

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi .NET

Usando todo o poder do TDataSetProvider

Usufrua de todos os recursos do DataSetProvider em suas aplicações



Adriano Santos

(falecom@adrianosantos.pro.br)

é desenvolvedor Delphi desde 1998. Professor e programador PHP. Bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Técnico, Colunista e Membro da Comissão Editorial da revista ClubeDelphi e WebMobile. Mantém o blog Delphi to Delphi (www.delphi-todelphi.blogspot.com) com dicas, informações e tudo sobre desenvolvimento Delphi.

Sem dúvida nenhuma, um dos componentes mais poderosos da VCL é o *DataSetProvider*, que, além de prover dados a toda a aplicação, é capaz de auxiliar em uma série de funcionalidades no sistema.

Veremos o que há de mais interessante nesse componente e dicas avançadas para usufruir ao máximo de suas propriedades e métodos. Em conjunto com os componentes *DBExpress* o *DataSetProvider* é capaz de realizar inúmeras tarefas. Veremos nesse artigo:

- Introdução ao *DataSetProvider*;
- Como usar o *UpdateMode* e *ProviderFlags*;
- Configurando dinamicamente o *UpdateMode* e *ProviderFlags*;
- Uso de *Constraints*;

Criaremos diversos exemplos para entendermos corretamente cada funcionalidade do *DataSetProvider*;

Entendendo o DataSetProvider

Basicamente o *DataSetProvider* é responsável por enviar e receber os *Data Packets* da aplicação cliente para o servidor de dados. Os *Data Packets* são os pacotes de dados trafegados na rede em uma aplicação duas camadas (“two-tier” ou “client/server”) ou “n” camadas (“n-tier”). Toda e qualquer requisição feita pela aplicação cliente é enviada ao *DataSetProvider* que por sua vez se encarrega de solicitar os dados ao servidor de aplicação. O *result set*, ou seja, o resultado da requisição é empacotado (“Data Packets”) e enviado de volta à aplicação cliente. Qualquer exceção levantada, seja na requisição ou no recebimento dos dados, é retornada a aplicação cliente.

Pode-se dizer que a utilização mais comum do *DataSetProvider* é feita em conjunto dos componentes da paleta *DBExpress* acrescido do componente *ClientDataSet*. Um exemplo de uso pode ser visto na **Figura 1**.

O que pouca gente sabe é que o *DataSetProvider* pode se tornar mais do que um simples componente de conexão com o banco de dados por ser farto de propriedades e eventos.

Constraints e DataSetProvider

Como sabemos, as *constraints* de uma aplicação podem ser inseridas de diversas formas em uma aplicação. Em aplicações *two-tier*, por exemplo, temos apenas dois lugares onde elas podem ser incluídas: no lado *servidor* ou na *aplicação cliente*. No servidor devemos inseri-las diretamente no SGBD através de *Stored Procedures*, *domains*, *rules*, *triggers* etc. É uma excelente idéia, pois centralizamos nossas regras de negócios em um único lugar. Porém, corremos o risco de ficarmos presos ao banco de dados que estamos trabalhando por conta da linguagem empregada no SGBD. Em outras palavras quando programamos diretamente no banco necessitamos usar a linguagem de programação SQL, também chamada ANSI, para criar nossas próprias instruções. Isso pode se tornar uma dor de cabeça em uma eventual mudança de banco de dados ou mesmo se o produto vier a ser comercializado com BD's diferentes. Imagine um software que precisa ser capaz de se conectar aos bancos Interbase/Firebird, SQL Server 2005 Express e Oracle. De início já teremos algumas complicações entre Interbase e Firebird dependendo da versão de ambos, haja vista que muitas modificações no BD foram feitas ao longo dos anos. Por isso, da mesma forma que o esquema é interessante, também pode tornar-se um trauma.

Por outro lado é possível incluir *constraints* diretamente na aplicação cliente, porém temos uma enorme desvantagem: a *decentralização* de nossas regras de negócios. Elas precisam ser refeitas para cada nova aplicação e espalhadas por todo o código fonte ou em *Units*, componentes ou *dll's*. Essa prática é altamente perigosa, pois podem ocorrer momentos em que uma regra de negócio pode não ser alterada por esquecimento dos membros da equipe

de desenvolvimento, o que acarretaria em inconsistência dos dados.

Em aplicações *n-tier* (multi-camadas) podemos recorrer ao poderoso DataSnap e programar nossas regras de negócio diretamente no servidor de aplicação, tornando a aplicação mais consistente e inteligente. Essas regras residem no código fonte do servidor de aplicações, e ficam centralizadas.

