

# Teoria sobre Firebird, DBExpress e ClientDataSet



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

### Índice

Firebird .....	3
DBExpress.....	18
SQLConnection .....	20
SQLDataSet .....	28
SQLQuery.....	35
SQLStoredProc .....	38
SQLTable .....	41
SQLMonitor .....	45
SimpleDataSet.....	48
SQLClientDataSet .....	49
ClientDataSet .....	50
DataSetProvider .....	76

# Firebird

## Introdução

Para muitos o nome Interbase/Firebird é totalmente desconhecido, para outros é sinônimo de eficiência e fácil manutenção (fácil ? Bem, na verdade a manutenção de um banco de dados IB/FB é quase 0).

O Interbase não é um produto novo. Ele está no mercado há mais de 12 anos, e nesse tempo adquiriu respeito e admiração de muitos programadores, desenvolvedores e clientes (entre eles podemos citar a NASA, o exército americano, etc...). O Firebird é a continuação do Interbase Open Source e vem sendo desenvolvido por uma comunidade de programadores espalhados por todo o mundo.

Mas então, se o Interbase/Firebird é tão bom, porque ele não é tão reconhecido como o Oracle, o Microsoft SQL Server e outros servidores SQL ? Aparentemente, o maior problema enfrentado pelo Interbase durante todos os anos de sua existência foi a falta de marketing e divulgação adequada por parte da Borland/Inprise/ISC nos meios especializados (revistas, etc...). No entanto, após ter seu código liberado como Open Source a situação começou a mudar rapidamente pois agora as licenças de utilização e distribuição são totalmente FREE ! Isso mesmo, custo 0, de graça !!! Isso quer dizer que você não precisará mais utilizar as famosas (e já mais do que ultrapassadas) base de dados padrão xBase ou Paradox para diminuir o custo do seu cliente. Você vai poder contar com um Banco de dados poderoso, eficiente e seguro e seu cliente não vai precisar pagar nada a mais por isso !

A seguir citarei algumas características que fazem com que o Interbase/Firebird se equipare e também se sobressaia em muitos aspectos aos seus concorrentes.

### Sistema "Multi-Generacional"

Uma das maiores vantagens do Interbase/Firebird sobre seus concorrentes é o uso de um sistema otimista de "concorrência" no acesso ao banco de dados. Seguindo uma linha totalmente diferente da maioria dos bancos de dados, onde você tem bloqueio de páginas inteiras (PAGE LOCKS), o Interbase/Firebird "guarda" várias versões dos registros mantendo assim uma visão consistente dos dados durante uma transação, independente de alguma informação ter sido alterada após a transação ter sido iniciada.

Usando um exemplo prático, geralmente você tem situações onde várias pessoas estão utilizando um mesmo banco de dados ao mesmo tempo. Alguns rodando relatórios, outros inserindo ou alterando registros. Imagine que entre o preview de um relatório na tela e a sua impressão definitiva, um outro usuário altere ou insira uma informação no banco de dados que influa no resultado desse relatório. Nos BD convencionais, com certeza você obterá um relatório impresso diferente do que você viu na tela (preview) pois os dados alterados interferiram no resultado final do relatório. No Interbase/Firebird, através dos diversos tipos de isolamento de transação, você pode ter uma mesma "imagem" dos dados inalterada pelo tempo que quiser, sem impedir que outros usuários continuem acessando ou alterando as informações.

Quando um usuário altera uma informação de um campo em um registro, o Interbase/Firebird cria um novo registro com os campos que tiveram seus dados alterados. Esse registro contém um "timestamp" que permite ao banco de dados saber qual é a informação mais atualizada. O registro contendo os dados anteriores não é descartado enquanto houver uma transação ativa "enchergando-os". Após o registro ser liberado, ele é automaticamente "marcado como lixo", removido ou tem seu espaço re-utilizado pelo banco de dados.

O mecanismo de timestamp também é utilizado para saber se mais de uma pessoa tentou alterar os mesmos dados num determinado período, o que gera automaticamente uma exceção (evento de erro) no BD conhecido como deadlock, garantindo assim a consistência das informações.

### Configuração e Manutenção

Uma outra grande vantagem do Interbase/Firebird é quanto à definição das características físicas do banco de dados. Em muitos servidores SQL, o analista ou DBA deve previamente estimar, definir e preparar um espaço do disco rígido para ser usado pelo BD. Os BDs criados no Interbase/Firebird são arquivos comuns do sistema operacional (geralmente com a extensão GDB), que crescem e diminuem conforme a necessidade, sem que seja necessária a intervenção do DBA ! A manutenção de um banco de dados Interbase/Firebird é praticamente 0, assim como sua configuração.

O Interbase/Firebird automaticamente mantém sua base de dados limpa e consistente através de rotinas automáticas de manutenção (SWEEP) executadas geralmente quando o BD está em "idle" (estado de espera). Cada banco de dados Interbase/Firebird consiste de apenas 1 arquivo... todas as informações, tabelas, índices, etc... ficam armazenados dentro desse arquivo, facilitando muitas as operações de administração e backup dos mesmos. Caso o arquivo ultrapasse o tamanho máximo suportado pelo sistema operacional você pode dividir o banco em múltiplos arquivos, sendo que o gerenciamento desses arquivos é totalmente transparente para o usuário.

### Suporte a domínios

O Interbase/Firebird suporta o uso de domínios na definição de campos. Na verdade, todo campo criado em uma tabela no Interbase/Firebird possui um domínio próprio, criado automaticamente pelo sistema ou definido previamente pelo analista. Com essa tecnologia, fica muito fácil fazer alterações em cascata em campos do mesmo tipo.

Um exemplo : Imagine que no seu banco de dados existam 20 campos do tipo Numérico, definidos com 2 casas decimais. Você pode definir um domínio chamado DINHEIRO, especificando o tipo de informação (Numérica) e o número de casas decimais utilizadas pelo mesmo e atribuir esse domínios aos 20 campos do seu BD. Se no futuro você tiver a necessidade de alterar as casas decimais de 2 para 4, basta você alterar o domínio DINHEIRO e todas as colunas referenciadas por esse domínio serão automaticamente alteradas.

Aqui entra uma outra característica do IB. Quando você altera um domínio, os dados armazenados utilizando esse domínio não são imediatamente convertidos para o novo formato, o que ocasionaria um certo overhead no servidor. Por exemplo : Imagine que você tenha um domínio definindo o tipo VARCHAR(10). Em um dado momento, você necessita alterar o tamanho do campo, de 10 para 20. Você então altera o domínio para VARCHAR(20), mas os dados que já estão gravados só serão convertidos ao novo formato quando forem utilizados de alguma maneira. Observe no entanto, que isso fica transparente ao usuário... você encherá todos os dados como VARCHAR(20) !!

### Multi-plataforma

O Firebird atualmente roda nos sistemas operacionais Windows, Linux (i386), Solaris (Sparc), HP-UX (i386), MacOS X, FreeBSD, Solaris (i386) e AIX. O Interbase foi o primeiro banco de dados profissional a ter uma versão disponível para o Linux.

### Outras características :

- Otimização de Queries. O próprio servidor otimiza uma Query de maneira a buscar os resultados da maneira mais eficiente e rápida possível. Também é possível definir um plano de otimização (PLAN) manualmente.
- Suporte a campos BLOB, que podem conter qualquer tipo de dado, desde textos até objetos gráficos ou binários.
- Suporte a diversos protocolos : Local, TCP/IP, NetBeui, IPX/SPX (Novell).
- Funções definidas pelo usuário : Esse é um recurso muito poderoso do Interbase/Firebird . Através dele, você pode utilizar dentro de seus bancos de dados, funções criadas por você mesmo usando qualquer linguagem que gere DLLs.
- Suporte a múltiplas transações (multi-transacional)
- Acesso nativo através da API (sem necessidade da BDE) através de componentes como o Interbase Objects, FreeIB+, IBX (que acompanha o Delphi 5,6 e 7), ZeosLib, DBExpress, e alguns outros.
- Campos Inteiros de 64 bits : O Interbase/Firebird 6.0 (dialect 3) oferece suporte a campos numéricos inteiros de 64 bits. Ele utiliza uma técnica em que números de ponto flutuante podem ser armazenados no banco de dados como inteiros, removendo assim o risco de erros de arredondamento. Toda a conversão entre inteiros e floats é transparente ao usuário.
- Replicação : O Interbase/Firebird 6.0 oferece suporte à replicação de dados através de componentes de terceiros (que infelizmente, até o momento não são free, mas que tem um custo bem acessível). Visite a página de downloads da Interbase-BR para ver uma lista dos replicadores disponíveis.

### Firebird x Interbase

O Firebird nasceu do Interbase 6.0 (Open Source). Após a Borland abrir o código do Interbase na versão 6.0, ela decidiu que continuaria mantendo uma versão comercial do produto (com o código fechado). Nesse momento, um grupo de pessoas (algumas delas que já trabalhavam com o Interbase dentro da própria Borland) decidiu dar continuidade à versão Open Source criando o Firebird. Hoje temos disponível o Interbase 6.0 (Open Source) que não sofreu mais atualizações por parte da Borland desde o release 6.0.2, o Interbase 6.5 e 7.0 que são produtos comerciais desenvolvidos pela Borland e o Firebird 1.0 (e 1.5alpha) Open Source que vem crescendo e ganhando novos recursos a cada dia.

### Resumo

Como você pode ver, o Interbase/Firebird é muito mais do que um simples banco de dados. Ele oferece uma solução completa para pequenas e médias empresas e possui recursos que competem de igual para igual com outros servidores SQL mais conhecidos, com a vantagem de ser Open Source e grátis !

### Links úteis

#### Interbase-BR

<http://www.interbase-br.com>

#### IBPhoenix

<http://www.ibphoenix.com>

#### Firebird

<http://www.firebirdsql.com>

#### Interbase

<http://www.borland.com/interbase>

**Claudio Valderrama**

<http://www.cvalde.com>

**Lista de discussão em português**

<http://br.groups.yahoo.com/group/interbase-br>

*Texto originalmente escrito por Carlos H. Cantu, consultor especialista em banco de dados Firebird/InterBase 6, criador e mantenedor do site [www.firebase.com.br](http://www.firebase.com.br) (referência nacional do IB/FB) e autor do livro Firebird Essencial - disponível nas melhores livrarias de todo o Brasil, ou através do site da FireBase.*

### Firebird - Instalação

Atualmente o Firebird está na versão 1.5 e pode ser baixado no site [www.firebirdsql.com](http://www.firebirdsql.com). Na versão para Windows temos apenas um instalador no qual optamos entre o Client ou Server durante a instalação.

Demonstraremos passo a passo a instalação da versão Server e em seguida a versão Client

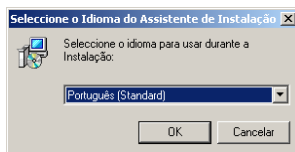
Quando instalamos o Servidor, podemos optar por **Super Server** ou **Classic Server**. Há muitos detalhes técnicos que as diferenciam, por exemplo, a versão **Classic** trabalha baseada em processos, já a versão **Super Server** trabalha baseada em Threads, esta é a versão que promete ser o futuro do Firebird.

Ambas não são diretamente concorrentes, possuem vantagens e desvantagens de acordo com a situação, portanto é interessante consultarmos as diferenças técnicas para sabermos qual se adequará melhor para nós. A documentação está disponível no próprio site do Firebird [www.firebirdsql.com](http://www.firebirdsql.com).

Vale lembrar que estamos utilizando a versão 1.5.2 para demonstração.

### Firebird – Instalação do Server

Execute o instalador (**Firebird-1.5.2.4731-Win32.exe**) e logo será exibida a tela para selecionarmos o idioma.



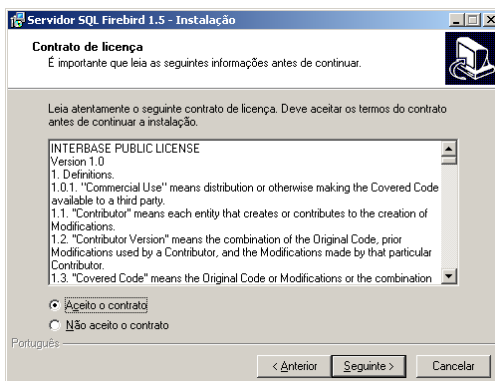
*Instalando Firebird Server - Seleção do Idioma*

Selecione o Idioma Português ou outro de sua preferência e confirme clicando no botão **Ok**.



*Instalando Firebird Server - Apresentação*

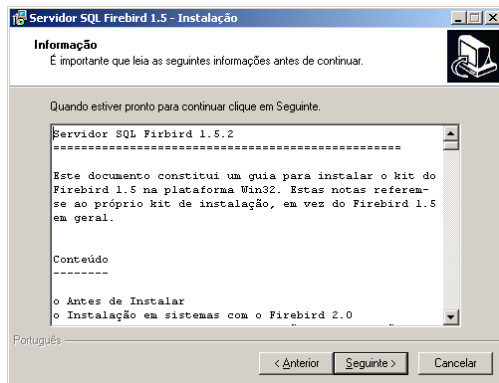
Clique no botão **Seguinte** para prosseguirmos.



*Instalando Firebird Server - Contrato*

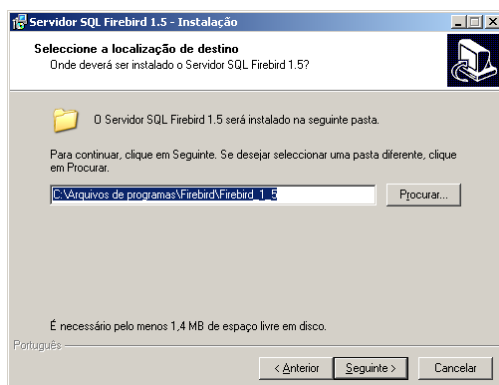
Nesta tela opte por Aceitar o Contrato e prossiga clicando no botão **Seguinte**.





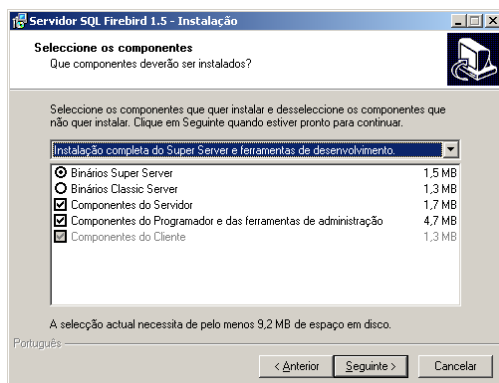
Instalando Firebird Server - Informações sobre a instalação

Aqui temos algumas informações sobre a instalação. Podemos prosseguir clicando no botão **Seguinte**.



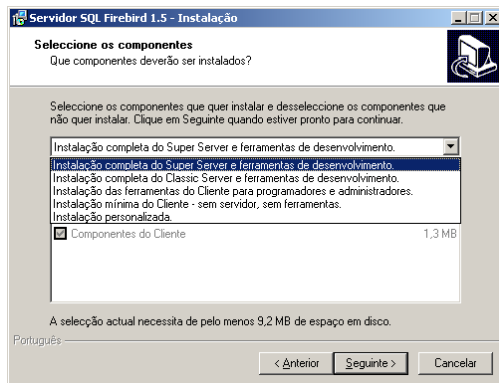
Instalando Firebird Server - Caminho de Destino

Este é o local onde informamos o caminho que será instalado o Servidor. O padrão é C:\Arquivos de programas\Firebird\Firebird\_1\_5. Mantenha este caminho ou altere se preferir outro diretório e clique no botão **Seguinte** para prosseguir.



Instalando Firebird Server - Componentes

Esta é a parte mais importante da instalação, pois é aqui onde informamos entre **Client ou Server** e **Super Server ou Classic Server**.



Instalando Firebird Server - Componentes

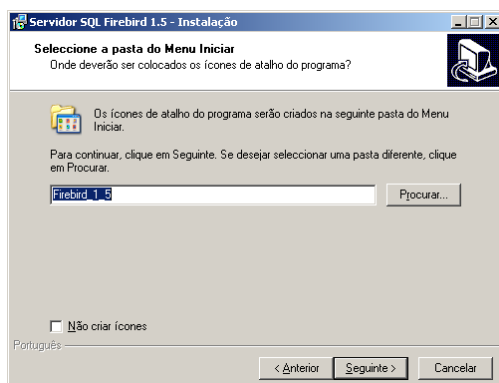
### Tipos de instalação

As duas primeiras opções são para instalação do Server. A primeira é a Super Server e a segunda Classic Server.

A terceira e quarta opção são para as máquinas *Clients*, a terceira é com ferramentas adicionais e a quarta opção é a mais simples, instala apenas a biblioteca para acesso ao servidor de banco de dados.

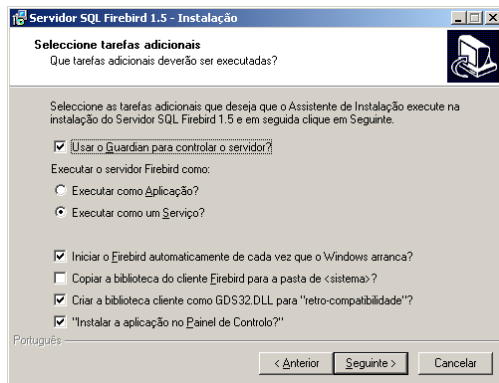
A última opção é a personalizada, na qual podemos seleccionar os componentes manualmente ao invés de utilizarmos as opções já disponíveis.

Em nosso exemplo, como estamos instalando o Server, utilizaremos a versão **Super Server** que é a mais recomendada. Em seguida podemos confirmar clicando no botão **Seguinte**.



Instalando Firebird Server - Pasta para os ícones do programa

Nesta tela devemos informar o nome da Pasta onde serão criados os ícones do Firebird. Depois de informado, clique no botão **Seguinte**.



Instalando Firebird Server - Tarefas

Este é outro ponto importante também que devemos nos atentar, pois precisamos definir se o Firebird será um serviço ou aplicação, detalhes sobre a biblioteca, entre outros.

### Usar o Guardian para controlar o servidor

Habilita um guardião que fica verificando se o Firebird está no ar, caso caia, ele inicia-o novamente.

### Executar o Firebird como Aplicação ou Serviço

Apenas determina se o Firebird será executado como uma Aplicação ou Serviço (caso o sistema operacional permita).

### Iniciar o Firebird automaticamente de cada vez que o Windows arranca

Determina se o Firebird será iniciado logo que entrar no Windows

### Copiar biblioteca do cliente Firebird para a pasta de <sistema>

Copia a biblioteca cliente (fbclient.dll) para o diretório de sistema. Isso é necessário caso o servidor precise executar alguma aplicação que dependa da biblioteca.

### Criar a biblioteca cliente como GDS32.DLL para “compatibilidade”

O arquivo GDS32.DLL é a biblioteca cliente do Interbase e o Firebird utilizava este mesmo nome até antes da versão 1.5. A partir desta, os nomes dos arquivos mudaram e a biblioteca passou a se chamar FBCLIENT.DLL.

