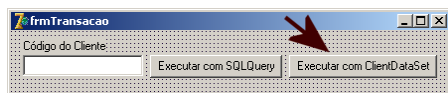

Entendendo o Código

A transação é iniciada e encerrada da mesma forma que fizemos no modelo anterior, o único detalhe que devemos nos atentar é a forma que executamos o método ApplyUpdates nos ClientDataSets para depois sim executar o Commit. Perceba que verificamos se o retorno do método ApplyUpdates de ambos são iguais a zero, pois desta forma temos a certeza de que tudo foi aplicado com sucesso, se algum erro ocorresse, não seria gerada uma exceção, a função retornaria um valor diferente de zero e conseqüentemente nossa comparação seria False, logo executaríamos o método RollBack para desfazer o que foi aplicado.

Poderíamos ter ajustado o código para executar o método CancelUpdates em caso de algum erro para limpar as pendências do ClientDataSet, neste caso não se faz necessário, não teremos essas pendências em uma segunda execução, pois fechamos o ClientDataSet caso o mesmo esteja aberto.

Ajustando o formulário de transação

Abra o formulário **frmTransacao** e insira um botão com o nome de **btnExecutaComCDS**.



Formulário de Transação com ClientDataSet

No evento **OnClick** do botão insira o seguinte código:

```
procedure TfrmTransacao.btnExecutaComCDSClick(Sender: TObject);  
begin  
    dmTransacao.ExecutaProcessoComCDS(StrToInt(edtCodigo.Text));  
end;
```

Testando a aplicação

O mesmo tipo de teste que fizemos utilizando SQLQuery poderia ser feito neste, gerando uma exceção e percebendo que os dados são desfeitos da mesma forma.

Eventos BeforeUpdateRecord e AfterUpdateRecord do Provider

Na utilização de Generators vimos o quanto é importante o evento BeforeUpdateRecord, pois é o momento em que as atualizações estão prestes a serem aplicadas ao banco, portanto, podemos fazer ajustes nos dados e deixar a atualização prosseguir com as novas mudanças.

O evento AfterUpdateRecord também é muito importante, como o próprio nome já diz, é disparado após a atualização do registro, é comum sua utilização para tratamento de dados, por exemplo, podemos ajustar uma determinada tabela no banco depois que determinado registro foi atualizado.

Um diferencial do evento BeforeUpdateRecord é que, por ser disparado antes da atualização do registro, podemos gerar uma exceção para abortar o processo ou melhor, podemos ignorar uma determinada atualização, bastando para isso ligar o parâmetro Applied que recebemos no evento.

Decidir qual evento utilizar dependerá muito do caso, se precisarmos fazer algo antes da atualização, utilizamos o evento BeforeUpdateRecord, se for após, utilizamos o evento AfterUpdateRecord.

É importante sabermos que, quando esses eventos são disparados, uma transação já foi iniciada pelo Provider automaticamente (caso não tenha sido iniciado manualmente), como vimos anteriormente, portanto, se executarmos queries (utilizando a mesma conexão dentro destes eventos) para atualizarmos outras tabelas por exemplo, tudo estará sendo feito na mesma transação, portanto temos a garantia de que tudo será aplicado com sucesso ou tudo será descartado.

Para colocarmos em prática, utilizaremos os eventos para o seguinte objetivo:

Não permitiremos a exclusão física de um Pedido, quando o mesmo for excluído, definiremos apenas uma data da exclusão (campo PED_DATAEXCLUSAO) no registro e anularemos o processo de exclusão, portanto, neste caso utilizaremos o evento BeforeUpdateRecord, pois somente nele que podemos anular uma atualização no banco.

A outra regra é que precisamos definir a data de modificação (campo PED_DATAMODIFICACAO) no registro quando o mesmo tiver sofrido algum ajuste, portanto utilizaremos o evento AfterUpdateRecord, pois neste momento sabemos que o registro já foi modificado com sucesso.

Colocando em prática

Abra o DataModule **dmCadPedido** e adicione os seguintes componentes:



TSQLQuery (dbExpress)

Name: qryAtualizaDataModificacao

SQLConnection: dmPrincipal.SQLConnPrincipal

SQL:

```
UPDATE
  PEDIDO
SET
  PED_DATAMODIFICACAO = :PED_DATAMODIFICACAO
WHERE
  PED_NUMERO = :PED_NUMERO
```



TSQLQuery (dbExpress)

Name: qryAtualizaDataExclusao

SQLConnection: dmPrincipal.SQLConnPrincipal

SQL:

```
UPDATE
  PEDIDO
SET
  PED_DATAEXCLUSAO = :PED_DATAEXCLUSAO
WHERE
  PED_NUMERO = :PED_NUMERO
```

O próximo passo agora será ajustar os parâmetros.

