Delphi

.NET - XML - InterBase - Firebird - SQL - POO - Cartas

www.clubedelphi.net feita para desenvolvedores

ClubeDelphi

ano 5

Edição 64 • R$ 8,90

www.clubedelphi.net

Apoio Borland®

DevMedia GROUP

Desenvolvimento MULTI CAMADAS

Migração de aplicações duas camadas para n-tier

Técnicas avançadas com DataSnap e COM+

Iniciante

Curso de SQL
Tudo sobre Selects, Joins, Updates e mais!

QuantumGrid
Conheça o melhor Grid para Delphi

InterBase 7.5
Rodando múltiplas instâncias do servidor

Segurança de aplicações
Restringindo acesso a campos e operações em tabelas

Crystal Reports
Relatórios na Web com ASP.NET

Intermediário

Delphi Language
Usando Class Helpers

Aplicação completa para IB/FB
Triggers, Stored Procedures e regras de negócio no BD

Avançado

MMLogs
Criando logs para exceções

ISSN 1517990-7

Proteja seu Software

ANTIPIRATARIA

DISTRIBUIÇÃO SEGURA

AUMENTO DOS SEUS LUCROS

PROTEÇÃO DA
PROPRIEDADE INTELECTUAL

Compact500®

Segurança. Qualidade. Eficiência.

Isto é a solução Compact-500. Você tem a garantia de proteção do seu software contra pirataria e engenharia reversa, além de controle total do gerenciamento de seu programa. Tudo isto de maneira fácil e extremamente eficiente.

Só uma empresa com um histórico de inovação, bom atendimento e sucesso pode garantir a segurança do seu bem mais valioso: suas informações.

Compact-500
A segurança do seu software

SafeNet®
The Foundation of Information Security

São Paulo (11) 4208 7700 http:brsafenet-inc.com

ClubeDelphi

Edição 64 . Novembro. 2004 . Ano IV . R$ 8,50

Coordenador editorial
Gladstone Matos
gladstone@clubedelphi.net

Editor geral
Guinther Pauli
guinther@clubedelphi.net

Editor técnico e Revisor
Luciano Almeida Pimenta
lucianopimenta@clubedelphi.net

Contribuíram nesta edição
Bruno Sonnino, Mendo Leonel, Fábio Sarturi Prass , Fernando Sarturi Prass, Marcelo Pereira Rocha , Adail Muniz Retamal, Ricardo Balbinot, Fladhymir C. Castello, Daniel K. Silva , Ricado do Amaral Soares, Gustavo Chaurais, Andreano Lanusse , Laércio Queiroz, Everson Volaco, Luciano Pimenta

Arte
Antonio de Sousa Jr.
antonio@devmedia.com.br

Capa
André Pinto
andre@casullo.com.br

Supervisão técnica
Alfredo Ferreira
alfredo@devmedia.com.br

Publicidade
David Niegeski
publicidade@devmedia.com.br

Gerente de Marketing
Kaline Dolabella
kaline@clubedelphi.net

Na Web
www.clubedelphi.net

Atendimento ao leitor
Cristiane Queiroz
admin@clubedelphi.net
(21) 2283-9012

Distribuição
Fernando Chinaglia Distribuidora S/A

A senha deste mês para o Portal do Assinante é: **ZAPNOVL**

Editorial

O desenvolvimento para DataSnap utilizando técnicas avançadas do COM+ sempre foi um tema em alta e bastante solicitado por nossos leitores, de forma que nesta edição iniciaremos uma série de artigos sobre técnicas relacionadas ao assunto.

Você aprenderá sobre migração de aplicações duas camadas para três camadas, otimização de acesso a banco de dados, invocação de métodos remotos, criação de regras de negócio no servidor de aplicação, configuração de *Connection Pooling* e *Object Pooling* (para aumento de escalabilidade), uso de transações COM+, segurança na camada de negócio, uso de *Shared Connections*, chamadas assíncronas com MSMQ, *CallBacks* e muito mais!

Começando pela migração facilitada de aplicações client/server para multicamadas, a série de artigo aborda diversos outros conceitos muitas vezes desconhecidos pela maioria de nós desenvolvedores, que podem fazer a diferença na hora de projetar a arquitetura de um sistema de médio a grande porte. Esse será o nosso foco nesta e nas próximas edições, nos artigos do nosso novo colaborador Gustavo Chaurais.

Um abraço a todos, uma ótima leitura e sucesso nos projetos multicamadas com o Delphi!

Guinther Pauli
guinther@clubedelphi.net
Editor

Realização

Apoio

Índice

06 - **DevExpress QuantumGrid**
Incrementando o uso de Grids
por Bruno Sonnino

13 - **InterBase 7.5**
Rodando múltiplas instâncias do servidor
por Mendo Leonel

15 - **Structured Query Language**
Guia de referência
Parte II: Comandos DML
por Fábio Sarturi Prass e Fernando Sarturi Prass

20 - **Crystal Reports**
Utilizando arquivos XSD para definição de relatórios no Delphi 2005
por Marcelo Pereira Rocha e Adail Muniz Retamal

23 - **MMLogs**
Uma biblioteca para auxílio à depuração de falhas
por Ricardo Balbinot, Fladhymir C. Castello, Daniel K. Silva e Ricado do Amaral Soares

27 - **Aplicações Multicamadas**
Migração simplificada de duas camadas para n-tier
por Gustavo Chaurais

31 - **Date, Time e TimeStamp no InterBase**
Diferenças e operações básicas com os tipos de dados para armazenamento de data / hora
por Andreano Lanusse

35 - **Novidades da Delphi Language**
Parte I - Usando Class Helpers
por Laércio Queiroz

37 - **Sistema SysCash**
Parte II – Criando as regras de negócio
por Everson Volaco

47 - **Segurança de aplicações**
Gerenciando usuários, permissões e restrições
por Luciano Pimenta

Cartas

Cache de dados no ASP.NET

Sou leitor da revista há anos e gostaria de parabenizar toda a equipe ClubeDelphi. A qualidade técnica das publicações é incomparável, os artigos têm ajudado muito em meus estudos com o Delphi. Aproveito a oportunidade para tirar uma dúvida com relação ao ASP.NET. Tenho ouvido falar muito no uso de DataSets em Cache, para aumento de performance de minhas aplicações de banco de dados. Como usar esse recurso em aplicações com o Delphi for .NET?

Marco Valério Oppa (Gringo)

Olá Marco. Agradecemos os elogios. Continuaremos empenhados ao máximo para trazer o melhor do Delphi a cada mês, mantendo sempre nosso padrão de qualidade e diferencial. Vamos esclarecer sua dúvida com um pequeno exemplo.

Aplicações ASP.NET são *state-less*. Isso significa dizer que o servidor trata cada requisição cliente como se fosse uma nova solicitação. Nenhuma informação é armazenada na memória do servidor após o envio da página HTML para o browser. É claro, o ASP.NET utiliza alguns recursos especiais (como *ViewState*) para recuperar o estado de controles.

Em algumas situações, no entanto, é preciso armazenar um objeto no servidor para que possa ser recuperado em requisições subseqüentes. O ASP.NET oferece vários recursos para armazenar dados em cache, através de objetos como *Session, Cache* e *Application*. O *Session* permite que você guarde informações para um determinado usuário, como os itens de um carrinho de compra, por exemplo.

Diferente do *Session*, os dados colocados no objeto *Cache* são armazenados em nível de aplicação, ou seja, são visíveis para todos os usuários. Imagine uma página de cadastro que possui uma lista de opções, para que o usuário selecione um país de origem (Brasil, USA etc.). Esses dados estão cadastros em uma tabela, raramente mudarão no BD e normalmente são sempre os mesmos, não importa qual o usuário que solicita a página. Outro exemplo é uma listagem de vestibular, onde os aprovados ou as fichas de desempenho final dificilmente mudam. Para que fazer uma consulta ao BD se a saída é sempre a mesma?

O objeto *Cache* é especial nesse caso, pois permite que a consulta à tabela seja feita uma única vez para preencher a lista. Quando um segundo usuário solicitar a página, o ASP.NET usa os dados em cache.

Para um exemplo, inicie uma nova aplicação ASP.NET no Delphi. Insira no formulário um *BdpConnection* e configure nele uma nova conexão ao banco *Employee.gdb* do InterBase ou Firebird. Coloque no formulário um *DBWebGrid* e um *Label*. Declare na sessão *private* da classe um método chamado "CarregaDados" e implemente-o conforme a listagem a seguir.

```
procedure TWebForm1.CarregaDados;
var
  dts: DataSet;
  adp: BdpDataAdapter;
  ds: DBWebDataSource;
begin
  if Cache['PAISES'] = nil then
  begin
    adp := BdpDataAdapter.Create(
    'select * from COUNTRY',BdpConnection1);
    dts := DataSet.Create;
    adp.Fill(dts,'COUNTRY');
    Cache['PAISES'] := dts;
    Label1.Text := 'A cache foi carregada agora';
  end
  else
   Label1.Text := 'Estamos utilizando os dados que
      já estavam em cache';
  ds := DBWebDataSource.Create;
  ds.DataSource := Cache['PAISES'] as DataSet;
  DBWebGrid1.DBDataSource := ds;
  DBWebGrid1.TableName := 'COUNTRY';
end;
```

No código anterior carregamos os dados da tabela *Country* em um *DataSet* e o armazenamos no objeto *Cache*, passando uma *string* para identificar o objeto (nesse caso "PAISES"). Com isso, você pode armazenar mais objetos em cache (por exemplo, outros *DataSets*). Chame o método *CarregaDados* no evento *Load* do formulário.

Compile e execute a aplicação. Faça um teste: abra a aplicação em um browser e visualize os dados. Copie a URL para a área de transferência. Desligue o InterBase, abra um segundo navegador e cole a URL copiada. Os dados ainda estarão lá, comprovando que a consulta SQL não foi feita e que os dados foram retornados a partir da cache.

Cartas publicadas podem ser editadas por motivos de clareza ou extensão.

Guinther Pauli - Redação
Luciano Pimenta - Redação
revista@clubedelphi.net

DevExpress QuantumGrid

por **Bruno Sonnino**

Incrementando o uso de Grids

A apresentação e entrada de dados no formato de tabela é uma coisa bastante comum nos programas Delphi. Muitas vezes, estamos procurando uma característica especial, que não é fornecida nos componentes que vêm com o Delphi para essa finalidade: *StringGrid*, *DrawGrid* e *DBGrid*.

Algumas dessas funcionalidades mais procuradas são o desenho especial, como quando desenhamos os números negativos em vermelho, o desenho de *memos* ou figuras nas células da grade, ou o uso de editores especiais nas células: *ComboBoxes, CheckBoxes, Memos* etc.

Os fornecedores de componentes apresentam uma variedade enorme de componentes desse tipo, que implementam alguma funcionalidade, mas dificilmente algum Grid tem tudo o que necessitamos. Terminamos assim usando diversos componentes do tipo Grid, um para cada finalidade. O ideal é que pudéssemos usar apenas um componente, que permitisse englobar todos os usos que desejamos.

Na busca pelo Grid ideal, encontrei o *DevExpress QuantumGrid*, um componente que foi apontado como melhor componente VCL pelos leitores da revista *Delphi Informant* (*Delphi Informant Magazine's Readers Choice Awards*) nos anos de 2001 a 2004.

Introdução ao DevExpress QuantumGrid

A *DevExpress*, desenvolvedora do *QuantumGrid*, não disponibiliza uma versão de teste, embora tenha uma política de devolução de dinheiro até os 60 dias de registro. Ao instalar esse componente, a primeira impressão que temos é de intimidação: são criados 13 diretórios, quatro guias na paleta de componentes e instalados mais de 70 novos componentes. Além do Grid, são instalados diversos editores, em versões não ligadas a dados como também *Data-Aware*.

A segunda surpresa vem ao colocar o componente *cxGrid* no formulário: onde estão as propriedades do componente? O componente tem relativamente poucas propriedades, a maioria relativa à aparência (bordas, alinhamento). Como é feita a ligação do *DataSource* ao Grid?

Na realidade, não há nada de errado com a instalação. Esta grid tem uma filosofia diferente das demais, visando sua flexibilidade. O Grid é um "container", guardando mais de uma visualização dos dados. Desta maneira, um mesmo Grid pode mostrar diversos dados de uma vez ou o mesmo dado, de maneiras diferentes. O esquema de trabalho com o Grid é o seguinte: um Grid pode conter diversos níveis, que dão uma noção hierárquica dos dados. Por exemplo, se quisermos mostrar uma relação mestre/detalhe no Grid, como os clientes e seus pedidos, usaremos dois níveis, um principal, para os clientes e um sub-nível, para os pedidos.

Cada nível pode conter diversas visualizações. Por exemplo, podemos ver os clientes como uma tabela ou no formato de "cartões", onde cada campo está em uma linha do quadro. Com essa maneira de organizar os dados temos a flexibilidade de mostrar as informações que queremos, podendo facilmente alterar a maneira de apresentá-los ou mesmo mudar o que será apresentado.

Vamos então criar nossa primeira aplicação usando o *QuantumGrid*.

Criando o primeiro exemplo com o Grid

Crie uma nova aplicação, colocando um *cxGrid*, configurando sua propriedade *Align* para *alClient*. Coloque um *ClientDataSet*, dando um clique de direita com o mouse e selecionando *Load from MyBase table*. Selecione o arquivo *animals.xml* que encontra-se por padrão em *Arquivos Comuns\Borland\Data* dentro do diretório *Arquivos de Programas*. O arquivo XML é carregado para o *ClientDataSet*.

Adicione um *DataSource*, ligando-o ao *ClientDataset*. O passo seguinte é configurar o Grid, ligando-o ao *DataSet*. Na parte inferior do Grid há um quadro, como mostrado na **Figura 1**.

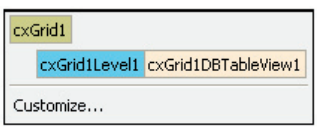


Figura 1. Quadro com a configuração default da cxGrid

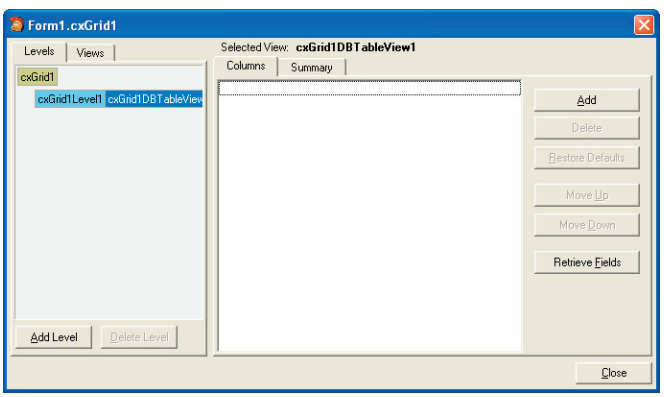


Figura 2. Tela para personalização da Grid

Quando se insere um Grid no formulário, são criados um nível e uma *view* automaticamente. Clique no botão *Customize* para personalizar o Grid. Abre-se uma janela semelhante à da **Figura 2**. A ligação

entre o Grid e os dados é feita na *View*, usando um *DataController*. Essa é uma abstração de dados que permite ligar tanto dados de uma tabela como dados vindos da memória ou de um arquivo.

Clicando em *cxGrid1DBTableView1* podemos personalizar a *view*. A primeira providência é ligar o *DataSource* ao Grid. Configuramos o *DataSource* da propriedade *DataController*, indicando o *DataSource1*. Em seguida, clicamos no botão *Retrieve Fields*. Os campos da tabela são recuperados e inseridos no Grid. Clicando no botão *Close*, a janela é fechada e os dados são mostrados no Grid.

Executando a aplicação, vemos que, sem uma linha de código, geramos uma aplicação relativamente complexa, que permite classificar os dados com um clique na barra de título, filtrar os dados, selecionando o filtro no *ComboBox* da barra de título ou mesmo agrupar os dados, arrastando a coluna que queremos agrupar para a parte superior da janela.

Como podemos notar na **Figura 3**, as imagens não são mostradas. Para que isso aconteça, devemos fazer duas coisas: através do botão *Customize*, selecionar a *view* e em *OptionsView*, configuramos a propriedade *CellAutoHeight* para *True*.

Isso faz com que a célula tome a altura necessária para apresentar o campo. Em seguida, devemos personalizar o Grid e selecionar o campo relativo à imagem, alterando a propriedade *Properties* para *Image*. Ao fechar a janela de configuração, as imagens são mostradas (**Figura 4**).

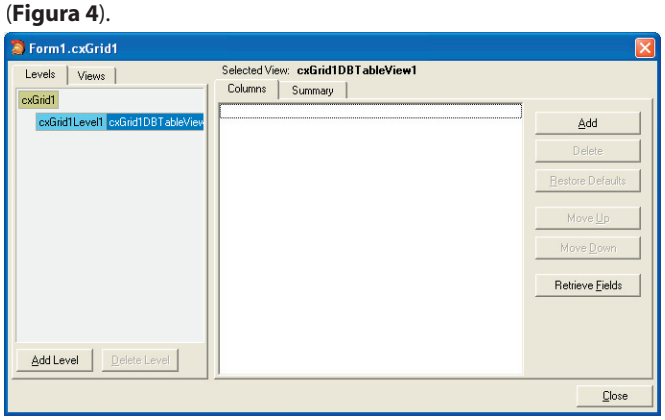


Figura 3. Projeto em execução com dados agrupados por área

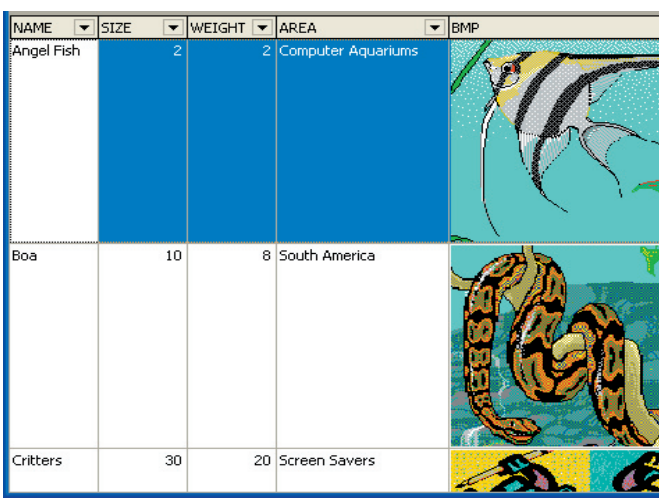


Figura 4. Grid com imagens

Múltiplas visualizações dos dados

Como foi citado anteriormente, a organização do Grid permite que se possa ter mais de uma visualização do mesmo dado. Para isso, deve-se criar uma nova *view* e associá-la ao nível desejado. Clique em *Customize* para abrir a caixa de diálogo de personalização e na guia *Views* clique em *Add Views*, selecionando *DB Card View*. Faça a ligação à propriedade *DataController|DataSource* e adicione os campos clicando em *Retrieve Fields*.

Selecione *OptionsView* e configure a propriedade *CellAutoHeight* para *True*. Selecione o campo BMP e altere a propriedade *Properties* para *Image*. Em seguida, iremos alternar entre as duas visualizações com um *RadioGroup*.

No projeto, selecione o Grid e altere a propriedade *Align* para *alNone*. Abra um espaço na parte inferior da tela e insira um *Panel*, mudando sua propriedade *Align* para *alBottom*. Limpe a propriedade *Caption*. Coloque um *RadioGroup* no *Panel*, alterando a propriedade *Columns* para "2". Na propriedade *Items*, insira os seguintes valores: "Tabela" e "Ficha". Altere a propriedade *ItemIndex* para "0". No evento *OnClick* do *RadioGroup*, insira o seguinte código:

```
case RadioGroup1.ItemIndex of
  0: cxGrid1Level1.GridView := cxGrid1DBTableView1;
  1: cxGrid1Level1.GridView := cxGrid1DBCardView1;
end;
```

Dessa maneira, ao clicar no *RadioGroup*, alteramos a visão dos dados. Altere a propriedade *Align* do Grid para *alClient* e execute a aplicação. A **Figura 5** mostra a aplicação em execução, com a visualização em forma de ficha. Note que também nessa visualização podemos facilmente filtrar os dados, bastando para isso selecionar o dado desejado no *ComboBox* correspondente.

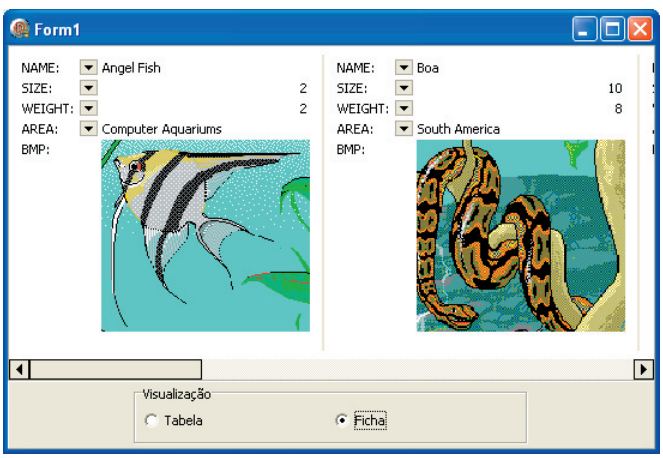


Figura 5. Aplicação em execução com dados em forma de ficha

Apresentação de dados mestre/detalhe

Para apresentar os dados de duas tabelas, no formato mestre/detalhe, devemos criar dois níveis no Grid, um principal, para a tabela mestre e um sub-nível para a tabela detalhe. Em cada um desses níveis criamos uma *view* que é ligada ao *DataSet* correspondente.

Crie um novo projeto, acrescentando dois *ClientDataSets* e dois *DataSources*. Clique com o botão direito no primeiro *ClientDataSet*, selecionando *Load from MyBase table* e carregue o arquivo *customer.xml* conforme técnica utilizada no exemplo anterior.

Faça o mesmo com o segundo *ClientDataSet*, selecionando o arquivo *orders.xml*. Ligue os *DataSources* aos *ClientDataSets*.

Coloque um Grid (*cxGrid*) no formulário, alterando a propriedade *Align* para *alClient*. Em seguida, iremos criar os níveis e a visualização. Clique com o botão direito no nível (*Level*) criado automaticamente no Grid e selecione *Add Level*. Isso adiciona um sub-nível ao nível atual. Esse sub-nível não tem nenhuma *view* associada a ele. Clique com o botão direito do mouse nele e selecione *Create View|DB Table*.

Assim, criamos nosso nível detalhe com sua respectiva *view*. Ligue a *view* mestre ao primeiro *DataSource*, preenchendo a propriedade *DataController|DataSource* e faça o mesmo com o segundo *DataSource*, ligando-o à *view* detalhe. Adicione os campos nas *views* através do botão *Retrieve Fields* no editor do Grid.

O último passo é indicar os campos que ligam a tabela mestre à detalhe: isso é feito selecionando a *view* detalhe e preenchendo as propriedades *MasterKeyFieldsNames*, *DetailKeyFieldsNames* e *KeyFieldsNames* com o nome do campo que liga a tabela mestre e detalhe (esse campo é *CustNo* em ambos os arquivos). A tabela detalhe deve estar indexada pelo campo chave para que os dados sejam encontrados. Para isso, devemos colocar o campo *CustNo* na propriedade *IndexFieldNames* do *ClientDataSet2*.

Ao executar, temos um Grid mestre/detalhe, onde podemos classificar os dados ou mesmo agrupá-los por uma coluna. A **Figura 6** mostra o Grid agrupado por país, com os dados dos clientes e seus respectivos pedidos.

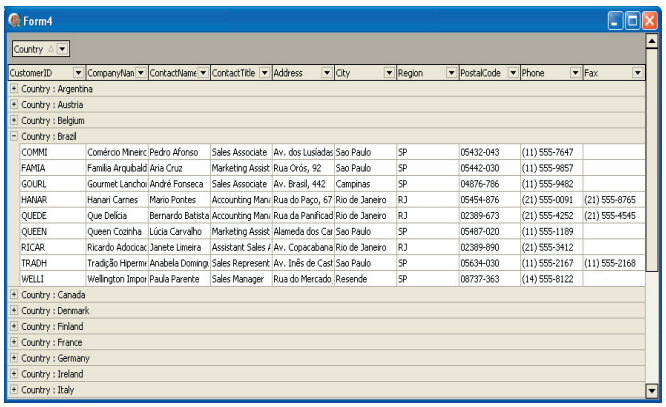


Figura 6. Grid mestre/detalhe

Totalizando Grupos

Iremos incrementar um pouco esse Grid, acrescentando totais nos grupos e nos rodapés. Inicialmente, colocaremos a quantidade de pedidos e o valor total no rodapé dos detalhes. Clique em *Customize*, no Grid, para personalizar os detalhes.

Na janela de diálogo, selecione a *view* de detalhe e clique na guia *Summary*. Na guia *Footer* clique no botão *Add* para adicionar um novo item de sumário. No *Object Inspector*, configure a propriedade *Column* para *cxGrid1DBTableView2ItemsTotal*, a coluna referente ao total do pedido.

Configure as propriedades *Format* para ",#0.00" e *Kind* para *skSum*. Dessa maneira, o total dos pedidos será apresentado na linha de sumário. Em seguida adicione mais um item de sumário, configurando as propriedades *Column*, *Format* e *Kind* para *cxGrid1DBTablEView2OrderNo*, "Total de pedidos: 0" e *skCount*, respectivamente. Para que o rodapé da tabela seja apresentado, devemos selecionar a propriedade *OptionsView|Footer* da *view* para *True*.

Para adicionar um total de grupo, devemos selecionar a *view* mestre e, na guia *Summary*, selecionar a sub-guia *Default For Groups*. Clique no botão *Add* para adicionar um novo item, configurando a propriedade *Column* para *cxGrid1DBTableView1CustNo*, *Format* para "Total de clientes: 0" e *Kind* para *skCount*, respectivamente.

Dessa maneira, criamos um total de grupos para a tabela mestre, que será mostrado quando essa estiver agrupada. Execute a aplicação e arraste o cabeçalho do campo *Country* para a parte superior do Grid, de forma a agrupar os clientes por país. O total de clientes em cada país é mostrado ao lado do grupo. Abrindo-se o detalhe de um cliente, podemos observar o número de pedidos e o seu total na última linha do Grid detalhe. A **Figura 7** mostra a aplicação com os sumários.