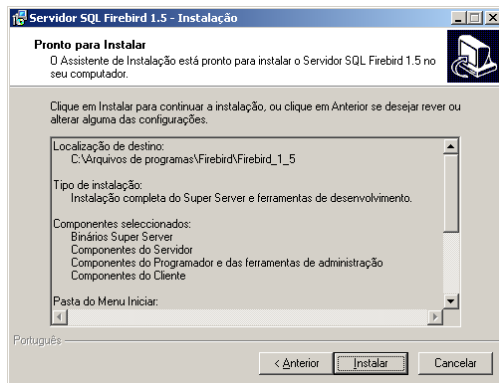
Portanto, para manter compatibilidade com as aplicações que dependam desta biblioteca, podemos habilitar esta opção para que o arquivo FBCLIENT.DLL seja replicado com o nome GDS32.DLL.

***Deixaremos esta opção habilitada, pois faremos acesso ao banco de dados utilizando o IBExpert que utiliza esta biblioteca e também o driver DBExpress em nosso projeto, que por padrão, faz uso da mesma.***

### Instalar a aplicação no Painel de Controle

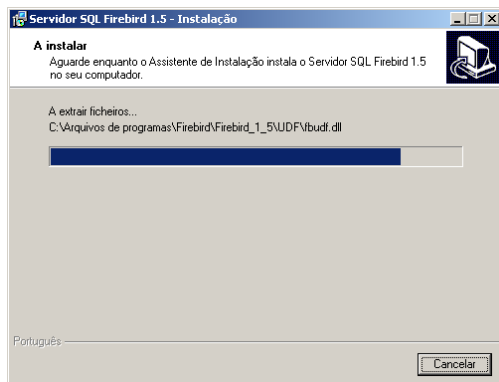
Determina se o Firebird deverá estar disponível no Painel de Controle para podermos ajustar alguns detalhes futuramente.

Depois de ter marcado as opções, clique no botão **Seguinte** para prosseguir.

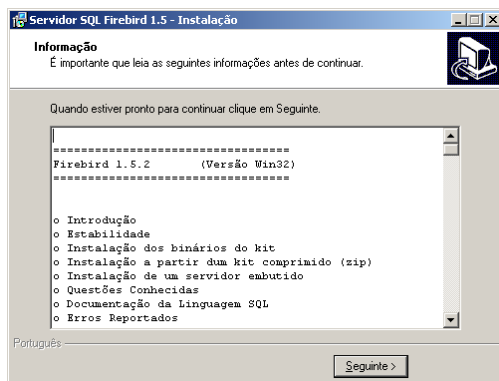


Instalando Firebird Server - Pronto para Instalar

Neste momento o Firebird está pronto para ser instalado. Confirme clicando no botão **Instalar**.



Instalando Firebird Server - Copiando Arquivos



Instalando Firebird Server - Informações

Depois de feita a instalação, é exibida uma tela com algumas informações sobre a versão instalada. Clique botão **Seguinte** para prosseguir.



*Instalando Firebird Server - Instalação Concluída*

Neste momento está concluída a instalação. Apenas habilitamos a opção **Iniciar o Serviço Firebird agora** para termos ele rodando de imediato. Esta opção estará disponível apenas se o sistema operacional permitir trabalhar com 'serviços'.

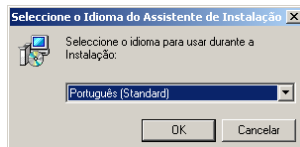
Podemos perceber o quanto é simples a instalação do Server, agora veremos a instalação do Client, que é mais fácil ainda.

### Firebird – Instalação do Client

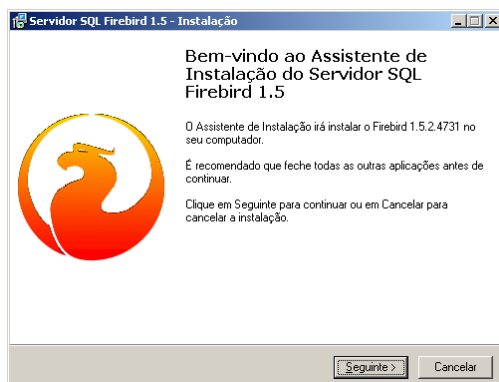
Como havíamos comentando, o instalador do Client é o mesmo do Server, inclusive os passos que seguiremos, o que modificará é na tela onde optamos pelo Tipo de Instalação.

Execute o instalador (**Firebird-1.5.2.4731-Win32.exe**).

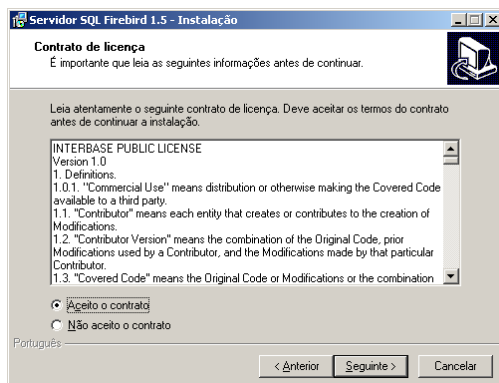
Todos os passos já foram comentados na versão Server, detalharemos apenas a tela de Tipo de Instalação.



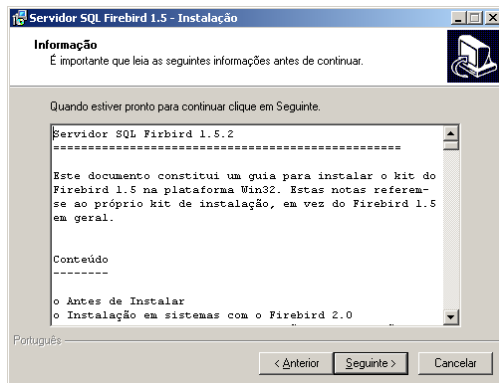
*Instalando Firebird Client - Seleção do Idioma*



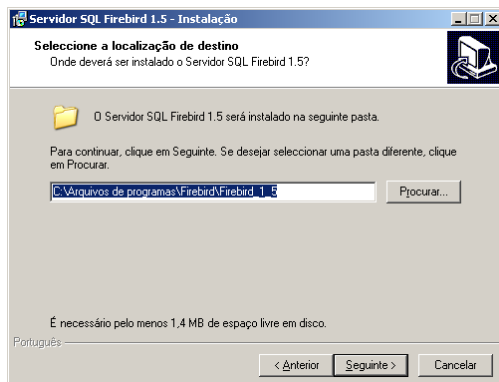
*Instalando Firebird Client - Apresentação*



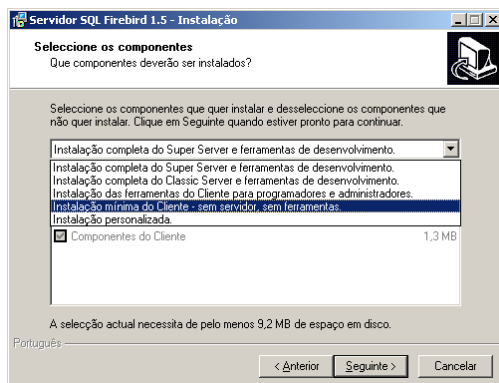
*Instalando Firebird Client - Contrato*



Instalando Firebird Client - Informações sobre a instalação



Instalando Firebird Client - Caminho de Destino

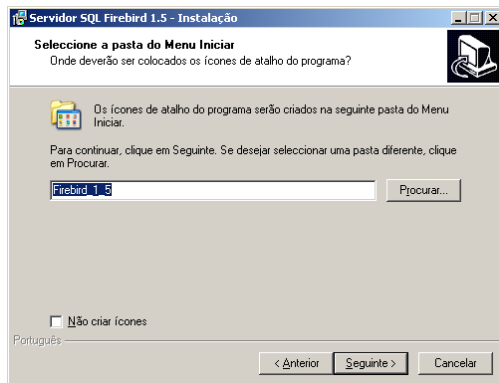


Instalando Firebird Client - Tipo de Instalação

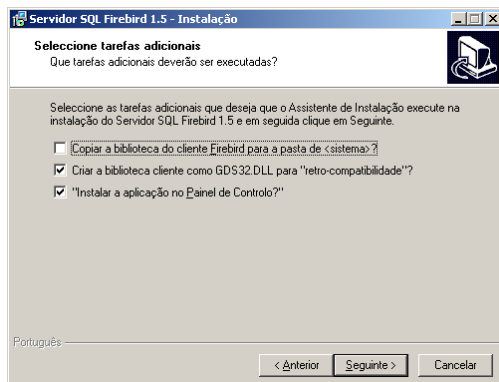
É nesta tela que devemos optar pelo **Client** seleccionando a opção **Instalação mínima do Cliente**.

Poderíamos também escolher o item anterior (Instalação das ferramentas do Cliente para programadores e administradores), que é uma instalação Client também porém mais completa, contendo ferramentas para manutenção do banco.

Confirmamos clicando no botão **Seguinte**.



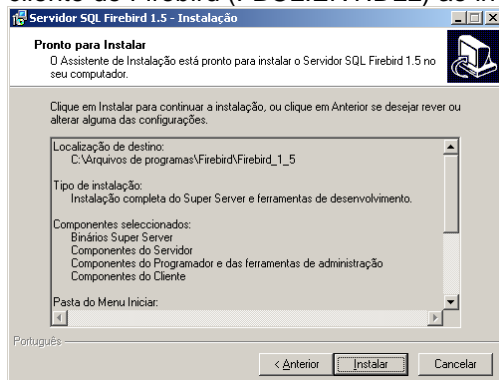
*Instalando Firebird Client - Pasta para os ícones do programa*



*Instalando Firebird Client - Tarefas*

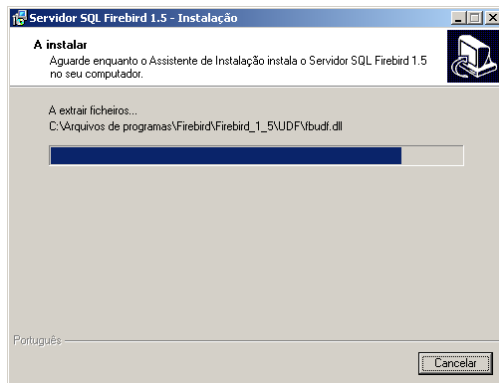
Aqui percebemos que existem menos opções do que na versão Server. Os conceitos são os mesmo já vistos anteriormente.

No caso da primeira opção, apenas seria necessário caso a aplicação utilizasse a biblioteca cliente do Firebird (FBCLIENT.DLL) ao invés da GDS32.DLL.

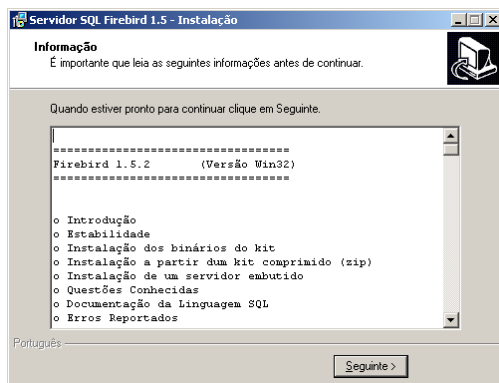


*Instalando Firebird Client - Pronto para Instalar*





*Instalando Firebird Client - Copiando Arquivos*



*Instalando Firebird Client - Informações*



*Instalando Firebird Client - Instalação Concluída*

### DBExpress

#### Introdução

A partir da versão 6 do Delphi foi disponibilizada uma nova tecnologia de acesso à banco de dados, chamada dbExpress.

Esta tecnologia veio como uma revolução, mais uma vez foi feito um trabalho excelente por parte da Borland, pois podemos perceber diversas vantagens na utilização desta engine de acesso.

DBExpress é multiplataforma, podendo trabalhar com diversos servidores de bancos de dados e possui uma alta performance devido ao novo conceito utilizado nesta engine.

Veremos a seguir algumas diferenças técnicas e suas respectivas vantagens comparando-a com BDE.

#### BDE x DBExpress

Em BDE temos a possibilidade de trabalhar diretamente com Tables, facilitando a manutenção e o acesso aos dados, porém isso acaba consumindo muitos recursos do cliente e do servidor. Em servidores de banco de dados, é altamente recomendável a utilização de Querys e Stored Procedures para extração e manutenção dos dados. DBExpress trabalha somente neste conceito, através de instruções SQL podemos ter o total controle da seleção, otimizando assim o acesso aos dados e obtendo um alto ganho de performance.

Utilizando BDE, a performance do servidor é afetada também devido as querys internas que são executadas para tornar os dados navegáveis, ter acesso a campos blobs, metadado, etc. Com DBExpress isso não acontece, somente são executadas as instruções SQL enviadas pelo cliente, não sendo executado querys extras, evitando assim afetar a performance do servidor.

Outro aspecto importantíssimo é que a DBExpress retorna somente cursores unidirecionais, não fazendo cache, conseqüentemente consumindo menos recursos no lado do cliente e do servidor. Quando precisamos fazer cache dos dados para navegação por exemplo, utilizamos o ClientDataSet, no qual contamos também com diversos recursos para manutenção dos registros.

#### Principais aspectos positivos

##### **Super leve, baixo overhead e alta performance**

DBExpress é uma camada fina sobre a API de banco de dados, não gerando muita sobrecarga (overhead) às operações, conseqüentemente possuindo uma alta performance.

##### **Cursores unidirecionais**

Como vimos, DBExpress retorna somente cursores unidirecionais, portanto os DataSets provenientes são unidirecionais, não fazem cache, por isso consomem menos recursos tanto no cliente como no servidor.

Um DataSet unidirecional significa que ele trabalha com os registros numa única direção, apenas para frente, portanto não faz cache, logo não conseguiríamos editá-lo, associá-lo a um dbGrid, usá-lo como lookup ou coisas do gênero, claro que, para esses casos, temos a solução, basta usarmos em conjunto com componente que trabalham com cache, que é o caso do ClientDataSet que veremos mais adiante.

### Drivers para diversos bancos

Um item importante quando falamos de desenvolvimento de aplicação é o que diz respeito à compatibilidade com outros servidores de bancos de dados. Neste caso, a dbExpress não deixa a desejar, pois outro aspecto importante também é o fato de trabalhar com diversos bancos e sem a necessidade da troca de componentes, apenas a substituição da respectiva DLL do banco de dados.

Na versão 6 do Delphi os drivers disponíveis eram apenas para Interbase, Oracle, DB2 e MySQL. Depois empresas terceiras desenvolveram drivers para seus respectivos bancos e na versão 7 já os temos disponíveis, que seriam para os bancos SQLServer e Informix. Somente no caso do Sysbase é necessário acessar o site do fabricante para download do driver.

### Multi-Palataforma

DBExpress está disponível tanto para Windows como para diversas versões do Linux. Podemos desenvolver uma aplicação no Delphi utilizando os componentes dbExpress e depois transportar para o Kylix, pois nele temos os mesmos componentes disponíveis.

### Fácil instalação

No BDE tínhamos mais ou menos 10mb de instalação, já com dbExpress precisamos de apenas uma DLL que não passa de 150kb.

## Distribuição

Como já vimos, a distribuição é baseada em apenas uma DLL, vejamos abaixo o nome das respectivas DLLs de cada banco:

Interbase – dbexpint.dll

Firebird – dbexpint.dll (a mesma do interbase, porém existe também drivers de terceiros específicos para este banco)

Oracle – dbexpora.dll

Db2 – dbexpdb2.dll

MySQL – dbexpmysql.dll

SQL Server – dbexpmss.dll

Informix – dbexpinf.dll

Podemos também, ao invés de distribuí-las, compilá-las na aplicação, bastando incluir a respectiva unit na cláusula uses do seu projeto. Por exemplo, no caso do Interbase/Firebird, adicionamos a unit **dbexpint**.

## Componentes

A paleta dbExpress é composta por diversos componentes que nos auxilia no acesso ao banco de dados, execução de instruções DML's (insert, update, delete), instruções SQL's, Stored Procedures entre outros recursos.

Os componentes existentes no Delphi 6, são praticamente os mesmos existentes na versão 7, diferenciando apenas em um componente, o TSQLClientDataSet que está disponível apenas na versão 6 e na versão 7 foi substituído pelo TSimpleDataSet. Veremos detalhes desta mudança mais adiante.



### SQLConnection

É o principal componente, pois ele é quem faz a conexão com o banco de dados.

Comparando com BDE, este componente seria equivalente ao TDataBase.

### Propriedades

#### ActiveStatements: Longword

Retorna o número de statements (comandos, instruções SQL) ativos no servidor de banco de dados para conexão atual.

#### Autoclone: Boolean

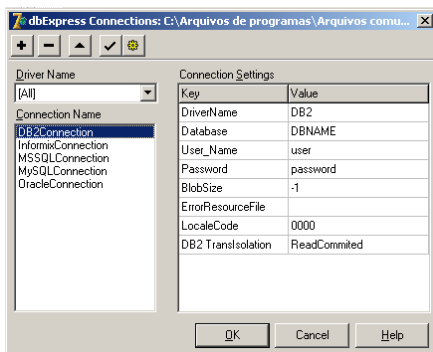
Especifica se o SQLConnection deverá ser clonado automaticamente quando o número de statements (comandos, instruções SQL) no servidor de banco de dados exceder o limite por conexão, que é definido na propriedade MaxStmtsPerConn. Se Autoclone estiver como false, o componente SQLConnection não será clonado e uma exceção será gerada.

#### Connected: Boolean

Retorna e especifica se a conexão está ou não ativa.

#### ConnectionName: string

Especifica o nome da conexão que usaremos no componente. As conexões são criadas através do Editor de Conexões, que pode ser acessado dando um duplo clique no componente:



Editor de Conexões do dbExpress

Toda conexão criada através do Editor de Conexões do SqlConnection é salva no arquivo dbxConnections.ini (C:\Arquivos de Programas\Arquivos Comuns\Borland Shared\DBExpress).

Este arquivo só será obrigatório levar junto com aplicação caso ligue a propriedade LoadParamsOnConnect estiver ligada, veja detalhes na descrição desta propriedade.

Podemos também estar ignorando a propriedade ConnectionName e trabalhar diretamente com as propriedades DriverName e Params, onde em Params informamos todos os parâmetros de conexão sem a necessidade de se criar uma conexão através do Editor.

#### ConnectionState: TConnectionState

Retorna o estado atual do componente.

*csStateClosed*: A conexão está fechada.

*csStateOpen*: A conexão está aberta.

*csStateConnecting*: A conexão está sendo estabelecida, porém ainda não foi concluída.

*csStateExecuting*: Está sendo executado um comando SQL que o componente passou para o servidor.

*csStateFetching*: O componente está buscando informações do servidor.

*csStateDisconnecting*: O componente está desconectando, porém ainda não terminou o processo.

**DriverName: string**

Especifica o driver de conexão, ou seja, qual servidor de banco de dados será utilizado.

**GetDriverFunc: string**

Especifica o nome da função que faz chamada a função que implementa o driver para acesso a dbExpress. É preenchida automaticamente após informarmos a propriedade DriverName.

**InTransaction: Boolean**

Retorna se existe alguma transação aberta. Uma transação só é finalizada quando chamamos o comando Commit ou Rollback.

**KeepConnection: Boolean**

Especifica se a conexão deverá permanecer ativa mesmo após todos os dataset's ligados ao SQLConnection tiverem sido fechados.