Selecione o componente **qryAtualizaDataModificacao**, clique na propriedade **Params** e ajuste os parâmetros da seguinte forma:

PED_DATAMODIFICACAO

DataType: ftDate

ParamType: ptInput

PED_NUMERO

DataType: ftInteger

ParamType: ptInput

Selecione o componente **qryAtualizaDataExclusao**, clique na propriedade **Params** e ajuste os parâmetros da seguinte forma:

PED_DATAEXCLUSAO

DataType: ftDate

ParamType: ptInput

PED_NUMERO

DataType: ftInteger

ParamType: ptInput

Codificaremos os eventos do Provider **dspPedido** para termos o resultado que planejamos.

Codificando o evento BeforeUpdateRecord:

```
procedure TdmCadPedido.dspPedidoBeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind = ukInsert then
  begin
    if SourceDs = qryPedido then
    begin
      qryGen.Open;
      try
        FNumPedido := qryGen.FieldName('PED_NUMERO_NOVO').AsInteger;
        DeltaDs.FieldName('PED_NUMERO').NewValue := FNumPedido;
      finally
        qryGen.Close;
      end;
    end else
    if (SourceDs = qryPedItem) then
      if cdsPedItem.FieldName('PED_NUMERO').AsInteger < 0 then
        DeltaDs.FieldName('PED_NUMERO').NewValue := FNumPedido;
  end;

  if UpdateKind = ukDelete then
  begin
    if SourceDs = qryPedido then
    begin
      qryAtualizaDataExclusao.ParamByName('PED_NUMERO').Value :=
DeltaDs.FieldName('PED_NUMERO').Value;
      qryAtualizaDataExclusao.ParamByName('PED_DATAEXCLUSAO').Value :=
Date;
      qryAtualizaDataExclusao.ExecSql;
      Applied := True;
    end;
    if SourceDs = qryPedItem then
    begin
      if cdsPedido.IsEmpty then
      begin
        Applied := True;
      end;
    end;
  end;
end;
```

Entendendo o código

O código está extenso, porém acrescentamos apenas o bloco em destaque.

O que fizemos foi, verificamos se a operação que está sendo realizada é uma exclusão, sendo, checamos se refere a Pedido, neste caso, executamos nossa query que atualizará o campo Data de Exclusão do Pedido e logo em seguida **ligamos a variável Applied**, desta forma indicamos ao Provider que a atualização **já foi aplicada**, ou seja, ele **não executará a exclusão do Pedido**.

Quando não se refere a Pedido, checamos se a exclusão é da tabela de Itens, sendo, em seguida checamos se o ClientDataSet de Pedidos está vazio, pois se estiver, sabemos que o Pedido acabou de ser excluído, então como anulamos a exclusão física do Pedido, fazemos o mesmo para os Itens ligando a variável **Applied**.

No caso estar modificando um Pedido e apenas excluir um item, a tabela de Pedidos não estará vazia, portanto neste caso a exclusão do item ocorre normalmente.

Codificando o evento AfterUpdateRecord:

```
procedure TdmCadPedido.dspPedidoAfterUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind);
begin
  if SourceDs = qryPedido then
    if UpdateKind = ukModify then
      begin
        qryAtualizaDataModificacao.ParamByName('PED_NUMERO').Value :=
          DeltaDs.FieldByName('PED_NUMERO').OldValue;
        qryAtualizaDataModificacao.ParamByName('PED_DATAMODIFICACAO').Value
          := Date;
        qryAtualizaDataModificacao.ExecSQL;
      end;
    end;
end;
```

Entendendo o código

Percebemos que o código neste evento é mais simples, apenas verificamos se a operação está sendo realizada no Pedido e se é uma modificação comprando o UpdateKind com ukModify. Sendo verdadeiro, em seguida executamos nossa query que atualizará o campo Data de Modificação do Pedido.

O que precisamos observar é que utilizamos a propriedade **OldValue** do Delta para obter o número do Pedido, isto foi necessário pois não houve mudanças neste campo, portanto a propriedade **Value** estará definida como NULL neste evento.