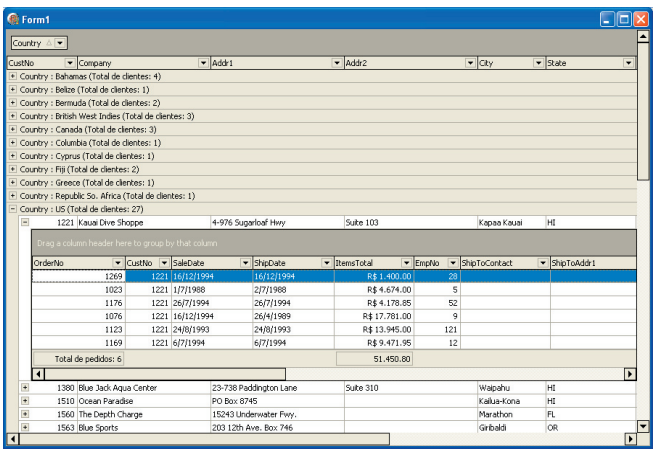


Figura 7. Aplicação mestre/detalhe com sumários por grupo e no detalhe

"Zebrando" o Grid

Podemos agora melhorar a aparência de nosso Grid, acrescentando cores a ele. Inicialmente, faremos o Grid detalhe mostrar os dados como um formulário "zebrado", onde as cores das linhas alternam-se entre azul e branco. Selecione a *view* detalhe e na propriedade *Styles*, selecione a propriedade *ContentEven*. Selecione a opção *Create a new style in the new repository*.

O Grid guarda os estilos definidos em um componente *cxStyleRepository*. Quando queremos alterar o estilo de algum componente do Grid, deve-se referir a um estilo armazenado no repositório. Ao selecionar a opção, um *cxStyleRepository* é adicionado ao formulário.

Altere a propriedade *Color* de *ContentEven* para *clSkyBlue*. Dessa maneira, as linhas pares terão cor azul. Para a *view* mestre, alteraremos a aparência do Grid como um todo. Para isso, usaremos um dos estilos predefinidos. Dê um clique duplo no componente *cxStyleRepository* para abrir seu editor e, na guia *Style Sheets*, clique no botão *Predefined* para carregar os estilos predefinidos.

Selecione o estilo *DevExpress* e clique em *Load*. Com esse estilo carregado, podemos atribuí-lo à *view*. Feche o editor e selecione a *view* mestre. Selecione a propriedade *Styles|StyleSheet* e preencha com *GridTableViewStyleSheetDevExpress*, o estilo que acabamos de carregar.

O Grid muda completamente sua aparência. Ao executar o programa, podemos verificar a alteração da aparência, tanto na tabela mestre como na detalhe (**Figura 8**).

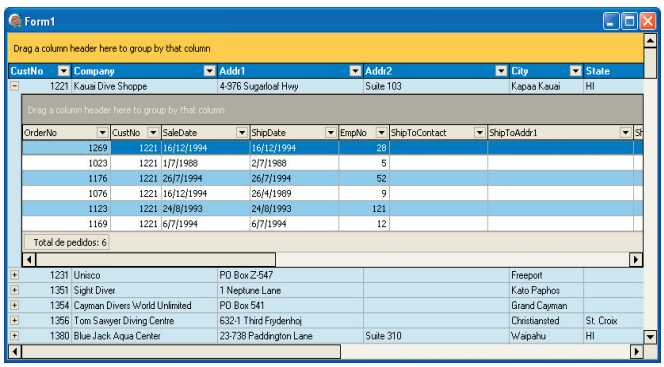


Figura 8. Grid com aparência modificada utilizando estilos predefinidos

Apresentação de dados não vinculados a um BD

Até aqui vimos Grids ligados a arquivos de dados (também poderíamos usar tabelas do banco de dados). O *QuantumGrid* permite apresentar dados que vêm de informações da memória ou de arquivos texto. Ele define três tipos de acesso a dados: *Bound*, *Unbound* e *Provider*. O primeiro modo é ligado aos dados de uma tabela, como vimos até agora. O segundo não é ligado aos dados e é semelhante a usar um *StringGrid* normal, onde dizemos quantos itens o Grid irá conter e informamos o conteúdo das células. O modo *Provider* liga o Grid a uma fonte de dados personalizada, solicitando as informações, quando necessário.

Para ilustrar o modo *Unbound*, criaremos uma Grid mestre/detalhe que mostrará na *view* mestre os arquivos e diretórios do Drive C. Na *view* detalhe será mostrado o conteúdo dos subdiretórios. Para não prolongar muito o programa, isso será feito apenas com o primeiro nível de diretório.

Crie um novo projeto, inserindo um *cxGrid* no formulário. Esse Grid já vem com uma *DBTableView*, que não será usada. Selecione-a, clicando com o botão direito do mouse, escolhendo a opção *Delete View*. Em seguida, clique com o botão direito do mouse, selecionando *Create View*, escolhendo *Table*. Após, abra o menu de contexto do nível, selecionando *Add Level* para adicionar o sub-nível.

Finalmente, adicione uma *Table* a esse nível. Clique em *Customize* para criar as colunas. Como esse Grid não está ligado a dados, devemos criar nossas colunas manualmente. Selecione a *view* mestre e adicione três colunas, configurando as propriedades *Caption* para "Nome", "Tamanho" e "Data", em cada uma. Configure a propriedade *Width* para "250", "100" e "100", respectivamente.

Devemos ainda informar o tipo dos dados que preencherão a coluna. Altere a propriedade *DataBinding|ValueType* das colunas para *String*, *Integer* e *DateTime*, respectivamente.

Uma vez configuradas as colunas da *view* mestre, iremos configurar as colunas da *view* detalhe. Selecione a *view* detalhe e crie três colunas iguais à da *view* mestre. O passo seguinte é criar o código que preenche o Grid. No evento *OnCreate* digite o código da **Listagem 1**.

Listagem 1. Código para mostrar pastas e diretórios

```
procedure TForm1.FormCreate(Sender: TObject);
var
  SRec, SRec1: TSearchRec;
  Done, Done1: Integer;
  NumArqs, NumArqs1: Integer;
  ADetDataController: TcxCustomDataController;
begin
  cxGrid1TableView1.BeginUpdate;
  try
   { Pesquisa arquivos no diretório raíz }
    Done := FindFirst('c:\*.*', faAnyFile,Srec);
    NumArqs := 0;
    cxGrid1TableView1.DataController.RecordCount := 0;
    while Done = 0 do
    begin
      if (SRec.Name <> '.') and
       (SRec.Name <> '..') then
      begin
        { Achou arquivo, adiciona registro }
        cxGrid1TableView1.DataController.RecordCount:=
          cxGrid1TableView1.DataController.RecordCount+1;
        cxGrid1TableView1.DataController.Values
          [NumArqs, 0] := SRec.Name;
        cxGrid1TableView1.DataController.Values
          [NumArqs, 1] := SRec.Size;
        cxGrid1TableView1.DataController.Values
          [NumArqs, 2] := FileDateTodateTime(
          SRec.Time);
        if (SRec.Attr and faDirectory) <> 0 then
        begin
          { Diretório - pega dados do subdiretório }
          ADetDataController :=
            cxGrid1TableView1.DataController.
            GetDetailDataController(NumArqs, 0);
          with ADetDataController do
          begin
            BeginUpdate;
            try
              RecordCount := 0;
              Done1 := FindFirst('c:\'+
                SRec.Name+'\*.*', faAnyFile, SRec1);
              NumArqs1 := 0;
              while Done1 = 0 do
              begin
                if (SRec1.Name <> '.') and
                  (SRec1.Name <> '..') then
```

```
                begin
                 { Achou arquivo no subdiretório,
                  adiciona ao detalhe }
                  RecordCount := RecordCount+1;
                  Values[NumArqs1, 0] := SRec1.Name;
                  Values[NumArqs1, 1] := SRec1.Size;
                  Values[NumArqs1, 2] :=
                    FileDateTodateTime(SRec1.Time);
                  Inc(NumArqs1);
                end;
                Done1 := FindNext(SRec1);
              end;
            finally
              EndUpdate;
            end;
          end;
        end;
        Inc(NumArqs);
      end;
      Done := FindNext(Srec);
    end;
    FindClose(SRec);
  finally
    cxGrid1TableView1.EndUpdate;
  end;
end;
```

A maior parte desse código refere-se à pesquisa dos subdiretórios. Quando achamos um arquivo ou diretório no raiz, adicionamos ao Grid. Se o arquivo encontrado for um subdiretório, pesquisamos os arquivos dele, adicionando-os ao Grid detalhe. Para obter o *DataController* do Grid detalhe, usamos o código:

```
ADetDataController :=
  cxGrid1TableView1.DataController.
    GetDetailDataController(NumArqs, 0);
```

Com isso, podemos adicionar os dados ao Grid detalhe. Uma última alteração deve ser feita à *view* mestre: deve-se selecioná-la e alterar a propriedade *OptionsView|ExpandButtonsForEmptyDetails*, configurando-a para *False*. Isso faz com que os botões para expandir os detalhes só sejam apresentados para os subdiretórios. A **Figura 9** mostra a lista de arquivos do Drive C da minha máquina.

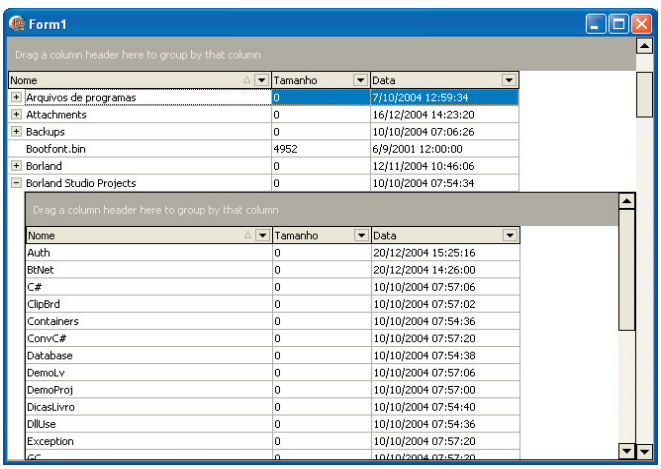


Figura 9. Mestre/detalhe não ligado a dados, utilizando diretórios e subdiretórios

Criando uma classe de cxCustomDataSource

O terceiro modo de utilização do Grid é usando o *Provider Mode*. Nesse modo, os dados são provenientes de uma fonte de dados customizada. Deve-se criar uma classe derivada de *cxCustomDataSource*. Nessa classe, implementamos os métodos *GetRecordCount*, *GetValue*, *SetValue*, *InsertRecord*, *AppendRecord* e *DeleteRecord*.

Esses métodos serão usados pelo Grid para obter e alterar os dados. Para exemplificar esse modo de acesso a dados, criaremos um Grid que mostra dados de um arquivo texto separados por tabulação. A primeira linha indica os nomes dos campos. O arquivo exemplo (*customer.txt*) pode ser baixado com os demais códigos dos projetos desenvolvidos neste artigo.

Os dados desse Grid serão criados dinamicamente. Assim, ele poderá mostrar qualquer arquivo texto onde os dados estão separados por tabulações, pois os dados da primeira linha indicam as colunas do Grid.

Crie um novo projeto e coloque um *CxGrid*, configurando sua propriedade *Align* para *alClient*. Clicando com o botão direito na *view* criada automaticamente, selecione *Delete View*. Nossa *view* não está ligada a dados, portanto devemos apagar a *view* criada automaticamente, criando uma nova, não ligada a dados.

Clique com o botão direito do mouse no nível, selecionando a opção *Create View|Table*. Como iremos criar nossas colunas em tempo de execução, não há mais nenhuma configuração a ser feita. Devemos criar uma classe derivada de *cxCustomDataSource*, que conterá os métodos para acesso aos dados. Crie uma nova unit e digite o código da **Listagem 2**.

Listagem 2. Criando uma nova classe descendente de CxCustomDataSource

```
unit Unit5;

interface

uses
  Classes, cxCustomData, Variants;

type
  TArqTextoDataSource = class(TcxCustomDataSource)
  private
    FArquivo: string;
    FArqLista: TStringList;
    FColunas: TStringList;
  protected
    function AppendRecord: TcxDataRecordHandle;
      override;
    procedure DeleteRecord(
      ARecordHandle: TcxDataRecordHandle); override;
    function GetRecordCount: Integer; override;
    function GetValue(
      ARecordHandle: TcxDataRecordHandle;
      AItemHandle: TcxDataItemHandle): Variant;
      override;
    function InsertRecord(
      ARecordHandle: TcxDataRecordHandle):
      TcxDataRecordHandle; override;
    procedure SetValue(
      ARecordHandle: TcxDataRecordHandle;
      AItemHandle: TcxDataItemHandle;
      const AValue: Variant); override;
  public
```

```
    constructor Create(AArquivo: string);
    destructor Destroy; override;
    property Colunas: TStringList read FColunas;
  end;

implementation

procedure QuebraLinha(Linha: string;
  Lista: TStringList);
var
  APos: Integer;
begin
  { Quebra uma linha delimitada por tabulações em
    uma StringList }
  Lista.Clear;
  while Linha <> '' do
  begin
    APos := Pos(#9,Linha);
    if APos > 0 then
    begin
      Lista.Add(Copy(Linha,1,APos-1));
      Linha := Copy(Linha,APos+1,Length(Linha));
    end
    else begin
      Lista.Add(Linha);
      Linha := '';
    end;
  end;

 if (AIndex >= 0) and (AIndex < ALista.Count) then
      Result := ALista[AIndex];
  finally
    ALista.Free;
  end;
end;

function TArqTextoDataSource.InsertRecord(
  ARecordHandle: TcxDataRecordHandle):
  TcxDataRecordHandle;
begin
  { Insere novo registro na lista }
  FArqLista.Insert(Integer(ARecordHandle),'');
  Result := TcxDataRecordHandle(ARecordHandle);
  DataChanged;
end;

procedure TArqTextoDataSource.SetValue(
  ARecordHandle: TcxDataRecordHandle;
  AItemHandle: TcxDataItemHandle;
  const AValue: Variant);
var
  ALista: TStringList;
  ALinha : Integer;
  AIndex : Integer;
  i : Integer;
begin
  { Muda valor do dado da lista }
  if Integer(ARecordHandle) >= FArqLista.Count then
    exit;
  ALista := TStringList.Create;
  try
    ALinha := Integer(ARecordHandle);
    { Quebra linha nos elementos }
    QuebraLinha(FArqLista[ALinha], ALista);
    AIndex := GetDefaultItemID(Integer(AItemHandle));
    if (AIndex >= 0) and (AIndex < ALista.Count) then
    begin
     { Substitui o elemento desejado }
      if not VarIsNull(AValue) then
 ALista[AIndex] := AValue
      else
        ALista[AIndex] := '';
      FArqLista[ALinha] := '';
      { Remonta a linha }
      for i := 0 to Pred(ALista.Count) do
        FArqLista[ALinha] := FArqLista[ALinha] + #9 +
          ALista[i];
      FArqLista[ALinha] := Copy(FArqLista[ALinha], 2,
        Length(FArqLista[ALinha]));
    end;
  finally
    ALista.Free;
  end;
end;

destructor TArqTextoDataSource.Destroy;
var
  Linha: string;
  i: Integer;
begin
  Linha := '';
  { Monta a linha das colunas }
  for i := 0 to Pred(FColunas.Count) do
    Linha := Linha + #9 + FColunas[i];
  Delete(Linha,1,1);
  { Insere linha na lista }
  FArqLista.Insert(0,Linha);
  { Salva arquivo modificado }
end;

function TArqTextoDataSource.AppendRecord:
   TcxDataRecordHandle;
begin
  { Ao adicionar um novo registro, adiciona linha na
    Stringlist }
  Result := TcxDataRecordHandle(FArqLista.Add(''));
  DataChanged;
end;

constructor TArqTextoDataSource.Create(
  AArquivo: string);
begin
  inherited Create;
  FArquivo := AArquivo;
  FColunas := TStringList.Create;
  FArqLista := TStringList.Create;
  FArqLista.LoadFromFile(AArquivo);
  { Pega primeira linha e cria colunas }
  if FArqLista.Count > 1 then
    QuebraLinha(FArqLista[0],FColunas);
  { Deleta a primeira linha, da lista com as colunas }
  FArqLista.Delete(0);
end;

procedure TArqTextoDataSource.DeleteRecord(
  ARecordHandle: TcxDataRecordHandle);
begin
  { Deleta linha da lista }
  FArqLista.Delete(Integer(ARecordHandle));
  DataChanged;
end;

function TArqTextoDataSource.GetRecordCount: Integer;
begin
  { Retorna número de registros }
Result := FArqLista.Count;
end;
```

Esse código implementa os métodos necessários para fornecer os dados para o Grid. Lemos o arquivo em um StringList, montamos as colunas quebrando a primeira linha com as tabulações. Quando um item é necessário, pegamos a linha correspondente no StringList, a desmembramos para obter os diversos valores listados e retornamos o valor requerido. Ao final, o arquivo é salvo usando o método SaveToFile.

Declare uma variável pública chamada "FArqTextoDatasource" do tipo TArqTextoDataSource. Declare no uses a unit criada anteriormente e que contém a nossa classe. No evento OnCreate do formulário, devemos criar uma instância dessa classe, criar as colunas do Grid e atribuir a instância criada ao DataController, através do seguinte código:

```
procedure TForm4.FormCreate(Sender: TObject);
var
  i: integer;
begin
 { Cria instância do CustomDataSource }
  FArqTextoDataSource := TArqTextoDataSource.
    Create('customers.txt');

 { Cria colunas da grid }
  for i := 0 to Pred(FArqTextoDataSource.Colunas.Count)do
    with cxGrid1TableView1.CreateColumn do
    begin
      Caption := FArqTextoDataSource.Colunas[i];
      DataBinding.Data := Pointer(i);
    end;

 { Atribui a instância criada ao DataController }
  cxGrid1TableView1.DataController.CustomDataSource :=
    FArqTextoDataSource;
end;
```

Devemos apenas destruir a instância criada no evento OnDestroy, para liberar a memória e salvar os dados:

```
procedure TForm4.FormDestroy(Sender: TObject);
begin
  FArqTextoDataSource.Free;
end;
```

Ao executarmos o programa, o Grid carrega os dados do arquivo, permitindo editá-los e salvá-los novamente, como se fosse um *DataSet* (**Figura 10**). Todas as características do Grid estão disponíveis para uso. Uma mudança interessante a ser feita é alterar a propriedade *OptionsView|ColumnAutoWidth* da *view* para *True*, de maneira que as colunas se redimensionem quando o Grid é dimensionado.

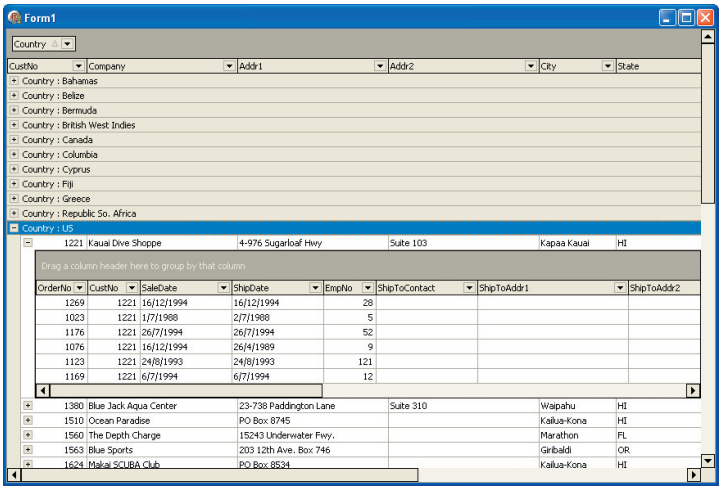


Figura 10. Grid mostrando dados de um arquivo texto, agrupados pelo campo Country

Conclusões

Esse é um Grid muito completo, suas opções de apresentação e customização cobrem uma gama muito grande de aplicações, permitindo visualizações bastante diferentes e complexas, utilizando pouco código. A mudança de visualizações, tanto no que se refere aos dados quanto à apresentação, é feita de maneira muito simples. O Grid ainda permite uma série de opções não mostradas aqui, como editores especiais, salvar e recuperar os dados do Grid em diversos formatos, entre outros.

O Grid ainda pode mostrar tanto dados ligados a uma tabela de banco de dados como informações provenientes de outras fontes, como memória ou arquivo. Pode-se, inclusive misturar num mesmo Grid, colunas ligadas e colunas não ligadas a dados.

Embora sua filosofia seja um pouco diferente dos demais Grids, o trabalho para aprender a usá-la compensa, pois o resultado final é sem dúvida melhor que qualquer outro Grid disponível no mercado.

Links

www.devexpress.com
Site da DevExpress, fabricante do produto

Bruno Sonnino *(sonnino@netmogi.com.br) é consultor de empresas e desenvolvedor Delphi desde 1995. É autor dos livros "Delphi e Kylix – Dicas para turbinar seus programas", "Kylix – Delphi para Linux" e "365 Dicas de Delphi". Foi palestrante das três Borcon Brasil e da 11ª Borcon USA, em Long Beach, Califórnia. Ele mantém um blog com dicas de Delphi em:*
www.revolution.com.br/blogdelphi.

Download:
www.devmedia.com.br/clubedelphi/downloads

InterBase 7.5

por **Mendo Leonel**

Rodando múltiplas instâncias do servidor

Temos uma nova versão do InterBase: a 7.5. Essa versão traz várias características que tornam o InterBase mais robusto, elevando ainda mais sua posição no mercado perante concorrentes. Neste artigo, vamos falar um pouco de uma das muitas novas características, a *Multi-Instance*, ou seja, a possibilidade de executarmos mais de uma instância do serviço em uma mesma máquina.

Você pode estar se perguntando: mas para que eu precisaria de mais de uma instância de um serviço de IB em uma mesma máquina? A resposta é tão simples quanto a pergunta. Você pode ter uma versão anterior do IB e uma aplicação que faz uso de recursos específicos. Simplesmente instalar um novo serviço poderia trazer vários problemas para a aplicação, sendo necessária uma manutenção prévia da aplicação.

Outra situação: ter um serviço do Firebird instalado, nesse caso, as incompatibilidades são ainda maiores, pois depois que a Borland decidiu voltar atrás e não mais fornecer o IB como um produto *open source*, as duas distribuições estão cada vez mais distantes em termos de compatibilidade, portanto tomem cuidado, a migração de bancos de dados entre os serviços do FB e IB pode não ser tão transparente quanto parece.

Instalando o InterBase com Multi-Instance

Na instalação do produto devemos considerar a possibilidade de utilizar várias instâncias e então decidir se vamos instalar com essa característica ou se simplesmente, podemos utilizar a atual. Mais adiante, entraremos em maiores detalhes sobre o serviço, como iniciá-lo e em qual porta ele "responde", nesse momento vamos nos concentrar em instalar o IB.

A instalação é simples, no entanto, logo após escolher a opção de instalação do *Server/Client*, será apresentada uma janela perguntando se você deseja instalar o IB com a opção de *multi-instance*. Responda *Yes* e continue a instalação. Na tela seguinte você deverá informar qual porta o serviço que está sendo instalado "responderá" (veja box *Port e Instance Name*) e com qual nome esse serviço será identificado (**Figura 1**).

Para este artigo eu instalei em uma mesma máquina o Firebird 1.5 e duas instâncias do IB 7.5. Os serviços podem ser vistos na **Figura 2**. Perceba que as instâncias do IB são nomeadas de acordo com o *Service Name* indicado na instalação.

As configurações do exemplo são as seguintes: o FB responde as requisições na porta *default 3050/tcp* com o *Service Name* de *gds_db*, a primeira instância do IB responde na porta *3051/tcp* com o nome de *gds_db75* e a segunda instância do IB responde na porta *3052/tcp* com o nome de *gds_db75_2*.

Pronto, já temos várias instâncias do IB rodando em uma mesma máquina e ainda por cima com um FB também rodando nessa máquina.

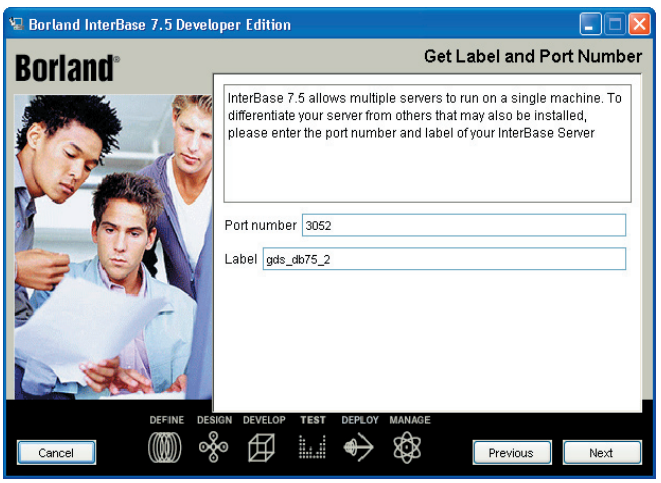


Figura 1. Definindo a porta de comunicação do IB e nome do serviço

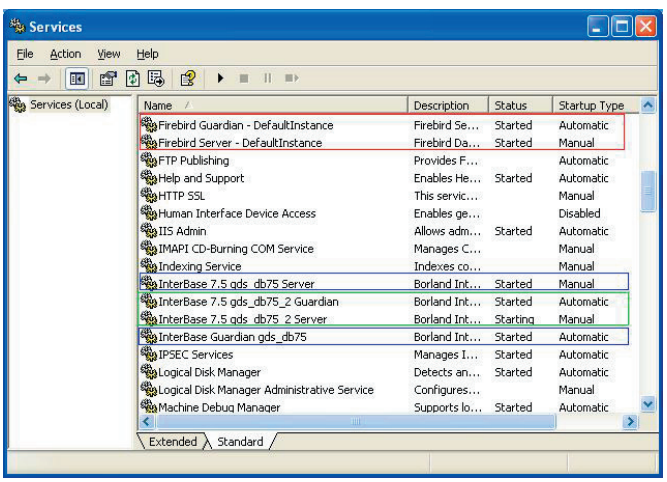


Figura 2. Serviços sendo executados ao mesmo tempo

Acessando vários serviços pelo IBConsole

Normalmente, para conectarmos a um banco IB/FB, utilizamos a *string* de conexão *Servidor:Caminho do banco de dados* (ex.:*localhost:c:\dados\MeuBanco.ib*), com isso fica subentendido que estamos usando o serviço *gds_db* na porta 3050, *default* do serviço de IB.

Usando múltiplas instâncias, sua aplicação deverá informar também qual o serviço deseja acessar. Veja a seguir como ficaria a *string* de conexão para o serviço *gds_db75*:

```
localhost/gds_db75:c:\dados\MeuBanco.ib
```

No IBConsole, não é diferente, devemos no momento do registro informar qual a porta que o serviço está sendo executado.