**LibraryName: string**

Especifica o nome da biblioteca dbExpress que faz acesso ao servidor de banco de dados. É preenchida automaticamente após informarmos a propriedade DriverName.

**LoadParamsOnConnect: Boolean**

Especifica se os parâmetros serão lidos a partir do arquivo dbxConnections.ini antes de abrir a conexão caso esteja utilizando a propriedade ConnectionName.

Por padrão é false, pois quando ajustamos a propriedade ConnectionName, todos os parâmetros da conexão serão definidos no componente na propriedade DriverName e Params, sendo assim, não há necessidade da existência do arquivo dbxConnections.ini quando for rodar a aplicação.

Ligando esta propriedade, sempre antes de conectar-se, será lido as configurações a partir do arquivo dbxConnections.ini ignorando os parâmetros atuais definidos no componente, neste caso, será obrigatório à existência do arquivo dbxConnections.ini e a respectiva conexão definida na propriedade ConnectionName.

**LocaleCode: TLocaleCode**

Especifica LocaleCode usado na conexão. Ajustando esta propriedade, automaticamente será atualizado o parâmetro LocaleCode na propriedade Params do componente.

LocaleCode determina a ordem de apresentação das colunas associadas ao DataSet. Devemos tomar cuidado ao alterá-lo, pois possui um grande impacto na performance, recomenda-se deixar com o valor padrão, 0 (zero).

**LoginPrompt: Boolean**

Especifica se as informações de login deverão ser requisitadas ao abrir a conexão.

Se definirmos esta propriedade como False, deveremos então informar o login e senha nas propriedades da conexão, caso contrário, será requisitado o login/senha através do formulário padrão do Delphi, que poderá ser mudado, bastando implementar o evento OnLogin do componente.

### **MaxStmtsPerConn: Longword**

Especifica o número máximo de statements (comandos, instruções SQL) permitidos na conexão. Informando o valor 0 (zero), significa que não há limites, um valor maior, determinará a quantidade máxima permitida, nesta situação, caso exceda, o componente se comportará de forma diferente de acordo com o que foi definido na propriedade Autoclone.

### **Metadata: ISQLMetadata**

Provê acesso às Interfaces que contém informações (metadata) do banco de dados. Não é comum sua utilização, só é recomendada quando houver a necessidade de ganho de performance. É mais simples trabalharmos com os métodos GetTableNames, GetIndexNames, GetFieldNames, etc.

### **MultipleTransactionsSupported: Boolean**

Retorna se o servidor de banco de dados permite trabalhar com conexão simultâneas.

### **Params: TParams**

Especifica os parâmetros da conexão. Podemos utilizá-la para especificar os parâmetros sem a necessidade de criar uma conexão através do Editor de Conexões.

### **ParamsLoaded: Boolean**

Retorna se os parâmetros foram lidos a partir do arquivo dbxconnections.ini. Não é recomendada sua utilização, já que é utilizado apenas internamente pelo Editor de Conexões.

### **SQLConnection: ISQLConnection**

Provê acesso à interface da conexão. Não é necessária sua utilização, pois os métodos e propriedades existentes já são suficientes para trabalharmos com o componente.

### **SQLHourGlass: Boolean**

Determina se o cursor do mouse deverá mudar quando uma instrução SQL estiver sendo executada.

### **TableScope: TTableScopes**

Especifica os tipos de tabelas retornadas quando forem requisitadas através do método GetTableNames.

*tsSynonym*: Informações de sinônimos.

*tsSysTable*: Tabelas de sistema.

*tsTable*: Tabelas criadas pelos usuários.

*tsView*: Views.

### **TraceCallbackEvent: TSQLCallbackEvent**

Provê o acesso ao método Callback definido no SetTraceCallbackEvent.

### **TransactionsSupported: Boolean**

Retorna se o servidor de banco de dados suporta transações.

### **VendorLib: string**

Especifica o nome da biblioteca cliente do servidor de banco de dados. É preenchida automaticamente após informarmos a propriedade DriverName.

### SQLConnection - Métodos

#### **CloneConnection: TSQLConnection**

Retorna uma cópia do objeto SQLConnection.

#### **CloseDataSets**

Fecha todos os DataSet's ligados ao SQLConnection sem desconectar-se do banco de dados.

#### **Close**

Fecha a conexão com o banco de dados.

#### **Open**

Abre a conexão com o banco de dados.

#### **Commit(TransDesc: TTransactionDesc)**

Salva todas operações realizadas dentro de uma transação.

#### **Parâmetros**

**TransDesc:** Objeto com as informações da transação, o mesmo passado como parâmetro no método StartTransaction.

#### **Execute(const SQL: string; Params: TParams; ResultSet: Pointer=nil): Integer**

Executa uma instrução SQL no servidor, sem a necessidade, por exemplo, da utilização dos componentes SQLQuery, SQLDataSet.

Esta função retorna o número de registros afetados na execução.

#### **Parâmetros**

**SQL:** instrução SQL a ser executada.

**Params:** Parâmetros da instrução SQL, se não houver, poderá ser passado como NIL.

**ResultSet:** Caso a instrução SQL retorne um DataSet, poderá ser passado uma variável do tipo TCustomSQLDataSet, neste caso, será criada uma instância desta classe e o objeto será alimentado com os registros retornados pela execução do SQL. No caso de não retornar registros, este parâmetro poderá ser passado como NIL.

#### **ExecuteDirect**

Executa uma instrução SQL no servidor, sem a necessidade, por exemplo, da utilização dos componentes SQLQuery, SQLDataSet.

Este método é semelhante ao Execute, porém mais simples, pois não nos permite retornar uma DataSet e passar parâmetros para instrução SQL.

O seu retorno também difere do método Execute, pois retorna 0 quando executado com sucesso, caso contrário, retorna o código de erro da dbExpress.

#### **GetFieldNames (const TableName: String; List: TStringList);**

Retorna no nome dos campos de uma determinada tabela.

#### **Parâmetros**

**TableName:** Nome da tabela a ser extraído o nome dos campos.

**List:** Objeto a ser alimentando com o nome dos campos.

#### **GetIndexNames(const TableName: string; List: TStringList);**

Retorna o nome dos índices de uma determinada tabela.

### Parâmetros

**TableName:** Nome da tabela a ser extraído o nome dos índices.

**List:** Objeto a ser alimentado com o nome dos índices.

### **GetPackageNames (List: TStrings);**

Retorna a lista de packages existentes no banco de dados, utilizado no caso do servidor Oracle.

### Parâmetros

**List:** Objeto a ser alimentado com o nome das packages.

### **GetProcedureNames (List: TStrings);**

**GetProcedureNames (const PackageName: string; List: TStrings);**

Retorna o nome das Stored Procedures existentes no banco de dados.

### Parâmetros:

**PackageName:** Nome da package que contém as Stored Procedures, utilizado no caso do servidor Oracle.

**List:** Objeto a ser alimentado com o nome das Stored Procedures.

### **GetProcedureParams(ProcedureName: String; List: TList);**

**GetProcedureParams(ProcedureName, PackageName: String; List: TList);**

Retorna informações dos parâmetros existentes em uma determinada StoredProcedure.

### Parâmetros:

**ProcedureName:** Nome da Stored Procedure na qual será extraída as informações dos parâmetros.

**PackageName:** Nome da package que contém a Stored Procedure, utilizado no caso do servidor Oracle.

**List:** Objeto a ser alimentado com as informações dos parâmetros. Para cada parâmetro existente na Stored Procedure, é adicionado um item a esta lista. Este item é um ponteiro para o tipo SPPParamDesc (record) contendo as informações do respectivo parâmetro. É importante saber que é de nossa responsabilidade liberar esses ponteiros alocados para cada parâmetro, podemos fazer isso utilizando o método FreeProcParams.

### **GetTableNames(List: TStrings; SystemTables: Boolean = False);**

Retorna o nome das tabelas existentes no banco de dados.

### Parâmetros:

**List:** Objeto a ser alimentando com o nome das tabelas.

**SystemTables:** Se for definido como true, serão retornadas apenas as tabelas de sistema, caso contrário, retornará as tabelas de acordo com o que foi definido na propriedade TableScope do componente SQLConnection.

### **LoadParamsFromIniFile(AFileName : String = "");**

Alimenta os parâmetros da conexão a partir de um arquivo INI, ou seja, as propriedades DriverName e Params serão ajustadas da mesma forma que estão no arquivo INI da respectiva conexão definida no componente na propriedade ConnectionName.

### Parâmetros:

**AFileName:** Nome do arquivo INI a ser carregado. Caso esteja em branco, será utilizado o arquivo dbxconnections.ini.

### **Rollback**

Desfaz as operações realizadas dentro de uma transação.



### Parâmetros

**TransDesc:** Objeto com as informações da transação, o mesmo passado como parâmetro no método StartTransaction.

### **SetTraceCallbackEvent(Event: TSQLCallbackEvent; IClientInfo: Integer);**

Define a função Callback que será executada cada vez que uma mensagem for passada entre o driver dbExpress e o servidor de banco de dados.

### Parâmetros:

**Event:** Função que será executada a cada mensagem interceptada entre o driver dbExpress e o servidor de banco de dados. A estrutura desta função será comentada logo abaixo.

**IClientInfo:** Valor definido pelo usuário. Estará disponível na função através do parâmetro CInfo.

O tipo **TSQLCallbackEvent** é uma função com a seguinte estrutura:

### **function(CallType: TRACECat; CInfo: Pointer): CBRType;**

**CallType:** Tipo da função Callback. Somente um valor pode ser atribuído: cbTRACE. Este tipo indica que a função é chamada a cada mensagem passada entre o driver dbExpress e o servidor de banco de dados.

**CInfo:** Ponteiro para o tipo SQLTRACEDesc. Este tipo contém as informações: texto da mensagem, categoria e o valor definido pelo usuário, passado no método SetTraceCallbackEvent, através do parâmetro IClientInfo.

Esta função retorna um valor que indica como a monitoração foi procedida. O padrão é cbrUSEDEF.

### **StartTransaction(TransDesc: TTransactionDesc);**

Inicia uma transação no banco de dados. Para checar se o servidor de banco de dados suporta transações, verifique a propriedade TransactionsSupported,

### Parâmetros

**TransDesc:** Record com as informações da transação.

Quando trabalhamos com transações no dbExpress, temos um identificador para cada uma, pois a dbExpress nos permite trabalhar com diversas transações ao mesmo tempo. Por este motivo, sempre temos que declarar uma variável do tipo TTransactionDesc e alimentarmos com as informações da transação e assim, passá-la como parâmetro para os métodos StartTransaction, Commit e Rollback.

O tipo **TTransactionDesc** é um record com os seguintes campos:

**TransactionID:** ID para identificação da transação, este não poderá conflitar com IDs de outras transações abertas.

**GlobalID:** Uso exclusivo para transações no servidor Oracle.

**IsolationLevel:** Nível de isolamento para outras transações, pode utilizar um dos seguintes valores:

`xiDIRTYREAD`: Enxerga todas alterações mesmo que não tenham sido confirmadas (commit) ainda. Este nível de isolamento não está disponível para o driver Oracle.

`xiREADCOMMITTED`: Enxerga somente informações confirmadas (commit).

`xiREPEATABLEREAD`: Enxerga apenas as alterações confirmadas (commit) antes de iniciar a transação, ou seja, este nível de isolamento sempre visualizará os dados originais, conforme estavam no início da transação, novas alterações somente serão enxergadas quando for reiniciada a transação.

`xiCUSTOM`: Utilizado em conjunto com a propriedade `CustomIsolation`, pois nela definiremos um nível de isolamento diferente das disponíveis atualmente, isso dependerá do suporte de cada servidor banco de dados.

*CustomIsolation*: Nível de isolamento customizado, de acordo com o servidor de banco de dados. Utilizada quando optamos por `xiCUSTOM` na propriedade `IsolationLevel`.

### SQLConnection - Eventos

#### **AfterConnect**

Ocorre após ter sido efetuada a conexão.

#### **AfterDisconnect**

Ocorre após ter sido efetuada a desconexão.

#### **BeforeConnect**

Ocorre antes de ter sido efetuado a conexão.

#### **BeforeDisconnect**

Ocorre antes de ter sido efetuada a desconexão.

#### **OnLogin**

Ocorre quando forem requisitados os dados de login. Quando ativamos a propriedade LoginPrompt, podemos modificar a tela padrão de login do Delphi, para isso, bastaríamos chamar nossa tela de login neste evento, e em seguida, alimentar o parâmetro Params ajustando o usuário e senha informados na tela.

#### **Parâmetros**

**DataBase:** Referência ao componente de Conexão, o próprio SQLConnection.

**LoginParams:** Parâmetros referentes aos dados de login.



### SQLDataSet

O SQLDataSet como o próprio nome já diz, é um DataSet que nos permite executar Queries e StoredProcedures. Este componente é bem semelhante ao SQLQuery que veremos mais adiante, porém seu grande diferencial está no fato de poder trabalhar de três formas, *Query* (instruções SQL's), *Table* (acesso direto ao nome da tabela) e *SP's* (Stored Procedures), enquanto que o componente SQLQuery permite-nos apenas trabalhar com *Queries*.

### Propriedades

#### Active: Boolean

Indica se o DataSet está ativo

#### CommandText: string


Esta propriedade varia de acordo com o valor informado em *CommandType*.

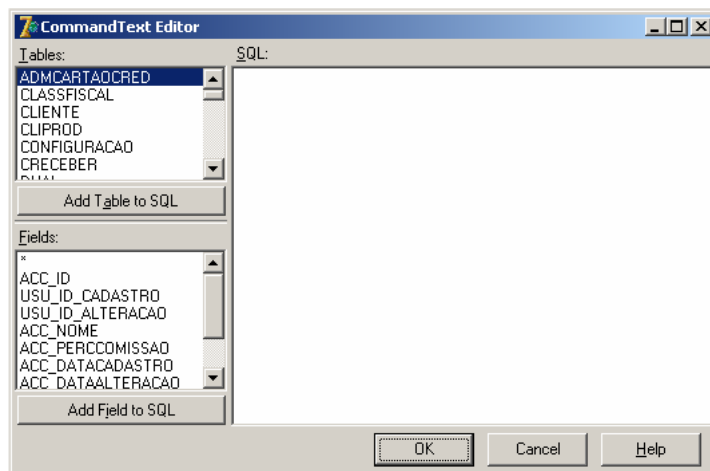
Quando a propriedade **CommandType** for:

*ctQuery*: Informamos a instrução SQL.

*ctStoredProc*: Exibe-se uma lista com o nome das StoredProcedures disponíveis no banco.

*ctTable*: Exibe-se uma lista com nome das Tabelas disponíveis no banco.

No caso de ser *ctQuery*, ao clicarmos no botão  será exibida uma tela para montarmos nossa instrução SQL, conforme a figura abaixo:



Editor SQL do componente SQLDataSet

#### CommandType: TSQLCommandType

Tipo de Comando a ser trabalhado. Esta é a propriedade que comentamos logo acima, que nos permite optar entre *ctQuery*, *ctTable* ou *ctStoredProc*. Veja os comentários da propriedade **CommandText** para saber detalhes.

**DataSource: TDataSource**

Permite-nos informar o DataSource Master para um relacionamento mestre detalhe entre query's.

**DesignerData: string**

Não possui um objetivo específico, é uma propriedade livre para usarmos com qualquer finalidade, muito semelhante à propriedade Tag, porém podemos especificar uma string nesta propriedade.

**IndexDefs: TIndexDefs**

Contém as definições de índices existente no banco de dados para o respectivo DataSet.

A classe TCustomSQLDataSet não utiliza informações desta propriedade, a ordem dos registros é definida através da cláusula order by do SQL.

No caso do componente SQLTable, esta propriedade pode ser usada para identificar o índice usado na propriedade IndexName. Para outros componentes descendentes de TCustomSQLDataSet, esta propriedade pode ser útil para compor o comando SQL que necessita de um índice no banco de dados.

**GetMetadata: Boolean**

Informa se o metadado será extraído junto à requisição dos registros. O metadado contém informações de índices da tabela. Deixando esta opção como False, ganhamos performance, pois não será necessário nenhum SQL adicional para extrair o metadado, porém em uma atualização, poderá ficar mais lento, pois será necessário obter o metadado. Portanto, é recomendado apenas no caso de DataSet read-only, ou seja, em consultas.

**MaxBlobSize: Integer**

Determina o número máximo de bytes a ser obtido dos campos Blobs do DataSet. Caso seja -1, não terá limites, sendo 0, o limite será o valor especificado nos parâmetros da conexão, caso contrário, estará determinando o tamanho máximo real.

**NoMetadata: Boolean**

Esta propriedade tem o mesmo objetivo da propriedade GetMetadata que comentamos logo acima, porém está em desuso, é provável que seja eliminada nas próximas versões do Delphi.

**NumericMapping: Boolean**

Usado para determinar como serão controlados os campos BCD.

**ObjectView: Boolean**

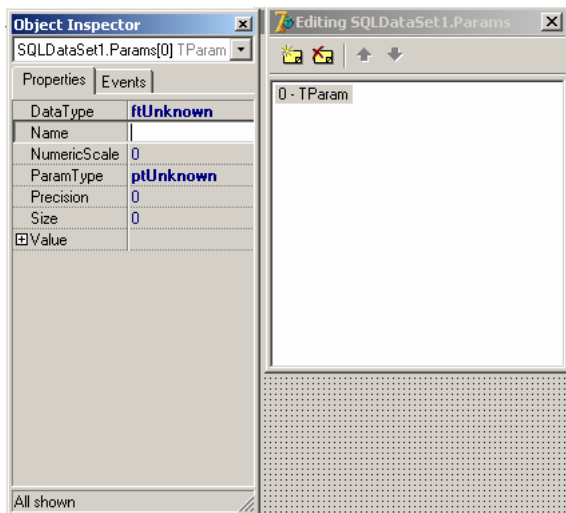
Determina como os campos serão armazenados e acessados. O padrão é False, ou seja, serão acessados através do TFields, caso seja True, serão acessados através do objeto.

**ParamCheck: Boolean**

Determina se os parâmetros serão gerados novamente quando ocorrerem mudanças na instrução SQL do componente, claro que, somente em queries parametrizadas.

**Params: TParams**

Nesta propriedade ficam armazenados os parâmetros no caso de queries parametrizadas, são criados automaticamente, porém podemos abrir este editor para ajustar algumas propriedades de cada parâmetro, como por exemplo, seu DataType.



Editor de Parâmetros

### **Prepared: Boolean**

Indica se o comando SQL deverá ser preparado antes de sua execução.

Quando preparamos, estamos alocando recursos para a instrução juntamente com seus parâmetros, isto é altamente importante para ganho de performance quando executamos diversas vezes a mesma instrução, modificando apenas valores dos parâmetros, pois desta forma, não será necessário uma re-preparação (re-alocação de recursos) para cada execução, neste caso, basta ajustarmos a propriedade para True que será alocado recursos uma vez apenas.

Caso queira que a query seja re-preparada antes da sua execução, no caso por exemplo de mudar alguma cláusula na instrução SQL, poderá então estar definindo para False esta propriedade.

