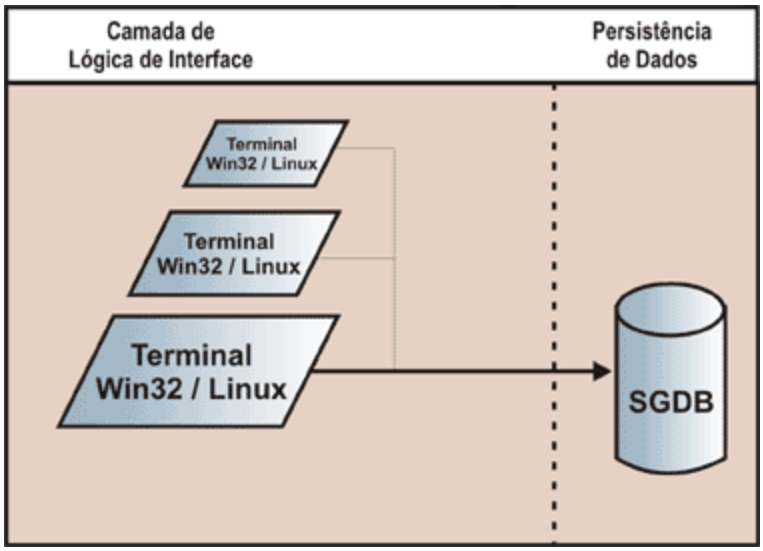
Modelo Client-Server



Desvantagens do Modelo Client server

Até aqui tivemos um pequeno resumo do modelo C/S. Agora iremos conduzir esta avaliação tentando mostrar alguns pontos negativos em virtude desta escolha.

Em primeiro lugar, é bom observar se ao invés de uma deficiência em si, os questionamentos que se seguem ocorrem mais em virtude de: a própria evolução dos negócios no tocante ao uso da WEB; distribuição dos aplicativos para uma rede cada vez mais heterogênea; independência de fornecedores de tecnologia (principalmente SGDBs).

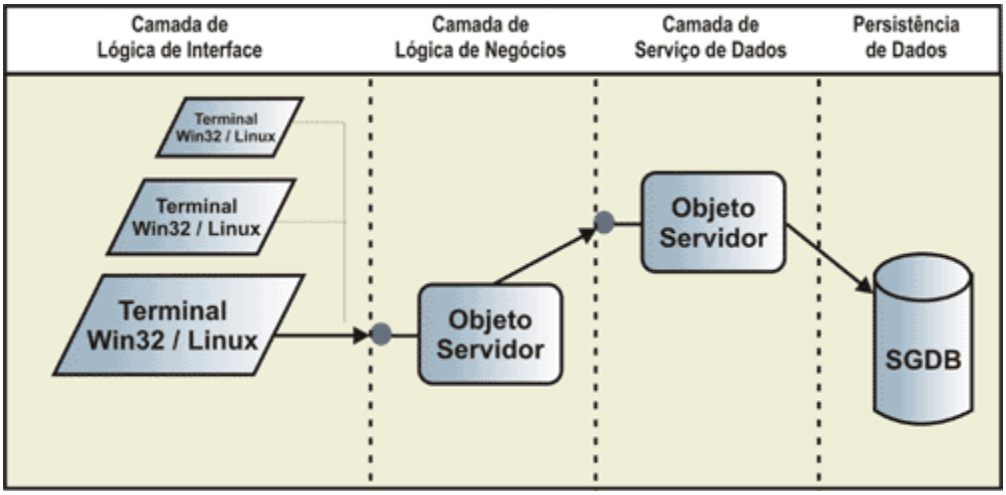
No tocante ao lado cliente, que resumimos em três partes (aplicativo, Middleware e cliente DB) há uma dupla inconveniência: a primeira diz respeito à necessidade de se administrar a camada Middleware em todas as estações cliente, trazendo um impacto quando da troca ou manutenção destes pacotes que acarretaria em ter de percorrer cada ponto desta rede implementando tais procedimentos. A segunda inconveniência é igualmente indesejável, por apresentar o mesmo problema que a primeira quanto a necessidade de se atualizar o mecanismo *Cliente DB* em virtude de uma atualização do Servidor de Banco de Dados para uma nova versão.

Agora focando o lado Servidor do modelo, ressaltamos a impossibilidade da troca do Servidor de Banco de Dados, uma vez que na adoção do modelo C/S foi implementado o conceito de lógica de negócio e manipulação de dados na forma de *triggers* e *procedures*, tornando impraticável, por exemplo, a troca de um SGDB Oracle por um SGDB Interbase que possibilitaria uma economia no custo com a licença de Banco de Dados. Um outro exemplo seria a constatação de que um SGDB como o MSSQL não atende mais às exigências do negócio quanto a performance.

Que atitude devemos tomar? Migrar para Oracle reescrevendo cada procedimento na linguagem procedural deste novo SGDB ou simplesmente não migrar e não atender às novas exigências do negócio?

O modelo C/S divide a solução em duas camadas distintas, distinguindo também quais as tecnologias deste modelo residem em qual parte, esgotando imediatamente as possibilidades. É aí que reside o problema, com novas tecnologias emergindo, escalabilidade cada vez mais solicitada, busca por independência de fornecedores, levam modelo C/S à exaustão, acenando para uma solução com mais camadas envolvidas no processo.

O Modelo Baseado em Camadas



Já que fomos ao centro do problema igualmente iremos apresentar a solução. A primeira atitude é dividir a camada cliente em pelo menos dois módulos (Figura 2). Ficando na primeira camada somente a interface de aplicação, excluindo-se do seu código todos os procedimentos que se relacionam diretamente com a camada *Middleware* (acesso a dados*)*. Na segunda camada, em forma de DLL ou EXE, residirá o código retirado da aplicação responsável pelo acesso a dados, o mecanismo *Middleware* e *software client* do DB. Da mesma forma iremos tratar o servidor de Banco de Dados, mantendo nesta camada (3ª camada) somente os dados e suas estruturas de índices, transformando suas *triggers* e *stored procedures* em métodos dentro das classes nas tais DLLs ou EXEs da segunda camada.

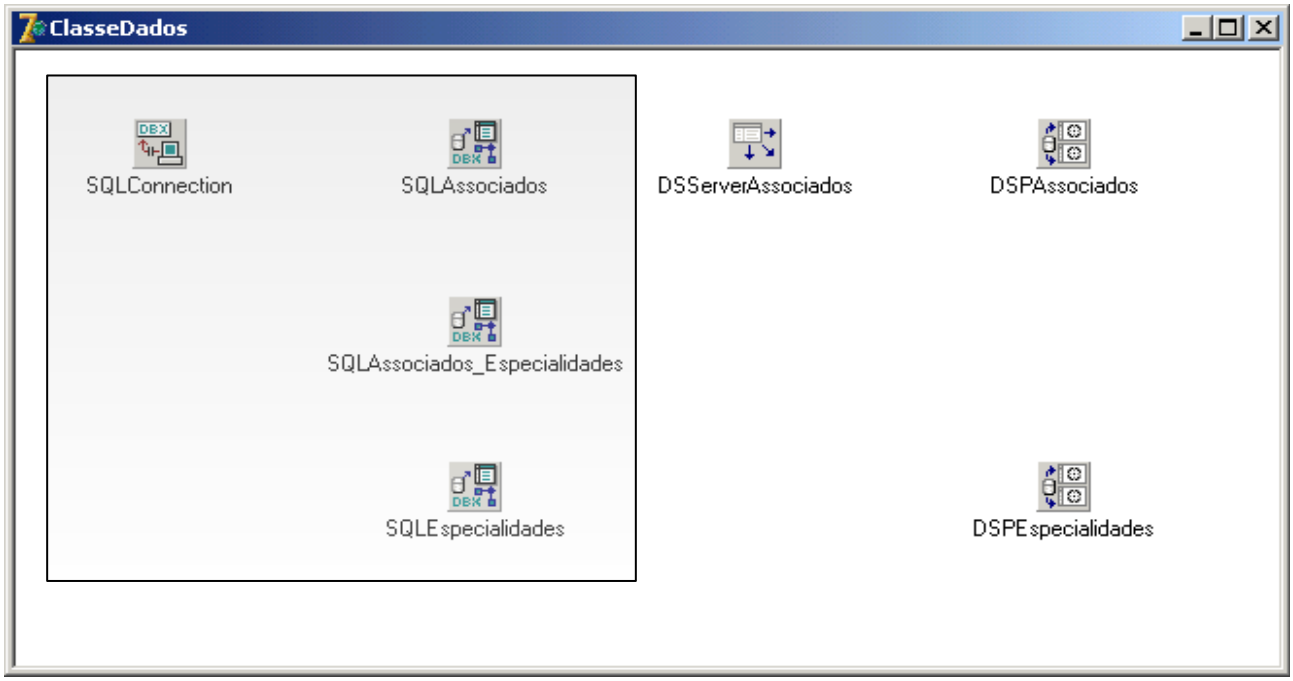
Isto é o que podemos chamar desenvolvimento *n-tier*, ou seja, múltiplas camadas, ou ainda, Objeto Distribuído (OD).