Nota: Somente podemos ter um serviço local registrado, logo, todas as outras instâncias deverão ser configuradas como remotas, sendo que no local da configuração do servidor, podemos digitar *localhost* ou o número do IP da máquina, por exemplo: 127.0.0.1.

Veja na **Figura 3** todos os serviços registrados e conectados ao mesmo tempo no IBConsole.

Conclusões

Com a *Multi-Instance* podemos realizar comparativos de performance e descobrir quem responde melhor a aplicação em um mesmo ambiente, fazer uma migração de forma gradual e com baixo impacto na aplicação e até mesmo executar dois ambientes de maneira segura. Essa é só mais uma das várias características que vem fazendo o IB subir alguns degraus, no que diz respeito a aplicações de banco de dados relacional. Em futuros artigos vamos trazer mais informações sobre a última versão do IB, um abraço e até a próxima.

Mendo Leonel *(mendo@presence.com.br) é Bacharel em Ciência da Computação, com Pós Graduação em Eng. de Software. Trabalha com Delphi desde a versão 3, em projetos cliente/servidor, multicamadas e Web Services. Atualmente é gerente de TI da Presence Tecnologia, empresa parceira da Borland e responsável pela comercialização e divulgação do InterBase no Brasil.*

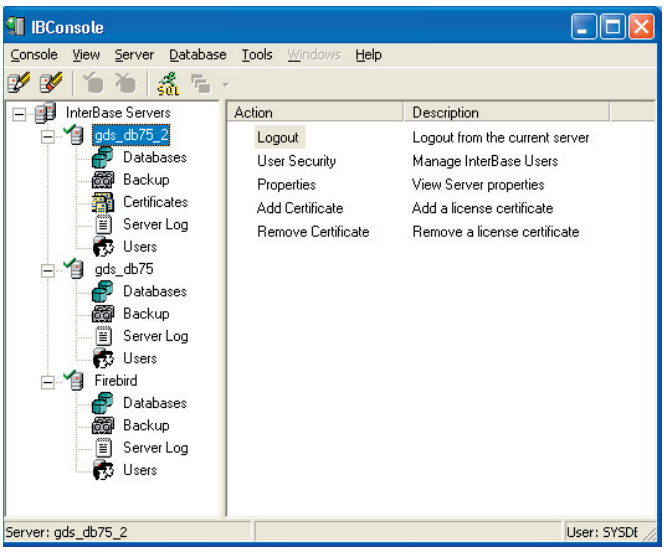


Figura 3. Várias instâncias sendo executadas ao mesmo tempo

Port e Instance Name

Como todo serviço que roda no protocolo TCP/IP, o IB tem uma porta onde ele fica "escutando" requisições clientes, enviando as devidas respostas. Por *default*, o IB utiliza a porta 3050. Esse número pode ser alterado, contudo é imprescindível que tenhamos certeza que nenhuma outra aplicação está usando o número que pretendemos atribuir ao serviço.

Instance Name ou *Service Name* é o nome que identifica o serviço. O exemplo a seguir é um trecho do arquivo *services* que pode ser encontrado em *<Diretório de sistema>\Drivers\etc*, nesse arquivo está identificado o nome do serviço, a porta/tipo e um comentário que representa o serviço, se necessário podemos alterar as configurações manualmente nesse arquivo.

```
gds_db          3050/tcp            #Firebird
gds_db75        3051/tcp            #Borland InterBase Server
gds_db75_2      3052/tcp            #Borland InterBase Server
```

Lembre-se que para cada instância do IB é necessário fazer um registro com uma licença válida, não é possível ter duas instâncias com a mesma licença rodando ao mesmo tempo.

Structured Query Language

por **Fábio Sarturi Prass e Fernando Sarturi Prass**

Guia de referência
Parte II: Comandos DML

Em continuação ao artigo anterior (edição 63), onde foram apresentados os comandos da DDL (*Data Definition Language*), mostraremos agora os comandos da DML (*Data Manipulation Language* ou Linguagem de Manipulação de Dados). Tais comandos abrangem as principais operações executadas no banco de dados: inclusão, edição, exclusão e consulta.

O comando Insert

A inclusão de dados se dá através do comando *Insert*. A sintaxe desse comando é bastante simples, basta informar a tabela, os campos e os seus respectivos valores:

```
INSERT INTO <nome da tabela>
[(<coluna> [, <coluna>, ...)]]
VALUES
(<valor> [,<valor> , ...] | <comando de seleção>)
```

Para incluir um registro na tabela *Aluno*, por exemplo, utilize o código:

```
INSERT INTO ALUNO (ID_ALUNO, NOME)
  VALUES (1, 'Fabio Sarturi')
```

Como podemos notar, nem todos os campos da tabela *Aluno* estão no código anterior (faltaram os campos *Data_Nasc*, *Telefone* e *Email*). Esses campos puderam ser suprimidos porque não são obrigatórios (restrição *Not Null*), ou seja, se um campo é obrigatório ele necessariamente deve estar presente no *Insert*, claro, do contrário o comando falhará.

A exceção ocorre quando formos preencher todos os campos da tabela. Nesse caso, podemos simplesmente suprimir o nome de todas as colunas, tomando apenas o cuidado de colocar todos os valores na mesma ordem em que a tabela foi criada:

```
INSERT INTO ALUNO VALUES (2, 'Jose da Silva',
  '02/02/1980','48 2335566','silva@provedor.com');
```

A inclusão de dados pode ainda ser feita via comando de seleção (*Select*). A vantagem desse recurso é que podemos facilmente migrar dados de uma tabela para outra, veja um exemplo de utilização:

```
INSERT INTO tabelaX
(C1, C2)
(SELECT C3, C4 FROM tabelaY)
```

Evidentemente, o tipo de dado do campo C1 deve ser compatível com tipo de dado do C3, assim como os dos campos C2 e C4. Além disso, é preciso tomar cuidado para que os valores não causem duplicidade em algum campo que contenha a restrição *Unique*, como chaves-primárias. Note que, quando incluímos dados através de um comando *Select* (estudaremos ele ainda neste artigo), suprimimos a palavra *Values* da instrução.

Registros das tabelas
Para fazer os exemplos que serão apresentados neste artigo, inclua na tabela *Aluno* os registros da **Tabela 1**, na tabela *Curso* os registros: 1- Delphi, 2- Firebird, 3- Java e na tabela *Aluno_Curso* os registros da **Tabela 2**.

Id_Aluno	*Nome*
1	Fabio Sarturi Prass
2	Jose da Silva
3	Alice dos Santos
4	Pedro Paulo Junior
5	Paloma Rosana Duarte
6	Joana Augusto Soares
7	Fernando Sarturi Prass

Tabela 1. Dados a serem incluídos na tabela Aluno

Id_Aluno_Curso	*Nr_Turma*	*Nota1*	*Nota2*	*Nr_Faltas*	*Id_Aluno*	*Id_Curso*
1	1	8.0	7.0	3	1	1
2	1	7.8	8.2	4	2	1
3	2	8.5		2	3	1
4	1	6.3	9.5	0	5	2
5	1	7.9	8.2	1	6	2
6	2	9.5		0	7	1
7	2	7.9	8.4	1	1	2
8	2	9.5		0	7	2

Tabela 2. Dados a serem incluídos na tabela Aluno_Curso

O comando Update

O *Update* permite a atualização de dados, sua sintaxe é:

```
UPDATE <nome da tabela>
SET <coluna> = <novo valor>
 [, <coluna> = <novo valor>, ...]
[WHERE <condição>]
```

Por exemplo, para alterar o nome *Fabio Sarturi*, que foi incluído sem o último sobrenome, basta fazer:

```
UPDATE ALUNO
SET NOME = 'Fabio Sarturi Prass'
WHERE ID_ALUNO = 1
```

O *Update* também permite a atualização simultânea de vários campos:

```
UPDATE ALUNO
SET NOME = 'Fabio Sarturi Prass',
  EMAIL = 'fsprass@yahoo.com.br'
WHERE ID_ALUNO = 1
```

É preciso tomar cuidado com a cláusula *where*, ela não é obrigatória, porém se não for utilizada, todos os registros da tabela serão alterados, obviamente. Entretanto, mesmo com a existência do *where*, é preciso tomar cuidado. Se no código anterior, trocarmos a condição "*Id_Aluno = 1*" para "*Id_Aluno >= 1*", todos os registros serão alterados, claro, uma vez que não temos nenhum valor menor do que 1 no campo *Id_Aluno*.

O comando Delete

O *Delete* permite apagar registros de tabelas. A sintaxe do comando é mostrada a seguir:

```
DELETE FROM <nome da tabela>
[WHERE <condição>]
```

Vale aqui a mesma observação feita no comando *Update*, se a cláusula *where* for suprimida, todos os registros da tabela serão apagados.

O código a seguir exclui o aluno cujo *Id* seja igual a 1:

```
DELETE FROM ALUNO WHERE ID_ALUNO = 1
```

Se o leitor tentar executar o comando apresentado, não obterá sucesso. Quando criamos a tabela *Aluno_Curso*, adicionamos a ela uma restrição de chave estrangeira para exclusão (*On Delete No Action* - ver o artigo na edição anterior). Para que pudéssemos excluir o aluno com o *Id* igual a 1, teríamos antes que excluir todas as linhas na tabela *Aluno_Curso* em que ele é referenciado (no entanto, não execute os comandos, apenas considere como exemplo):

```
DELETE FROM ALUNO_CURSO WHERE ID_ALUNO = 1;
DELETE FROM ALUNO WHERE ID_ALUNO = 1;
```

Se a restrição de exclusão não tivesse sido adicionada, ao excluir o aluno com *Id* igual a 1, ficaríamos com registros na tabela *Aluno_Curso* de um aluno que não existe mais, ou seja, teríamos uma inconsistência no banco de dados.

Ainda sobre a restrição de chave-estrangeira, se tivéssemos utilizado a opção *On Delete Cascade* na tabela *Aluno_Curso*, ao excluir o aluno da tabela *Aluno*, todos os registros da tabela *Aluno_Curso* cujo *Id_Aluno* for igual a 1 também seriam excluídos.

Comando Select

Sem dúvida esse é o principal comando da linguagem SQL. O *Select* permite selecionar registros no banco de dados. Sua sintaxe básica é a seguinte:

```
SELECT * | <coluna> [, <coluna> …]
FROM <tabela> [, <tabela> …]
[WHERE <condição>]
[ORDER BY <coluna> [, <coluna> …]]
```

Select identifica os campos que serão mostrados, *From* identifica a lista de tabelas que serão consultadas, *Where* a condição para que o registro seja selecionado e, por último, *Order By* indica a ordenação em que o resultado será mostrado.

Segue um exemplo simples de utilização do comando *Select*:

```
SELECT * FROM ALUNO_CURSO
```

Nota sobre o uso do asterisco (*) em consultas

O uso de expressões do tipo "*Select * From tabela*" deve ser sempre evitado. Toda vez que utilizamos o "*", estamos trazendo todos os campos da(s) tabela(s) da instrução de seleção. Isso pode gerar um tráfego de dados desnecessário na rede, tornando o sistema lento e gerando, certamente, reclamações por parte dos usuários.

O primeiro comando de consulta apresenta todos os registros da tabela *Aluno_Curso*. Se essa tabela fizesse parte de um sistema que controla os alunos de uma instituição de ensino do mundo real, teríamos na tabela um grande número de registros.

Dessa forma, para não gerarmos um tráfego de dados desnecessário, e também para que a nossa listagem possa ser lida com clareza, devemos adicionar algumas restrições nos comandos. Por exemplo: mostrar os alunos da Turma 1 (*Nr_Turma* = 1), do curso de Delphi (*Id_Curso* = 1), ordenado pelo nome do aluno:

```
SELECT ID_ALUNO, NOTA1, NOTA2
FROM ALUNO_CURSO
WHERE
  NR_TURMA = 1 AND
  ID_CURSO = 1
ORDER BY ID_ALUNO
```

Se o leitor executar o comando apresentado, terá como resultado duas linhas (1; 8.0; 7.0 e 2; 7.8; 8.2). A leitura desses dados é difícil, uma vez que apenas o número do aluno foi apresentado e não o seu nome. Para mostrar o nome do aluno, necessitamos fazer uma junção (um *Join*) entre as tabelas *Aluno* e *Aluno_Curso*. A forma mais simples de fazer isso é utilizar o seguinte código:

```
SELECT A.NOME, AC.NOTA1, AC.NOTA2
FROM ALUNO_CURSO AC, ALUNO A
```

```
WHERE
  AC.ID_ALUNO = A.ID_ALUNO AND
  AC.NR_TURMA = 1 AND
  AC.ID_CURSO = 1
ORDER BY A.NOME
```

Note que em *From* foi acrescentada a tabela *Aluno*. Sempre que estivermos selecionando dados em mais de uma tabela, devemos dar um *alias* (ou apelido) a cada uma delas, como os usados no código anterior: "AC" para *Aluno_Curso* e "A" para *Aluno*. O uso de *alias* não é obrigatório, entretanto se não tivéssemos utilizado o mesmo, seria necessário escrever o nome da tabela antes de cada um dos campos, cujo nome existe em mais de uma tabela.

Nesse caso, sempre que *Id_Aluno* for usado, seria necessário fazer algo como *Aluno_Curso.Id_Aluno* ou *Aluno.Id_Aluno*, o que torna o código maior e menos legível. Se no *From* temos mais de uma tabela, devemos identificar na cláusula *where* como essas tabelas estão relacionadas, em outras palavras, precisamos informar como se dá a "junção" entre ambas (daqui para frente usaremos o termo *Join*).

O *Join* entre tabelas é feito através da chave-estrangeira (*Foreign Key*). Nesse caso, o campo *Id_Aluno* é a referência, então devemos informar ao banco de dados que sempre que *Id_Aluno* aparecer na tabela *Aluno_Curso*, queremos que seja apresentado o registro correspondente na tabela *Aluno*, isso é feito da seguinte forma: *AC.Id_Aluno = A.Id_Aluno*.

Se não especificarmos como as tabelas são relacionadas, o banco de dados trará como resultado o produto cartesiano das tabelas envolvidas (no caso *Aluno_Curso* x *Aluno*). Exclua do comando a expressão *AC.Id_Aluno = A.Id_Aluno and* e execute o mesmo. Ao invés de apenas dois registros, retornarão 14, ou seja, o produto cartesiano da tabela *Aluno* (sete registros) com a tabela *Aluno_Curso* onde *Nr_Turma* e *Id_Curso* forem igual a 1 (dois registros).

É possível ainda fazer a junção de tabelas através do comando *Inner Join* conforme veremos no próximo artigo.

Funções

Bons bancos de dados trazem consigo uma série de funções que ajudam na hora de selecionar registros. Como queremos atender a todos os leitores, apresentaremos apenas as funções mais usuais (**Tabela 3**), que estão presentes em todos os bancos de dados. Os exemplos de utilização estão na **Listagem 1**.

Função	Descrição
LOWER	Transforma todos os caracteres em minúsculos.
UPPER	Transforma todos os caracteres em maiúsculos.
SUM	Soma todos os valores da coluna.
MAX	Maior valor existente na coluna.
MIN	Menor valor existente na coluna.
AVG	Média dos valores da coluna.
COUNT(coluna)	Conta o número de registros.

Tabela 3. Lista de funções para colunas

Listagem 1. Exemplos de utilização de funções

```
Seleciona a maior, a menor e a média da primeira nota
SELECT MAX(NOTA1), MIN(NOTA1), AVG(NOTA1)
FROM ALUNO_CURSO

Conta quantos alunos não tiveram a segunda nota informada
SELECT COUNT(*) FROM ALUNO_CURSO
WHERE NOTA2 IS NULL

Lista todos os alunos, mostrando o nome em letras maiúsculas
SELECT UPPER(NOME) FROM ALUNO
```

Operadores Lógicos

Além dos operadores lógicos mais conhecidos: igual a (=), maior que (>), maior ou igual que (>=), menor que (<) e menor ou igual que (<=), existem outros que podem ser usados para restringir os dados na cláusula *where*, os mais usados estão na **Tabela 4**.

Operador	Descrição
IS NULL	Verifica se o valor da coluna é nulo.
BETWEEN	Verifica se o valor está entre dois outros.
IN	Verifica se o valor pertence a um conjunto de valores.
LIKE	Pesquisa parte de uma *string*.

Tabela 4. Lista de operadores lógicos

Todos os operadores listados na tabela podem ainda ser combinados com o operador *Not*, que nega o resultado da condição. Por exemplo, *Nota2 Is Null* traria somente os registros onde a coluna *Nota2* não estivesse preenchida, *Nota2 Is Not Null* faria o contrário. Exemplo:

```
SELECT A.NOME
FROM ALUNO_CURSO AC, ALUNO A
WHERE
  AC.ID_ALUNO = A.ID_ALUNO AND
  AC.NOTA2 IS NULL
```

Operador IN

Verifica se o valor de uma coluna está presente em um conjunto de valores. Para selecionar os alunos que fizeram o curso de Delphi ou de Firebird, por exemplo, usaríamos o seguinte código:

```
SELECT A.NOME
FROM ALUNO_CURSO AC, ALUNO A
WHERE
  AC.ID_ALUNO = A.ID_ALUNO AND
  AC.ID_CURSO IN (1,2)
```

Operador Between

Seleciona apenas os registros cujos valores estão entre dois outros indicados. Para recuperar todos os alunos cujo número de faltas esteja entre 2 e 4, por exemplo, utilizaríamos o seguinte código:

```
SELECT A.NOME, AC.NR_FALTAS
FROM ALUNO_CURSO AC, ALUNO A
WHERE
  AC.ID_ALUNO = A.ID_ALUNO AND
  AC.NR_FALTAS BETWEEN 2 AND 4
```

Operador Like

Permite a busca por parte de uma *string*. Possui um papel muito importante, já que é comum o usuário buscar um texto sabendo apenas parte dele. Com o *Like* é possível localizar um texto independentemente da posição que se encontra, isso é feito com o uso de dois caracteres especiais:

- "%" (*percent*): chamado de coringa, representa qualquer quantidade de caracteres. Significa que admite em seu lugar um caractere, um conjunto de caracteres ou nenhum caractere;
- "_" (*underline*): Indica que pode existir qualquer caractere na posição desejada.

A **Listagem 2** mostra uma série de exemplos do uso do Like e "%". Vale lembrar que, embora o operador *Like* possua recursos extremamente avançados e úteis, seu uso deve ser controlado, pois se ele for usado no começo ou no meio da expressão o gerenciador do banco de dados não utilizará índice para realizar a pesquisa, o que pode tornar a mesma demorada quando a tabela possuir um número muito grande de registros.

Conclusões

Começamos a ver neste artigo os comandos que fazem parte da DML, foram apresentados os comandos *Insert*, *Update* e *Delete*. Também apresentamos o comando *Select*, entretanto, por questões de espaço, não é possível mostrar em um único artigo tudo o que ele pode fazer. Na próxima edição veremos outros comandos de seleção avançados, como: *Group By*, *Having* e o uso de *Subselects*. Além disso, conforme já comentado, mostraremos como realizar a junção de tabelas com o comando *Inner Join*. Até a próxima edição!

Listagem 2. Exemplos de utilização do operador LIKE

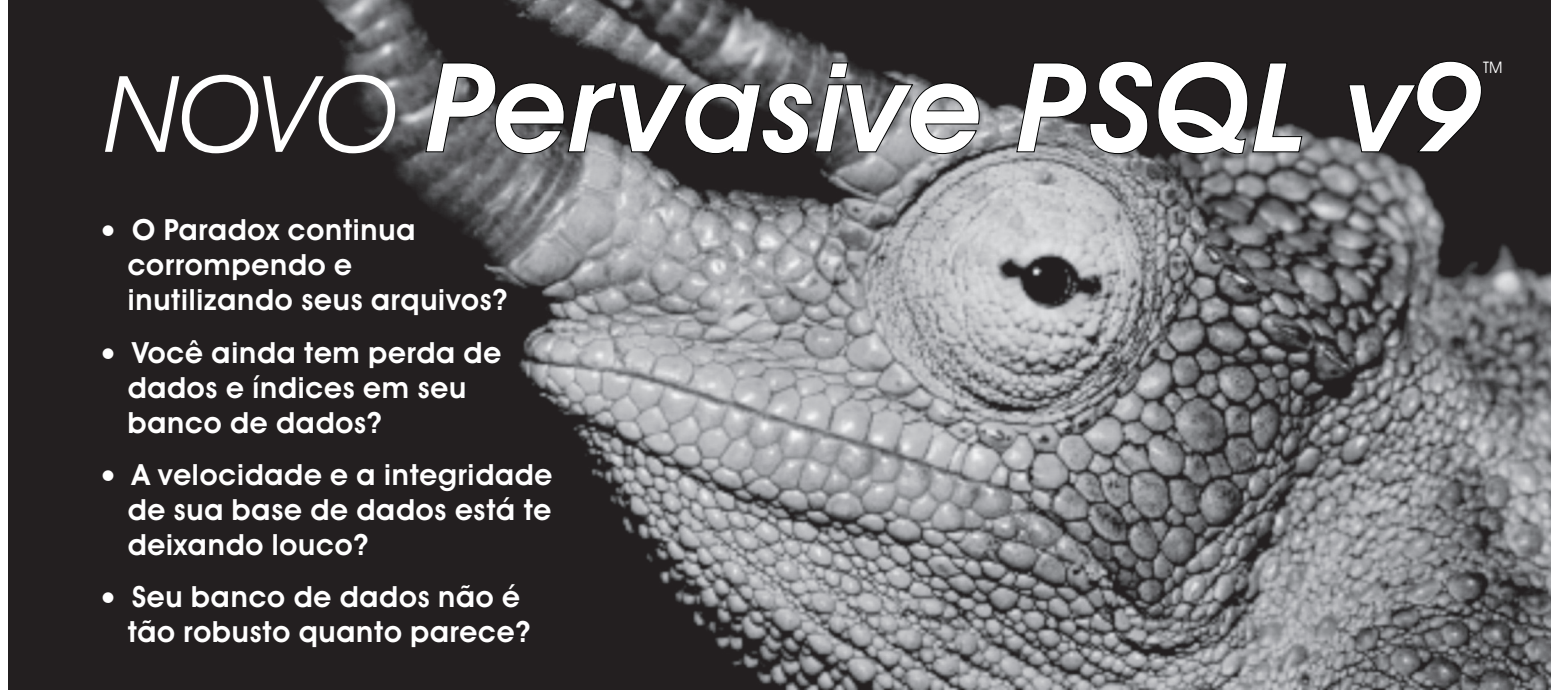
```
Alunos que contenham no nome a palavra 'SARTURI' em qualquer
parte do nome
SELECT NOME FROM ALUNO
WHERE UPPER(NOME) LIKE '%SARTURI%'

Alunos cujo nome não comece com a letra 'P'
SELECT NOME FROM ALUNO
WHERE UPPER(NOME) NOT LIKE 'P%'

Alunos cujos nomes terminam com a palavra 'SANTOS'
SELECT NOME FROM ALUNO
WHERE UPPER(NOME) LIKE '%SANTOS'
```

Fábio Sarturi Prass *(fsprass@yahoo.com.br) é Bacharel em Sistemas de Informação. Trabalha na FEESC - Fundação de Ensino e Engenharia de Santa Catarina (www.feesc.ufsc.br), onde é encarregado do desenvolvimento PL/SQL do Sistema de Arrecadação Tributária (SIATU) das Prefeituras de Belo Horizonte e Blumenau.*

Fernando Sarturi Prass *(fprass@yahoo.com.br) é Mestre em Ciência da Computação pela UFSC, Analista de Sistema da FEESC, professor da Faculdade Estácio de Sá de Santa Catarina (www.sc.estacio.br) e das Faculdades Integradas UNIVEST (www.sle.br).*

Download:
www.devmedia.com.br/clubedelphi/downloads

NOVO Pervasive PSQL v9™



- O Paradox continua corrompendo e inutilizando seus arquivos?
- Você ainda tem perda de dados e índices em seu banco de dados?
- A velocidade e a integridade de sua base de dados está te deixando louco?
- Seu banco de dados não é tão robusto quanto parece?

Migre para o novo Pervasive PSQL v9™

Totalmente compatível com o modelo orientado a registros do Paradox, além de ser 100% ANSI SQL. Com os componentes nativos 100% compatíveis com o TTable/TQuery, você não precisa reescrever a sua aplicação. Além de solucionar diversos problemas, você terá os seguintes benefícios:

- Não haverá mais perda de dados ou corrupção de índices/tabelas, instale e esqueça!
- Alto desempenho na rede que não degrada com o aumento do tamanho das tabelas ou número de usuários
- Acessa o banco de dados transacional/relacional sem a necessidade de mapear um diretório
- Não vai mais precisar instalar e configurar o BDE/dbExpress nas estações
- Reduz drasticamente o suporte ao cliente, pois o banco de dados é totalmente isento de manutenção

Vantagens

- Projetado e desenvolvido baseado nas necessidades e restrições das pequenas e médias empresas
- Maior economia para o desenvolvedor e para o usuário, devido ao seu design integrado, ser auto-configurável e auto-ajustável, o Pervasive PSQL v9™ dispensa a administração minimizando os custos de suporte e maximizando a satisfação dos clientes
- Compatível com padrões do mercado. Inclui drivers ODBC, OLE/DB, JDBC, .NET e componentes/interfaces de acesso para Delphi, C++ Builder, C/C++, VB e COBOL
- Rápido e confiável em sistemas operacionais Linux, NetWare e Windows
- Escalável desde um único usuário até ambiente cliente/servidor sem alterações no aplicativo

Novidades

- Performance relacional e transacional incomparável
- Novas funcionalidades e sintaxes de SQL
- Componentes nativos para Delphi (Table e Query)
- Suporte nativo a COBOL (Occurs e Redefines)
- Suporte a arquivos de 128 GB sem extensões
- Novos utilitários gráficos e de linha de comando
- Funções definidas pelo usuário

Soluções de segurança para o Pervasive PSQL v9™

Pervasive DataExchange™

Replicação e sincronização de dados para o Pervasive PSQL v9™

Pervasive AuditMaster™

Auditoria e rastreamento de modificações dos dados do Pervasive PSQL v9™

PERVASIVE® • **www.stern.com.br • (11) 3078-1690 • info@stern.com.br**

Crystal Reports

por **Marcelo Pereira Rocha e Adail Muniz Retamal**

Utilizando arquivos XSD para definição de relatórios no Delphi 2005

Os desenvolvedores que optaram pelo Crystal Reports como ferramenta para criação de relatórios talvez já tenham se deparado com uma situação incômoda: a necessidade de que o relatório abra uma conexão com o banco de dados para que possa ser elaborado ou executado.