### **RecordCount: Integer**

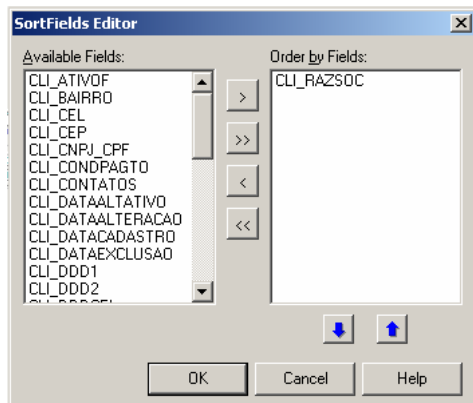
Retorna a quantidade de registros existentes no DataSet. Esta propriedade só não poderá ser utilizada no caso de Stored Procedures ou queries parametrizadas.

### **SchemaName: string**

Nome do Schema pertencente à tabela.

### **SortFieldNames: string**

Nesta propriedade informamos como serão ordenados os registros do DataSet. Internamente, quando o DataSet for extrair os dados do banco, ele acrescentará à cláusula 'order by' utilizando os campos definidos nesta propriedade.



SortFields Editor

### SQLConnection: TSQLConnection

Esta é uma propriedade muito importante, é nela que informamos qual é o nosso componente de conexão. No BDE informávamos uma 'string' na propriedade DatabaseName indicando qual era o nosso Alias ou Databasename (definido no componente Database), no caso da dbExpress, sempre teremos esta propriedade para associarmos ao componente SqlConnection.

### TransactionLevel: Smallint

Indica a qual transação o DataSet pertence. Se não for especificado um valor para esta propriedade, a transação utilizada será a mais recente iniciada.

Esta propriedade é raramente utilizada, pois depende muito de servidor de banco de dados permitir sobreposição de transações.

### SQLDataSet - Métodos

**CreateBlobStream (Field: TField; Mode: TBlobStreamMode): TStream;**

Retorna a instância de um Stream a partir de um campo blob.

**Parâmetros**

**Field:** Campo BLOB que será utilizado como base para instanciar o Stream.

**Mode:** Modo como o Stream será criado (bmRead, bmWrite ou bmReadWrite).

**GetBlobFieldData(FieldNo: Integer; var Buffer: TBlobByteData): Integer**

Alimenta a variável Buffer com os dados de um campo BLOB e retorna para a função o tamanho do buffer.

Este método é utilizado internamente. Para acessar os dados de um BLOB, utilize a função CreateBlobStream.

**Parâmetros**

**FieldNo:** Índice do campo BLOB.

**Buffer:** Array dinâmico de bytes que contém os dados do BLOB.

**GetDetailLinkFields(MasterFields, DetailFields: TList);**

Retorna a lista de campos utilizados no relacionamento mestre/detalhe.

**Parâmetros**

**MasterFields:** Lista dos campos Master

**DetailFields:** Lista dos campos Detail

**GetFieldData(Field: TField; Buffer: Pointer): Boolean;**

**GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean): Boolean**

**GetFieldData(FieldNo: Integer; Buffer: Pointer): Boolean;**

Alimenta a variável buffer com o valor do campo e retorna True ou False indicando se o dados foram obtidos com sucesso.

Não é comum a utilização deste método nas aplicações, ele é utilizado na implementação do método GetData da classe TField.

**Parâmetros**

**Field:** Campo a ser extraído o dado

**Buffer:** Esta variável será alimentada com o valor do campo

**NativeFormat:** Indica se o campo está em seu formato nativo. Caso este parâmetro seja False, será feita uma conversão para o formato nativo.

**FieldNo:** Número do campo a ser extraído o dado

**GetKeyFieldNames(List: TStringList): Integer**

Alimenta uma lista de índices existentes no banco de dados com seus respectivos campos e retorna o número de índices encontrados.

**Parâmetros**

**List:** Lista a ser alimentada com o nome dos índices e campos.

**GetQuoteChar: string**

Retorna o(s) caracter(es) utilizado na delimitação de strings nos comandos SQL.



### **IsSequenced: Boolean**

Retorna se o DataSet pode usar a propriedade RecNo para indicar a ordem dos registros. Caso esta função retorne False, não poderemos utilizar a propriedade RecNo, sempre retornará -1. No caso do SQLDataSet, sempre será False, pois este componente, como já dissemos, trabalha com o cursor unidirecional.

### **ParamByName (const Value: string): TParam;**

Retorna um parâmetro através do seu nome.

### **Parâmetros**

**Value:** Nome do parâmetro a ser localizado

### **SetSchemaInfo(SchemaType: TSchemaType; SchemaObjectName, SchemaPattern: string);**

Indica se o Dataset representa um metadado, se sim, seu respectivo tipo. Após definirmos o tipo de metadado, ao abrirmos o DataSet, o mesmo será alimentado com o respectivo tipo de metadado informado.

### **Parâmetros**

**SchemaType:** Indica o tipo de informação a ser obtido, pode ser:

*stNoSchema:* Nenhum tipo de informação. O dataset será alimentado com o resultado da Query ou Stored Procedure (especificado em CommandText) ao invés do metadado.

*stTables:* Informações sobre as tabelas do banco de dados, de acordo com o que foi definido na propriedade TableScope do componente SQLConnection.

*stSysTables:* Informações sobre as tabela de sistema do servidor de banco de dados.

*stProcedures:* Informações sobre as Stored Procedures do banco de dados.

*stColumns:* Informações sobre as colunas de uma determinada tabela, especificada na propriedade SchemaObjectName.

*stProcedureParams:* Informações sobre os parâmetros de uma Stored Procedure, especificada no parâmetro SchemaObjectName.

*stIndexes:* Informações sobre todos os índices de uma tabela, especificada no parâmetro SchemaObjectName.

**SchemaObjectName:** Indica o nome da tabela no caso de usarmos o SchemaType como *stColumns*, *stIndexes*. No caso de *stProcedureParams*, será usado para indicar no nome da Stored Procedure. Já nos demais casos, não terá finalidade, será ignorado.

**SchemaPattern:** Indica a mascara a ser utilizada para filtrar o resultado. Maiores detalhes poderão ser obtidos no help do Delphi.

### SQLDataSet - Eventos

**AfterClose**

Ocorre após o DataSet ter sido fechado.

**AfterOpen**

Ocorre após o DataSet ter sido aberto.

**AfterRefresh**

Ocorre após ter sido chamado o Refresh no DataSet.

**AfterScroll**

Ocorre após uma movimentação de registro no DataSet.

**BeforeClose**

Ocorre antes de o DataSet ter sido fechado.

**BeforeOpen**

Ocorre antes de o DataSet ter sido aberto.

**BeforeRefresh**

Ocorre antes de ter sido chamado o Refresh no DataSet.

**BeforeScroll**

Ocorre antes de uma movimentação de registro no DataSet.

**OnCalcFields**

Ocorre quando é preciso recalcular os campos calculados.



### SQLQuery

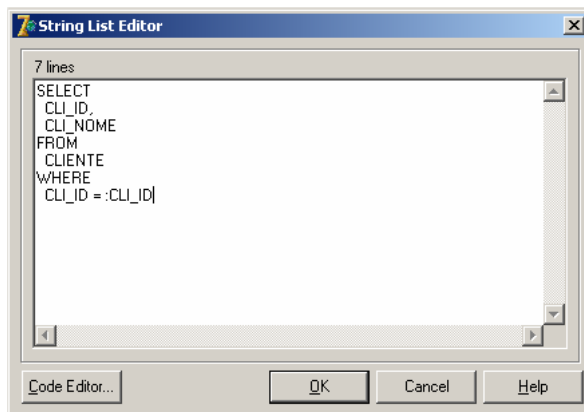
Conforme já mencionamos, este componente é muito semelhante ao componente `SQLDataSet`, porém ele é mais simples de usar, pois trabalha apenas com execução de instruções SQL.

Comparando com BDE, este componente seria equivalente ao `TQuery`, porém não podemos editar e visualizar os dados em um `DataControl` diretamente, precisamos de algum componente que faça o cache, no caso o `ClientDataSet`.

### Propriedades

#### SQL: TStrings

Nesta propriedade definimos o comando SQL a ser executado no servidor.



*Editor SQL*

#### Text: string

Retorna a instrução SQL definida no componente. Possui o mesmo efeito de acessarmos a propriedade `Text` da propriedade `SQL` do componente.

As propriedades a seguir não serão detalhadas, pois possuem as mesmas finalidades já descritas no componente `TSQLDataSet`.

**DataSource, DesignerData, IndexDefs, MaxBlobSize, NoMetadata, ParamCheck, Params, Prepared, RecordCount, RowsAffected, SQLConnection, TransactionLevel.**

### SQLQuery - Métodos

#### **ExecSql(ExecDirect: Boolean = False): Integer**

Executa uma query que não retorna registros. Utilizada em instruções do tipo Insert, Update, Delete, Create Table, etc. Esta função retornará o número de registros afetados pela execução do comando.

#### **Parâmetros**

**ExecDirect:** Indica se a query será executada diretamente sem a necessidade de prepará-la. Definindo como False, a query será preparada antes da execução.

#### **PrepareStatement**

Prepara a query para execução. Possui a mesma finalidade quando ativamos a propriedade Prepared, como já visto anteriormente na classe TSQLDataSet. O help do delphi diz ser mais recomendável a utilização da propriedade à que este método.

Os métodos a seguir não serão detalhados, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**CreateBlobStream,      GetBlobFieldData,      GetDetailLinkFields,      GetFieldData,  
GetKeyFieldNames, GetQuoteChar, IsSequenced, ParamByName, SetSchemaInfo.**

### SQLQuery - Eventos

Os eventos a seguir não serão detalhados, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**AfterClose, AfterOpen, AfterRefresh, AfterScroll, BeforeClose, BeforeOpen, BeforeRefresh, BeforeScroll, OnCalcFields.**



### SQLStoredProc

Este componente é específico para manipulação de Stored Procedures. Com ele podemos executar qualquer tipo de Stored Procedure que retorne registros, parâmetros ou que apenas executam instruções no banco mas não retornam valores.

No caso de Stored Procedure que retornam registros, segue-se o mesmo conceito já visto sobre os DataSets da dbExpress, ou seja, são unidirecionais, isso significa que se quisermos visualizar os registros, teremos de associar a um componente que trabalhe com cache, no caso, ClientDataSet que veremos mais adiantes.

Comparando com BDE, este componente seria “equivalente” ao TStoredProc, porém, no caso de uma StoredProcedure que retorna dados, não podemos visualizar diretamente em um DataControl, precisamos de um componente que faça o cache, no caso o ClientDataSet.

### Propriedades

#### **PackageName: string**

Indica o nome da package que contém a Stored Procedure a ser selecionada na propriedade StoredProcName, utilizada somente no caso do Oracle. Se a Stored Procedure não estiver em uma package, basta deixar esta propriedade em branco.

#### **StoredProcName: string**

Como próprio nome já diz, nesta propriedade informamos o nome da Stored Procedure que iremos utilizar.

As propriedades a seguir não serão detalhadas, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**DataSource, DesignerData, IndexDefs, MaxBlobSize, NoMetadata, ParamCheck, Params, Prepared, RecordCount, SQLConnection, TransactionLevel.**

### SQLStoredProc - Métodos

#### **ExecProc**

Executa uma Stored Procedure que não retorna um cursor. Esta função retorna o número de registros afetados pela execução.

Para ganho de performance, aconselhasse a preparar a Stored Procedure (definindo True na propriedade Prepared) antes de chamar o método pela primeira vez.

No caso de Stored Procedure que retorna dados, utilize o método Open ou ligue a propriedade Active.

#### **NextRecordSet: TCustomSQLDataSet**

Retorna o próximo conjunto de dados retornados pela Stored Procedure.

Existem Stored Procedures que retornam mais de um conjunto de dados. Ativando a Stored Procedure, obteremos apenas o primeiro conjunto, utilizando o NextRecordSet, teremos uma instância de um TCustomSQLDataSet onde estará alimentado com o próximo conjunto de dados a ser extraído.

Quando chamamos o método pela primeira vez, será obtido o segundo conjunto de dados, na segunda chamada, será obtido o terceiro conjunto, e assim por diante. Quando o método retornar **NIL**, é por que não há mais dados a serem obtidos.

#### **PrepareStatement**

Possui a mesma finalidade e mesma orientação descrita no componente TSQLQuery.

Os métodos a seguir não serão detalhados, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**CreateBlobStream, GetBlobFieldData, GetDetailLinkFields, GetFieldData, GetKeyFieldNames, GetQuoteChar, IsSequenced, ParamByName, SetSchemaInfo.**

### SQLStoredProc - Eventos

Os eventos a seguir não serão detalhados, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**AfterClose, AfterOpen, AfterRefresh, AfterScroll, BeforeClose, BeforeOpen, BeforeRefresh, BeforeScroll, OnCalcFields.**





### SQLTable

Este componente também é muito semelhante ao `SQLDataSet`, porém não nos permite trabalhar com instruções SQL, apenas informamos o nome da tabela a ser usada.

Internamente este componente gera uma instrução SQL no servidor para extrair todos os registros da respectiva tabela definida na propriedade `TableName`.

Comparando com BDE, este componente seria “equivalente” ao `TTable`, porém não podemos editar e visualizar os dados em um `DataControl` diretamente, precisamos de algum componente que faça o cache, no caso o `ClientDataSet`.

### Propriedades

#### **IndexFieldCount: Integer**

Retorna a quantidade de campos existentes no índice atual.

#### **IndexFieldNames: string**

Determina a ordenação dos registros, especificamos os campos separando-os com ; (ponto-e-vírgula). Esta propriedade é alimentada automaticamente quando fazemos um relacionamento mestre detalhe através das propriedades `MasterSource` e `MasterFields`, ela será alimentada com o nome dos campos envolvidos no relacionamento.

Utilizando esta propriedade, internamente o componente acrescentará a cláusula ‘order by’ (com os campos informados) à instrução SQL que será enviada ao banco.

#### **IndexFields[Index: Integer]: TField;**

Permite acessarmos diretamente os campos utilizados no índice atual.

#### **IndexName: string**

Determina o índice a ser utilizado para ordenação dos registros. A propriedade `IndexName` e `IndexFieldNames` são mutuamente exclusivas, ou seja, trabalhamos com uma ou com a outra, nunca com ambas. Nesta propriedade serão listados todos os índices existentes na tabela.

Esta propriedade também será alimentada automaticamente caso seja utilizado relacionamento mestre/detalhe.

#### **MasterFields: string**

Utilizado para relacionamentos mestre/detalhe. Indica o nome do campo chave no relacionamento.

#### **MasterSource: TDataSource**

Utilizado para relacionamentos mestre/detalhe. Indica o `DataSource` que está ligado ao `DataSet` Master.

#### **TableName: string**

Nesta propriedade informamos o nome da tabela a ser utilizado no componente. Serão listados os nomes de todas as tabelas existentes no banco de dados.

## Teoria sobre Firebird, DBExpress e ClientDataSet

---

As propriedades a seguir não serão detalhadas, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**DataSource, DesignerData, IndexDefs, MaxBlobSize, NoMetadata, Prepared, RecordCount, SQLConnection, TransactionLevel.**

### SQLTable - Métodos

#### **DeleteRecords**

Exclui todos os registros da tabela.

#### **GetIndexNames(List: Tstrings)**

Extrai a lista de índices disponíveis na tabela.

#### **Parâmetros:**

**List:** Lista dos índices.

#### **PrepareStatement**

Gera a query (SELECT) utilizada para poder extrair os registros da tabela.

Não execute este método diretamente na aplicação, utilize a propriedade Prepared definindo-a para True, evitando assim que o SQLTable gere a query toda vez que executada.

Os métodos a seguir não serão detalhados, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**CreateBlobStream, GetBlobFieldData, GetDetailLinkFields, GetFieldData, GetKeyFieldNames, GetQuoteChar, IsSequenced, ParamByName, SetSchemaInfo.**

### SQLTable - Eventos

Os eventos a seguir não serão detalhados, pois possuem as mesmas finalidades já descritas no componente TSQLDataSet.

**AfterClose, AfterOpen, AfterRefresh, AfterScroll, BeforeClose, BeforeOpen, BeforeRefresh, BeforeScroll, OnCalcFields.**



### SQLMonitor

Este componente nos permite monitorar as instruções SQL que estão sendo enviadas ao servidor de banco de dados. Isso é essencial quando precisamos debugar a aplicação, verificar questões de performance entre outros detalhes.

#### Propriedades

**Active: Boolean**

Determina se o monitoramento está ativo.

**Autosave: Boolean**

Podemos guardar o Log em um arquivo, é nesta propriedade que ativamos este recurso, ela é utilizada em conjunto com a propriedade FileName, que veremos em seguida.

**FileName: string**

Esta propriedade é utilizada em conjunto com a propriedade Autosave, como mencionado, é nela que indicamos o caminho onde o arquivo de Log será gerado. Esta propriedade possui também outra finalidade: se caso chamarmos os métodos SaveToFile ou LoadFromFile com o parâmetro requisitado em branco, eles utilizarão o valor desta propriedade, e como o próprio nome já diz, esses métodos são específicos para gravação e leitura do arquivo de Log.

**SqlConnection: TSQLConnection**

Aqui informamos o componente SqlConnection que estaremos monitorando.

**TraceList: TStrings**

Nesta propriedade serão armazenadas as mensagens interceptadas.

Vale lembrar que, além de guardarmos as mensagens interceptadas na propriedade TraceList e/ou no arquivo de Log, podemos também manipulá-las e eventualmente tratá-las através dos eventos OnLogTrace e OnTrace do componente.

**MaxTraceCount: Integer**

Não é tão comum, mas podemos limitar o número de mensagens a serem armazenadas na propriedade TraceList e/ou gravadas no arquivo de Log.

Definimos seu valor da seguinte forma:

-1: Indicamos que não possui limites

0: Indicamos que o limite realmente é zero, ou seja, nada será logado

Qualquer valor diferente de 0 e -1, será a quantidade limite de mensagens a serem logadas.

**TraceCount: Integer**

Como o próprio nome já diz, indica a quantidade de mensagens logadas.

### SQLMonitor - Métodos

#### **LoadFromFile(AFileName: string)**

Carrega um arquivo para propriedade TraceList.

#### **Parâmetros**

**AFileName**: Caminho do arquivo, caso não seja informado, será utilizado a propriedade FileName.

#### **SaveToFile(AFileName: string)**

Salva o conteúdo da propriedade TraceList em um arquivo.

#### **Parâmetros**

**AFileName**: Caminho do arquivo, caso não seja informado, será utilizado a propriedade FileName.

### SQLMonitor - Eventos

#### **OnLogTrace (Sender: TObject; CInfo: pSQLTRACEDesc)**

Ocorre logo após a mensagem interceptada ter sido adicionada à propriedade TraceList e antes de a mesma ter sido inserida no arquivo de Log (caso a propriedade Autosave esteja ligada).

#### **Parâmetros**

**Sender:** Representa o próprio componente SqlMonitor

**CInfo:** pSQLTRACEDesc

Representa um ponteiro para o tipo SQLTRACEDesc, que é uma estrutura da mensagem interceptada. Podemos obter a string da mensagem utilizando: *CInfo.pszTrace*

#### **OnTrace (Sender: TObject; CInfo: pSQLTRACEDesc; var LogTrace: Boolean)**

Ocorre antes de a mensagem interceptada ter sido adicionada à propriedade TraceList e ao arquivo de Log.