Testando a aplicação

Quando testarmos a exclusão, veremos que o Pedido realmente será removido da tela de cadastro, porém ele permanecerá fisicamente no banco e com uma data de exclusão definida. Podemos comprovar isso utilizando o IBExpert.

Claro que se pesquisarmos o Pedido pela aplicação, ainda estará disponível, pois não fizemos nenhuma condição para que sejam restringidos os Pedidos excluídos, bastaria adicionar uma condição na pesquisa do tipo: ...AND PED_DATAEXCLUSAO IS NULL.

No caso da modificação, podemos testar fazendo qualquer alteração no Pedido e após gravarmos, verificamos que na tabela a data de modificação foi ajustada, podemos checar pelo IBExpert também.

Trabalhando com Múltiplas Tabelas – Definindo qual será Atualizada

Na maioria dos cadastros, fazemos um SELECT envolvendo apenas uma tabela, ou no máximo fazemos um JOIN para obter informações extras de outras tabelas.

Podemos ter algum caso no qual precisamos fazer um SELECT em mais de uma tabela unindo-as e deixando disponíveis os dados para o usuário fazer as devidas manutenções. Neste caso, quando chamarmos o método ApplyUpdates, o Provider fará a atualização na primeira tabela da cláusula FROM, porém pode não ser exatamente a tabela que desejamos atualizar.

Por este motivo o Provider nos disponibiliza uma forma alterarmos o NOME da tabela que será atualizada, fazemos isso através do seu evento **OnGetTableName**.

Unindo este recurso mais os ProviderFlags, podemos determinar o Nome da tabela e os campos que poderão ser atualizados, desta forma passamos a ter um total controle da atualização.

Em nosso projeto aplicaremos este recurso de forma simples, teremos uma tela exibindo os Itens e os dados do Pedido na mesma tabela fazendo um JOIN entre elas e o usuário poderá fazer as alterações nos campos dos Itens. Na atualização, veremos que o Provider tentará aplicar as mudanças à tabela de Pedidos, pois será a primeira tabela que colocaremos na cláusula FROM, em seguida ajustaremos de forma que atualize a tabela de Itens.

Colocando em prática

Crie um novo DataModule, ajuste o nome para **dmMultiTabelas** e salve-o como **udmMultiTabelas.pas**.

Adicione na cláusula **uses** a unit **udmPrincipal**.

```
...  
implementation  
  
uses  
    udmPrincipal;  
...
```

Em seguida, adicione os seguintes componentes:



TClientDataSet (Data Access)

Name: cdsPedidoltens

ProviderName: dspPedidoltens



TDataSetProvider (Data Access)

Name: dspPedidoltens

DataSet: qryPedidoltens



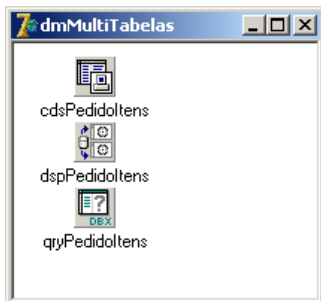
TSQLQuery (dbExpress)

Name: qryPedidoltens

SQLConnection: dmPrincipal.SqlConnPrincipal

SQL:

```
SELECT
  PED.PED_NUMERO,
  PED.PED_DATA,
  PED.CLI_CODIGO,
  PI.PROD_CODIGO,
  PI.PI_DESCRICAO,
  PI.PI_VALUNIT,
  PI.PI_QTDE,
  PI.PI_VALTOTAL
FROM
  PEDIDO PED
  INNER JOIN PEDITEM PI ON PI.PED_NUMERO = PED.PED_NUMERO
```



DataModule MultiplasTabelas

Para visualizarmos o erro que o Provider irá gerar na atualização, implemente o evento **OnReconcileError** do **cdsPedidoltens** da seguinte forma:

```
procedure TdmMultiTabelas.cdsPedidoItensReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  MessageDlg(E.Message, mtError, [mbOk], 0);
end;
```

Em seguida adicione a unit **Dialogs** no DataModule devido ao fato de termos utilizado o método MessageDlg.

