DevMedia - Versão para impressão

DataSnap XE, Generics, RTTI e DAO

Transfira e persista objetos em aplicações multicamadas

Do que trata o artigo

O artigo trata do desenvolvimento de aplicação DataSnap utilizando a nova arquitetura DataSnap XE, lançando mão de recursos avançados da orientação a objetos, como a reflexão computacional, padrões de projeto como DAO além de Generics, um dos recursos mais interessantes disponíveis na linguagem Delphi.

Para que serve

O desenvolvimento em camadas serve para criarmos aplicações escaláveis, de fácil manutenção e distribuição, uma vez que as regras de negócios ficam encapsuladas no servidor. Desenvolver uma aplicação em camadas com recursos de OO como Generics e RTTI aumenta a produtividade da equipe permitindo um maior reaproveitamento de códigos e como consequência uma redução de prazo e custo.

Em que situação o tema é útil

Utilizar conceitos avançados de desenvolvimento orientado a objetos é útil quando se quer aumentar o nível de abstração do projeto e como consequência tornar o código mais coeso e menos acoplado. Assim a aplicação fica muita mais limpa, fácil de manter e muito flexível quando há a necessidade de implementar novos recursos evitando o retrabalho e efeitos colaterais.

Resumo do DevMan

Nos últimos tempos assuntos como padrões de projetos, metodologias ágeis, boas práticas entre outros estão sendo discutidos como nunca. Com o avanço da tecnologia, prazos cada vez menores e clientes cada vez mais exigentes, a busca por métodos e procedimentos que aumentem a produtividade se torna uma obrigação. No desenvolvimento orientado a objetos existem alguns paradigmas que proporcionam maior produtividade como o uso da herança e polimorfismo, mas somente estes paradigmas de forma isolada não trazem o benefício esperado no atual cenário tecnológico em que estamos. As equipes estão cada vez mais heterogêneas, os projetos cada vez maiores, as opções de linguagens e frameworks são bem vastas e para manter tudo isso muito bem alinhado é preciso uma gestão eficiente. Mas isso é mais assunto para gerência de projeto. Neste artigo vamos observar como estes avanços influenciam a vida do desenvolvedor e como podemos aplicá-los em nossos projetos

desenvolvidos em Delphi.

Rodrigo Carreiro Mourão
Graduado em gestão de TI com especialização em gerência de projetos (PMI), Governança de TI (COBIT) e Gestão de Serviços de TI (ITIL) pelo instituto Infnet. Grande entusiasta de metodologias ágeis, padrões de projetos e gestão por processos. Ultimamente tem dedicado-se a desbravar a área de Software Quality Assurance para ajudar a fomentar esta boa prática em empresas de todos os tamanhos e segmentos. Saiba mais em: http://about.me/rodrigomourao

No que tange a área de desenvolvimento, a criação dos padrões de projetos trouxe um enorme benefício uma vez que apresentam soluções para problemas recorrentes no desenvolvimento de projetos orientados a objetos. Com isso, ao iniciar um projeto já sabemos o caminho a seguir para evitar determinado problema. A orientação a objetos evoluiu, ou melhor, as linguagens orientadas a objetos começaram a incorporar novos recursos, recursos estes que vieram trazer enorme benefício ao desenvolvimento de software orientado a objetos. As linguagens e ferramentas também evoluíram incorporando em seus IDEs novas Features, Wizards e utilitários, sempre com o foco na produtividade.

Quando falamos de Delphi também não ficamos atrás. A linguagem Delphi evoluiu nos últimos anos incorporando alguns destes recursos como Generics, Anonymous Methods, nova RTTI, DBX4 e muito mais. Este ano tivemos o lançamento do Delphi XE que conta com uma série de ferramentas incorporadas ao IDE como integração com Subversion, CodeSite, FinalBuilder, entre outros.

Um capítulo à parte quando se trata de Delphi está por conta da arquitetura DataSnap. Não podemos negar que o desenvolvimento em camadas sempre foi um diferencial no Delphi. Aplicações em camadas sempre foi uma boa opção para projetos grandes que demandam escalabilidade e facilidade de manutenção.

Com este tipo de aplicação podemos concentrar nossas regras de negócio no servidor criando assim aplicações clientes mais enxutas.

A distribuição da aplicação também é facilitada, uma vez que não há necessidade de instalação de drivers para acesso a dados ou qualquer outra tecnologia de conexão a banco de dados.

Nas últimas versões do Delphi a arquitetura DataSnap evoluiu e muito, permitindo que os desenvolvedores Delphi possam lançar mão do que há de mais recente no mundo do desenvolvimento, como Cloud Computing, RIA, aplicações RESTFull e muito mais.

Agora imagine o quão interessante seria unir toda a produtividade do Delphi em sua arquitetura DataSnap com padrões e recursos avançados da orientação a objetos. Atente para o fato de todos os padrões de desenvolvimento, framework ágil, metodologia ou filosofia de desenvolvimento não está vinculada a linguagem, ou seja, não existe restrição quanto à linguagem e/ou ferramenta utilizada para você desenvolver uma aplicação baseado em RIA por exemplo.

Muito menos alguma exigência para a aplicação do padrão DAO (veremos mais adiante) a não ser que a linguagem que você for trabalhar seja orientada a objetos. O mesmo vale para os padrões de projeto.

Nota

Anonymous Methods é um recurso que foi adicionado no Delphi 2009 e é conhecido

também como "referências a métodos", pois sua declaração é feita com a sintaxe "reference to function/procedure". Em resumo criar um método anônimo é criar um tipo que faz referência a um procedure ou function com uma assinatura específica, por exemplo:

```
TCalculo = reference to function(const N1, N2: double): double;

procedure ExecutaCalculo(Calculo: TCalculo);

ExecutaCalculo(function(const N1, N2: double): double
                  begin
                       result := N1 + N2;
                  end);
```

O Delphi XE trouxe como principal atrativo a integração do IDE com diversas ferramentas utilitárias há muito utilizadas pela comunidade, são elas:
Subversion - também conhecido por SVN, é um sistema de controle de versão desenhado especificamente para ser um substituto moderno do CVS, que se considera ter algumas limitações.
FinalBuilder - ferramenta para gerenciamento de compilações, que traz uma maneira fácil e prática de criar e rodar scripts de compilações. Com uma interface amigável para o usuário esse software permite realizar a compilação de um ou vários projetos.
CodeSite - é como um super gerador de arquivos de log, pois permite enviar dados dos erros para o log, inclusive tipos personalizados. O log pode então ser visualizado localmente, no viewer ou gravado num arquivo. As mensagens podem ser de erros, avisos, informações, sendo mostradas com ícones diferentes, podendo ser filtradas, para esconder mensagens desnecessárias.

Desenvolvimento ágil de software ou método ágil é um conjunto de metodologias. Providencia uma estrutura conceitual para reger projetos de Engenharia de Software. Entre as metodologias mais conhecidas podemos citar o XP, Scrum, FDD, DSDM etc.
AQTime - ferramenta para analisar a performance do código-fonte, facilitando o descobrimento dos gargalos, permitindo otimizações mais precisas mostrando onde realmente está o problema. AQTime é ao ferramenta de Profile líder há anos entre os desenvolvedores Delphi.

Como o Delphi é uma linguagem 100% orientada a objetos fica fácil entender que todos estes padrões podem ser aplicados em soluções desenvolvidas em Delphi e esta é a proposta deste artigo.

Vamos ver como aplicar alguns dos padrões de desenvolvimento e ainda, lançar mão de recursos avançados da orientação a objetos como Generics, trabalhar com reflexão computacional através da RTTI do Delphi, tudo isso em uma aplicação em camadas utilizando a nova arquitetura DataSnap.