Analisando este modelo podemos observar claramente os benefícios em relação ao modelo C/S citando como primeiro ganho o fato dele apresentar uma solução para a estação cliente livre de tecnologias proprietárias, tornando esta camada independente e mais leve quanto os requisitos computacionais. Retirar da camada de Banco de Dados a lógica de programação, em uma primeira análise não traz outro ganho que não a diminuição do processamento neste servidor (que por si só justificaria). Mas observando mais profundamente, temos o ganho de independência do fornecedor desta tecnologia, já que não haverá nenhum impedimento em migrarmos para o *software* de outro fornecedor.

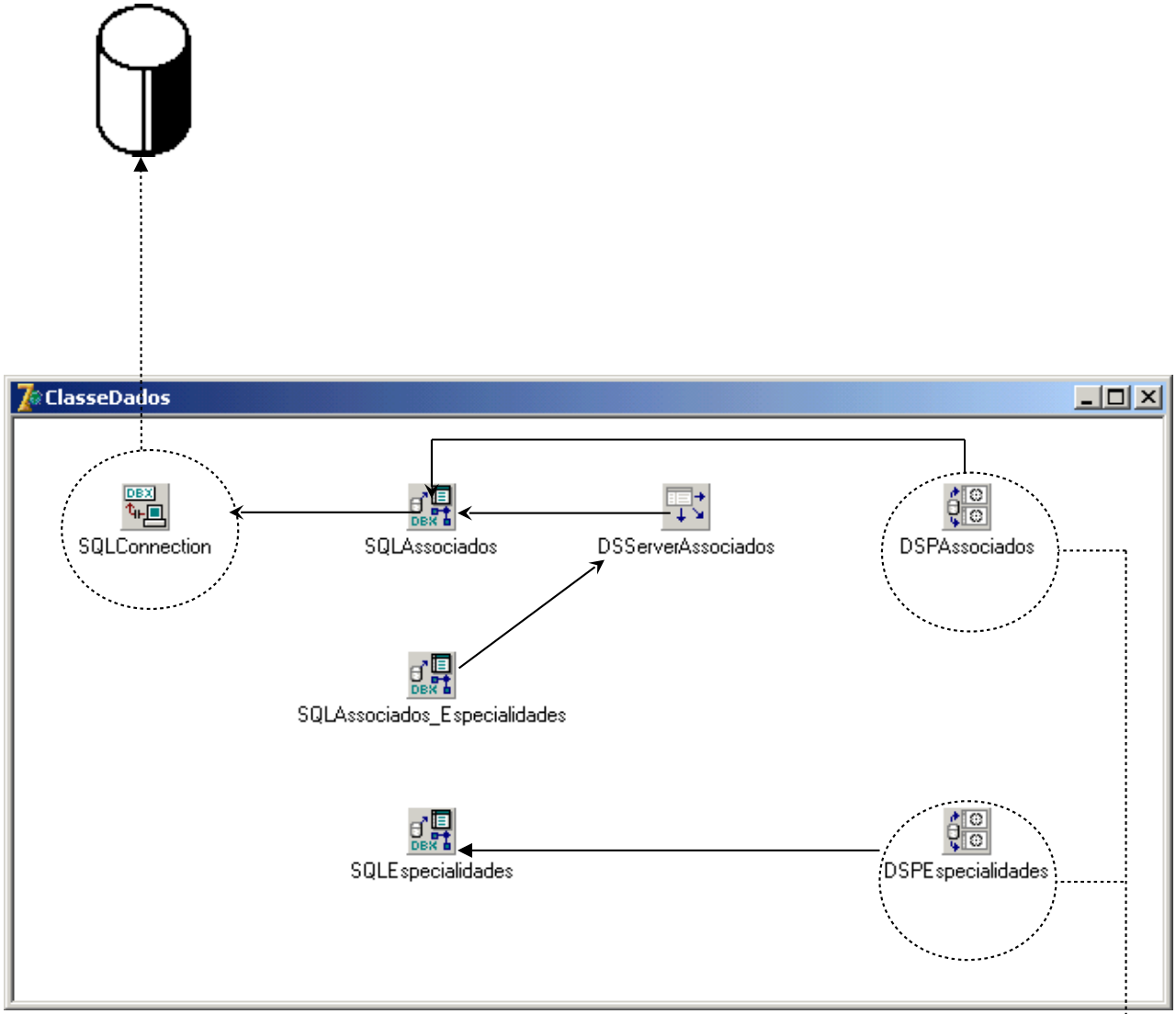
Repare que, na prática, a segunda camada atende plenamente aos propósitos de manter métodos substitutos aos anteriores procedimentos de banco de dados, no tocante a facilidade de manutenção das regras de negócio, bem como anula a necessidade de se manter tecnologias de acesso a dados em todos os pontos da rede.

A Indepedência de Middleware

Observando a figura 17, sobretudo o retângulo sombreado, é fácil entendermos o porque da tão aclamada independência de tecnologia de acesso a dados. Imaginem se por motivo não definido, necessitemos trocar dbExpress por ADO ou outra middleware de mercado. Bastaria simplesmente trocar o componentes de conexão ao banco de doados e os DataSets pelos respectivos componentes desta nova tecnologia e nada mais.
É explícito a independência da camada composta pelo componente DataSetProvider no lado servidor, e dos controles ClientDataSet implementados nas aplicações clientes, sejam clientes da rede ou servidor de web. Tal separação garante o investimento intelectual aplicado aos aplicativos de modo geral.



O Diagrama Completo



Implementando Regras de Domínio e Regras de Negócio

Uma das principais motivações para adoção do modelo de objetos distribuídos, é a possibilidade de podermos implementar código de validação de dados de forma centralizada, permitindo manutenção nestas regras independente das implementações clientes que estão submetidas as mesmas.
Também é correto, neste ponto do sistema, implementarmos códigos oriundos de triggers de banco de dados que eventualmente proviam estas funcionalidades.
Como exemplo, aplicaremos um regra que compara o valor estabelecido para o campo Data, do DataSet SQLAssociados considerando-o como válido caso tal valor seja inferior a data atual em cinco dias no máximo. De outra forma, deverá ser gerado uma exceção com uma mensagem ao cliente que demandou a execução.