Resumindo, uma das melhores alternativas certamente é fazer uso desta última opção, ou seja, utilizar um servidor de aplicação que fará o intercâmbio entre *bando de dados* (SGBD) e *aplicação cliente*, pois além de propiciar maior controle sobre as regras de negócios, ainda podemos disponibilizar a aplicação para acesso remoto através da internet.

Propriedades do TField

Algumas propriedades do *TField* são automaticamente passadas da aplicação servidora para o cliente e a maioria dos valores são determinados em tempo de projeto baseando-se na estrutura da tabela no SGBD. Podemos usufruir de alguns recursos somente disponíveis nas propriedades do *TField*, seja em aplicações *single tier*, *two-tier* ou *n-tier* e que não estão disponíveis no DataSnap. Em suma, é possível criarmos pequenas *constraints* diretamente na aplicação cliente e que podem ser facilmente configuradas.

Isso é bastante interessante, visto que reduz drasticamente a quantidade desnecessária de código e ainda permite que sejam personalizadas as mensagens de exceção geradas. Essas propriedades são:

- **Read Only:** Como o próprio nome indica, podemos "setar" um campo como somente leitura;

- **Required:** Essa propriedade estando configurada como True, faz com que a aplicação obrigue o usuário final a digitar um valor nela. Grosseiramente falando, campos Not Null no banco de dados são fortes candidatos a serem campos Required, tanto é, que quando adicionamos um campo Not Null aos Field's Editor's do DataSet, este já vem como True em sua propriedade;

- **DefaultExpression:** Valor padrão para inserção no banco de dados. Aqui podemos configurar um valor que será gravado no banco caso esteja vazio;

- **CustomConstraint:** É exatamente nessa propriedade que definimos nossa constraint. Digitamos a expressão e posteriormente o sistema fará a validação. Ex.: $X > 100$ and $X < 200$;

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra como trabalhar com constraints e ClientDataSet.

www.devmedia.com.br/artigos/viewcomp.asp?comp=567&hl=

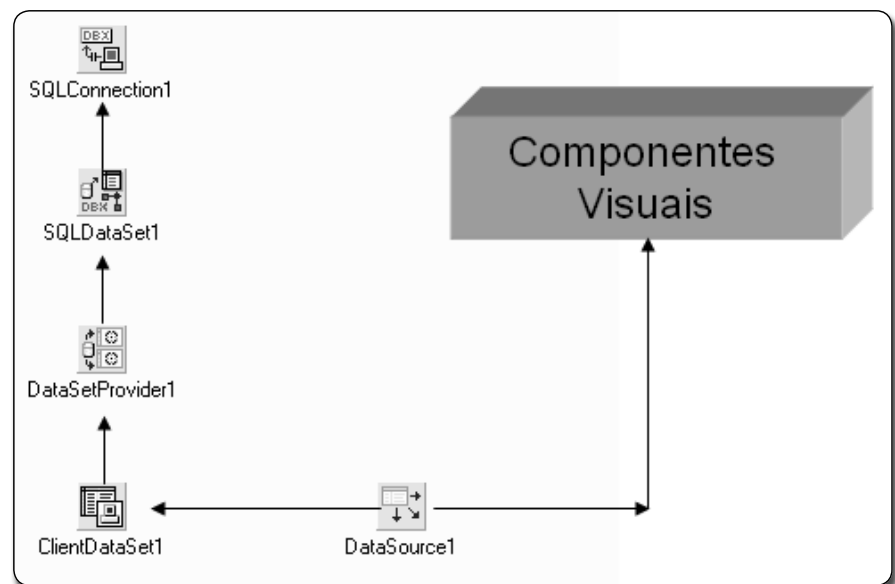


Figura 1. Exemplo de uso do componente DataSetProvider

• **ConstraintErrorMessage:** Por último a mensagem de erro que será exibida ao usuário final da aplicação;

Devemos nos lembrar que isso tudo só é possível com campos persistentes, ou seja, com tipificação de campos no projeto. Esse recurso é utilizado quando adicionamos os campos da tabela no *Field's Editor* do *TDataSet*, do contrário não será possível.