Generics

Generics é um dos recursos mais interessantes em Orientação a Objetos em minha opinião, pois flexibiliza muito a escrita do código.

Como o próprio nome sugere este recurso nos permite trabalhar com uma estrutura

genérica, o que faz toda a diferença no caso do Delphi que é uma linguagem fortemente tipada.

Quando falamos que uma linguagem é fortemente tipada significa que as estruturas definidas nesta linguagem obrigatoriamente têm que ter um tipo de dado no momento de sua declaração e este tipo de dado não poderá mudar, de uma forma geral seria dizer que o que nasce inteiro, vive inteiro e morre inteiro.

Ocorre que um dos pilares da Orientação a Objetos é o polimorfismo e para que possamos implementar o polimorfismo temos que lançar mão da herança, onde geralmente temos uma classe básica da qual descendem os demais objetos.

Com isso definimos uma variável deste tipo básico e instanciamos um objeto de um tipo descendente, até aí tudo perfeito. Mas quando precisamos acessar um método ou propriedade exclusiva do objeto descendente somos obrigados a fazer um Type Casting, e esta tarefa onera e muito o processamento além de permitir que erros ocorram.

Através dos Generics pode-se definir uma estrutura genérica customizada, ou melhor, tipada de acordo com a necessidade.

O objeto, parâmetro ou variável não tem seu tipo definido no momento da declaração e sim no momento do uso. Um caso clássico para quem trabalha com orientação a objetos são as listas de objetos. É muito comum utilizar a classe TList para trabalhar com listas de objetos, porém a TList pode conter qualquer tipo de objeto e quando se acessa um item da lista recebe-se um TObject, o que nos obriga a realizar o Type Casting. Sem falar que não há garantias de que em uma lista de clientes, por exemplo, não será inserido um botão, pois não há validação em tempo de projeto.

Nota

RIA - Rich Internet Application - são Aplicações Web que têm características e funcionalidades de softwares tradicionais do tipo Desktop. RIA típicos transferem todo o processamento da interface para o navegador da internet, porém, mantêm a maior parte dos dados (como por exemplo, o estado do programa, dados do banco) no servidor.

O conceito de computação em nuvem (em inglês, Cloud Computing) refere-se à utilização da memória e das capacidades de armazenamento e cálculo de computadores e servidores compartilhados e interligados por meio da Internet, seguindo o princípio da computação em grade (grid). O acesso a programas, serviços e arquivos é remoto, através da Internet - daí a alusão à nuvem. O uso desse modelo (ambiente) é mais viável do que o uso de unidades físicas.

Type Casting ou Type Convertion é um recurso que muitas linguagens de programação possuem e que fica mais evidente seu uso nas linguagens ditas fortemente tipadas. A função do Type Casting, como o nome sugere, é realizar a conversão de tipo de um determinado objeto. Em POO é muito comum este tipo de procedimento quando trabalhamos com objetos em estrutura hierárquica, onde um objeto de um determinado tipo é convertido para outro para que se possa acessar uma propriedade ou método específico.

A Orientação a Objetos é baseada em 4 pilares a saber: Herança, Polimorfismo, Encapsulamento e Abstração.
Herança ou generalização é o mecanismo pelo qual uma classe pode estender outra classe, aproveitando seus métodos e propriedades.
Encapsulamento consiste na separação de aspectos internos e externos de um objeto e é

utilizado amplamente para impedir o acesso direto ao estado de um objeto.
Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais.
Polimorfismo consiste na capacidade que uma linguagem tem de fazer um mesmo método se comportar de maneira diferente de acordo com o objeto de onde foi chamado.

Uma solução seria criar uma lista exclusiva de clientes herdando de TList, mas isto pode se tornar um problema se tivermos que fazer isso para cada tipo de objeto em uma aplicação, tornando-se inviável.

Com Generics isso é resolvido de uma forma extremamente simples, utilizando a classe TList tipada, ou seja, no momento do uso você define qual tipo de dados a lista vai armazenar. Com isso o parâmetro solicitado no método Add será do tipo definido na criação da lista, assim como todos os demais métodos e propriedades que se referem ao tipo.

RTTI

Outro recurso muito importante na orientação a objetos é a reflexão computacional. Este recurso trata da capacidade que a linguagem tem de extrair informações de um objeto em tempo de execução, daí a sigla RTTI que significa Run Time Type Information ou Informação de tipo em tempo de execução. Um exemplo clássico de RTTI no Delphi que é muito citado para ilustrar o recurso é o Object Inspector. A função dele dentro do IDE é permitir o desenvolvedor alterar as propriedades de um dado componente, neste caso, o componente selecionado no momento. Não podemos esquecer que este componente é na verdade um objeto de um tipo específico, seja ele um TButton, TEdit, TTable etc.

E para cada componente selecionado o Object Inspector mostra suas respectivas propriedades e eventos.

Note que as propriedades são montadas dinamicamente e isso só é possível devido a RTTI. Quando o objeto é selecionado, via RTTI o IDE extrai todas as propriedades publicadas do componente em questão para então permitir-nos alterá-las conforme a necessidade.

E também, através de RTTI é que os valores das propriedades são alterados nos objetos que estão instanciados em memória. Com isso chega-se a conclusão que reflexão computacional nos permite extrair informações de um objeto e também alterar os valores de suas propriedades.

Até o lançamento do Delphi 2010 a RTTI era muito engessada e limitada, mas com a chegada desta nova versão a coisa mudou de figura.

Novas classes foram introduzidas da VCL e RTL, o compilador passou a dar mais informações sobre os objetos, a RTTI ficou muito mais flexível e intuitiva e o melhor, muitas das limitações foram jogadas por terra. Isso quebrou uma barreira muito grande, uma vez que o Delphi é compilado em Delphi, pois com uma flexibilidade maior, novos recursos podem ser adicionados ao IDE. Sem falar que novos frameworks e novos componentes podem e já começaram a surgir.

Nota

Entende-se por algumas limitações da antiga estrutura RTTI do Delphi o fato de se dever declarar as propriedades como published, a necessidade das diretivas {$M+}, o fato de não ter como identificar os métodos e propriedades de um objeto e os métodos e propriedades que este mesmo objeto herdou de seu ancestral, entre outras.

Nota

Utilizar a classe TList tipada é muito simples, basta no momento da declaração da variável informar qual o tipo de objeto deseja guardar na lista. Observe:

```
var
  Lista: TList;
begin
  Lista := TList.Create;
  (…)
end;
```

O Delphi 2010 introduziu uma nova RTTI com novas classes, tipos, métodos, units e funções. Como principal estrutura temos a presença do record TRttiContext que extrai as informações do tipo que está sendo mapeado, além de um tipo para cada estrutura utilizada nas classes que desenvolvemos, como por exemplo, TRttiMethod, TRttiProperty, TRttiType, TRttiParameters, TRttiAttributes etc.

Lançar mão de estruturas baseadas em RTTI significa adicionar aos projetos orientados a objetos uma flexibilidade muito grande. Utilizar RTTI é elevar o nível de reutilização de códigos, é permitir que uma mesma estrutura possa atender as mais diversas situações, sem falar que a manutenção se torna mais simples e rápida. Prova disso são os frameworks de mapeamento objeto relacional que existem no mercado, não só para Delphi, mas para outras linguagens que se utilizam de reflexão computacional para realizar o que chamamos de mapeamento objeto relacional, extraindo informações dos objetos em memória para gerar as instruções SQL e então, persisti-los em banco de dados.