Neste artigo apresentaremos uma maneira simples para elaborar relatórios no Crystal Reports com total independência da camada de persistência e sem permitir que sejam feitas conexões diretas à base de dados por meio do relatório. Usaremos para isso arquivos XSD.

O que é XSD?

O XSD (*XML Schema Definition*) é um meio para definir a estrutura, conteúdo e semântica de documentos XML. Ele é especificado pela *World Wide Web Consortium* (W3C). No exemplo deste artigo iremos utilizá-lo para definir quais campos serão exibidos no relatório.

O ambiente de dados

Antes de construirmos a aplicação exemplo, vamos definir o que deverá ser exibido pelo relatório. O exemplo é bem simples: uma tabela de clientes contendo apenas código e nome, sendo classificada por um tipo de cliente. Crie uma base de dados no InterBase e as tabelas conforme a **Listagem 1**.

Listagem 1. Criando as tabelas para o exemplo de relatório

```
CREATE TABLE TIPOSCLIENTE (
  CODIGO INTEGER NOT NULL,
  DESCRICAO VARCHAR(40) NOT NULL,
  CONSTRAINT PK_TIPOS_CLIENTE PRIMARY KEY (CODIGO));

CREATE TABLE CLIENTES (
  CODIGO INTEGER NOT NULL,
  NOME VARCHAR(40) NOT NULL,
  CODIGOTIPO INTEGER NOT NULL,
  CONSTRAINT PK_CLIENTES PRIMARY KEY (CODIGO),
  CONSTRAINT FK_TIPOS_CLIENTE
  FOREIGN KEY (CODIGOTIPO) REFERENCES TIPOSCLIENTE);
```

Uma vez definido o que podemos ler, vamos pensar no que queremos exibir no relatório. Primeiramente, adicione alguns valores às tabelas. O máximo que podemos fazer aqui é emitir uma lista de clientes classificados pelos tipos aos quais pertencem. É o que iremos fazer usando o seguinte SQL:

```
SELECT C.CODIGO, C.NOME, T.DESCRICAO
FROM CLIENTES C, TIPOSCLIENTE T
WHERE C.CODIGOTIPO = T.CODIGO
ORDER BY C.NOME
```

Agora já podemos pensar em criar um arquivo XSD que representa um resultset dessa sentença. Vamos chamá-lo de "DataSetCliente.xsd", conforme a **Listagem 2**.

Listagem 2. Criando o arquivo XSD

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="DataSetCliente"
targetNamespace="DataSetCliente.xsd" xmlns:
mstns="DataSetCliente.xsd"

xmlns="DataSetCliente.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:
msdata="urn:schemas-microsoft-com:xml-msdata"

attributeFormDefault="qualified" elementFormDefault=
  "qualified">
  <xs:element name="DataSetCliente"msdata:IsDataSet="true"
     msdata:Locale="pt-BR">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Table">
                <xs:complexType>
                      <xs:sequence>
                        <xs:element name="CODIGO"
                          type="xs:decimal" />
                        <xs:element name="NOME"
                          type="xs:string" />
                        <xs:element name="DESCRICAO"
                          type="xs:string" />
                      </xs:sequence>
                </xs:complexType>
              </xs:element>
        </xs:choice>
      </xs:complexType>
      <xs:unique name="Constraint1" msdata:
         PrimaryKey="true">
        <xs:selector xpath=".//mstns:Table" />
        <xs:field xpath="mstns:CODIGO" />
        <xs:field xpath="mstns:NOME" />
        <xs:field xpath="mstns:DESCRICAO" />
      </xs:unique>
  </xs:element>
</xs:schema>
```

Nota: Você pode usar o *Bloco de Notas* do Windows para criar o arquivo ou qualquer editor de XML.

A aplicação

O exemplo que criaremos, mostra como definir o arquivo XSD, elaborar e publicar um relatório em ambiente Web. O primeiro passo para isso é criar uma aplicação ASP.NET (*File|New>ASP.NET Web Application – Delphi for .NET*). Vamos chamá-la de "MinhaAplicacaoRelatWeb" e para o servidor Web escolha o de sua preferência (IIS ou Cassini).

Note que automaticamente um arquivo ASPX é criado com o nome padrão *WebForm1*. É nesse ASPX que iremos incluir um controle para visualização do relatório. Mas antes é necessário criar o relatório. Para isso, basta usar a opção do menu *File|New>Other>Crystal Reports>Report*.

No assistente de criação, escolha a opção *As a blank Report*, assim não utilizaremos o assistente, iremos criar a conexão e adicionar os campos manualmente. Para criar a conexão, basta clicar no item *Database Fields* do painel direto do relatório e escolher a opção *Add/Remove Database*.

Aparecerá a janela *Database Expert*, onde iremos indicar o tipo de conexão do relatório, neste exemplo um arquivo XSD. Clique no item *More Data Sources|ADO.NET (XML)* e será aberta uma janela para a escolha do arquivo XSD (**Figura 1**). Selecione o item *Table* (que está contido em *DataSetCliente*) e dê um duplo clique no item para enviá-lo para a lista de tabelas (*Selected Tables*). Para concluir, clique em OK.

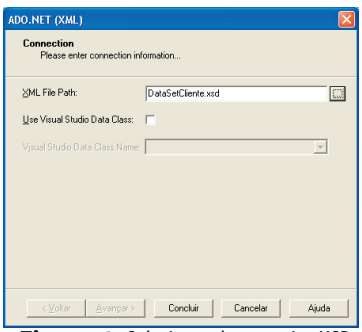


Figura 1. Selecionando o arquivo XSD

Já com todos os campos carregados (*Database Fields*), agora é só estruturar o relatório arrastando cada um dos campos para a seção *Detail*. Isso criará um título correspondente na seção *Page Header*. Uma vez definido o relatório, vamos adicionar o controle de visualização do Crystal (*CrystalReportViewer*, encontrado na categoria *Crystal Report*) no *WebForm1*. Após, adicione a unit *Report1 Unit* (criada para representar o *Report1.rpt*) na cláusula *uses* do *WebForm1.pas*.

Utilizando o método SetDataSource

Se todo o relatório está definido baseado em uma abstração do *Resultset*, então de alguma forma precisamos fazer com que o relatório conheça esse *Resultset*. Para isso utilizamos o método *SetDataSource*, cujo único parâmetro é um *DataSet* (não confundir com *TDataSet*, pois não estamos utilizando VCL).

Adicione no evento *Load* o código da **Listagem 3**. Declare em *uses* o namespace *Borland.Data.Provider*. Rode a aplicação e visualize o relatório. É claro que estamos criando tudo dinamicamente, apenas porque se trata de um exemplo. Você pode obter seu *DataSet* de onde for mais conveniente para a situação.

Listagem 3. Código para enviar os dados de uma consulta para o arquivo XSD

```
var
  connection: BdpConnection;
  adapter: BdpDataAdapter;
  cmd: BdpCommand;
  ds: DataSet;
  strConnection: string;
  myreport: Report1;
begin
  strConnection := 'database=<caminho banco>;'+
    'assembly=Borland.Data.Interbase, '+
    'Version=2.0.0.0, Culture=neutral,'+
    'PublicKeyToken=91d62ebb5b0d1b1b;'+
    'vendorclient=gds32.dll;provider=Interbase;'+
    'username=sysdba;password=masterkey';
  ds := DataSet.Create;
  adapter := BdpDataAdapter.Create;
  connection := BdpConnection.Create(strConnection);
  cmd := connection.CreateCommand;
  if connection.State = ConnectionState.Closed then
    connection.Open;
  cmd.CommandText := 'SELECT C.CODIGO, C.NOME, '+
    'T.DESCRICAO FROM CLIENTES C, TIPOSCLIENTE T '+
    'WHERE C.CODIGOTIPO = T.CODIGO ORDER BY C.NOME';
  adapter.SelectCommand := cmd;
  adapter.Fill(ds);
  if ds.Tables.Count > 0 then
    ds.Tables[0].TableName := 'Table';
  myreport := Report1.Create;
  myReport.SetDataSource(ds);
  CrystalReportViewer1.ReportSource := myreport;
end;
```

O mais importante é que não foi necessário passar informações de conexão para o relatório (tanto em *design-time* quanto em *run-time*). O relatório apenas precisa conhecer um *DataSet* e mais nada. Finalmente, ao executar a aplicação Web, teremos um relatório que ainda poderá ser exportado para PDF, DOC, XLS etc. (o Crystal Reports provê essa funcionalidade).

Links

www.borland.com.br/delphi
Site oficial do Delphi
www.businessobjects.com
Site oficial da Business Objects (empresa desenvolvedora do Crystal Reports)
www.w3.org
Site da W3C, que especifica a XSD

Marcelo Pereira Rocha *(mrocha@borland.com) é Borland Delphi 7 Certified Developer, Técnico em Informática Gerencial formado pelo Colégio Cotemig e Tecnólogo em Processamento de Dados formado pelo Centro Universitário da União de Negócios e Administração de Minas Gerais (UNA-MG). Atualmente desenvolve sistemas Web em .NET voltados para a área governamental. Atua como consultor para a Borland Latin America (Professional Services Organization).*

Adail Muniz Retamal *(aretamal@borland.com) é engenheiro eletrônico, desenvolvedor e instrutor certificado Delphi. Pesquisa e divulga a Engenharia de Software há muitos anos, incluindo metodologias de desenvolvimento e modelos de qualidade de software. Hoje é consultor, palestrante e engenheiro de sistemas da Borland Latin America.*

Download:
www.devmedia.com.br/clubedelphi/downloads

MMLogs

por **Ricardo Balbinot, Fladhymir C. Castello, Daniel K. Silva e Ricado do Amaral Soares**

Uma biblioteca para auxílio à depuração de falhas

Um dos maiores problemas na criação de sistemas é o monitoramento das circunstâncias de operação dos mesmos e o modo de reportar erros, permitindo uma depuração adequada dos problemas que, de outra maneira, não seriam de conhecimento do desenvolvedor ou chegariam com um relatório apenas parcial do ocorrido.

Um dos mecanismos mais utilizados para esse fim são os ditos *logs* de operação, que visam permitir uma visualização de problemas ou circunstâncias de operação de um sistema. Os *logs* são usados para verificar o que ocorreu com um sistema. Obviamente existem outros mecanismos de monitoração de erros em um sistema (levantamento dos valores de registradores do processador etc.).

A idéia deste artigo é apresentar uma biblioteca voltada ao processamento e detalhamento de *logs* de operação de sistemas, desenvolvida pelo GPARC-TI (Grupo Multimídia) com o intuito de permitir o uso facilitado desse recurso. A biblioteca é denominada *MMLogs*, contando na atualidade com mais de 20 classes distintas que fornecem capacidades que vão da geração ao armazenamento desses registros.

Sendo um artigo introdutório, iremos apresentar algumas funcionalidades básicas da biblioteca e montaremos um exemplo mostrando como as exceções em um sistema podem ser monitoradas e tratadas de forma adequada, permitindo seu registro e futura correção.

A idéia da biblioteca

Sua concepção básica envolve o uso de um "caminho de dados" (denominado *Log Path*), onde as mensagens de *log* podem trafegar, tomando diferentes ações a depender das classes e/ou componentes pelos quais passam.

Assim sendo, a operação básica da biblioteca envolve a criação de um caminho para as mensagens de *log* que gere uma reação mais adequada para cada tipo de mensagem, como por exemplo, registro em arquivo, envio pela rede, envio por e-mail etc.

Download

O GPARC-TI (*Grupo de Pesquisas Avançadas em Redes de Comunicação e Tecnologia da Informação*), presente dentro da PUC-RS, é um grupo acadêmico e seu objetivo primário é a pesquisa e desenvolvimento nas áreas de redes de comunicação e tecnologia da informação, não objetivando lucro, atuando em conjunto com o governo (FINEP e CNPq) e empresas em projetos de interesse comum, que visem o desenvolvimento e capacitação tecnológica brasileira.

Muitos dos projetos desenvolvidos pelo grupo não são acessíveis ao público em geral de uma maneira simplificada, mas o *MMLogs* pode ser obtido facilmente através do site *mmlogs.gparc.org*, sendo necessário realizar um cadastro. Você receberá uma confirmação e uma senha, por e-mail, que lhe dará acesso ao download da biblioteca.

Instalação

Para que a biblioteca seja instalada e funcione corretamente, você terá que ter instalado anteriormente as bibliotecas INDY na versão 9.011 (não foram observados problemas com o uso de versões superiores), JVCL e JCL, na versão 1.22 e 2.0, respectivamente (para versões superiores da JVCL e JCL existem alterações nos nomes das classes que podem causar problemas).

A instalação da biblioteca pode ser realizada de forma simplificada pelo uso do instalador obtido através do download. A biblioteca *MMLogs* foi desenvolvida para uso nas versões 6 e 7 do Delphi, usuários de versões anteriores, a instalação fica por sua conta e risco. Automaticamente a biblioteca detectará entre as versões 6 e 7 do Delphi, instaladas na máquina do usuário. Após esse procedimento, a biblioteca será instalada na máquina e pronta para uso.

Utilização

No Delphi serão acrescentadas as paletas *MMLogs*, *MMLogsUtilities* e *MMLogFilters* (**Figura 1**). O uso mais correto para a biblioteca de *logs*, com relação a um sistema como um todo, parte da utilização de um ou mais objetos do *MMLogSystem*, muito embora seu uso não seja absolutamente obrigatório. A classe *TMMLogSystem* define mecanismos globais e padronizados para a geração de registros de *logs*.

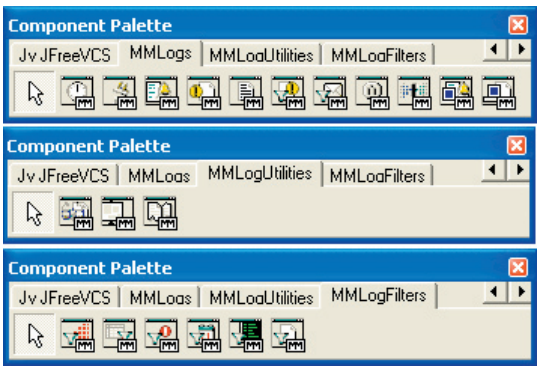


Figura 1. Componentes MMLogs instalados no Delphi

Isso é importante para a manutenção da consistência do processo de *logs*, visto que o conteúdo das mensagens, como os códigos de severidade, tipos e outras informações, são de abrangência local no que se referem as demais classes existentes na biblioteca. Agregado à biblioteca existe uma ferramenta bastante simples, o *Log Definitions Editor* (**Figura 2**), que encontra-se na pasta *Utilities\LogDefinitions*, permitindo a padronização das mensagens e geração de um arquivo de trabalho com o *TMMLogSystem*. O arquivo gerado pela ferramenta será semelhante à **Listagem 1**, possuindo extensão INI.

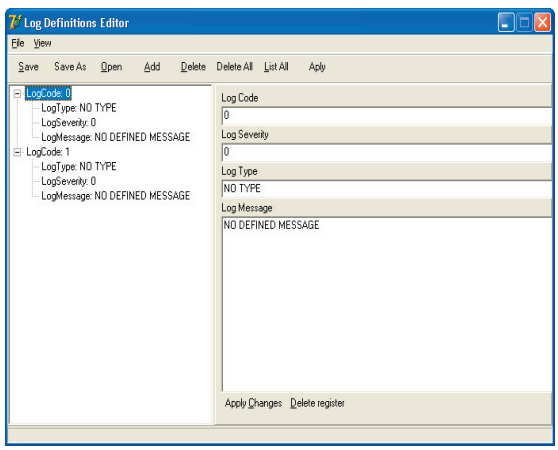


Figura 2. Tela do Log Definitions Editor

Listagem 1. Arquivo de Log gerado

```
[HEADER]
VERSION=1.0.1.0
NUMBEROFREG=2
REG0=5
REG1=10
[5]
LogType=INFO
LogSeverity=10
LogMessage=Nova conexão estabelecida.
[10]
LogType=SEVERE EXCEPTION
LogSeverity=255
LogMessage=Exceção fatal. Causada por.....
```

Outras funcionalidades presentes na biblioteca são:

- Monitoramento de exceções, possibilitando a geração automática de registros de operação na ocorrência de exceções. Com o uso adequado da JCL e JVCL, essa funcionalidade permite determinar, além da exceção, a linha de código e unidade onde a exceção foi gerada;
- Armazenamento em arquivos ou tabelas de um banco de dados;
- Envio via rede ou e-mail;
- Replicação de caminhos, permitindo o processamento de um mesmo registro por mais de um caminho;
- Filtros, que permitem que o processamento da mensagem de *log* seja condicional a certas situações de uso.

Classes de uso geral

Como o objetivo é apresentar uma visão geral da biblioteca, iremos destacar as principais funcionalidades de alguns componentes considerados de maior relevância e apenas comentaremos a existência de outros. Uma das funcionalidades mais interessantes da biblioteca é permitir o armazenamento dos registros gerados em um arquivo padronizado.

Isso é possível pela utilização de componentes descendentes da classe *TMMLogFile*, a qual especifica as propriedades e métodos fundamentais para trabalho com o arquivo. Essa classe permite que o arquivo de *log* receba um número máximo de registros (*MaxRegisters*), o período máximo de abrangência do arquivo (*MaxPeriod*) ou mesmo o tamanho máximo do arquivo (*MaxFileSize*).

Essas facilidades visam permitir que o desenvolvedor possa especificar como os arquivos devem ser trabalhados a fim de permitir uma recuperação facilitada do conteúdo dos mesmos posteriormente. O caminho onde os arquivos devem ser armazenados é especificado através da propriedade *LogFilePath*. Todos os arquivos gerados possuem no nome a data/hora de geração, além de um prefixo que é estabelecido através da propriedade *PrefixName*, permitindo a customização do nome do arquivo e a fácil identificação do período de abrangência do mesmo.

Junto ao diretório especificado é salvo um arquivo de configuração dos *logs*. Esse arquivo guarda uma série de informações que permite a posterior recuperação do status de operação do objeto quando ocorrer uma nova execução e utilização do mesmo por parte do sistema. É interessante, portanto, que dois sistemas não armazenem seus registros de *log* em um mesmo diretório, sob pena de ocorrerem perdas na seqüência de operação.

A única penalidade resultante da quebra dessa regra é que serão gerados múltiplos arquivos de *log* quando um mesmo ou mais de um sistema for executado diversas vezes.

Log de Exceptions

Uma outra funcionalidade da biblioteca é o monitoramento de exceções através de *logs*, o que pode ser feito com o uso de dois componentes diferentes, que são: *MMLogExceptions* e *MMLogFilteredExceptions*. Ambos oferecem técnicas que possibilitam o monitoramento de todas as exceções que ocorrem no sistema, permitindo (no caso da disponibilidade de um arquivo de descrição detalhado - arquivo *map* - do executável) inclusive, a indicação da unidade, método e linha onde ocorreu a exceção.

A classe de uso mais simples é a *TMMLogExceptions*, a qual simplesmente, através da propriedade *HookExceptions* (com valor *True*), permite o monitoramento das exceções no sistema. Como a classe não oferta nenhuma forma padronizada de tratamento do registro de *log*, é necessário encaminhar a mesma pelo *LogPath* apropriado, lançando o registro, por exemplo, em um e-mail.

A classe *TMMLogFilteredExceptions* adiciona mecanismos que permitem a verificação automática da classe da exceção, possibilitando que somente as exceções fora de um conjunto seleto (dito conjunto de restrição de exceções - *RestrainedExceptions*) sejam encaminha-

das normalmente no *LogPath* padrão. Os registros de *logs* também podem ser lançados via rede ou e-mail. No caso de operação via rede são utilizados dois componentes distintos: *MMLogClient* e *MMLogServer*.

O *MMLogClient* possibilita ao usuário estabelecer uma conexão TCP/IP com um servidor remoto *MMLogServer*. Ambos os componentes são implantados visando a utilização de um mesmo protocolo. Dessa forma é possível a monitoração em tempo real dos registros de operação de certo sistema (o que permitiria construir uma facilidade de *trace* remoto, o que, contudo, não é objetivo da biblioteca).

Log por e-mail

Para o envio de e-mail, à classe é a *MMLogMailDispatch*. Essa permite o envio de um registro de *log* para um endereço de e-mail pré-estabelecido. Para o modo de funcionamento padrão da classe a mesma deve se encontrar no *LogPath* desejado ou ser acionada através do método padrão *LogMessage*. A cada registro de *log* gerado é estabelecida a conexão com o servidor SMTP e então enviada a mensagem. Caso ocorram problemas com a conexão ou com a formatação da mensagem é levantada uma exceção correspondente. Obviamente, é necessário que o usuário possua uma conta de e-mail em um servidor apropriado que possa ser utilizado. Na pasta *examples/LogUse* do diretório de instalação do *MMLog* existe um exemplo com a utilização de e-mail, para o envio de *logs*.

Uma facilidade adicional fornecida pela biblioteca *MMLogs* é a criação de múltiplos e diferentes *LogPaths* que permitem a replicação de um registro de *log* por diferentes caminhos de processamento. Essa funcionalidade é provida pela classe *TMMLogSpliter*.

A *TMMLogSpliter* define não somente o caminho padrão expresso pela propriedade *LogPath*, mas também diversos caminhos distintos para classes herdeiras da *TMMLogBase*, os quais podem ser manipulados pelos métodos *AddAdditionalPath*, *ListAdditionalPaths* e *RemoveAdditionalPath*. A mensagem que chega no *MMLogSpliter* pelo método *LogMessage* é replicada para todos os caminhos especificados, começando pelo caminho estabelecido pela *LogPath* e indo pelos demais caminhos adicionados, um por um.

A biblioteca define um mecanismo de filtragem de registros de *log* a fim de permitir ao usuário especificar diferentes ações para diferentes registros, alterando o comportamento padrão do *LogPath*.

A classe base para utilização dos filtros é definida por *TMMLogBaseFilter*. Ela apenas fornece os mecanismos necessários para ser inserida no caminho de processamento dos *logs* e para ser associada com um objeto filtro, que é herdeiro da classe *TMMBaseFilter*.

São definidos na biblioteca os seguintes filtros (todos, herdeiros da *TMMBaseFilter*):

- *TMMRegCodeFilter*: filtragem pelo código do registro de *log*;
- *TMMRegDateFilter*: filtragem pela data do registro de *log*;
- *TMMRegNumberFilter*: filtragem pela numeração da mensagem de *log*;
- *TMMRegSeverityFilter*: filtragem através do campo de severidade;
- *TMMRegTypeFilter*: filtra as mensagens através da informação de tipo das mesmas.

Podemos também citar algumas das principais funcionalidades das seguintes classes da biblioteca:

- *TMMLogEvent*: simplesmente oferece o acionamento de um evento indicando a chegada de um novo registro de *log*;
- *TMMLogBaseCounter*: oferece a possibilidade de caracterizar um processo de contagem local e automática dos registros de *log* gerados no sistema;
- *TMMLogSysEvent*: oferece a funcionalidade de lançamento do registro de *log* no registro de eventos do sistema operacional (somente para sistemas Win32);
- *TMMLogStream*: oferece uma forma simplificada de lançar os registros de *log* que passam pelo seu *LogPath* em um *stream* associado (qualquer descendente de *TStream*);
- *TMMLogFileReader*: oferece um modo simplificado de leitura de um arquivo de *logs*;
- *TMMLogShow*: oferece um mecanismo simplificado de visualização dos *logs* em um *Grid*.

Exemplo básico

Para um exemplo simples, utilizamos os componentes *MMLogExxception* e *MMLogFile* de acordo com a **Figura 3**. Para o exemplo, indique na propriedade *LogPath* do *MMLogExceptions1* o *MMLogFile1* e altere para *True* a propriedade *HookExceptions*. No código a seguir, do evento *OnCreate* do formulário, levantamos uma exceção que será armazenada em um *log*:

```
procedure TFormLog.FormCreate(Sender: TObject);
begin
  MMLogFile1.Active := True;
  raise Exception.Create('Teste de Log');
end;
```

Na **Figura 4** temos o arquivo de *log* gerado. Na **Tabela 1** apresentamos o arquivo de *log* gerado pela exceção que criamos anteriormente e as informações logadas.

Links

www.gparc.org
Grupo de Pesquisas Avançadas em TI
mmlogs.gparc.org
Biblioteca MMLogs
www.indyproject.org
Biblioteca Indy
jvcl.sourceforge.net
versão JVCL e JCL

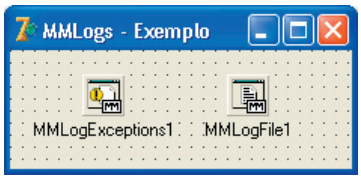


Figura 3. Componentes para o exemplo de criação de um log de Exception

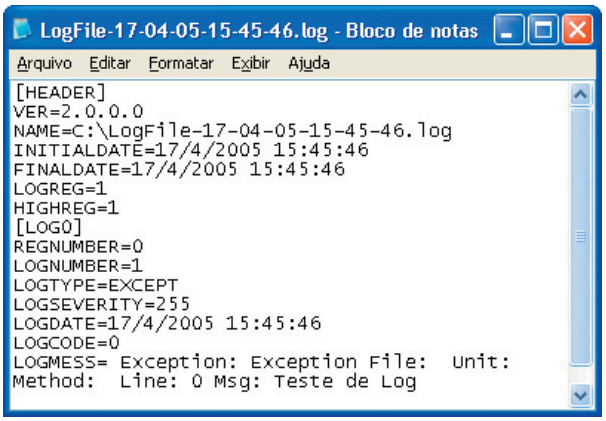


Figura 4. Arquivo de log gerado

[HEADER]	DESCRIÇÃO
VER=2.0.0.0	Deve indicar a versão
NAME= C:\LogFile-17-04-05-15-45-46.log	Nome e caminho do arquivo
INITIALDATE=17/4/2005 15:45:46	Data de criação do arquivo
FINALDATE=17/4/2005 15:45:46	Última operação com o arquivo
LOGREG=1	Número de registros no arquivo
HIGHREG=1	Mais alto registro no arquivo
[LOG0]	Seção de armazenagem de dados para o registro 0
REGNUMBER=0	Indica o número do registro (sistema de arquivo)
LOGNUMBER=1	Indica o número real do registro
LOGTYPE=EXCEPT	Indica o tipo do registro
LOGSEVERITY=225	Indica a severidade do registro
LOGDATE=17/4/2005 15:45:46	Indica a data e hora do registro
LOGCODE=0	Indica o código do registro
LOGMESS=Exception: Exception File: Unit: Method: Line: 0 Msg: Teste de Log	Indica a mensagem do registro

Tabela 1. Arquivo de Log com a informação gerada

Ricardo Balbinot *(rbalbinot@gparc.org) atua como engenheiro na área de telecomunicações (Vivo), além de ser docente na PUCRS desde 2002, tendo desenvolvido diversos aplicativos e classes para uso com o Delphi, com o qual trabalha desde 1996. É um dos pesquisadores responsáveis no projeto Locutus, além de ter envolvimento em diversos outros projetos do grupo GPARC-TI. É coordenador do grupo GPARC-TI.*

Fladhimyr Camara Castello *(fcastello@gparc.org) é Mestre em Engenharia Elétrica pela Pontifícia Universidade Católica do Rio Grande do Sul, formado em análise de sistemas pela UNISINOS. É coordenador do grupo de pesquisas Multimídia do GPARC-TI há mais de dois anos, atuando nas áreas de aplicações multimídia e redes de alta velocidade. Analista de sistemas da empresa Mobisol – Soluções em Tecnologia, desenvolvendo aplicações para dispositivos móveis (celulares, smartphone etc).*

Daniel Korndörfer Silva *(dsilva@gparc.org) é Graduando em Ciência da Computação na PUCRS. Trabalha com redes de computadores e programação desde 1997. Atualmente é pesquisador do GPARC-TI, sendo um dos principais desenvolvedores do projeto Locutus.*

Ricado do Amaral Soares *(rsoares@gparc.org) é Técnico em Informática, Técnico em Redes de Computadores e graduando em Engenharia de Controle e Automação na PUCRS. Atualmente atuando na pesquisas e desenvolvimento de novas tecnologias para redes de computadores no GPARC-TI.*

Aplicações Multicamadas

por **Gustavo Chaurais**

Migração simplificada de duas camadas para n-tier

Na medida em que a arte de desenvolver softwares cresce no mundo, a arquitetura multicamadas vem ganhando espaço em detrimento da arquitetura cliente/servidor. Com isso, muitas empresas investem em soluções grandes e complexas, mas acabam por esquecer o pequeno cliente que gostaria apenas de um "programinha" para cadastrar suas vendas.

Será que toda a sua clientela está disposta a pagar por isso? Vejamos como trabalhar com a portabilidade, visando manter apenas um único padrão para sistemas de duas ou n-camadas.

Escolhendo o caminho

Mais uma vez estamos vivenciando uma quebra do paradigma na construção de um software. O mundo está voltando os olhos para objetos distribuídos e arquiteturas orientadas a serviços. Nosso leque de opções para o desenvolvimento de um sistema cresce a cada dia e um bom gerente de projetos deve saber discriminar a tecnologia a ser empregada em cada caso.

A programação multicamadas também está inserida nesse contexto, no tocante a quando e como deve ser utilizada. O box *Vantagens da arquitetura* relembra alguns tópicos importantes na tomada de decisão.

A euforia de estarmos sempre inovando e usando as últimas novidades do mercado, pode acabar levando a alguns erros no planejamento de um software. Raramente adotaríamos uma arquitetura multicamadas para uma padaria ou uma farmácia. Para um projeto de grande porte, servidores de aplicações geralmente são aconselhados, pois seu custo é inferior ao da atualização periódica de todo o maquinário da empresa. Isso não acontece em pequenos projetos com poucas máquinas.

Existe ainda um outro conceito, chamado sensibilidade, que também deve ser levado em conta. Se inserirmos uma nova camada em nossa aplicação, certamente teremos uma queda de performance que, às vezes, é quase imperceptível. É preciso saber quanto o seu cliente suportaria de atraso no acesso às funcionalidades do sistema. Desse modo, poderemos levantar os dados necessários à escolha da infra-estrutura a ser utilizada.

Frameworks OO

A engenharia de software oferece uma eficiente solução para diversos problemas relacionados ao desenvolvimento de sistemas. Frameworks de aplicações Orientados a Objetos são conceituados como softwares inacabados, destinados a certo domínio do problema e possuem como fundamentos a abstração, herança, composição e modularização. Têm como resultados esperados: alta produtividade, maior qualidade do produto final, redução nos custos e maior manutenibilidade do software.

Para a construção de um framework OO, faz-se necessária uma equipe com conhecimentos avançados em diversos tópicos como a ferramenta em questão, o tipo de software a ser desenvolvido e principalmente Orientação a Objetos (abstração). Alguns conceitos específicos, ditados pela engenharia de software, também são indispensáveis, outros, como *design patterns*, melhoram substancialmente a qualidade do projeto.

Contudo, esse processo de construção geralmente não é realizado da melhor maneira. Muitas empresas não investem corretamente na construção do mesmo. Desse modo, não conseguimos o resultado esperado. Uma das soluções mais usadas nesse caso é a terceirização.

Visão mercadológica

Existem hoje muitas empresas que mantêm simultaneamente clientes com alto e baixo poder de investimento. Isso resulta em alguns produtos portadores da arquitetura cliente/servidor e outros n-camadas. Tais empresas, chegam a certo ponto em que gostariam de utilizar a mesma infra-estrutura de uma aplicação de três camadas, em aplicações de duas camadas. Isso mostra a necessidade de portabilidade de código entre as aplicações. Como conseguimos isso? Uma boa saída seria o uso de um framework comum a ambas as arquiteturas. Assim, usufruiríamos de todos os recursos relacionados anteriormente. Aí entra o DataSnap.

Voltando nosso foco ao Delphi, podemos observar que algumas empresas adotam uma saída não muito adequada para conseguir essa portabilidade. Diversas vezes chegam a colocar o cliente e o servidor rodando em cada estação de trabalho da empresa, emulando uma arquitetura em duas camadas (**Figura 1**)**.** O programa cliente se comunica com a camada de regras de negócios que, por sua vez, se comunica com o SGBD como em uma arquitetura em três camadas; porém, como o servidor de aplicações roda em cada máquina, temos uma emulação de duas camadas.

Em se tratando de DataSnap, veremos como é possível chegar à portabilidade sem queda na performance e perda de manutenibilidade, colocando apenas um programa rodando na máquina cliente.

Figura 1. Solução adotada por muitas empresas com o intuito de manter um só framework de aplicações

Mãos à obra

Construiremos agora um cadastro de usuários na arquitetura cliente/servidor. Em seguida, criaremos um *middleware* para nossa aplicação, fazendo-a rodar em três camadas. Usaremos no exemplo o banco de dados Firebird, mas sinta-se à vontade para usar o SGBD de sua preferência.

Nota: Presumimos que você já esteja familiarizado com a construção de aplicativos simples DataSnap e dbExpress.

Crie um novo banco de dados chamado "USUARIOS.FDB" e adicione uma tabela de nome "USUARIOS" com o seguinte código:

```
CREATE TABLE USUARIOS (
  CODIGO INTEGER NOT NULL PRIMARY KEY,
  NOME VARCHAR(40),
  LOGIN VARCHAR(15),
  SENHA VARCHAR(15))
```

No Delphi, inicie uma aplicação e salve os arquivos em uma pasta de sua escolha. Insira um DataModule (com o nome de "DM"), salvando-o como "uDM.pas". Adicione um *SQLConnection* ("SqlConn"), um *SQLQuery* ("sqrUsuarios"), um *DataSetProvider* ("dspUsuarios") e um *LocalConnection* ("LocalConn").

Dê um duplo clique na conexão e configure o acesso ao banco de dados criado anteriormente (através dos parâmetros *Database*, *User_Name* e *Password*) usando o driver para o SGBD escolhido. Não esqueça de deixar a propriedade *LoginPrompt* do *SqlConn* como *False*. O DataModule deve estar semelhante ao mostrado na **Figura 2**.

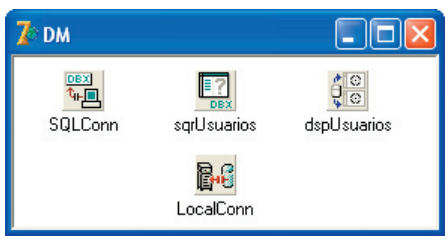


Figura 2. Visualização dos componentes do DataModule

Configure a propriedade *SqlConnection* do *sqrUsuarios* para *SqlConn* e *SQL* para "select * from USUARIOS". Ligue também o *dspClientes* ao *sqrUsuarios* através da propriedade *DataSet*.

Em uma aplicação cliente/servidor usual, teríamos o *DataSetProvider* ligado diretamente ao *ClientDataSet*. Já em uma arquitetura multicamadas, os dois componentes ficariam em aplicações separadas, trocando seus pacotes de dados através da interface *IAppServer*.

O DataSnap fornece um componente chamado *LocalConnection* que será usado para simularmos um ambiente de três camadas em duas. O *ClientDataSet* agora será "ligado" indiretamente a nossa conexão (*LocalConn*) e então continuará recebendo os pacotes do *DataSetProvider* como antes.

Quando quisermos portar a aplicação, simplesmente trocaremos essa conexão para a do servidor criado e teremos nosso cliente. Isso será visto adiante.

No formulário principal adicione um *ClientDataSet* ("cdsUsuarios") e um *ConnectionBroker* ("ConnBrk"), sendo esse último responsável pela abstração da conexão, ou seja, podemos ligar todos os *ClientDataSets* a um ou mais componentes *ConnectionBroker* e mudar facilmente a conexão com o servidor através da propriedade *Connection*.

Continuando, na seção *uses* da unit declare o seguinte código:

```
{$IFNDEF NTIER}
uses uDM;
{$ENDIF}
```

Isso fará com que o DataModule só seja referenciado se estivermos usando uma arquitetura de duas camadas. Veremos mais a frente como isso é possível.

Conecte o *cdsUsuarios* ao *ConnBrk* através da propriedade *ConnectionBroker* e digite "dspUsuarios" em *ProviderName*. No *ConnBrk* aponte a propriedade *Connection* para *DM.LocalConn*. Dê um duplo clique em *cdsUsuarios* e adicione todos os *TFields*, em seguida, arraste-os para o formulário, criando assim, os controles *Data-Aware* correspondentes. Nomeie o *DataSource* para "srcUsuarios". Coloque um *DBNavigator*, configurando *DataSource* para *srcUsuarios*.

A **Figura 3** mostra um possível modo de organização de seus componentes. Adicione o seguinte código ao evento *AfterPost* do *cdsUsuarios*:

Figura 3. Visualização dos componentes do formulário principal da aplicação

```
{ Aplica os registros modificados }
cdsUsuarios.ApplyUpdates(0);
```

No evento *OnShow* do formulário digite:

```
{ Abre cdsUsuarios, iniciando a requisição
```

```
  de pacotes a dspUsuarios }
cdsUsuarios.Open;
```

Compile e rode a aplicação. Como você pode ver, temos um programa simples que acessa o banco de dados normalmente, porém usando DataSnap como "intermediador".

Este exemplo só funcionou porque o *LocalConnection* implementa a interface *IAppServer*, assim como o *RemoteDataModule*. Essa interface é manipulada internamente pelo *ClientDataSet* para executar operações como a obtenção da lista dos *Providers*, dos registros e a atualização dos dados.

Agora vamos adicionar um código ao DataModule para verificar se determinado LOGIN já se encontra cadastrado. Implemente o método *LoginJaCadastrado* conforme a **Listagem 1.** E no evento *BeforePost* do *cdsUsuarios* digite o código da **Listagem 2**.

Esse código faz com que *LoginJaCadastrado* seja chamado localmente ou remotamente, dependendo da arquitetura escolhida. Certamente essa aplicação poderia ser melhorada com o uso de mais um DataModule, que representaria a camada cliente em nossa simulação. Além disso, para simplificar, estamos aplicando as atualizações diretamente após a chamada ao método *Post*, o que limita o aproveitamento das capacidades do *ClientDataSet*.

Listagem 1. Implementação do método LoginJaCadastrado

```
function TDM.LoginJaCadastrado(
  const PLogin: string): Boolean;
var
  ASqlQuery: TSqlQuery;
begin
  ASqlQuery := TSqlQuery.Create(nil);
  try
    ASqlQuery.SQLConnection := SQLConn;
    ASqlQuery.Sql.Add('select CODIGO from USUARIOS'+
     ' where LOGIN = ' + QuotedStr(PLogin));
    ASqlQuery.Open;
    Result := not ASqlQuery.IsEmpty;
  finally
    ASqlQuery.Free;
  end;
end;
```

Listagem 2. Evento BeforePost do cdsUsuarios

```
{$IFDEF NTIER}
if ConnBrk.AppServer.LoginJaCadastrado(
  cdsUsuariosLOGIN.AsString) then
{$ELSE}
if DM.LoginJaCadastrado(
  cdsUsuariosLOGIN.AsString) then
{$ENDIF}
  raise Exception.Create('Login já cadastrado');
```

Portando para COM+

Chegou a hora de migrarmos nossa aplicação para uma arquitetura de três camadas. Vejamos como essa operação tornou-se simples.

Crie uma biblioteca *ActiveX* (aba *ActiveX* do *Object Repository*). Adicione um *Transactional Data Module* (aba *Multitier* do *Object Repository*), configurando "DMServer" para a *CoClass*. Adicione ao projeto (*Project|Add to Project*) a unit do DataModule usado na aplicação anterior e a declare na sessão *uses* do *DMServer*.

Quando o *ClientDataSet* da aplicação cliente chama o método *AS_GetProviderNames* (*IAppServer*) de *DMServer*, ele espera receber uma lista com todos os *Providers* registrados no *Transactional Data Module*. O procedimento normal seria, então, colocar os componentes provedores diretamente sobre o *DMServer*; desse modo, ele próprio se encarregaria de registrá-los.

Entretanto, queremos aproveitar a estrutura construída anteriormente e, para isso, chamaremos manualmente o método *RegisterProvider* para cada *DataSetProvider* do DataModule reaproveitado. Declare *Provider* na seção *uses* de *DMServer* e digite no evento *OnCreate* o código da **Listagem 3**.

Listagem 3. Evento OnCreate do DMServer

```
procedure TDMServer.MtsDataModuleCreate(
  Sender: TObject);
var
  i: Integer;
begin
  DM := TDM.Create(Self);
  for i := 0 to Pred(DM.ComponentCount) do
    if DM.Components[i] is TDataSetProvider then
      RegisterProvider(
        TDataSetProvider(DM.Components[i]));
end;
```

Não podemos esquecer de exportar nosso método *LoginJaCadastrado* para que agora possa ser invocado remotamente. Abra a *Type Library* (*View|Type Library*), selecione a interface *IDMServer*, clique no botão *New Method* () e nomeie o novo método como "LoginJaCadastrado". Na aba *Parameters*, escolha em *Return Type* a opção *WordBool*. Adicione um parâmetro de *Name* "PLogin" e *Type* como *WideString*. Isso criará um método como o seguinte código:

```
function LoginJaCadastrado(
  const PLogin: WideString): WordBool; safecall;
```

Nota: Estamos usando a linguagem Pascal para a edição da *Type Library*. Para a edição na linguagem IDL, configure a opção *Language* da aba *Type Library* do *Environment Options*.

A diretiva *safecall* garante que qualquer exceção disparada seja passada para aplicação cliente. O tipo de retorno, *WordBool*, é semelhante ao tipo padrão *Boolean*; *WideString* se assemelha ao tipo *string*. Na implementação do método gerado coloque apenas: **Nota:** Estamos usando a linguagem Pascal para a edição da *Type Library*. Para a edição na linguagem IDL, configure a opção *Language* da aba *Type Library* do *Environment Options*.

```
Result := DM.LoginJaCadastrado(PLogin);
```

Para instalar o servidor, clique em *Run|Install COM+ Objects*, marque *DMServer* e crie uma nova aplicação (aba *Install into new Application* da janela *Install COM+ Object*) com o nome de "CadUsuarios".

Na **Figura 4** você pode visualizar a aplicação instalada no item *Serviços de Componentes* do Windows, acessado através do *Painel de Controle*. Retorne ao projeto anterior e adicione ao formulário principal um *DCOMConnection* ("conServer") apontando seu *ServerName* para "NomedoProjeto.DMServer" e configure o *ConnBrk* para acessar *conServer* (propriedade *Connection*), em vez da conexão local do DataModule.

Figura 4. Aplicação CadUsuarios rodando no COM+

Para finalizar, declare a diretiva condicional "NTIER", assim, nossa aplicação entenderá a nova arquitetura multicamadas. Faça isso, através do menu *Project|Options*, na aba *Directories|Conditionals*, como mostra a **Figura 5**. Para compilar a aplicação para usar duas camadas, basta remover a diretiva.

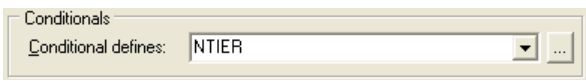


Figura 5. Configurando a diretiva condicional NTIER

Conclusões

Como o investimento em uma arquitetura multicamadas não é sempre aconselhável, devemos conhecer uma maneira eficiente para contornar a situação. Vimos, portanto, como construir uma estrutura adequada tanto para uma aplicação de duas quanto para de três camadas. Com isso podemos portar nossas aplicações e frameworks, facilitando um eventual processo de migração.

Vantagens da arquitetura

Concentração das regras de negócio: Confere ao código uma manutenção facilitada e comporta melhor as mudanças dos requisitos.
Interoperabilidade: Vários programas, escritos em linguagens diferentes, podem ser usados para o acesso a uma camada intermediária, desde que implementem o mesmo protocolo. É possível, por exemplo, que um servidor de aplicação seja acessado simultaneamente por um programa rodando no Windows, outro rodando em um servidor Web e outro rodando em um PDA.
Economia nas licenças com bancos de dados: O custo da licença de uso de alguns bancos de dados é calculado com base no número de conexões com os mesmos. A única conexão nesse caso seria a do servidor de aplicações.
Thin-Clients (Clientes "Magros"): Além de o programa cliente não consumir tantos recursos dá máquina "hospedeira", a distribuição do mesmo é muito facilitada, aumentando a escalabilidade da aplicação.

Gustavo Chaurais *(gustavo.chaurais@zerog.com) é Borland Delphi 7 Advanced Product Certified, graduando em Sistemas de Informação pela UFSC e membro do Quality Assurance Team da Proversoft/Zero G Software, empresa responsável pelo projeto InstallAnywhere. Especializado no desenvolvimento de frameworks de aplicações orientados a objetos para projetos de grande e médio porte.*

Download:
www.devmedia.com.br/clubedelphi/downloads

Date, Time e TimeStamp no InterBase

por **Andreano Lanusse**

Diferenças e operações básicas com os tipos de dados para armazenamento de data / hora

O InterBase nasceu com apenas um *datatype* (tipo de dados) para armazenamento de data e hora. A partir do InterBase 6.0, foram criados novos tipos para permitir o armazenamento de data, hora e data/hora (*Date, Time* e *TimeStamp*). Este artigo vai esclarecer todas as dúvidas e apresentar as melhores práticas para utilização de cada um desses tipos de dados.

Veremos como inserir dados nesses formatos e como eles são armazenados no IB. Mostraremos também como é feito cálculo sobre esses tipos, comparação e conversão de dados do tipo data.

Versões anteriores ao InterBase 6 suportavam apenas o tipo *Date*, onde data e hora são armazenadas juntas em um único campo. No InterBase 6 e posteriores, o *Date* sofreu algumas modificações. *Time* e *TimeStamp* foram incorporados, sendo esses novos tipos só suportados no dialeto 3. O *Date* passou a ser *TimeStamp* e armazena data e hora como nas versões anteriores, já o *Date* passou a armazenar apenas data e o *Time* apenas a hora.

Antes de conhecermos as diferenças e vermos as principais operações envolvendo esses tipos de dados, é importante saber como o Interbase armazena datas.

Como as datas são armazenadas no InterBase

O InterBase grava os valores de data incluindo o valor do ano inteiro, com quatro posições. Se o usuário informar apenas dois dígitos para o ano, o InterBase usa um algoritmo chamado de *sliding window* para gerar os quatro dígitos. Quando as informações forem consultadas no banco de dados, os quatro dígitos do ano serão retornados. Através do algoritmo o InterBase define o século para gravação, quando são informados apenas dois dígitos do ano. Esse algoritmo é interno do InterBase, não podendo assim, ser descrito seu funcionamento e características.

O tipo *Date* permite diversos formatos para inserir as informações no banco de dados, que são:

- YYYYpMMpDD;
- MMpDDpYYYY;
- DDpMMpYYYY;
- MMpDDpYY;
- DDpMMpYY.

Onde:
DD = dia composto de um ou dois dígitos;
MM = mês composto de um ou dois dígitos, ou de três letras abreviadas, ou do nome completo do mês em Inglês;
YY = último dois dígitos do ano;
YYYY = quatro dígitos do ano;
p = qualquer caractere que representa a separação das datas ("/" ou "."), TAB ou espaço são ignorados.
Temos algumas restrições como:

- No formato *Ano-Mês-Dia*, o ano sempre tem que estar com quatro dígitos;
- No formato *Mês-Dia-Ano*, o ano pode estar com dois ou quatro dígitos, no caso de dois, o algoritmo *sliding window* será aplicado;
- Se uma forma toda numérica for utilizada e o ano vier por último, o InterBase supõe que o formato é *Dia-Mês-Ano*. Exemplo: 12.04.2002 é interpretado como "dia 12 de Abril de 2002", mas 12-04-02 é interpretado como "dia 4 de dezembro de 2002".

Convertendo Datas (Cast)

As conversões de data são comuns em bancos de dados e são conhecidas como *Cast*, por exemplo, a conversão de uma data para *string* e vice-versa. Vamos agora esclarecer quais os formatos permitidos nas conversões. Na **Figura 1**, temos um exemplo de como efetuar conversões de data para *string* através de comando SQL.

Convertendo-se *Date* para *string*, os seguintes formatos são permitidos:

- yyyy-mm-dd
- yyyy/mm/dd
- yyyy mm dd
- yyyy:mm:dd
- yyyy.mm.dd

Em todos os formatos citados anteriormente, podemos trocar os dois dígitos do mês pelas três letras do mês abreviadas em inglês

ou o nome do mês também em inglês.

Outras formas também são permitidas:

- mm-dd-yy
- mm-dd-yyyy
- mm/dd/yy
- mm/dd/yyyy
- mm dd yy
- mm dd yyyy
- mm:dd:yy
- mm:dd:yyyy
- dd.mm.yy
- dd.mm.yyyy

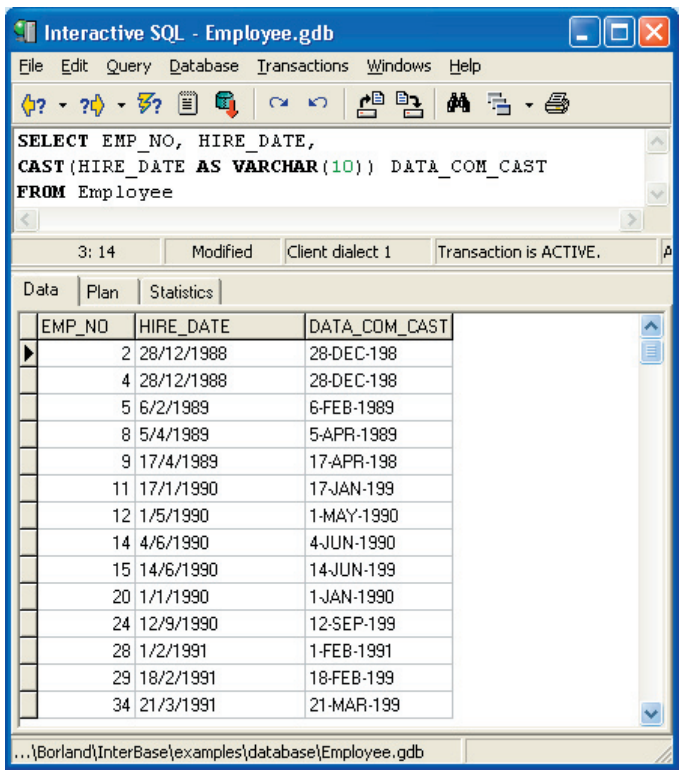


Figura 1. Exemplo de utilização de Cast

Se a data informada estiver apenas com dois dígitos no ano, o algoritmo *sliding window* será utilizado para definir o ano. Se você usar o nome do mês ou a abreviação de três letras para representar o mês, lembre-se que o dia ou o mês deve ser informado primeiro. Os exemplos da **Listagem 1** usam "xxx" para representar o nome do mês ou a abreviação, essas formas são permitidas.

Na **Figura 2** temos um exemplo de como converter uma *string* em formato de data. O retorno da coluna *Data_To_String* será do tipo *Date*. Convertendo-se uma *string* para *Date* o resultado será obtido no formato *Ano-Mês-Dia*, onde "MM" são os dois dígitos representando o mês. Se a *string* enviada não estiver nesse formato, uma *exception* (erro) será chamada.

Listagem 1. Formas permitidas de abreviação de datas

```
Inserindo a data "20 de janeiro de 2005" em diversos for-
matos
INSERT INTO t1 VALUES ('2005-01-20');
INSERT INTO t1 VALUES ('01/20/2005');
INSERT INTO t1 VALUES ('20.01.2005');
INSERT INTO t1 VALUES ('jan 20 2005');