Neste evento também podemos impedir que determinada mensagem seja logada, desligando o parâmetro LogTrace que recebemos. Ideal para fazermos qualquer tipo de tratamento com a mensagem interceptada, inclusive podendo modificá-la.

#### **Parâmetros**

**Sender:** Representa o próprio componente SqlMonitor

**CInfo:** Mesmo tipo de parâmetro do evento OnLogTrace, a diferença é que, neste momento, podemos ajustar seu valor para que seja refletida na propriedade TraceList e no arquivo de Log.

**LogTrace:** Define se a mensagem será logada ou não.



### SimpleDataSet

Como já mencionamos, todo DataSet proveniente da dbExpress é unidirecional, para visualizar os registros e editá-los, temos que usá-lo em conjunto com algum componente que trabalha com cache em memória, no caso o ClientDataSet.

Com o SQLSimpleDataSet isso não é necessário, pois ele já faz o papel do ClientDataSet também, é o famoso 3 em 1 (ClientDataSet + Provider + DataSet). Na versão 6 do Delphi, tínhamos o TSQLClientDataSet, devido aos diversos bugs, foi substituído por este componente.

A grande vantagem deste componente é sua simplicidade, pois evitamos trabalhar com 3 componentes separados, todos já estão embutidos nele. Porém a grande desvantagem é que nos limita a trabalharmos no modelo 2 camadas, gerando uma grande manutenção quando for necessária a migração para o modelo multicamadas, já que neste modelo o Provider/DataSet estarão no servidor, portanto é necessário que estejam separados.

### Propriedades

#### Connection: TSQLConnection

Podemos associar a esta propriedade um componente SQLConnection ou utilizar o que já existe internamente, neste caso, basta deixarmos como 'InternalConnection' (limpando o valor da propriedade) e clicar no sinal de + para ajustar as propriedades do SQLConnection interno.

#### DataSet: TDataSet

Nesta propriedade temos um DataSet (TSQLDataSet) que será utilizado pelo Provider internamente. Clicando no sinal de +, veremos todas as propriedades que já estudamos deste componente.

***As demais propriedades, eventos e métodos serão vistos diretamente no componente ClientDataSet, pois ambos são derivados de TCustomClientDataSet, portanto, terão a mesma funcionalidade.***





### SQLClientDataSet

Como havíamos comentado anteriormente, este é o componente que existe no Delphi 6 e foi substituído pelo TSimpleDataSet no Delphi 7 devido a diversos bugs.

Não entraremos em detalhes deste componente, pois o mesmo é Deprecated, ou seja, está descontinuado, portanto não é recomendável sua utilização.



### ClientDataSet

#### Introdução

O ClientDataSet é um componente derivado de um TDataSet, com grande potencial para desenvolvimento de aplicações multicamadas, porém muito utilizado também em aplicações 2 camadas (cliente/servidor).

Outro ponto importante desta ferramenta, é o fato de tornar a DBExpress bi-direcional, pois como vimos, ele é unidirecional, precisando de algum componente que trabalhe com cache.

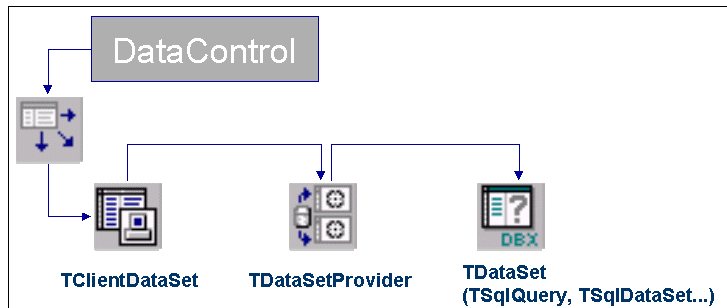
O ClientDataSet não é um componente muito complexo, pois por ser derivado de um TDataSet, contém quase todos os métodos e propriedades que uma TTable possui por exemplo, diferenciando apenas no conceito de tratamento de registros e ligações entre os componentes dependentes.

Uma grande mudança de conceito comparando ClientDataSet com TTable está na forma que ele manipula os dados. Este componente trabalha 'desconectado' do banco, ou seja, primeiro extraímos os registros do servidor, depois fazemos as manutenções em memória e ao final, aplicamos as atualizações no banco de dados. Isso é muito eficaz, pois o ganho de performance nas operações em memória é muito maior e evitamos também permanecer com transações abertas durante a manutenção dos registros.

Podemos perceber o ganho de performance quando abrimos um range de registros e chamamos o método FindKey, Locate, Filter, etc., ou até mesmo a ordenação dos registros por uma determinada coluna, a operação é feita de modo instantâneo, pois os dados estão em memória.

### ClientDataSet - Funcionamento

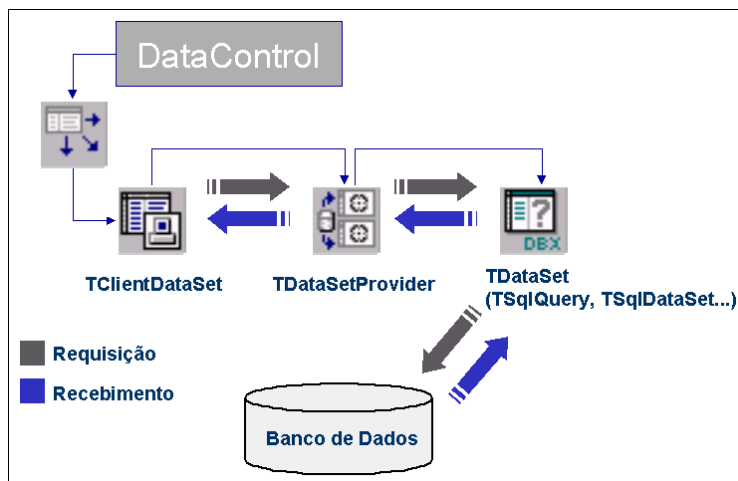
#### Ligação entre os componentes



Ligação entre os componentes

O ClientDataSet trabalha sempre em conjunto: TClientDataSet, TDataSetProvider e TDataSet. Ele se conecta ao TDataSetProvider e este último se conecta a um TDataSet que será responsável em obter os dados do banco. Manipulamos os registros sempre no ClientDataSet, então os métodos Insert, Append, Edit, Post, entre outros, deverão ser executados no próprio ClientDataSet.

#### Requisição / Recebimento de Dados

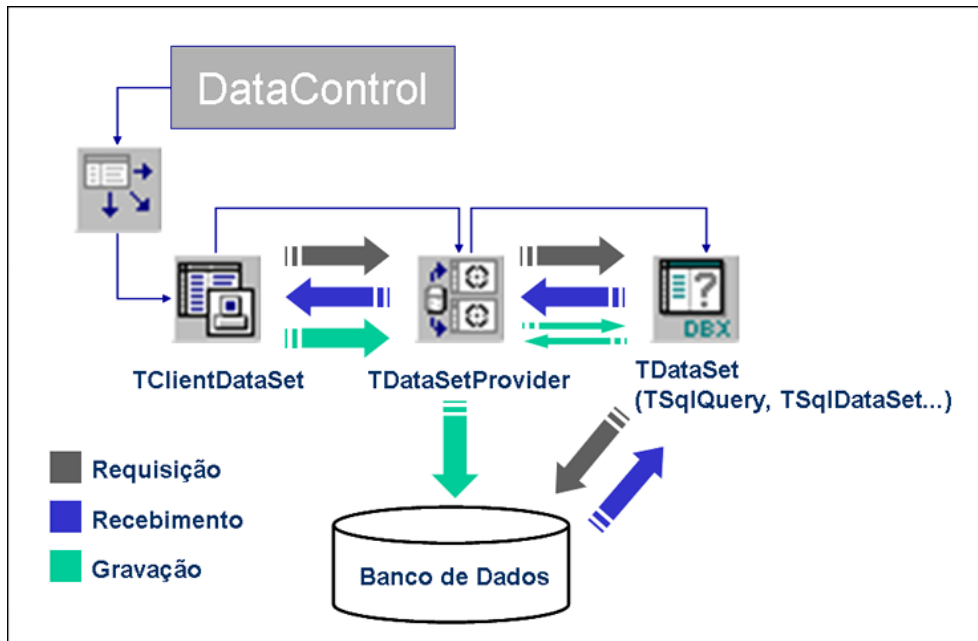


Requisição / Recebimento de dados

Quando fazemos uma requisição de dados, ocorre o seguinte:

O ClientDataSet faz a requisição ao DataSetProvider, logo este abre uma transação e em seguida faz a requisição ao DataSet no qual está ligado, este DataSet faz a leitura dos dados no banco e os retornam ao DataSetProvider, em seguida o DataSetProvider retorna estes dados ao ClientDataSet e a transação é encerrada. Desta forma, os dados ficam na memória do ClientDataSet para podermos realizar todas operações e ao final gravarmos os dados fisicamente no banco.

### Gravação dos Dados



Gravação dos dados

Quando pedimos para o ClientDataSet gravar os dados, ele envia ao Provider todas as operações realizadas (inclusão, alteração, exclusão). Em seguida, o Provider abre uma transação, obtém algumas informações do DataSet ao qual está ligado e executa instruções SQLs no banco de acordo com as operações realizadas. Feito isso, se tudo foi executado com sucesso, a transação é finalizada com um Commit, caso contrário, com um RollBack.

### ClientDataSet - Recursos / Vantagens

#### Compatibilidade com qualquer banco de dados e tipo de conexão.

Podemos utilizá-lo com qualquer banco e tipo de conexão, sem a necessidade de grandes modificações quando for necessária a troca. Isso ocorre pois, como vimos anteriormente, não é o ClientDataSet que se conecta diretamente no banco e sim o DataSet ao qual o Provider está ligado. Então se estamos utilizando por exemplo TQuery (BDE) no DataSet do Provider, podemos trocá-lo por um TSqlQuery (dbExpress) sem alterar nada no ClientDataSet.

#### Redução no tempo de transações abertas

Pelo fato de o ClientDataSet trabalhar com os dados em memória, desconectado do banco, ele não necessita de uma transação aberta enquanto edita os dados por exemplo. Conforme demonstrado na figura de Requisição/Recebimento, percebemos que a transação é aberta somente na leitura dos dados mas logo em seguida é encerrada, sendo aberta novamente somente na atualização dos dados no servidor.

#### Dados somente em memória

Um recurso muito interessante é a possibilidade de termos um ClientDataSet somente em memória, sem a necessidade da existência física de um banco para manipulação dos dados.

Este recurso é muito útil naquela situação típica onde temos um relatório complexo envolvendo diversas tabelas e precisamos unir todos os dados numa única tabela. O caminho mais comum a ser seguido, é a criação de uma tabela física temporária. Com o ClientDataSet, não há esta necessidade, pois podemos definir a estrutura da tabela no componente, alimentá-lo e utilizar como um DataSet qualquer, sem a necessidade da existência física da tabela. Neste caso, não teríamos o Provider/DataSet envolvidos.

#### Gravação dos dados em outros formatos

Podemos utilizar o ClientDataSet para extrair/gravar os dados não somente em um banco de dados, mas também em arquivos binários e XML.

#### Ordenação dos registros em memória, sem a necessidade de índices físicos

Podemos ordenar os registros diretamente em memória sem precisar fazer uma requisição ao banco de dados, evitando assim a necessidade da existência de um índice físico no servidor.

Os registros podem ser ordenados em memória através das propriedades IndexName ou IndexFieldNames, o resultado é instantâneo, justamente pelo fato de a ordenação ser em memória.

Obtemos uma melhor performance no caso de índices persistentes, no qual podemos criá-los no ClientDataSet através da propriedade IndexDefs possibilitando inclusive definir diversos aspectos, tais como: Ascendente, Descendente, Primário, Único, Case-Insensitive, etc.

#### Clones

Clonar é outro recurso interessante do ClientDataSet. Como o próprio nome diz, é simplesmente ter uma cópia fiel de um ClientDataSet em outro.

Quando clonamos um ClientDataSet, toda e qualquer alteração feita no ClientDataSet *original*, refletirá no ClientDataSet *clone* e vice-versa.

O mais interessante é que cada um possui controle do seu estado e posição do cursor, por exemplo, se chamarmos o método Append de um ClientDataSet, o **clone** não entrará em modo de edição, se mudarmos a posição do registro em um ClientDataSet, o outro será mantido no mesmo local.

Um exemplo de utilização deste recurso é:

Em um cadastro de Pedidos com uma tabela Detail de Produtos, caso o usuário insira um produto que já esteja no ClientDataSet Detail, teremos que alertá-lo, impedindo esta situação. Então, neste caso, teríamos que fazer um FindKey na tabela Detail antes de gravar o produto, porém isso daria um certo trabalho, já que a tabela Detail estaria em modo de edição e não poderíamos chamar o método FindKey, pois ele gravaria o registro.

Neste caso vem a grande utilidade do **clone**, pois basta fazermos um clone do ClientDataSet Detail e chamarmos o método FindKey no **clone**, pois desta forma não afetaria o estado de edição do ClientDataSet original.

### Novos Tipos de Campos: InternalCalc, Aggregate

#### Campos InternalCalc

Este tipo de campo é bem semelhante o campo Calculate, porém com algumas diferenças:

#### **Editável a qualquer momento**

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

#### **Ganho de performance**

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

#### **Indexação**

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

### **Campos Aggregate**

O campo Aggregate, como o próprio nome diz, é um campo agregado que criamos no ClientDataSet para executar operações com base em um conjunto de dados, elas podem ser: *Sum, Avg, Count, Min e Max*.

Uma situação muito comum é aquela na qual temos uma lista de registros e no final precisamos exibir a soma de uma determinada coluna.

Existem várias formas de fazermos isso, poderíamos fazer uma query com “select sum” ou fazer um loop no DataSet totalizando a coluna de valor.

Com o campo Aggregate, esta tarefa se torna muito simples, pois criaríamos um campo do tipo Aggregate e ajustaríamos apenas algumas propriedades. Feito isso, associaríamos este campo a um TDbText para o valor ser exibido.

A maior vantagem está no fato de que esta operação é feita em memória, assim ganhamos performance e evitamos fazer requisições ao servidor para este tipo de cálculo.

### **SavePoint – Desfazendo alterações a partir de um ponto**

O ClientDataSet nos disponibiliza diversas formas para desfazermos as alterações feitas nos registros, porém vale a pena destacarmos o recurso **SavePoint** devido a sua grande eficácia.

Este recurso permite marcarmos um ponto inicial no ClientDataSet, fazer as manutenções nos registros e ao final, voltar ao ponto em que marcamos, desfazendo tudo o que foi feito a partir deste ponto.

### **Aplicações multicamadas (n-tier)**

Este é um dos recursos principais do ClientDataSet.

Aplicações multicamadas é uma aplicação dividida em várias camadas: interface(cliente), regras de negócio (servidor de aplicação) e banco de dados.

Normalmente desenvolvemos aplicações 2 camadas, onde temos apenas a parte interface (cliente) e o servidor de banco de dados, sendo que as regras de negócios podem estar no cliente ou no banco de dados.

Colocando as regras de negócios no lado cliente, exigiremos que o cliente tenha uma boa máquina para executar a aplicação. Já colocando as regras de negócios no banco de dados, poderemos ter uma queda de performance no lado do servidor se tiverem muitos processos sendo executados simultaneamente.

O desenvolvimento de aplicações multi-camadas resolve estes tipos de problemas, pois o lado cliente será bem leve, apenas a interface, não tendo nenhum tipo de processo pesado, e as regras de negócios estarão em outra camada, na qual podemos distribuir em diversos servidores evitando assim a sobrecarga em um único servidor.

No caso do servidor de banco de dados, ele será responsável apenas para armazenar e retornar os dados, não será mais responsável pelas regras de negócio.

Podemos exemplificar o caso de uma aplicação robusta, onde temos diversos processos do tipo: Fechamento de Caixa, Fechamento de Estoque, Baixas de Duplicatas, etc..

Cada um desses, poderia ser um processo pesado sendo requisitado ao mesmo tempo. Se for uma aplicação 2 camadas e as regras de negócios estiverem no banco de dados, os processos poderão ficar lentos, pois tudo estará sendo processado no servidor de banco de dados, que estará em uma única máquina, sem a possibilidade de dividir as tarefas para outras máquinas.

Já com aplicações multicamadas podemos ter um, dois, três ou até mais servidores, sendo que um seria dedicado para o banco de dados e os demais seriam para processar as regras de negócios, por exemplo. Desta forma, teríamos um grande ganho de performance, pois estaríamos dividindo a tarefa de cada servidor.



### ClientDataSet – Disponibilidade, Licença e Distribuição

#### Disponibilidade e Licença

O ClientDataSet está disponível desde a versão 3 do Delphi. A partir da versão 7, ele é 100% free para utilizarmos em nossas aplicações, já nas versões anteriores é necessário o pagamento de licenças (royalties) caso esteja utilizando no modelo multicamadas.

#### Distribuição

Quando desenvolvemos uma aplicação utilizando ClientDataSet, temos que distribuir o arquivo **midas.dll**, no qual deverá ser adicionado ao diretório "Windows\System" da máquina, ou se preferirmos, podemos incluir a **unit MidasLib** em nossa aplicação ao invés do uso da dll. Optando pela distribuição da dll, em alguns casos é necessário registrá-la utilizando o utilitário **regsvr32.exe** (ex: regsvr32 midas.dll).

### ClientDataSet - Propriedades

**Active: Boolean**

Indica se o DataSet está ativo.

**ActiveAggs[Index: Integer]: TList**

Retorna a lista dos Aggregates ativos para um determinado GroupingLevel que é especificado no parâmetro Index.

**Aggregates: TAggregates**

Lista de todos objetos TAggregate aplicados ao ClientDataSet.

Um objeto Aggregate, é um objeto capaz de calcular um valor agregado (SUM, AVG, MIN, MAX) com base nos dados do ClientDataSet.

Podemos também criar campos agregados, sem a necessidade de trabalhar diretamente com o objeto TAggregate, pois este trabalho já é feito automaticamente quando criamos este tipo de campo.

**AggregatesActive: Boolean**

Define se o cálculo de valores agregados está ativo. O padrão é false para evitar uma sobrecarga, portanto, deve ser ativado somente quando necessário.

**AppServer: IAppServer**

Prove acesso à interface do ClientDataSet usada para comunicar com o Provider. Não é necessário acessarmos diretamente o AppServer, todos os métodos e propriedades do ClientDataSet já fazem a comunicação automaticamente quando necessário.

**CanModify: Boolean**

Indica se o ClientDataSet pode ser modificado (insert, edit, delete).

Quando ligamos a propriedade ReadOnly, esta propriedade automaticamente passa a ser False.

Outro caso que esta propriedade sofre alteração é quando ajustamos a propriedade Options do Provider ao qual o ClientDataSet está ligado, nele podemos especificar se serão permitidos inserções, edições ou exclusões.

**ChangeCount: Integer**

Retorna o número de alterações pendentes no ClientDataSet que não foram enviadas para o banco de dados.