Outras duas informações são importantíssimas. A primeira é que as mensagens de erro são resultantes da violação das regras impostas nos *TField's* do *TDataSet*, e conseqüentemente gerarão exceções. Se outras regras também forem definidas no servidor de dados, ou seja, diretamente no SGBD, estas também levantarão exceções, porém somente uma das duas mensagens será exibida. A segunda informação diz respeito à alteração de regras no SGBD. Algumas das regras de negócios impostas diretamente no BD precisam ser replicadas para a aplicação, como é o caso de campos requeridos ("Require's"). Caso modifique de *True* para *False* ou vice-versa em um determinado campo, este precisa ser removido ou alterado em tempo de projeto. Vejamos um exemplo:

Abra o Delphi e crie uma nova aplicação usando *File\New>Application* e salvando-a como "Properties.dpr" e o formulário principal como "Principal.pas". Insira no formulário um componente *SQLConnection* e um *TSQLDataSet*, ambos da paleta

dbExpress. Dê um clique duplo no *SQL-Connection* e crie uma nova conexão com o banco de dados *Employee.fdb* presente no diretório de instalação do Firebird, normalmente em *C:\Arquivos de Programas\Firebird\Firebird_Versão\examples\emp-build\Employee.fdb*. Conecte o *SQLDataSet* ao *SQLConnection* usando a propriedade *SQLConnection* e em seguida digite a instrução SQL a seguir para selecionar os dados da tabela *SALES* do banco.

```
SELECT * FROM SALES
```

Arraste agora um *DataSetProvider* e um *ClientDataSet* da paleta *Data Access*. Ligue o *DataSetProvider* ao *SQLDataSet* pela propriedade *DataSet* e o *ClientDataSet* ao *Provider* usando *ProviderName*. Abra seu gerenciador de banco de dados preferido e visualize a estrutura da tabela *SALES* (**Figura 2**).

Note que os campos *PO_NUMBER*, *CUST_NO*, *ORDER_STATUS*, *ORDER_DATE*, *QTY_ORDERED*, *TOTAL_VALUE*, *DISCOUNT* e *ITEM_TYPE*, estão marcados como *Not Null*, ou seja, não permitem que sejam gravados sem nenhum valor. Experimente adicionar todos os campos da tabela *SALES* ao *Field's Editor* do *ClientDataSet* usando o menu de contexto e a opção *Add all fields*. Em seguida clique em cada campo e observe a propriedade *Required*. Perceba que cada campo *Not Null* teve sua propriedade *Required* modificada para *True*, isso significa que o

sistema obrigará o usuário a inserir um valor nesses campos (**Figura 3**).

Propriedade adicionais do TField usadas pelo DataSetProvider

Agora vejamos algumas das propriedades mais importantes em um *TField* e que são utilizadas fortemente pelo *DataSetProvider*. A propriedade *Options* do DSP ("DataSetProvider") suporta a passagem de propriedades adicionais do *TField's*. Quando configuramos como *True* a propriedade *polncFieldProps*, podemos passar propriedades do *TField* para o DSP. Essas propriedades são:

- *Alignment*;
- *Currency*;
- *DisplayFormat*;
- *DisplayLabel*;
- *DisplayWidth*;
- *EditFormat*;
- *EditMask*;
- *MaxValue*;
- *MinValue*;
- *Visible*;

O que poucos sabem, é que podemos configurar outras propriedades do DSP capazes de nos auxiliar em diversas tarefas. Ainda na propriedade *Options* vejamos algumas delas:

- *poReadOnly*: Desabilita qualquer alteração no DataSet ligado a ela;
- *poDisableInserts*: Desabilita a inserção de dados na tabela;
- *poDisableEdits*: Desativa a edição dos registros;
- *poDisableDeletes*: Não permite a exclusão de registros;

Retorne ao projeto criado anteriormente e desenhe uma tela semelhante à **Figura 4**. Selecione o *DataSerProvider* e configure a propriedade *Options>poReadOnly* para *True*, ative o *ClientDataSet* e execute a aplicação. Perceba que o *DB-Navigator* permite apenas a navegação dos registros (**Figura 5**) e não mais as demais opções.

Seguindo a mesma linha de raciocínio, agora modifique a propriedade *Options>DisableEdits*, salve o projeto e execute. Na seqüência, experimente clicar no botão de alteração e note o erro (**Figura 6**).