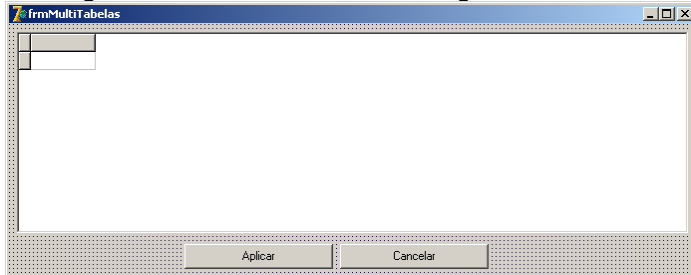
Criando o Formulário

Crie um novo formulário, ajuste o nome para **frmMultiTabelas** e salve-o como **ufrmMultiTabelas.pas**.

Adicione na cláusula **uses** a unit **udmMultiTabelas**.

```
...  
implementation  
  
uses  
    udmMultiTabelas;  
...
```

Em seguida, monte o formulário da seguinte forma:



Formulário MultiTabelas

Ajuste o nome dos componentes na respectiva ordem:

DBGrid: dbgrdPedidoltens

Buttons: btnAplicar e btnCancelar

Adicione um **DataSource**:



DataSource

Name: dtsPedidoltens

DataSet: dmMultiTabelas.cdsPedidoltens

Em seguida ajuste a propriedade **DataSource** do **DBGrid** para **dtsPedidoltens**.

Codifique o evento **OnClick** dos botões:

Aplicar:

```
procedure TfrmMultiTabelas.btnAplicarClick(Sender: TObject);  
begin  
    if dmMultiTabelas.cdsPedidoItens.ApplyUpdates(0) <> 0 then  
        dmMultiTabelas.cdsPedidoItens.CancelUpdates;  
end;
```

Cancelar:

```
procedure TfrmMultiTabelas.btnCancelarClick(Sender: TObject);  
begin  
    dmMultiTabelas.cdsPedidoItens.CancelUpdates;  
end;
```

Codifique os eventos do formulário:

OnCreate:

```
procedure TfrmMultiTabelas.FormCreate(Sender: TObject);
begin
    dmMultiTabelas := TdmMultiTabelas.Create(Self);
    dmMultiTabelas.cdsPedidoItens.Open;
end;
```

OnDestroy:

```
procedure TfrmMultiTabelas.FormDestroy(Sender: TObject);
begin
    dmMultiTabelas.Free;
    dmMultiTabelas := nil;
end;
```

OnClose:

```
procedure TfrmMultiTabelas.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    Action := caFree;
    frmMultiTabelas := nil;
end;
```

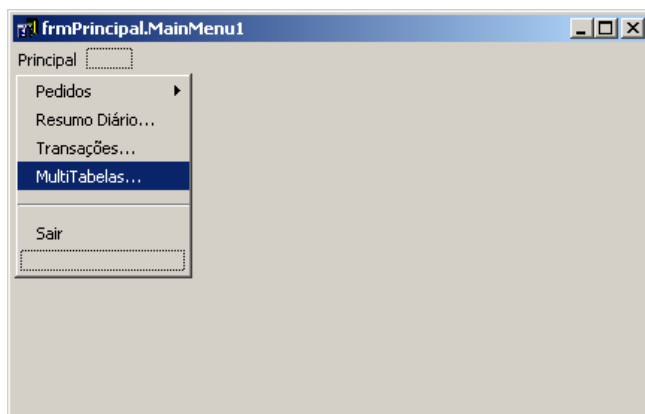
Ajustando o Formulário Principal

Abra o formulário **frmPrincipal** e adicione na cláusula **uses** a unit **ufrmMultiTabelas**.

```
...
implementation

uses
    ufrmCadPedido,
    ufrmCadTipoPedido,
    ufrmPesqPedido,
    ufrmResumoDiario,
    ufrmTransacao,
    ufrmMultiTabelas;
...
```

Em seguida, no **MainMenu** crie um Item chamado **MultiTabelas**



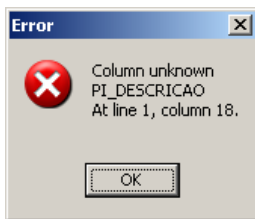
Criando Item MultiTabelas

No evento **OnClick** do Item, insira o seguinte código:

```
procedure TfrmPrincipal.Multitabelas1Click(Sender: TObject);
begin
    if not Assigned(frmMultiTabelas) then
        frmMultiTabelas := TfrmMultiTabelas.Create(Self);
    frmMultiTabelas.Show;
end;
```

Testando a aplicação

Execute a aplicação e entre no formulário que criamos. Teremos disponíveis os dados dos Itens juntamente com os dados do Pedido, portanto, tente alterar por exemplo, a descrição de algum item e em seguida, clicar no botão **Aplicar** para gravar fisicamente no banco de dados. A seguinte mensagem de erro será exibida:



Mensagem de erro ao gravar as alterações no item

Isso ocorre justamente pelo fato de o Provider estar tentando atualizar a tabela PEDIDO, e nesta tabela não temos mesmo o campo PI_DESCRICAO, portanto, precisamos ajustá-lo de forma atualize a tabela PEDITEM.