**CloneSource: TCustomClientDataSet**

Retorna o clone do ClientDataSet, ou seja, retorna o ClientDataSet que compartilha o mesmo cursor, mesmos dados.

Como já vimos, uma das vantagens do ClientDataSet, é a possibilidade de fazer Clones. Mais adiante veremos detalhes do método CloneCursor.

**CommandText: string**

Especifica o comando SQL a ser enviado ao servidor de banco de dados. Seria o mesmo que definirmos a instrução SQL no DataSet ao qual o Provider está ligado, porém com algumas vantagens e desvantagens.

Quando estamos utilizando o `SQLDataSet` por exemplo, esta propriedade poderá representar o próprio comando SQL, nome da Tabela ou Stored Procedure dependendo do que foi definido na propriedade `CommandType` do `SQLDataSet`.

Uma grande vantagem de sua utilização é em relação à portabilidade. Se precisarmos trocar o tipo de acesso DBExpress por outro, poderemos fazer sem grandes preocupações, pois as instruções SQL sempre estarão no `ClientDataSet`.

A grande desvantagem é quando trabalhamos no modelo multicamadas, pois um dos grandes pontos positivos acaba sendo perdido, que é distribuição de regras de negócios em servidores ao invés de centralizá-las na aplicação cliente, evitando sobrecarga e a manutenção no lado dos clientes. Esta vantagem será perdida, pois qualquer mudança em alguma instrução SQL deverá ser feita no `ClientDataSet`, e o mesmo sempre estará no lado cliente neste modelo.

### **ConnectionBroker: TConnectionBroker**

Especifica o componente `ConnectionBroker` que manipulará a conexão entre o cliente e o servidor de aplicação, utilizado no modelo multicamadas.

### **Data: OleVariant**

Representa os dados do `ClientDataSet` em seu formato original, empacotado. Estes dados são originados do Provider, de um arquivo ou de outro `ClientDataSet`.

### **DataSetField: TDataSetField**

Determina o `TDataSetField` utilizado pelo `ClientDataSet`.

Quando trabalhamos com `NestedDataset` (outra forma de mestre/detalhe), utilizamos esta propriedade no `ClientDataSet` detail ao invés de usarmos o Provider para obtermos os dados, pois neste modelo, os dados são mantidos no `ClientDataSet` Master e nele estará disponível um campo do tipo `TDataSetField` que utilizaremos nesta propriedade.

### **DataSetSize: Integer**

Indica o número de bytes ocupados pelos dados (propriedade `Data`) do `ClientDataSet`.

### **DataSource: TDataSource**

Indica o `DataSource` no caso de o `ClientDataSet` ser um detail em um relacionamento mestre/detalhe. É o mesmo que acessarmos a propriedade `MasterSource`, porém esta é `ReadOnly`.

### **Delta: OleVariant**

Representa o pacote de alterações logadas no `ClientDataSet`, neste pacote contém todas inserções, modificações e exclusões ocorridas no `DataSet`.

É com base neste pacote que o Provider atualiza o banco de dados quando chamamos o método `ApplyUpdates` do componente. Após a atualização no banco, o Delta é limpo para que não acumule log de alterações, porém isso dependerá do parâmetro passado ao método `ApplyUpdates` que indica o número de erros permitidos na atualização. Veremos mais detalhes na explicação deste método.

### **DisableStringTrim: Boolean**

Determina como será o tratamento de espaços em branco (espaços à direita) nos campos quando forem gravados.

Especificando `True`, os valores não serão alterados, serão gravados da forma que foram preenchidos, ou seja, se tiverem espaços em branco à direita, serão permanecidos. Já deixando como `False`, serão retirados os espaços em branco.

### **FetchOnDemand: Boolean**

Determina a forma que será controlada a busca de registros no Provider quando utilizamos a propriedade `PacketRecords`, que especifica quantos registros devem ser transferidos ao `ClientDataSet` na requisição de um pacote. O padrão é -1, ou seja, todos registros obtidos pelo Provider serão transferidos ao `ClientDataSet`, isso é o que utilizamos no modelo 2 camadas, já em multicamadas, podemos especificar uma quantidade para diminuir o tráfego na rede, já que os registros estarão centralizados em outra máquina onde estará o Provider.

Utilizando `PacketRecord` com um valor maior que -1, caso a propriedade `FetchOnDemand` esteja como `True` (padrão), ao chegarmos no último registro do `ClientDataSet` ele automaticamente fará uma nova requisição ao Provider para que o mesmo possa enviar mais registros a ele, de acordo com o número especificado em `PacketRecord`. Caso deixemos esta propriedade `FetchOnDemand` como `False`, seremos obrigados a fazer o processo de requisição manualmente através do método `GetNextPacket`.

Esta propriedade também influencia quando trabalhamos com `NestedDataset` e campos BLOB para podermos requisitar sob demanda. Por exemplo, incluindo a opção *`poFetchBlobsOnDemand`* na propriedade `Options` do Provider, o mesmo manterá os campos BLOBS, então podemos desligar a propriedade `FetchOnDemand` e chamar manualmente o método `FetchBlobs` do `ClientDataSet` para obtermos esses campos somente quando necessário. No caso do `NestedDataset` o conceito é o mesmo, podemos requisitar somente quando necessário, neste caso o método a ser chamado é o *`FetchDetails`* e a opção a ser ligada na propriedade `Options` do Provider é a *`poFetchDetailsOnDemand`*.

### **FileName: string**

Como já vimos, uma das vantagens do `ClientDataSet`, é a possibilidade de trabalhar sem a necessidade da existência de um servidor de banco de dados. Podemos também trabalhar com as informações somente em memória ou gravá-las diretamente em um arquivo.

Nesta propriedade especificamos o nome do arquivo de onde os dados serão lidos e gravados. Podemos também trabalhar com os métodos `LoadFromFile` e `SaveToFile` que veremos mais adiante.

### **Filter: string**

Especifica um filtro para os registros do `ClientDataSet`. Este filtro é aplicado em memória, nos dados mantidos no `ClientDataSet`, nenhuma requisição é feita ao banco de dados para que o filtro seja aplicado.

### **GroupingLevel: Integer**

Retorna o `GroupingLevel` suportado pelo índice atual.

O `GroupingLevel` é especificado quando criamos um campo `Aggregate` ou um índice, sua função é determinar como os registros serão agrupados.

Definindo o valor 1, o agrupamento será pelos registros que tiverem o mesmo valor no primeiro campo contido no índice atual. Se definirmos `GroupingLevel` 2, então o agrupamento será pelos registros que tiverem o mesmo valor no primeiro e segundo campo contido no índice atual e assim por diante. Caso o `GroupingLevel` seja zero (padrão), não haverá agrupamentos.

Esses agrupamentos são muito utilizados quando trabalhamos com campos `Aggregates`, em situações onde queremos fazer uma sumarização por um determinado grupo de registros por exemplo.

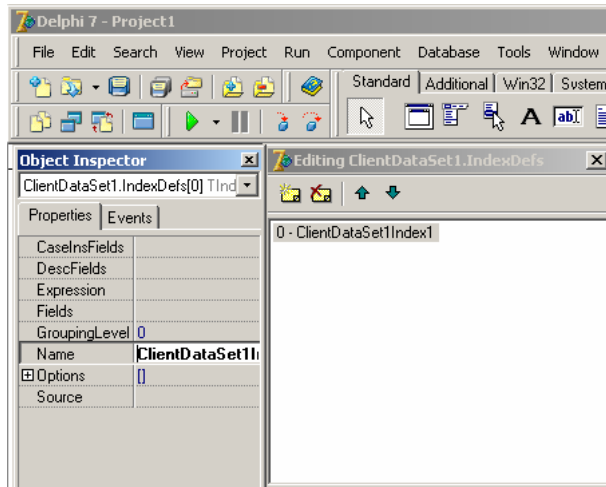
### **HasAppServer: Boolean**

Indica se o `ClientDataSet` está conectado a um Provider local ou remoto.

### IndexDefs: TIndexDefs

Definição dos índices criados no ClientDataSet. Estes índices não são os índices do banco de dados, mas sim os que criamos no próprio ClientDataSet.

Ao clicarmos nesta propriedade, a seguinte tela é exibida:



#### Editor de índices

Nesta tela podemos criar os índices clicando com o botão direito do mouse e acessando a opção de menu **Add**.

Para cada índice, temos as seguintes propriedades:

**CaseInsFields:** Especifica quais são os campos case-insensitive

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

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

**Fields:** Contém os campos pertencentes ao índice.

**GroupingLevel:** Nível de agrupamento, como explicado anteriormente na propriedade GroupingLevel do ClientDataSet.

**Options:** Opções do índice (primário, único, case-insensitive, etc.)

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

### IndexFieldCount: Integer

Retorna o número de campos do índice atual.

### IndexFieldNames: string

Determina por quais campos os registros do ClientDataSet serão ordenados. No caso de ordenações simples, podemos especificar diretamente o nome dos campos nesta propriedade ao invés de utilizarmos a propriedade IndexName, tendo que criar o respectivo índice na propriedade IndexDefs.

### IndexFields[Index: Integer]: TField

Permite o acesso aos campos do índice atual.

### IndexName: string

Especifica o nome do índice a ser utilizado no ClientDataSet quando criamos através da propriedade IndexDefs.

**KeyExclusive: Boolean**

Determina se no Range aplicado serão inclusos ou não os registros que coincidirem exatamente com os valores iniciais e finais do Range. O padrão é false, significa que são inclusos, sendo True, não farão parte do Range.

**KeyFieldCount: Integer**

Determina o número de campos utilizados em uma busca quando o índice é composto por múltiplos campos.

Por exemplo, se estivermos utilizando um índice composto por 3 campos, definindo a propriedade KeyFieldCount para 2, a busca será realizada com base apenas nos dois primeiros campos.

**KeySize: Word**

Retorna o tamanho atual da chave utilizada pelo ClientDataSet. Corresponde ao tamanho dos campos chave do índice primário.

**LogChanges: Boolean**

Determina se as alterações feitas no ClientDataSet serão logadas para serem aplicadas ao banco de dados quando chamarmos o método ApplyUpdates. O valor padrão é True, ou seja, tudo que é feito é logado no Delta para que o Provider possa saber as operações ocorridas e aplicar ao banco. Quando desligamos esta propriedade, as alterações serão feitas apenas no ClientDataSet sem que sejam logadas no Delta, conseqüentemente, chamando o método ApplyUpdates, nada será aplicado ao banco.

É interessante desligarmos quando trabalhamos com dados ReadOnly ou em caso de extrairmos as informações através de um arquivo, pois já que não serão gravadas no servidor de banco de dados, não se faz necessário logarmos, pois o Provider não estará em ação. Desta forma estamos evitando o consumo de recursos desnecessários.

**MasterFields: string**

Utilizado para relacionamentos mestre/detalhe. Indica o nome do campo chave no relacionamento.

**MasterSource: TDataSource**

Utilizado para relacionamentos mestre/detalhe. Indica o DataSource que está ligado ao DataSet Master.

**PacketRecords: Integer**

Determina o número ou tipo de registro a ser incluso no pacote de dados recebido do Provider.

Esta propriedade é muito utilizada no modelo multicamadas, pois os registros ficam centralizados no Provider que estará em um servidor, então podemos determinar quantos registros ele deverá nos enviar por cada requisição, evitando assim um tráfego de rede e ganhando performance dependendo do caso.

Podemos especificar:

- 1: Todos registros serão inclusos no pacote.
- 0: Utilizado para extrair o metadado do DataSet
- >0: Indica a quantidade de registros por pacote

**Params: TParams**

Define os parâmetros a serem enviados ao Provider para que sejam utilizados no DataSet ao qual está ligado.

Para que os parâmetros sejam enviados e utilizados corretamente, devemos criá-los com os mesmos nomes e tipos correspondentes aos parâmetros da Query ou Stored Procedure a ser executada. Podemos utilizar o método `FetchParams` do `ClientDataSet`, pois desta forma ele alimentará a propriedade `Params` com todos os parâmetros pré-definidos no `DataSet` do `Provider`.

**ProviderName: string**

Define o `Provider` do `ClientDataSet`, utilizado para extrair os dados e aplicar as atualizações no banco.

No modelo 2 camadas, o `Provider` fica no mesmo local onde está o `ClientDataSet`, portanto nesta propriedade indicamos o `'Name'` do `Provider`. Já no modelo multicamadas, o `Provider` estará em um servidor de aplicação, em outra máquina, neste caso deverá ser acessado remotamente, onde o mesmo poderá ser registrado no sistema e conseqüentemente será listado nesta propriedade.

**ReadOnly: Boolean**

Determina se o `ClientDataSet` é somente leitura. Especificando `True` nesta propriedade, não poderemos editar, excluir ou incluir registros. Esta definição também poderá ser feita através da propriedade `Options` do `Provider` que veremos mais adiante.

**RecNo: Integer**

Retorna o número do registro atual. Podemos nos posicionar em um determinado registro ajustando o valor desta propriedade.

**RecordCount: Integer**

Retorna a quantidade de registros existentes no `ClientDataSet`.

**RecordSize: Integer**

Retorna o tamanho físico em bytes do registro atual.

**RemoteServer: TCustomRemoteServer**

Define o componente de conexão que o `ClientDataSet` utilizará para se comunicar com o `Provider` quando ele está em um servidor de aplicação, utilizado no modelo multicamadas. Também podemos utilizar a propriedade `ConnectionBroker`, desta forma centralizamos e abstraímos a conexão facilitando futuras manutenções que poderão surgir caso precisemos trocar o tipo de protocolo por exemplo.

**SavePoint: Integer**

Retorna o estado atual das alterações logadas no `ClientDataSet`. Normalmente utilizamos esta propriedade para reverter alterações ocorridas no `ClientDataSet` a partir de um certo ponto, para isso, bastaríamos guardar o estado atual (valor da propriedade) em uma variável e em determinado momento, podemos voltar ao estado original atribuindo a esta propriedade o valor guardado na variável, obtido antes de iniciar as alterações.

**StatusFilter: TUpdateStatusSet**

Define um filtro de registros por status. Podemos filtrar os registros que estejam em determinados estados (inseridos, alterados, excluídos). Nesta propriedade definimos um conjunto de valores, que podem ser: `usUnmodified`, `usModified`, `usInserted` e `usDeleted`.

**StoreDefs: Boolean**

Determina se as definições de índices e campos serão salvas juntos com o `ClientDataSet`, no `Form` ou `DataModule` que estiver.

### **XMLData: OleVariant**

Representa os dados do ClientDataSet em formato XML.



### ClientDataSet - Métodos

**AddIndex(const Name, Fields: string; Options: TIndexOptions; const DescFields: string = ""; const CaseInsensitiveFields: string = ""; const GroupingLevel: Integer = 0 )**

Cria um índice no ClientDataSet.

#### Parâmetros

**Name:** Nome do índice

**Fields:** Campos pertencentes ao índice

**Options:** Opções do índice, tais como: único, secundário, case-insensitive.

**DescFields:** Campos que serão ordenados em ordem decrescente.

**CaseInsensitiveFields:** Campos case-insensitive.

**GroupingLevel:** GroupingLevel do índice (este conceito já foi abordado na propriedade GroupingLevel do ClientDataSet).

**AppendData(const Data: OleVariant; HitEOF: Boolean);**

Adiciona um novo pacote de registros ao ClientDataSet, que poderá ser obtido através de um Provider ou até mesmo de outro ClientDataSet (passando a propriedade Data como parâmetro).

#### Parâmetros

**Data:** Pacote de dados

**HitEOF:** Diante de alguns testes, não percebemos diferenças passando True ou False, mas no help diz:

'HitEOF indicates whether the provider encountered the end of the dataset when it fetched the records from the database server.'

#### **ApplyRange**

Aplica o range estabelecido pelos métodos SetRangeStart e SetRangeEnd.

**ApplyUpdates(MaxError: Integer): Integer**

Aplica todas alterações pendentes (inserções, modificações e exclusões) ao banco de dados e nos retorna o número de erros ocorridos.

#### Parâmetros

**MaxError:** Indica o número de erros permitidos na atualização.

0: Indica que não permitimos nenhum erro.

-1: Indica que não há limites de erros.

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

Este parâmetro afeta a forma que será tratada as pendências quando ocorrerem erros nas atualizações. Caso o número de erros não ultrapasse o número permitido, as pendências que não tiveram erros serão aplicadas ao banco de dados, caso contrário, tudo será descartado, não sendo aplicado nada ao banco de dados.

Seja qual for a situação, todas pendências que sofrerem erros permanecerão no ClientDataSet, portanto se chamarmos o método ApplyUpdates na sequência, elas serão aplicadas novamente ao banco de dados. Podemos eliminar utilizando o método CancelUpdates.

**BookmarkValid(Bookmark: TBookmark): Boolean**

Verifica se um determinado Bookmark é válido.

### Parâmetros

**Bookmark:** Bookmark a ser testado.

### **Cancel**

Cancela a edição ou inserção do registro.

### **CancelRange**

Cancela o range aplicado.

### **CancelUpdates**

Cancela as atualizações pendentes no ClientDataSet limpando o log de pendências.

**CloneCursor(Source :TCustomClientDataSet; Reset: Boolean; KeepSettings: Boolean = False);**

Como o próprio nome diz, clona um cursor. Podemos ter uma cópia fiel de um ClientDataSet clonando seu cursor em outro ClientDataSet, desta forma qualquer mudança ocorrida em um, afetará o outro. Os registros apontarão sempre para o mesmo ponteiro (endereço de memória).

### Parâmetros

**Source:** Indica o ClientDataSet a ser clonado.

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

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

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

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

**CompareBookmarks (Bookmark1, Bookmark2: TBookmark): Integer;**

Compara 2 Bookmark's retornando um número caso haja diferença entre eles. Retornando 0, significa que os Bookmarks são idênticos.

### Parâmetros

**Bookmark1:** Bookmark a ser comparado com o Bookmark2.

**Bookmark2:** Bookmark a ser comparado com o Bookmark1.

### **ConstraintsDisabled: Boolean**

Retorna se as Constraints estão ou não desabilitadas no ClientDataSet.

**CreateBlobStream (Field: TField; Mode: TBlobStreamMode): TStream;**

Retorna a instância de um Stream a partir de um campo blob.

Quando utilizamos este método, devemos nos atentar à propriedade FetchOnDemand do ClientDataSet, pois caso esteja desligada e o Provider não estiver com o campo BLOB incluso em seu pacote, deveremos chamar o método FetchBlobs antes de executar o CreateBlobStream.

### Parâmetros

**Field:** Campo BLOB que será utilizado como base para instanciar o Stream.

**Mode:** Modo como o Stream será criado (bmRead, bmWrite ou bmReadWrite).

### **CreateDataSet**

Cria o DataSet com a estrutura definida e o abre em seguida. Este método é utilizado quando trabalhamos com o ClientDataSet somente em memória, onde não extraímos os dados do servidor, definimos a estrutura da tabela no próprio ClientDataSet, portanto executamos este método ao invés de Open.

Ele também poderá ser executado em tempo de projeto clicando com o botão direito do mouse no componente e em seguida no item CreateDataSet.