Regra de Negócio 01:

Campo Data >= Data Atual – 5 dias

```
1.  procedure TClasseDados.DSPAssociadosBeforeUpdateRecord(Sender: TObject;
2.    SourceDS: TDataSet; DeltaDS: TcustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
3.  begin
4.     if UpdateKind <> ukDelete then
5.       begin
6.         if (DeltaDS.FieldByName('Data').Value < (Date - 5)) then
7.           begin
8.             Raise Exception.Create('Data de cadastramento não válida.');
9.             Applied:=True;
10.                end;
11.           end;
12.      end;
```

Comentários:
O evento onBeforeUpdateRecord ocorre imediatamente antes do DataSetProvider implementar as atualizações junto ao banco de dados. Desta forma, podemos interceptar as operações demandadas pela aplicação cliente por intermédio do objeto DeltaDS, que contem os registros incluídos, alterados ou excluídos no processo cliente.
Este objeto é uma implementação de um DataSet, de onde podemos acessar o registro, para ler seus valores (campos) ou até mesmo manipular estes valores.
Além deste DataSet, o evento também expõe um objeto denominado UpdateKind, que retorna a operação que determinou a ocorrência do procedimento, ou seja, ukDelete, ukModify e ukInsert, respectivamente operação de exclusão, alteração ou inclusão.
Diante do exposto, podemos implementar qualquer regra, adaptando o valor original da operação, ou gerando uma exceção que invalide a operação.

Linha 05 Identifica qual operação está disparando o procedimento.
Linha 07 Aplica a regra de negócio.
Linha 09 Caso a regra seja ferida, levanta uma excption.
Linha 10 Manipula o valor da variável, forçando o cancelamento do processo.

Uma outra aplicabilidade, talvez mais convincente para esta camada, seria a implementação de uma regra de domínio. Um bom exemplo seria criar uma validação para o campo Nome, do DataSet SQLAssociados, que não aceitasse conteúdo com menos de cinco caracteres. Para melhor ilustrar, imaginemos que o objetivo seja não aceitar nomes de associados com menos de cinco letras.

Regra de Negócio 01:

Campo Nome tem que conter mais de quatro caracteres.

```
1.  if UpdateKind <> ukDelete    then
2.       begin
3.         if (Length(DeltaDS.FieldByName('Nome').Value) < 5) then
4.           begin
5.          Raise Exception.Create('Nome deve conter mais de quatro letras.');
6.               Applied:=True;
7.           end;
8.       end;
```

Comentários:
Da mesma forma que o exemplo anterior, na ocorrência de uma inclusão ou alteração de registro, é disparado o procedimento que aplica a regra com todos os cuidados necessários (se válido prossegue se não gera erro).
Na prática, os dois exemplos, diferem na natureza. Enquanto o primeiro implementa uma regra de negócio (está mais próximo de uma condição que muda com o interesse do negócio), o segundo exemplo implementa uma regra mais rígida, que tende a não mudar com as possíveis novas definições do negócio.
Não há nada de errado em implementar essas regras nesta camada (camada de persistência), mas, um ótimo modelo, seria estender a operação a uma camada de validação exclusiva as regras de negócio.
Isto ocorre, porque teoricamente, as regras de negócio não estão obrigatoriamente relacionadas as operações de manipulação e validação de dados. E também, resguardamos da possibilidade de termos operações que não necessariamente passem por essa camada de dados, mas que necessitem de implementar as validações impostas pelas regras de negócio.
Um outro ponto positivo da adoção desta técnica, é que se já conseguimos total separação da lógica de interface com a lógica de negócio, prosseguimos na busca deste modelo, também estabelecendo a implementação das regras de negócio em separado.

Criando Um Servidor DataSnap - REST

Passo 01
Ponha em execução o ambiente IDE do Delphi XE5.

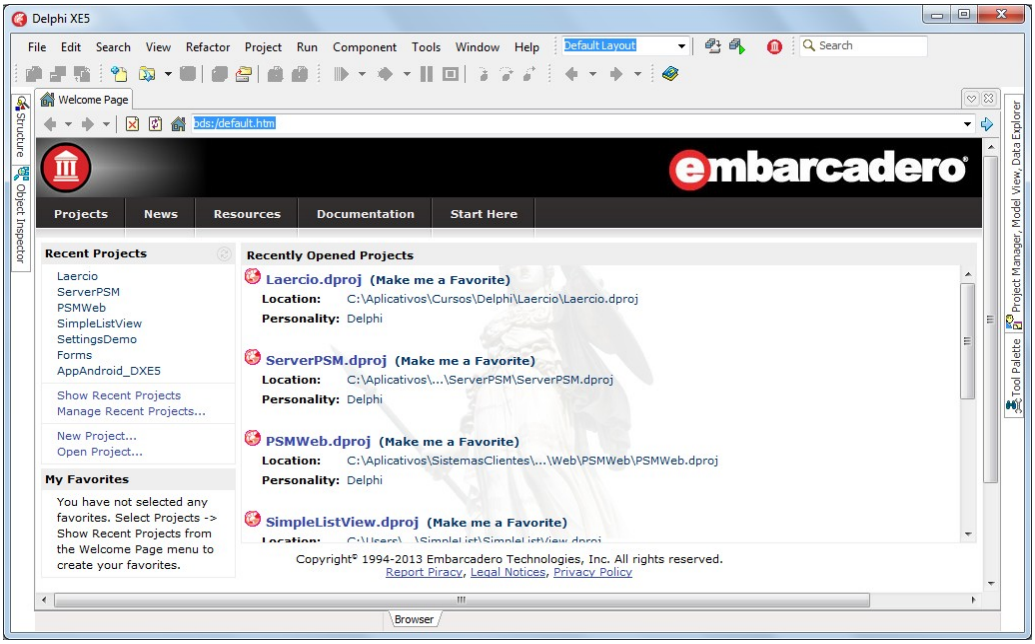


Figura 01

Passo 02

Crie um projeto DataSnap REST conforme ilustrado pela Figura 02, acessando a opção de Menu File | New | Other... | DataSnap Server. Clique em Ok.

Repare que temos outras opções de servidores, mas, devido ao fato de desejarmos atender (ser consumido) clientes não alinhados com tecnologia Embarcadero (Delphi, C++Bilder entre outras), utilizaremos o padrão REST como forma de integração, principalmente na troca de dados.

Como propõe este trabalho, exemplificaremos com uma aplicação Android 4.3, consumindo serviços do servidor em questão.

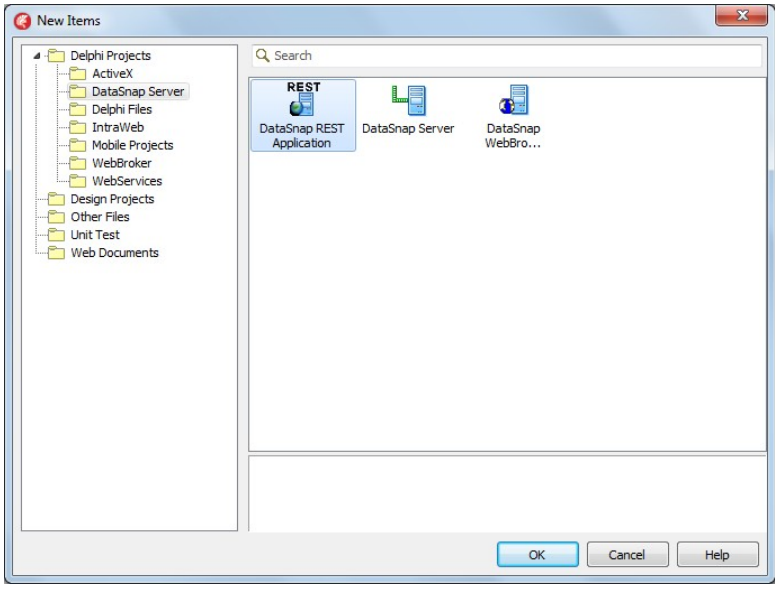


Figura 02

Passo 03

Nesta etapa, selecione a opção Stand-alone VCL application. Esta opção permite a implantação do serviço de forma transparente, sem a necessidade de um gerenciador de serviço do ambiente operacional. O próprio projeto, quando em execução, cria as condições.

A segunda opção, equivale a primeiro, exceto que não apresenta uma interface (Form) para gerenciamento visual.

Quanto a opção ISAPI, exclusiva do ambiente Windows, necessita que seja configurada no Gerenciador do Internet Information Servervice. Existe justificativa para assim proceder, quanto a questões de performance e segurança, mas isso é discutível e contornável.

Clique no botão Next e sigamos em frente.