Irá gravar "20 de janeiro de 2005"
INSERT INTO t1 VALUES ('01/20/05');

dd-xxx-yy
dd-xxx-yyyy
xxx-dd-yy
xxx-dd-yyyy
dd xxx yy
dd xxx yyyy
xxx dd yy
xxx dd yyyy
dd:xxx:yy
dd:xxx:yyyy
xxx:dd:yy
xxx:dd:yyyy
```

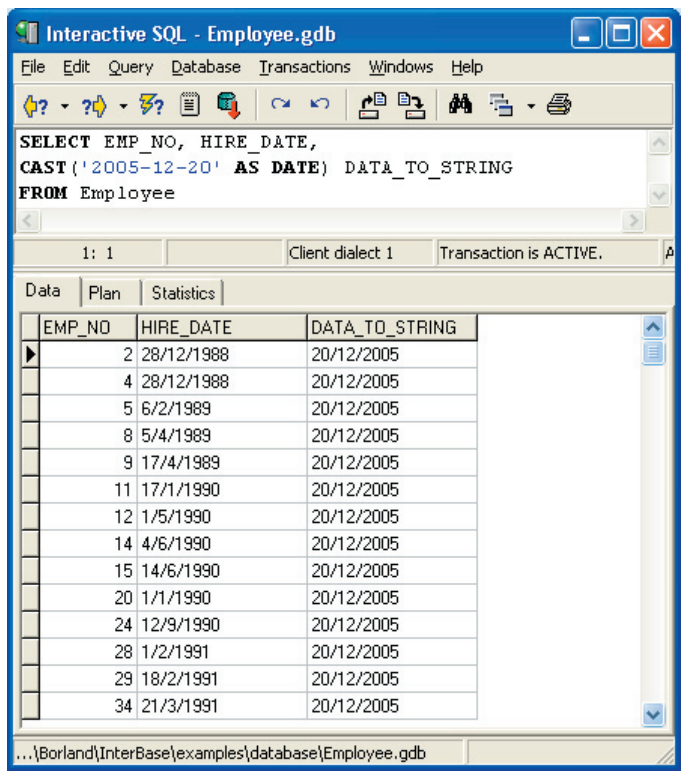


Figura 2. Convertendo data para string

Somando e subtraindo tipo Date, Time e TimeStamp

Na **Tabela 1** apresentamos as operações e seus efeitos ao fazer cálculos com *Date*, *Time*, *TimeStamp* com valores numéricos (*Integer*, *Numeric* ou *Decimal*). É importante fixar que o campo *TimeStamp* é formado por data/hora e internamente é uma espécie de *Float*, onde a parte inteira é a data e a parte decimal representa as horas.

Comparando Date e Time

Finalmente, é importante saber como o IB faz a comparação entre *strings* e campos do tipo *Date* e *Time*. Considere o seguinte exemplo:

```
Tabela.DataCadastro <= '12/31/1999'.
```

Nesse caso, o IB converte automaticamente a *string 12/31/1999* para o tipo *Date* para efetuar a comparação internamente, obtendo o resultado desejado.

Agora considere a seguinte expressão:

```
'30.6.2000' < '1.7.2000'
```

Observe que a comparação está sendo feita entre duas *strings*, obviamente, o que pode gerar um resultado diferente se os valores fossem realmente do tipo *date*. Nesse caso, a solução é converter as *strings* para o tipo desejado através da função *Cast*:

```
CAST('30.6.2000' AS DATE) < CAST('1.7.2000' AS DATE)
```

É claro, a *string* deve ser válida para essa comparação, como foi explicado anteriormente.

Conclusões

Vimos neste artigo como o InterBase manipula internamente os tipos *Date*, *Time* e *TimeStamp* e as diversas formas de se trabalhar com esses tipos. Até a próxima!

Andreano Lanusse *(andreano.lanusse@borland.com ou andreanobr@yahoo.com) é System Engineer da Borland Latin America, especialista em InterBase há 10 anos, larga experiência em desenvolvimento de aplicações, é certificado Borland Instructor, Delphi, C++ Builder, JBuilder, Together, StarTeam, CaliberRM, MCP em SQL Server e Windows 2000. Possui uma coluna no site www.sqlmagazine.com.br*

Tabela 1. Exemplos de soma de campos Date, Time e TimeStamp

Operador 1	Operador	Operador 2	Resultado
Date	+	*Date*	erro
Date	+	*Time*	*TimeStamp* (concatenação)
Date	+	*TimeStamp*	erro
Date	+	Valor Numérico	*Date* + número de dias (parte fracional é ignorada)
Time	+	*Date*	*TimeStamp* (concatenação)
Time	+	*Time*	erro
Time	+	*TimeStamp*	erro
Time	+	Valor Numérico	*Time* + número de segundos, módulo aritmético de 24 horas
TimeStamp	+	*Date*	erro
TimeStamp	+	*Time*	erro
TimeStamp	+	*TimeStamp*	erro
TimeStamp	+	Valor Numérico	*TimeStamp*, date número de dias; *Time* + fração de dias convertidas para segundos
Date	-	*Date*	*Decimal*(9,0) representando o número de dias
Date	-	*Time*	erro
Date	-	*TimeStamp*	erro
Date	-	Valor Numérico	*Date* = número de dias (parte fracional é ignorada)
Time	-	*Date*	erro
Time	-	*Time*	*Decimal*(9,4) representando o número de segundos
Time	-	*TimeStamp*	erro
Time	-	Valor Numérico	*Time* - número de segundos, módulo aritmético de 24 horas
TimeStamp	-	*Date*	erro
TimeStamp	-	*Time*	erro
TimeStamp	-	*TimeStamp*	*Decimal*(18,9) representando dia e fração do dia
TimeStamp	-	Valor Numérico	*TimeStamp*: date – número de dias; *Time*: fração do dia convertida para segundos

Novidades da Delphi Language

por Laércio Queiroz

Parte I - Usando Class Helpers

Este é o primeiro de uma série de artigos que mostrará os novos recursos incluídos no compilador do Delphi 2005. Muitos recursos que foram incluídos no compilador do Delphi 8 for .NET estão agora disponíveis para o Win32, de forma que também serão apresentados nesta série. Para facilitar, cada mês apresentaremos uma novidade, que são muitas! Conheceremos o *for in* (o equivalente ao *foreach* do C#), a diretiva *inline*, *class constructors*, *class properties*, novos especificadores de visibilidade (*strict private* e *strict protected*), classes *sealed*, métodos *final*, suporte a atributos e muito mais.

Vamos começar por uma excelente e interessante melhoria, a possibilidade de anexar funcionalidades a uma classe sem precisar usar o recurso de herança, através da utilização de *class helpers*. Para facilitar o entendimento, veremos neste artigo como utilizar essa técnica em uma aplicação bastante simples.

O que são Class Helpers

Class Helper é uma classe especial que permite introduzir métodos e propriedades em outra classe, sem usar herança. Todos os métodos implementados serão "herdados" pela classe "alvo", incluindo assim funcionalidades em toda uma hierarquia. Quando declaramos um *Class Helper*, indicamos o nome da classe "auxiliar" e o nome da classe que ela vai estender. Veja um exemplo:

```
ClasseAuxiliar = class helper for Classe
```

Exemplo usando Class Helpers

Vamos criar um exemplo que introduz uma propriedade e dois métodos à classe *System.Windows.Forms.TextBox*. No Delphi 2005, crie uma aplicação do tipo *Windows Forms Application* e vá até o *menu File>New>Other>Delphi for .NET Projects>New Files>Unit*

Nota: Apesar deste exemplo usar .NET lembre-se, novamente, que o recurso também é válido para aplicações Win32.

É nessa unit que iremos implementar o *Class Helper*. Digite nela o código da **Listagem 1**. Observe que a referência à propriedade *Text* da classe *TextBox* é feita como se a classe *TextBoxAUX* herdasse os dados da classe mapeada.

Listagem 1. Usando class helpers

```
uses
  System.Windows.Forms;
...
type
 TextBoxAUX = class helper for TextBox
 protected
  { Métodos de acesso à propriedade PropAUX }
   procedure SetAUX(value: string);
   function GetAUX: string;
 public
  { Declarando a propriedade PropAUX }
  property PropAUX: string read GetAUX write SetAUX;
end;

implementation

{ TextBoxAUX }

function TextBoxAUX.GetAUX: string;
begin
 { GET de PropAUX }
  Result := 'Conteúdo do TextBox: "'+Text+'"';
end;

procedure TextBoxAUX.SetAUX(value: string);
begin
 { Set de PropAUX }
 if Text <> value then
  Text := value;
end;

end.
```

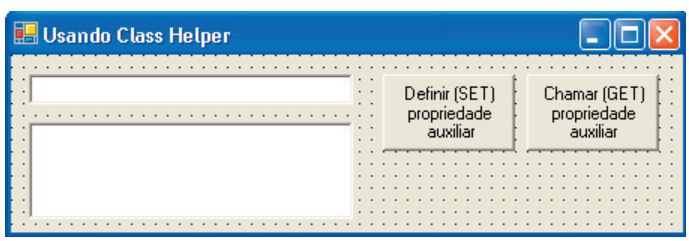


Figura 1. Formulário com os componentes para o exemplo

No formulário, coloque dois *TextBoxes* e dois *Buttons* (**Figura 1**). Adicione a unit criada anteriormente na cláusula *uses* do formulário. No evento *Click* do *Button1* digite o seguinte código:

```
{ Definindo um valor para a propriedade
  auxiliar PropAUX }
TextBox1.PropAUX :=
  'ClubeDelphi - Usando Class Helper';
```

No código anterior, definimos a propriedade adicionada na classe auxiliar. Observe que a propriedade *PropAUX* foi "injetada" na classe *TextBox*. No evento *Click* do *Button2*, digite o seguinte código, onde lemos o conteúdo da propriedade *PropAux* do *TextBox1*:

```
{ Lendo o conteúdo da propriedade Text
  mapeada por PropAUX }
TextBox2.Text := TextBox1.PropAUX;
```

Vimos neste artigo um exemplo simples da utilização de *class helpers*, um novo e poderoso recurso do compilador do Delphi 2005. Continuamos a série na próxima edição, um abraço a todos e até lá!

Laércio Santos de Queiroz *(laercio_queiroz@hotmail.com) é Webdesigner e desenvolvedor especializado em tecnologias Web e multicamadas, além de possuir conhecimentos em ASP, ASP.NET, Java e Delphi.*

Download:
www.devmedia.com.br/clubedelphi/downloads

Sistema SysCash

por **Everson Volaco**

Parte II – Criando as regras de negócio

Nessa segunda parte da série, implementaremos as *triggers* e *stored procedures* contendo as regras de negócio do sistema *SysCash*. Iniciaremos também o desenvolvimento da aplicação Delphi, criando as telas de cadastro, previsões (contas a pagar/receber) e lançamentos (avulsos e baixas).

Gerando o histórico

Todas as operações realizadas nas tabelas *Cax_Contas*, *Cax_Previsoes* e *Cax_Lancamentos* serão armazenadas na tabela de histórico do sistema. Cada vez que o usuário do sistema incluir, alterar ou apagar um registro dessas tabelas, será disparada uma *trigger* que armazenará as informações do registro em questão, na tabela de histórico (*Cax_Historico*).

Todo o processo de geração do histórico será feito através de *triggers* vinculadas a eventos das tabelas. Utilizaremos os eventos *After Insert*, *After Update* e *After Delete* para inserir as informações na tabela de histórico. Veja na **Listagem 1** as instruções SQL para geração do histórico da tabela *Cax_Contas*.

Listagem 1. Triggers para geração do histórico referente às operações efetuadas na tabela Cax_Contas

```
/*CAX_CONTAS - Após inserir um novo registro*/
CREATE TRIGGER "TRG_INS_CON_HIS" FOR "CAX_CONTAS"
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
  INSERT INTO CAX_HISTORICO (HIS_PRELANCON, HIS_TIPO,
    HIS_CODREFERENCIA, HIS_DESCRNOVA, HIS_VALORNOVO)
  VALUES ('C', 'I', NEW.CON_CODIGO, NEW.CON_DESCRICAO,
  NEW.CON_SALDOINICIAL);
END;

/*CAX_CONTAS - Após alterar um registro*/
CREATE TRIGGER "TRG_UPD_CON_HIS" FOR "CAX_CONTAS"
ACTIVE AFTER UPDATE POSITION 0
AS
BEGIN
  IF ((OLD.CON_DESCRICAO <> NEW.CON_DESCRICAO) OR
  (OLD.CON_SALDOINICIAL <> NEW.CON_SALDOINICIAL)) THEN
  BEGIN
    INSERT INTO CAX_HISTORICO (HIS_PRELANCON, HIS_TIPO,
    HIS_CODREFERENCIA, HIS_DESCRANTIGA, HIS_DESCRNOVA,
   HIS_VALORANTIGO, HIS_VALORNOVO)
      VALUES ('C', 'A', NEW.CON_CODIGO,
      OLD.CON_DESCRICAO, NEW.CON_DESCRICAO,
      OLD.CON_SALDOINICIAL, NEW.CON_SALDOINICIAL);
  END
END;

/*CAX_CONTAS - Após apagar um registro*/
CREATE TRIGGER "TRG_DEL_CON_HIS" FOR "CAX_CONTAS"
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
  INSERT INTO CAX_HISTORICO (HIS_PRELANCON, HIS_TIPO,
  HIS_CODREFERENCIA, HIS_DESCRANTIGA, HIS_VALORANTIGO)
   VALUES ('C', 'D', OLD.CON_CODIGO, OLD.CON_DESCRICAO,
   OLD.CON_SALDOINICIAL);
END;
```

Para o campo *His_PreLanCon* da tabela *Cax_Historico* armazenamos os valores "C" (Contas), "P" (Previsões) ou "L" (Lançamentos) para identificar a origem do registro. No campo *His_Tipo* armazenaremos os valores "I" (Inclusão), "A" (Alteração) ou "D" (*Delete/*Exclusão) para indicar a operação que originou o registro no histórico.

Na tabela de histórico serão incluídas apenas as informações mais importantes de cada tabela. Na *trigger Trg_Upd_Con_His*, por exemplo, só geraremos um histórico da alteração caso os campos *Con_Descricao* ou *Con_SaldoInicial* tenham sido alterados pelo usuário.

Dica: A variável *Old* guarda temporariamente os valores antigos das colunas, enquanto que a variável *New* armazena os novos valores a serem inseridos/alterados nas colunas.

Você pode baixar o *script* contendo as instruções SQL para definição das *triggers* para geração de histórico das tabelas *Cax_Previsoes* e *Cax_Lancamentos* a partir do endereço de download deste artigo, como também os arquivos do banco para FB 1.5, IB 6 e IB 7.5.

Controlando a inclusão/alteração do saldo inicial das contas

Cada vez que uma conta é cadastrada ou alterada, o sistema permite que o usuário entre com um saldo inicial. Essa opção é interessante, visto que quando uma empresa passa a utilizar um sistema informatizado para controlar suas contas, como caixa e contas bancárias, por exemplo, normalmente essas contas já possuem algum saldo (positivo ou negativo).

O saldo diário de cada conta cadastrada no sistema será armazenado junto com a data na tabela *Cax_Saldos*. Caso algum valor seja incluído ou alterado no campo *Con_SaldoInicial* da tabela *Cax_Contas,* utilizaremos uma *stored procedure* para atualizar o saldo na tabela *Cax_Saldos*. Para criar a SP, utilize a instrução da **Listagem 2**.

Utilizamos o comando *Exists* para verificar se já existe um registro com o código da conta e a data informada nos parâmetros de entra-

da da SP. Faremos a execução dessa SP, através de *triggers* vinculadas a eventos da tabela *Cax_Contas*. Utilizaremos os eventos *After Insert* e *Before Update* para fazer a atualização do saldo. Veja na **Listagem 3** as instruções SQL para criação das duas *triggers*.

Ambas as *triggers* passarão para a SP a data corrente, fazendo a inclusão ou alteração do saldo sempre para um dia anterior ao dia atual. Essa regra de negócio é aplicada no sistema principalmente devido ao processo de inclusão de uma nova conta. Quando o usuário cadastrar uma conta com saldo inicial diferente de zero, o sistema gerará uma movimentação diária com a data do cadastro da conta e com o saldo igual ao informado no campo saldo inicial.

Nota: A instrução *Position 5* utilizada em *Trg_Inc_Contas_Saldo* indica que ela deve ser disparada após a execução da *trigger Trg_Ins_Con_His* (que possui o valor "0"), que também está vinculada ao evento *After Insert* da tabela *Cax_Contas*.

Listagem 2. Criando uma Stored Procedure para atualizar o saldo

```
CREATE PROCEDURE PROC_ATUALIZA_SALDO_CONTA (
  CODCONTA INTEGER, DT DATE, VALOR NUMERIC(18, 2))
AS
BEGIN
  /*Caso já exista um saldo para a conta na data
  informada no parâmetro DT, atualiza o valor*/
  IF (EXISTS(SELECT 1 FROM CAX_SALDOS
      WHERE CON_CODIGO = :CODCONTA AND
      SAL_DATASALDO = :DT)) THEN
  BEGIN
    UPDATE CAX_SALDOS SET SAL_SALDO =
      SAL_SALDO + :VALOR
     WHERE CON_CODIGO = :CODCONTA AND
      SAL_DATASALDO = :DT;
  END
  ELSE
  /*Caso não haja saldo para a conta na data informada
   no parâmetro DT, inclui o valor*/
  BEGIN
    INSERT INTO CAX_SALDOS(CON_CODIGO, SAL_DATASALDO,
      SAL_SALDO) VALUES (:CODCONTA, :DT, :VALOR);
  END
END
```

Listagem 3. Triggers para execução da stored procedure pra atualização do saldo diário da conta a partir do saldo inicial

```
CREATE TRIGGER "TRG_INC_CONTAS_SALDO" FOR "CAX_CONTAS"
ACTIVE AFTER INSERT POSITION 5
AS
DECLARE VARIABLE DT DATE;
BEGIN
  DT = 'NOW';
  DT = DT - 1;
  /*Caso saldo inicial seja diferente de 0
    durante o cadastro*/
  IF (NEW.CON_SALDOATUAL <> 0) THEN
    EXECUTE PROCEDURE PROC_ATUALIZA_SALDO_CONTA(
    NEW.CON_CODIGO, DT, NEW.CON_SALDOATUAL);
END;

CREATE TRIGGER "TRG_UPD_CONTAS_SALDO" FOR "CAX_CONTAS"
ACTIVE BEFORE UPDATE POSITION 0
AS
DECLARE VARIABLE SALDO NUMERIC(18,2);
DECLARE VARIABLE DT DATE;
BEGIN
  DT = 'NOW';
  DT = DT - 1;
  /* Caso saldo inicial tenha sido alterado */
  IF (OLD.CON_SALDOINICIAL <> NEW.CON_SALDOINICIAL) THEN
  BEGIN
    NEW.CON_SALDOATUAL = (NEW.CON_SALDOATUAL -
      OLD.CON_SALDOINICIAL) + NEW.CON_SALDOINICIAL;
    SALDO = NEW.CON_SALDOINICIAL - OLD.CON_SALDOINICIAL;
    EXECUTE PROCEDURE PROC_ATUALIZA_SALDO_CONTA(
      NEW.CON_CODIGO, DT, SALDO);
  END
END;
```

Controlando lançamentos avulsos e baixas

O sistema possibilita o cadastro de previsões de contas a receber/pagar além de lançamentos avulsos e de baixas a partir de uma previsão pré-cadastrada. Utilizaremos a tabela *Cax_Lancamentos* para armazenar tanto os lançamentos avulsos (sem previsão) como as baixas feitas pelo usuário. Para que possamos diferenciar uma baixa de um lançamento avulso, guardaremos o código da previsão baixada no campo *Lan_CodPrevisaoRef*.

Quando uma baixa for feita pelo usuário, o sistema deverá guardar a data da baixa e marcar a previsão como "baixada". Para executar esses procedimentos criaremos uma *trigger* vinculada ao evento *After Insert* da tabela *Cax_Lancamentos*. Veja o código da **Listagem 4** para criação da *trigger*.

Listagem 4. Trigger para marcar a previsão como baixada

```
/* Exceção utilizada na trigger de atualização
   da data da baixa da previsão lançada */
CREATE EXCEPTION "ERRO_LAN_UPD_PRE"
  'Erro ao atualizar a data da baixa da previsao';

CREATE TRIGGER "TRG_LAN_UPD_PRE" FOR "CAX_LANCAMENTOS"
ACTIVE AFTER INSERT POSITION 5
AS
BEGIN
  /* Caso seja um lançamento de baixa de previsão */
  IF (NEW.LAN_CODPREVISAOREF IS NOT NULL) THEN
  BEGIN
    UPDATE CAX_PREVISOES SET PRE_DATABAIXA =
      NEW.LAN_DATALANCAMENTO, PRE_PREVISAO = 'N'
    WHERE PRE_CODIGO = NEW.LAN_CODPREVISAOREF;
    WHEN ANY DO
      EXCEPTION ERRO_LAN_UPD_PRE;
  END
END;
```

Na *trigger* verificamos se o campo *Lan_CodPrevisaoRef* contém algum valor indicando que o lançamento foi feito a partir de uma previsão cadastrada. Caso haja um valor, é feita a alteração na previsão baixada guardando a data da baixa e marcando a previsão como "lançada". Caso algum erro ocorra durante a operação, a exceção *Erro_Lan_Upd_Pre* é levantada.

Atualização do saldo atual e saldo diário

Cada vez que um lançamento for incluído, alterado ou excluído, o saldo atual e o saldo diário da conta vinculada ao lançamento

devem ser atualizados. Para a atualização do saldo diário na tabela *Cax_Saldos*, utilizaremos uma *stored procedure* para recuperar o valor anterior da conta em relação à data do lançamento em questão. Utilize a instrução SQL da **Listagem 5** para criar a SP.

A SP retornará o saldo diário mais atual e que seja menor que a data passada no parâmetro de entrada (*DT*), referente à conta informada no parâmetro *CodConta*. Caso não exista um saldo diário referente à conta vinculada ao lançamento, na data em que o lançamento foi efetuado, utilizaremos o valor retornado pela SP como base para gerar um novo registro na tabela *Cax_Saldos*, armazenando o saldo diário da conta para a data do lançamento.

Listagem 5. Criando a procedure que retornará o Saldo Anterior

```
CREATE PROCEDURE PROC_RETORNA_SALDO_ANTERIOR (
  CODCONTA INTEGER, DT DATE)
RETURNS (VALOR NUMERIC(18, 2))
AS
BEGIN
  SELECT SAL_SALDO FROM CAX_SALDOS
   WHERE CON_CODIGO = :CODCONTA
     AND SAL_DATASALDO = (SELECT MAX(SAL_DATASALDO)
       FROM CAX_SALDOS WHERE SAL_DATASALDO < :DT)
    INTO :VALOR;
END;
```

O próximo passo é criarmos as *triggers* que farão a atualização do saldo atual e diário da conta utilizando a SP recém criada. Vincularemos as *triggers* aos eventos *After Insert*, *After Update* e *After Delete* da tabela *Cax_Lancamentos*. Veja na **Listagem 6** as instruções SQL para criação das três *triggers*.

Cada vez que um lançamento é incluído, alterado ou excluído, uma *trigger* é disparada fazendo a atualização do campo *Con_SaldoAtual* (saldo atual) da tabela *Cax_Contas* e do campo *Sal_Saldo* (saldo diário) da tabela *Cax_Saldos*.

Caso o lançamento em questão seja o primeiro do dia para a conta vinculada, a *trigger* utiliza a SP *Proc_Retorna_Saldo_Anterior* para recuperar o saldo anterior e criar um novo registro na tabela *Cax_Saldos* referente ao saldo diário corrente.

Nota: A *trigger Trg_Lan_Upd_Con_Sal* vinculada ao evento *After Update* da tabela *Cax_Lancamentos* utiliza vários recursos disponíveis no IB/FB, tornando-se a *trigger* mais complexa e extensa do sistema *SysCash*.

As principais regras de negócio do sistema estão vinculadas à tabela *Cax_Lancamentos* e suas *triggers*. No terceiro artigo da série veremos como extrair as informações referentes a saldos e valores através de SP e criaremos relatórios com o Rave Reports.

Listagem 6. Triggers para atualização do saldo atual e diário da conta vinculada ao lançamento

```
CREATE TRIGGER "TRG_LAN_INS_CON_SAL" FOR
   "CAX_LANCAMENTOS"
ACTIVE AFTER INSERT POSITION 10
AS
DECLARE VARIABLE VALOR NUMERIC(18,2);
DECLARE VARIABLE SALDO_ANTERIOR NUMERIC(18,2);
BEGIN
  /* Caso o lançamento seja de crédito
    (conta a receber) */
  IF (NEW.LAN_CREDEB = 'C') THEN
    VALOR = - NEW.LAN_VALOR;
  ELSE
    VALOR = NEW.LAN_VALOR;
  /* Atualiza o Saldo Atual da conta na
    tabela de Contas */
  UPDATE CAX_CONTAS SET CON_SALDOATUAL =
    CON_SALDOATUAL - :VALOR
  WHERE CON_CODIGO = NEW.CON_CODIGO;
  /* Insere ou atualiza o saldo do dia na
     tabela de Saldos */
  IF (EXISTS(SELECT 1 FROM CAX_SALDOS
      WHERE CON_CODIGO = NEW.CON_CODIGO AND
      SAL_DATASALDO = NEW.LAN_DATALANCAMENTO)) THEN
  BEGIN
    UPDATE CAX_SALDOS SET SAL_SALDO =
      SAL_SALDO - :VALOR
    WHERE CON_CODIGO = NEW.CON_CODIGO AND
      SAL_DATASALDO = NEW.LAN_DATALANCAMENTO;
  END
  ELSE
  BEGIN
    EXECUTE PROCEDURE PROC_RETORNA_SALDO_ANTERIOR(
      NEW.CON_CODIGO, NEW.LAN_DATALANCAMENTO)
     RETURNING_VALUES (:SALDO_ANTERIOR);
    VALOR = SALDO_ANTERIOR - VALOR;
    INSERT INTO CAX_SALDOS (CON_CODIGO, SAL_DATASALDO,
      SAL_SALDO) VALUES (NEW.CON_CODIGO,
      NEW.LAN_DATALANCAMENTO, :VALOR);
  END
END;

CREATE TRIGGER "TRG_LAN_UPD_CON_SAL" FOR
   "CAX_LANCAMENTOS"
ACTIVE AFTER UPDATE POSITION 5
AS
DECLARE VARIABLE VALOR NUMERIC(18,2);
DECLARE VARIABLE SALDO_ANTERIOR NUMERIC(18,2);
BEGIN
  /* Verifica se o flag LAN_APAGADO foi
     alterado para 'S' */
  IF ((OLD.LAN_APAGADO = 'N') AND
    (NEW.LAN_APAGADO = 'S')) THEN
  BEGIN
    IF (OLD.LAN_CREDEB = 'C') THEN
      VALOR = OLD.LAN_VALOR;
    ELSE
      VALOR = - OLD.LAN_VALOR;
 /* Atualiza o saldo atual da conta em questão na tabela de contas */
    UPDATE CAX_CONTAS SET CON_SALDOATUAL =
      CON_SALDOATUAL - :VALOR
    WHERE CON_CODIGO = OLD.CON_CODIGO;
 /* Atualiza o saldo do dia ref. a conta na tabela de Saldos */
    UPDATE CAX_SALDOS SET SAL_SALDO =
      SAL_SALDO - :VALOR WHERE CON_CODIGO =
      OLD.CON_CODIGO AND SAL_DATASALDO =
      OLD.LAN_DATALANCAMENTO;
  END
  ELSE
  /* Verifica se o campo valor, credeb, datalancamento
     ou codconta foi modificado */
  IF ((OLD.LAN_VALOR <> NEW.LAN_VALOR) OR
     (OLD.LAN_CREDEB <> NEW.LAN_CREDEB) OR
     (OLD.LAN_DATALANCAMENTO <> NEW.LAN_DATALANCAMENTO)
```