Definindo o nome da tabela a ser atualizada

Como havíamos comentado, para definirmos o nome da tabela a ser atualizada, utilizamos o evento **OnGetTableName** do **Provider**, que é disparado quando o mesmo precisa saber o nome da tabela para poder montar a query de atualização, portanto, este será o evento que trataremos.

Abra o DataModule **dmMultiTabelas** e no evento **OnGetTableName** do **dspPedidoltens**, adicione o seguinte código:

```
procedure TdmMultiTabelas.dspPedidoItensGetTableName(Sender: TObject;
    DataSet: TDataSet; var TableName: String);
begin
    TableName := 'PEDITEM';
end;
```

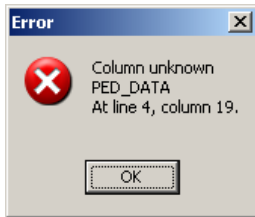
Entendendo o código

Quando este evento for disparado, a variável **TableName** estará com o nome da tabela PEDIDO, portanto, trocamos definindo para PEDITEM para que assim o Provider possa utilizá-la na montagem da query de atualização.

O parâmetro DataSet identifica o respectivo DataSet ao qual o Provider está ligado, no caso o qryPedidoltens.

Testando a aplicação

Ao simularmos o mesmo tipo de teste que fizemos, teremos agora uma nova mensagem de erro:

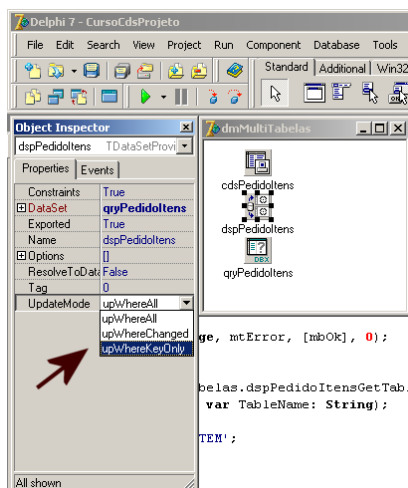


Erro ao gravar os dados do item

Isto está acontecendo devido ao problema que já estudamos anteriormente, a cláusula WHERE que o Provider está montando contém todos os campos, inclusive os campos da tabela de Pedidos, é isso que está gerando o problema, pois pedimos agora para atualizar a tabela PEDITEM e nela não temos os campos do Pedido.

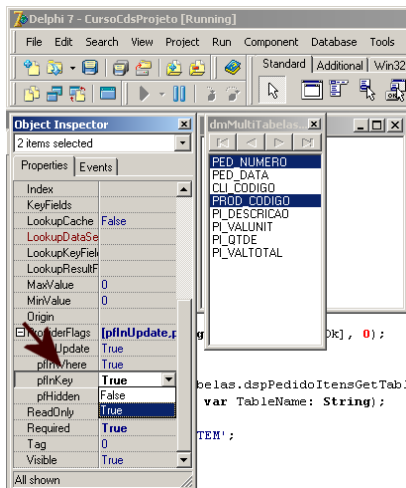
Faremos o ajuste indicando ao Provider para utilizar somente os campos chave na cláusula WHERE, ao invés de desligar campo a campo a opção `pflnWhere` do `ProviderFlags`.

Selecione o componente **dspPedidoltens** e defina a propriedade **UpdateMode** para **upWhereKeyOnly**.



Ajustando UpdateMode do dspPedidoltens para upWhereKeyOnly

Em seguida, temos que definir os campos chaves, portanto, abra o **FieldsEditor** do componente **qryPedidoltens** e adicione todos os campos. Depois selecione os campos **PED_NUMERO** e **PROD_CODIGO** (campos chaves para atualizar a tabela de Itens) e ajuste a propriedade **ProviderFlags** ligando a opção `pflnKey`.



Ligando opção *pfInKey* da propriedade *ProviderFlags* dos campos *PED_NUMERO* e *PROD_CODIGO*

Testando aplicação

Podemos simular o mesmo teste alterando a descrição do item e gravando, a tabela será atualizada com sucesso.