Figura 03

Passo 04

Aqui procedemos testes para saber se neste Host, a porta 9876 está liberada. Neste caso, em uma máquina de desenvolvimento, dificilmente será verdade, uma vez que a porta 8080, quase um padrão, já estaria em uso. Observe a Figura 03, que ilustra este passo.

Clique no botão Test Port, procedendo assim o teste para a porta definida em HTTP Port. Caso não possa ser usada (ver Figura 04), clique no botão Find Open Port para que o Wizard DataSnap REST Application possa obter uma porta liberada para uso.

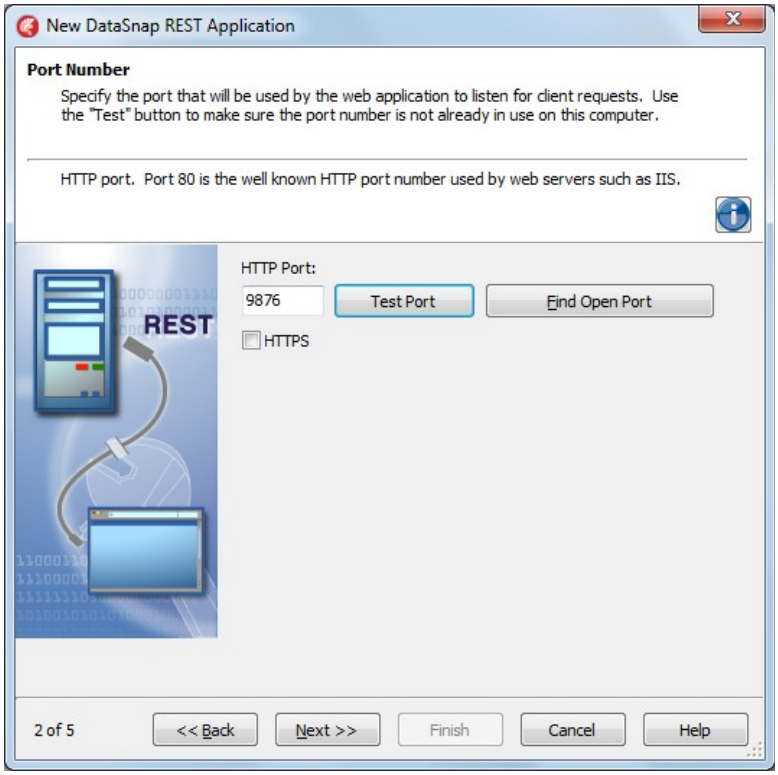


Figura 04

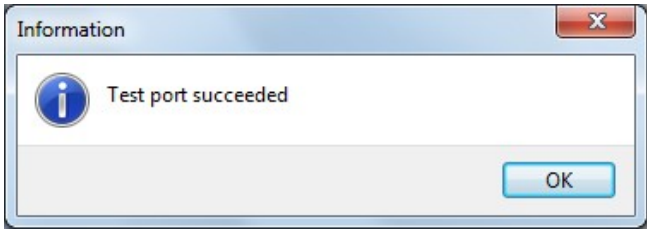


Figura 05

Nota:

A abordagem sobre qual porta utilizar para o serviço, vai além de uma simples escolha aleatório, sendo assunto a ser definido pela equipe responsável pela segurança do ambiente operacional.

Passo 05

A imagem da Figura 06 trata de uma questão não menos importante, pois configura nosso Servidor de Aplicação DataSnap a fornecer conectores para aplicativos Mobile. Em nosso caso, conforme abordado na parte final deste projeto, iremos gera em forma de Proxy, as classes Java (entre outras) necessárias para a interoperabilidade entre o projeto um aplicativo Android e nosso serviço.

Assinale a opção Mobile Connectors e clique em Next para prosseguir.

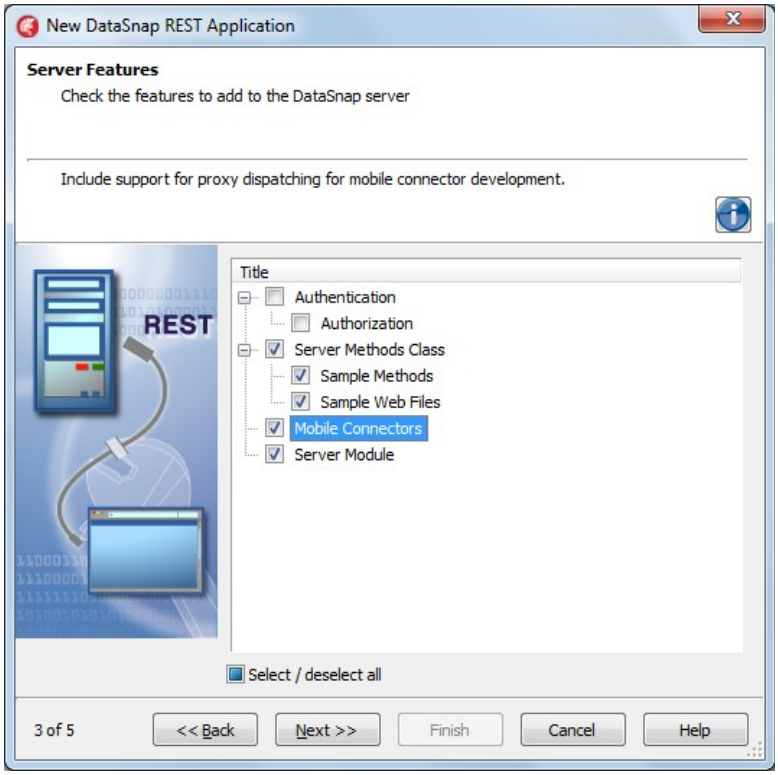


Figura 06

Passo 06
"TDSServerModule expõe conjuntos de dados e métodos do servidor para aplicativos cliente." Esta é a definição oficial, encontrada em:

http://docwiki.embarcadero.com/RADStudio/XE3/en/Configuring_TDSServerModule

Em termos práticos, pensando em desenvolvimento baseado em camadas, seja camada lógica ou física, utilizamos classes tipadas como TDSServerModule para acomodar rotinas CRUD, métodos de negócio e regras de negócio. Isso aponta para as boas práticas de OOP, dando aderência aos padrões de projetos inerentes, sobretudo, cenário perfeito para representar a camada Model do padrão MVC.

Nota:
Este assunto é polêmico, pois os puristas de OOP, sempre se detendo as formas clássicas de implementação de classes na forma Java e C#, não consideram desta forma. Estão invariavelmente se prendendo a codificação na "unha" dos método e atributos, ignorando o aspecto RAD (Rapid Application Developer), mas isso deixo para os especialistas.
Vida que segue, clique no botão Next conforme Figura 07 para prosseguir.

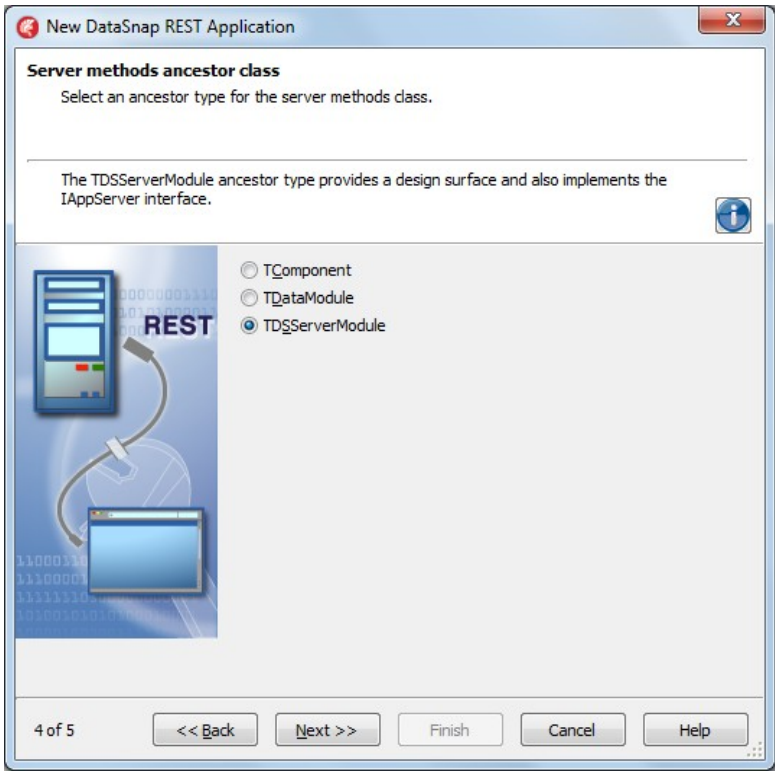


Figura 07

Passo 07

Agora, finalizando, se oriente pelas imagens das Figura 08 para escolha de uma pasta em seu sistema para acomodar os fontes do projeto.