```
      OR (OLD.CON_CODIGO <> NEW.CON_CODIGO)) THEN
  BEGIN
    /* Desfaz o lançamento com seus valores antigos */
    IF (OLD.LAN_CREDEB = 'C') THEN
      VALOR = OLD.LAN_VALOR;
    ELSE
      VALOR = - OLD.LAN_VALOR;
    /* Atualiza o saldo atual da conta em questão na
       tabela de contas */
    UPDATE CAX_CONTAS SET CON_SALDOATUAL =
      CON_SALDOATUAL - :VALOR
    WHERE CON_CODIGO = OLD.CON_CODIGO;
    /* Atualiza o saldo do dia ref. a conta na
       tabela de Saldos */
    UPDATE CAX_SALDOS SET SAL_SALDO =
      SAL_SALDO - :VALOR
    WHERE CON_CODIGO = OLD.CON_CODIGO AND
      SAL_DATASALDO = OLD.LAN_DATALANCAMENTO;

    /* Refaz o lançamento com seus valores novos */
    IF (NEW.LAN_CREDEB = 'C') THEN
      VALOR = - NEW.LAN_VALOR;
    ELSE
      VALOR = NEW.LAN_VALOR;
    /* Atualiza o saldo atual da conta em questão
      na tabela de contas */
    UPDATE CAX_CONTAS SET CON_SALDOATUAL =
      CON_SALDOATUAL - :VALOR
    WHERE CON_CODIGO = NEW.CON_CODIGO;
    /* Atualiza ou insere o saldo na nova data
      ou conta */
    IF (EXISTS(SELECT 1 FROM CAX_SALDOS
        WHERE CON_CODIGO = NEW.CON_CODIGO AND
        SAL_DATASALDO = NEW.LAN_DATALANCAMENTO)) THEN
    BEGIN
      UPDATE CAX_SALDOS SET SAL_SALDO =
        SAL_SALDO - :VALOR
      WHERE CON_CODIGO = NEW.CON_CODIGO AND
        SAL_DATASALDO = NEW.LAN_DATALANCAMENTO;
    END
    ELSE
    BEGIN
      EXECUTE PROCEDURE PROC_RETORNA_SALDO_ANTERIOR(
        NEW.CON_CODIGO, NEW.LAN_DATALANCAMENTO)
        RETURNING_VALUES (:SALDO_ANTERIOR);
      VALOR = SALDO_ANTERIOR - VALOR;
      INSERT INTO CAX_SALDOS (CON_CODIGO, SAL_DATASALDO,
        SAL_SALDO) VALUES (NEW.CON_CODIGO,
        NEW.LAN_DATALANCAMENTO, :VALOR);
  END
  END
END;

CREATE TRIGGER "TRG_LAN_DEL_CON_SAL" FOR "CAX_LANCAMEN-
TOS"
ACTIVE AFTER DELETE POSITION 5
AS
DECLARE VARIABLE VALOR NUMERIC(18,2);
BEGIN
  /* Verifica se o lancamento já não havia sido
     marcado como apagado */
  IF (OLD.LAN_APAGADO <> 'S') THEN
  BEGIN
    IF (OLD.LAN_CREDEB = 'C') THEN
      VALOR = OLD.LAN_VALOR;
    ELSE
      VALOR = - OLD.LAN_VALOR;
    /* Atualiza o saldo atual da conta em questão
       na tabela de contas */
    UPDATE CAX_CONTAS SET CON_SALDOATUAL =
     CON_SALDOATUAL - :VALOR
    WHERE CON_CODIGO = OLD.CON_CODIGO;
    /* Atualiza o saldo do dia ref. a conta na
       tabela de Saldos */
    UPDATE CAX_SALDOS SET SAL_SALDO =
      SAL_SALDO - :VALOR
    WHERE CON_CODIGO = OLD.CON_CODIGO AND
      SAL_DATASALDO = OLD.LAN_DATALANCAMENTO;
  END
END;
```

Filtrando previsões ativas no sistema

Quando o usuário cadastra uma previsão de conta a pagar ou a receber, ele informa ao sistema a data de vencimento. Quando o pagamento ou o recebimento cadastrado na previsão ocorre efetivamente, o usuário executa a baixa dessa previsão no sistema.

Na tela de baixas devemos disponibilizar para o usuário apenas as previsões que ainda não foram baixadas. Para realizar esse filtro na tabela *Cax_Previsoes*, criaremos uma *view* no banco de dados. Ela será responsável em filtrar apenas as previsões que ainda estão ativas no sistema. Utilize a instrução SQL da **Listagem 7** para criar a *view*.

O objetivo dessa *view* é fazer um *join* entre as tabelas *Cax_Previsoes* e *Ger_CliFor* para trazer o nome do cliente ou fornecedor junto com os dados de cada previsão. Através do campo *Pre_Previsao*, a *view* faz a filtragem trazendo apenas as previsões ativas no sistema.

Nota: Uma *view* nada mais é que uma "visão" personalizada de informações contidas em uma ou mais tabelas do banco de dados. Podemos dizer que uma *view* seria o equivalente a uma instrução SQL que fica armazenada dentro do banco. Em alguns casos, através de uma *view*, podemos incluir, alterar e excluir registros como se fosse uma tabela. A *view* não faz duplicação dos registros, isso é, os dados por ela retornados não ficam armazenados fisicamente no banco.

Listagem 7. Criando uma view no banco de dados

```
CREATE VIEW VW_PREVISOES (
  CODIGO,
  CF_CODIGO,
  CF_TIPO,
  NOME_CLIFOR,
  DESCRICAO,
  EMISSAO,
  VENCIMENTO,
  VALOR,
  CREDEB)
AS
SELECT P.PRE_CODIGO, P.CF_CODIGO, CF.CF_TIPO,
  CF.CF_NOME, P.PRE_DESCRICAO, P.PRE_DATAEMISSAO,
  P.PRE_DATAVENCIMENTO, P.PRE_VALOR, PRE_CREDEB
FROM CAX_PREVISOES P
INNER JOIN GER_CLIFOR CF ON P.CF_CODIGO = CF.CF_CODIGO
WHERE P.PRE_PREVISAO = 'S';
```

Inicia construção do sistema no Delphi

Após a definição dos objetos e regras de negócio no banco de dados IB/FB, já podemos iniciar o desenvolvimento da aplicação Delphi.

Nota: Neste artigo foi utilizado o Delphi 2005 Win32 para a criação da aplicação. Você pode utilizar sem problemas o Delphi 7 nessa etapa, visto que o mesmo também possui a tecnologia dbExpress para acesso a dados e o Rave Reports para geração de relatórios.

Crie uma nova aplicação no Delphi 2005 (*File|New>VCL Forms Application – Delphi for Win32*) ou Delphi 7 (*File|New>Application*). Altere o nome do *Form1* para "FrmPrincipal" e salve a unit como "untFrmPrincipal.pas".

Salve o arquivo de projeto como "SysCash.dpr". Crie um DataModule e altere seu nome para "DMPrincipal" e salve a unit como "untDMPrincipal.pas". Adicione um *SQLConnection* (*dbExpress*) e configure-o para acessar o banco *SYSCASH.CD*.

Nota: Utilizaremos por enquanto o usuário *SYSDBA* fixo dentro do componente para acessar a base de dados IB/FB.

Criando o formulário base

Antes de criar as telas de cadastros de previsões e lançamentos, vamos construir um formulário que servirá de base para as demais telas. Nesse formulário iremos definir o layout padrão que será utilizado pelas telas do sistema, além de armazenar as rotinas comuns utilizadas por elas.

Crie um novo formulário, altere seu nome para "FrmCadastro" e salve a unit como "untFrmCadastro.pas". Adicione alguns componentes visuais e não-visuais e configure-os como mostrado nas **Figura 1** e **2**.

A tela base do sistema será utilizada tanto para manipulação dos dados como também para a consulta dos mesmos. Utilizaremos componentes do tipo *SQLStoredProc* para acessar as SPs armazenadas na base de dados, responsáveis pelas operações básicas de inclusão, alteração e exclusão de cada tabela (chamaremos de *spInserir*, *spAlterar* e *spApagar*).

O componente *cdsConsultar* será responsável por passar as instruções SQL da aplicação cliente para a base de dados, afim de obter o resultado das consultas efetuadas pelo usuário na aba *Consulta* do formulário.

Nota: No formulário base não haverá nenhum componente vinculado a um objeto do banco de dados, isso será feito nos formulários que herdarão as características do formulário base, os formulários descendentes. Assim, não será realizada nenhuma ligação dos componentes do formulário base com o *SQLConnection* do DataModule.

Nesse formulário podemos adicionar também métodos básicos que serão utilizados por todos os formulários descendentes. Métodos para controle dos botões e campos, por exemplo, são ótimos candidatos para serem incluídos dentro do formulário base.

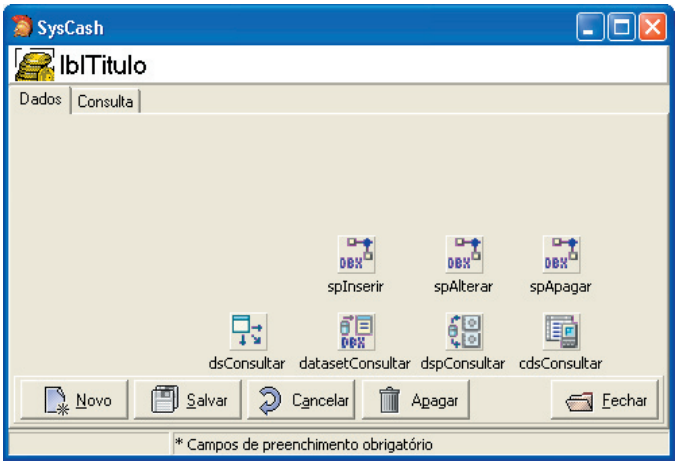


Figura 1. Tela base do sistema para inclusão, alteração e exclusão dos dados

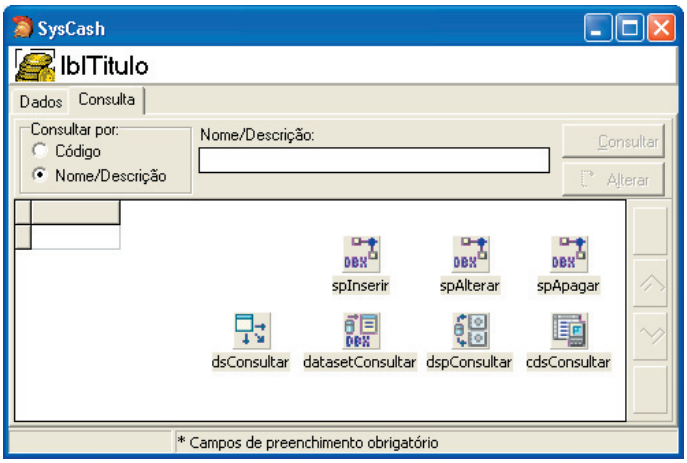


Figura 2. Tela base do sistema para consulta por código, nome ou descrição

Cadastros básicos

O sistema *SysCash* possuirá três telas de cadastro: *Cidades*, *Contas* e *Clientes/Fornecedores*. Todas serão herdadas do formulário base (*FrmCadastro*). Para criar um novo formulário, descendente de *FrmCadastro*, selecione a opção *File|New>Other>Inheritable Items>FrmCadastro* ou *File|New>Other>SysCash>FrmCadastro* no Delphi 7.

Ao criar o formulário, ele trará consigo todos os componentes do formulário base. Para o formulário recém criado, dê o nome de "FrmCadCidade" e salve a unit como "untFrmCidade.pas". Como as operações básicas estão definidas em SPs na base de dados, não há necessidade de utilizarmos componentes *Data-Aware*, como *DBEdit* e *DBComboBox*, por exemplo.

Utilizaremos componentes da paleta *Standard*, como *Edit* e *ComboBox* para manipular os dados dentro dos cadastros. Inclusive, isso fará com que a aplicação fique mais rápida e otimizada, neste caso. O primeiro passo é apontarmos os componentes *SQLStoredProc* da tela de cadastro de Cidades para as SPs, referentes à tabela *Ger_Cidades*.

Nota: Poderíamos usar *Data-Aware* também (como em uma aplicação comum), assim as SPs poderiam ser "acionadas" no evento *BeforeUpdateRecord* do *DataSetProvider*. Veja como realizar essa técnica no artigo de dicas da edição 60.

Indique na propriedade *StoredProcName* dos componentes *spInserir*, *spAlterar* e *spApagar* os valores *Proc_Ins_Cidades*, *Proc_Upd_Cidades* e *Proc_Del_Cidades*, respectivamente.

Verifique os parâmetros das SPs através da propriedade *Params*. Para o *datasetConsultar* digite a instrução SQL da **Listagem 8,** usando a propriedade *CommandText*. Através dessa instrução, podemos adicionar os campos da tabela *Ger_Cidades* dentro do *cdsConsultar* e configurar o *DBGrid* presente na aba *Consulta*.

Listagem 8. CommandText do datasetConsultar

```
select CID_CODIGO, CID_NOME, CID_UF, CID_ATIVO
from GER_CIDADES
where CID_APAGADO <> 'S'
order by CID_NOME
```

Nota: Quando o usuário efetuar uma pesquisa, o *cdsConsultar* substituíra a instrução SQL enviando os parâmetros fornecidos. Para que isso seja possível, devemos alterar a propriedade *Options|poAllowCommandText* do *dspConsultar* para *True*.

Dentro da aba *Dados*, adicione alguns componentes visuais da paleta *Standard* e configure-os de acordo com a **Figura 3**. Para o campo *UF* adicione a lista de estados existentes através da propriedade *Items*. Os botões *Novo* e *Cancelar* serão utilizados apenas para habilitar ou não os componentes para alterações do usuário. As chamadas às SPs serão feitas apenas nos botões *Salvar* e *Apagar*. Veja na **Listagem 9** o código para os botões *Salvar* e *Apagar*.

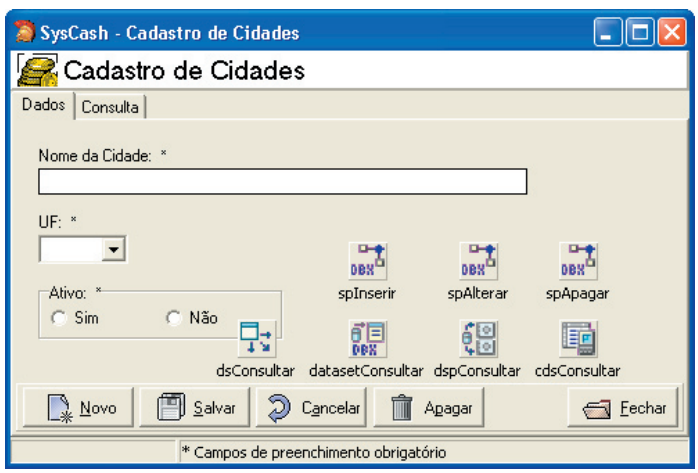


Figura 3. Tela de cadastro de Cidades do sistema SysCash

Listagem 9. Botões Salvar e Apagar do formulário de cadastro de Cidades

```
procedure TFrmCadCidade.btnSalvarClick(
  Sender: TObject);
begin
  { ValidaCampos está implementado no formulário base }
  if not ValidaCampos then
    Exit;
  case strOper of
    'I':
    begin
      with spInserir do
      begin
        Close;
        ParamByName('NOME').AsString := edtNome.Text;
        ParamByName('UF').AsString := cbUF.Text;
        try
          ExecProc;
        except
          on E : Exception do
          begin
            MessageDlg(E.Message, mtError, [mbOk], 0);
            Abort;
          end;
        end;
      end;
    'A':
    begin
      with spAlterar do
      begin
        Close;
        ParamByName('CODIGO').AsInteger :=
          cdsConsultarCID_CODIGO.AsInteger;
        ParamByName('NOME').AsString :=
          edtNome.Text;
        ParamByName('UF').AsString := cbUF.Text;
        ParamByName('ATIVO').AsString :=
          rgAtivo.Items[rgAtivo.ItemIndex][1];
        try
          ExecProc;
        except
          on E : Exception do
          begin
            MessageDlg(E.Message, mtError, [mbOk], 0);
            Abort;
          end;
        end;
      end;
    end;
  end;
  inherited;
end;

procedure TFrmCadCidade.btnApagarClick(Sender: TObject);
begin
  if MessageDlg('Confirma exclusão?', mtWarning,
    [mbYes, mbNo], 0) = mrNo then
    Exit;
  with spApagar do
  begin
    Close;
    ParamByName('CODIGO').AsInteger :=
      cdsConsultarCID_CODIGO.AsInteger;
    ParamByName('DEFINITIVO').AsString := 'N';
    try
      ExecProc;
    except
      on E : Exception do
```

```
      begin
        MessageDlg(E.Message, mtError, [mbOk], 0);
        Abort;
      end;
    end;
  end;
  inherited;
end;
```

No botão *Salvar* utilizamos uma variável (*strOper*) declarada na seção *Protected*, dentro do formulário base, para identificar se o usuário está fazendo uma inclusão ou uma alteração. Dependendo da operação, executamos a SP relacionada. Para o botão *Apagar* utilizamos o *spApagar* para excluir o registro corrente a partir do seu código chave.

Nota: Para o parâmetro *Definitivo* da SP responsável por apagar um registro, passamos o valor "N", indicando que o registro não deverá ser excluído fisicamente da base de dados e sim apenas ter seu *flag* alterado para "T", para que não apareça mais para o usuário dentro do sistema.

Para o botão *Consultar* dentro da aba *Consulta* inclua o código da **Listagem 10**. Através do componente *cdsConsultar* passamos uma instrução SQL para a base de dados, de acordo com a opção e valor informado pelo usuário.

Listagem 10. Botão Consultar

```
procedure TFrmCadCidade.btnConsultarClick(Sender: TObject);
begin
  inherited;
  with cdsConsultar do
  begin
    Close;
    CommandText := 'select CID_CODIGO, CID_NOME, '+
     'CID_UF, CID_ATIVO from GER_CIDADES';
    case rgConsultar.ItemIndex of
      0: CommandText := CommandText +
         ' where CID_CODIGO = ' + edtConsultar.Text;
      1: CommandText := CommandText +
         ' where UPPER(CID_NOME) like ' + QuotedStr(
         AnsiUpperCase(edtConsultar.Text) + '%');
    end;
    CommandText := CommandText +
      ' and CID_APAGADO <> ' + QuotedStr('S') +
      ' order by CID_NOME';
    Open;
    if IsEmpty then
    begin
      MessageDlg('Nenhum registro encontrado!' + #13 +
       'Refaça a pesquisa', mtInformation, [mbOk], 0);
      if edtConsultar.CanFocus then
        edtConsultar.SetFocus;
    end;
  end;
end;
```

Para criar os cadastros de *Contas* e *Clientes/Fornecedores* utilize os mesmos conceitos, criando os formulários como descendentes do formulário base. Para o cadastro de *Contas*, mantenha o campo *Con_SaldoAtual* como somente-leitura ou utilize um *Label* para mostrar seu valor para o usuário do sistema. Esse campo não deve ser alterado pelo usuário, pois seu valor é gerenciado pelo sistema através de *triggers* e SPs.

Antes de criar a tela de cadastro de *Clientes/Fornecedores*, vamos criar mais um DataModule, que será responsável em armazenar os *DataSets* para pesquisas que serão utilizados nas próximas telas. Altere o nome do novo DataModule para "DMPesquisa" e salve sua unit como "untDMPesquisa.pas". Adicione alguns componentes de acesso a dados como mostra a **Figura 4**.

Para o *datasetContas* digite a instrução SQL da **Listagem 11**.

O código da listagem anterior seleciona apenas as contas ativas e que não estejam "apagadas". Insira a instrução SQL equivalente para os demais *DataSets*. Na construção do cadastro de *Clientes/Fornecedores* utilize o *DataSet* de pesquisa *cdsCidades* para mostrar as cidades e armazenar seu código no campo *Cid_Codigo* da tabela *Ger_CliFor*.

Listagem 11. Código para selecionar apenas as contas ativas

```
CON_CODIGO, CON_DESCRICAO
from CAX_CONTAS
where CON_ATIVO = 'S'
and CON_APAGADO <> 'S'
order by CON_DESCRICAO
```

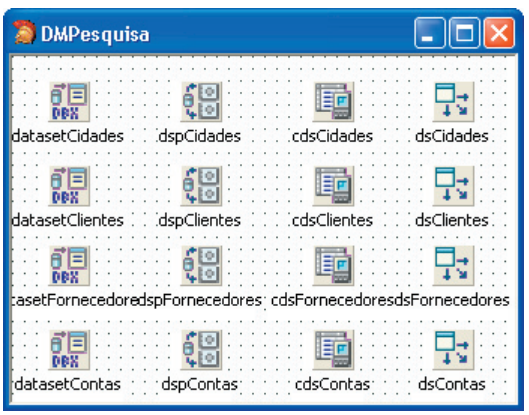


Figura 4. DataModule para armazenamento dos DataSets de pesquisas

Nota: Apesar de termos indicados todos os passos e códigos fundamentais para a construção do exemplo, você pode optar por fazer o download do código-fonte completo e das telas de cadastro do sistema, a partir do endereço indicado no final do artigo.

Previsões e Lançamentos

Como as regras de negócio estão armazenadas na base de dados, as telas de previsões e lançamentos funcionam como uma espécie de tela de "cadastro", contendo apenas métodos e operações básicas.

Para a construção da tela de previsões (contas a pagar/receber) siga os mesmos conceitos utilizados nas telas de cadastros básicos.

Crie um novo formulário herdando do formulário base e adicione os componentes *Standard* para a manipulação dos dados da tabela *Cax_Previsoes*.

Nota: Para preencher a lista, onde o usuário fará a seleção do *Cliente/Fornecedor*, utilize os *DataSets* de pesquisa *cdsClientes* e *cdsFornecedores*.

Os lançamentos no *SysCash* poderão ser feitos de duas maneiras diferentes, através da baixa de uma previsão cadastrada ou de forma avulsa, como um lançamento não previsto pelo usuário. A tela de lançamentos avulsos também possuirá a mesma estrutura das demais telas, pois todas as suas regras de negócio estão armazenadas dentro de *triggers* e SPs no banco de dados IB/FB. Crie um novo formulário descendente de *FrmCadastro* e adicione os componentes e código para manipulação da tabela *Cax_Lancamentos*, seguindo os passos descritos para os cadastros anteriores.

A tela de lançamento de baixas será a única tela de cadastro/movimentação do sistema que não será herdada do formulário base. Esse formulário possuirá duas abas, onde a primeira mostrará as previsões cadastradas e ativas no sistema, enquanto que a segunda aba será utilizada para efetuar a baixa da previsão ou alteração do lançamento depois de baixado.

Crie um novo formulário, altere seu nome para "FrmBaixa" e salve a unit como "untFrmBaixa.pas". Adicione alguns componentes visuais para a aba *Previsões* e configure-os de acordo com a **Figura 5**.

Através do botão *Consultar*, o usuário pode informar o código da previsão a ser baixada e selecioná-la através do *Grid*. Com o botão *Baixar* é possível iniciar a baixa da previsão em questão. Para o *datasetPrevisoes* digite a seguinte instrução SQL:

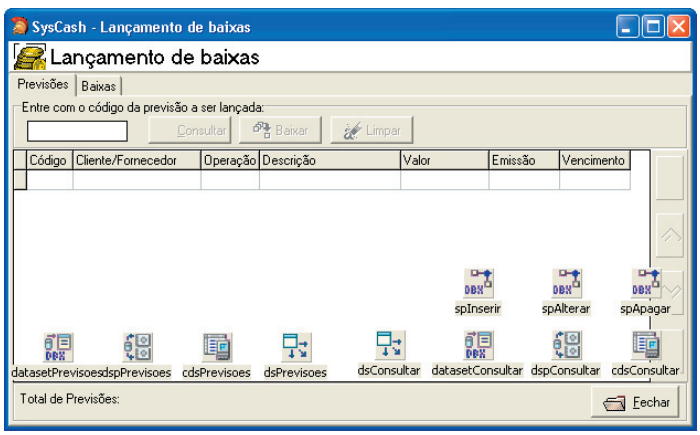


Figura 5. Previsões ativas para baixa no sistema a partir da tela de lançamento de baixas

```
select * from VW_PREVISOES
order by CODIGO
```

Utilizamos aqui a *view Vw_Previsoes* criada no banco, para trazer apenas as previsões ativas e ainda não baixadas no sistema. No evento *OnClick* do botão *Consultar* digite o código da **Listagem 12**.

Listagem 12. Botão Consultar para as previsões ativas e não baixadas