DAO

DAO significa Data Access Object ou Objeto de Acesso a dados. É um padrão de desenvolvimento de softwares que trata da persistência de objetos e prega a separação das camadas de negócio e acesso a dados. De uma forma mais tradicional, para cada classe de negócio há uma classe de persistência correspondente que tem a função de encapsular toda a comunicação com o banco de dados para a realização do procedimento de CRUD. O padrão DAO está geralmente associado a aplicações J2EE, mas seu conceito pode ser aplicado a qualquer linguagem dita orientada a objetos.

A principal vantagem deste tipo de padrão é justamente a possibilidade de manter em estruturas separadas todos os comandos relativos à conexão com banco de dados, consultas SQL etc., permitindo assim que o código da aplicação fique mais coeso. Sem falar que como a parte de acesso a dados está encapsulada em uma camada separada, há a flexibilidade de alterar a tecnologia de acesso a dados ou mesmo o banco de dados a se persistir os objetos, sem que se tenha que alterar as classes de negócio e o funcionamento da aplicação.

O problema é que o número de classes aumenta consideravelmente no caso de se optar pelo padrão DAO para persistência dos dados, mas lançando mão de recursos como RTTI e Generics, pode-se criar um mecanismo dinâmico para geração de comando SQL sem que para isso tenha-se que criar um para cada classe de negócio. Esta é a ideia deste artigo, mostrar que existem alternativas e principalmente, possibilidades.

Aplicação Servidora

Os conceitos vistos até aqui fazem mais sentido quando são colocados em prática em uma

aplicação DataSnap, e isso pode ser feito criando uma nova aplicação no Delphi XE. Para isso deve ser selecionado um modelo de DataSnap Server.

A Figura 1 exibe o Wizard que permite selecionar os protocolos de transporte, suporte a autenticação e inclusão de métodos de exemplo.

Nota

Criando o servidor de aplicação

Mapeamento objeto relacional ou ORM, do inglês: Object Relational Mapping é uma técnica de desenvolvimento utilizada para reduzir a impedância da programação orientada a objetos utilizando bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes. Com esta técnica, o programador não precisa se preocupar com os comandos em linguagem SQL, devendo apenas usar uma interface de programação simples que faz todo o trabalho de persistência.

CRUD é o acrônimo de Create, Retrieve, Update e Delete que são as quatro operações básicas utilizadas em bancos de dados relacionais ou em interface para usuários para inserção, consulta, atualização e exclusão de dados.

[abrir imagem em janela]

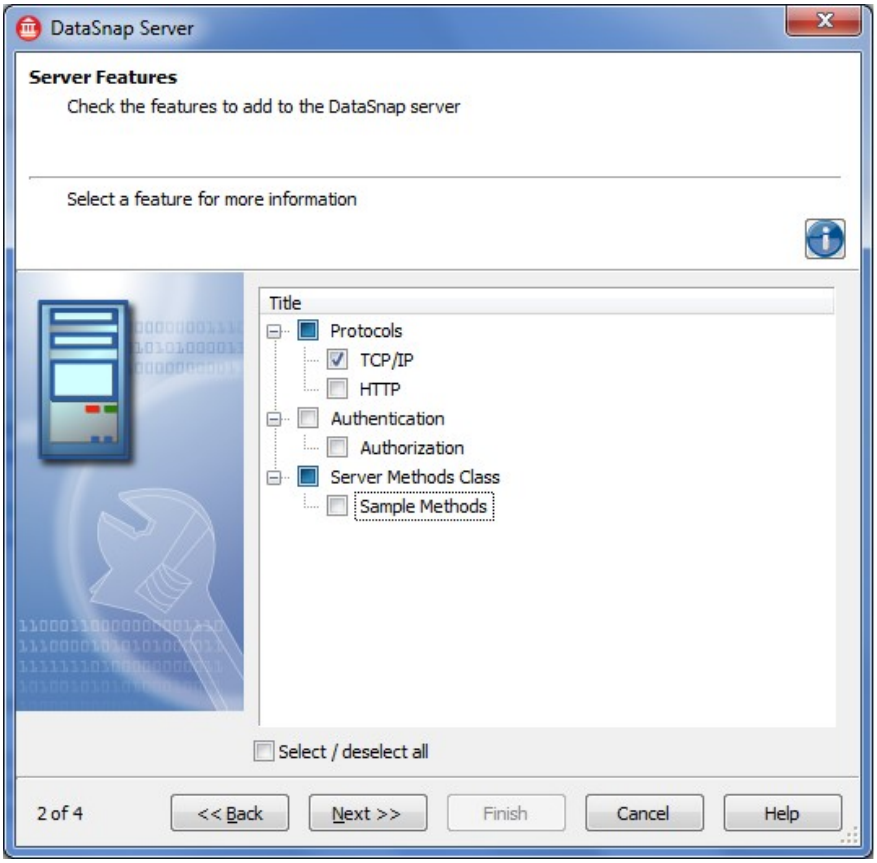

Figura 1. Criando o servidor de aplicação

A interface do servidor de aplicação não possui nenhum recurso extraordinário até porque o foco está nos serviços, ou melhor, nos métodos que o servidor irá disponibilizar para seus clientes. A Figura 2 mostra uma sugestão de interface apenas com um pequeno texto e dois botões, um para iniciar o servidor e outro para interrompê-lo. Para os botões de Start e Stop

sugere-se os códigos da Listagem 1.

Desde a versão 2010 do Delphi foi adicionado ao framework DataSnap um novo e forte aliado, chamado JSON. Os dados podem ser trafegados entre o cliente e o servidor em notação JSON, facilitando assim o desenvolvimento. Desta forma, um objeto de negócio na aplicação pode ser convertido para notação JSON, enviado ao servidor de aplicação para que possa ser novamente convertido para notação tradicional e persistido na base de dados.

A VCL traz para algumas de suas units, classes e métodos específicos para se trabalhar com objetos e dados em notação JSON. Os dois métodos principais são o Marshal, que tem por objetivo converter um objeto tradicional em um objeto JSON e o método unMarshal que tem função oposta, ou seja, converter um objeto JSON em um objeto tradicional.

O JSON (Javascript Object Notation) é uma espécie de padronização para transporte de informações entre ambientes heterogêneos e que tem como linguagem o Javascript.

Padrão, transporte, ambiente heterogêneo, tudo isso remete a XML, então eis a questão: Por que JSON e não XML? O principal motivo seria pelo fato da notação JSON ser extremamente mais simples do que a do XML. Não existem tags diversas nem regras complexas para escrita e muito menos mecanismos complexos de parser.

Classes de negócio

Em uma aplicação real temos muitas classes de negócio com associações com outras classes e muitas outras regras de negócio. Torna-se inviável criar um método para converter cada uma das classes com que se pode trabalhar, sem falar que novas classes podem surgir. É neste momento que o uso do recurso de Generics se encaixa perfeitamente. Pode-se ter um método que converta um tipo genérico para um objeto JSON, tipo este definido em tempo de execução e da mesma forma um segundo método que receba um objeto JSON e o converta em um objeto de tipo genérico, que também será definido em tempo de execução.

A Listagem 2 mostra um exemplo de como isso pode ser implementado. Primeiro é definida uma classe chamada de TBase contendo dois métodos de classe. Cada método define um tipo genérico, representado pela letra T. Este tipo genérico possui a constraint class.

Este também especifica o tipo dos parâmetros bem como de retorno no caso do método JSONtoObject, assim, se em tempo de execução definir o tipo como um TButton logo o parâmetro será do tipo TButton e o retorno da função JSONtoObject será do tipo correspondente.