Após a escolha do diretório para armazenamento do projeto, clique no botão Finish e pronto. Dever cumprido. Conforme prometido, seis passos e o mínimo de configuração e temos um Servidor DataSnap.

Agora, conforme veremos na sequência, basta implementar métodos de acesso a dados e utilizá-los em projetos cliente REST como Php, clientes baseado em Delphi como Form Win (descktop), Servidor Web Intraweb, Mobile Android/iPhone e por ai vai.

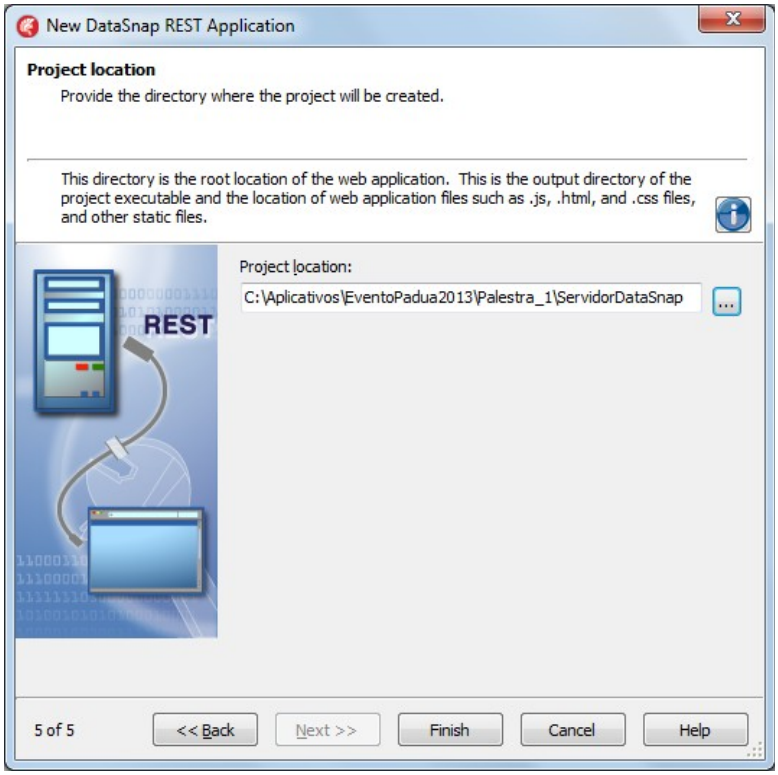


Figura 08

Listagens e Interfaces Visuais dos Módulos do Projeto

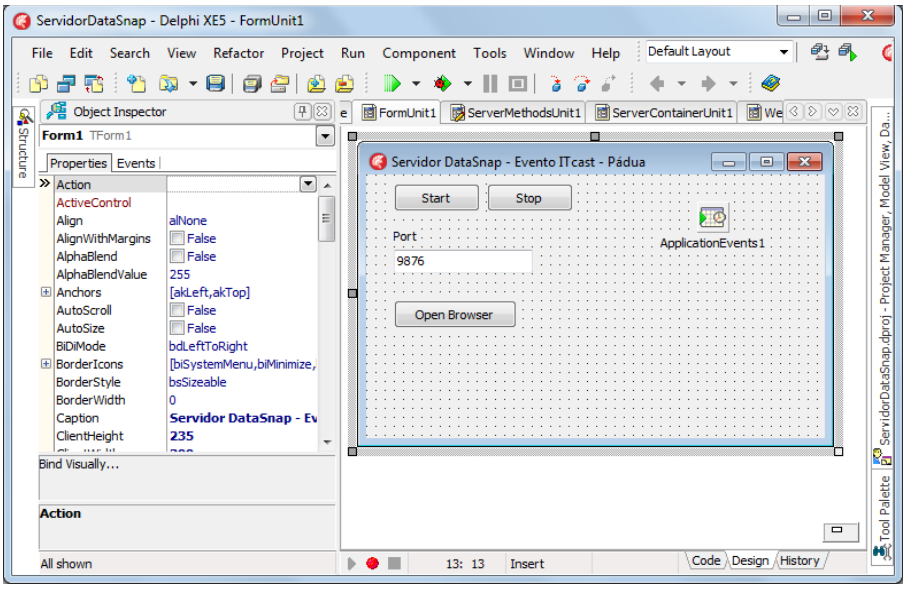


Figura 09 - Interface de Grenciamento

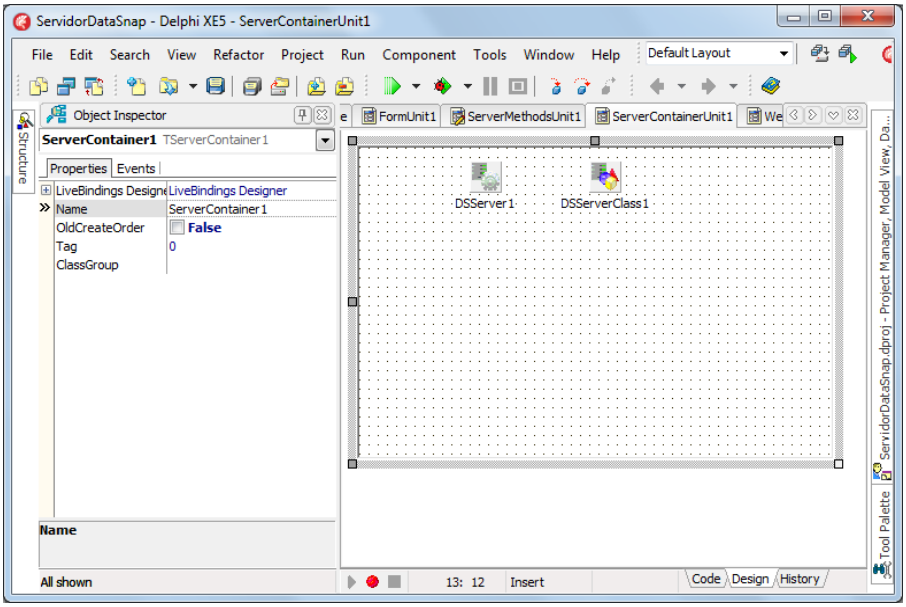


Figura 10

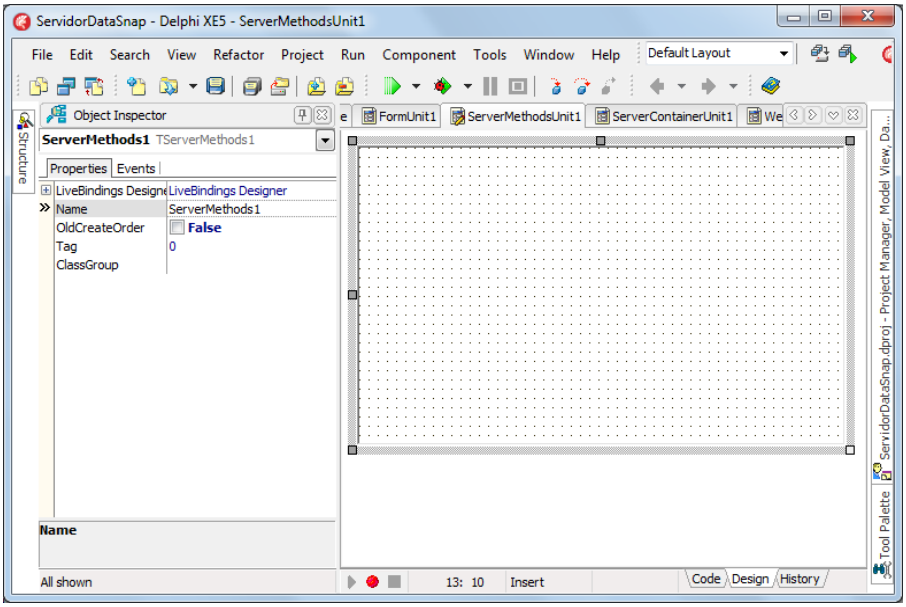


Figura 11 - DataModule ServerMethods

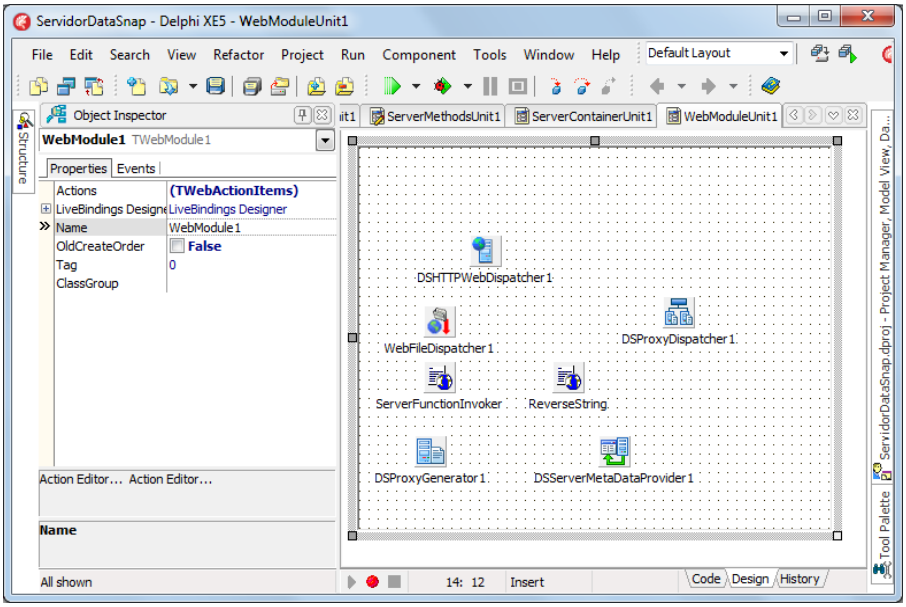


Figura 12 - WebDataModule

Primeiro Teste do Projeto DataSnap REST

– Execute o projeto observando o resultado na Figura 13.

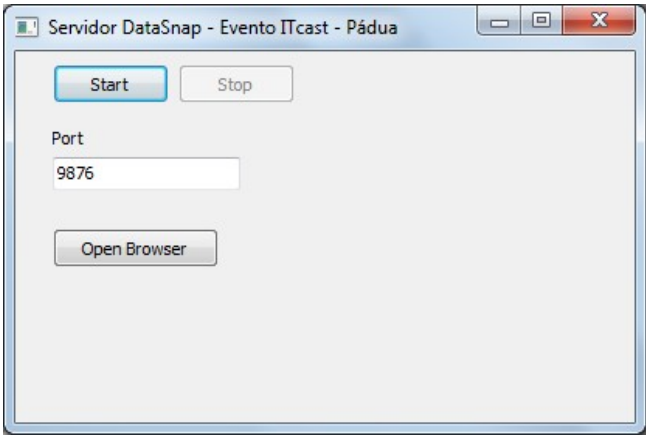


Figura 13

- Naturalmente, o serviço de segurança do SO solicitará autorização (Figura 14) para implantar o serviço no Host.

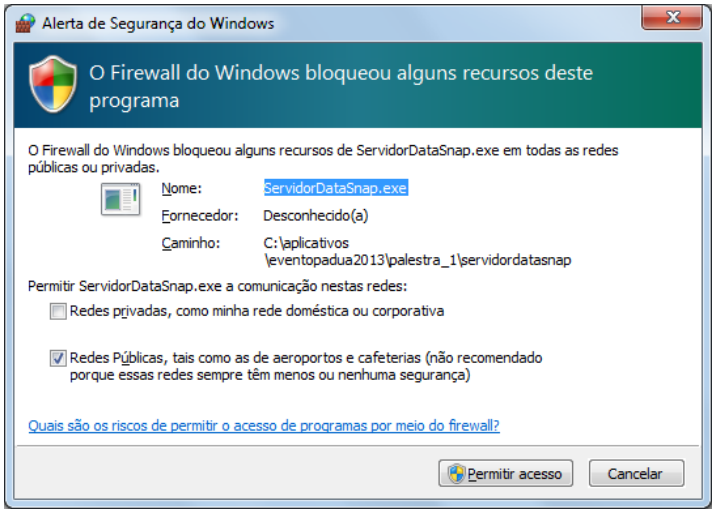


Figura 14

- Ainda com foco na Figura 03, clique no botão Open Browser, acompanhando os teste dos métodos de exemplo do projeto. Esses passos podem ser observado pelas imagens representadas pelas Figuras 15 e 16