### **DataRequest(Data: OleVariant): OleVariant**

Chama o evento OnDataRequest do Provider que está associado. Esta função é utilizada quando precisamos fazer algum tipo de comunicação com o Provider externo, então passamos determinada informação por parâmetro e obtemos um resultado vindo do Provider.

#### **Parâmetros**

**Data:** Informação a ser enviada ao Provider

### **DeleteIndex(const Name: string)**

Remove um índice específico criado no ClientDataSet.

#### **Parâmetros**

**Name:** Nome do índice a ser removido

### **DisableConstraints**

Desabilita as Constraints do ClientDataSet, isto pode ser interessante dependendo o caso, para ganho de performance.

### **EditKey**

Habilita o ClientDataSet para realizar uma busca. Este método é utilizado em conjunto com o método GotoKey, após executarmos, definimos os valores dos campos e em seguida chamamos o método GotoKey para localizar o registro.

### **EditRangeEnd**

Habilita definição de valores finais nos campos quando estamos trabalhando com Range. Utilizado em conjunto com os métodos EditRangeStart e ApplyRange.

### **EditRangeStart**

Habilita definição de valores iniciais nos campos quando estamos trabalhando com Range. Utilizado em conjunto com os métodos EditRangeEnd e ApplyRange.

### **EmptyDataSet**

Exclui todos registros do ClientDataSet.

### **EnableConstraints**

Habilita as Constraints do ClientDataSet.

### **Execute**

Executa uma Query ou Stored Procedure ligada ao Provider que não retorne um cursor.

### **FetchBlobs**

Obtém os valores dos campos BLOBS que estão armazenados no Provider e não foram enviados ao ClientDataSet.

Devemos chamar este método quando trabalhamos com a propriedade `FetchOnDemand` do `ClientDataSet` desligada e configuramos o `Provider` para armazenar os campos `BLOBS`, pois assim os dados dos campos `BLOBS` não estarão diretamente no `ClientDataSet` e seremos obrigados a buscá-los no `Provider` utilizando este método.

### **FetchDetails**

Possui o mesmo conceito do método `FetchBlobs`, porém especificamente para registros `details` quando trabalhamos com `NestedDataset`.

### **FetchParams**

Preenche a propriedade `Params` do `ClientDataSet` com todos os parâmetros definidos no `DataSet` ao qual o `Provider` está ligado.

Executamos este método para trabalharmos com os parâmetros diretamente no `ClientDataSet`, evitando o acesso direto ao `DataSet` do `Provider`, no qual estará na mesma aplicação quando trabalhamos no modelo multicamadas.

### **FindKey (const KeyValues: array of const): Boolean;**

Executa uma busca exata nos registros do `ClientDataSet` com base no índice atual. Esta função retorna `True` se for encontrado o registro ou `False` caso contrário.

#### Parâmetros

**KeyValues:** Valor chave a ser procurada. Podemos informar mais de um valor caso tenhamos mais de um campo no índice. Cada valor é separado por ponto-e-vírgula fazendo referência aos campos existentes no índice atual na respectiva ordem. Caso o número de valores seja inferior ao número de campos definido no índice, os demais serão considerados como `NULL`.

### **FindNearest (const KeyValues: array of const);**

Possui o mesmo conceito do método `FindKey` explicado acima, a diferença está no fato de ser uma busca aproximada ao invés de uma busca exata como ocorre com o `FindKey`.

### **GetCurrentRecord (Buffer: PChar): Boolean;**

Faz uma cópia do registro atual para um `Buffer` alocado pela aplicação.

### **GetFieldData(Field: TField; Buffer: Pointer): Boolean;**

### **GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean): Boolean**

### **GetFieldData(FieldNo: Integer; Buffer: Pointer): Boolean;**

Alimenta a variável `buffer` com o valor do campo e retorna `True` ou `False` indicando se a informação foi obtida com sucesso.

Não é comum a utilização deste método nas aplicações, ele é utilizado na implementação do método `GetData` da classe `TField`.

#### Parâmetros

**Field:** Campo a ser extraído o dado

**Buffer:** Variável a ser alimentada com o valor do campo

**NativeFormat:** Indica se o campo está em seu formato nativo. Caso este parâmetro seja `False`, será feita uma conversão para o formato nativo.

**FieldNo:** Número do campo a ser extraído o dado

### **GetGroupState(Level: Integer): TGroupPosInds**

Retorna a 'posição' do registro dentro do agrupamento atual, utilizado nos casos onde utilizamos `Aggregates` e fazemos uso de índices especificando o `GroupingLevel`.

Esta função retorna um conjunto de valores:

*gbMiddle*: Indica que o registro não é o primeiro nem o último do grupo.

*gbFirst*: Indica que o registro é o primeiro do grupo.

*gbLast*: Indica que o registro é o último do grupo.

*gbFirst,gbLast*: Indica que ele é o primeiro e o último registro, ou seja, é o único registro do grupo.

### **Parâmetros**

**Level**: Determina o nível (GroupingLevel) a ser comparado.

### **GetIndexInfo(IndexName: string)**

Popula internamente o ClientDataSet com informações sobre o índice atual.

### **Parâmetros**

**IndexName**: Nome do índice atual. Se não for informado, não serão extraídas informações sobre o GroupingLevel.

### **GetIndexNames(List: TStrings)**

Extraí a lista de índices disponíveis no ClientDataSet.

### **Parâmetros**

**List**: Lista a ser alimentada com os índices disponíveis.

### **GetNextPacket: Integer**

Obtém o próximo pacote de registros do Provider e retorna o número de registros obtidos. Caso o valor de retorno seja 0, significa que não há mais registros disponíveis.

Este método é utilizado quando definimos a quantidade de registros por pacote na propriedade PacketRecords e desligamos a propriedade FetchOnDemand, pois assim tornamos este processo manual.

### **GetOptionalParam(const Param: string): OleVariant**

Retorna a informação personalizada definida no ClientDataSet através do método SetOptionalParam.

Podemos guardar uma informação adicional no ClientDataSet e ela será mantida junta ao pacote de dados e gravada quando for salvo em um arquivo ou stream.

### **Parâmetros**

**Param**: Identificador da informação a ser extraída.

### **GotoCurrent(DataSet: TCustomClientDataSet)**

Posiciona o cursor no mesmo registro do DataSet passado no parâmetro. Este deverá ser um clone do DataSet.

### **Parâmetros**

**DataSet**: DataSet no qual queremos nos basear para se posicionar no registro..

### **GotoKey: Boolean**

Executa uma busca exata de um registro pelas chaves especificadas previamente pelos métodos SetKey e EditKey. Esta função retorna True e posiciona o cursor no registro caso encontre-o. Não localizando, a função retorna False e mantém o cursor no registro atual.

### **GotoNearest**

Executa uma busca aproximada de um registro pelas chaves especificadas previamente pelos métodos SetKey e EditKey.

**LoadFromFile(const FileName: string = '')**

Carrega o ClientDataSet a partir de um arquivo.

### **Parâmetros**

**FileName:** Nome do arquivo que contém os dados. Caso não seja especificado, será utilizado o nome informado na propriedade FileName.

### **LoadFromStream(Stream: TStream)**

Carrega o ClientDataSet a partir de um Stream.

### **Parâmetros**

**Stream:** Objeto Stream de onde será carregado.

### **Locate (const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions): Boolean**

Executa uma busca no ClientDataSet por determinados valores. Esta função retorna True se localizou o registro ou False caso contrário.

### **Parâmetros**

**KeyFields:** Nome dos campos separados por ponto-e-vírgula que serão utilizados na busca.

**KeyValues:** Valores dos campos que serão utilizados na busca. Neste parâmetro não separamos os valores com ponto-e-vírgula, sendo mais de um valor, utilizamos a função VarrArrayOf que nos retorna um variant com os valores especificados em seu parâmetro.

**Options:** Conjunto de opções de busca (loCaseSensitive, loPartialKey).

### **Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant**

Funciona da mesma forma que o método Locate explicado anteriormente, porém a diferença é que ao invés de executar a busca, a função retorna um array contendo todos os valores dos campos (do registro encontrado) especificados em ResultFields.

### **Parâmetros**

**KeyFields:** Nome dos campos separados por ponto-e-vírgula que serão utilizados na localização.

**KeyValues:** Valores dos campos que serão utilizados na localização. Neste parâmetro não separamos os valores com ponto-e-vírgula, sendo mais de um valor, utilizamos a função VarrArrayOf que nos retorna um variant com os valores especificados em seu parâmetro.

**ResultFields:** Nome dos campos que deverão ser retornados os valores para a função.

### **MergeChangeLog**

Consolida as alterações logadas no ClientDataSet.

Difícilmente chamaremos este método manualmente quando trabalhamos com Provider, pois o mesmo já é executado automaticamente pelo método ApplyUpdates. Utilizamos quando trabalhamos com o ClientDataSet baseado em arquivos e o LogChanges está ativado.

### **Post**

Salva o registro atual somente em memória. Para gravarmos fisicamente no banco de dados devemos chamar o método ApplyUpdates.

### **Reconcile(const Results: OleVariant): Boolean**

Reconcilia o ClientDataSet de acordo com as mudanças aplicadas.

Este método é chamado internamente pelo ApplyUpdates após o Provider ter executado as atualizações, para que assim o ClientDataSet seja refletido com as mudanças aplicadas.

Esta função retorna True caso tudo tenha sido reconciliado com sucesso, ou False, quando algo não foi reconciliado.

### **Parâmetros**

**Results:** Informações sobre os registros que não foram atualizados com seus respectivos erros. Este parâmetro é passado com o Resultado obtido pelo método ApplyUpdates do Provider.

### **RefreshRecord**

Atualiza o registro para refletir os valores atuais existentes no servidor.

Este método é diferente do método Refresh, pois atualiza apenas o registro atual mantendo e prevalecendo o log de alterações. Já o método Refresh atualiza todos os registros e não permite que tenha alterações pendentes, gerando uma exceção caso seja executado nesta situação.

### **RevertRecord**

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

### **SaveToFile(const FileName: string; Format: TDataPacketFormat=dfBinary)**

Salva os dados do ClientDataSet em arquivo.

### **Parâmetros**

**FileName:** Nome do arquivo de destino. Caso não seja especificado, será utilizado o nome informado na propriedade FileName.

**Format:** Formato dos dados: Binário(dfBinary), XML(dfXML) ou UTF8-based XML(dfXMLUTF8).

### **SaveToStream(Stream: TStream; Format: TDataPacketFormat=dfBinary)**

Salva os dados do ClientDataSet em Stream.

### **Parâmetros**

**Stream:** Stream onde será gravado.

**Format:** Formato dos dados: Binário(dfBinary), XML(dfXML) ou UTF8-based XML(dfXMLUTF8).

### **SetAltRecBuffers(Old, New, Cur: PChar)**

Define um ponteiro para o buffer do registro, que será utilizado quando acessarmos as propriedades OldValue, CurValue e NewValue do campo.

Não é comum utilizarmos este método na aplicação.

### **Parâmetros**

**Old:** Ponteiro para OldValue.

**New:** Ponteiro para NewValue.

**Cur:** Ponteiro para CurValue.

### **SetKey**

Prepara o ClientDataSet para executarmos uma busca. Após chamarmos este método, definimos os valores dos campos e em seguida realizamos a busca com o método GotoKey.

### **SetOptionalParam (const ParamName: string; const Value: OleVariant; IncludeInDelta: Boolean = False)**

Define uma informação personalizada no ClientDataSet. Ela será adicionada junta ao pacote de dados e gravada quando salvo em arquivo ou stream. Podemos obter esta informação através do método GetOptionalParam.

#### **Parâmetros**

**Param:** Identificador da informação.

**IncludeInDelta:** Especifica se a informação será mantida no Delta.

### **SetProvider(Provider: TComponent)**

Define o Provider que será usado pelo ClientDataSet. Este método é utilizado somente no caso de o Provider estar no mesmo local do ClientDataSet, caso contrário devemos utilizar a propriedade ProviderName.

#### **Parâmetros**

**Provider:** Componente derivado de TCustomProvider que será associado ao ClientDataSet.

### **SetRange(const StartValues, EndValues: array of const)**

Define um range de valores a serem aplicados ao ClientDataSet para filtrar os registros que serão exibidos.

#### **Parâmetros**

**StartValues:** Array de valores iniciais baseados na mesma ordem dos campos definidos no índice atual. Caso o número de campos do índice seja superior ao número de valores definidos, os demais campos serão assumidos como NULL.

**EndValues:** Array de valores finais baseados na mesma ordem dos campos definidos no índice atual. Caso o número de campos do índice seja superior ao número de valores definidos, os demais campos serão assumidos como NULL.

### **SetRangeEnd**

Habilita o ClientDataSet a receber os valores finais dos campos para ser aplicado o range com o método ApplyRange.

Após executarmos este método, definimos os valores dos campos e em seguida, executamos o método ApplyRange para que o range seja aplicado.

### **SetRangeStart**

Habilita o ClientDataSet a receber os valores iniciais dos campos para ser aplicado o range com o método ApplyRange.

Após executarmos este método, definimos os valores iniciais dos campos, depois executamos o método SetRangeEnd para definirmos os valores finais e logo após, executamos o método ApplyRange para que o range seja aplicado.

### **UndoLastChange (FollowChange: Boolean): Boolean**

Desfaz a ultima inclusão, alteração ou exclusão ocorrida no ClientDataSet retornando True caso tenha sido restaurado com sucesso.

#### **Parâmetros**

**FollowChange:** Indica se o cursor deverá ser posicionado no registro restaurado.



### **UpdateStatus: TUpdateStatus**

Retorna o estado atual de atualização do registro enquanto o mesmo não é aplicado.

Esta função nos retorna um dos seguintes valores:

*usUnmodified*: O registro não sofreu alterações.

*usModified*: O registro foi modificado.

*usInserted*: O registro foi inserido.

*usDeleted*: O registro foi excluído

### ClientDataSet - Eventos

A maior parte dos eventos desta classe são do tipo **TRemoteEvent**, eles possuem um funcionamento muito interessante que é importante observarmos.

Este tipo de evento possui a seguinte estrutura:

**XXX(Sender: TObject; var OwnerData: OleVariant);**

#### Parâmetros

**Sender:** Objeto do tipo Provider

**OwnerData:** Variável disponível para alimentarmos com informação adicionais a fim de que sejam passadas entre o cliente (ClientDataSet) e o servidor de aplicação (Provider).

Este tipo de evento funciona da seguinte forma.

Sempre existirá um 'par' (BeforeXXX e AfterXXX) para cada evento e o mesmo 'par' também existirá no DataSetProvider que veremos mais adiante.

Esses 'pares' existem para que no momento de disparo desses eventos o ClientDataSet possa se comunicar com o Provider e vice-versa, pois eles poderão estar em ambientes diferentes quando trabalhamos no modelo multicamadas.

Essa comunicação é feita através do parâmetro **OwnerData** que este tipo de evento nos fornece, pois este parâmetro é passado entre todos eventos (2 do ClientDataSet e 2 do Provider) que formam os 'pares', pois eles sempre serão executados numa sequência lógica:

- 1.º - BeforeXXX (ClientDataSet)
- 2.º - BeforeXXX (DataSetProvider)
- 3.º - AfterXXX(DataSetProvider)
- 4.º - AfterXXX (ClientDataSet)

Percebemos que primeiro é disparado o evento BeforeXXX no ClientDataSet, neste evento podemos alimentar o parâmetro OwnerData da forma que quisermos, pois o mesmo é do tipo OleVariant.

Em seguida é executado o evento BeforeXXX no DataSetProvider que receberá o parâmetro OwnerData alimentado no primeiro evento, é neste momento que dizemos que há uma comunicação entre o ClientDataSet e o Provider, pois enxergamos o parâmetro OwnerData da forma que foi alimentada no primeiro evento do ClientDataSet, ou seja, o ClientDataSet poderia ter enviado alguma informação adicional neste parâmetro. Neste evento também podemos ajustar este parâmetro para ser repassado para o próximo evento.

Depois é executado o evento AfterXXX no DataSetProvider que receberá novamente o parâmetro OwnerData e poderá ser modificado. Neste momento podemos ajustar este parâmetro para enviar alguma informação adicional ao ClientDataSet, já que o próximo evento a ser disparado pertence a ele.

Finalmente é executado o evento AfterXXX no ClientDataSet recebendo novamente o parâmetro OwnerData.

Percebemos o quanto faz sentido a ordem de disparos dos eventos, justamente com o objetivo de termos um total controle de passagem de informações entre aplicação cliente e servidora.

Os eventos do tipo TRemoteEvent são:

**BeforeApplyUpdates:** Ocorre antes de o ClientDataSet aplicar as alterações.

**BeforeExecute:** Ocorre antes de o ClientDataSet receber o resultado da execução do método Execute.

**BeforeGetParams:** Ocorre antes de o ClientDataSet receber os parâmetros do Provider quando executamos o método FetchParams.

**BeforeGetRecords:** Ocorre antes de o ClientDataSet receber o pacote de dados do Provider.

**BeforeRowRequest:** Ocorre antes de o ClientDataSet obter informações do registro atual.

**AfterApplyUpdates:** Ocorre após o ClientDataSet aplicar as alterações.

**AfterExecute:** Ocorre após o ClientDataSet receber o resultado da execução do método Execute.

**AfterGetParams:** Ocorre após o ClientDataSet receber os parâmetros do Provider quando executamos o método FetchParams.

**AfterGetRecords:** Ocorre após o ClientDataSet receber o pacote de dados do Provider.

**AfterRowRequest:** Ocorre após o ClientDataSet obter informações do registro atual.

O evento abaixo não é do tipo TRemoteEvent, portanto explicaremos mais detalhadamente.

**OnReconcileError(DataSet: TCustomClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction)**

Ocorre quando o ClientDataSet necessita reconciliar as atualizações que não puderam ser aplicadas.

### Parâmetros

**DataSet:** ClientDataSet que contém o registro com erro.

**E:** Objeto que contém as informações de erro.

**UpdateKind:** Tipo de atualização que gerou o erro.

**Action:** Ação a ser tomada com o erro. Podemos utilizar os seguintes valores:

*raSkip:* Pula o registro deixando-o pendente no Delta.

*raAbort:* Aborta todo processo de atualização, cancelando tudo.

*raMerge:* Atualiza o registro mesmo que tenha sido modificado.

*raCorrect:* Ajusta o registro do ClientDataSet com as novas definições feitas nos campos (por exemplo quando ajustamos os valores da coluna Modified Value da tela de reconciliação), para que sejam aplicadas na próxima atualização.

*raCancel:* Não atualiza o registro, remove do cache de atualizações e volta ao seu estado original.

*raRefresh:* Desfaz as alterações do registro e busca as informações atuais no banco mantendo-o atualizado.



### DataSetProvider

Este é o componente intermediário que fica entre o ClientDataSet e o DataSet ao qual está ligado.

Sua função principal é de **prover** os dados ao ClientDataSet e também **aplicar** as atualizações no servidor de banco de dados.

No modelo Client/Servidor, ele estará no mesmo local (aplicação) que o ClientDataSet, já no modelo multicamadas, poderia estar em outra aplicação (juntamente com seu DataSet), em um servidor exclusivo, no qual faríamos o acesso remotamente.