Claro que se tentarmos alterar os campos relativos à tabela de Pedido, uma mensagem de erro seria exibida quando tentássemos gravar no banco, neste caso poderíamos barrar definindo os campos do Pedido como *ReadOnly*, ou se fosse necessário atualizá-los, usaríamos o evento *AfterUpdateRecord* do *Provider* para executar uma query de atualização na tabela de Pedidos.

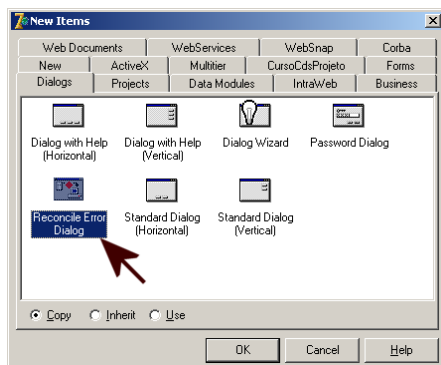
ReconcileError - Tratamento de Erros

O ClientDataSet nos disponibiliza um recurso chamado ReconcileError, que é uma forma de reconciliarmos os erros ocorridos durante as atualizações. Podemos deixar o usuário visualizar e tomar decisão sobre cada erro ocorrido.

O Delphi possui um formulário padrão já montado especificamente para este recurso, onde nele visualizamos os dados do registro que está sendo atualizado, o motivo do erro e optamos pela ação que tomaremos com o respectivo erro.

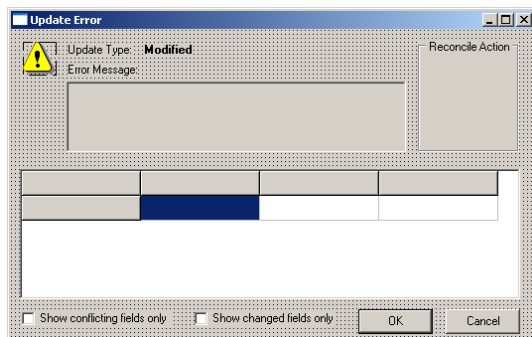
Colocando em prática

Vá ao menu **File-New-Other** e clique na paleta **Dialogs**. Em seguida selecione o item **Reconcile Error Dialog** e clique no botão **OK**.



Adicionando o Reconcile Error Dialog ao projeto

Será adicionado ao projeto o seguinte formulário:



Formulário Reconcile Error Dialog

Salve-o como **ufrmReconcileError.pas**.

É interessante analisar o código do formulário para entendermos como são feitos os controles internos, preenchimento do grid com as informações, etc..

O importante é atentarmos ao método público existente chamado **HandleReconcileError**, ele é o responsável em exibir o formulário para que possamos fazer a reconciliação. Devemos executá-lo no evento **OnReconcileError** do ClientDataSet que faremos o tratamento.

Aplicaremos este recurso no Cadastro de Tipos Pedido, pois será mais fácil para entendermos e visualizarmos o resultado da reconciliação, já que lá podemos aplicar uma atualização em múltiplos registros.

Utilizando o formulário ReconcileError

Abra o DataModule **dmCadTipoPedido** e adicione na cláusula **uses** a unit **ufrmReconcileError**.

```
...  
implementation  
  
uses  
    udmPrincipal,  
    ufrmReconcileError;  
...
```

Em seguida, adicione o seguinte código no evento **OnReconcileError** do **cdsTipoPedido**:

```
procedure TdmCadPedido.cdsPedidoReconcileError(  
    DataSet: TCustomClientDataSet; E: EReconcileError;  
    UpdateKind: TUpdateKind; var Action: TReconcileAction);  
begin  
    Action := HandleReconcileError(DataSet, UpdateKind, E);  
end;
```

Entendendo o código

A linha de código é muito simples, apenas atribuímos à variável **Action** a ação selecionada pelo usuário no formulário, assim o ClientDataSet prosseguirá fazendo o devido tratamento de acordo com esta ação.

Os parâmetros que recebemos neste evento são:

DataSet: É o próprio ClientDataSet.

E: Objeto que contém informações do erro.

UpdateKind: Tipo de operação que ocasionou o erro.