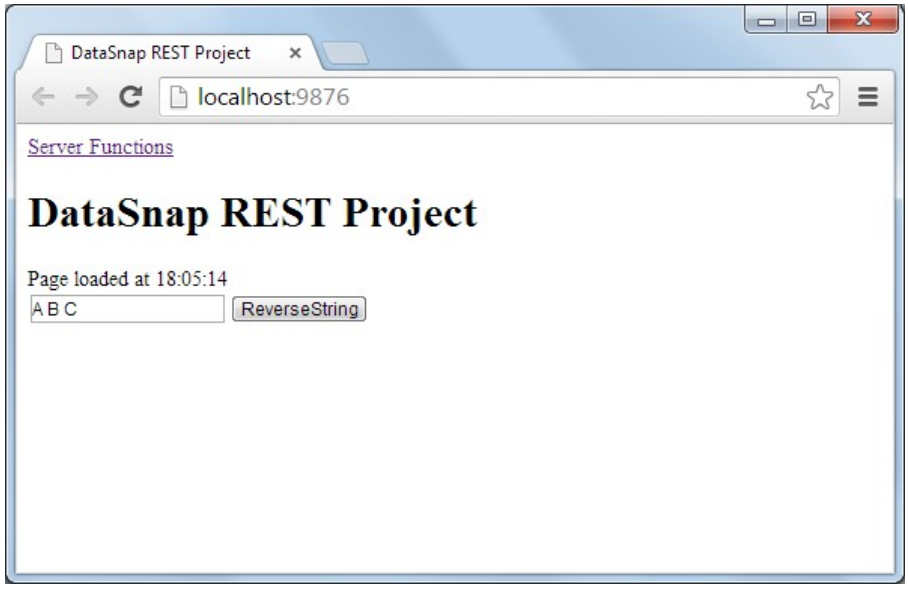


Figura 15

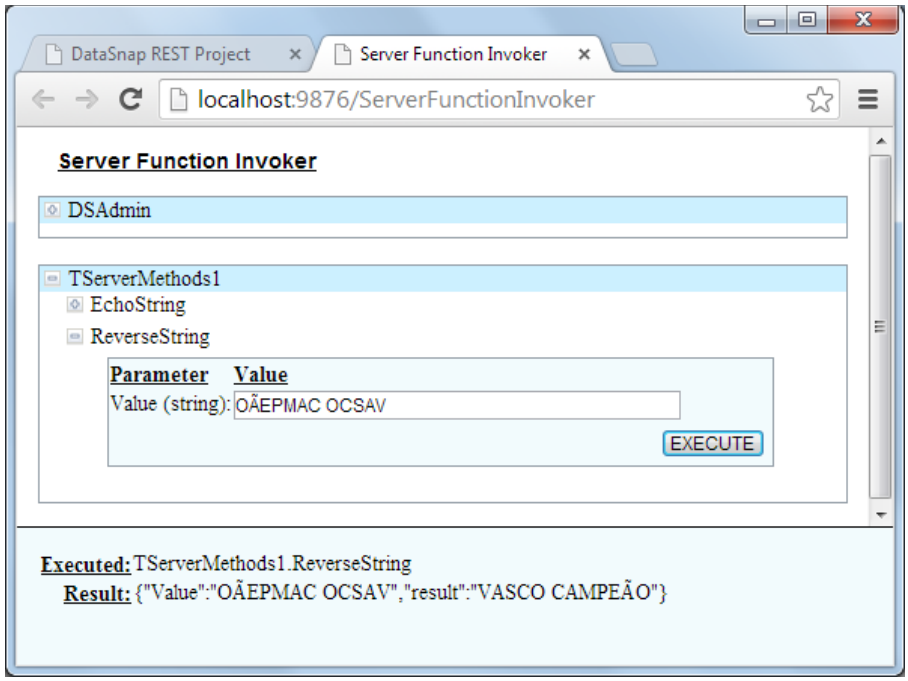


Figura 16

Escolhendo Plataforma para Compilação

- Um aspecto importante na tecnologia DataSnap, é que ela é projetada para atender clientes heterogêneos quanto as plataformas de desenvolvimento (linguagens e frameworks), e ainda, em ambiente Windows fornece suporte a 32 e 64 bits.

- Clicando com o botão direito do mouse sobre o nó Target Plataforms, na caixa de dialogo Project Manager, escolha a opção Add Plataform... e teremos a caixa de dialogo representada pela Figura 17 permitindo a escolha de uma plataforma específica.

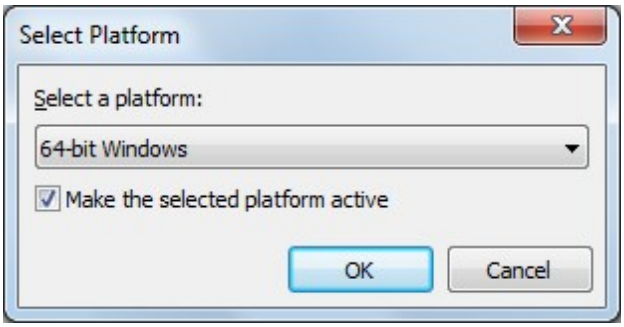


Figura 17

Acrescentando DataSets e Métodos de Negócio a Classe TDSServerModule

- Conforme demonstrado na Figura 01 desta etapa, implementamos dois DataSets, com respectivos DataSetProvider visando permitir que clientes DataSnap possam ter acesso aos dados expostos, e por intermédio dos mesmos, utilizando um componente ClientDataSet, criar o ambiente CRUD necessário.

- Agora atente para o fato, de que clientes heterogêneos como aplicativos Android, entre outros, não suportam a tecnologia DataSnap, logo, não implementam uma classe ClientDataSet. Então, voltando os olhos para o cerne da questão, lembramos que esta apresentação foca justamente o consumo de serviços por um cliente Android. Isso justifica a existência de dois controles ClientDataSet, pois os métodos de negócio os utilizarão para implementar a lógica de acesso a dados.