Listagem 1. Código dos botões

```
if not SC.DSServer.Started then
begin
  SC.DSServer.Start;
  BtnStart.Enabled := False;
  btnStop.Enabled  := True;
end;

if SC.DSServer.Started then
begin
  SC.DSServer.Stop;
  BtnStart.Enabled := True;
  btnStop.Enabled  := False;
end;
```

Listagem 2. Classe base com métodos de conversão

```
uses DBXJSON, DBXJSONReflect;


TBase = class
```

```
public
class function ObjectToJSON<T: class>(aObject: T): TJSONValue;
class function JSONToObject<T: class>(aJSON: TJSONValue): T;
end;

class function TBase.JSONToObject<T>(aJSON: TJSONValue): T;
var
UnMarshal: TJSONUnMarshal;
begin
if aJSON is TJSONNull then
  Exit(nil);
UnMarshal := TJSONUnMarshal.Create;
Result := T(UnMarshal.Unmarshal(aJSON));
UnMarshal.Free;
end;

class function TBase.ObjectToJSON<T>(aObject: T): TJSONValue;
var
Marshal: TJSONMarshal;
begin
if Assigned(aObject) then
begin
  Marshal := TJSONMarshal.Create(TJSONConverter.Create);
  Result := Marshal.Marshal(aObject);
  Marshal.Free;
end
else
  Exit(TJSONNull.Create);
end;
```

[abrir imagem em janela]

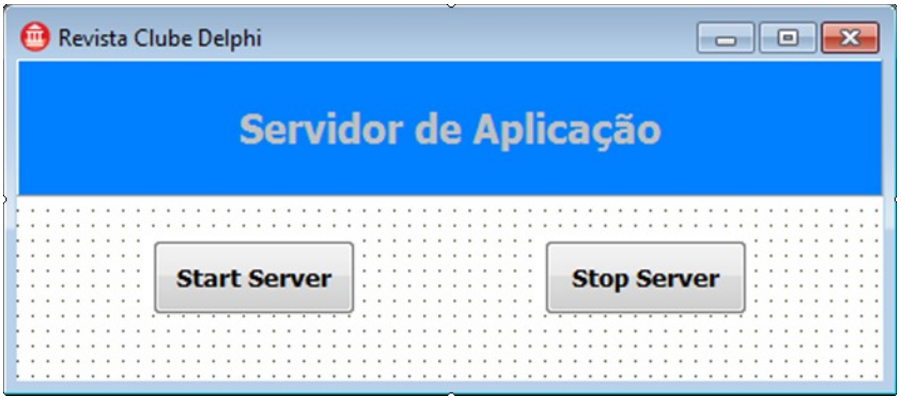

Figura 2. Interface do servidor

Para a conversão de um objeto JSON para um objeto tradicional, é usada a classe TJSONUnMarshal, que através do método unMarshal realiza todo o trabalho pesado. Ainda é possível notar um Type Casting do objeto retornado pela função para o objeto do tipo T, ou seja, o tipo genérico que será definido em tempo de execução. Já o processo para a conversão de um objeto tradicional para notação JSON faz uso da classe TJSONMarshal que invoca o método Marshal, onde da mesma forma que no processo anterior, faz todo o trabalho de conversão. Caso o objeto passado não esteja devidamente instanciado a função retorna um objeto JSON nulo evitando assim erros de Access Violation.

Métodos de classe nada mais são do que métodos que pertencem à classe e não ao objeto. Assim, não se faz necessária a criação do objeto para invocar o método, podendo invocá-lo diretamente da classe. Suponha que a classe TMyClass possua um método de classe

DoFoo, para invocá-lo basta fazer TMyClass.DoFoo, não sendo necessário instanciar o objeto. Para se definir um método como sendo da classe, basta precedê-lo da palavra class, por exemplo, class procedure e class function.

Com a classe Base implementada tudo fica muito simples, toda e qualquer classe que herde desta terá a capacidade de converter qualquer tipo em um objeto JSON e vice e versa.

Um detalhe importante a ser citado é que uma classe qualquer como, por exemplo, TCliente, pode ter seu método ObjectToJSON invocado para converter um objeto do tipo TFornecedor. O tipo não está restrito à classe que invoca o método e sim ao tipo genérico que é definido no momento de utilização do método.

Na Listagem 3 é possível ver a declaração da classe TContato utilizada para demonstrar a transferência de objetos entre a aplicação cliente e a aplicação servidora.

Classe de conexão

Como citado no início deste artigo uma dos padrões de desenvolvimento apresentado aqui é o DAO, com isso é fato que se fará necessário ter uma conexão com o banco de dados para que os objetos possam ser persistidos. Sendo assim, uma das maneiras que temos para controlar o acesso a dados em uma aplicação orientada a objetos é ter uma classe que gerencia esta conexão dentro da aplicação. Como geralmente em uma aplicação é utilizado apenas um banco de dados, nada melhor do que aplicar o padrão de projeto Singleton.

A classe de conexão utilizada para acesso ao banco de dados está descrita na Listagem 4 e se faz necessário ressaltar alguns pontos.

Listagem 3. Declaração da classe TContato

```
TContato = class(TBase)
public
  property OID: Integer;
  property Nome: String;
  property Email: String;
  property Telefone: String;
end;
```

Listagem 4. Classe de conexão com singleton

```
TDBConnection = class
  strict private
                    FConnection: TDBXConnection;
FCommand: TDBXCommand;
class var FInstance: TDBConnection;
constructor CreatePrivate;
public
constructor Create;
class function GetInstance: TDBConnection;
property Command: TDBXCommand read FCommand;
end;

constructor TDBConnection.Create;
begin
raise Exception.Create(Para obter uma instância de TDBConnection
  utilize TDBConnection.GetInatance);
end;

constructor TDBConnection.CreatePrivate;
begin
FConnection :=
```

```
  TDBXConnectionFactory.GetConnectionFactory.GetConnection(DBContato,
  root,root);
FCommand := FConnection.CreateCommand;
end;

class function TDBConnection.GetInstance: TDBConnection;
begin
if not Assigned(FInstance) then
FInstance := TDBConnection.CreatePrivate;
Result := FInstance;
end;
```

Nota

Como o nome indica, as constraints em Generics nada mais são do que restrições sobre os tipos reais que podem ser usados para parametrizar um tipo genérico. Ou seja, quando em tempo de execução definir é definido o tipo real com que se vai trabalhar, este tipo deve satisfazer a condição definida. Atualmente pode-se definir 4 tipos de restrições com suas variações:
Class - Define que o tipo real a ser utilizado deve obrigatoriamente ser um objeto descendente de TObject, exemplo de definição: myfunction(Obj: T). Caso queira especificar que este objeto deve herdar de uma classe mais específica basta definir: myfunction(Obj: T). Neste caso o objeto real passado deve ser obrigatoriamente descendente da classe TCustomEdit.
Interface - Define que o tipo real a ser utilizado deve obrigatoriamente descender de uma determinada interface, ou ser um objeto que implemente esta interface, por exemplo: myfunction(Obj: T). Neste caso o objeto passado deve ser do tipo desta interface, descendente desta interface ou implementar esta interface.
Record - Define que o tipo real a ser utilizado deve ser obrigatoriamente uma estrutura de valor e não de referência, podendo ser um valor ordinário, ponto flutuante ou um Record, por exemplo: myfunction(Obj: T).
Constructor - Define que o tipo real a ser utilizado deve possuir obrigatoriamente um construtor declarado que não possua parâmetros. Isso é muito comum quando internamente na função há a necessidade de se instanciar este tipo genérico. Assim há a certeza que pode-se invocar o método T.create sem que hajam erros em tempo de execução. Um exemplo de definição seria: myfunction(Obj: T).