Fields	Constraints	Indices	Dependencies	Triggers	Data	Master/Detail View	
AGED COMPUTED BY ((ship_date - order_date))							
#	PK	FK	UNQ	Field Name	Field Type	Size	Not Null
1				PO_NUMBER	CHAR	8	<input checked="" type="checkbox"/>
2				CUST_NO	INTEGER		<input checked="" type="checkbox"/>
3				SALES_REP	SMALLINT		<input type="checkbox"/>
4				ORDER_STATUS	VARCHAR	7	<input checked="" type="checkbox"/>
5				ORDER_DATE	TIMESTAMP		<input checked="" type="checkbox"/>
6				SHIP_DATE	TIMESTAMP		<input type="checkbox"/>
7				DATE_NEEDED	TIMESTAMP		<input type="checkbox"/>
8				PAID	CHAR	1	<input type="checkbox"/>
9				QTY_ORDERED	INTEGER		<input checked="" type="checkbox"/>
10				TOTAL_VALUE	DECIMAL	9	<input checked="" type="checkbox"/>
11				DISCOUNT	FLOAT		<input checked="" type="checkbox"/>
12				ITEM_TYPE	VARCHAR	12	<input checked="" type="checkbox"/>
13				AGED	NUMERIC	18	<input type="checkbox"/>

Figura 2. Estrutura da tabela *SALES* do banco *Employee.fdb*

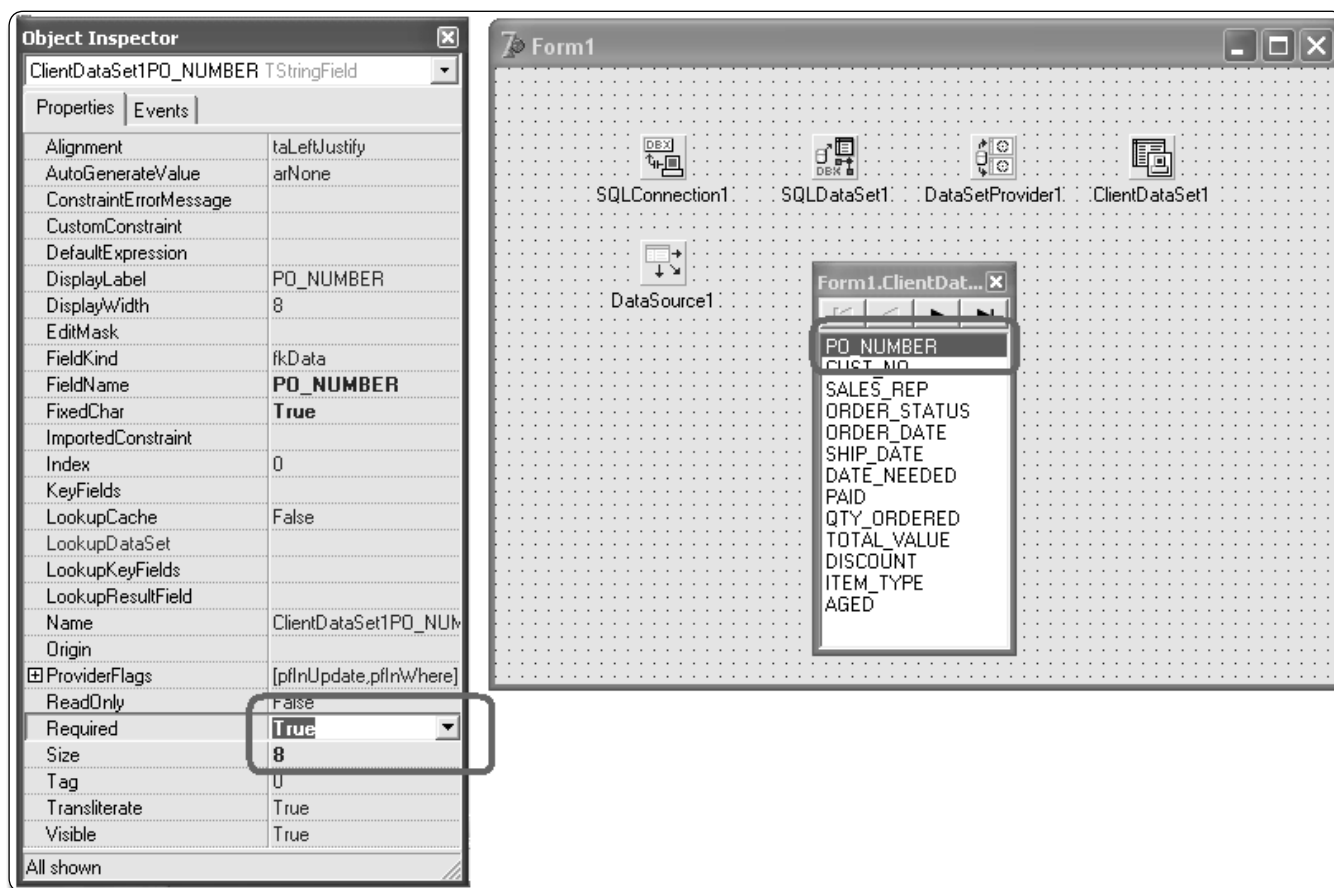


Figura 3. Configuração do TField adicionado automaticamente

Isso acontece, porque informamos ao *DSP* que não é permitida a alteração de registros por parte da aplicação cliente, logo a mensagem "ClientDataSet1: Modifications are not allowed.", traduzindo, "ClientDataSet1: Modificações não são permitidas."