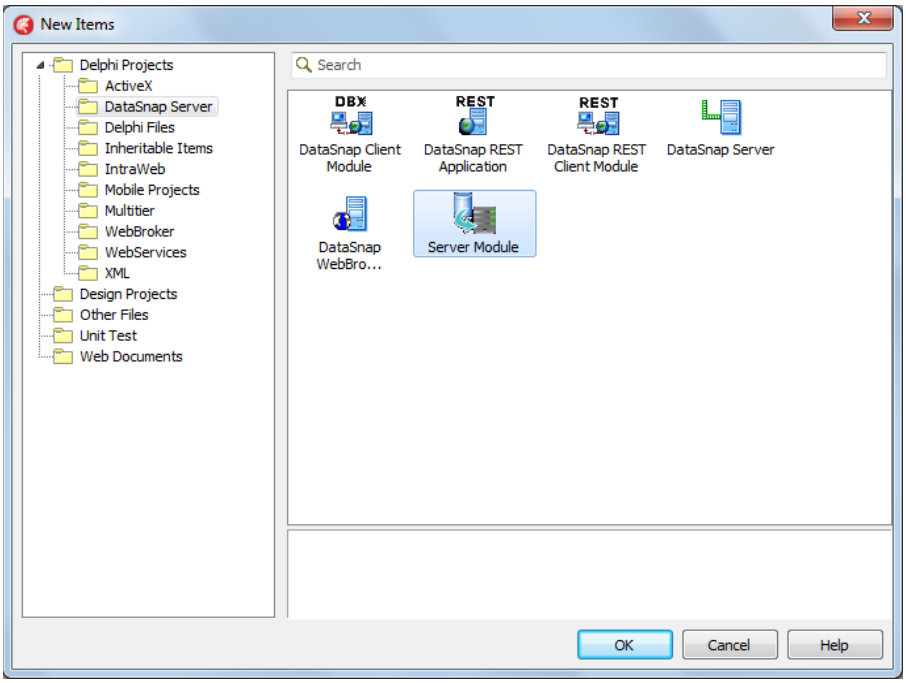


Figura 18

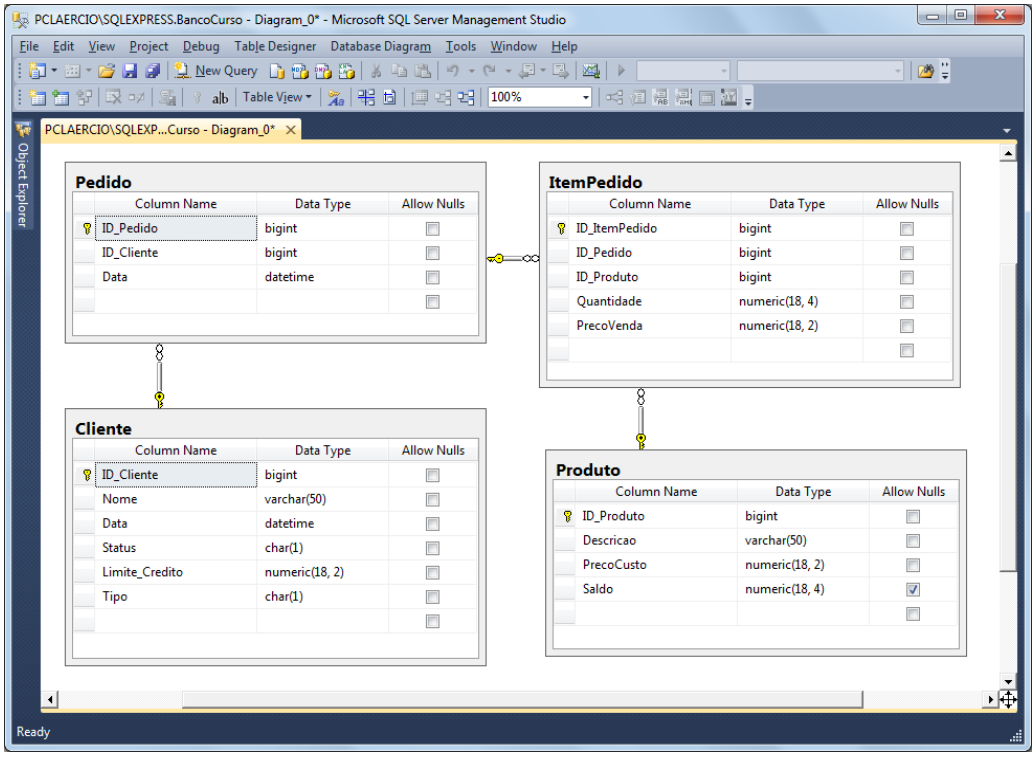


Figura 19

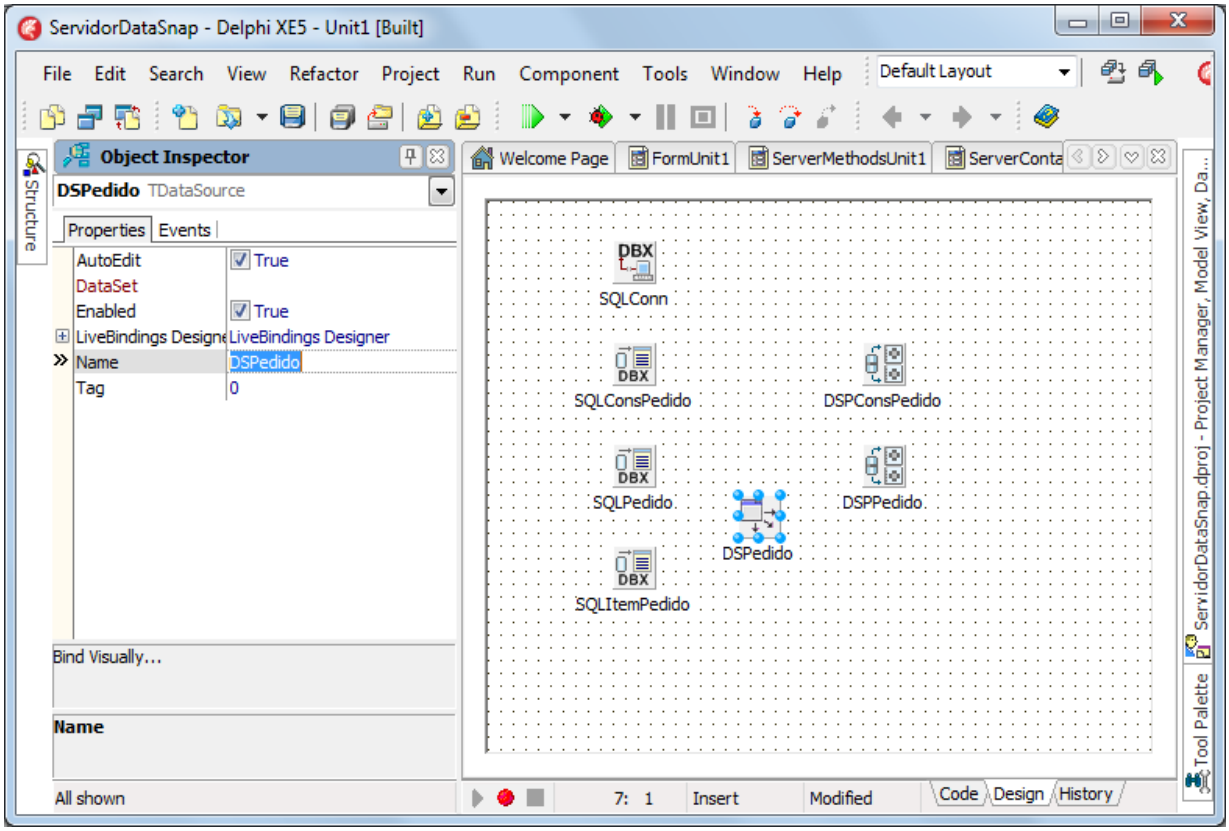


Figura 20

Objeto: SQLConnection	
Propriedade	**Valor**
Name	SQLConn
Driver	MSSQL
Driver - Database	BancoCurso
Driver - HostName	localhost\SQLExpress (server / nome da instância)
Driver - Password	123
Driver - UserName	sa
LoginPrompt	false
Tabela 01	

Objeto: SQLDataSet	
Propriedade	**Valor**
Name	SQLConsPedido
SQLConnection	SQLConn
CommandText	select Pedido.*, Cliente.Nome from Pedido, Cliente where Pedido.ID_Cliente = Cliente.ID_Cliente AND Cliente.Nome Like :Nome
Params - DataType	:Nome = ftString
Tabela 02	

Objeto: DataSetProvider	
Propriedade	**Valor**
Name	DSPConsPedido
DataSet	SQLConsPedido
Tabela 03	

Objeto: SQLDataSet	
Propriedade	**Valor**
Name	SQLPedido
SQLConnection	SQLConn
CommandText	select Pedido.*, Cliente.Nome from Pedido, Cliente where Pedido.ID_Cliente = Cliente.ID_Cliente AND Pedido.ID_Pedido = :ID_Pedido
Params - DataType	:ID_Pedido = ftLargeint
Tabela 04	

Objeto: DataSetProvider	