Action: Ação a ser tomada. Podemos definir um dos seguintes valores:

raSkip: Pula o registro deixando-o pendente no Delta.

raAbort: Aborta todo processo de atualização cancelando tudo.

raMerge: Atualiza o registro mesmo que tenha sido modificado.

raCorrect: Ajusta o registro do ClientDataSet com as novas definições feitas nos campos (por exemplo quando ajustamos os valores da coluna Modified Value da tela de reconciliação), para que sejam aplicadas na próxima atualização.

raCancel: Não atualiza o registro, remove do cache de atualizações e volta ao seu estado original.

raRefresh: Desfaz as alterações do registro e busca as informações atuais no banco mantendo-o atualizado.

Testando a aplicação

Para testarmos a reconciliação, temos que ajustar a chamada do `ApplyUpdates`, pois estamos executando com parâmetro Zero, e como nosso objetivo é simular e tratar os erros, então vamos informar ao método que não limite a quantidade de erros, portanto, passaremos o valor -1 como parâmetro.

Abra o formulário **frmCadTipoPedido** e ajuste o evento **OnClick** do botão “**Gravar no Servidor**” da seguinte forma:

```
procedure TfrmCadTipoPedido.btnGravarServidorClick(Sender: TObject);
begin
    dmCadTipoPedido.cdsTipoPedido.ApplyUpdates(-1);
end;
```

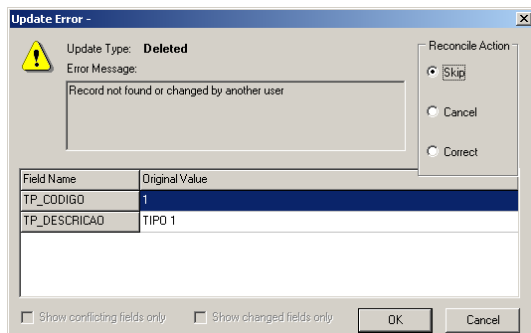
Para verificarmos o efeito, simularemos a típica situação onde o usuário tentará remover um registro já excluído por outro usuário.

Abra duas instâncias da aplicação deixando aberto o Cadastro de Tipos de Pedidos, certifique-se de que temos pelos menos 2 registros para realizarmos os testes.

Na primeira instância, exclua os dois registros e logo em seguida aplique a atualização clicando no botão **Gravar no Servidor**.

Agora volte à segunda instância e perceba que continuamos visualizando os registros excluídos pela primeira, isso acontece porque os dados já estavam em memória e não chamamos o método `Refresh` para atualizá-los. Então exclua os registros na segunda instância e em seguida, faça outra modificação que não gere um erro, por exemplo, inclua um registro qualquer. Feito isto, aplique as mudanças clicando no botão **Gravar no Servidor** e logo aparecerá a tela de Reconciliação.

O mais interessante é observarmos que ela será exibida duas vezes, pois dois erros foram gerados, já que tentamos excluir dois registros inexistentes.



Erro na exclusão do primeiro registro

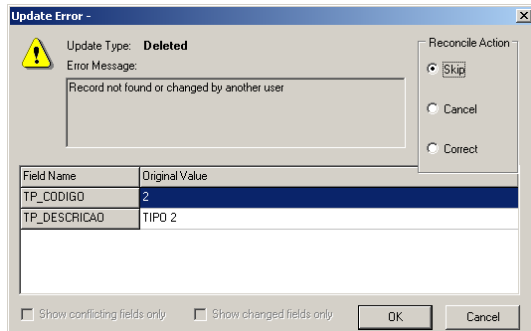
Perceba que temos 3 ações disponíveis que comentamos no evento `OnReconcileError`.

Clicando em **Skip** pulamos a atualização do registro, porém o mesmo ficará em cache e será aplicado na próxima atualização.

Clicando em **Cancel** o registro volta ao seu estado original e é removido da cache de atualização.

Clicando em **Correct**, o ClientDataSet ajustará o registro com as novas mudanças feitas nos campos.

Independente da opção, após confirmarmos, novamente a tela será exibida, agora com o erro da exclusão do segundo registro.



Erro na exclusão do segundo registro

Após confirmarmos teremos o ClientDataSet ajustado de acordo com as opções selecionadas, e a inclusão do registro que fizemos, foi aplicada com sucesso ao banco de dados.

Neste exemplo, para obtermos um resultado satisfatório, poderíamos optar pela ação **Cancel** e após a chamada do método ApplyUpdates, poderíamos chamar o método Refresh. Desta forma eliminaríamos do cache as atualizações que não foram aplicadas e teríamos o ClientDataSet atualizado com as novas modificações feitas no banco de dados.