O primeiro deles é o fato do construtor da classe gerar uma exceção. Isso é necessário, pois como o objeto implementa o padrão de projeto Singleton, logo o canal para criação deste deve ser apenas o método GetInstance. A função do método GetInstance é verificar, através da variável de classe FInstance, se o objeto já está instanciado e em caso afirmativo este objeto é retornado, caso contrário o primeiro objeto é criado e armazenado nesta mesma variável. Outro detalhe está no construtor privado, onde a conexão com o banco de dados é realizada utilizando as classes da versão 4 do DBX framework. Através da classe TDBXConnectionFactory foi solicitado ao framework uma nova conexão com o banco de dados cujo os parâmetros de conexão são:
- DBContato - Nome de uma conexão com o banco de dados MySql registrada no IDE do Delphi. Neste banco de dados há uma tabela chamada TBL_CONTATO cujo script pode ser visto na Listagem 5;
- root - Usuário utilizado para acessar a base de dados;

- root - Senha utilizada para acessar a base de dados.

O objeto de conexão com o banco de dados é então armazenado no Field FConnection, e em seguida um objeto do tipo TDBXCommand é criado. Este será responsável por executar os comandos SQL necessários no banco de dados.

Nota

Por fim o objeto DBXCommand é exposto através da propriedade Command, permitindo assim que comandos SQL possam ser enviados a classe de conexão e então executados no banco de dados.

Nota: Neste artigo assumiu-se o banco de dados MySql para desenvolver o assunto. Utilizando o DBExpress 4 é necessário lançar mão de duas units específicas, a saber: DBXCommon e DbxMySql. A primeira é padrão independente do banco de dados escolhido, já a segunda varia de acordo com o banco que se está trabalhando.

Atributos customizados

Um dos novos recursos disponíveis no Delphi é a possibilidade de utilizar atributos customizados.

Aqui são definidos dois atributos customizados, um para definir o nome da tabela em que o objeto deverá ser persistido e outro para definir quais das propriedades serão utilizadas como chave para que ao ser mapeadas as instruções de update e delete possamos saber exatamente qual objeto remover do banco de dados.

Listagem 5. Script de criação da tabela

```
CREATE TABLE dbcontato.tbl_contato (
  CONTATOID INTEGER NOT NULL AUTO_INCREMENT,
  NOME VARCHAR(50) NOT NULL,
  TELEFONE VARCHAR(20) NOT NULL,
  EMAIL VARCHAR(255) NOT NULL,
  PRIMARY KEY (CONTATOID)
);
```

Nota

A versão 4 do framework DBExpress foi totalmente reescrita em Delphi e está 100% orientada a objetos, com dezenas de classes e métodos que facilitaram e muito a vida de quem desenvolve utilizando o conceito de orientação a objetos. Além disso o framework é compatível com a plataforma Win32 e .NET. As principais classes deste framework são a TDBXConnection que tem por função realizar e gerenciar uma conexão com o banco de dados e a classe TDBXCommand que entre outras coisas tem a função de executar comandos SQL de toda natureza no banco de dados.

Os padrões de projetos são soluções descritas e catalogadas para problemas no desenvolvimento de softwares orientados a objetos. Eles descrevem soluções testadas e que podem ser reutilizadas sem um números de vezes. Os padrões de projetos mais conhecidos no meio são os 23 definidos pela GOF (gang of four) ou Gangue dos quatro. Entre eles temos o mais simples e eficaz padrão chamado Singleton. Ele é utilizado

quando se quer garantir que exista apenas uma instância de um determinado objeto em uma aplicação. Para isso ele utiliza um método de classe que tem por função testar e então decidir se deve instanciar um objeto ou retornar um já criado anteriormente. Este padrão é muito utilizado em classes de conexão com banco de dados, controle de usuários entre outros.

O Atributo TableName está exposto na Listagem 6 enquanto o atributo KeyField é descrito na Listagem 7 e ambos possuem apenas uma propriedade a saber, a propriedade Name, que é utilizada para que se possa acessar o conteúdo a ser utilizado no mapeamento objeto relacional.

Com os atributos já definidos podemos voltar a classe TContato e inserir os dois atributos que serão utilizados para realizar o mapeamento objeto relacional. Sobre a declaração da classe é adicionado o atributo TableName e nele é inserido o nome da tabela no banco de dados que conterá os dados da classe TContato.

Já sobre a propriedade OID é inserido o atributo KeyField e como valor do atributo informamos o nome do campo chave primária na tabela tbl_contato. Na Listagem 8 a seguir é possível ver como os atributos são declarados na classe TContato.

Classe DAO Genérica

Uma vez que as estruturas necessárias para o início do mapeamento objeto relacional já estão prontas é possível definir a classe que irá realizar este mapeamento (Listagem 9). Como foi mostrado anteriormente, o padrão DAO prega que para cada classe de negócio deve existir uma classe de mapeamento que seja responsável pela persistência.

Neste artigo está sendo empregado o recurso de Generics para que possa ser criada uma estrutura que atenda a todas as classes de uma forma geral. Há nesta estrutura um método para cada tipo de operação a ser realizada no banco de dados, além de dois métodos adicionais: GetTableName que tem a função de retornar o nome da tabela que será utilizada para persistir o objeto que esta sendo mapeado no momento, e GetAll que tem a função de retornar um objeto do tipo TDBXReader com a lista de contatos persistidos no banco de dados. Como é possível verificar todos os métodos são tipados com tipos genéricos, assim não temos por obrigação que os parâmetros sejam de um tipo específico como é comum na maioria dos frameworks de persistências. O TDBXReader é uma classe do DBExpress 4 que fornece um leitor unidirecional para um conjunto de linhas de base de dados. TDBXReader é retornado por TDBXCommand. ExecuteQuery. O método TDBXReader.Next é utilizado para acessar as linhas do result set. Os valores da linha podem ser acessados usando a propriedade Value que é um array. A propriedade Value é sobrecarregada e podemos utilizar tanto a posição da coluna quanto seu nome. O método GetTableName é utilizado pelos demais métodos, sua implementação é descrita na Listagem 10 e de uma forma simples é possível notar como o uso do RTTI proporciona extrair informações importantes de um objeto em tempo de execução.

Listagem 6. Classe do atributo TableName, utilizado na persistência

```
TableName = class(TCustomAttribute)
private
  FName: String;
public
  constructor Create(aName: String);
  property Name: String read FName write FName;
end;
```

```
constructor TableName.Create(aName: String);
begin
  FName := aName
end;
```

Listagem 7. Classe do atributo KeyField, utilizado na persistência

```
KeyField = class(TCustomAttribute)
private
  FName: String;
public
  constructor Create(aName: String);
  property Name: String read FName write FName;
end;

constructor KeyField.Create(aName: String);
begin
  FName := aName;
end;
```

Listagem 8. Atributos aplicados

```
[TableName(TBL_CONTATO)]
TContato = class(TBase)

[KeyField(CONTATOID)]
property OID: Integer read FOID write SetOID;
```

Listagem 9. Declaração da classe GenericDAO

```
TGenericDAO = class
private
  class function GetTableName<T: class>(Obj: T): String;
public
  class procedure Add<T: class>(Obj: T);
  class procedure Delete<T: class>(Obj: T);
  class procedure Update<T: class>(Obj: T);
  class function GetAll<T: class>(Obj: T): TDBXReader;
end;
```

Listagem 10. Implementação do método GetTableName

```
class function TGenericDAO.GetTableName<T>(Obj: T): String;
var
  Contexto: TRttiContext;
  TypObj: TRttiType;
  Atributo: TCustomAttribute;
  strTable: String;
begin
  Contexto := TRttiContext.Create;
  TypObj := Contexto.GetType(TObject(Obj).ClassInfo);
  for Atributo in TypObj.GetAttributes do
  begin
    if Atributo is TableName then
      Exit(TableName(Atributo).Name);
  end;
end;
```

Ao criar um objeto do tipo TRttiContext é possível lançar mão do método GetType que

efetivamente atribui à variável TypObj todas as informações providas pelo compilador. Feito isso, um loop é realizado nesta estrutura em todos os atributos que ela possui e caso seja encontrado um atributo que é do tipo TableName, através do método exit o método é abandonado retornando o conteúdo do atributo TableName que nada mais é do que o nome da tabela onde o objeto em questão será persistido.