Propriedade	**Valor**
Name	DSPPedido
DataSet	SQLPedido
Tabela 05	

Objeto: DataSource	
Propriedade	**Valor**
Name	DSPedido
DataSet	SQLPedido
Tabela 06	

Objeto: SQLDataSet	
Propriedade	**Valor**
Name	SQLItemPedido
SQLConnection	SQLConn
DataSource	DSPedido
CommandText	select ItemPedido.*, Produto.Descricao from ItemPedido, Produto where itemPedido.ID_Produto = Produto.ID_Produto AND ItemPedido.ID_Pedido = :ID_Pedido
Params - DataType	:ID_Pedido = ftLargeint
Tabela 07	

Objeto: SQLDataSet	
Propriedade	**Valor**
Name	SQLItemPedido
SQLConnection	SQLConn
DataSource	DSPedido
CommandText	select ItemPedido.*, Produto.Descricao from ItemPedido, Produto where itemPedido.ID_Produto = Produto.ID_Produto AND ItemPedido.ID_Pedido = :ID_Pedido
Params - DataType	:ID_Pedido = ftLargeint

Tabela 07

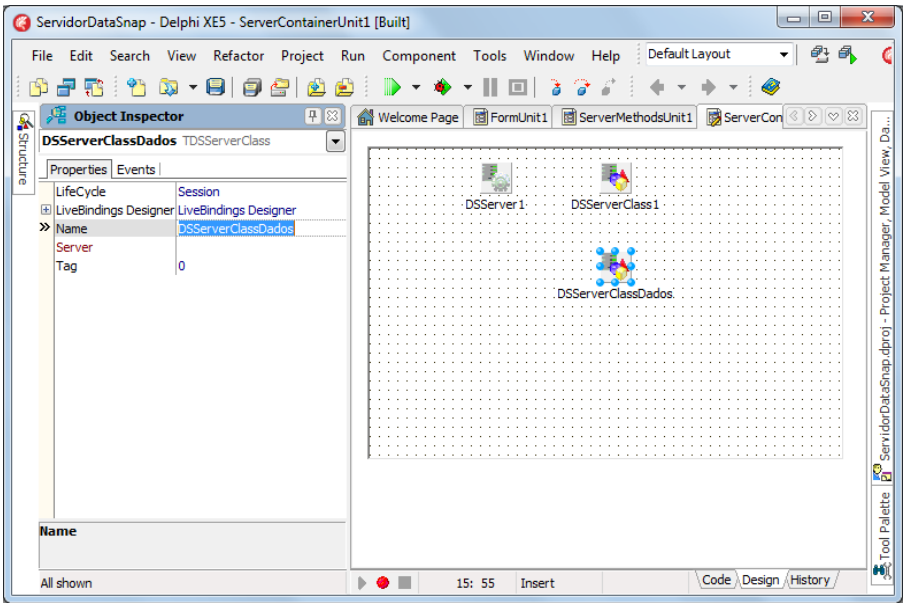


Figura 21

```
procedure TServerContainer1.DSServerClassDadosGetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
   PersistentClass := U_DSServerModuleDados.TDSServerModuleDados;
end;
```

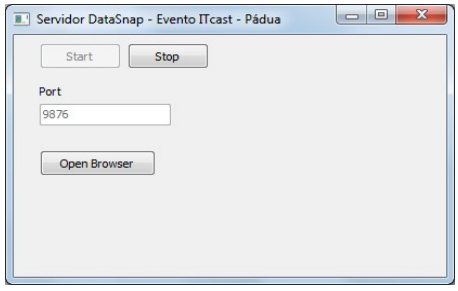


Figura 22