### Propriedades

#### Constraints: Boolean

Define se as constraints definidas no servidor de banco de dados serão enviadas ao ClientDataSet através do Metadata.

#### DataSet: TDataSet

Determina o DataSet que o Provider utilizará para obter o pacote de dados e aplicar as atualizações.

#### ResolveToDataSet: Boolean

Determina se as atualizações serão aplicadas pelo Provider ou pelo DataSet ao qual está ligado.

Por padrão esta propriedade é False, ou seja, o Provider será o responsável em aplicar as mudanças no banco de dados, neste caso a propriedade Resolver do Provider é apontada para o componente TSQLResolver que aplicará as mudanças diretamente no banco de dados.

Caso contrário, sendo True, o DataSet do Provider é quem será o responsável em aplicar as mudanças, neste caso a propriedade Resolver do Provider será apontada para o componente TDataSetResolver que aplicará as mudanças diretamente no DataSet, desta forma, todos os eventos do DataSet serão disparados de acordo com as mudanças aplicadas.

Só podemos deixar as atualizações por conta do DataSet caso o mesmo possa executar esta tarefa, que não é o caso dos DataSets unidirecionais, pois são somente leituras, portanto, no caso dos componentes dbExpress que são unidirecionais, somos obrigados a deixar esta responsabilidade por conta do Provider, o que inclusive torna o processo mais rápido.

#### Options: TProviderOptions

Determina algumas opções no Provider onde as mesmas determinarão a forma como os dados serão tratados, o que será incluso no pacote de dados, etc., e conseqüentemente influenciando assim na performance.

Veja abaixo algumas customizações que podemos fazer nesta propriedade:

- Determinar se NestedDataSet e campos BLOBS serão inclusos no pacote de dados ou obtidos separadamente
- Determinar se no pacote de dados serão inclusas propriedades das colunas como: formatação, display names, valores mínimo e máximo.
- Determinar as permissões sobre os registros: somente-leitura, inclusão, edição ou exclusão.
- Determinar se as mudanças feitas nos campos chaves da tabela master deverão refletir nos datasets details, atualizando-os também.
- Determinar se uma atualização poderá afetar mais de um registro.
- Determinar se os registro do Cliente serão atualizados após ter sido aplicado as atualizações.
- Determinar se o Cliente poderá enviar comandos SQL para substituir o DataSet ao qual o Provider está ligado.

Nesta propriedade podemos determinar as seguintes opções:

**poFetchBlobsOnDemand:** Campos BLOBS não serão inclusos no pacote de dados, o cliente deverá requisitar quando necessário. Quando ligamos a propriedade FetchOnDemand do ClientDataSet, esta requisição é feita automaticamente, caso contrário, somos obrigados a fazer isso manualmente através do método FetchBlobs do ClientDataSet.

**poFetchDetailsOnDemand:** NestedDataSet (dataset details) não serão inclusos no pacote de dados, o cliente deverá requisitar quando necessário. Quando ligamos a propriedade FetchOnDemand do ClientDataSet, esta requisição é feita automaticamente, caso contrário, somos obrigados a fazer isso manualmente através do método FetchDetails do ClientDataSet.

**poIncFieldProps:** No pacote serão inclusas algumas propriedades dos campos, tais como: Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask, DisplayValues.

**poCascadeDeletes:** Avisa o servidor de banco de dados para excluir os registros details automaticamente quando os registros master forem excluídos. Isso só poderá ser feito caso estejamos trabalhando com master/detail e o servidor de banco de dados suportar atualização em cascata definida na integridade referencial.

**poReadOnly:** O ClientDataSet será somente leitura, as atualizações não serão aplicadas pelo Provider.

**poAllowMultiRecordUpdates:** Uma atualização poderá afetar múltiplos registros. Caso isso ocorra e esta propriedade não esteja ligada, a atualização é abortada.

**poDisableInserts:** Inserções não poderão ser aplicadas pelo Provider.

**poDisableEdit:** Alterações não poderão ser aplicadas pelo Provider.

**poDisableDeletes:** Exclusões não poderão ser aplicadas pelo Provider.

**poNoReset:** Ignora a reinicialização das flags quando o método As\_GetRecords é chamado.

**poAutoRefresh:** Executa um refresh no ClientDataSet após ter aplicado as atualizações.

**poPropagateChanges:** Determina que as mudanças feitas nos registros através dos eventos BeforeUpdateRecord ou AfterUpdateRecord do Provider serão enviadas ao ClientDataSet.

**poAllowCommandText:** Determina se o cliente poderá substituir o comando SQL, nome da tabela ou Stored Procedure definido no DataSet ao qual o Provider está ligado.

**poRetainServerOrder:** Determina que o ClientDataSet não poderá modificar a ordenação dos registros, isto só poderá ser feito pelo Provider.

### Resolver: TCustomResolver

Retorna o componente utilizado para aplicar as atualizações e resolver os erros.

### **UpdateMode: TUpdateMode**

Determina como os registros serão localizados na atualização quando forem modificados ou excluídos. Isto determinará quais campos pertencerão à cláusula WHERE do comando SQL que será montado pelo Provider para atualizar os registros.

Podemos determinar se os registros serão localizados baseando-se em todas as colunas, nas modificadas ou somente nas colunas chaves.

Esta propriedade trabalha em conjunto com a propriedade ProviderFlags dos campos do DataSet ao qual o Provider está ligado, pois em ProviderFlags definimos por exemplo, quais são os campos chaves, os campos que serão atualizados, etc.

Podemos definir um dos seguintes valores para a propriedade UpdateMode:

**upWhereAll:** Determina que a localização será feita com base nos valores originais de todos os campos existentes no DataSet. Esta opção garante que o registro seja atualizado somente se for encontrado no mesmo estado original que foi editado, evitando o problema de o registro já ter sido modificado por outro usuário, porém pode perder performance, já que todos os campos pertencerão à cláusula WHERE.

**upWhereChanged:** Determina que a localização será feita com base nos valores originais apenas dos campos modificados.

**upWhereKeyOnly:** Determina que a localização será feita com base apenas nos campos chaves. Esta é a melhor opção quando não estamos preocupados se o registro foi modificado por outro usuário, pois somente os campos chaves pertencerão à cláusula WHERE. Desta forma sempre prevalecerá às modificações feitas pelo último usuário.

Para determinarmos os campos chaves, basta ligarmos a opção pfInKey da propriedade ProviderFlags do respectivo campo no DataSet ao qual o Provider está ligado.

### **Data: OleVariant**

Retorna o pacote de dados contendo todos os registros do Provider. Quando acessamos esta propriedade, internamente o Provider faz chamada ao método GetRecords passando -1 como parâmetro.

### **Exported: Boolean**

Determina se o Provider estará disponível para as aplicações clientes quando o mesmo reside em um servidor remoto.

Ligando esta propriedade podemos ter o acesso utilizando a interface IAppServer, porém devemos registrar o provedor remoto. Utilizando o RemoteDataModule, ele é registrado automaticamente, caso contrário, utilizamos o método RegisterProvider para realizar esta tarefa.

### DataSetProvider - Métodos

**ApplyUpdates(const Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer); OleVariant**

**ApplyUpdates(const Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant); OleVariant**

Aplica as atualizações contidas no Delta ao Banco de Dados.

Este método é executado automaticamente quando chamamos o ApplyUpdates no ClientDataSet, onde ele passa seu Delta como parâmetro e retorna o número de erros que o parâmetro ErrorCount informa.

Esta função retornará os registros que não puderam ser aplicados.

#### **Parâmetros**

**Delta:** Pacote de dados a serem aplicados ao banco de dados.

**MaxErrors:** Número máximo de erros permitidos. Possui o mesmo objetivo do parâmetro passado no método ApplyUpdates do ClientDataSet.

**ErrorCount:** Número de erros ocorridos na atualização.

**OwnerData:** Informação customizada que poderá ser utilizada e alterada nos eventos BeforeApplyUpdates e AfterApplyUpdates.

#### **DataRequest (Input: OleVariant): OleVariant**

Gera o evento OnDataRequest. Possui o mesmo efeito que chamarmos o método DataRequest do ClientDataSet, pois internamente o ClientDataSet faz uso deste método.

#### **Parâmetros**

**Input:** Informação customizada a ser utilizada para fazer comunicação entre aplicação Cliente e Servidora.

#### **Execute(const CommandText: WideString, var Params, OwnerData: OleVariant)**

Executa a Query, Stored Procedure ou uma instrução SQL.

#### **Parâmetros**

**CommandText:** Define o comando SQL, nome da tabela ou Stored Procedure que substituirá o que está associado no DataSet ao qual o Provider está ligado. Este parâmetro só será válido caso esteja inclusa a opção poAllowCommandText na propriedade Options do Provider.

**Params:** Contém os parâmetros que serão utilizados pelo SQL associado ao CommandText, pela Query ou Stored Procedure ao qual o Provider está ligado.

**OwnerData:** Informação customizada que poderá ser utilizada e alterada nos eventos BeforeExecute e AfterExecute.

#### **GetParams(var OwnerData: OleVariant); OleVariant**

Obtém os valores atuais dos parâmetros associados ao Provider.

#### **Parâmetros**

**OwnerData:** Informação customizada que poderá ser utilizada e modificada nos eventos BeforeGetParams e AfterGetParams.

**GetRecords(Count: Integer; out RecsOut: Integer; Options: Integer); OleVariant**  
**GetRecords(Count: Integer; out RecsOut: Integer; Options: Integer; const CommandText: WideString; var Params, OwnerData: OleVariant); OleVariant**  
Retorna o pacote de dados associado ao Provider.

### Parâmetros

**Count:** Número de registros a serem extraídos. 0 indica para extrair apenas o metadata, -1 extrai todos registros e maior que -1 indica a quantidade de registros a ser extraído.

**RecsOut:** Retorna quantidade de registros do pacote.

**Options:** Determina as informações que serão incluídas no pacote. O valor deste parâmetro é uma combinação das constantes de GetRecordOption:

*MetaDataOption (grMetaData):* Inclui o metadado de estrutura dos registros.

*ResetOption (grReset):* O pacote iniciará no primeiro registro, desconsiderando o conteúdo dos próximos registros previamente enviados no mesmo pacote.

*XMLOption (grXML):* Os dados serão transmitidos no formato XML.

*XMLUTF8Option (grXMLUTF8):* Os dados serão transmitidos no formato XML mas na extensão UTF-8.

A combinação das opções é feita somando as constantes, exemplo: MetaDataOption + XMLOption.

**CommandText:** Define o comando SQL, nome da tabela ou Stored Procedure que substituirá o que está associado no DataSet ao qual o Provider está ligado. Este parâmetro só será válido caso esteja incluída a opção poAllowCommandText na propriedade Options do Provider.

**Params:** Contém os parâmetros que serão utilizados pelo SQL associado ao CommandText, pela Query ou Stored Procedure ao qual o Provider está ligado.

**OwnerData:** Informação customizada que poderá ser utilizada e alterada nos eventos BeforeGetRecords e AfterGetRecords.

**RowRequest(const Row: OleVariant; RequestType: Integer; var OwnerData: OleVariant): OleVariant**

Retorna informações de um registro específico.

Este método é executado automaticamente quando os métodos FetchDetails, FetchBlobs ou RefreshRecords são chamados pelo ClientDataSet.

### Parâmetros

**Row:** Descreve as informações do registro que queremos obter o resultado.

**RequestType:** Indica o tipo de informação que queremos obter. O valor deste parâmetro é uma combinação do tipo TFetchOptions:

*foRecord:* Valores dos campos.

*foBlobs:* Valores dos campos BLOBS.

*foDetails:* Valores dos campos DataSetFields, ou seja, dos NestedDataSet.

Para utilizarmos uma destas opções, devemos convertê-las para o tipo integer, por exemplo: Ord(foBlobs) + Ord(foDetails).

**OwnerData:** Informação customizada que poderá ser utilizada e alterada nos eventos BeforeRowRequest e AfterRowRequest.



### DataSetProvider - Eventos

#### **OnGetDataSetProperties (Sender: TObject; DataSet: TDataSet; out Properties: OleVariant)**

Disparado quando o Provider precisa obter informações adicionais do DataSet.

Neste evento podemos incluir informações customizadas que serão enviadas ao ClientDataSet junto com o pacote de dados. Desta forma, poderemos utilizar estas informações nos eventos do ClientDataSet.

#### **Parâmetros**

**Sender:** Referência ao Provider que está criando o pacote de dados.

**DataSet:** Referência ao DataSet que representa o pacote de dados.

**Properties:** Informações adicionais a serem incluídas no pacote.

#### **OnGetTableName (Sender: TObject; DataSet: TDataSet; var TableName: string)**

Ocorre quando o Resolver precisa obter o nome da tabela para aplicar as atualizações.

Neste evento podemos modificar o nome da tabela a ser atualizada. Muito utilizado nos casos em que utilizamos JOINS, UNIONS, envolvendo mais de uma tabela.

#### **Parâmetros**

**Sender:** Provider que necessita do nome da tabela para ser atualizada.

**DataSet:** DataSet que será aplicado as alterações.

**TableName:** Nome da tabela a ser atualizada. A alteração deste parâmetro influenciará na instrução SQL a ser gerada para atualização dos dados.

#### **AfterUpdateRecord(Sender: TObject; SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind)**

Ocorre após o registro ter sido atualizado, ou seja, após ter sido aplicado ao banco de dados.

Podemos utilizar este evento por exemplo, para logar os registros atualizados com sucesso.

#### **Parâmetros**

**Sender:** Provider que está aplicando a atualização.

**SourceDs:** DataSet de origem dos dados, normalmente o DataSet ao qual o Provider está ligado.

**DeltaDs:** Contém todos os dados do registro que está sendo atualizado.

**UpdateKind:** Tipo de atualização que está sendo aplicada: ukInsert(inserindo), ukModify(modificando) ou ukDelete(excluindo).

#### **BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean)**

Ocorre antes de o registro ser atualizado, ou seja, antes de ser aplicado ao banco de dados.

Neste evento podemos fazer modificações no registro antes de ser enviado ao banco, para isto, basta modificarmos o Delta definindo novos valores aos campos através da propriedade NewValue. Podemos também acessar os valores originais dos campos através da propriedade OldValue e os valores atuais através da propriedade CurValue.

Para que as alterações feitas no Delta sejam refletidas no ClientDataSet, é necessário que o Provider esteja com a opção poPropagateChanges incluída na sua propriedade Options.

Este evento também é utilizado para validar informações abortando quando necessário e até mesmo anular determinadas atualizações.

### **Parâmetros**

**Sender:** Provider que está aplicando a atualização.

**SourceDs:** DataSet de origem dos dados, normalmente o DataSet ao qual o Provider está ligado.

**DeltaDs:** Contém todos os dados do registro que está sendo atualizado.

**UpdateKind:** Tipo de atualização que está sendo aplicada: `ukInsert`(inserindo), `ukModify`(modificando) ou `ukDelete`(excluindo).

**Applied:** Determina se o Provider deverá ou não aplicar a atualização no banco. O valor padrão deste parâmetro é `False`, isso significa que a atualização não foi aplicada, portanto o Provider aplicará. Definindo como `True`, estamos indicando que a atualização já foi aplicada, logo o Provider não aplicará.

### **OnGetData (Sender: TObject; DataSet: TCustomClientDataSet)**

Ocorre após o Provider ter obtido os dados porém antes de terem sido enviados ao ClientDataSet.

Neste evento podemos fazer modificações nos dados do DataSet do parâmetro antes de serem enviados ao ClientDataSet.

### **Parâmetros**

**Sender:** Provider que enviará os dados ao ClientDataSet.

**DataSet:** DataSet com os dados que serão enviados ao ClientDataSet. Neste DataSet podemos fazer quaisquer modificações que as mesmas serão enviadas ao ClientDataSet.

### **OnUpdateData (Sender: TObject; DataSet: TCustomClientDataSet)**

Ocorre quando o Provider inicia as atualizações.

Neste evento podemos validar campos, gerar exceções e também modificar os valores quando necessário.

### **Parâmetros**

**Sender:** Provider que está aplicando as atualizações.

**DataSet:** DataSet recebido do Cliente (ClientDataSet) com o pacote de dados e atualizações a serem aplicadas. Podemos fazer modificações neste DataSet para que sejam refletidas na atualização.

### **OnUpdateError(Sender: TObject; DataSet: TCustomClientDataSet; E: EUpdateError; UpdateKind: TUpdateKind; var Response: TResolverResponse)**

Ocorre quando o Provider não consegue atualizar o registro devido algum erro ocorrido.

### **Parâmetros**

**Sender:** Provider que está tentando aplicar a atualização

**DataSet:** ClientDataSet temporário utilizado apenas para acessar os dados no processo de atualização. Para checarmos os valores dos campos, utilizamos as propriedades `OldValue`, `NewValue` ou `CurValue` da classe `TField`.

**E:** Objeto que contém informações do erro.

**UpdateKind:** Tipo de atualização que causou o erro.

**Response:** Tipo de ação a ser tomada com o respectivo erro:

*rrSkip:* Pula a atualização do registro, mantendo-o no cachê de alterações do ClientDataSet.

*rrAbort:* Aborta todo processo de atualização, cancelando tudo

*rrMerge:* Faz uma fusão dos dados do Cliente com os dados do servidor, aplicando somente os campos modificados pelo cliente.

*rrApply:* Aplica as alterações de acordo com os valores ajustados no evento.

*rrIgnore:* Ignora o registro, não aplica ao banco e não devolve para o ClientDataSet tentar resolver depois.

### **OnDataRequest(Sender: TObject; Input : OleVariant) : OleVariant**

Ocorre quando o Provider é chamado pelo método DataRequest do ClientDataSet.

Neste evento podemos ler o parâmetro Input que foi passado pelo Cliente e fazer algo resultando em um valor que será enviado ao ClientDataSet.

### **Parâmetros**

**Sender:** Provider que está recebendo a requisição.

**Input:** Informação adicional que foi passado pelo Cliente.

Os eventos abaixo são do tipo TRemoteEvent. Na classe TClientDataSet, abordamos como esses eventos funcionam e seus respectivos parâmetros, portanto, veremos abaixo apenas o momento em que cada um é disparado.

**BeforeApplyUpdates:** Ocorre antes de o Provider aplicar as atualizações.

**BeforeExecute:** Ocorre antes de o Provider executar o comando SQL, query ou Stored Procedure.

**BeforeGetParams:** Ocorre antes de o Provider criar os parâmetros que serão enviados ao ClientDataSet.

**BeforeGetRecords:** Ocorre antes de o Provider ter criado o pacote de dados que serão enviados ao ClientDataSet.

**BeforeRowRequest:** Ocorre antes de o Provider criar um pacote com as informações do registro atual.

**AfterApplyUpdates:** Ocorre após o Provider ter aplicado as atualizações.

**AfterExecute:** Ocorre após o Provider ter executado o comando SQL, query ou Stored Procedure.

**AfterGetParams:** Ocorre após o Provider ter criado os parâmetros e enviados ao ClientDataSet.

**AfterGetRecords:** Ocorre após o Provider ter criado o pacote de dados e enviado ao ClientDataSet.

**AfterRowRequest:** Ocorre após o Provider ter obtido informações do registro atual.