Usando os eventos do DSP

Podemos utilizar três eventos para validar *constraints* em uma aplicação usando o *DataSetProvider*. São eles:

- *BeforeUpdateRecord*: Acontece antes que os dados sejam atualizados na aplicação remota;
- *OnUpdateData*: Acontece ao aplicar os dados recebidos pela aplicação cliente, no servidor de dados;
- *OnUpdateError*: Quando ocorrem erros ao tentar efetuar o *Update*.

Usamos o evento *BeforeUpdateRecord* quando necessitamos fazer a validação individual dos registros enviados ao servidor. Também podemos modificar dados recebidos da aplicação cliente. A

assinatura completa do evento podemos ver a seguir, juntamente com a descrição de cada parâmetro:

```
BeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS:
  TClientDataSet;
  UpdateKind: TUpdateKind;
  var Applied: Boolean);
```

- *Sender*: *DataSetProvider* que disparou o evento;
- *SourceDS*: Grosseiramente falando é a fonte de dados, os dados em si;
- *DeltaDS*: Pacote de dados enviado pela aplicação cliente;
- *UpdateKind*: Tipo de update, atualização, que será feita. Os valores possíveis são: *ukInsert*, *ukModify* e *ukDelete*;
- *Applied*: Indica se as modificações serão aplicadas.

Usando o evento BeforeUpdateRecord

Vejamos um exemplo prático. No evento *BeforeUpdateRecord* digite o código da **Listagem 1**. Nesse caso estamos verificando primeiramente se o tipo de atua-

lização não é uma instrução para *deletar* o registro, pois caso seja não há necessidade em se validar um registro que será apagado. Em seguida testamos se o valor recebido é diferente de *Null* ("VarIsNull") e diferente de *vazio* ("VarIsEmpty"). Por fim, verificamos se o valor do campo *ORDER_DATE* não é superior ao *SHIP_DATE*. Havendo quaisquer divergência uma exceção é levantada.

Usando o evento OnUpdateData

O evento *OnUpdateData* ocorre uma vez ao iniciar a aplicação dos dados no servidor, ou seja, no recebimento dos registros Delta do *ClientDataSet*. É nesse momento que interceptamos as modificações e aceitamos ou até modificamos os dados se necessário. Aqui também podemos enviar mensagens à aplicação cliente caso seja necessário.

Veja a assinatura do evento logo em seguida:

```
OnUpdateData(Sender: TObject; DataSet:
  TClientDataSet);
```

- *Sender*: DSP que disparou o evento;
- *DataSet*: Pacote de dados, *Data Packet*;

No caso do evento anterior, *BeforeUpdateData*, não temos acesso total aos dados que estão vindo, e sim a um registro em particular. Já no caso do *OnUpdateData*, recebemos o *DataSet*, que contém todos os dados alterados e não alterados. Sendo assim, podemos ler individualmente o status de cada registro da tabela através do *Delta*. Insira mais um componente *ClientDataSet*("Delta") e

um *DataSource*("DataSource1") ao sistema. Desconecte o *ClientDataSet* do *provider* limpando a propriedade *ProviderName*. Conecte o *DataSource1* ao *Delta*. Insira um *Button* e um *DBGrid*. Ligue o *DBGrid* ao *dsDelta* e digite o código da **Listagem 2** no evento *OnClick* do *Button* (**Figura 7**).

O que estamos fazendo é muito simples. Apenas atribuímos à propriedade *Data* do *Delta* o conteúdo da propriedade *Delta* do *ClientDataSet1*, que contém os dados originais acrescido das alterações

efetuadas. Execute o programa, efetue algumas alterações na tabela e em seguida clique no botão *Delta*. Perceba que o 2º *DBGrid* mostra apenas alguns registros. Eles fazem parte do *Data Packet*, ou seja, o pacote de dados que será enviado ao DSP para montagem das instruções de inclusão, alteração e exclusão dos registros na base de dados. São assim que chegam os dados ao DSP (**Figura 8**).