```
procedure TFrmBaixa.btnConsultarClick(Sender: TObject);
begin
  with cdsPrevisoes do
  begin
    Close;
    CommandText := 'SELECT * from VW_PREVISOES '+
      ' where CODIGO = '+ edtCodPrevisao.Text +
      ' order by CODIGO';
    Open;
    if IsEmpty then
    begin
      MessageDlg('Previsão não encontrada!',
        mtInformation, [mbOk], 0);
      Close;
      CommandText := 'SELECT * from VW_PREVISOES '+
        ' order by CODIGO';
      Open;
      edtCodPrevisao.SetFocus;
      Exit;
    end;
    lblPrevisoes.Caption := 'Total de Previsões: ' +
      IntToStr(RecordCount);
    if btnBaixar.CanFocus then
      btnBaixar.SetFocus;
  end;
end;
```

Utilizando o *cdsPrevisoes* refazemos a instrução SQL em cima da *view Vw_Previsoes*, passando o código informado pelo usuário para a pesquisa. Através do botão *Baixar*, fazemos a passagem dos dados da previsão selecionada para os campos na aba *Baixas*. Adicione alguns componentes visuais dentro da aba *Baixas* e configure-os como mostra a **Figura 6**.

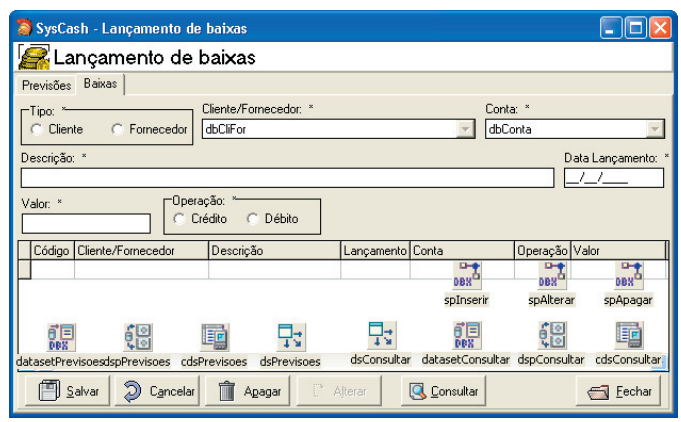


Figura 6. Efetuando a baixa de uma previsão no sistema

O código utilizado no evento *OnClick* dos botões da aba *Baixas* são equivalentes aos utilizados nas telas de cadastros básicos. Para os itens *Cliente/Fornecedor* e *Conta* utilize os *DataSets* de pesquisa disponíveis no *DMPesquisa* para preencher a lista de opções.

Antes de fazermos a chamada às telas criadas a partir do formulário principal, precisamos alterar o modo de criação dos formulários. Selecione a opção *Project|Options* a partir do menu principal do Delphi para abrir a janela de opções do *SysCash*. Na aba *Forms*, mantenha o *DMPrincipal* e *DMPesquisar*, além do formulário *FrmPrincipal* dentro

da seção *Auto-create forms* (certifique-se que o os objetos estejam nessa ordem). Mova os demais formulários para a seção *Available forms*, pois faremos a criação dos mesmos em tempo de execução.

Fazendo a chamada aos formulários

Selecione o *FrmPrincipal* e adicione um *MainMenu*. Crie os menus e seus itens com a seguinte estrutura:

- Arquivo
  - Caixa Sobre
  - Sair
- Cadastros
  - Clientes/Fornecedores
  - Cidades
  - Contas
- Caixa
  - Previsões (Conta a pagar/receber)
  - Lançamentos
    - Avulsos
    - Baixas

Para fazer as chamadas às telas do sistema vamos criar um método denominado *CriarForm*. Esse método será responsável em criar e mostrar o formulário a partir do nome de sua classe. Digite o código da **Listagem 13** para criar o método *CriarForm*.

Listagem 13. Método CriarForm

```
procedure TFrmPrincipal.CriarForm(
  const ClasseForm: string);
var
 C: TFormClass;
 Objeto:TObject;
begin
  Objeto := nil;
  C := TFormClass(FindClass(ClasseForm));
  if not Assigned(Objeto) then
    Objeto := C.Create(nil);
  try
    if (Objeto is TForm) then
      (Objeto as TForm).ShowModal;
  finally
    FreeAndNil(Objeto);
  end;
end;
```

Para que possamos utilizar o método *FindClass* dentro do método *CriarForm*, devemos registrar as classes dos formulários utilizados na aplicação. Para isso utilize o método *RegisterClasses* dentro da seção *Initialization*, com o seguinte código:

```
{ adicione no uses as units referentes
  a cada formulário }
initialization
  RegisterClasses([TFrmCadCidade, TFrmCadCliFor,
    TFrmCadConta, TFrmCadPrevisao, TFrmCadAvulso,
    TFrmBaixa]);
```

Para o evento *OnClick* do menu *Cadastros|Cidades*, por exemplo, utilize o seguinte código, onde iremos criar e mostrar a tela de cadastro de Cidades:

```
CriarForm('TFrmCadCidade');
```

Faça a mesma chamada para as demais telas do sistema. Compile e salve a aplicação. Nesse momento já podemos rodar e testar as telas de cadastro, previsões e lançamentos do sistema. Adicione algumas Cidades, Contas e Clientes/Fornecedores para que possa testar as telas de previsões e lançamentos.

Após cadastrar algumas previsões e lançamentos no sistema, abra o banco de dados através do *IBExpert* (ou algum outra ferramenta de administração do IB/FB) e verifique o saldo das contas, tabela de histórico e saldos diários gravados na tabela *Cax_Saldos*. Todas as regras referentes ao controle de saldos e informações do sistema são realizadas através das *triggers* e SPs definidas dentro da base de dados.

Conclusões

Nessa segunda parte da série, definimos as *triggers* e SPs com as regras de negócio que serão utilizadas pelo *SysCash*. Iniciamos também a implementação da aplicação Delphi criando as telas de cadastros básicos, previsões e para os lançamentos avulsos e de baixas.

No próximo artigo criaremos os relatórios de fluxo de caixa, movimentação diária e por período, além do relatório de histórico utilizando SPs para seleção dos dados. Criaremos ainda os usuários para acesso ao sistema e banco de dados e aplicaremos as permissões de acesso aos objetos armazenados na base IB/FB. Um abraço e até a próxima edição!

Everson Borges Volaco *(everson@rhealeza.com.br) é desenvolvedor e instrutor certificado Borland, com experiência em aplicações cliente/servidor, usando Delphi, InterBase e Oracle. Possui três certificações oficiais: Borland Delphi 7.0, Borland CaliberRM 6.0 e Borland StarTeam 6.0.*

Download:

www.devmedia.com.br/clubedelphi/downloads

Segurança de aplicações

Luciano Pimenta

Gerenciando usuários, permissões e restrições

Todo sistema comercial possui algum tipo de restrição a usuários, por exemplo, não permitir a alteração de determinadas informações ou inclusão de novos registros. É fundamental que cada desenvolvedor saiba como aplicar regras de controle em suas aplicações, visando garantir a integridade e segurança das informações apresentadas ao usuário.

Neste artigo apresentarei um exemplo que construí utilizando técnicas avançadas e interessantes, para aumento da segurança da aplicação e controle de permissões. O sistema permite, por exemplo, que você diga que um determinado usuário só pode alterar registros da tabela de *empregados*, mas não incluir ou excluir. Além disso, será possível dizer, por exemplo, que um determinado usuário não pode visualizar um campo em específico (o salário de um empregado, por exemplo). Para facilitar o entendimento dos passos indicados aqui e acompanhar as técnicas apresentadas, sugiro que o leitor faça download do projeto completo a partir do endereço deste artigo.

Criando as tabelas

Primeiramente, precisamos criar as tabelas que armazenarão as permissões dos usuários, bem como os tipos de permissões que vamos ter na aplicação. Na **Listagem 1** temos o código para criação de três tabelas no banco *Employee.fdb* do Firebird (você pode utilizar o *Employee* do InterBase, bem como qualquer outro banco de dados). Criamos as tabelas *Usuarios*, *Permissoes* e *Permissoes_Usuarios*.

A lógica consiste em cadastrar usuários e permissões, assim o usuário só poderá realizar determinadas tarefas, como incluir, excluir e salvar um registro em um determinado formulário (*Nome_Form*) se estiver cadastrado na tabela. Se o usuário não tiver nenhuma permissão no formulário, o mesmo não poderá nem mesmo abri-lo.

Adicione alguns registros na tabela *Usuarios* e dê algumas permissões ao mesmo. Criaremos dois formulários na aplicação, chamados *Customer* e *Employee*.

Listagem 1. Script para criação das tabelas no banco

```
CREATE TABLE USUARIOS (
    ID             INTEGER NOT NULL,
    USER_NAME      VARCHAR(15) NOT NULL,
    SENHA          CHAR(6) NOT NULL,
    ADM            CHAR(1) NOT NULL,
    LEMBRAR_SENHA  CHAR(1));

ALTER TABLE USUARIOS
ADD CONSTRAINT PK_USUARIOS
PRIMARY KEY (ID);

CREATE TABLE PERMISSOES (
    ID_PERMISSAO  INTEGER NOT NULL,
    PERMISSAO     VARCHAR(30) NOT NULL);

ALTER TABLE PERMISSOES
ADD CONSTRAINT PK_PERMISSOES
PRIMARY KEY (ID_PERMISSAO);

CREATE TABLE PERMISSOES_USUARIOS (
    ID             INTEGER NOT NULL,
    ID_USER        INTEGER NOT NULL,
    ID_PERMISSAO   INTEGER NOT NULL,
    NOME_FORM      VARCHAR(30) NOT NULL);

ALTER TABLE PERMISSOES_USUARIOS
ADD CONSTRAINT PK_PERMISSOES_USUARIOS
PRIMARY KEY (ID);
```

Criando a aplicação

Crie uma nova aplicação no Delphi, adicione um DataModule e salve os arquivos em um diretório.

Nota: Tomarei por base que o leitor sabe criar uma conexão com a tecnologia dbExpress. Em caso de dúvida, vários artigos já foram publicados na revista e no site da ClubeDelphi (*www.clubedelphi.net*). Caso queria, pode utilizar outra tecnologia de acesso a dados.

No DataModule, criaremos uma conexão dbExpress com o banco *Employee.fdb* (ou ao banco que você criou as tabelas). Adicione um trio de componentes (*SQLDataSet* + *DataSetProvider* + *ClientDataSet*) para acessar as tabelas *Customer* e *Employee* (um trio para cada tabela). Adicione mais um trio para acessar a tabela *Permissoes_Usuarios*, utilizando a seguinte instrução SQL na propriedade *CommandText* do *SQLDataSet*:

```
select PU.ID, PU.ID_USER, PU.ID_PERMISSAO,
  PU.NOME_FORM, PERM.PERMISSAO
from PERMISSOES_USUARIOS PU
inner join PERMISSOES PERM on
  (PERM.ID_PERMISSAO = PU.ID_PERMISSAO)
where ID_USER=:ID_USER
```

Finalmente, adicione o último trio de componentes, para acessar a tabela *Usuarios*, utilizando a seguinte instrução SQL:

```
select * from USUARIOS where ID_USER=:ID_USER
```

Configure a propriedade *Params* de ambos os *SQLDataSets* anteriores com o *DataType* do tipo *ftInteger*. Clique de direita nos *ClientDataSets* (os dois últimos inseridos) e escolha a opção *Fetch Params*. Seu DataModule deve estar semelhante ao da **Figura 1**.

Criamos também alguns métodos nesse DataModule, para recuperar dados e filtrar registros, como: *VerificaSenha, FiltraPermUser* etc. Você pode fazer o download do projeto completo e do BD a partir do endereço para download deste artigo.

Adicionamos também um trio de componentes para fazer as pesquisas necessárias no banco, passando as instruções SQL diretamente no *ClientDataSet*. No formulário principal, foram adicionados alguns componentes: barra de ferramentas, barra de status, menus etc. Para os menus e botões, utilizamos um *ActionList* afim de centralizar os códigos, assim, o código do botão para o cadastro de clientes é o mesmo para o menu que chama o cadastro de clientes.

Essa técnica facilita o controle de acesso, pois se o usuário não tiver permissão de acesso ao formulário, não precisaremos desabilitar o menu e botão, mas somente a *Action* vinculada a ele. Veja na **Figura 2** a tela principal da aplicação.

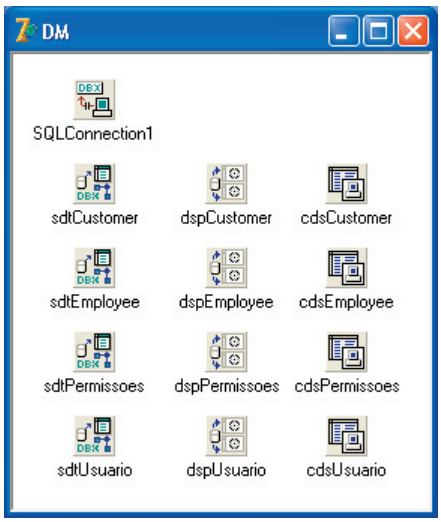


Figura 1. DataModule da aplicação

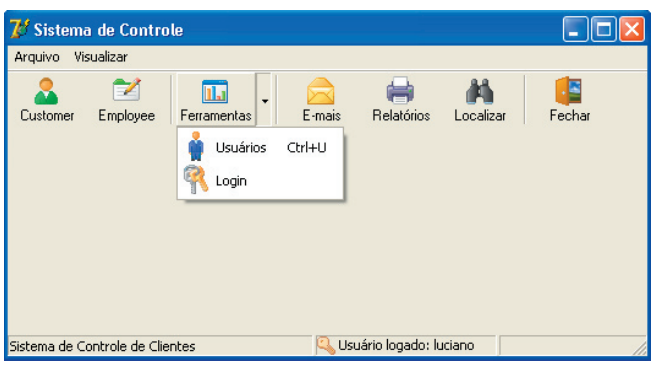


Figura 2. Tela principal da aplicação

Tela de Login

Criaremos uma tela de login, para que o usuário possa entrar no sistema e para filtrar as permissões do mesmo. Na **Figura 3**, temos a tela de login do sistema. Não colocamos nessa tela nenhum tipo de componente para acesso à tabela *Usuarios*. Para mostrar os nomes dos usuários, criamos uma função no DataModule (*PreencheCombo*) que retorna esses dados e preenchemos assim o *ComboBox*.

Poderíamos utilizar um *DBLookupComboBox* para mostrar os usuários (seria até mais fácil), mas é interessante "centralizar" tudo no DataModule. O código da **Listagem 2** mostra o evento *OnClick* do botão de login. Nesse código verificamos em *VerificaSenha* (que retorna um *Boolean*), se a senha cadastrada confere com o valor digitado no *Edit*.

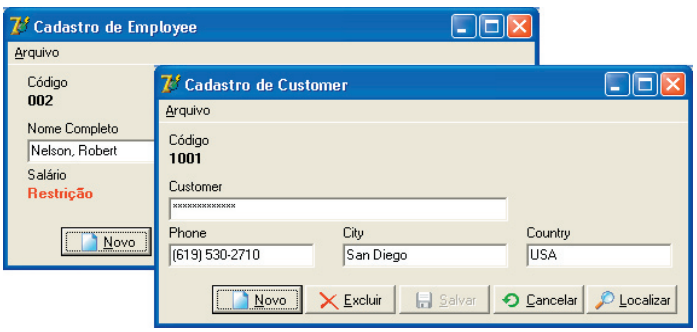


Figura 3. Tela de login do sistema

Listagem 2. Código do botão de Login

```
var
  aIdUser: integer;
begin
  if DM.VerificaSenha(cboUser.Text, EdtSenha.Text,
    aIdUser) then
  begin
    Application.CreateForm(TfrmPrincipal,
      frmPrincipal);
    try
      frmPrincipal.IdUser := aIdUser;
      Release;
      frmPrincipal.ShowModal;
    finally
      frmPrincipal.Free;
    end;
  end
  else
  begin
    EdtSenha.Text := '';
    raise Exception.Create('Senha inválida');
  end;
```

Note que na função temos três parâmetros, para o nome do usuário (que faz parte da cláusula *where* do *select*), a senha digitada e um código do usuário. Por que o código do usuário? Na verdade esse parâmetro é de retorno, ele trará o valor do código do usuário, para podermos filtrar as suas respectivas permissões.

Esse parâmetro preencherá a propriedade *IdUser*, criada no formulário principal. Ao ser alterada (é uma propriedade que possui um

método *Set* vinculado a ela), chamamos o método *FiltraPermUser* do DataModule e também habilitamos/desabilitamos os botões de cadastro.

Utilizamos o método *Locate* do *cdsPermissoes* para verificar se o usuário tem alguma permissão em relação a um formulário, como no código a seguir:

```
begin
  DM.FiltraPermUser(Value);
  actCustomer.Enabled := DM.cdsPermissoes.Locate(
    'NOME_FORM', 'Customer', []);
  actEmployee.Enabled := DM.cdsPermissoes.Locate(
    'NOME_FORM', 'Employee', []);
  ...
end;
```

Assim, se o usuário não possuir nenhuma permissão para o formulário de clientes, o mesmo não será habilitado. Veja que estamos codificando a *Action* vinculada ao menu e botão do formulário de clientes, de forma que não teremos duplicação de código na aplicação.

Permissões no cadastro

Criamos dois formulários de cadastro para as tabelas *Employee* e *Customer*. Na **Figura 4** temos as duas telas para você visualizar como os formulários foram criados. Note que novamente estamos utilizando um *ActionList* para os botões. Para habilitar/desabilitar os botões, utilizamos a mesma técnica do exemplo anterior, só que agora estamos localizando o formulário e o tipo de permissão. No evento *OnShow* do formulário temos o seguinte código:

```
procedure TfrmClientes.FormShow(Sender: TObject);
begin
  actNovo.Enabled := DM.cdsPermissoes.Locate(
    'NOME_FORM;PERMISSAO', VarArrayOf(['Customer',
    'Novo']), []);
  actExcluir.Enabled := DM.cdsPermissoes.Locate(
    'NOME_FORM;PERMISSAO', VarArrayOf(['Customer',
    'Excluir']), []);
...
end;
```

Assim, se o usuário logado tiver permissão, por exemplo, de adicionar um novo registro, o botão *Novo* será habilitado.

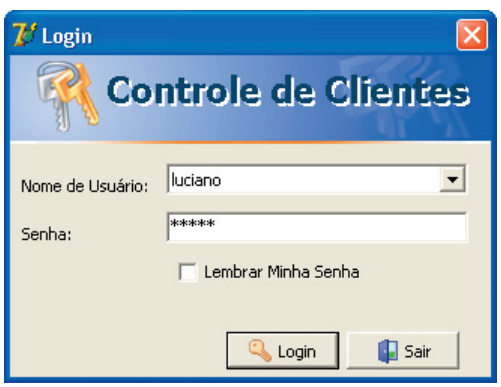


Figura 4. Telas de cadastro

Restrições em campos

Vamos implementar agora uma dica bem interessante, onde iremos restringir a visualização de determinado campo para o usuário logado. Por exemplo, imagine que o seu cliente pediu que um usuário não visualizasse o valor do salário dos funcionários cadastrados (no nosso exemplo o campo *Salary* da tabela *Employee*).

Primeiramente, vamos criar uma nova tabela no banco, que indicará quais os campos restritos para o usuário. Na **Listagem 3** temos o script da tabela. Adicione no DataModule mais um trio de componentes e utilize a seguinte instrução SQL para o *SQLDataSet*:

```
select * from RESTRICAO where ID_USER=:ID
```

Nos formulários, vamos implementar um método que irá varrer o formulário e verificar, através da propriedade *DataField* dos componentes *DBEdit*, se o campo está na tabela de restrições. Na **Listagem 4** temos o código completo.

Veja que estamos desabilitando o *DBEdit* e também indicando uma máscara para o campo, caso o usuário tenha restrição sobre o campo. Poderíamos também "esconder" o componente, colocando, por exemplo, um rótulo indicativo de restrição de campo.

Também poderíamos manipular diretamente o *DataSet*, para utilizar as propriedades do *TField*, como *EditMask*, *ReadOnly* etc. Na **Figura 5** você pode ver um exemplo utilizando o rótulo de restrição e a manipulação diretamente no *TField*.

Listagem 3. Script para a criação da tabela de Restrições

```
CREATE TABLE RESTRICAO (
  ID            INTEGER NOT NULL,
  ID_USER       INTEGER NOT NULL,
  TABELA        VARCHAR(20) NOT NULL,
  CAMPO         VARCHAR(20) NOT NULL);


ALTER TABLE RESTRICAO
ADD CONSTRAINT PK_RESTRICAO
PRIMARY KEY (ID)
```

Listagem 4. Método para restrição de campo

```
procedure TfrmClientes.Restricoes;
var
  i: integer;
  aCampo: string;
begin
  for i := 0 to ComponentCount - 1 do
  begin
    if Components[i] is TDBEdit then
    begin
      aCampo := (Components[i] as TDBEdit).DataField;
      if DM.cdsRestricoes.Locate('TABELA;CAMPO',
        VarArrayOf(['CUSTOMER', aCampo]), []) then
      begin
        (Components[i] as TDBEdit).Enabled := False;
        (Components[i] as TDBEdit).PasswordChar := '*';
      end;
    end;
  end;

end;
```

IT Brasil

14ª Feira e Congresso Internacional de Tecnologia e Eletrônicos

www itbrasilexpo.com.br

Um evento focado no Canal de TI e em empresas SMB, onde 84% dos visitantes desejam adquirir produtos e tecnologias. E então, vai ficar de fora?

ÉPOCA

O melhor marketplace de TI da América Latina mudou de nome. Agora se chama IT BRASIL.

Chegando em sua 14ª edição, a principal feira de TI apresenta seus novos formato e nome, cheia de novidades. Visite nosso site e saiba muito mais: www.itbrasilexpo.com.br

16 a 19 de agosto de 2005
Pavilhão de Exposições do Anhembi
São Paulo - SP - BRASIL

Filiada: UBRAFE União Brasileira dos Promotores de Feiras

Transportadora aérea oficial: TAM

Apoio: anhembi

Promoção e Organização: Messe Frankfurt Feiras

Tel.: 11 4688.6000
www.messefrankfurtfeiras.com.br

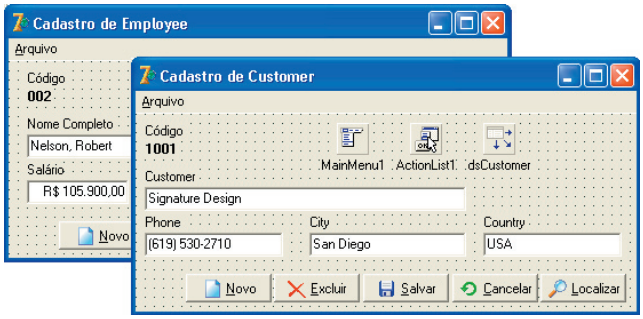


Figura 5. Mostrando um rótulo de restrição ou trabalhando diretamente no DataSet

Cadastro de Usuário/Permissões/Restrições

O cadastro de usuário possui os dados do usuário, bem como as permissões e restrições, sendo que somente o usuário que tem permissão de *Administrador* pode incluir um novo registro (usuário, permissões e restrições). Na **Figura 6** temos a tela do cadastro de usuários.

Nas abas *Permissões* e *Restrições de Campo*, temos as respectivas Permissões e Restrições do usuário logado, onde o mesmo (se for administrador) pode adicionar/excluir. Na **Figura 7** temos as telas para adicionar restrições e permissões.

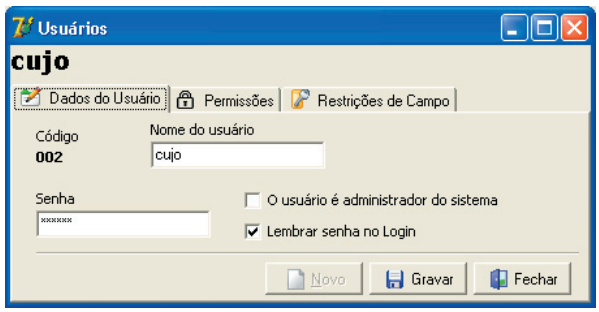


Figura 6. Cadastro de Usuários

Conclusões

As dicas apresentadas aqui mostram o "caminho" para a criação de um controle de permissões e segurança, pois uma ferramenta desse tipo varia muito de acordo com o sistema ou a necessidade do usuário. Se você deseja algo mais específico para a sua aplicação, com este artigo, você terá condições de criar o que deseja.

Poderíamos ainda melhorar este exemplo, colocando criptografia nas senhas, criar grupos de permissões, um controle para o Administrador adicionar/remover usuários, permissões e restrições etc. Quem quiser enviar sugestões de melhoria, fique a vontade. Publicarei aqui na revista ou no site da ClubeDelphi (*www.clubedelphi.net*) as alterações enviadas pelos leitores! Forte abraço e até a próxima!

Figura 7. Telas para adicionar restrições e permissões do Usuário

Luciano Pimenta *(lucianopimenta@clubedelphi.net) é acadêmico do curso de Sistemas de Informação (Unifra), Técnico em Processamento de Dados, Editor Técnico da Revista ClubeDelphi e Editor do Portal ClubeDelphi.NET (www.clubedelphi.net).*

Download:
www.devmedia.com.br/clubedelphi/downloads

Thiago, programador de Web,
agora também hospeda
os sites dos seus clientes.

Revenda LocaWeb.

A LocaWeb é a melhor empresa de hospedagem de sites do Brasil. E agora você pode oferecer toda esta infra-estrutura para seus clientes como se fosse sua. Basta contratar a Revenda LocaWeb e usufruir de tecnologia, segurança e liberdade para que possa administrar como quiser os sites, contas de e-mail, bancos de dados, espaço em disco e outras ferramentas. Você e seus clientes ganham. Hospedagem de sites é o nosso negócio e agora é o seu também.

• Domínios e contas de e-mail ilimitadas • Painel Plesk 7.5 Reloaded • Infra-estrutura no Brasil • Link de 1 Gbps com a Embratel • Servidores com DNS próprio e muito mais.

Acesse www.locaweb.com.br

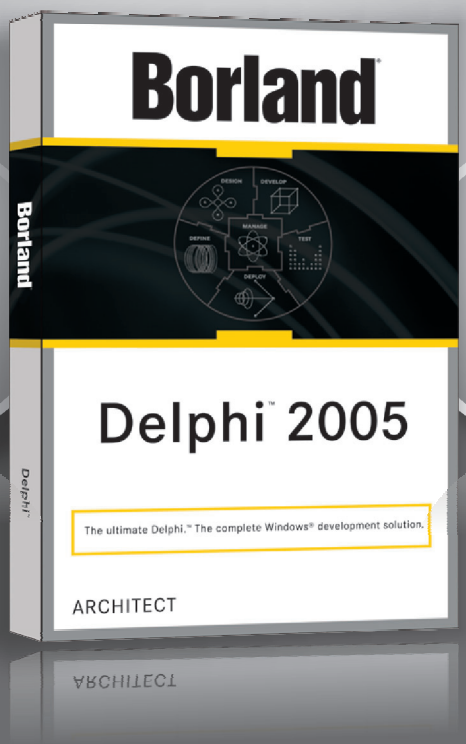
LOCAWEB
HOSPEDAGEM DE SITES

Mais uma vez o Delphi te leva para o futuro

Delphi 10 anos

SCM

WIN32



.NET

C#

O Delphi 2005 traz para você o StarTeam - o SCM (Software Configuration Management) da Borland - que irá economizar tempo de administração, reduzir a necessidade de infra-estrutura adicional, aumentar a comunicação e a segurança de seus recursos, melhorando a produtividade e a qualidade e reduzindo o tempo total do projeto. E mais*:

✓ Aplicações Win32 em Delphi ✓ Multi-Tier com Remoting e ADO .NET ✓ .NET com suporte a Delphi e C#
✓ Teste Unitário ✓ Refactoring ✓ Modelos em UML ✓ Web Services

Visite http://info.borland.com.br/delphi e saiba as últimas novidades sobre o produto

www.borland.com.br

Borland®