O método Add (Listagem 11) tem a função de receber um objeto qualquer e utilizar RTTI para gerar uma instrução de insert no banco de dados. Novamente como no método anterior, um objeto do tipo TRttiContext é criado e em seguida todas as informações do objeto são extraídas para a variável TypObj. Três variáveis
do tipo string são utilizadas para a geração da instrução SQL. Para cada propriedade é verificada a existência ou não de atributos e havendo um ou mais atributos em uma propriedade é verificado se este atributo é do tipo KeyField. Caso seja um KeyField a propriedade não é mapeada, pois este campo é auto incremental no banco de dados.

Caso não seja chave primária, o nome da propriedade passa a fazer parte da instrução insert a ser gerada para que em seguida seja feita uma verificação para mapear o tipo da propriedade, que deve ser feita utilizando RTTI.

Caso não seja chave primária, o nome da propriedade passa a fazer parte da instrução insert a ser gerada para que em seguida seja feita uma verificação para mapear o tipo da propriedade, que deve ser feita utilizando RTTI.

Além disso, todos os valores obtidos devem ser convertidos para string para serem concatenados na montagem da instrução. Após esta etapa, são removidas as últimas vírgulas adicionadas no processo de montagem já mencionado anteriormente, por último a instrução é gerada e executada no banco de dados utilizando a classe Singleton já criada.

O método Update, exibido na Listagem 12, segue os mesmos procedimentos que já foram descritos anteriormente.

As diferenças aqui são duas: a primeira é que o comando SQL gerado neste momento é de Update e a segunda é que agora ao identificar que uma das propriedades possui o atributo KeyField ela não será mais desprezada ou “passada”, pelo contrário.

Ao identificar o atributo KeyField em uma propriedade o conteúdo do atributo é adicionado à variável que será usada na montagem da cláusula where assim como o conteúdo da mesma. As demais propriedades são mapeadas e para cada propriedade encontrada novamente é verificado seu tipo e concatenados nome e valor da propriedade às variáveis utilizadas para gerar o comando SQL de Update.

O método Delete é bem mais simples, novamente o objeto passado como parâmetro é mapeado via RTTI. Ao localizar a propriedade que possui o atributo KeyField a mesma é utilizada na cláusula where para que no fim a instrução delete seja executada no banco de dados. Como o método é bastante semelhante aos anteriores, omitimos a implementação aqui, mas você pode encontrar o código completo para download.

Listagem 11. Implementação do método Add

```
class procedure TGenericDAO.Add<T>(Obj: T);
var
        Contexto: TRttiContext;
        TypObj: TRttiType;
        Prop: TRttiProperty;
        strInsert, strFields, strValues: String;
        Atributo: TCustomAttribute;
        IsKey: Boolean;
begin
        strInsert := ;
        strFields := ;
        strValues := ;
```

```
        strInsert := INSERT INTO + GetTableName(Obj);
        Contexto := TRttiContext.Create;
        TypObj := Contexto.GetType(TObject(Obj).ClassInfo);
        for Prop in TypObj.GetProperties do
        begin
          IsKey := False;
          for Atributo in Prop.GetAttributes do
            if Atributo is KeyField then
              IsKey := True;
            if not IsKey then
            begin
              strFields := strFields + Prop.Name + ,;
              case Prop.GetValue(TObject(Obj)).Kind of
                tkWChar, tkLString, tkWString, tkString,
                tkChar, tkUString:
                  strValues := strValues +
            QuotedStr(Prop.GetValue(TObject(Obj)).AsString) + ,;
                tkInteger, tkInt64:
                  strValues := strValues +
            IntToStr(Prop.GetValue(TObject(Obj)).AsInteger) + ,;
                tkFloat:
                  strValues := strValues +
            FloatToStr(Prop.GetValue(TObject(Obj)).AsExtended) + ,;
                else
                 raise Exception.Create(Type not Supported);
              end;
            end;
        end;
        strFields := Copy(strFields, 1, Length(strFields) - 1);
        strValues := Copy(strValues, 1, Length(strValues) - 1);
        strInsert := strInsert + ( + strFields + ) VALUES ( + strValues + );
        TDBConnection.GetInstance.Command.Text := strInsert;
        TDBConnection.GetInstance.Command.ExecuteUpdate;
end;
```

Listagem 12. Implementação do método Update

```
class procedure TGenericDAO.Update<T>(Obj: T);
var
  Contexto: TRttiContext;
  TypObj: TRttiType;
  Prop: TRttiProperty;
  strUpdate, strSet, strKeyField, strKeyValue: String;
  Atributo: TCustomAttribute;
  isKey: Boolean;
begin
  strUpdate   := ;
  strSet      := ;
  strKeyField := ;
  strKeyValue := ;
  strUpdate := UPDATE + GetTableName(Obj);
  Contexto := TRttiContext.Create;
  TypObj := Contexto.GetType(TObject(Obj).ClassInfo);
  for Prop in TypObj.GetProperties do
  begin
    isKey := False;
    for Atributo in Prop.GetAttributes do
      if Atributo is KeyField then
```

```
      begin
        strKeyField := KeyField(Atributo).Name;
        strKeyValue := IntToStr(Prop.GetValue(TObject(Obj)).AsInteger);
        isKey := True;
      end;
      if isKey then Continue;
      strSet := strSet + Prop.Name + = ;
      case Prop.GetValue(TObject(Obj)).Kind of
        tkWChar, tkLString, tkWString, tkString,
        tkChar, tkUString:
          strSet := strSet +
          QuotedStr(Prop.GetValue(TObject(Obj)).AsString) + ,;
        tkInteger, tkInt64:
          strSet := strSet +
          IntToStr(Prop.GetValue(TObject(Obj)).AsInteger) + ,;
        tkFloat:
          strSet := strSet +
          FloatToStr(Prop.GetValue(TObject(Obj)).AsExtended) + ,;
        else
          raise Exception.Create(Type not Supported);
      end;
  end;
  strSet := Copy(strSet, 1, Length(strSet) - 1);
  strUpdate := strUpdate + SET +
    strSet + WHERE + strKeyField + = + strKeyValue;
  TDBConnection.GetInstance.Command.Text := strUpdate;
  TDBConnection.GetInstance.Command.ExecuteUpdate;
end;
```

Nota

O método GetValue que é invocado da classe TRttiProperty retorna um objeto do tipo TValue e, apesar do TValue possuir métodos AsString, AsExtended, AsInteger entre outros, estes métodos não realizam Casting dos tipos, ou seja, é necessário testar o tipo da propriedade para que possa utilizar o método correto para obter seu valor.

Para finalizar a classe GenericDAO falta apenas a implementação do método GetAll (Listagem 13). Novamente é lançada mão da classe de conexão com o banco de dados e através dela é executado o comando de seleção. O método ExecuteReader retorna um objeto do tipo TDBXReader, retornado pela função.