O primeiro registro com o código *V91E0210*, foi marcado para exclusão. Já

Figura 4. Exemplo de tela

Listagem 1. Código do evento *BeforeUpdateRecord*

```
procedure TForm1.DataSetProvider1BeforeUpdateRecord(
  Sender: TObject; SourceDS: TDataSet; DeltaDS:
  TClientDataSet; UpdateKind: TUpdateKind;
  var Applied: Boolean);
begin
  if UpdateKind <> ukDelete then
    if ((VarIsNull(DeltaDS.FieldByName('ORDER_DATE'))
      .NewValue) = False) and
      (VarIsEmpty(DeltaDS.FieldByName('ORDER_DATE'))
      .NewValue) = False) then
      if DeltaDS.FieldByName('ORDER_DATE').NewValue >
        DeltaDS.FieldByName('SHIP_DATE').OldValue then
        raise Exception.Create('Data do pedido não pode ser
          superior a data de compra.');
```

end;



Figura 5. DBNavigator apenas com funções de navegação

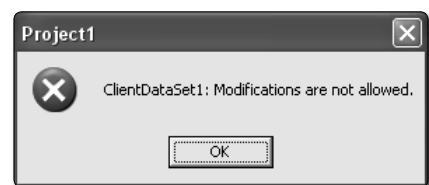


Figura 6. Exceção gerada pelo DSP

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra como funciona a arquitetura do Data e Delta do ClientDataSet.

www.devmedia.com.br/articles/viewcomp.asp?comp=5717&hl=



Nota do DevMan

Se você quiser mostrar mensagens personalizadas de erros em seu DSP, basta fazer uso dos eventos *OnEditError*, *OnPostError* e *OnDeleteError* presentes no *ClientDataSet*. Para isso basta acessar o evento que deseja modificar e inserir a mensagem desejada. Ex:

```
procedure TForm1.ClientDataSet1EditError(
  DataSet: TDataSet;
  E: EDatabaseError; var Action:
  TDataAction);
begin
  MessageDlg('Não são permitidas
    alterações nessa tabela.',
    mtInformation, [mbOk], 0);
  Action := daAbort;
end;
```

Note que estamos alterando o parâmetro *Action* para *daAbort*, dessa forma a mensagem original do *ClientDataSet* não será exibida, mostrando apenas nossa caixa de diálogo.

o segundo registro, V92J1003, teve o campo *CUST_NO* alterado de 1010 para 300, por isso aparece na visualização apenas o campo *CUST_NO* alterado. Por fim os dois últimos registros foram incluídos na tabela. A partir do momento que o método *ApplyUpdates* do *ClientDataSet* for chamado, o *DSP* fará a montagem das instruções SQL e as enviará ao servidor de dados, que por sua vez executará todo o processo de atualização.

Usando o evento *OnUpdateError*

Nas seções anteriores vimos que é possível incluir mensagens personalizadas na aplicação para lidar com erros do sistema. Aqui veremos como trabalhar com o evento *OnUpdateError*. Nele podemos interceptar o erro, processá-lo, corrigi-lo e até mesmo modificá-lo para que o usuário final tenha mais confiança no aplicativo e entenda melhor tudo que está acontecendo. A seguir encontramos uma breve descrição de cada parâmetro do evento, bem como sua assinatura:

```
OnUpdateError(Sender: TObject;
  DataSet: TClientDataSet;
  E: EUpdateError;
  UpdateKind: TUpdateKind;
  var Response: TResolverResponse);
```

- *Sender*: *DataSerProvider* que disparou o erro;
- *DataSet*: *DataSet* temporário para acessar o erro;
- *E*: Objeto de exceção;
- *UpdateKind*: Tipo de *update*;
- *Response*: Ação de resposta ao erro;

Assim como os eventos que já vimos, aqui nós podemos usar os valores *NewValue*("Novo Valor") e *OldValue*("Valor Anterior") do *TField*. Também é possível utilizar o valor *CurrentValue* ("Valor atual"), que indica o valor atual do banco de dados. Com isso podemos visualizar o valor original, valor armazenado atualmente e o valor que deverá ser aplicado.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra como a arquitetura do Data Packets no *ClientDataSet*.