Monitorando mensagens do Banco de Dados

Monitorar mensagens entre nossa aplicação e o banco de dados é uma tarefa de extrema importância quando precisamos saber as instruções SQL que estão sendo executadas no servidor, pois assim podemos resolver os famosos problemas de lentidão em consultas, relatórios, atualizações, etc.

Utilizando BDE, podemos fazer este monitoramento com o uso do **SQL Monitor**, que é um utilitário que acompanha o Delphi. Já com **dbExpress**, a forma de monitoramento não é feita através de um utilitário e sim de um componente específico para isso, o **TSQLMonitor**, que intercepta as mensagens que ocorrem entre o componente **SqlConnection** (ao qual está ligado) e o Banco de Dados.

Este componente nos permite acessar as mensagens monitoradas através da sua propriedade **TraceList**, além disto temos o recurso de gravá-las em arquivo automaticamente a cada mensagem interceptada, isto é feito ligando sua propriedade **AutoSave** e especificando o caminho do arquivo na propriedade **FileName**.

Colocando em prática

Abra o DataModule **dmPrincipal** e adicione o componente **TSQLMonitor** ajustando as seguintes propriedades:



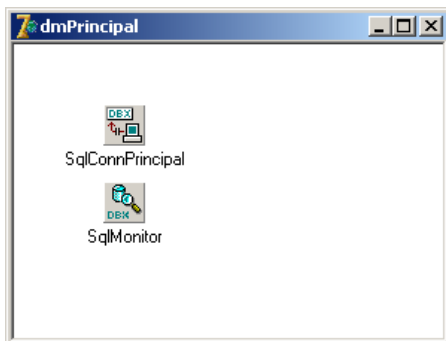
TSQLMonitor (dbExpress)

Name: SqlMonitor

AutoSave: True

FileName: c:\sqlmonitor.txt

SqlConnection: SqlConnPrincipal



DataModule Principal com SQLMonitor

Propriedades utilizadas

AutoSave

Por padrão, as mensagens monitoradas são adicionadas à propriedade **TraceList**, porém, ligando esta propriedade, determinamos também que elas deverão ser gravadas automaticamente em arquivo, definido na propriedade **FileName**.

FileName

Path do arquivo onde as mensagens monitoradas serão gravadas.

SqlConnection

Componente **SqlConnection** a ser monitorado.

Para que a monitoração de mensagens seja ativada, devemos ligar a propriedade **Active** do componente, portanto faremos isso no evento **OnCreate** do DataModule:

```
procedure TdmPrincipal.DataModuleCreate(Sender: TObject);
begin
    SqlMonitor.Active := True;
end;
```

Vale lembrar que não é recomendado deixarmos o monitoramento ativo sempre na aplicação, fazemos isso somente quando necessário, assim evitamos consumos de recursos desnecessários.

Testando a aplicação

Para checarmos as mensagens que estão sendo monitoradas e gravadas, podemos por exemplo, abrir o Cadastro de Pedidos, pois o Provider executa instruções SQL no banco para extração dos dados, portanto, após sua abertura, checamos o arquivo *c:\sqlmonitor.txt* e conferimos as mensagens monitoradas.

Vale lembrar que as mensagens monitoradas poderiam também ser visualizadas através da propriedade **TraceList** do componente **SQLMonitor**.

Distribuindo a aplicação

A distribuição da aplicação é feita de forma bem simples, não precisamos de todos aqueles disquetes que precisávamos antes para instalação do BDE.

Em nosso caso precisamos apenas da DLL **dbexpint.dll** (C:\Arquivos de programas\Borland\Delphi7\Bin\dbexpint.dll), ela é necessária para utilização da DBExpress com Interbase/Firebird. Além desta, precisamos também da DLL **midas.dll** (localizada no diretório de sistema do Windows) para utilização do ClientDataSet.

Para rodar a aplicação, basta estar com essas dlls no mesmo diretório do executável ou no diretório do Windows. Em alguns casos, ao executar o aplicativo uma mensagem de erro é exibida dizendo que não foi possível carregar a Midas.dll. Para resolver este problema, basta registrá-la, para isso, copie-a para o diretório do Windows e execute a seguinte linha de comando:

regsvr32 midas.dll

Uma alternativa para não utilizar as DLL's seria incluir as respectivas units no projeto: **DbExpint** e **MidasLib**.