Publicando os métodos no servidor

Até o momento todas as estruturas apresentadas são totalmente desacopladas e independentes, isso mostra o quanto uma aplicação orientada a objetos facilita o reaproveitamento de código. A classe TContato não possui vínculo com a classe TGenericDAO e esta por sua vez está apta a mapear qualquer objeto que a ela for passado. Além disso, a classe de conexão que presta serviço para a classe TGenericDAO pode, sem nenhuma restrição, ser utilizada para outro fim neste ou em outro projeto qualquer.

E o mais importante, em nenhum momento até agora foi citada qualquer referência à aplicação multicamadas, ou seja, tudo visto até o momento pode ser utilizado em uma aplicação client-server sem qualquer problema. Mas para que toda a infraestrutura criada possa ser consumida em uma aplicação multicamadas é necessário criar uma interface de

comunicação com os clientes.

Esta interface na nova arquitetura DataSnap chama-se Server Methods, e é ilustrada na Listagem 14.

Sua estrutura possui os métodos necessários para permitir que as aplicações clientes possam adicionar um novo contato, remover um contato existente, atualizar um contato ou retornar a lista de contatos cadastrados.

Note que todos os métodos, exceto GetAllContacts, operam sobre um objeto, neste caso do tipo JSONValue. Isso porque a comunicação entre a aplicação cliente e servidora se faz através de notação JSON como citado anteriormente.

Daí a necessidade dos métodos de conversão criados na classe TBase.

Ao receber um objeto em notação JSON a aplicação servidora terá que transformá-lo em um objeto convencional e persisti-lo utilizando a classe TGenericDAO. Isso é mostrado na Listagem 15.

A implementação dos métodos é extremamente simples e limpa e toda a inteligência de persistência é encapsulada na classe TGenericDAO.

Aplicação Cliente

A aplicação cliente é bem simplória, a Figura 3 mostra a interface com alguns componentes.

Listagem 13. Implementação do método GetAll

```
class function TGenericDAO.GetAll<T>(Obj: T): TDBXReader;
begin
  TDBConnection.GetInstance.Command.Text := Select * from +
    GetTableName(Obj);
  Result := TDBConnection.GetInstance.Command.ExecuteQuery;
end;
```

Listagem 14. Declaração da classe TSMContato

```
{$METHODINFO ON}
  TSMContato = class(TComponent)
  public
    procedure DeleteContact(aObject: TJSONValue);
    procedure UpdateContact(aObject: TJSONValue);
    function GetAllContacts: TDBXReader;
    procedure AddContact(aObject: TJSONValue);
  end;
  {$METHODINFO OFF}
```

Listagem 15. Implementação dos métodos da classe TSMContato

```
procedure TSMContato.AddContact(aObject: TJSONValue);
var
  Contato: TContato;
begin
  Contato := TContato.JSONToObject<TContato>(aObject);
  TGenericDAO.Add<TContato>(Contato);
end;


procedure TSMContato.DeleteContact(aObject: TJSONValue);
var
  Contato: TContato;
begin
  Contato := TContato.JSONToObject<TContato>(aObject);
```

```
  TGenericDAO.Delete<TContato>(Contato);
end;

function TSMContato.GetAllContacts: TDBXReader;
begin
  Result := TGenericDAO.GetAll(TContato.Create);
end;

procedure TSMContato.UpdateContact(aObject: TJSONValue);
var
  Contato: TContato;
begin
  Contato := TContato.JSONToObject<TContato>(aObject);
  TGenericDAO.Update<TContato>(Contato);
end;
```

[abrir imagem em janela]

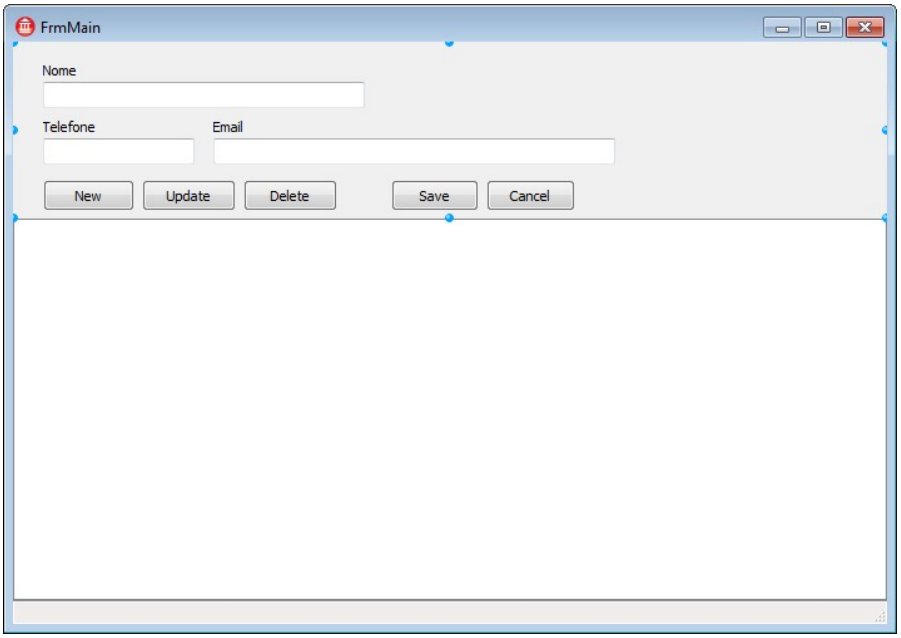

Figura 3. Interface da aplicação cliente

O componente TListView é ótimo para ser utilizado em aplicações que interagem com objetos pois, além de permitir que os dados de um objeto sejam listados ao usuário de forma simples e intuitiva, também dá a prerrogativa de associar o ponteiro do objeto para cada item listado. Isso significa que além de exibir os dados do objeto é possível manter seu ponteiro para posterior utilização.

Para conectar ao servidor de aplicação a aplicação cliente necessita de um componente TSqlConnection devidamente configurado utilizando o driver DataSnap do DBExpress. Com a conexão devidamente configurada é possível gerar a classe proxy que faz a interface de comunicação da aplicação cliente com o servidor de aplicação. Essa classe é gerada automaticamente ao selecionar a opção Generate DataSnap Client Classes.

A implementação dos botões New, Update, Delete e Cancel é feita na Listagem 16 onde basicamente, para os botões que se aplica, é mudado o conteúdo da variável FState declarada na seção private do formulário. É invocado o método ControlButtons, que controla os componentes visuais e o método clear que tem por função limpar o conteúdo dos edits. Ambos são exibidos na Listagem 17. Toda e qualquer conexão, ou melhor, interação com os métodos do servidor de aplicação, deve ser feita através da classe Proxy (Listagem 18 e Listagem 19).

No botão Save é feito o teste da variável FState e dependendo do seu conteúdo é decido qual método será invocado no servidor. Porém, como o que é trafegado entre o cliente e o servidor é sempre um objeto, é criado um objeto do tipo TContato, e, para ele são passados os dados contidos nos edits. Por último, um objeto do tipo TSMContatoClient, que nada mais é do que o proxy de comunicação com o servidor de aplicação, é criado e associado à variável Client. O método de classe ObjectToJSON é usado para converter o objeto TContato em um objeto JSON, para que então possa ser enviado ao servidor. É importante saber que nos casos de atualização ou exclusão de um objeto é de suma importância que se envie também o ID do objeto que se deseja deletar. A Figura 4 mostra a aplicação em execução.

Nota

Um proxy, em sua forma mais geral, é uma classe que funciona como uma interface para outra classe. Em padrões de projeto a função de uma classe proxy é prover um representante para outro objeto de modo a controlar o acesso a este. No caso do DataSnap pode-se dizer que há um proxy remoto, que é quando a classe proxy provê um representante local para um objeto em um espaço de endereçamento diferente, neste caso o servidor de aplicação.