www.devmedia.com.br/articles/viewcomp.asp?comp=5932&hl=

Listagem 2. Código do botão Delta

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  try
    Delta.Close;
    Delta.Data := ClientDataSet1.Delta;
    Delta.Open;
  except
    MessageDlg('Sem registros no Delta',
      mtWarning, [mbOK], 0);
  end;
end;
```

Figura 7. Exemplo de tela com o Delta

Form1

PO_NUMBER: aaaaaaaa CUST_NO: 1 SALES_REP: 1 ORDER_STATUS: aa

ORDER_DATE: 01/01/2007

SHIP_DATE: 01/01/2007

DATE_NEEDED: 01/01/2007

PAID: y QTY_ORDERED: 1 TOTAL_VALUE: 1 DISCOUNT: 12

ITEM_TYPE: software AGED: 1

PAID	QTY_ORDERED	TOTAL_VALUE	DISCOUNT	ITEM_TYPE	AGED
n	3	16000	10002980232	hardware	
n	1	490,69	0	software	32
n	5	2693	0	hardware	
y	1	100,02	0	software	1
y	1	1	12	software	1

PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE
V91E0210	1004	11	shipped	04/03/1991
V92J1003	1010	61	shipped	26/07/1992
	300			
V9ASDF31	200	12	shipped	01/01/2007
aaaaaaaa	1	1	aa	01/01/2007

Delta

Figura 8. Delta do ClientDataSet1 sendo visualizado

Warning

Status do pedido deve ser novo, aberto, vendido ou aguardando.

OK

Figura 9. Mensagem personalizada no evento OnUpdateError

O parâmetro de exceção, *E*, possui uma propriedade chamada *OriginalException*. Ele permite que você obtenha o valor original da exceção gerada pelo erro. Com BDE podemos usar a classe *EDBEngineError* para obter o código de erro.

É importante lembrar que dependendo da quantidade de erros mencionada na chamada ao *ApplyUpdates*, as exceções não serão propagadas, isso porque podemos limitar o número de erros tolerados pelo *ClientDataSet*.

Experimente digitar o código da **Lista-gem 2** no evento *OnUpdateError* do *DSP*, execute a aplicação e digite um valor qualquer no campo Status. Depois salve e clique em *Apply*. Repare na mensagem que é exibida na **Figura 9**.

Configurando UpdateMode e ProviderFlags

Trabalhar com *DSP* não é tão difícil quanto se pensa, mas muitos se atrapalham com suas configurações. Uma das coisas que mais sou indagado, diz respeito às propriedades *UpdateMode* do *DataSetProvider* e *ProviderFlags* do *ClientDataSet*. Não há segredos mirabolantes nessas duas propriedades, vejamos o que cada uma significa e como funciona.

Em suma temos apenas três estados do *UpdateMode* que são:

- *upWhereAll*: Utiliza todos os campos na cláusula *Where* para encontrar o registro;
- *upWhereChanged*: Utiliza apenas os campos alterados na cláusula *Where* para encontrar os registros;
- *upWhereKeyOnly*: Utiliza apenas os campos chave para procurar os registros.

Bem, pra ser mais objetivo toda vez que chamamos o método *ApplyUpdates*, o *ClientDataSet* envia os *Data Packet's* para o *DSP* que se encarrega de varrê-lo e montar as instruções SQL que serão enviadas ao servidor e posteriormente aplicadas ao banco. Quando passamos para o *DSP* a opção *upWhereAll*, ele montará uma instrução utilizando todos os campos da tabela em sua cláusula *Where* para que o registro seja

atualizado. Supondo que nossa tabela possui os campos *ID*, *NOME*, *ENDERECO*, *CIDADE*, *ESTADO* e *TELEFONE*, sendo que apenas o campo *ID* é chave, imagine a instrução montada pelo *DSP* quando apenas o campo *TEFONE* fosse alteração. Veja:

```
UPDATE CLIENTES SET TELEFONE='555-5525'
WHERE ID=ID, NOME=NOME, ENDereco=ENDERECO,
CIDADE=CIDADE, TELEFONE=TELEFONE
```

Perceba que a cláusula *Where* possui todos os campos da tabela, o que é desnecessário. Se o mesmo caso fosse aplicado utilizando a opção *upWhereChanged* a instrução seria:

```
UPDATE CLIENTES SET TELEFONE='555-5525'
WHERE TEFONE=TELEFONE
```

Nesse caso teríamos problemas, pois poderíamos perder a referência e atualizar o registro incorreto. O melhor de todos os casos é usar a opção *upWhereKeyOnly* que monta a instrução usando apenas os campos da chave primária, ex:

```
UPDATE CLIENTES SET TELEFONE='555-5525'
WHERE ID=ID
```

Por fim restam as configurações dos campos no *ClientDataSet*. Nesse caso a configuração acontece campo a campo nos *Field's Editor* do objeto. O principal objetivo dos *ProviderFlags* é informar ao *DataSetProvider* qual a providência será tomada com cada campo. O mais importante é entender que nem todos os campos precisam ser alterados no banco de dados, como campos vindos de *Joins* com outras tabelas. Como informar *DSP* que determinado campo pertence a outra tabela e não é necessário gravá-lo? É aí que entram os *ProviderFlags*. Suas opções são:

- *pfInUpdate*: Campo incluso nos *Updates*;
- *pfInWhere*: Campo incluso na cláusula *Where*; Usado para encontrar o registro original;
- *pfInKey*: Campo chave; Usado para encontrar o registro original;
- *pfHidden*: Campo oculto; Campo incluso do *Data Packet*, mas é usado apenas para encontrar o registro original;

Para ser mais direto, cada campo em um *Field's Editor* pode receber um

ProviderFlag diferente. Então pensamos na seguinte situação: nosso usuário entrou na tela de cadastro de Clientes e alterou apenas o nome do cliente. Ao efetuar um *ApplyUpdates*, o *DSP* entra em ação e fará a montagem da instrução. A instrução mais sensata que o *DSP* precisa montar consiste apenas em fazer um *Update* usando como campo de alteração o Nome, e na cláusula *Where* necessitamos apenas dos campos da chave, que nesse caso vamos imaginar o *ID* do cliente. Uma instrução seria:

```
UPDATE CLIENTES SET NOME='JOSE DA SILVA'
WHERE ID=ID
```

Para configurar o *ClientDataSet* de tal forma que essa seja a instrução base, precisamos modificar os *ProviderFlags* do campo *ID* parara [*pfInUpdate*, *pfInWhere*, *pfInKey*]. Os demais campos ficam configurados apenas com as duas primeiras opções. Dessa forma garantimos que os dados serão atualizados corretamente.

Configurando dinamicamente UpdateMode e ProviderFlags

Essa última etapa de nosso artigo demonstra como fazer a configuração do *UpdateMode* e *ProviderFlags* dinamicamente, o que na verdade não há segredo algum.

A propriedade *UpdateMode* do *DataSetProvider* possui apenas três opções, e para modificá-la em tempo de execução apenas atribua o valor diretamente a sua propriedade, veja:

```
DataSetProvider1.UpdateMode := upWhereAll;
```

Já o *ClientDataSet* a história muda de figura. Nele, sua propriedade *ProviderFlags* é do tipo enumerado, ou seja, pode ter mais de um valor. Mesmo assim ainda é bastante simples de fazer a implementação. Apenas atribua o valor entre colchetes ao campo que deseja adicionar os *ProviderFlags*, como segue:

```
ClientDataSet1.FieldByName('CustNo').
ProviderFlags := [pfInUpdate,pfInKey];
```

Como pode ver, é muito simples. Só é preciso tomar muito cuidado em qual o momento que serão adicionados ou

retirados os atributos do *ProviderFlags*, pois toda a regra de negócio ou parte dela, depende dessas propriedades bem configuradas.

Considerações finais

O componente *TDataSetProvider* ainda possui uma série de outras funcionalidades que não puderam ser vistas aqui por conta da sua complexidade, como por exemplo *Nested DataSets* e o uso de pacotes de dados personalizados para montagem de *token's* de validação ou *logs* de evento.

É altamente recomendável fazer um estudo aprofundado de todos os mecanismos e técnicas pra utilização de todos ou pelo menos grande parte dos recursos desse fantástico componente. Importante também lembrar que é perfeitamente possível utilizar outras estruturas de componentes com o *DSP*, como por exemplo *TDataBases>TQuery>TDataSetProvider>TClientDataSet*.

Conclusão

Falar sobre *TDataSetProvider* não é fácil, visto que sua complexidade é bastante alta e a quantidade de recursos existentes é fantástica. O uso de *DataSetProvider* no dia-a-dia, faz com que a programação seja sempre uma caixinha de surpresa, já que somos capazes de descobrir inúmeras finalidades e funcionalidades para o componente. Por isso é recomendável que se estude a fundo todas as suas particularidades. Nesse artigo vimos os principais aspectos da programação com *DSP* e seus principais recursos, dicas e macetes.

Espero que tenham gostado. Um forte abraço e até a próxima. ●

Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