Listagem 16. Implementação dos botões New, Update, Delete e Cancel

```
procedure TFrmMain.btnCancelClick(Sender: TObject);
begin
  FState := tsNone;
  Clear;
  ControlButtons;
end;

procedure TFrmMain.btnNewClick(Sender: TObject);
begin
  FState := tsNew;
  ControlButtons;
  Clear;
end;

procedure TFrmMain.BtnUpdateClick(Sender: TObject);
begin
  FState := tsUpdate;
  ControlButtons;
end;

procedure TFrmMain.BtnDeleteClick(Sender: TObject);
begin
  FState := tsDelete;
  ControlButtons;
end;
```

Listagem 17. Implementação dos métodos ControlButtons e Clear

```
procedure TFrmMain.Clear;
begin
  EdtNome.Clear;
  EdtEmail.Clear;
  EdtTelefone.Clear;
  If EdtNome.CanFocus then
```

```
    EdtNome.SetFocus;
end;

procedure TFrmMain.ControlButtons;
begin
  btnNew.Enabled := FState = tsNone;
  BtnUpdate.Enabled := (FState = tsNone) and
    (ListView1.Selected <> nil);
  btnSave.Enabled := (FState <> tsNone);
  btnDelete.Enabled := (FState = tsNone) and
    (ListView1.Selected <> nil);
  EdtNome.Enabled := (FState <> tsNone);
  EdtTelefone.Enabled := (FState <> tsNone);
  EdtEmail.Enabled := (FState <> tsNone);
end;
```

Listagem 18. Implementação do botão Save

```
procedure TFrmMain.btnSaveClick(Sender: TObject);
var
  Client: TSMContatoClient;
  Contato: TContato;
  M: TJSONMarshal;
  JSON: TJSONValue;
begin
  try
    Contato := TContato.Create;
    Contato.Nome := EdtNome.Text;
    Contato.Telefone := EdtTelefone.Text;
    Contato.Email := EdtEmail.Text;
    Client := TSMContatoClient.Create(DBServer.DBXConnection);
    case FState of
      tsNew:
      begin
        JSON := TContato.ObjectToJSON<TContato>(Contato);
        Client.AddContact(JSON);
      end;
      tsUpdate:
      begin
        Contato.OID := StrToInt(ListView1.Selected.Caption);
        JSON := TContato.ObjectToJSON<TContato>(Contato);
        Client.UpdateContact(JSON);
      end;
      tsDelete:
      begin
        Contato.OID := StrToInt(ListView1.Selected.Caption);
        JSON := TContato.ObjectToJSON<TContato>(Contato);
        Client.DeleteContact(JSON);
      end;
    end;
  finally
    FreeAndNil(Client);
    FreeAndNil(Contato);
    ShowReader;
    Clear;
    ControlButtons;
  end;
end;
```

Listagem 19. Implementação do ShowReader

```
procedure TFrmMain.ShowReader;
var
  I: Integer;
  LColumn: TListColumn;
  LRowItem: TListItem;
  aReader: TDBXReader;
  Client: TSMContatoClient;
begin
  ListView1.Clear;
  ListView1.Columns.Clear;
  ListView1.Update;
  Client := TSMContatoClient.Create(DBServer.DBXConnection);
  aReader := Client.GetAllContacts;
  for I := 0 to aReader.ColumnCount - 1 do
  begin
    LColumn := ListView1.Columns.Add;
    LColumn.Caption := aReader.ValueType[I].Name;
    LColumn.AutoSize := True;
  end;
  while aReader.Next do
  begin
    LRowItem := ListView1.Items.Add;
    for I := 0 to aReader.ColumnCount - 1 do
    begin
      if I = 0 then
        LRowItem.Caption := aReader.Value[I].AsString
      else        LRowItem.SubItems.Add(aReader.Value[I].AsString);
    end;
  end;
  ...
end;
```

[abrir imagem em janela]

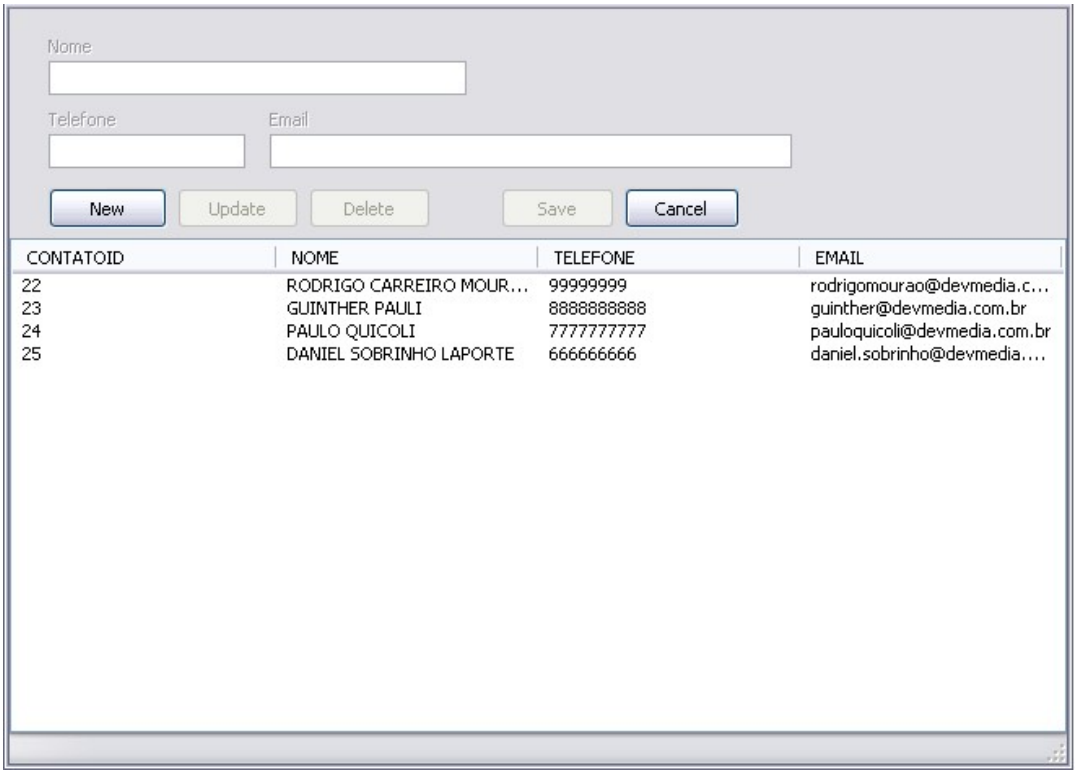

Figura 4. Aplicação em execução

Conclusão

O desafio está lançado, o conceito e a boa prática foram mostrados neste artigo. Uma experiência interessante seria criar novas classes de negócio e submetê-las à classe TGenericDAO para ver como a persistência irá se comportar. Obvio que há outros assuntos a serem abordados e tratados pela classe DAO, como o caso de associações entre classes do tipo composição e agregação, além do fato da otimização dos comandos SQL.

O que foi mostrado aqui é apenas um pontapé inicial, mostrando o que é e como é possível aplicar alguns padrões de desenvolvimento multicamadas na nova arquitetura DataSnap. Outras tecnologias podem ser agregadas como, por exemplo, criar um servidor totalmente RESTFull permitindo assim que não somente aplicações Win32 possam consumir nossos métodos no servidor de aplicação mas também qualquer aplicação Web que tenha suporte a Java Script por exemplo.